N° d'ordre : 3102

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ DE BORDEAUX I

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

Par **Aurélien ESNARD**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Analyse, conception et réalisation d'un environnement pour le pilotage et la visualisation en ligne de simulations numériques parallèles

---

Soutenue le : 12 décembre 2005

Après avis des rapporteurs :

Frédéric Desprez	Directeur de Recherche
Bruno Raffin	Chargé de Recherche

Devant la commission d'examen composée de :

Olivier Coulaud	Directeur de Recherche	Directeur de Thèse
Frédéric Desprez	Directeur de Recherche	Examinateur
Raymond Namyst ...	Professeur	Président & Rapporteur du Jury
Jean-Philippe Nominé	Ingénieur CEA	Examinateur
Bruno Raffin	Chargé de Recherche ..	Examinateur
Jean Roman	Professeur	Directeur de Thèse

– 2005 –

Remerciements

Une thèse, c'est avant tout un morceau de vie dont il faut un peu raconter l'histoire pour en remercier les protagonistes. Les deux premières années de ma thèse au LaBRI furent tout d'abord un long tunnel obscur passé dans la salle 121 (une sorte d'entrepôt de bienvenu réservé aux thésards) jusqu'à ma promotion dans la salle CVT (un autre entrepôt pour thésards plus âgés). A cette époque, notre réflexion sur la problématique du pilotage en était encore qu'à son balbutiement et nous n'avions aucun recul sur la tâche qui nous incombait, puisque cette thématique était tout à fait nouvelle dans l'équipe ScAlApplix. Il a fallu donc étudier les travaux existants, classifier d'innombrables articles plus ou moins pertinents, se tromper, chercher à comprendre et petit à petit nous faire une idée plus précise. C'est alors que nous avons décidé de nous lancer dans la réalisation d'un premier prototype de la plate-forme EPSN sans savoir où cela nous mènerait encore. C'est grâce aux nombreuses discussions répétées avec Olivier Coulaud que nous avons pu progressé lentement mais sûrement. C'est donc tout naturellement que je tiens à le remercier ici de m'avoir accompagné et conseillé durant tout ce temps avec cette bonne humeur qui le caractérise au quotidien. Je voulais également remercier Jean Roman, qui a co-encadré cette thèse avec Olivier et dont les conseils toujours pertinents ont su orienter mes travaux. Par ailleurs, il faut rendre un hommage tout particulier à son légendaire « crayon rouge » qui a déjà fait trembler toute une génération de thésards et qui n'a pas son pareil pour vous faire réviser votre copie. Merci donc à tous les deux de m'avoir donné la chance de faire cette thèse et de m'avoir soutenu dans les moments difficiles. Je tiens également à remercier Pascal Guitton avec qui nous avons démarré cette thèse et avec qui nous continuons de collaborer activement autour de la réalité virtuelle ; je pense que cet échange initié entre nos deux équipes continuera d'être fructueux.

Au cours de l'année 2002, le projet EPSN était accepté au titre d'une ACI GRID, ce qui permis de recruter un ingénieur. Cet ingénieur ne m'était pas tout à fait inconnu puisqu'il s'agissait de mon ancien binôme à l'ENSEIRB, Michael Dussère. J'avais alors pas mal baroudé avec Mick : plan clochard au Portugal en 1999, plan trappeur en Finlande en 2001, puis en Crête et en Corse, sans compter nos nombreuses expéditions dans les Pyrénées (été comme hivers). Tout cela tombait « pile-poil » comme dirait l'autre. Désormais, tout allait se dérouler beaucoup plus vite. Et nous nous lançâmes avec Mick dans le développement de la nouvelle version d'EPSN sans compter ni nos heures, ni les pizzas du soir, ni les bouteilles de bière. J'ai récemment compter qu'il y avait en tout 150 000 lignes de codes programmées par nos petits doigts. Aujourd'hui que Mick est parti vers de nouveaux horizons, c'est un peu une branche du CVS qui est morte. Je tiens donc à le remercier chaleureusement au titre de notre amitié et lui souhaite « tout le bonheur du monde » comme dit la chanson.

Et puis il y a « les copains du labo » qu'il faut remercier simplement d'avoir été là, et plus particulièrement tous les Scalapplixiens présents et passés : Pascal Hénon (pascalou), Pierre Ramet (pierrot), François Pellegrini (pellos), Olivier Beaumont, Bertrand Cirou (beber), Dimitri Lecas, Guillaume Anciaux (mon nouveau co-bureau), Pierre Fortin (« ... tape des mains »), Abdou Guermouche, Nicolas Richart (mon jeune padaone, qui débute sa thèse sur EPSN) et tous les autres que je n'aime pas moins mais qui sont trop nombreux pour que je les cite tous. Une raison de la bonne entente qui règne au sein de l'équipe tient en partie à la retraite qui nous a conduit ensemble au couvent du Haut-Carré, une annexe de l'université où nous avons déménagé lors de travaux au LaBRI. Ce fût l'âge d'or des barbecues, des pastisades, des parties en réseau de Unreal Tournament et du Spong, une variante « no-limit » du Ping-Pong qui emprunte des règles au Volley et aux sports de combat. En 2005, la fédération officielle de Spong comptait une dizaine de membres avec ses champions : Pierre et Pascal auto-proclamés « les winners for (n)ever », Mick « le boulet » (rien à voir avec des boulets de canon), Olivier Beaumont « le boucher » dont les victimes devaient porter le titre peu enviable de « steack-hâché » ainsi que moi-même plus communément appelé « la fouine » pour la pression tactique qu'il exerçait sur ses adversaires en les poussant irrémédiablement à la faute. Je me souviens aussi de nos discussions « à refaire le monde » avec Pierre, Pascal et Mick au Crokopain ou au Yorkshire. Tout l'enjeu était de savoir s'il faut mieux programmer archaïquement avec des macros comme Pierre le soutient ou s'il ne faut pas mieux utiliser les concepts modernes de la programmation orientée composant. Bref des histoires à n'en plus finir, dont une démonstration mémorable du sac-à-dos en temps linéaire !

Pour terminer, je souhaite remercier ma famille : mes grand-mères qui sont venues assister à ma soutenance,

mes petites sœurs jumelles (cakaouette et milou) et mes parents qui m'ont toujours soutenu et encouragé dans ma volonté de faire une thèse. Ils ont toute ma reconnaissance et ma gratitude. Enfin ces remerciements ne sauraient être complet sans dire un « grand merci » à Ysolde. Elle a été présente dès le début de ma thèse pour me porter bonheur et m'accompagner de son sourire, quand elle-même préparait le concours de médecine. Elle m'a supporté durant tout ce temps et je sais combien la fin de ma thèse a été dure pour tous les deux. Je mesure à présent la chance que j'ai eue de partager ma vie avec cet ange, quand nous vivions tous deux en compagnie de notre petit chat Chamboultou. Dans plusieurs années, je repenserai sûrement avec un brin de nostalgie à cette époque heureuse et révolue, que furent les dernières années de ma vie étudiante. Sans savoir de quoi sera fait l'avenir, je sais qu'une page est désormais tournée. Il ne tient qu'à nous d'écrire la suite...

Analyse, conception et réalisation d'un environnement pour le pilotage et la visualisation en ligne de simulations numériques parallèles

Résumé : Le domaine de la simulation interactive ou *computational steering* a pour but d'améliorer le processus de simulation numérique (modélisation, calcul, analyse) en le rendant plus interactif. Dans cette approche, le scientifique n'attend plus passivement les résultats de la simulation ; il peut visualiser « en ligne » l'évolution des données calculées et peut interagir à tout moment en modifiant certains paramètres à la volée ou plus généralement en pilotant le déroulement des calculs. Un tel outil peut s'avérer très utile pour la compréhension des phénomènes physiques modélisés et la détection d'erreurs dans le cas de simulations longues. L'objectif de cette thèse est de concevoir et de développer une plate-forme logicielle, appelée EPSN (Environnement pour le Pilotage des Simulations Numériques), permettant de piloter une application numérique parallèle en s'appuyant sur des outils de visualisation eux-mêmes parallèles. En d'autres termes, il s'agit de mettre au service des scientifiques les capacités de la visualisation parallèle et plus largement de la réalité virtuelle (environnement immersif, murs d'images), une étape aujourd'hui cruciale pour la conception et l'exploitation de simulations numériques complexes en vraie grandeur. La mise en œuvre d'un couplage efficace entre simulation et visualisation soulève deux problèmes majeurs, que nous étudions dans cette thèse et pour lesquels nous souhaitons apporter une contribution : le problème de la coordination efficace des opérations de pilotages en parallèle et le problème de la redistribution pour des données complexes (grilles structurées, ensembles de particules, maillages non structurés).

Mots-clés : pilotage des simulations, simulation numérique, parallélisme, visualisation scientifique, couplage de codes, redistribution de données.

Design of a Software Environment for the Online Visualization and the Computational Steering of Parallel Numerical Simulations

Abstract: The computational steering is an effort to make the typical simulation work-flow (modeling, computing, analyzing) more efficient, by providing online visualization and interactive steering over the on-going computational processes. The online visualization appears very useful to monitor and to detect possible errors in long-running applications, and the interactive steering allows the researcher to alter simulation parameters on-the-fly and to immediately receive feedback on their effects. Thus, the scientist gains an additional insight in the simulation regarding to the cause-and-effect relationship. The purpose of this thesis is to design and to develop a software environment, called EPSN (Environment for the Steering of Parallel Numerical Simulations) that enables to steer parallel simulations with visualization systems that can be parallel as well. In other words, we want to provide an environment that can benefit from immersive virtual reality technology (e.g. tiled display wall) and that might help scientists to better grasp the complexity of real-life simulations. Such a coupling between parallel numerical simulations and parallel visualization systems raises two crucial issues we investigate in this thesis: the problem of parallel coordination of steering operations and the problem of data redistribution of complex objects such as structured grids, particle set and unstructured meshes.

Keywords: computational steering, numerical simulation, parallelism, scientific visualization, code coupling, data redistribution.

Table des matières

Chapitre 1

Introduction

La mise à disposition de moyens de calculs hautes performances conventionnels et l'émergence des grilles de calcul font que les scientifiques disposent d'une puissance de calcul de plus en plus grande permettant de simuler des phénomènes physiques complexes avec toujours plus de réalisme (modèles plus fins, résolution plus élevée). Ces simulations d'une nouvelle génération utilisent généralement plusieurs modèles spécifiques couplés et distribués en plusieurs codes parallèles utilisant souvent un réseau rapide. Les applications simulant le système climatique de la Terre sont souvent citées à titre d'exemple : le projet CCSM (Community Climate System Model [6]) utilise un ensemble de quatre codes parallèles pour modéliser le comportement de la Terre (CLM), des océans (POP), de la glace (CICE) et de l'atmosphère (CAM). Ainsi, les applications du calcul scientifique deviennent de plus en plus complexes nécessitant de plus en plus de mémoire et impliquant des temps de calcul souvent très longs. L'analyse des résultats par la visualisation soulève pour ce type de calcul de réelles difficultés tant le volume de données à traiter est important, ce qui rend également nécessaire l'utilisation du parallélisme dans ce domaine. Par ailleurs, la démocratisation récente des environnements graphiques à base de grappes de PCs, de murs d'image ou de grands écrans, ainsi que les avancées récentes dans le domaine de la visualisation parallèle [35, 147, 148] font qu'il est désormais possible de traiter efficacement de grands ensembles de données.

Vers le pilotage des simulations numériques

Traditionnellement, les scientifiques qui manipulent des codes de simulation procèdent en mode *batch* (Fig. 1.1). La simulation produit des résultats, qui sont stockés dans des fichiers, puis ces résultats sont analysés à la fin de la simulation avec des outils de visualisation (*post-processing*). A ce niveau, le scientifique peut choisir de « rejouer » la simulation avec un nouveau jeu de paramètres, et ainsi de suite. Cette approche a l'avantage d'être simple à mettre en œuvre et de permettre un archivage des résultats produits par la simulation. Toutefois, elle nécessite beaucoup de manipulations, rendant l'analyse de sensibilité d'un grand ensemble de paramètres fastidieuse voir décourageante. En effet, comme les centres de calculs ne sont généralement pas très adaptés à la visualisation, il est souvent nécessaire de rapatrier l'ensemble des fichiers produits vers une station locale de visualisation (e.g. par FTP), ce qui retarde d'autant le travail d'analyse du scientifique.

La simulation interactive (ou *computational steering*) a pour but d'améliorer le processus de simulation numérique (modélisation, calcul, analyse) en le rendant plus interactif (Fig. 1.2). Dans cette approche, l'utilisateur n'attend plus passivement les résultats de la simulation. Au contraire, il visualise « en temps réel » l'évolution des calculs, et peut interagir à tout moment en modifiant certains paramètres du modèle et plus généralement en pilotant le déroulement des calculs. Nous n'opposons pas à proprement parler la simulation interactive avec la simulation en *batch* ; nous la considérons plutôt comme une approche alternative et complémentaire qui

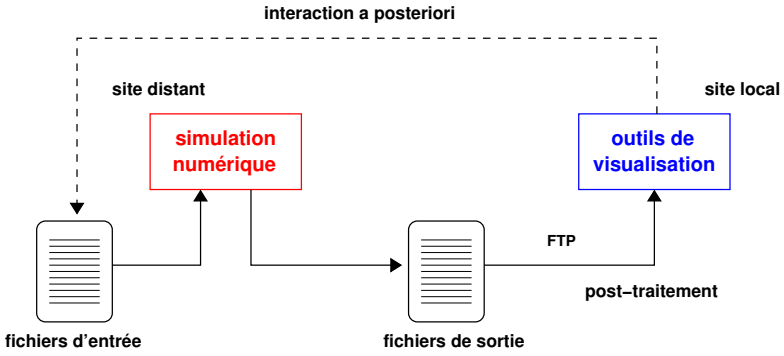


FIG. 1.1 – Processus de simulation numérique en mode *batch* (*post-processing*).

permet d'augmenter la productivité et l'efficacité de l'analyse en réduisant considérablement le temps entre le changement des paramètres et la visualisation des résultats. Cette approche peut s'avérer très utile pour la détection rapide d'erreurs, notamment dans le cas des simulations longues. De plus, en modifiant certains paramètres et en visualisant immédiatement les effets produits sur le modèle, les relations de cause-à-effet deviennent plus évidentes et le scientifique retrouvent une démarche expérimentale. Il peut faire jouer son intuition, explorer le modèle, élaborer des hypothèses et les vérifier rapidement. Du point de vue informatique, la simulation interactive s'inscrit dans une problématique plus générale de couplage de codes. En effet, les calculateurs dédiés au calcul hautes performances ne sont généralement pas adaptés pour la visualisation. Il est donc nécessaire de faire migrer les données par le réseau vers des stations graphiques distantes.

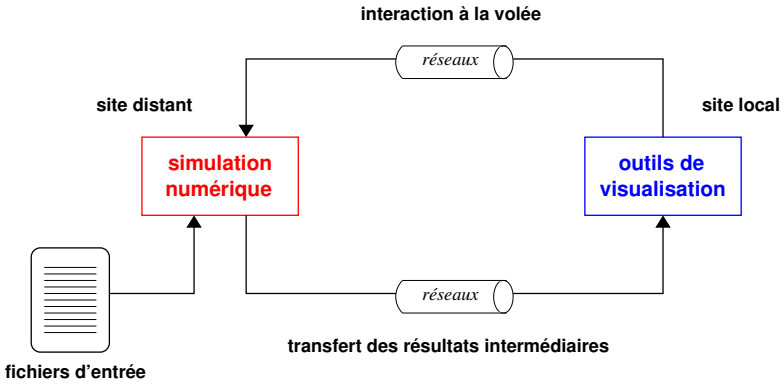


FIG. 1.2 – Simulation interactive.

Sans l'utilisation du parallélisme, la capacité globale en visualisation des ordinateurs ne peut suivre la croissance exponentielle des données générées par les calculateurs hautes performances. Pour illustrer notre propos, nous allons considérer plus particulièrement le code océan POP [124] du projet CCSM. Son modèle utilise une grille curvilinéaire de $3600 \times 2400 \times 40$ cellules pour représenter les océans (du fond à la surface). Le stockage en mémoire de la température ou de la salinité nécessite 1.4 Go de données en simple précision. La reconstruction du maillage pour la visualisation nécessite alors 4.2 Go de mémoire supplémentaire. Dans cet exemple, le calcul en parallèle d'une iso-surface en température, comme celle représentée sur la figure 1.3, prend, selon *Ahrens et al.* [128], 0.75 secondes sur 16 processeurs contre 8.25 secondes sur 1 seul ; le rendu en parallèle de cette iso-surface prend 4.12 secondes sur 16 processeurs contre 15.6 sur 1 processeur. En réduisant le niveau de détails, le temps de rendu n'est plus que de 0.67 secondes sur 16 processeurs. L'acheminement des informations de température, soit 1.4 Go de données, sur un réseau giga-ethernet atteignant une bande passante réelle de 90 Mo/s est de l'ordre d'une dizaine de secondes. Si l'on considère à présent un code de visualisation parallèle utilisant 16 PCs, possédant chacun sa propre interface réseau, le temps de transfert des données est idéalement divisé par 16 et devient inférieur à la seconde (si la capacité du *backbone* le permet). Cette exemple montre clairement que l'utilisation d'un système de visualisation parallèle couplé à la simulation peut être avantageuse. D'une part, il permet de traiter efficacement de grands ensembles de données. D'autre part, il permet d'accélérer le transfert

des données en utilisant des flux de communication parallèles, ce qui permet d'éviter le goulot d'étranglement lié à la sérialisation des messages vers un programme de visualisation séquentiel.

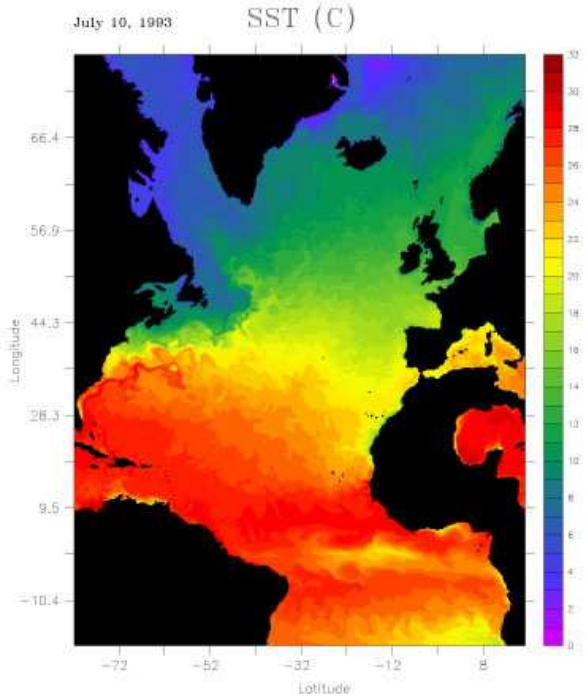


FIG. 1.3 – Visualisation de la température à la surface de l'atlantique nord, simulée par le code POP [124] du projet CCSM.

Objectifs de la thèse et contributions

L'objectif de cette thèse est de concevoir et de développer une plate-forme logicielle, appelée EPSN (Environnement pour le Pilotage des Simulations Numériques), permettant de piloter une application numérique parallèle en s'appuyant sur des outils de visualisation eux-mêmes parallèles. En d'autres termes, il s'agit de mettre au service des scientifiques les capacités de la visualisation parallèle et plus largement de la réalité virtuelle, une étape qui est aujourd'hui cruciale dans la conception de simulations numériques complexes en vraie grandeur. Depuis l'apparition à la fin des années 80 des premiers environnements de pilotage, des solutions ont été proposées pour piloter des applications parallèles, mais la prise en compte du parallélisme au niveau de la visualisation reste toujours un problème ouvert. La mise en œuvre d'un couplage efficace entre simulation et visualisation soulève deux problèmes majeurs que nous étudions dans cette thèse et pour lesquels nous souhaitons apporter une contribution :

Le problème de la coordination efficace des opérations de pilotage. Comment coordonner efficacement les opérations de pilotage en parallèle pour garantir la cohérence des traitements effectués ? Plus précisément, il s'agit de mettre en œuvre des techniques d'accès asynchrone aux données de la simulation tout en garantissant que les données extraites (en mémoire distribuée) soient globalement cohérentes en temps, c'est-à-dire qu'elles correspondent à la même étape élémentaire du calcul. Notre approche repose sur la définition d'un modèle de description de la simulation, baptisé MHT (Modèle Hiérarchique en Tâches), qui permet de se repérer dans l'exécution d'un code parallèle grâce à l'introduction d'un système de datation.

Le problème de la redistribution pour des données complexes. Comment redistribuer et transférer efficacement les données entre un code de simulation et d'un code de visualisation ? En effet, le couplage de ces deux codes parallèles peut conduire à des redistributions de données plus irrégulières que celles étudiées classiquement. Les données d'une simulation sont généralement distribuées pour optimiser les calculs

numériques, tandis que du côté de la visualisation, la distribution est plutôt issue d'un découpage spatial. Par ailleurs, il est important d'étendre les études précédentes pour prendre en compte des données complexes, comme des grilles structurées, des ensembles de particules ou des maillages irréguliers.

Organisation de la thèse

Cette thèse est divisée en trois parties. La première partie introduit les notions de base concernant la simulation numérique, le couplage de codes et la visualisation scientifique (chapitre 2), et présente un état de l'art sur la simulation interactive (chapitre 3). Dans la deuxième partie, nous définissons un modèle de haut-niveau pour le pilotage fin de simulations numériques parallèles, qui permet de résoudre le problème de la cohérence temporelle en coordonnant efficacement les opérations de pilotage (chapitre 4). Dans le chapitre suivant (chapitre 5), nous proposons un ensemble d'algorithmes pour la redistribution de données régulières (tableaux, grilles) ou irrégulières (particules, maillages non structurés). La dernière partie est une validation du modèle et des algorithmes exposés dans la partie précédente. En particulier, le chapitre 6 décrit la réalisation de l'environnement de pilotage EPSN basée sur la technologie CORBA [163] et s'appuyant sur la bibliothèque de redistribution RedSYM. Nous concluons au chapitre 7 en présentant des résultats expérimentaux sur des applications numériques dans divers domaines, comme la mécanique des fluides ou la dynamique moléculaire. Le chapitre 8 formulera quelques perspectives concernant l'ensemble de nos travaux.

Projet EPSN

EPSN[1] (Environnement pour le Pilotage de Simulations Numériques distribuées) est un projet réalisé au titre de l'ACI GRID, au sein du projet ScAlApplix[2] de l'INRIA Futurs, commun au LaBRI[3] et au MAB[4]. Les travaux de développements de la plate-forme logicielle EPSN ont été réalisés en collaboration avec Michaël Dussere, ingénieur expert à l'INRIA rattaché au projet EPSN. Par ailleurs, une partie des travaux concernant la redistribution des données (cf. chapitre 5) ont été initié au sein de l'ARC INRIA RedGRID[5] (Redistribution de données de grandes tailles dans les grilles de calcul).

[1] EPSN : `http ://www.labri.fr/epsn`
[2] ScAlApplix : Schémas et Algorithmes Hautes Performances pour les Applications Scientifiques Complexes, `http ://www.labri.fr/scalapplix`
[3] LaBRI : (Laboratoire Bordelais de Recherche en Informatique, UMR CNRS 5800, `http ://www.labri.fr`
[4] MAB : Mathématiques Appliquées de Bordeaux, UMR CNRS 5466, `http ://www.math.u-bordeaux.fr/maths/index.php?site=mab`
[5] RedGRID : `http ://graal.ens-lyon.fr/~desprez/REDGRID`

Première partie

Cadre général et positionnement de l'étude

Chapitre 2

Simulation numérique, couplage de codes et visualisation scientifique

Sommaire

Le pilotage des simulations est au croisement de plusieurs thématiques de recherche importantes : la simulation numérique, le couplage de codes et la visualisation scientifique. Dans ce chapitre, nous présentons les notions de base sur ces thématiques qui seront utiles à la compréhension de cette thèse.

2.1 La simulation numérique

Depuis l'invention de l'informatique, les scientifiques utilisent les ordinateurs pour simuler numériquement des phénomènes physiques complexes et répondre à des questions non accessibles par des voies purement expérimentales ou théoriques. La simulation numérique permet de mieux comprendre les phénomènes, d'affiner leur modélisation, et ce aussi bien pour des applications académiques qu'industrielles. Nous définissons de manière générale la simulation numérique comme étant le processus qui permet de reproduire numériquement sur ordinateur un phénomène décrit par un ou plusieurs modèles mathématiques.

2.1.1 Le processus de simulation numérique

Conceptuellement, on peut décomposer le processus de simulation numérique en plusieurs étapes : *modélisation*, *approximation* et *calcul*. Tout d'abord, le scientifique doit construire un modèle mathématique du problème physique qui l'intéresse. Ce modèle est généralement constitué d'un ensemble d'équations aux dérivées partielles (EDP) exprimant les lois d'évolution du phénomène. Il faut également spécifier le domaine de définition du problème : la géométrie du domaine (cube, sphère, profil d'une aile d'avion, *etc*) et les propriétés physiques associées (conductivité électrique, densité, viscosité, *etc*). A cette physique du problème sont associées des conditions initiales et des conditions aux limites. Les conditions initiales sont simplement les données initiales du problème considéré, comme par exemple les conditions atmosphériques courantes dans une simulation climatique. Les conditions aux limites, quant à elles, sont des contraintes souvent définies sur les bords du domaine de calcul et couplées au système d'équations du modèle. Elles peuvent représenter les forces extérieures du problème, comme par exemple la vitesse du vent à l'entrée d'un tunnel, les températures à la frontière du domaine, *etc*. Une fois la mise en équations réalisée, on obtient un système d'EDP qu'un ordinateur ne peut résoudre directement. Il convient donc de développer une approximation numérique en discrétisant les équations avec certaines méthodes comme celle des différences finies, des éléments finis ou des volumes finis. Cette étape consiste à remplacer les opérateurs continus par leurs approchés discrets, ce qui implique généralement une discrétisation du domaine de définition à l'aide d'un maillage et la mise en place d'un schéma numérique explicite (calcul direct) ou implicite (résolution de systèmes linéaires éventuellement creux). La solution numérique se présente alors comme un ensemble de valeurs discrètes aux nœuds du maillage. A ce niveau, on devine que la solution numérique sera d'autant plus précise, c'est-à-dire plus proche de la solution exacte de l'EDP, que le maillage sera fin. Cela conduit les scientifiques à résoudre des problèmes de plus en plus gros et complexes, pouvant atteindre jusqu'à un milliard d'inconnues et nécessitant donc des super-calculateurs pour y parvenir. Une fois la solution du problème trouvée, le scientifique doit déterminer si ce résultat est correct. Afin de valider ce résultat et plus largement le modèle, il existe plusieurs stratégies : calcul d'invariants, confrontation du résultat à des données expérimentales, vérification sur des cas simples dont il est possible de calculer une solution analytique, *etc*. Une fois que l'on est conforté dans l'idée que le résultat de la simulation est correct, commence le travail d'exploration, de dépouillement et d'analyse des données. Cela implique l'utilisation de programmes informatiques réalisant des post-traitements, comme les outils de visualisation scientifique dont nous allons parler à la section 2.3.

2.1.2 Caractérisation des données scientifiques

Les données manipulées dans les codes de simulations numériques sont généralement des informations quantitatives réelles (scalaires ou vecteurs), stockées en virgule flottante (norme IEEE), en simple ou double précision (resp. *float* ou *double* en langage C). Comme nous l'avons dit précédemment, les équations manipulées dans les codes de simulation nécessitent une discrétisation de l'espace, ce qui implique l'utilisation d'un maillage décrivant le domaine de calcul. Formellement, un maillage est un graphe (V, E), où V représente l'ensemble des nœuds (*vertex*) et E l'ensemble des arêtes (*edge*), et dont la spécificité tient au fait que les arêtes sont regroupées en mailles ou éléments géométriques (e.g. triangle, quadrilatère, tetraèdre, hexaèdre, *etc*.) [216]. De plus, comme les maillages ont une dimension spatiale (2D ou 3D), chaque nœud est repéré dans l'espace par une coordonnée géométrique (notée (x, y) en 2D ou (x, y, z) en 3D). Les inconnues sont associées soit aux nœuds, soit aux éléments du maillage (en général le barycentre de l'élément) et plus rarement aux arêtes.

Il existe deux familles de maillages : *les grilles* (ou *maillages structurés*) et *les maillages non structurés*. La principale différence tient au fait que les grilles possèdent une topologie régulière, généralement formée de quadrilatères (2D) ou d'hexaèdres (3D). De plus, les grilles possèdent un système naturel de coordonnées qui permet de désigner un élément avec sa coordonnée topologique (notée (i, j) en 2D ou (i, j, k) en 3D). A l'opposé, les maillages non structurés (Fig. 2.1) possèdent une topologie irrégulière et par conséquent, la connectivité entre les nœuds doit être donnée explicitement élément par élément (liste de connectivités). Dans ce cas, le nombre d'éléments partageant un nœud n'est pas nécessairement constant. En général, les maillages considérés sont conformes, c'est-à-dire que l'interface entre deux éléments connectés (arrête en 2D ou face en 3D) n'est jamais partielle. Un maillage est dit généralisé s'il mélange plusieurs types d'éléments, ce qui peut s'avérer utile dans certaines simulations pour capturer certains détails. Du point de vue de la simulation numérique, la méthode

des différences finies impose d'utiliser une topologie régulière. Par conséquent, les grilles structurées sont bien adaptées. Avec la méthode des éléments finis ou des volumes finis, cette contrainte est supprimée et cela peut conduire à utiliser des maillages complexes à topologie irrégulière, comme sur la figure 2.1.

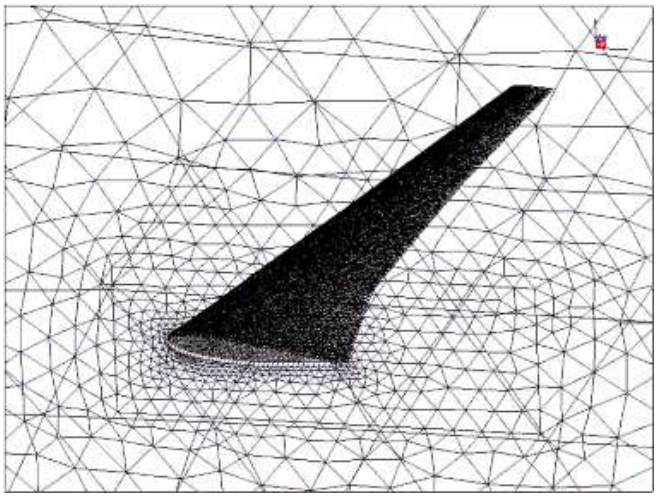


FIG. 2.1 – Maillage non structuré représentant le profil d'une aile d'avion.

Comme le montre la figure 2.2, on distingue trois types de grilles : les grilles rectilinéaires, curvilinéaires et irrégulières. Les grilles rectilinéaires et curvilinéaires ont une géométrie régulière, ce qui conduit au fait que les coordonnées géométriques des nœuds ne sont pas stockées mais calculées en fonction des coordonnées topologiques. A l'inverse, dans le cas des grilles irrégulières, chaque nœud possède explicitement en mémoire une coordonnée géométrique. Une grille rectilinéaire est dite uniforme, si l'espacement est identique selon toutes les directions de l'espace. Dans certains ouvrages faisant référence à l'imagerie médicale (IRM), on parle encore d'image 2D ou 3D pour désigner les grilles uniformes. De même qu'une image en 2D est constituée de pixels, l'image 3D est constituée de voxels (cubes). D'autres auteurs parlent d'ensemble de points structurés ou de champs de données (*field*). Il existe encore une dernière catégorie de maillages dégénérés, ne possédant pas de connectivités, *les ensembles de points non structurés*. Les points sont localisés dans l'espace de manière irrégulière. Ils sont fréquemment utilisés dans les simulations particulaires et en astro-physique.

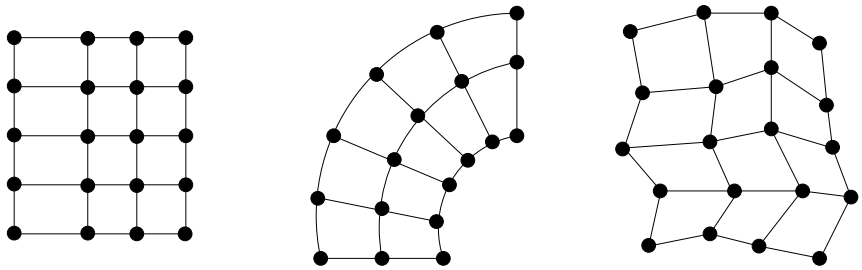


FIG. 2.2 – Exemples de grilles 2D.

En conclusion, il convient de souligner que ce catalogue n'est en rien exhaustif. Il faudrait encore parler des techniques de maillages adaptatifs (AMR), de la chimie moléculaire qui utilisent des descriptions standardisées du type PDB (Protein Data Bank) [26], ainsi que des formats de description de données scientifiques généralistes comme HDF5 (Hierarchical Data Format 5) [156] ou NetCDF [20].

2.1.3 Placement et distribution des données

Dans le cadre de la programmation des architectures parallèles à mémoire distribuée, le placement des données sur les processeurs s'avère de première importance. En effet, c'est ce placement qui conditionne à la fois

l'efficacité des algorithmes parallèles et la réduction du surcoût des communications, l'objectif étant de minimiser le temps total d'exécution. Un placement est dit *statique* s'il est pré-calculé avant le lancement du programme et n'est jamais remis en cause par la suite, et *dynamique* si les processus (et leurs données) peuvent être déplacés en cours d'exécution ; dans ce dernier cas, il faut faire appel à des outils de régulation de charge pour effectuer la migration des processus. De manière générale, le calcul d'un placement statique « optimal » va dépendre d'une fonction de coût permettant d'évaluer la distribution des calculs et la réduction des communications. Dans un contexte SPMD (Single Process Multiple Data), le problème de placement statique revient simplement à *distribuer les données* sur les processeurs. Les techniques de distribution des données diffèrent selon que les données sont régulières (e.g. tableaux, grilles) ou irrégulières (e.g. maillages, graphes). Les tableaux sont usuellement distribués *en bloc-cyclique*, tandis que pour les maillages, il faut recourir à des techniques plus complexes de *partitionnement*.

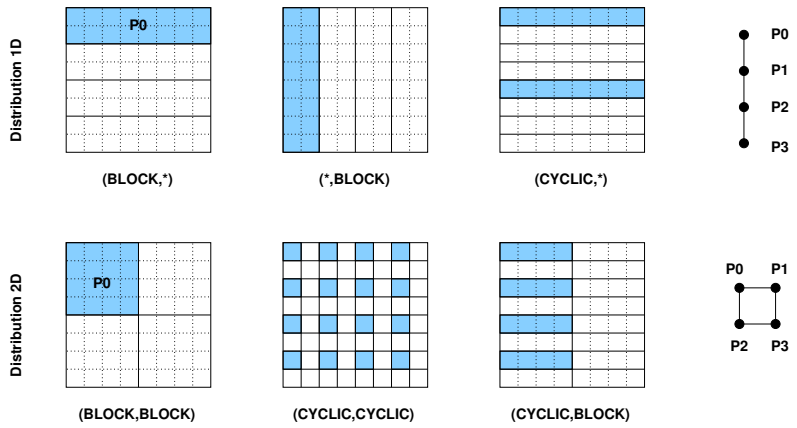


FIG. 2.3 – Distribution HPF classiques d'un tableau (virtuel) de taille 8×8 sur une grille de 4 processeurs. Pour chaque exemple, les données colorées sont associées au processeur P_0.

Les distributions bloc-cycliques de tableaux

Les distributions manipulées dans le domaine du calcul scientifique sont principalement des distributions bloc-cycliques de tableaux, car elles possèdent de bonnes propriétés, notamment en terme d'équilibrage de charge et d'efficacité des calculs. Comme le montre *Dongarra et al.* [73], ces distributions sont essentielles en algèbre linéaire pour réaliser des algorithmes efficaces. Ainsi un gros effort de standardisation de ces distribution a été effectué par le passé, largement motivé par les travaux autour de HPF (High Performance Fortran) [121] et des bibliothèques d'algèbre linéaire comme ScaLAPACK [50]. HPF propose un schéma de placement des données à deux niveaux. Tout d'abord, les données d'un tableau sont alignées sur un tableau virtuel, appelé *template* via la directive ALIGN. Puis ce tableau est distribué sur une grille cartésienne de processeurs via la directive DISTRIBUTE. L'alignement sur un *template* est un mécanisme puissant qui permet de s'abstraire des différentes stratégies de stockage utilisées dans les codes (e.g. offsets, strides, *row/column major*, *etc.*). La directive DISTRIBUTE est utilisée pour indiquer comment les données associés au *template* sont réparties dans la mémoire des divers processeurs. Cette directive spécifie, pour chaque dimension, les indices du *template* affectés à un processeur logique. Comme le montre la figure 2.3, chaque dimension d'un tableau peut être distribuée de trois façons : distribution en bloc (BLOCK), distribution cyclique (CYCLIC), pas de distribution ($*$). A titre d'exemple, considérons un tableau (ou *template*) 1D de taille n et un ensemble de p processeurs. Une distribution en bloc divise les indices du tableau en p blocs d'indices contiguës de taille n/p. Une distribution cyclique associe chaque p-ième indice au même processeur. Les distributions en bloc-cyclique (CYCLIC(s) avec s la taille du bloc) sont une généralisation des distributions en bloc (cas $s = \lfloor n/p \rfloor$) et des distributions cycliques (cas $s = 1$). Dans ce cas, le i-ème indice du *template* est affecté au processeur de rang (i/s) % p.

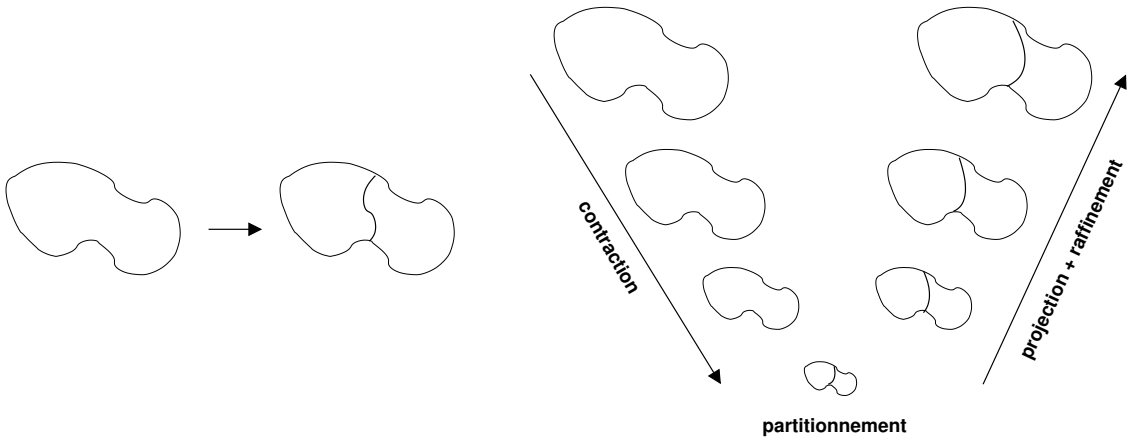


FIG. 2.4 – Algorithmes de partitionnement directs et multi-niveaux.

Le partitionnement de graphes et de maillages

Une des principales applications du partitionnement est le découpage des graphes de maillages utilisés en calcul parallèle, mais on peut également appliquer ces techniques à des objets plus complexes comme des graphes de molécules (Fig. 2.5). De manière générale, le problème de placement statique est NP-complet [87]. Par conséquent, la recherche d'une solution optimale nécessite une exploration combinatoire de l'espace des solutions, le plus souvent à l'aide d'un arbre de recherche. Les méthodes de type « *branch and bound* » permettent d'éviter de parcourir toutes les branches, en « élaguant » les sous-arbres menant à des solutions toutes moins bonnes qu'une solution donnée. De nombreuses heuristiques ont été proposées afin de trouver des solutions sous-optimales en un temps polynomial. Elles sont issues de nombreux domaines dépassant largement le cadre de cette thèse, tels la théorie des graphes, l'optimisation combinatoire, *etc*.. Dans la plupart de ces méthodes, le problème du placement est considéré dans sa globalité (méthodes directes) et l'espace des solutions explorées augmente rapidement par rapport à la taille du problème, ce qui rend ces méthodes coûteuses. Afin de réduire la taille des problèmes considérés, les algorithmes multi-niveaux (« *multi-level* ») utilisent une approche *divide & conquer* représentée sur la figure 2.4. Dans cette approche, on construit une famille de graphes de plus en plus petits (par contraction), puis on calcule un placement valide sur le plus petit graphe et on projette le résultat de graphe en graphe jusqu'au graphe initial. A la remontée du placement, il est possible d'appliquer un algorithme d'optimisation locale afin d'affiner le placement calculé (raffinement). Actuellement, les meilleurs algorithmes de placement connus sont des algorithmes multi-niveaux, qui utilisent un algorithme de décomposition pour calculer le placement sur le plus petit graphe contracté. Les méthodes de décomposition consistent à découper récursivement le graphe en plusieurs parties, habituellement deux dans les heuristiques de bipartitionnement [172]. L'obtention de bons placements nécessite que les bipartitionnements successifs concourent à la minimisation de la fonction coût choisie, généralement en minimisant la coupe entre les parties du graphe, ce qui a pour effet de favoriser la localité des communications. Parmi les outils de placement implantant ces techniques, on peut citer Metis [116] ou Scotch [173]. L'augmentation de la taille des problèmes fait que les maillages deviennent trop gros pour être stockés dans la mémoire d'une machine séquentielle, ce qui rend nécessaire l'utilisation d'outils de placement eux-mêmes parallèles (e.g. ParMetis [117]).

2.2 Le couplage de codes

Dans l'approche par couplage de codes, une application est vue comme un assemblage de plusieurs codes, le plus souvent parallèles, collaborant à la réalisation d'un travail commun. De manière plus générale, on parlera *d'applications parallèles distribuées*, dans la mesure où ce type d'applications possèdent à la fois des caractéristiques du calcul parallèle et du calcul distribué. En effet, si le modèle de programmation parallèle semble

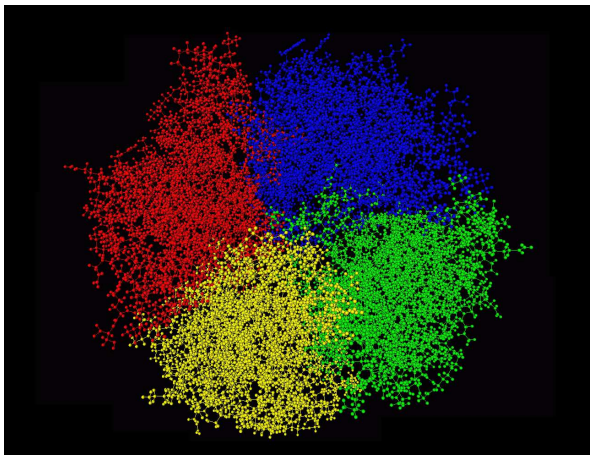


FIG. 2.5 – Partitionnement en 4 partie de la molécule d'urate-oxydase avec Scotch [173].

bien adapté pour la programmation des codes séparément, il semble plus approprié d'utiliser un modèle de programmation distribué pour réaliser le couplage entre ces codes. En effet, les applications « classiques » du parallélisme supposent mojoritairement une architecture homogène (réseaux, processeurs) et des ressources statiques (durée de vie, performance). Or le couplage de codes impose l'utilisation simultanée de plusieurs ressources hétérogènes, plus ou moins volatiles avec des performances réseaux non prévisibles (équilibrage de charge) et une certaine dynamicité entre les codes (relation client/serveur, pair-à-pair, *etc*.).

2.2.1 Quelques applications du couplage

Les simulations multi-physiques constituent certainement l'exemple le plus approprié d'application du couplage de codes. Dans ce type d'applications, le processus de résolution est généralement décomposé en plusieurs étapes, chaque étape correspondant à la résolution des équations associées à un seul type de physique (*operator splitting* [146]). Une telle approche permet de voir une application complexe comme un assemblage de plusieurs modules mono-physiques indépendants. Un tiers programme, appelé *coupleur*, est généralement en charge de coordonner les différents codes couplés. Au cours de la résolution, chaque code échange des données avec les autres codes, le plus souvent sur la frontière des domaines associés à chaque code (conditions aux limites dynamiques). Parmi les projets utilisant cette approche, on trouve les simulations du comportement climatique de la terre, les codes de prévision météorologique, les couplages de type fluide-structure, *etc*.

Les logiciels de couplage ont pour but de faciliter le développement d'applications multi-codes. Dans ce type de logiciel – dont la philosophie est proche de celle des composants logiciels [197] – une application se compose le plus souvent d'un ensemble de modules intégrés dans un *framework* gérant la connexion des modules entre eux (*driver*), la communication via des interfaces spécifiées et la coordination globale de l'exécution (*coupleur*). Certains logiciels sont spécifiques à un domaine particulier comme par exemple CCSM (Community Climate System Model) [6], OASIS [21] ou ESMF (Earth System Modeling Framework) [11] pour des exemples de simulations climatiques (Fig. 2.6). Dans CCSM, un coupleur central CPL6 permet d'interconnecter quatre codes parallèles modélisant les comportements de la terre (CLM), des océans (POP), de la glace (CSIM) et de l'atmosphère (CAM). Dans le projet ESMF, l'utilisateur a la possibilité d'instancier ses propres modèles (appelé *Gridded Component*) et de les coupler de manière très flexible (*Coupler Component*). L'infrastructure d'ESMF met l'accent sur l'interopérabilité, la facilité d'utilisation (et de réutilisation), la performance et la portabilité. D'autres logiciels comme PAWS [44], MpCCI [19], MCT [16] ou CACTUS [3] offrent des solutions plus génériques pour le couplage de codes, dont certains prennent en charge le problème délicat de redistribution des données. Nous y reviendrons au chapitre 5.

Outre les simulations multi-physiques, on trouve encore de nombreux exemples d'applications du couplage de codes dans des domaines plus ou moins reliés. En particulier, les GridRPCs [192] sont un exemple fréquemment cité, dédiés plus précisément au calcul scientifique en algèbre linéaire. Ce sont des ASPs (Application

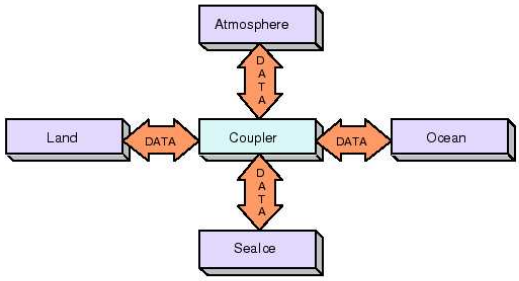


FIG. 2.6 – Schématisation du couplage dans les simulations climatiques ESMF ou CCSM.

Service Providers) fonctionnant selon un modèle client/agent/serveur. Dans NetSolve [57] ou Diet [8], un programme client soumet à un agent une requête de calculs. L'agent est alors en charge de distribuer les calculs sur un ensemble de serveurs parallèles (en fonction de leur disponibilité) avant de retourner le résultat au client. Les environnements de pilotage constitue un autre exemple d'application du couplage de codes. Dans les applications de pilotage, une simulation numérique est couplée à un processus interactif permettant à un utilisateur de contrôler le déroulement de la simulation. L'interface utilisateur intègre généralement un code de visualisation permettant de suivre l'évolution des calculs. Par exemple dans CUMULVS [122], plusieurs utilisateurs peuvent se connecter simultanément à une simulation en utilisant des programmes de visualisation (séquentiels) tel que AVS [2]. Nous étudierons plus en détails ce type d'application au chapitre suivant. Pour achever ce catalogue, il convient encore de citer quelques travaux reliés dans le domaine de *l'Internet Computing*, comme Seti@Home [28], Genome@Home [13] ou XtremWeb [81].

2.2.2 Caractéristiques du couplage

De manière générale, le couplage de codes a pour objectif d'améliorer la qualité logicielle en prônant la réutilisabilité des codes. Dans cette approche, chaque code peut être assimilé à *un module* ou à *un composant*, dont la conception, le développement et le déploiement doivent pouvoir s'effectuer de manière relativement indépendante des autres codes. Cette caractéristique est tout à fait critique si l'on considère qu'un individu seul, voire une institution, ne peut être expert dans tous les domaines de la physique qui sont considérés. Du fait de la complexité toujours croissante des simulations numériques, il n'est donc plus envisageable de développer de manière isolée des applications monolithiques, difficiles à maintenir et à faire évoluer [88]. D'autre part, il faut permettre aux institutions de préserver leurs applications « patrimoines » (*legacy applications*), c'est-à-dire de les intégrer avec les technologies nouvelles plutôt que de les reconstruire entièrement ; ceci encourage l'utilisation de modèles de programmation de haut-niveau (*middlewares*) permettant une structuration logique des codes couplés et une encapsulation des codes existants.

Par ailleurs, il faut prendre en compte la distribution géographique des codes sur différents sites et cela pour plusieurs raisons. Tout d'abord, la distribution des codes permet de mobiliser davantage de ressources de calculs, ce qui va donc favoriser la résolution de problèmes plus grands ou à grain plus fin. De plus, les applications distribuées doivent permettre d'utiliser la meilleur combinaison de technologies disponibles pour résoudre un problème en prenant compte des besoins spécifiques à chaque code (puissance de calcul, capacité mémoire, stockage, carte graphique, *etc*). L'architecture cible de ce type d'application est *une grille de calcul* [84], c'est-à-dire une grappe de calculateurs hétérogènes interconnectés par des réseaux plus ou moins performants (e.g. VTHD [32]). L'hétérogénéité des grilles est due à la grande diversité des technologies existantes en informatique (réseaux, machines, systèmes d'exploitation, langages), ce qui implique de prendre en charge des problèmes d'interopérabilité entre toutes ces technologies. Plus généralement, l'interopérabilité doit passer par la définition de protocoles et de normes « *neutral-vendor* » permettant de faire communiquer des implantations différentes d'une même technologie (e.g. les sockets pour TCP/IP, IMPI [91] pour MPI, IIOP pour CORBA [161], *etc*.) voire de réaliser des ponts entre différentes technologies (e.g. « RMI over IIOP » [27], CCM-EJB [162]).

Le déploiement d'applications sur la grille pose des problèmes supplémentaires d'ordre administratif. Ces problèmes sont principalement liés aux différentes politiques de sécurité et d'allocation des ressources (*batch scheduler*) entre les sites. Certains outils comme Globus [83] ou Legion [97] offrent une solution globale aux problèmes de la programmation et du déploiement de codes sur la grille. Dans ces travaux, la grille est alors vue comme un super-calculateur virtuel (*meta-computer*).

Comme nous venons de le voir, la conception, la réalisation et le déploiement d'applications (parallèles) distribuées sont des tâches complexes. De nombreuses caractéristiques doivent être prises en compte : la communication entre les applications, l'hétérogénéité de celles-ci, l'intégration de l'existant ainsi que l'interopérabilité des diverses technologies. A ces problèmes viennent s'ajouter d'autres difficultés rendant le développement d'applications distribuées encore plus complexe : la découverte des ressources sur le réseau, la sécurité, la réalisation de traitements transactionnels, la persistance des informations, le déploiement, *etc*. De nombreuses solutions ont été envisagées dans le domaine de l'informatique parallèle et répartie pour construire des applications parallèles distribuées et coupler des codes : paradigmes de communication par envoi de messages (MPI [99], PVM [90]), appels de procédure à distance (Sun RPC [140], DCE [164]), objets distribués (RMI [215], DCOM [137], CORBA [163]), environnements de *metacomputing* (Legion [97], Globus [83]) ou encore systèmes d'exploitation distribués comme Kerrighed [149]. Dans la suite, nous examinerons trois approches différentes du couplage de codes : l'approche parallèle avec MPI, l'approche distribuée avec CORBA et l'approche composant.

2.2.3 L'approche parallèle du couplage avec MPI

La grande majorité des codes scientifiques sont des applications parallèles utilisant des bibliothèques d'échange de messages pour communiquer comme PVM (Parallel Virtual Machine) [90] et plus particulièrement MPI (Message Passing Interface) [99] que l'on peut aujourd'hui considérer comme le standard *de facto* de la programmation parallèle par échange de messages. Une application MPI est composée d'un nombre de processus fixé à son démarrage (du moins pour MPI-1), communiquant par échange de messages, exécutant un ou plusieurs programmes en parallèle (SPMD ou MPMD). MPI permet de communiquer en point-à-point (*send()/receive()*) ou via des opérations collectives (*gather(), broadcast(), etc*.) et offre différents modes de dialogue (synchrone/asynchrone, bloquant ou non, avec tampon ou sans). MPI n'étant qu'un standard, il en existe de nombreuses implantations dont la plupart sont fournies par les constructeurs de machines parallèles. Ces implantations sont optimisées pour un type de machine particulier, mais sont rarement interopérables. La spécification IMPI (Interoperable MPI) [91, 194] vise à remédier à ce défaut, mais à ce jour très peu de constructeurs ont décidé de suivre cette norme. Quelques implantations de MPI comme MPICH [98] ou LAM/MPI [55] (conforme à la norme IMPI) permettent de déployer des applications MPI sur des clusters hétérogènes. Toutefois, le manque de dynamicité intrinsèque à MPI-1 ne permet pas d'utiliser cette technologie pour coupler des applications MPI indépendantes (i.e. démarrées séparément). Notons que la version MPI-2 [136] permet dans une certaine mesure la connexion dynamique de plusieurs programmes MPI selon un modèle client/serveur et l'ajout dynamique de processus au sein d'un communicateur (*spawn*).

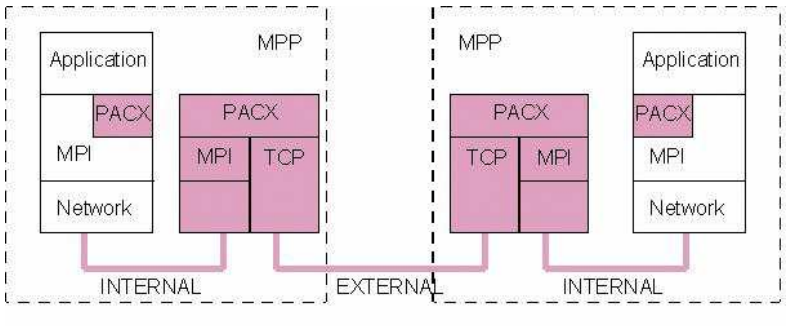


FIG. 2.7 – PACX-MPI [45].

Afin de coupler des codes avec MPI, tout en conservant un modèle de programmation « à la MPI », plusieurs solutions ont été envisagées. PACX-MPI [45] et PVMPI [76] sont des extensions à la bibliothèque MPI qui permettent de faire communiquer des MPIs issus de plusieurs constructeurs. L'idée consiste à utiliser le « meilleur > MPI sur chaque machine parallèle pour effectuer les communications internes (intra-codes) et à utiliser une autre technologie pour réaliser les communications externes (inter-codes) de manière transparente à l'utilisateur. Par exemple dans PACX-MPI, les communications externes sont bufferisées (encodage XDR) sur un noeud PACX dédié à l'envoi, puis sont acheminées via TCP/IP vers un noeud PACX distant dédié à la réception avant d'être redistribué (Fig. 2.7). PVMPI fonctionne selon le même principe, mais ajoute des possibilités dynamiques de PVM à MPI, comme notamment la connexion dynamique de différents codes MPI. Dans PVMPI, les communications intra-codes sont réalisées avec MPI et les communications externes (inter-codes) utilisent PVM.

MPICH-G2 [115] est une implantation de MPI (dérivée de MPICH) totalement intégrée à Globus [83]. Globus permet de programmer la grille comme un super-calculateur parallèle virtuel (*meta-computer*) et résout la plupart des problèmes liés à la communication et à la redirection des entrées/sorties (GlobusIO), au déploiement et à l'allocation des ressources (GRAM), et enfin à la sécurité (GSI). A la différence des environnements précédents, MPICH-G2 est « une vraie » implantation de MPI-1 pour la grille. En particulier, MPICH-G2 utilise la bibliothèque GlobusIO pour les communications externes et prend en charge le déploiement des codes sur la grille de manière assez transparente (*mpirun* au-dessus de GRAM). Toutefois, MPICH-G2 conserve les limitations de MPI-1 et ne permet pas de connecter dynamiquement plusieurs codes MPI. Notons tout de même que les services Globus ne sont en aucun cas limités au support de MPI et sont tout à fait réutilisables pour déployer d'autres types d'application sur la grille.

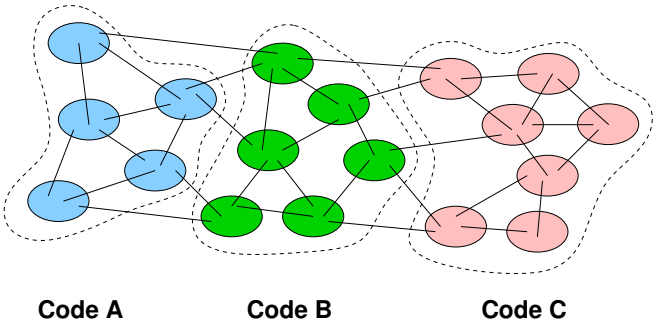


FIG. 2.8 – Vision « plate » des processus dans l'approche parallèle du couplage.

En conclusion, ces technologies permettent d'intégrer facilement des codes MPI existants, en conservant un modèle de programmation « à la MPI », ce qui implique très peu de modifications de ces codes et permet de mobiliser efficacement un grand nombre de ressources pour un parallélisme massif. Toutefois, le manque de dynamicité et l'absence de structuration des processus et des communications limitent l'utilisation de ces technologies pour coupler des codes. Comme le souligne la figure 2.8, tous les processus jouent le même rôle et les communications inter-codes et intra-codes sont au même niveau. L'absence de hiérarchisation de l'application et des communications rend difficile la gestion de ce type d'application sur une grappe de grappes.

2.2.4 Autour des middlewares et de CORBA

Généralités sur les middlewares

Un *middleware* est une plate-forme logicielle constituée d'un ensemble de services permettant d'interconnecter plusieurs programmes distribués sur une ou plusieurs machines [47]. Les services prennent en charge un ensemble de difficultés comme la communication, l'interopérabilité, l'intégration, la localisation des ressources ou la sécurité, ce qui facilite le développement d'applications distribuées. Parmi les *middlewares* les plus connus, on peut citer Sun RPC [140], DCE [164], Java RMI [215], COM/DCOM [137] ou encore CORBA [163]. Les *middlewares* se répartissent en deux grandes familles selon le mécanisme de communication qu'ils utilisent : des *appels de procédure à distance* (Remote Procedure Call ou RPC) ou des *invocations de méthodes à distance* (Remote

Method Invocation ou RMI). Contrairement à la communication par messages, les RPCs et les RMIs offrent des solutions plus transparentes s'intégrant naturellement dans les langages de programmation. En effet, la communication par messages est difficile à mettre en oeuvre pour le développement d'applications réparties, car elle repose sur une API de bas-niveau qui n'est pas intégrée au langage de programmation. Les RPCs permettent de concevoir une application distribuée comme étant un ensemble de procédures distribuées dans les divers processus serveurs. Le mécanisme RPC masque alors au processus client la couche de communication ainsi que la localisation de la procédure distante. L'appel d'une procédure distante se traduit par un appel de procédure local pris en charge par la souche cliente, qui transmet la requête via le réseau à la souche serveur (Fig. 2.9). Cette dernière est alors responsable d'appeler la procédure localement sur le serveur et de retourner les éventuels résultats au client. Dans les RPCs de Sun [140], les problèmes liés à l'hétérogénéité sont pris en charge de manière transparente par les souches grâce à une conversion des données échangées via le format XDR (eXternal Data Representation) [139]. Afin de faciliter le travail des développeurs, les souches sont générées automatiquement à partir d'une description des procédures dans le langage Sun RPCL (Remote Procedure Call Language).

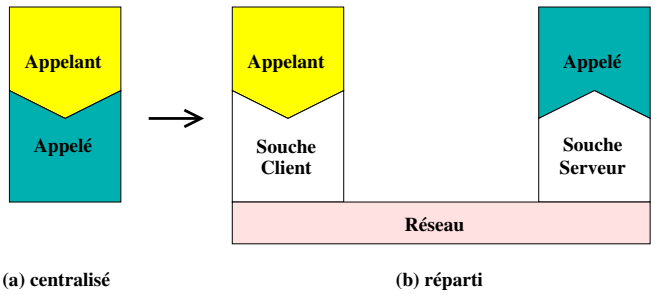


FIG. 2.9 – Passage du centralisé au réparti. Les souches (*stubs*) sont en charge de masquer l'hétérogénéité et de rendre transparent l'utilisation du réseau entre les applications.

Les RMIs sont en fait une extension des RPCs dans le contexte des objets, initialement introduits dans une version distribuée du langage SmallTalk [46]. La différence majeure entre les RPCs et les RMIs tient au fait que les RMIs manipulent des types représentés par la classe de l'objet. Une application distribuée est alors constituée d'un ensemble *d'objets distribués* communiquant de manière transparente par des appels de méthodes sur des objets distants. Dans un contexte centralisé, la communication entre les objets est prise en charge par le noyau d'exécution du langage utilisé. Dans un contexte réparti, le langage objet doit coopérer avec le système d'exploitation et les couches de communication réseau pour réaliser les invocations sur les objets distants. Comme dans le cas des RPCs, les objets distribués communiquent de manière transparente par l'intermédiaire des souches (Fig. 2.9). La souche cliente joue le rôle d'un *proxy* en représentant localement l'objet distant. Elle encapsule les informations de localisation de l'objet distant et implante les mécanismes de communication réseau pour invoquer l'objet distant.

CORBA

CORBA (Common Object Request Broker Architecture) [161] est une spécification de l'OMG (Object Management Group) [22] qui décrit une architecture portable capable de faire collaborer des applications distribuées et hétérogènes. CORBA propose un modèle client/serveur de coopération entre les applications réparties. Chaque application peut exporter certaines de ses fonctionnalités sous forme d'*objets*. L'application utilisant un objet joue le rôle du client et l'application en attente des requêtes du client joue le rôle du serveur. La partie du serveur implantant l'objet est appelé *servant* (Fig. 2.10). L'architecture de CORBA se base sur le concept de *bus* logiciel, appelé ORB (Object Request Broker) : l'ORB est l'élément central de l'architecture CORBA en charge de véhiculer les requêtes et d'assurer la collaboration entre applications. Les interactions entre les objets sont alors matérialisées par des appels de méthodes à distance (RMI). Grâce à la spécification des protocoles GIOP (General Inter-ORB Protocol) et IIOP (Internet Inter-ORB Protocol), CORBA masque les divers problèmes d'interopérabilité et assure la liaison entre plusieurs architectures & systèmes et entre plusieurs fournisseurs d'ORB.

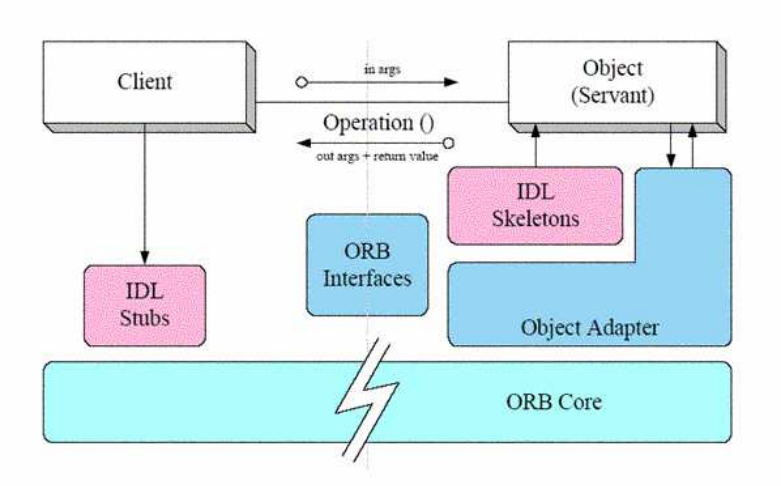


FIG. 2.10 – L'architecture de CORBA [22].

Un autre point fort de CORBA est qu'il permet de faire coopérer des applications réparties écrites dans des langages différents (C, C++, Cobol, ADA, Java, Python, *etc*.). Afin d'être indépendant d'un langage de programmation particulier, l'interface d'un objet CORBA est décrite dans un langage « neutre », appelé IDL (Interface Definition Language). La compilation d'une description IDL génère une souche côté client (*stub*) et un squelette côté serveur (*skeleton*). Remarquons que la projection des interfaces IDL dans un langage cible étant spécifié par l'OMG, il en résulte que l'implantation des objets CORBA est portable d'un ORB à l'autre. La souche cliente prend en charge le transport de la requête par le réseau vers le serveur. Côté serveur, la requête est interceptée par l'adaptateur d'objet, puis transmise au servant via le squelette. Le rôle de l'adaptateur d'objet (*object adapter*) est de gérer le cycle de vie des objets CORBA. En particulier, il est responsable de l'activation des objets, étape au cours de laquelle une référence d'objet CORBA ou IOR (Interoperable Object Reference) est associée à l'implantation de l'objet. Il existe de nombreux services annexes spécifiés par l'OMG permettant de faciliter le développement d'applications réparties : service de nommage, service d'évènements, service de persistance, service de transactions, *etc*. Parmi ces services, le service de nommage (*naming service*) est certainement l'un des plus fréquemment utilisés. Ce service joue le rôle d'un annuaire permettant de localiser facilement les objets CORBA sur le réseau.

2.2.5 L'approche distribuée du couplage avec CORBA

Les modèles de programmation distribuée, à l'instar de CORBA, possèdent de nombreux avantages pour le couplage de codes. Tout d'abord, ils offrent un bonne structuration des codes et des communications grâce notamment à l'utilisation d'interfaces spécifiées des différents codes et au mécanisme d'encapsulation offert par les objets. Par ailleurs, ils supportent naturellement la dynamicité et l'hétérogénéité entre les codes. Ainsi de nombreux travaux on choisi d'utiliser des *middlewares* pour coupler des codes et plus particulièrement la technologie CORBA. Dans le projet HOMA [89] par exemple, une application de couplage de codes est vue comme une composition d'invocations de méthodes sur des objets CORBA encapsulant les codes. Dans ce contexte, HOMA cherche à optimiser le mouvement des données entre les codes couplés. Dans le GridRPC Diet [8], les requêtes de calcul sont acheminées vers les serveurs via des invocations CORBA. Toutefois, si les *middlewares* sont séduisants pour construire des applications de couplage de codes, ils présentent généralement plusieurs limitations pour l'obtention de bonnes performances, le support de certains langages comme le Fortran ou encore le support du parallélisme dans le codes. Nous allons maintenant examiner ces limitations plus en détails et présenter quelques solutions existantes.

Afin d'obtenir de bonnes performances entre les codes couplés, il est nécessaire que les *middlewares* exploitent efficacement les *réseaux rapides* (Myrinet, SCI, VIA, *etc*.). Or la plupart des *middlewares* – dont CORBA – ne supportent pas ces réseaux. Les implantations conventionnelles de CORBA n'abordent pas en général le problème de la performance et sont le plus souvent limitées au protocole TCP/IP sur Ethernet. Cependant, certaines im-

plantations de CORBA comme OmniORB [23] offrent d'excellentes performances sur Ethernet grâce notamment à une approche « zéro-copie » [130]. Par ailleurs, on trouve quelques portages de CORBA sur réseaux rapides comme Mico [18] sur VIA ou OmniORB2 [23] sur ATM et SCI. Plus récemment, *Denis et al.* ont démontré avec la plate-forme PadicoTM [68, 69] qu'il était possible d'exploiter CORBA presque aussi efficacement que MPI sur les réseaux rapides. Ils ont pu ainsi obtenir une latence de 20 μs et un débit de $240MB/s$ pour OmniORB4 sur Myrinet-2000 [66], contre 11 μs pour MPI avec le même débit. Pour parvenir à ces résultats, PadicoTM réalise le portage de CORBA sur Madeleine [40], une bibliothèque de communication permettant de « virtualiser » les réseaux rapides.

Les objets parallèles CORBA

L'utilisation des *middlewares* pour le couplage de codes est actuellement limitée par le fait qu'ils ne permettent pas d'intégrer nativement des codes parallèles. De manière générale, les applications clientes et serveurs considérées dans les *middlewares* sont des codes purement séquentiels (même si les serveurs sont souvent multi-threads pour permettre l'exécution concurrente de plusieurs requêtes). Une première approche pour coupler des codes parallèles avec CORBA consiste à ajouter dans chaque code MPI un serveur CORBA dédié aux communications inter-codes. Lors d'une telle communication, les données distribuées sur les esclaves sont d'abord rassemblées sur le nœud maître via MPI (*gather()*), puis transférées vers le maître distant via CORBA et finalement disséminées sur les esclaves distants (*scatter()*). Cette approche est comparable à celle mise en œuvre dans des environnements comme PACX-MPI ou PVMPI (cf. section précédente) et présente les mêmes défauts. Le maître représente un goulot d'étranglement pour les communications. De plus, cette approche rompt le modèle de programmation SPMD strict et nécessite de restructurer les codes couplés selon une architecture maître/esclave.

La deuxième approche introduit le concept d'objet parallèle dans CORBA pour encapsuler des codes SPMD. Dans la plupart des travaux dans ce domaine (PARDIS, PaCO, Data Parallel CORBA, *etc.*), *un objet parallèle CORBA* est vue comme une collection d'objets CORBA identiques suivant un modèle d'exécution SPMD. Contrairement à l'approche précédente, les objets parallèles CORBA respectent le modèle de programmation SPMD et utilisent des flux de communication parallèles entre les codes couplés, ce qui permet d'agréger la bande-passante et donc d'obtenir de meilleure performance. Dans PARDIS [119, 120], un nouveau type de données nommé *distributed sequence* généralise les séquences CORBA au cas distribué. PARDIS utilise alors un compilateur IDL spécifique qui génère des souches et des squelettes permettant de gérer la distribution des données de manière transparente, mais non compatible avec les objets CORBA classiques. Par ailleurs, PARDIS introduit de nouvelles fonctionnalités permettant de réaliser des appels non bloquants. Tout comme PARDIS, PaCO [187] modifie le langage IDL pour prendre en compte la distribution des données. Les distributions supportées dans PaCO sont celles définies dans le langage HPF. PaCO nécessite de modifier l'ORB pour prendre en charge le parallélisme, ce qui limite la portabilité d'une telle solution. Data Parallel CORBA [34] est une spécification de l'OMG qui impose de modifier l'ORB, avec notamment l'introduction d'un nouveau gestionnaire d'objet, le PPA (Parallel Part Adapter), et l'introduction d'un *proxy* pour prolonger la compatibilité avec les ORBs classiques. Au contraire de PARDIS et de PaCO, Data Parallel CORBA ne modifie pas le langage IDL. Il ne gère pas à proprement parler la distribution des données, mais il repose sur une interface de bas-niveau permettant au client de décrire explicitement le contenu des messages qu'il faut transmettre au serveur parallèle. Ces trois solutions imposent de modifier la norme CORBA pour supporter des codes SPMD. Plus récemment, PaCO++ [67, 180] a proposé une définition portable des objets parallèles dans CORBA, qui ne modifie ni l'ORB, ni le langage IDL (Fig. 2.11). Afin d'être portable, les objets parallèles sont programmés comme une surcouche d'un ORB standard. Afin de prendre en charge la distribution des données, l'interface IDL est associée à un fichier de description XML, à partir duquel les souches et les squelettes parallèles sont générés. PaCO++ repose sur l'utilisation de bibliothèques externes pour la gestion de la redistribution. Il propose également un modèle pour la gestion des exceptions en parallèle. Par ailleurs, il permet de faire coopérer des clients séquentiels standards avec des serveurs parallèles de manière transparente.

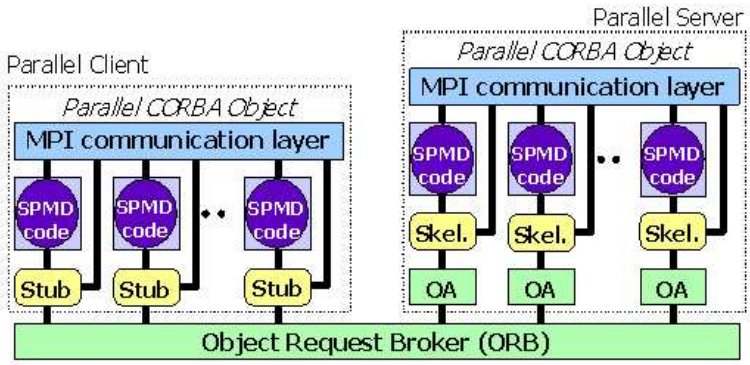


FIG. 2.11 – Les objets parallèles CORBA dans PaCO++ [180].

2.2.6 Les composants logiciels

Même si la notion de composant n'est pas neuve en soi (cf. composants électroniques, composants automobiles), elle n'a pris son essor en informatique que très récemment. L'objectif principal des composants en informatique est de réduire la complexité des logiciels tout en améliorant leur qualité. Les composants sont généralement réputés pour offrir une meilleure structuration des codes et une meilleur réutilisabilité que le modèle de programmation objet [174]. De plus le couplage de codes s'exprime naturellement dans ce modèle grâce à l'opération de *composition*. Une définition des composants logiciels communément acceptée est celle donnée par Szyperski [197] :

> « *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independantly and is subject to composition by third party.* »

Les composants sont comparables à des « boîtes noires » possédant une interface avec le monde extérieur clairement spécifiée (dans un méta-langage comme l'IDL). La propriété fondamentale du composant est *la composition*, c'est-à-dire l'aptitude d'un composant à se connecter à un autre composant pour construire une application plus complexe. La définition précédente souligne l'idée qu'un composant doit expliciter ses dépendances, c'est-à-dire qu'il doit spécifier les composants dont il aura besoin pour fonctionner. Une application est alors vue comme un assemblage de plusieurs composants, l'idée étant de favoriser la réutilisation de composants préfabriqués et de développer uniquement les composants qui sont vraiment spécifiques aux besoins. La définition souligne en particulier le fait que la composition peut être réalisée par une tierce personne, différente de celle qui a développé le composant et ne possédant pas de compétences particulières en programmation. Pour finir, notons que les modèles de composants (du moins les plus complets) ne se contentent pas uniquement de définir un nouveau modèle de programmation, ils s'efforcent de prendre en charge tout le cycle de vie d'un composant : depuis sa conception jusqu'à son déploiement dans des serveurs distants.

Il existe aujourd'hui plusieurs modèles de composants logiciels qui ont démontré leurs possibilités tels que le modèle COM/DCOM [137] de Microsoft et plus récemment .NET [138], le modèle EJB (Enterprise Java Beans) [141] de SUN, le modèle composant de CORBA (CCM) [162] spécifié par l'OMG ou encore le modèle CCA [39] du DOE (U.S. Department Of Energy). Parmi ces modèles, seuls les modèles CCM et CCA semblent adaptés à la problématique du calcul scientifique, les modèles de Microsoft et de Sun étant davantage orienté *business* et respectivement limités aux mondes Windows et Java.

CCM

CCM (CORBA Component Model) est le modèle de composant de CORBA correspondant à la spécification 3.0 de l'OMG (datant de juin 2002) [162]. CCM est en fait le prolongement de CORBA 2.0 (concernant les objets) dont il conserve les « bonnes » propriétés : indépendance du langage et du fournisseur, environnement

distribué et hétérogène. Un composant CORBA est constitué d'un ensemble de *ports* représentant les interfaces avec le monde extérieur et d'attributs désignant les propriétés configurables du composant (Fig. 2.12). On distingue quatre types d'interfaces dans un composant CORBA : les facettes, les réceptacles, les sources et les puits d'évènements. Les facettes et les réceptacles décrivent respectivement les services fournies (*provides*) et requis (*uses*) par un composant. Une facette est comparable à l'interface d'un objet CORBA classique dont l'implantation est hébergée par le composant. Elle représente un point de connexion potentiel pour les autres composants qui utilisent cette interface via un réceptacle. La communication entre les deux composants s'effectue via des invocations de méthodes CORBA classiques. Les sources et les puits d'évènements ont été introduits pour permettre un mode de communication asynchrone entre les composants basé sur la transmission d'évènements avec des messages structurés décrits en IDL (*eventtype*). La spécification de CCM est très complète. Elle étend d'une part le langage OMG IDL avec de nouveaux mots-clés pour permettre la description les composants de manière abstraite (IDL3). Elle introduit d'autre part un *framework* d'implantation (CIF) basé sur un nouveau langage de description, le CIDL (Component Implementation Definition Language). Ce *framework* fournit un modèle de programmation des composants prenant en charge automatiquement les services non-fonctionnels (persistance, transaction, sécurité, *etc*.). Un tel modèle a pour but de faciliter la tâche du développeur qui a uniquement besoin de prendre en charge la spécification du composant (IDL+CIDL) et son implantation (« la partie métier ») sans avoir à se soucier de détails techniques qui sont pris en charge de manière (plus ou moins) transparente par le *framework*. Une chaîne d'outils standards permet d'assembler les composants entre eux pour construire une application plus complexe, d'emballer les composants sous forme d'archive (*packaging*) et de les déployer dans des serveurs standards distribués et hétérogènes. Il existe plusieurs implantations du modèle de composant CCM, dont OpenCCM [160] en Java et MicoCCM [36] en C++. Notons par ailleurs que les composants CCM sont interopérables avec les objets CORBA (« *component-unaware* ») et les Enterprise JavaBeans (EJB) de Sun. Pour conclure, soulignons que ce modèle ne permet pas de supporter des codes parallèles. Le projet GridCCM [179] se propose de définir une extension parallèle du modèle de composant CCM en se basant sur PaCO++ [180], une version parallèle des objets CORBA.

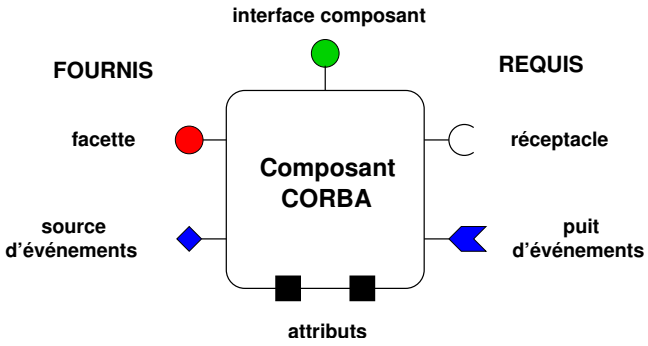


FIG. 2.12 – Un composant CORBA.

CCA

Sous l'impulsion du DOE (U.S. Department Of Energy), un ensemble de laboratoires et d'universités américaines participent dans le CCA Forum [4] à la conception du modèle de composant CCA (Common Component Architecture). L'un des premiers objectifs du CCA Forum est de définir un modèle de composants orienté HPC afin de permettre la mise en commun des codes de calcul scientifique du DOE. Le modèle de composant CCA peut être vu comme une simplification des composants CORBA utilisant uniquement deux types de ports : les facettes et les réceptacles [39]. Comme CORBA, CCA met l'accent sur l'interopérabilité et le support multi-langages. En particulier, il utilise une variante du langage IDL, baptisée SIDL (Scientific Interface Definition Language), intégrant les nombres complexes et les tableaux multi-dimensionnels à la Fortran. Toutefois à la différence de CCM, les composants CCA peuvent être parallèles (Single Component Multiple Data), comme par exemple dans les implantations CCAFFEINE [37] et SCIRun2 [65]. Comme le montre la figure 2.13, deux composants parallèles connectés – disons A et B – peuvent dialoguer localement au sein d'un même processus via les ports (relation *uses/provides*). La communication interne au composant parallèle n'est pas spécifiée,

c'est-à-dire que le développeur est *a priori* libre d'utiliser le mécanisme de son choix (i.e. MPI, PVM, *etc*). La notion de *port collectif* a été introduite pour prendre en charge les invocations entre des composants parallèles distants, ce qui nécessite de spécifier le *mapping* des données en mémoire et éventuellement de redistribuer les données. Pour prendre en charge le problème de la redistribution des données, CCA travaille à la spécification d'un composant de redistribution baptisée M×N [5], intégrant principalement les technologies PAWS [118] et CUMULVS [122]. CCA distingue deux types de connections entre les ports : la *connexion directe* pour des composants fortement couplés (e.g. préconditionneur/solveur) et la *connexion réseau* pour des composants distribués plus faiblement couplés (e.g. simulation/visualisation). Les connections directes permettent à des composants situés dans le même espace mémoire de réaliser des interactions rapides, avec un surcoût réduit lors des invocations de méthodes (de l'ordre de 2 à 3 appels de fonction F77). Le mécanisme de communication utilisé pour les connections réseaux est à la discrétion du *framework*. Par exemple, il repose dans l'implantation XCAT [95] sur les Web Services [211]. Notons pour conclure que les implantations existantes sont encore incomplètes et ne réalisent qu'une partie de toute l'architecture CCA.

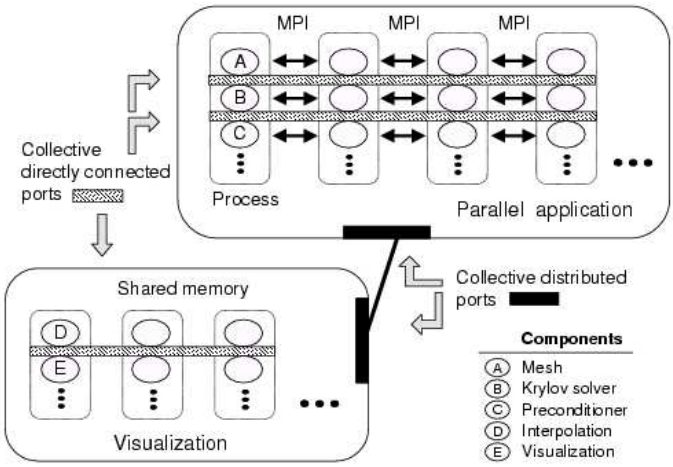


FIG. 2.13 – Exemple d'utilisation des composants parallèles dans CCA [39].

2.3 La visualisation scientifique

Dans la plupart des domaines du calcul scientifique, la *visualisation scientifique* joue un rôle important. En effet, l'analyse des données produites par les simulations numériques n'est généralement pas envisageable sans l'aide d'un logiciel de visualisation. Nous définissons la visualisation scientifique comme le processus permettant de convertir les données numériques en une représentation graphique facile à interpréter [72]. Cette section présente quelques notions de base concernant la visualisation scientifique, le rendu parallèle et la réalité virtuelle

2.3.1 Système de visualisation

Les systèmes de visualisation sont des programmes informatiques servant à lire, traiter et afficher des données sous forme d'images. On parle de système de visualisation scientifique lorsque la palette de fonctionnalités offertes est suffisamment large pour lire et traiter les données issues des simulations numériques. Parmi ces systèmes, il convient de citer des produits commerciaux célèbres comme IRIS Explorer [85], AVS [2], IBM Data Explorer (OpenDX) [112], Ensight [10] ou encore des produits « libres » comme Paraview [25] et VTK (Visualisation ToolKit) [33, 191]. Un système de visualisation repose essentiellement sur deux pipelines : *le pipeline de visualisation*, dont le rôle est de convertir les données numériques en primitives graphiques (points, lignes, triangles) et *le pipeline graphique*, dont le rôle consiste à afficher les primitives graphiques à l'écran sous forme

d'images (Fig 2.14). En d'autres termes, le pipeline de visualisation construit la représentation géométrique qui est rendu par le pipeline graphique.

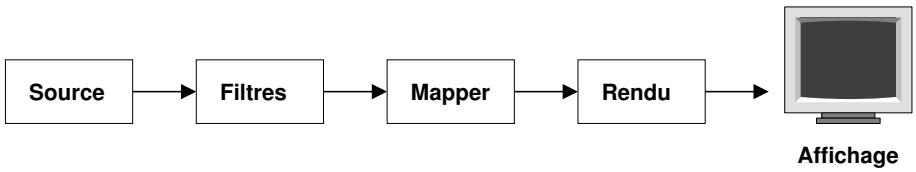


FIG. 2.14 – Pipeline de visualisation et rendu graphique.

Le pipeline de visualisation

Le modèle de visualisation, certainement le plus répandu, utilise le paradigme *data-flow*. Dans ce paradigme, une application de visualisation est décomposée en unités d'exécution, appelées *modules*. Chaque module possède un ensemble d'entrées et de sorties, appelés *ports*. Le port de sortie d'un premier module peut être connecté au port d'entrée d'un second, de telle sorte qu'à l'exécution du premier module la donnée produite en sortie est transmise à l'entrée du second module qui à son tour s'exécute, et ainsi de suite. L'ensemble des interconnexions entre modules forme un graphe *data-flow*, encore appelé *réseau*. On distingue deux modèles d'exécution d'un tel graphe : le modèle « par évènement » (*event-driven*) comme dans AVS et le modèle « sur demande » (*demand-driven*) comme dans VTK. Dans le premier modèle, l'exécution s'effectue en réponse à un évènement survenant à l'entrée d'un module comme la modification des données. Dans le second modèle, l'exécution du graphe est contrôlée par la demande de données et ne s'applique uniquement qu'à la portion du graphe utile à la génération des données requises. Dans ce modèle réputé plus économique, un ordre de mise-à-jour parcourt initialement le graphe pour rechercher la portion du graphe à exécuter. Dans la plupart des systèmes de visualisation et dans la littérature [200, 102], on distingue trois catégories de modules : les sources, les filtres et les *mappers*, logiquement connectés selon le pipeline représenté sur la figure 2.14.

Source – Les sources sont les modules à l'entrée du pipeline chargés d'importer les données dans le système de visualisation. Typiquement, les sources chargent les données à partir de fichiers utilisant un certain format de stockage (e.g. UCD pour les maillages non structurés dans AVS). Il existe également des sources, dites procédurales, capable de générer des objets géométriques simples (e.g. cube, sphère, *etc*.).

Filtre – Les filtres traitent les données brutes issues des modules sources, les manipulent et les modifient afin d'extraire l'information et d'enrichir cette information. Les filtres implantent généralement des opérations algorithmiques plus ou moins complexes, qu'il faut généralement combiner pour obtenir le résultat attendu. Il existe un multitude de filtres différents, comme par exemple les plans de coupe, les iso-surfaces, les seuillages, les triangulations, les lignes de courant, *etc*. Nous étudions brièvement les techniques de visualisation implantées dans les filtres dans la section suivante.

Mapper – Le *mapper* convertit les données issues des filtres en primitives géométriques (points, lignes, polygones), les combinent et construit un objet graphique 3D. Le mapper sert d'interface entre le pipeline de visualisation et le modèle graphique 3D. Finalement, les primitives géométriques issues des *mappers* sont traitées par un nouveau pipeline, le pipeline de rendu, pour produire une image, puis afficher cette image dans une fenêtre.

Techniques de visualisation

On trouve dans la littérature différentes classifications des algorithmes de visualisation. Une approche simple consiste à classer ces algorithmes selon le type de données qu'ils traitent. Cela permet de séparer les techniques de visualisation en trois grandes catégories : les techniques s'appliquant à des données scalaires (1D), celles s'appliquant à des données vectorielles (2D ou 3D) ou celles s'appliquant à des données tensorielles. On peut encore raffiner cette classification en précisant la dimension du domaine géométrique contenant ces données. De manière générale, on note un tel ensemble de données E_n^d avec n la dimension du domaine et d la taille du vecteur ou du tenseur. Par exemple, E_1^1 représente une série de scalaires, E_2^1 une image ou une grille

2D, E_3^1 une image 3D du type IRM, E_3^3 un champs de vecteurs dans l'espace (ou flux) et $E_3^{3\times3}$ un champs de tenseurs.

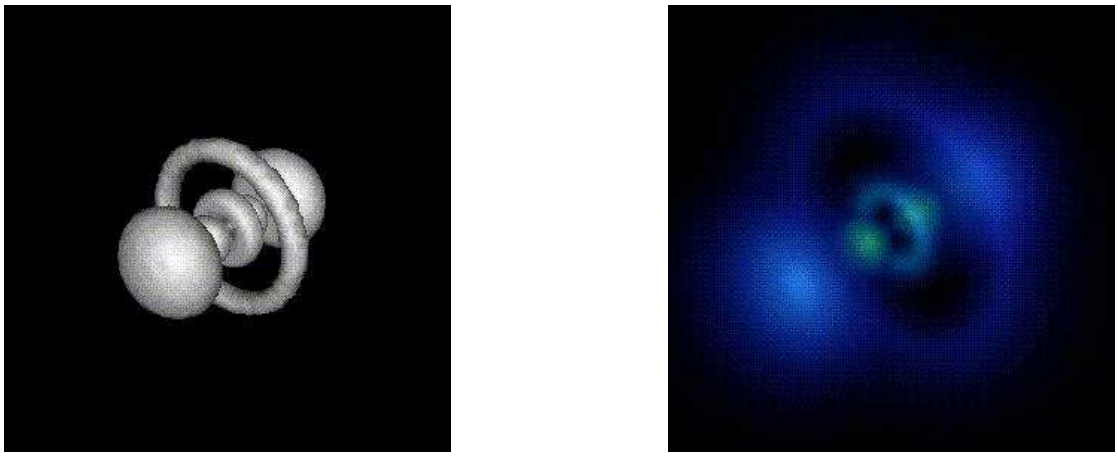


FIG. 2.15 – Visualisation d'un atome d'hydrogène : iso-surface à gauche et rendu volumique par *ray-tracing* à droite.

Parmi les techniques scalaires, la visualisation des données de type E_1^1 ou E_2^1 est généralement très simple : histogramme, affichage d'une image (échelle de couleur), *etc*. Pour visualiser des données scalaires volumiques (E_3^1), il existe plusieurs techniques décrites dans [52, 74]. Parmi ces techniques (Fig 2.15), on trouve les plans de coupe (*slicer*), le calcul d'iso-surface (algorithme du *marching-cube* [131]) ou encore les techniques de rendu volumique basée sur la visualisation en transparence de données denses (*ray-tracing* [109]). Les techniques vectorielles permettent de visualiser des champs de vecteurs 2D ou 3D. Ces techniques reposent classiquement sur l'utilisation de *glyphs* orientés, des objets 2D ou 3D (e.g. flèches, lignes) servant à représenter les vecteurs dans l'espace (*hedgehogs*). Une autre approche pour visualiser les données vectorielles consiste à dessiner les lignes de champs (*streamlines*), c'est-à-dire les courbes tangentes aux champs de vecteurs. Les données tensorielles (matrices 3×3) représentent généralement un déplacement ou une pression exercée sur un matériau 3D. Dans ce cas précis, les vecteurs propres et les valeurs propres de la matrice ont une signification physique particulière. Une façon de représenter ce type de données consiste à associer à chaque tenseur dans l'espace à une ellipsoïde (*glyph*) orientée selon les directions des vecteurs propres. D'autres techniques de visualisation échappent encore à cette classification, notamment les algorithmes réalisant des transformations géométriques et/ou topologiques comme par exemple les algorithmes de simplifications géométriques (cf. section 2.3.2), ainsi que toutes les techniques hybrides combinant plusieurs des techniques précédentes.

Le pipeline de rendu 3D

En infographie, le rendu d'une scène 3D repose sur un modèle graphique, définissant les éléments clés de la scène : objets, sources de lumière, caméras. Une scène se compose d'un ensemble d'objets 3D le plus souvent organisés en arbre comme dans OpenInventor [24] ou VRML (Virtual Reality Modeling Language) [31]. Les objets 3D sont généralement définis comme un ensemble de primitives géométriques auxquelles on associe une apparence (couleur, matériau, texture). Les sources de lumière (ponctuelle, directionnelle ou spot) éclairent les objets dans la scène, avec une intensité et une couleur propre. La caméra définit le point de vue selon lequel un œil fictif regarderait la scène. Le champs de vision de la caméra est modélisé par un cône tronqué aux extrémités par deux plans permettant d'éliminer les objets trop près ou trop loin (*clipping*). Pour calculer une image de la scène, les objets présents dans le champs de vision sont projetés sur un plan particulier, le plan de projection où l'image est formée.

Grâce à la connaissance de tous ces éléments, il est possible au moteur de rendu (*render engine*) de calculer effectivement une image de la scène ou *frame buffer*. La première étape du pipeline de rendu consiste à appliquer les transformations géométriques (translation, rotation, mise à l'échelle) aux objets pour calculer leurs coordon-

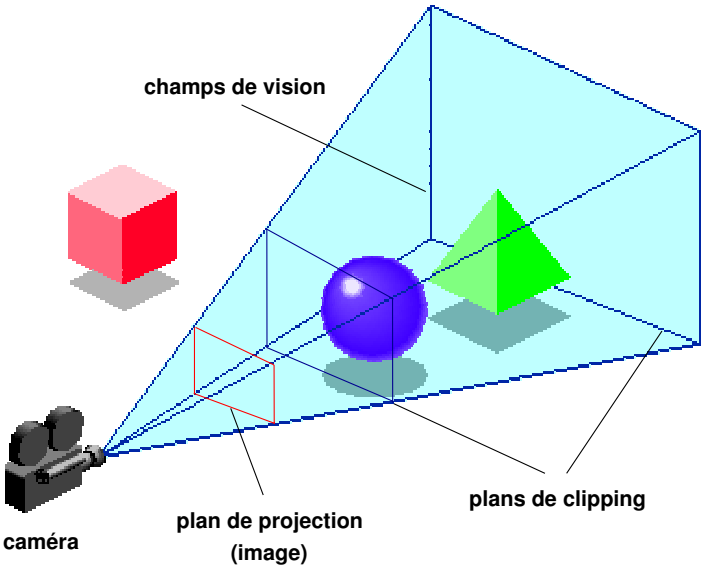


FIG. 2.16 – Objets 3D, caméra et champs de vision.

nées dans le repère canonique de la scène. La deuxième étape consiste à calculer l'éclairement de la scène, ce qui revient à déterminer la couleur des sommets des polygones (modèle d'illumination). L'étape suivante transforme les coordonnées des polygones dans le repère de la caméra. A partir de là, il est possible d'effectuer le clipping et de calculer efficacement la projection (conversion des polygones en coordonnées 2D). Finalement, la dernière étape (*rasterization*) convertit les objets en pixels. La couleur perçue d'un objet dans la scène dépend de l'interaction savante entre la lumière éclairant l'objet, la couleur propre de l'objet, l'incidence du rayon lumineux et la position de la caméra. Il existe plusieurs modèles, appelés modèles d'illumination, qui s'inspirent des lois de l'optique géométrique et cherchent à reproduire efficacement les principaux phénomènes lumineux (réflexion, réfraction, diffusion). On distingue l'approche locale, qui consiste à calculer l'illumination polygone par polygone, de l'approche globale qui résout l'éclairement de la scène globalement (e.g. radiosité [93]). Par ailleurs, comme le rendu d'un polygone pixel par pixel est beaucoup trop coûteux, on préfère en général calculer l'illumination uniquement sur les sommets. Puis à l'aide d'une interpolation, soit des couleurs (Gouraud [94]), soit des normales (Phong [175]), on déduit le rendu de la surface toute entière. Une autre difficulté, lorsque l'on calcule le rendu d'une scène, consiste à déterminer les facettes visibles parmi l'ensemble des polygones de la scène (algorithme de visibilité). Dans l'algorithme du Z-buffer [58], on tient à jour une image en profondeur de la scène appelée Z-buffer. Pour chaque polygone, on calcule sa projection et on effectue l'éclairement uniquement si les pixels projetés ont une profondeur inférieure à la valeur courante du Z-buffer. Dans l'algorithme de lancer de rayons [109] (*ray-tracing*), on procède de manière inverse. Un rayon lumineux est lancé dans la scène depuis l'œil vers chaque pixel de l'image. Il faut ensuite déterminer la première surface intersectée par le rayon, celle donc qui sera visible et dont le calcul de l'éclairement donnera la couleur du pixel.

2.3.2 La visualisation des grands ensembles de données

La visualisation scientifique doit permettre une visualisation aussi précise que possible tout en conservant une bonne interactivité. La précision est nécessaire au scientifique dès lors qu'il observe des informations quantitatives et souhaite comparer ces données. L'interactivité, quant à elle, permet de comprendre la structure générale des données en manipulant leur représentation et permet de rechercher les zones d'intérêt. Ces deux souhaits deviennent contradictoires dès lors qu'il s'agit de traiter de grands volumes de données, pouvant dépasser la mémoire d'un ordinateur. En effet, la capacité globale en visualisation des ordinateurs ne peut suivre la croissance exponentielle des données générées par les calculateurs hautes-performances, ce qui rend nécessaire l'utilisation du parallélisme. Parmi les travaux existants autour de la visualisation des grands ensembles de données, la plupart sont consacrés à la problématique du rendu en parallèle, et plus particulièrement au rendu photo-réaliste (lancer de rayons et radiosité) [184], au rendu des polygones [62] et au rendu vo-

lumique [214]. D'autres travaux se sont penchés sur la parallélisation des techniques de visualisation, comme le calcul des iso-surfaces [105], ou les techniques de simplifications géométriques servant à réduire le niveau de détails [107, 108, 110]. A ces travaux, il convient d'ajouter les travaux plus récents autour de VTK [35, 128, 147, 148], qui aujourd'hui sont une référence dans le domaine de la visualisation scientifique parallèle. Basiquement, on distingue trois types de parallélisme dans VTK : le parallélisme de tâches, le parallélisme de pipeline et le parallélisme de données. Le parallélisme de tâches consiste à exécuter des branches indépendantes du graphe *data-flow* dans des processus séparés. Le parallélisme de pipeline consiste à exécuter en parallèle une série de modules connectés sur des données indépendantes. Cette approche est particulièrement utile pour traiter des données variant dans le temps. Le parallélisme de données consiste à exécuter un module en parallèle sur des données distribuées. Cette approche, contrairement aux précédentes est *scalable*, et permet de traiter de grands volumes de données. Contrairement aux autres systèmes de visualisation qui proposent quelques solutions parallèles en mémoire partagée (multi-threads), VTK & Paraview incorporent une solution pour des architectures parallèles à mémoire distribuée (basée sur MPI). Dans VTK, la distribution du *pipeline* est rendue transparente grâce notamment à l'utilisation du paradigme RMI et au mécanisme de sérialisation des données. Nous allons maintenant présenter quelques techniques fréquemment utilisées permettant de traiter de grands ensembles de données ainsi que de faire du rendu parallèle.

Niveaux de détails

Une première approche permettant de traiter de grands ensembles de données consiste à simplifier la géométrie des objets de la scène, ce qui réduit la précision du rendu mais améliore en contre-partie l'interactivité. La technique la plus fréquemment utilisée pour calculer rapidement le rendu d'un grand ensemble de données est le sous-échantillonage. Cette technique consiste à réduire la taille des données en sélectionnant un sous-ensemble des données initiales, par exemple en éliminant les points selon une certaine période. Cette technique s'applique immédiatement pour des structures régulières comme les images, mais nécessite une retriangulation dans le cas des maillages non structurés. La décimation est une technique de réduction des polygones [107, 108, 110], qui élimine uniquement les nœuds du maillage vérifiant une certaine condition de co-linéarité ou de co-planarité, et qui remplace « le trou » apparu par une nouvelle triangulation. Contrairement à la technique d'échantillonage présentée juste avant, le choix des points à supprimer se base sur *un critère de décimation*, c'est-à-dire sur une mesure de l'erreur locale introduite par l'approximation du maillage (Fig. 2.17).

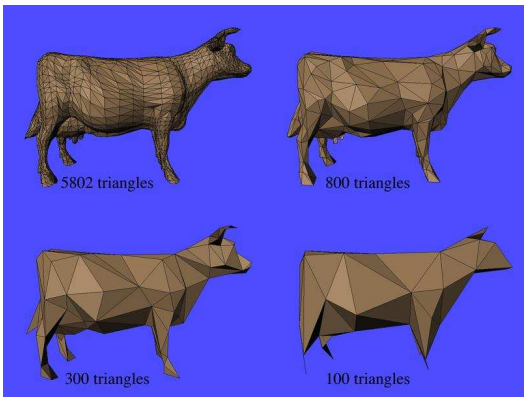


FIG. 2.17 – Réduction du niveau de détails par un algorithme de décimation.

Le streaming des données

Le *streaming* des données est une technique permettant de traiter des données dont la taille dépasse la capacité mémoire d'une machine sans utiliser le « swap ». Le principe consiste à diviser les données en régions de plus petites tailles pouvant être traitées indépendamment, région par région (algorithme *divide & conquer*). Les primitives géométriques de chaque région sont alors accumulées avant d'être transformées en image (Fig. 2.18). L'avantage majeur du streaming est qu'il réduit la taille maximale des données traversant le pipeline, et permet donc de réaliser des traitements intermédiaires consommant beaucoup de mémoire. A notre connaissance, VTK

est le seul système de visualisation utilisant cette technique [127]. Par ailleurs, VTK intègre de manière transparente dans son pipeline une version multi-threads du streaming, ce qui lui permet d'accélérer le traitement de données régulières (ou non) tout en réduisant la consommation mémoire.

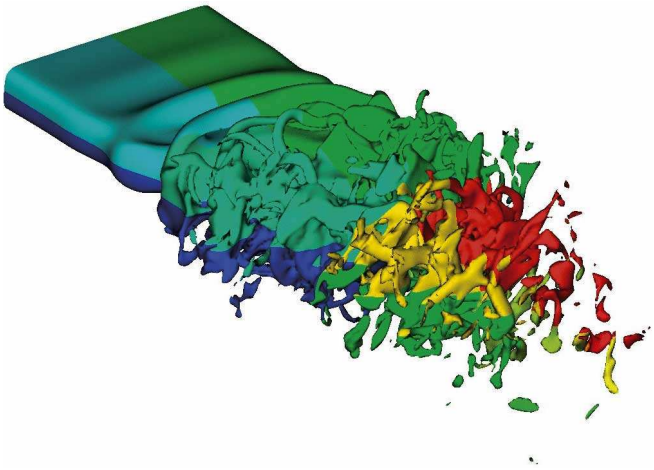


FIG. 2.18 – Exemple d'utilisation du *streaming* dans VTK [80]. Calcul d'une iso-surface basé sur un découpage en 16 régions d'un volume (une couleur par région).

Le rendu parallèle

Le parallélisme de données consiste à dupliquer le pipeline de visualisation sur un ensemble de processus. Comme dans le cas du *streaming*, les données sont découpées en régions indépendantes et chacune de ces régions est traitée en parallèle. Toutefois, si tout ou partie du pipeline de visualisation s'effectue en parallèle, le rendu et l'affichage sont généralement réalisés sur un seul écran. L'utilisation d'une seule carte graphique 3D pour afficher une scène complexe avec des résolutions soutenues provoque nécessairement un goulot d'étranglement. Pour surmonter cette difficulté, certaines techniques permettent de prolonger le parallélisme jusqu'au rendu, et d'afficher la scène non plus sur un seul écran mais sur un mur d'images (Fig. 2.19). Dans le cas du *rendu centralisé*, le nœud responsable de l'affichage collecte l'ensemble des primitives géométriques provenant des différents pipelines et effectue le rendu de la scène séquentiellement. Une autre approche consiste à exécuter les différents pipelines en parallèle de bout en bout, c'est-à-dire de la source jusqu'au rendu. On parle alors de *rendu parallèle*. Notons que le nombre de processus utilisés pour effectuer le rendu en parallèle peut être inférieur à celui utiliser pour traiter les données. Il existent plusieurs techniques de rendu parallèle comme le *sort-first*, le *sort-middle* ou le *sort-last* [145], que nous allons maintenant présenter.

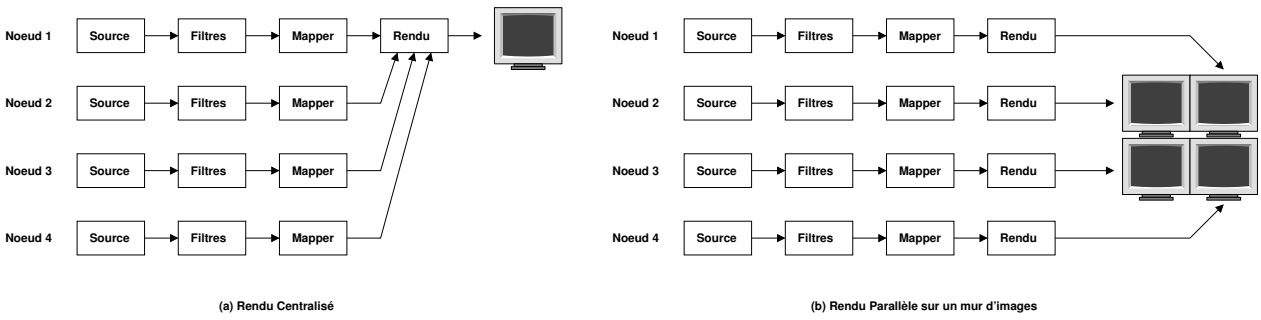


FIG. 2.19 – Pipeline de visualisation parallèle : rendu centralisé et parallèle.

Dans la technique du *sort-first* (Fig. 2.20) ou du *sort-middle*, l'image finale est décomposée spatialement, et chaque processus est responsable d'une portion de l'image. Initialement, les polygones sont distribués

arbitraitement sur les nœuds. Un tri préalable des polygones suivi d'une étape de communication est alors nécessaire pour affecter à chaque processus les polygones qui seront projetés dans sa portion de l'image. Dans le cas du *sort-first*, ce sont les primitives graphiques 3D « brutes » qui sont redistribuées, tandis que dans le cas du *sort-middle*, ce sont les primitives pré-transformées en coordonnées 2D de l'écran. Les polygones à cheval sur une frontière sont dupliqués lors de la phase de communication. Cette technique permet de réaliser naturellement des murs d'images. Notons que l'étape de communication peut être supprimée en travaillant en mémoire partagée ou en répliquant la scène sur chaque nœud. L'efficacité du tri peut être considérablement augmentée en regroupant les primitives gaphiques dans des volumes englobants. Parmi les bibliothèques permettant d'effectuer du rendu parallèle en *sort-first*, citons en particulier Chromium [111] (anciennement appelé WireGL). Chromium utilise une approche flexible permettant de rediriger de manière transparente les primitives OpenGL vers des serveurs de rendu distants, bénéficiant ainsi d'une accélération 3D.

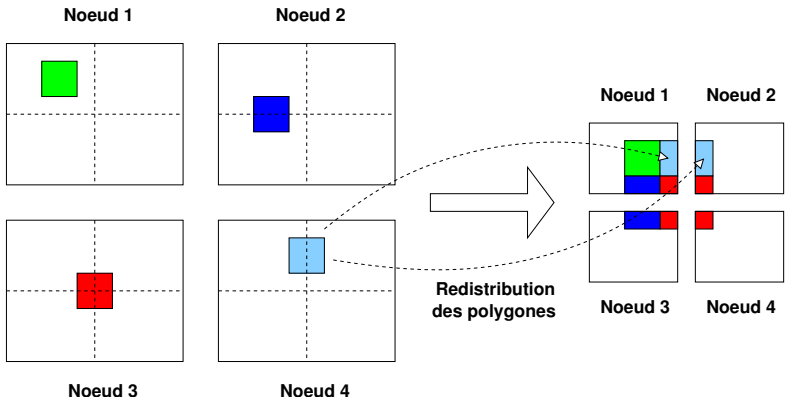


FIG. 2.20 – Rendu parallèle en *sort-first* et affichage sur un mur d'images.

Dans la technique du *sort-last* (Fig. 2.21), chaque pipeline construit une image partielle de la scène avec la même résolution que l'image finale. Ces images sont ensuite combinées selon un arbre binaire, simplement en comparant les Z-buffers partiels (test en profondeur des pixels). L'avantage de cette méthode, est qu'il n'est pas besoin de trier préalablement les polygones qui initialement sont distribués arbitrairement sur les nœuds de manière à équilibrer la charge. Par ailleurs, cette méthode est très *scalable* pour principalement deux raisons : (1) le rendu des polygones peut bénéficier de l'accélération matérielle des cartes graphiques 3D ; (2) il suffit de communiquer des images 2D d'un ordre de grandeur généralement inférieur aux données en entrées (charge réseau constante). Par ailleurs, les images échangées peuvent être avantageusement compressées pour réduire le volume des communications. Parmi les bibliothèques graphiques permettant d'effectuer du rendu parallèle en *sort-last*, on peut citer Chromium [111] et VTK [128]. Par ailleurs, la bibliothèque ICE-T (Image Composition Engine for Tiles) [147, 148] est spécialement dédiée au rendu parallèle de type *sort-last* sur des murs d'images de très grande taille. Notons que cette bibliothèque a été récemment intégrée dans Paraview/VTK.

Pour améliorer l'interactivité, Paraview [128] combine le parallélisme de données en *sort-last* à des techniques de décimation polygonale des objets et de sous-échantillonage des images combinées. Cela permet d'accélérer le rendu pendant l'interaction afin de manipuler efficacement de gros objets polygonaux, tout en gardant la possibilité de visualiser le maximum de détails à la fin de l'interaction. En effet, tout déplacement de la caméra ou d'un objet dans la scène se traduit systématiquement par un nouveau calcul du rendu, impliquant des communications inter-processus lorsque le rendu s'effectue en parallèle. Si le goulot d'étranglement se situe au niveau des communications (image haute-résolution en *sort-last*), la technique de sous-échantillonage va permettre de réduire la taille des images combinées et donc d'accélérer le transfert. Si le goulot d'étranglement se situe maintenant au niveau des traitements de visualisation, la décimation géométrique permettra de réduire les objets géométriques manipulés et donc de diminuer efficacement les temps de calculs.

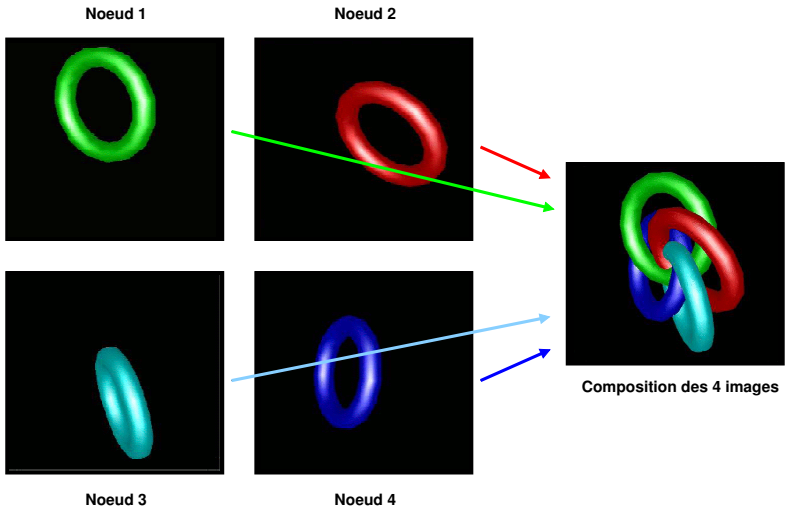


FIG. 2.21 – Rendu parallèle en *sort-last*.

Chapitre 3

État de l'art sur le pilotage des simulations numériques

Sommaire

Nous avons présenté au chapitre précédent un ensemble de notions autour de la simulation numérique, du couplage de codes et de la visualisation scientifique. Dans ce chapitre, nous présentons l'état de l'art sur le pilotage des simulations numériques. Nous commençons par introduire quelques notions générales sur le pilotage. Puis dans la deuxième section, nous présentons les caractéristiques générales des environnements de pilotage. Nous complétons cette étude en examinant plus précisément quelques environnements représentatifs de l'état de l'art. Enfin, nous concluons ce chapitre en observant les points forts et les limites des solutions actuelles afin de positionner nos travaux.

3.1 Introduction

3.1.1 Généralités

En 1987, un rapport influent de l'U.S. National Science Foundation on Scientific Visualization [135] publiait ce texte :

> « *Scientists not only want to analyse data that result from their super-computations ; they also wants to interpret what is happening to the data during super-computations. Researchers wants to steer calculations in close to real-time ; they want to be able to change parameters, resolution or presentation, and see the effects. They want to drive the scientific discovery process ; they want to interact with their data.* »

Bien que ce rapport fut publié il y a près de quinze années, il reconnaissait l'importance dans le processus de découverte scientifique d'un nouveau domaine de recherche, *la simulation interactive* ou *le pilotage des simulations numériques* (en anglais, *computational steering*). Le pilotage des simulations numériques est une discipline récente qui a pour but d'améliorer le processus de calcul et d'analyse scientifique en le rendant plus interactif. Dans cette approche, l'utilisateur n'attend plus passivement les résultats de la simulation. Au contraire, il visualise « en temps-réel » l'évolution des calculs, et peut interagir à tout moment en modifiant certains paramètres du modèle à la volée et plus généralement « en pilotant » le déroulement des calculs.

On trouve dans la littérature plusieurs définitions de la simulation interactive. *Mulder et al.* [153] définissent le pilotage comme le contrôle interactif d'un processus de calcul pendant son exécution. Pour *Vetter et al.* [207], c'est le contrôle « en temps-réel » d'une application et de ses ressources, dans le but d'expérimenter les paramètres de l'application ou d'améliorer ses performances. Pour *Parker* [168], la simulation interactive doit permettre d'extraire efficacement des informations scientifiques et de modifier les paramètres ou les données de la simulation de manière cohérente. Ces définitions soulignent plusieurs caractéristiques fondamentales de la simulation interactive. Tout d'abord, l'interaction se fait « en temps-réel », c'est-à-dire pendant l'exécution de la simulation. Ensuite, l'interaction doit se réaliser de manière cohérente, c'est-à-dire en conservant la sémantique du modèle physique sous-jacent. Finalement, l'interaction doit être efficace dans le sens où elle doit s'effectuer le plus rapidement possible et/ou introduire le moins de perturbation possible.

FIG. 3.1 – Pilotage de la simulation NAMD avec le système IMD [195] : introduction d'un ion sodium dans le canal de la gramicidine A.

Au cours des quinze dernières années, beaucoup de solutions de pilotage sont apparues. Parmi ces travaux, on distingue des solutions de pilotage spécifiques à une application ou à un domaine et des solutions de pilotage plus génériques et réutilisables, appelées *environnement de pilotage*. Parmi les tous premiers travaux fréquemment cités, *Marshall et al.* ont proposé une solution spécifique pour visualiser et piloter un modèle de turbulence 3D du lac Erie [134]. Leurs expériences confirmaient les bénéfices de cette approche et les besoins d'outils modulaires et réutilisables. Plus récemment dans le domaine de la biologie moléculaire, le code de dynamique moléculaire NAMD [199] peut être connecté à l'outil de visualisation VMD [30] via le système d'interaction IMD (Interactive Molecular Dynamics) [195]. A chaque pas de temps, IMD communique la position des atomes à l'outil de visualisation. En retour, l'utilisateur peut manipuler les molécules à l'aide d'une souris ou d'un périphérique haptique à retour d'effort qu'il utilise pour appliquer des forces aux atomes de la simulation NAMD (Fig. 3.1). L'avantage d'une approche spécifique est évident : elle permet de définir une solution de pilotage sur mesure parfaitement adaptée aux besoins des utilisateurs et généralement simple à mettre en

œuvre. Toutefois, la mise en place du pilotage nécessite un investissement important en développement pour une solution qui n'est pas *a priori* réutilisable dans un autre contexte. Par ailleurs, cette solution s'avère peu flexible pour les utilisateurs car il est souvent difficile voire impossible d'étendre les fonctionnalités de pilotage. Par exemple, il n'est pas possible avec IMD de définir de nouvelles interactions ou d'extraire d'autres données que celles initialement prévues par ses développeurs.

3.1.2 Environnement de pilotage

Un environnement de pilotage est une plate-forme logicielle qui permet de coupler une simulation numérique avec une interface utilisateur (UI), à partir de laquelle un utilisateur (le pilote) a la possibilité d'interagir avec la simulation. Comme le montre la figure 3.2, un environnement de pilotage se divise en trois composantes logiques : *la simulation numérique*, *l'interface utilisateur* et *le système de communication* reliant ces deux composantes.

Conceptuellement, le pilotage d'une simulation repose sur une abstraction en terme *d'objets de pilotage*. Ces objets encapsulent les informations essentielles décrivant la simulation et les interactions possibles. Ils peuvent être de nature très différente selon les environnements : données, fonctions du code, configuration des ressources, point d'arrêts, *etc*. On parle d'abstraction dans le sens où ces objets fournissent à l'environnement de pilotage une représentation intermédiaire de la simulation (une vue), que l'utilisateur peut consulter et manipuler via l'interface utilisateur. Progress [206] est l'un des premiers environnements à introduire la notion d'objet de pilotage. A titre d'exemple, Progress distingue quatre types d'objets : les données, les points de synchronisation, les fonctions et les scripts.

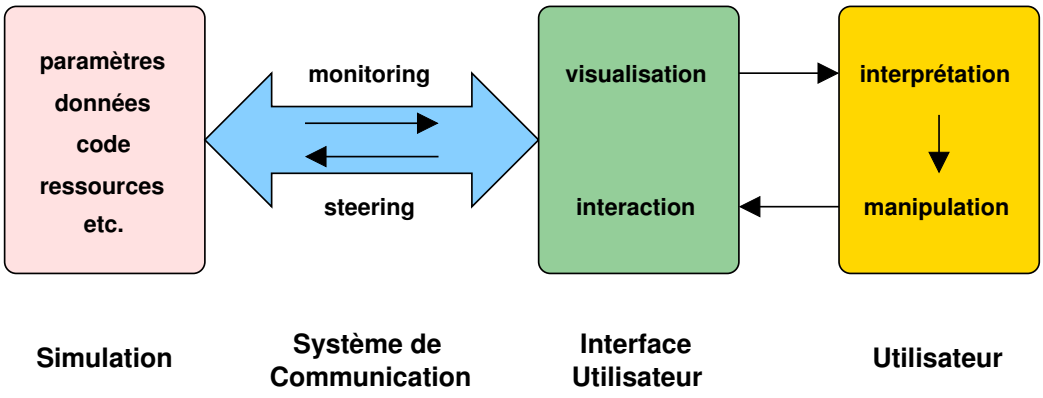


FIG. 3.2 – Environnement de pilotage.

Dans la majeur partie des travaux existants, le pilotage d'une simulation repose sur deux opérations fondamentales : le *monitoring* et le *steering*.

Monitoring – Le *monitoring* est l'opération qui consiste à observer le comportement d'un programme au cours de son exécution. Plus précisément, il s'agit d'extraire dynamiquement des informations relatives aux objets de pilotage (un état) et de suivre l'évolution de cet état au cours du temps. Le rôle du système de communication est de transférer ces informations à l'interface utilisateur, pour qu'elles y soient présentées graphiquement sous forme de textes ou d'images. Les techniques de visualisation jouent alors un rôle prépondérant dans l'analyse et le traitement des données transmises. On parle de *visualisation en ligne* (*on-line visualization*) pour suggérer le fait que l'on suit l'évolution des calculs en temps-réel, image par image, de manière comparable à une animation graphique.

Steering – Le *steering* est l'opération qui consiste à modifier le comportement de la simulation. L'interface utilisateur offre des moyens d'interaction avec la simulation (e.g. ligne de commandes, interacteurs graphiques ou *widgets*, interacteurs haptiques, *etc*.) qui permettent à l'utilisateur de déclencher des *actions de pilotage*, comme par exemple la modification d'un paramètre. Ces actions sont acheminées par le système de com-

munication jusqu'à la simulation, sous forme de requêtes (ou de messages). La simulation est alors en charge de traiter les requêtes entrantes et éventuellement de retourner un résultat à l'interface utilisateur.

3.1.3 Classification des environnements de pilotage

On distingue dans la littérature [153] trois classes d'environnements selon les fonctionnalités qu'ils proposent : les environnements pour *l'exploration du modèle*, les environnements pour *l'expérimentation algorithmique*, et ceux pour *l'optimisation des performances*.

Exploration du modèle – L'utilisateur s'intéresse particulièrement aux données de la simulation dans le but d'améliorer sa compréhension du modèle physique. Ces environnements, comme par exemple CUMULVS [122], permettent de visualiser en ligne les résultats intermédiaires produits par la simulation. Par ailleurs, l'utilisateur a la possibilité de modifier certains paramètres du modèle. Grâce à un retour visuel direct sur les actions qu'il produit, les relations de cause-à-effet deviennent plus évidentes pour l'utilisateur. Cette classe d'environnement est caractérisée par le transfert fréquent de gros volumes de données entre la simulation et le code de visualisation.

Expérimentation algorithmique – L'utilisateur s'intéresse au code en lui-même dans le but d'expérimenter diverses méthodes de résolution et d'adapter manuellement les paramètres des algorithmes. Par exemple, l'utilisateur peut décider de diminuer la valeur du seuil de convergence d'une méthode numérique afin d'obtenir un résultat plus précis ou encore d'aiguiller le flot d'exécution vers une nouvelle méthode numérique. Dans certains environnements comme VASE [113], il est également possible « d'injecter du code » au cours de l'exécution afin de reprogrammer dynamiquement le comportement de certaines procédures. Les environnements pour la *construction interactive de programmes* entrent également dans cette catégorie. Par exemple dans SCIRun [170], il est possible de changer les paramètres d'un solveur de calcul, voire de changer le solveur lui-même sans qu'il soit nécessaire de redémarrer l'application.

Optimisation des performances – L'utilisateur cherche à analyser et à optimiser les performances d'une application numérique. Plus précisément, il interagit avec la configuration de l'application, en vue d'améliorer l'utilisation des ressources. Par exemple dans l'environnement Falcon [100], l'utilisateur peut contrôler la charge des processeurs au cours de l'exécution parallèle et commander si nécessaire un rééquilibrage, en modifiant manuellement la distribution des données entre les processeurs. L'extraction des données doit être la plus efficace possible pour ne pas perturber trop les performances de la simulation observée. Les données considérées sont en général de petites tailles (ensemble de scalaires caractérisant l'état de la simulation).

3.2 Caractéristiques des environnements de pilotage

Les environnements de pilotage existants (ou ayant existé) présentent une très grande diversité aussi bien dans les modèles élaborés que dans les technologies utilisées. Dans cette section, nous examinons les principales caractéristiques des environnements de pilotage afin de donner une vision d'ensemble du domaine au lecteur. Notre étude se découpe en trois sous-sections, correspondant aux trois unités logiques formant un environnement de pilotage, à savoir : la simulation numérique, le système de communication et l'interface utilisateur (Fig. 3.2).

3.2.1 Intégration des simulations numériques

L'intégration d'une simulation numérique est le processus qui permet de la transformer en une simulation interactive (ou pilotable). Afin d'être aussi générique que possible, l'intégration d'une simulation repose sur un *modèle de représentation des simulations*, décrivant les éléments impliqués dans le processus de pilotage (i.e. les objets de pilotage). L'abstraction introduite par ce modèle permet de considérer la problématique du pilotage indépendamment d'un code de simulation particulier. La description d'une simulation dans un modèle abstrait est généralement fournie par l'utilisateur, qui annote le code source selon un technique appelée *l'instrumentation*. Par la suite, nous détaillons les étapes de l'instrumentation d'un code, ainsi que les modèles de représentation

utilisés pour décrire un code de simulation et ses données. Nous achevons cette section en présentant quelques travaux reliés.

Instrumentation

L'instrumentation est une technique qui consiste à annoter manuellement le code source d'une simulation pour l'intégrer dans l'environnement de pilotage. L'instrumentation repose sur une bibliothèque de programmation (API) qu'il faut « *linker* » à la simulation au moment de la compilation. L'utilisateur positionne dans le corps de la simulation un ou plusieurs *points d'instrumentation*, qui seront le support des diverses interactions (envoi/réception de données, exécution d'actions déclenchées, point d'arrêt, *etc*.). En pratique, un point d'instrumentation est matérialisé dans le code par un appel de fonction (de l'API) dont le rôle principal est « de passer la main » à l'environnement de pilotage pour qu'il effectue les traitements appropriés. Par exemple, dans CUMULVS [165], un point d'instrumentation unique, incarné par la fonction *sendToFE()*, permet à lui seul de contrôler la boucle principale de calcul d'une simulation SPMD, d'envoyer ou de recevoir des données (Fig. 3.3). Dans d'autres environnements comme Progress [206], Magellan [209] ou DAQV [132], il est possible d'utiliser plusieurs points d'instrumentation disséminés dans le code. Chaque point d'instrumentation est associé à une ou plusieurs données et sert de point d'accès synchrone pour ces données. Magellan distingue deux types de points d'instrumentation : les *sensors* (accès en lecture) et les *actuators* (accès en écriture). Dans VASE [113], les points d'instrumentation sont assimilés à des points d'arrêt (*breakpoints*) permettant de suspendre l'exécution de la simulation, d'accéder aux données en lecture/écriture ou encore d'exécuter des scripts d'interaction (programmés à distance par le client).

L'instrumentation d'un code de simulation est généralement décomposée en trois phases. Tout d'abord, *la phase d'initialisation* permet de mettre en place l'environnement de pilotage. L'utilisateur décrit la simulation et les objets de pilotage qu'il juge pertinent pour son problème, ce qui a pour effet de masquer le reste de la simulation à l'environnement de pilotage. Cela revient le plus souvent à déclarer les données et les paramètres qui seront accessibles au cours de l'exécution (*publication des données*). A l'issue de la phase d'initialisation, l'environnement de pilotage est mis en place et la simulation est prête à interagir avec des programmes extérieurs : c'est *la phase d'interaction* au cours de laquelle les utilisateurs se connectent à l'environnement de pilotage, observent et modifient le comportement de la simulation. De manière générale, le rôle de l'instrumentation dans cette phase consiste à localiser les endroits précis du code où les diverses interactions pourront avoir lieu. En effet, il faut bien considérer que ces interactions (et plus particulièrement l'accès aux données) ne peuvent pas survenir n'importe où et quand dans le code de la simulation, ce qui poserait un sérieux problème de *cohérence*. Finalement, intervient *la phase de terminaison*, qui permet de libérer les ressources utilisées par l'environnement de pilotage.

```
1  /* initialize "myapp" for CUMULVS */
2  stv_init("myapp",msgtag,nproc,nodeid);
3
4  /* start main iterative loop */
5  do {
6    /* usual calculation */
7    done = work(&timestep);
8    // pass control to CUMULVS
9    nchanged = stv_sendToFE();
10   /* program response to nchanged steered parameters */
11 } while(!done);
12
13 /* exit */
14 stv_exit();
```

FIG. 3.3 – Pseudo-code d'instrumentation d'une simulation avec CUMULVS.

L'instrumentation est une approche flexible qui permet d'intégrer un large éventail d'applications, mais pouvant nécessiter une bonne expertise du code. En effet, la plupart des environnements basés sur l'instrumentation ne font pas de distinction précise entre le développeur du code et l'utilisateur final. Les codes les plus fréquemment rencontrés dans le domaine du calcul scientifique sont des programmes séquentiels ou parallèles (SPMD)

écrits en C, Fortran ou C++. Ces codes communiquent le plus souvent par échange de messages avec des bibliothèques comme MPI ou PVM. Idéalement, un environnement de pilotage se doit de proposer une solution portable la plus générique possible, et ne pas faire d'hypothèses trop fortes sur la nature des codes à piloter.

Représentation des simulations numériques

L'instrumentation des simulations repose généralement sur une vision du code relativement simplifiée : la boucle de calcul principale d'une simulation parallèle dans CUMULVS, un ensemble de points d'instrumentation « nommés » sans structure particulière dans Progress [206], Magellan [209], CSE ou DAQV [132]. D'autres environnements utilisent un modèle de représentation dit *structuré*, donnant une vision de la simulation plus réaliste. Un tel modèle permet de se repérer plus précisément dans le code de la simulation. Sur la base d'une telle représentation, il est possible d'affiner le pilotage et de coordonner les interactions plus efficacement. Dans VASE par exemple, la simulation est vue comme un ensemble de tâches et de sous-tâches (blocs de codes) connectés à l'aide d'un *control-flow graph* (CFG). Le CFG est obtenu par annotation du code source et un outil dédié permet de faciliter l'insertion de points d'arrêt sur les arcs du CFG. Dans PathFinder [106], la simulation est vue comme un ensemble de *transactions*. Une transaction représente un bloc de code logique, éventuellement réparti entre plusieurs processus, comme par exemple une étape de communication. La bibliothèque de communication PVM a été *wrappée* pour automatiser l'instrumentation des transactions de communication de type *send/recv*. L'utilisateur a également la possibilité d'instrumenter le code pour définir ses propres transactions.

Certains environnements, comme DISCOVER [133] ou MOSS [79], proposent un modèle de représentation des simulations *orienté objet*. Dans ces environnements, la simulation est alors vue comme un ensemble d'objets d'interaction (*interaction object*), comparables à des objets distribués. C'est une approche très flexible qui permet à l'utilisateur de spécifier entièrement les interactions possibles sous forme d'invocations de méthodes distantes (RMI). Cette approche s'applique essentiellement pour l'intégration de codes de simulation orientés objets (Java ou C++). Ces codes sont typiquement organisés en objets de calcul (*computational object*), encapsulant les structures de données de l'application. Les codes « non orienté objet » (C/Fortran) doivent être préalablement convertis en objet grâce à l'utilisation d'un *wrapper* C++, ce qui peut impliquer une restructuration plus ou moins importante de ce code. Dans DISCOVER, un objet d'interaction se compose de méthodes donnant une vue de l'objet (*views*) et de méthodes servant à modifier cet objet (*commands*). Cet objet peut être localisé sur un seul nœud de calcul ou distribué sur tout ou partie des nœuds de la simulation. Au cours de l'exécution, les objets d'interaction peuvent être créés, détruits, ou peuvent encore migrer vers un autre nœud. MOSS (Mirror Object Steering System) utilise également une approche orientée objet distribué pour décrire la simulation. Un objet d'interaction dans MOSS se compose d'un ensemble d'attributs (les données) et de méthodes (les actions). MOSS introduit la notion d'objet « miroir » (*mirror object*), reflétant à distance les objets de calcul définis dans la simulation. Un objet « miroir » possède la même interface que l'objet de calcul qu'il reflète et partage le même état grâce à un mécanisme de mise à jour automatique des attributs.

Description des données

Le modèle de description des données doit être suffisamment flexible pour supporter une grande variété de structures de données. Ce modèle est de première importance dans un environnement de pilotage, car il conditionne à la fois la nature des données pilotables et l'efficacité de l'accès aux données. En outre, ce modèle doit offrir une abstraction suffisante pour permettre à l'interface utilisateur de découvrir dynamiquement les informations relatives aux données publiées par la simulation (e.g. identifiant, type, taille, unités physiques, *etc.*). Ces informations peuvent être utilisées pour associer une représentation graphique aux données et construire une application de visualisation adaptée à la nature des données (cf. Sec. 2.3.1). La description des données régulières comme les tableaux (à une ou plusieurs dimensions) ne pose généralement pas de problème. En revanche, la description de structures de données plus irrégulières comme des maillages ou des graphes restent une difficulté majeure en raison de l'absence d'un modèle de représentation standard et unique. La représentation de ces données peut varier considérablement d'un code à l'autre, et à notre connaissance, aucun environnement de pilotage ne permet de manipuler nativement de telles structures de données. Un autre aspect important d'un modèle de données est qu'il doit permettre de décrire les données directement en mémoire

(*wrapping*), sans qu'il soit nécessaire pour l'utilisateur de recopier les données et/ou de les convertir dans une structure de données interne à l'environnement de pilotage. Cela permet d'économiser l'espace mémoire et d'accélérer le transfert des données (envoi zéro-copie).

En général, les environnements de pilotage se limitent à des structures de données relativement simples. Ce sont essentiellement des données de type scalaire et des tableaux à une ou plusieurs dimensions, le plus souvent contiguës en mémoire. Les grilles structurées sont généralement assimilées à des tableaux denses dans ces environnements. Par exemple, dans les premiers environnements de pilotage comme Progress, Magellan ou VASE, les données manipulées sont uniquement des scalaires ou des tableaux 1D. Dans CSE, il est également possible de gérer des chaînes de caractères. Très peu d'environnements proposent un modèle de description des données distribuées, le plus souvent parce qu'ils ne considèrent que des simulations numériques parallèles en mémoire partagée (simulations multi-threads), ou bien simplement parce que ces environnements ne prennent pas en compte la distribution des données. Dans ce dernier cas, les données sont décrites séparément sur chaque processeur (CSE, Magellan, Falcon, *etc.*). D'autres environnements comme VIPER, DAQV, DISCOVER ou CUMULVS possèdent un « vrai » modèle de description des données en mémoire distribuée. Dans la première version de DAQV, les données supportées sont uniquement des tableaux distribués par des directives HPF. Ce modèle a été étendue dans DAQV-II pour supporter des distributions rectilinéaires plus générales. Dans DISCOVER, le programme doit déclarer sur chaque processeur une paire de fonctions *gather()/scatter()* prenant en compte la distribution des données.

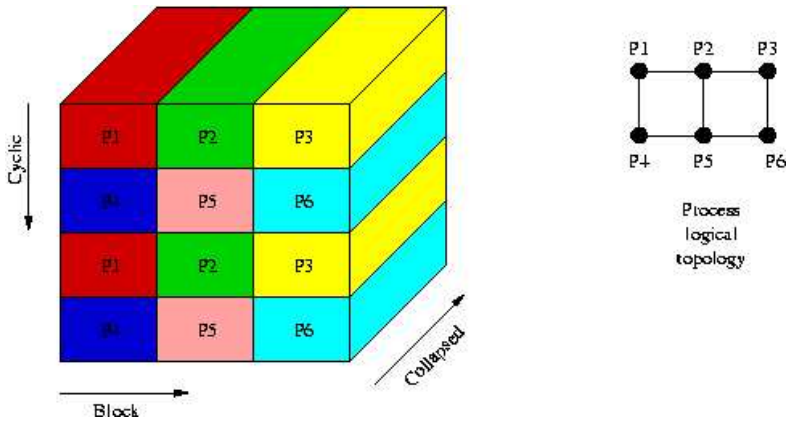


FIG. 3.4 – Décomposition des données dans CUMULVS.

CUMULVS est certainement l'environnement de pilotage possédant le modèle de description des données le plus avancé. Il permet de prendre en compte des paramètres scalaires, des tableaux denses multi-dimensionnels et des ensembles de particules. Les tableaux dans CUMULVS (*field*) peuvent être locaux (i.e. situés sur un seul processeur), répliqués sur tous les processeurs ou distribués. La description des tableaux distribuées se divise en deux étapes : la description du *layout*, où comment les données sont réparties entre les processeurs, et la description du stockage, c'est-à-dire comment les données sont organisées dans la mémoire de chaque processeur. CUMULVS supporte des distributions de type bloc-cyclique selon un formalisme inspiré de HPF (cf. Sec. 2.1.3). Pour chaque processeur, le programme doit spécifier le domaine globale et la décomposition de ce domaine, afin de calculer les blocs en sa possession. La décomposition du tableau selon chaque axe peut être donnée explicitement par l'utilisateur ou obtenue en précisant un type de décomposition standard : bloc, cyclique, ou *collapsed*. Comme le montre la figure 3.4, les blocs résultant de cette décomposition sont ensuite associés à une grille de processeurs logiques. Les données en mémoire dans CUMULVS sont supposées contiguës, mais l'espace de stockage alloué peut éventuellement dépasser la taille réelle des blocs, ce qui permet de prendre en compte le cas où des « cellules fantômes » (*ghost cells*) entourent le tableau. Les *ghosts* sont fréquemment utilisés dans les simulations physiques pour échanger les données aux frontières des blocs, et assurer « la continuité » des calculs nécessitant la connaissance d'un voisinage. Les ensembles de particules considérés dans CUMULVS sont distribués de manière identique aux tableaux. Plus précisément, chaque particule possède une coordonnée entière dans le domaine et est associée localement à un bloc. L'utilisateur a

la possibilité d'attacher des tableaux de données aux particules.

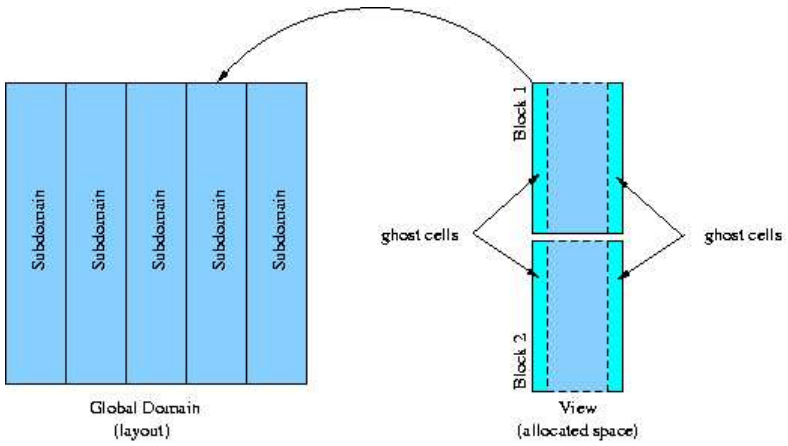


FIG. 3.5 – Représentation des données dans PAWS.

Parmi les travaux reliés, il convient de citer PAWS [44, 118], Global Arrays [157, 158] et CCA M×N [5]. Ces travaux abordent plus largement la problématique du couplage de codes et de la redistribution des données entre des codes couplés. PAWS supporte deux types de données : des scalaires et des tableaux multidimensionnels distribués (Fig. 3.5). Dans PAWS, il faut déclarer explicitement sur chaque processeur le domaine global (l'espace d'indices du tableau) ainsi que les sous-domaines effectivement stockés sur ce processeur. Notons que l'ensemble des sous-domaines de tous les processeurs doit nécessairement recouvrir la totalité du domaine. Pour chaque sous-domaine, l'utilisateur a la possibilité de multiples pointeurs lorsque les blocs de données (*view blocks*) ne sont pas contigus en mémoire. PAWS permet de prendre en compte des *ghost cells* dans les blocs de données alloués. Le modèle de données de PAWS est très général et très flexible. Il permet de prendre en compte tout type de distributions rectilinéaires denses, contrairement à CUMULVS qui se limite aux distributions bloc-cycliques. Tout comme PAWS, Global Arrays [157, 158] permet de gérer des tableaux multidimensionnels distribués en mémoire. L'originalité de Global Arrays tient au fait qu'il propose un modèle de données en mémoire partagée, permettant de voir l'ensemble des tableaux distribués comme un tableau virtuel global (*global array*). Deux modes d'accès aux données sont prévus : un accès direct à l'espace de stockage local et un accès au tableau global. Dans ce dernier cas, un appel de fonction explicite permet de transférer les données modifiées dans le tableau global vers les espaces de stockage locaux. Global Arrays se distingue des DSMs (Distributed Shared Memory) dans la mesure où l'utilisateur contrôle explicitement la localité des données et les accès distants. Pour terminer, il convient de souligner certains travaux récents menés dans le projet CCA M×N dont l'objectif est de définir un composant de redistribution. Ce composant repose sur la spécification d'un modèle de données distribuées, intégrant la plupart des travaux précédents (CUMULVS, PAWS et Global Arrays). Notons que ces travaux n'abordent actuellement que le problème de la redistribution pour des données régulières.

Autres approches et travaux reliés

Pour certains utilisateurs, l'annotation du code source peut sembler une contrainte importante, fastidieuse voire rédhibitoire. Certains travaux reliés ont donc cherché par diverses techniques à simplifier et à automatiser le processus d'intégration. Par exemple, l'environnement CAVEStudy [185] permet d'intégrer une simulation existante sans modifier les sources (i.e. uniquement à partir d'un binaire), simplement en exploitant les fichiers entrées et de sorties de la simulation. Dans VASE [113], une phase d'annotation préliminaire a pour objet de décrire un squelette du code (insertions de commentaires). L'instrumentation est ensuite réalisée avec une interface graphique permettant d'assister l'insertion de points d'instrumentation. SWIG (Simplified Wrapper and Interface Generator) [42, 168] est un outil qui permet de connecter des langages scripts (Python, Perl, TCL, *etc.*) avec un programme C/C++. En utilisant SWIG, il est possible par exemple de remplacer la fonction *main()* d'un programme C par un interpréteur de scripts à partir duquel on contrôle l'application (appels de fonctions). Dans [43], SWIG démontre ses possibilités pour piloter une application en dynamique moléculaire.

On trouve encore des travaux reliés dans des domaine plus éloignés comme le déboguage en parallèle (e.g. P2D2 [60]) ou l'introspection dynamique de codes (DynInst [9], ParaDyn [142], AutoPilot [188], *etc*.). Avec ces outils, il n'est pas nécessaire d'instrumenter le code car toute l'information utile est générée automatiquement à la compilation. De tels outils peuvent permettre d'alléger la phase d'instrumentation manuelle du code, mais une intervention de l'utilisateur est toujours nécessaire pour décrire la distribution des données, déterminer des points d'interaction cohérents avec le code, c'est-à-dire respectant la physique du modèle sous-jacent.

Les PSEs (Problem Solving Environment) sont des environnements de programmation visuelle pour la construction interactive de simulation numérique. Dans un PSE, une application est vue comme un réseau de modules interconnectés selon le paradigme *data-flow* (Fig. 3.6). Certains PSEs comme SCIRun [170] ou COVISE [126] possèdent des modules particuliers dédiés à la visualisation et au pilotage des simulations. Dans cette approche, modélisation, visualisation et pilotage sont réalisés en même temps. La puissance d'un PSE repose sur la bibliothèque de modules fournis et sur sa capacité à intégrer de nouveaux modules. Un tel système est prévu pour construire et tester rapidement de nouvelles applications, mais n'est pas bien adapté pour étudier des applications existantes. Même si la plupart des PSEs proposent des solutions pour intégrer de nouveaux modules, l'intégration complète d'une simulation nécessiterait de restructurer l'ensemble du code en modules (*recasting*). Par exemple dans SCIRun, la définition d'un nouveau module nécessite l'implantation d'une classe C++.

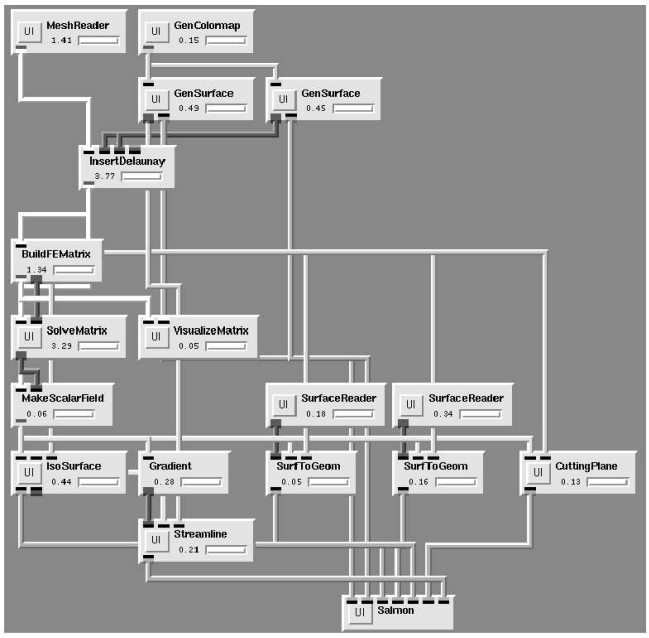


FIG. 3.6 – Exemple d'un réseau *data-flow* dans le PSE SCIRun.

3.2.2 Système de communication

Le système de communication est l'élément central dans un environnement de pilotage ; il a principalement deux rôles fonctionnels : le transfert des données (*monitoring*) et l'acheminement des requêtes de pilotage (*steering*). En outre, il est responsable de *coordonner* les diverses interactions, afin de garantir la cohérence des opérations effectuées. Idéalement, le système de communication prend en charge de nombreuses difficultés (interopérabilité, performance, fiabilité, dynamicité, sécurité) qui doivent s'intégrer logiquement au sein d'un environnement de pilotage.

Architecture

A l'exception des PSEs, la plupart des environnements de pilotage considèrent le code de simulation et de visualisation comme des applications séparées. A ce titre, le pilotage s'apparente à un problème de couplage , et c'est précisément le rôle du système de communication de réaliser ce couplage. On trouve parmi les environnements de pilotage essentiellement deux types d'architecture : l'architecture client/serveur simple et l'architecture client/serveur/client (Fig. 3.7).

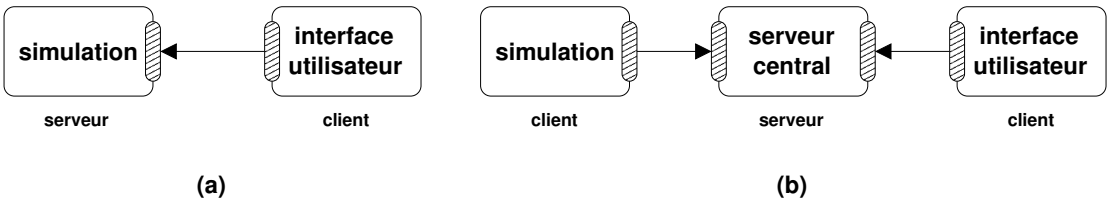


FIG. 3.7 – Architecture (a) client/serveur simple et (b) client/serveur/client.

Dans le modèle *client/serveur* (C/S), la simulation et l'interface utilisateur sont reliées directement. Le client émet des requêtes auprès d'un serveur « à l'écoute ». A la réception des requêtes, le serveur effectue typiquement certains traitement et transmet éventuellement une réponse au client. En pratique, la distinction entre le client et le serveur est uniquement pertinente au moment de la connexion, car le serveur doit être préalablement lancé en attente des connexions clientes. En revanche, au cours d'une session de pilotage, il arrive fréquemment que les rôles de client et de serveur soient inversés (émission de requêtes dans les deux sens). Dans Magellan [208], CUMULVS [122] et d'autres, la simulation joue le rôle de serveur, auquel l'interface utilisateur se connecte. Dans le PSE COVISE [126], la simulation est déportée dans un serveur et un module client permet de connecter le PSE à la simulation. Notons le cas particulier de Visit [77], où c'est la simulation qui joue le rôle de client et se connecte à un module serveur dans le système de visualisation AVS [2]. Dans le modèle *client/serveur/client* (C/S/C), le système de communication est basé sur un serveur central reliant indirectement la simulation et l'interface utilisateur, tous deux assimilées à des clients. On parle également d'architecture à serveur central. Le rôle du serveur est de gérer les communications et le transfert des données entre les différents processus clients. Les solutions diffèrent selon que le serveur centralise les données ou qu'il sert uniquement à coordonner les transferts. Dans VIPER [183] par exemple, le serveur supervise le transfert directement des données vers une base de données cliente, alors que dans CSE [203], le serveur central joue lui-même le rôle de la base de données. Dans DAQV [104], les données distribuées d'une simulation parallèle sont temporairement agrégées sur un processus central avant de migrer vers les clients. Dans VASE [113], le serveur central sert uniquement à configurer les transferts qui s'effectuent directement entre les différentes applications couplées. L'architecture de DISCOVER [133] est organisée selon un modèle 3-tiers, basé sur un serveur *web* central, des UIs sous forme de navigateur (*browser*) et des agents d'interaction intégrés aux applications pilotables. Dans certains cas comme CSE ou DAQV-II, l'environnement s'apparente à un modèle pair-à-pair dans la mesure où il n'y a pas de distinction formelle entre les différents types de clients, et que c'est la même API qui sert à instrumenter un client de calcul ou de visualisation.

Les environnements de pilotage sont par nature des applications distribuées permettant d'interconnecter plusieurs codes disséminés sur un réseau (Fig. 3.8). Idéalement, un environnement doit permettre à une interface utilisateur d'être connectée à plusieurs simulations en même temps (environnement *multi-applications*) et autoriser plusieurs interfaces utilisateurs à se connecter simultanément à la même simulation (environnement *multi-utilisateurs*). Historiquement, les premiers environnements se limitaient comme Progress au pilotage d'une seule application à la fois, ou comme VASE à la connexion d'une seule interface utilisateur. D'autres environnements plus récents comme CSE, CUMULVS ou DISCOVER n'ont plus ces limitations. Par ailleurs, un utilisateur doit pouvoir se connecter et se déconnecter de la simulation à tout moment. A ce titre, la relation client/serveur est bien adaptée à la problématique du pilotage dans la mesure où le couplage réalisé doit être *dynamique*. Cet aspect est particulièrement intéressant si l'on considère le cas de simulations longues pouvant durer plusieurs jours. Dans ce cas, l'utilisateur souhaite pouvoir se connecter périodiquement pour surveiller

l'évolution des calculs et détecter une éventuelle erreur. Une fois tous les clients déconnectés, la simulation doit poursuivre son exécution normale.

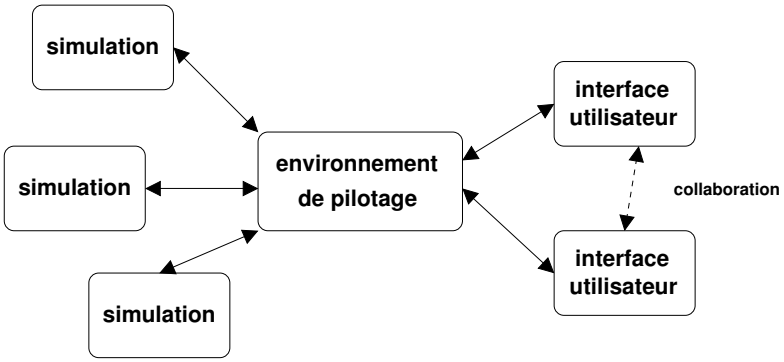


FIG. 3.8 – Environnement multi-utilisateurs & multi-applications.

Lorsque plusieurs utilisateurs ont la possibilité d'être connectés simultanément, l'environnement de pilotage doit prendre en charge *la collaboration* des différents utilisateurs. En effet, ces derniers pourraient par exemple générer des ordres contradictoires et incohérents. Pour faciliter la collaboration entre les utilisateurs, les environnements proposent différentes stratégies. Dans CUMULVS par exemple, les modification des paramètres est contrôlée par un simple jeton (*token*) que les utilisateurs prennent ou relâchent. Dans DAQV, l'environnement n'autorise la connexion que d'un seul client de pilotage à la fois (*control client*) et éventuellement de plusieurs clients de *monitoring* (*data client*). Dans DISCOVER [133], les interactions concurrentes des clients avec l'application sont protégés par un simple mécanisme de *lock* qu'il faut prendre et relâcher explicitement. Par ailleurs, DISCOVER intègre un ensemble de services permettant aux clients de pilotage de diffuser des informations au sein d'un groupe de collaboration (service *chat* pour échanger du texte, service *whiteboard* pour échanger des images, *etc*.). Dans COVISE, un utilisateur (jouant le rôle du maître) a la possibilité de démarrer une session collaborative. Plusieurs utilisateurs distants peuvent ensuite rejoindre la session et visualiser la scène manipulée par le maître. A tout moment, un utilisateur peut demander à devenir maître afin de piloter la session. De plus, un système de vidéo-conférence intégré à COVISE permet aux utilisateurs d'entrer en relation pendant les sessions collaboratives.

Modèles de communication

Dans cette section, nous présentons les différents *modèles de communication* utilisés par les environnements de pilotage pour mettre en relation la simulation et l'interface utilisateur. Ce modèle doit prendre en charge l'accès aux données en mémoire et leur transfert sur un réseau hétérogène. En outre, il doit proposer des mécanismes pour déclencher des actions de pilotage à distance de la simulation. L'accès aux données (de la simulation) est une opération particulièrement critique. En effet, il faut bien considérer que cette opération ne peut pas survenir n'importe où dans le code de la simulation, ce qui poserait un sérieux problème de cohérence (*cohérence spatiale*). En effet, au cours des étapes de calcul, les données ne sont pas en permanence valides, ni continuellement modifiables. Par exemple, une variable ne doit pas être consultée pendant les étapes du calcul qui la modifie, car elle peut contenir des valeurs temporaires non interprétables du point de vue de la physique du modèle. Afin de garantir la cohérence des données transférées, le système de communication doit synchroniser l'accès aux données avec le code de simulation. C'est précisément le rôle de l'instrumentation que d'indiquer au système de communication les endroits précis du code où les données sont accessibles (consultables et/ou modifiables). L'accès aux données distantes peut alors être *synchrone* ou *asynchrone*, dans le sens où il s'applique de manière concurrente à l'exécution de la simulation ou non. Dans le cas synchrone, l'accès (lecture/écriture) est contraint en un point précis, alors que dans le cas asynchrone, il est étendu à une plage d'accès dans le code plus ou moins grande.

Modèle *data-flow*. Le modèle *data-flow* est principalement utilisé dans les PSEs (SCIRun, COVISE). Inspiré des systèmes de visualisation tel que AVS [2], le paradigme *data-flow* est basé sur la notion de modules effectuant une

tâche de calcul à partir de données en entrée et produisant des données en sortie. Les modules interconnectés via des ports forment un graphe orienté par lequel le flux de données circule. La communication entre les modules s'effectue généralement en mémoire centrale, comme par exemple dans SCIRun. Dans COVISE, un autre PSE, certains modules ont la possibilité d'être distribués. Dans ce cas, un courtier, le COVISE Request Broker, prend en charge le transfert du flux de données vers les modules distants via des sockets TCP/IP. Dans la version 2 de SCIRun, basée sur le modèle de composants CCA, les modules ont la possibilité d'être parallèles, reposant sur un mécanisme baptisé PRMI (Parallel Remote Method Invocation) pour transmettre et redistribuer le flux de données d'un composant parallèle vers un autre [65]. L'environnement VASE utilise également une approche de type *data-flow*. Dans VASE, les ports sont associés à des points d'arrêts placés dans le code. Un port peut connecter le port d'une application distante pour former un *data channel*. A chaque exécution du point d'arrêt, les données sont transférées d'une application à l'autre. Le transfert dans VASE repose sur une bibliothèque de type *message passing* (NCSA DTM) prenant en charge l'encodage des données sur un réseau hétérogène. Dans CSE, la communication des clients (appelés satellites) avec la base de données centrale repose sur le paradigme *read/write* implanté au dessus de TCP/IP. Un mécanisme d'évènement associé à l'écriture d'une donnée dans la base permet de construire un flux de données entre les satellites. Typiquement, ce mécanisme est utilisé pour transférer les données entre un satellite de calcul et un satellite de visualisation.

Modèle par échange de messages. Dans ce modèle de communication, l'instrumentation du code repose sur l'utilisation de primitives de communication permettant d'échanger des données sous forme de messages, de manière comparable à MPI. Par exemple, VISIT [77], CUMULVS [165], ou POSSE [144] utilisent le paradigme *send/recv* pour échanger des messages (couplage direct). En particulier, VISIT et POSSE utilisent des sockets TCP/IP. CUMULVS tire profit de la technologie PVM pour définir sa couche de communication. Une fonction unique *sendToFE()* placée dans la boucle de calcul principal permet de communiquer (envoi & réception) avec les différents clients connectés. L'envoi et la réception des données s'effectue de manière asynchrone bufferisée (*pvm_send()*), ce qui permet de recouvrir le transfert des données, du moins partiellement jusqu'au prochain appel de *sendToFE()* à l'itération suivante. Chaque appel à *sendToFE()* implique une copie des données, ce qui permet d'introduire simplement de l'asynchronisme dans les communications tout en garantissant la cohérence des données extraites. En outre, l'utilisateur a la possibilité de paramétrer la fréquence d'envoi (*frame rates*), de sélectionner uniquement un sous-ensemble des données et éventuellement de sous-échantillonner les données pour limiter le volume des transferts.

Modèle par flux d'événements. D'autres environnements, tels Falcon [100], Progress [206], Magellan [208] ou PathFinder [106], utilisent le modèle par flux d'événements (*event stream*) pour communiquer. *Vetter et al.* proposent dans [207] une formalisation de ce modèle appliquée aux environnements de pilotage, en s'inspirant du formalisme largement utilisé dans le domaine de l'analyse des performances et des traces d'exécution. Ce modèle repose sur une instrumentation du code avec des détecteurs ou *sensor*, générant des événements au cours de l'exécution de la simulation. Formellement, un événement est un message structuré de la forme <*event class, processor ID, timestamp, state*>. Le *timestamp* fait référence à l'instant où l'évènement a été capturé (e.g. horloge du processeur, pas de temps logique, *etc*). Basiquement, un évènement est généré à chaque fois que la simulation rencontre au cours de son exécution un point d'instrumentation particulier, appelé *sensor*. Le *sensor* capture un certain état (*state*) caractéristique de la simulation, puis le transmet à un serveur central, appelé *agent*. L'agent stocke, analyse et combine les évènements pour former des événements de plus haut-niveau (*filtering*, *reordering*, *clustering*, *etc*.) transmis à l'interface utilisateur. Dans ce modèle, les actions de pilotage sont vues comme une généralisation des événements de *monitoring*, permettant de modifier l'état de la simulation. Ils sont décrits par un message de la forme <*action class, processor ID, condition, state*>. La condition généralise la notion de *timestamp*, dans le sens où l'action sera activée uniquement lorsque la condition sera localement vérifiée sur un point d'instrumentation particulier appelé *actuator*. Progress, Magellan, Falcon et MOSS reposent sur l'utilisation des bibliothèques DataExchange & PBIO [78], pour le transfert des événements dans un environnement distribué et hétérogène. Afin de contrôler le flux d'évènements (i.e. le transfert des données) et d'éviter l'envoi d'évènements inutiles, la plupart des environnements offrent la possibilité d'activer ou désactiver les *sensors*, ce qui ne résout que partiellement le problème dans le cas de multiples consommateurs. Dans certains environnements, il est également possible de régler la fréquence d'émission des *sensors*. Dans MOSS, le transfert des évènements entre les producteurs et les consommateurs s'effectue directement (sans passer par un serveur central), et si au-

cun consommateur n'est inscrit, aucun transfert n'intervient. Les *sensors* et les *actuators* sont un mécanisme de communication exclusivement synchrone et bloquant, qui permet de contraindre les accès en lecture/écriture en des points précis du code et de garantir la cohérence des accès. Afin d'optimiser les transferts dans Falcon, l'accès aux données repose sur l'utilisation d'un thread dédié à la communication, permettant d'accéder aux données concurremment à l'exécution de la simulation (via la mémoire partagée). La cohérence des accès est alors garantie par des mécanismes de synchronisation et d'exclusion mutuelle classiques dans les bibliothèques de threads (*mutex*, variables de condition). Pour terminer, notons que ces environnements implantent généralement un mécanisme de communication supplémentaire, appelé *probe*, permettant de consulter ou de modifier les données de manière complètement asynchrone, mais sans aucune garantie de cohérence.

Modèle de haut-niveau et approches hybrides. On trouve encore d'autres modèles de communication, que nous qualifions de haut-niveau dans le sens où ils reposent sur un paradigme de communication de type RPC ou RMI issues des *middlewares*. Dans ces environnements, le pilotage est le plus souvent basé sur *un système de requêtes* dirigé par le client. Cette approche s'oppose au modèle évènement entièrement dirigé par la simulation. En pratique, les systèmes à base d'évènements cités précédemment utilisent également un système de requêtes plus ou moins évolué permettant par exemple d'activer ou de désactiver les *sensors*. Par exemple, VIPER utilise la technologie RPC (Remote Procedure Call) pour communiquer et XDR pour l'encodage des données. Dans certains environnements, le système de communication repose sur un modèle de communication *hybride*, fusionnant deux modèles de communication différents. En général, un système de requêtes est combiné avec un système de transfert de plus bas-niveau (envoi de messages, flux d'évènements) reposant sur une technologie différente. Par exemple, RealityGrid [53] utilise un système de requêtes basé sur SOAP (Simple Object Access Protocol) [51], mais le transfert des données s'effectue via GlobusIO selon le paradigme de communication *read/write*. L'environnement DISCOVER [133] utilise un système de communication hybride, basé sur des objets distribués. Un serveur *web* central communique avec des applications numériques via les protocoles standards Java RMI, CORBA, mais aussi TCP/IP. Les clients (UI) sont des navigateurs *web* classiques utilisant le protocole HTTP (requêtes GET, POST) pour dialoguer avec le serveur. Dans MOSS [79], le pilotage s'exprime également par des invocations de méthodes à distance (RMI) sur des objets distribués (implantés en CORBA). MOSS introduit la notion d'objet « miroir » (*mirror object*), des objets distribués particuliers partageant les méthodes de l'interface mais également l'état, c'est-à-dire les données déclarées dans l'interface IDL sous forme d'attributs. Un mécanisme de mise à jour automatique (dont on peut paramétrer la fréquence) permet alors de maintenir l'état de l'objet « miroir » cohérent avec la simulation. Ce mécanisme repose sur un modèle de communication par flux d'évènements implanté au dessus de DataExchange & PBIO [78].

Coordination

Le système de communication est également responsable de coordonner les interactions survenant entre la simulation et l'interface utilisateur, principalement pour garantir la *cohérence temporelle* des opérations effectuées. Le problème de la cohérence temporelle survient principalement lorsque l'on s'intéresse au pilotage de simulations numériques parallèles en mémoire distribuée. Ce problème devient évident si l'on considère l'extraction des données pour la visualisation en ligne. Chaque processeur transmet une portion des données à l'interface utilisateur qu'elle rassemble pour construire une image. Dans ce cas, il ne suffit pas de garantir que l'accès aux données réalisé localement sur chaque processeur s'effectue de manière cohérente en des endroits autorisés du code (*cohérence spatiale*). Il faut encore garantir que l'ensemble des informations extraites correspondent au même « instant » dans la simulation, pour prévenir une éventuelle désynchronisation lors de l'accès aux données (*cohérence temporelle*). Sans la mise en place de mécanisme de coordination, l'image produite par l'interface utilisateur peut mélanger des pas de temps différents et donc ne plus être cohérente dans sa globalité. Plus généralement, le problème de la cohérence temporelle se pose pour toute interaction avec un code distant nécessitant de transférer plusieurs informations corrélées en temps, qu'il faut regrouper pour les interpréter ensemble. Ce problème peut également survenir avec des simulations séquentielles, dans le cas où l'image observée nécessite la combinaison de plusieurs séries de données, comme par exemple les différentes valeurs associées aux nœuds d'un maillage (e.g. position, pression). Dans ce cas, il faut garantir que les données transmises correspondent au même pas de temps physique. Dans cette section, nous examinons les différentes stratégies utilisées dans les environnements de pilotage pour coordonner les interactions en parallèle.

Tout d'abord, il faut constater que certains environnements n'abordent pas cette problématique, et préfèrent laisser le soin à l'utilisateur de garantir lui-même la cohérence des traitements qu'il effectue. C'est la cas par exemple de MOSS [79] ou de DISCOVER [133]. Dans Progress [206], Magellan [209] ou VASE [113], un mécanisme de points de synchronisation (*synch. points*) permet de suspendre momentanément l'exécution de la simulation et de réaliser des interactions sur ces points. Toutefois, sans coordination, rien ne garantit que tous les processus sont stoppés à la même itération, ni même que certains processus ne soient pas bloqués en attente sur des communications ou des barrières.

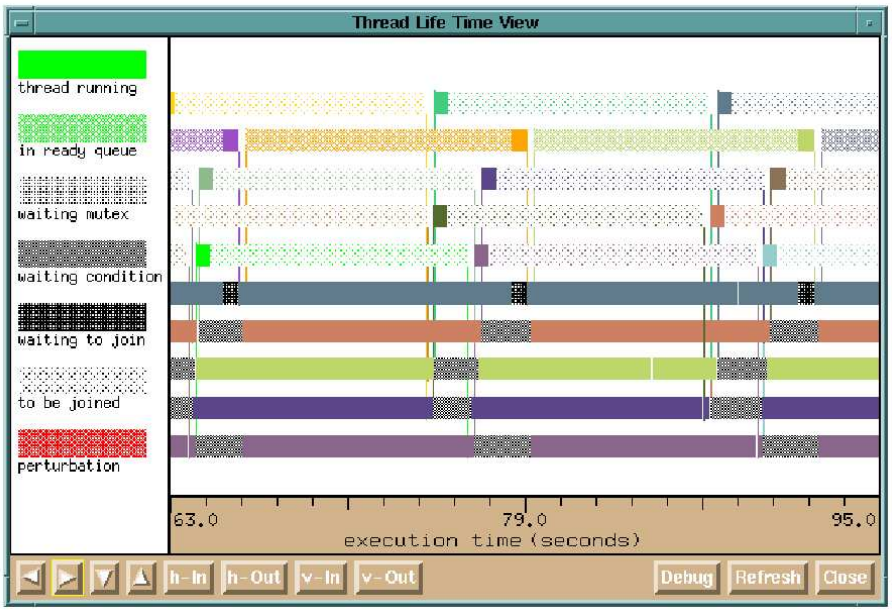


FIG. 3.9 – Suivi « temps-réel » de l'activité des threads dans Falcon.

Reordering. Une première stratégie, appelée *reordering*, est utilisée par Falcon [100], un environnement de pilotage basé sur un système d'évènements. Cette stratégie n'implique *a priori* aucune synchronisation entre la simulation et l'interface utilisateur. Comme nous l'avons vu dans la section précédente (modèle par flux d'évènements), chaque *sensor* est programmé pour envoyer systématiquement les évènements produits vers un serveur central découplé de l'UI. Ce serveur stocke temporairement une file d'évènements, les réordonnent entre eux et les transmet à l'UI qui joue le rôle du consommateur d'évènements. La définition d'un ordre sur les évènements est un problème classique dans les systèmes d'analyse de traces d'exécution parallèle. Les techniques généralement appliquées dans ces systèmes reposent sur la notion de *timestamp* pour dater précisément les évènements avec l'horloge interne du processeur. Toutefois ces techniques peuvent conduire à des erreurs lorsque les horloges des différents processeurs sont faiblement synchronisés, nécessitant la mise en place de stratégies plus complexes (e.g. ordre causal [125]). Toutefois, ces solutions restent difficiles à mettre en œuvre dans les environnements de pilotage dont la nature *on-line* empêche de connaître par avance tous les évènements produits, l'ordre de réception des évènements n'étant évidemment pas fiable. Pour remédier à ce problème, Falcon utilise un filtre d'évènements reposant sur des règles de causalité entre les évènements. Lorsqu'une règle est satisfaite, l'évènement associé à la règle sort du filtre. Cette stratégie a permis dans Falcon d'ordonner les évènements associés à une bibliothèque de threads et de les visualiser en temps-réel (Fig. 3.9). A titre d'exemple, l'évènement *mutex_unlock(t,m,n)* est transmis à l'interface utilisateur uniquement si le thread t et le mutex m ont été initialisés (évènements *thread_init(t)* et *mutex_init(m)*) et si le mutex a été précédemment acquis (évènement *mutex_end_lock(t,m,n)*) avec le numéro de séquence n. La stratégie de *reordering* s'avère relativement bien adaptée pour le *monitoring* des performances (*on-line*), mais n'a jamais été utilisée à notre connaissance pour coordonner l'extraction de données distribuées en mémoire et produire de la visualisation en ligne.

Optimistic steering. Dans PathFinder [106, 143], la simulation est modélisée en termes de transactions. Une transaction représente une étape logique du code perçue comme étant atomique. A chaque fin de transaction, un évènement portant le numéro de transaction local au processus est envoyé à un serveur central (*snapshot manager*). Ce serveur est alors responsable de maintenir un historique des transactions du système et de les ordonner. Pour y parvenir, PathFinder exploite les numéros de transaction locaux et les transactions de communication *send/recv* (considérée comme atomique) pour calculer des relations de dépendance entre les transactions et déterminer des transactions concurrentes, c'est-à-dire un ensemble de transactions distribuées correspondant à une étape logique du programme. Grâce à la connaissance des transactions concurrentes, les informations locales (*local snapshot*) collectées sur le *snapshot manager* peuvent être regroupées en une information globale cohérente en temps (*global snapshot*) qu'il est possible de transmettre à l'interface utilisateur. Dans PathFinder, les opérations de pilotage sont également vues comme des transactions modifiant l'état de la simulation. En particulier, une transaction de pilotage est dite consistante si elle n'est concurrente avec aucune des transactions du programme. PathFinder utilise un stratégie dite « optimiste » (*optimistic steering*) pour coordonner les opérations de pilotage. Dans cette approche, les opérations de pilotage sont d'abord appliquées « à la volée » sans aucune garantie de cohérence. En dépouillant l'historique des transactions, l'agent de coordination décide *a posteriori* si l'opération de pilotage effectuée est cohérente ou non. Si elle est cohérente, la simulation peut continuer son exécution normalement. Dans le cas contraire, PathFinder détermine la dernière date cohérente (*consistent time*) et corrige l'incohérence de manière transparente grâce à un mécanisme de reprise (*checkpointing*).

Synchronisation forte. Dans DAQV-II [104, 132], la coordination des accès aux données en parallèle repose sur un contrôle centralisé de l'exécution de la simulation par un serveur maître (Fig. 3.10). A chaque point d'instrumentation (*probe, mutate*), tous les processus de la simulation se synchronisent sur le serveur maître, puis interrogent le serveur pour savoir si le client a effectué une demande explicite de synchronisation (requête *yield*). Si aucune requête n'est présente, la simulation peut continuer son exécution jusqu'au point d'instrumentation suivant. Dans le cas contraire, la simulation reste suspendue et l'utilisateur peut effectuer des accès aux données en toute sûreté (*data access*). Un ordre *continue* permet finalement de débloquer la simulation. Cette stratégie permet de garantir la cohérence des traitements en parallèle mais impose une très forte synchronisation entre les processus de la simulation, le serveur maître et les clients.

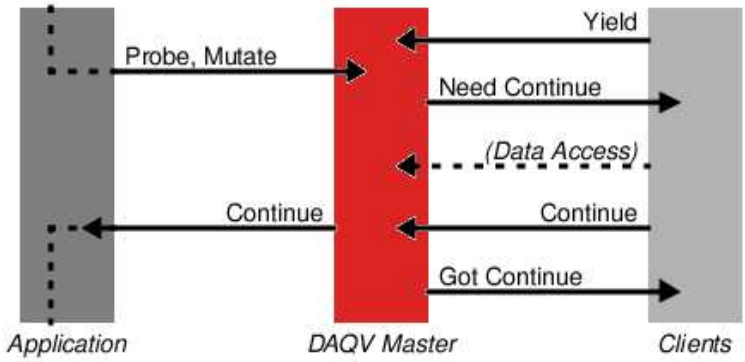


FIG. 3.10 – Synchronisation des traitements dans DAQV.

Loose synchronization. Une dernière stratégie, baptisée *loose synchronization*, est utilisée par CUMULVS [165] pour coordonner l'accès aux données distribuées d'une simulation parallèle. Cette stratégie consiste à planifier la date de traitement d'une requête puis à effectuer ce traitement indépendemment sur chaque processus de la simulation. A chaque itération, un fonction unique (*sendToFE()*) placée dans la boucle principale d'une simulation SPMD permet de gérer et de coordonner les diverses interactions avec les clients connectés (*viewers*). Lorsqu'un *viewer* demande à recevoir des données (*field request*) avec une certaine fréquence p, CUMULVS débute un algorithme de planification des envois découpé en trois phases. Pour commencer, chaque processus P_i communique son itération courante t_i au *viewer* et continue son exécution jusqu'à l'itération suivante $t_i + 1$. Les P_i sont alors en attente d'un message du *viewer* indiquant l'itération t' à partir de laquelle l'envoi des données

pourra commencer en parallèle. La date t' correspond au maximum des itérations non encore dépassé par les P_i. Une fois que les P_i ont reçu la date planifiée, ils sont libres de continuer leurs exécutions jusqu'à ce point. Une fois la planification établie, CUMULVS envoie les données à intervalle régulier, tous les p itérations, sans qu'il soit nécessaire de répéter l'algorithme précédent. Dans CUMULVS, l'envoi des données s'effectue de manière asynchrone (bufferisée), ce qui permet de recouvrir les communications d'une itération à l'autre. Notons que le *viewer* est responsable d'envoyer un acquittement (*XON*) au processus de la simulation pour signaler qu'il a fini de traiter les données reçues. Cet acquittement permet de réguler le transfert entre la simulation et le *viewer*, en ralentissant éventuellement la simulation si la fréquence d'envoi est trop élevé. Contrairement à DAQV, cette stratégie n'implique pas de synchronisation forte entre les processus de la simulation et se trouve particulièrement bien adaptée pour la visualisation en ligne des résultats intermédiaires.

3.2.3 Interface utilisateur

L'interface utilisateur (UI) remplit deux fonctions essentielles. Premièrement, elle présente à l'utilisateur les informations extraites de la simulation. Deuxièmement, elle permet d'interagir avec les objets de pilotage. Sa définition doit être simple et intuitive pour permettre au scientifique d'interpréter et de manipuler facilement les objets de pilotage définis dans la simulation. Les solutions existantes vont de la simple interface textuelle, à l'interface graphique (GUI), et jusqu'aux techniques de réalité virtuelle dans des environnements immersifs. A l'exception des PSEs, l'interface utilisateur se trouve en général découplée de la simulation grâce à l'abstraction mise en place par l'environnement de pilotage. Ainsi, l'UI s'intègre à l'environnement de pilotage de manière symétrique à la simulation, en utilisant une API. Cette API est parfois appelée « *front-end* » par opposition à l'API utilisée pour annoter le code de la simulation appelée « *back-end* ». On peut distinguer plusieurs types d'interfaces utilisateurs, selon le niveau de généricité qui est proposé à l'utilisateur. D'un côté, on trouve des UIs *spécifiques* dédiées à une simulation particulière et de l'autre des UIs *génériques*, réutilisables pour différentes simulations.

Le développement d'une UI *spécifique* (à l'aide de l'API *front-end*) peut s'avérer intéressant pour plusieurs raisons. Tout d'abord, cette solution est nécessaire si l'environnement de pilotage ne fournit pas de solutions plus génériques. Elle permet à l'utilisateur d'adapter l'interface à ses besoins, afin d'utiliser des techniques de visualisation et d'interaction spécifiques. Par ailleurs, cette approche permet de réutiliser une UI existante servant déjà au dépouillement des résultats produits en fichiers. Toutefois, la programmation de ces interfaces est souvent délicate à réaliser et il est généralement préférable de fournir aux utilisateurs une ou plusieurs solutions clés en main. Les UI *génériques* permettent de connecter tout type de simulations (déjà instrumentées) et offrent des moyens d'interaction relativement simples, dépendant des fonctionnalités offertes par l'environnement de pilotage : contrôle du flot d'exécution (*play/stop*), contrôle du flux de données, affichage des informations générales relatives à la simulation, présentation des données dans une table numérique, modification des paramètres, *etc*. Ce type d'interface est particulièrement utile pour prendre en main un environnement de pilotage et valider l'instrumentation d'une simulation. En général, ce sont des interfaces graphiques (GUI), n'intégrant pas ou peu d'outils de visualisation, et construites sur des bibliothèques comme TCL/TK, Java Swing, QT ou encore Motif. Certains environnements proposent encore des interfaces utilisateurs que nous qualifions de *semi-génériques*. Ces UIs sont dédiées au pilotage d'une classe particulière de problèmes, comme par exemple la visualisation et la manipulation des tableaux 2D ou 3D, des maillages ou des molécules. Ce sont généralement des applications modulaires et réutilisables s'intégrant sous forme de classes ou de modules dans un système de visualisation externe, comme par exemple dans VTK [33], OpenInventor [24], AVS/Express [2] ou encore IRIS Explorer [85].

Visualisation

La présentation des informations extraites de la simulation peut être réalisée de différentes manières, allant de l'affichage textuelle à la simple représentation graphique de courbes et d'histogrammes, jusqu'à des techniques de visualisation scientifique 2D et 3D. Les informations à visualiser sont très diverses selon la nature des simulations à piloter (e.g. mécanique des fluides, dynamique moléculaire). Ces informations peuvent être de simples variables scalaires représentant la charge des processeurs ou les paramètres physiques du modèle, ou des ensembles de données plus complexes représentant des objets physiques souvent très volumineux (grilles structurées 2D ou 3D, des maillages non structurés, des ensembles de particules, des molécules, *etc*). Pour

visualiser ces objets, il est nécessaire de leur associer une représentation graphique, impliquant l'utilisation de techniques de visualisation plus ou moins complexes (cf. Sec. 2.3.1). Dans [196], *Kuo et al.* analysent et commentent l'utilisation de différentes techniques de visualisation pour une simulation 3D en électromagnétisme, manipulant des champs de vecteurs (électrique et magnétique). Parmi les techniques évaluées, l'utilisation des plans de coupe et du rendu volumique s'est montrée la plus pertinente pour leur problème. L'utilisation des lignes de champs (*streamlines*) ou des iso-surfaces ne s'est pas montrée très concluante dans leurs expériences en partie à cause des discontinuités présentes dans le champs électromagnétique simulé. Leurs résultats confirment l'idée qu'il n'y a pas une technique de visualisation parfaite à tout point de vue, même pour une simulation particulière. Dans ce contexte, l'utilisation d'un système de visualisation scientifique modulaire et entièrement *configurable* est un atout majeur pour les utilisateurs. Parmi ces systèmes, AVS/Express et IRIS Explorer sont certainement ceux les plus fréquemment utilisés par les environnements de pilotage (Fig. 3.11).

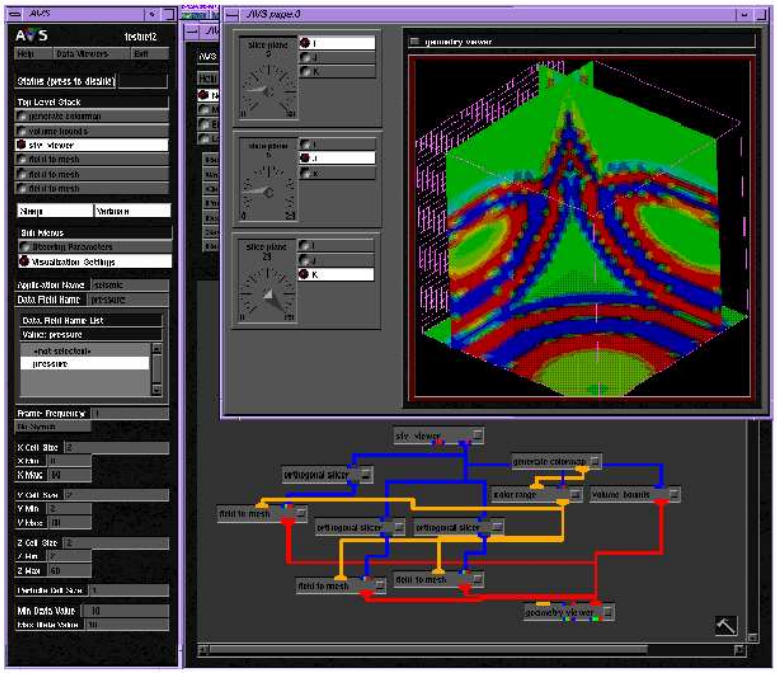


FIG. 3.11 – Utilisation de AVS avec CUMULVS.

Parmi les données que l'utilisateur souhaite examiner, certaines sont continuellement modifiées au cours des étapes de calcul de la simulation, ce qui nécessite de mettre à jour leur représentation graphique afin de garantir qu'elle reflète bien l'état courant de la simulation. Dans ce cas précis, on parle de « visualisation en ligne » pour suggérer le fait que l'on observe l'évolution des données à mesure qu'elles sont calculées. Notons que la visualisation en ligne est une approche pertinente uniquement pour des simulations numériques impliquant un processus de calcul itératif, comme par exemple les schémas de résolution en temps (boucle de calcul). La visualisation des résultats intermédiaires se traduit alors simplement sous la forme d'une animation en temps. Dans ce cas, la représentation calculée et affichée au pas de temps t de la simulation est remplacée au pas de temps suivant, et ainsi de suite. Dans le cas d'une simulation « rapide », il est souvent nécessaire de réguler le flux de données entrant dans le pipeline de visualisation, par exemple en contrôlant la fréquence d'envoi des données ou bien en ralentissant volontairement la simulation afin de pouvoir suivre les étapes de calcul une par une (CUMULVS, MOSS, VIPER, *etc.*). Ces techniques sont utilisées afin d'éviter un engorgement trop important du pipeline de visualisation, ce qui pourrait détériorer considérablement l'interactivité du programme. De manière plus générale, la représentation calculée peut dépendre de plusieurs pas de temps précédents, dont il faut conserver l'historique. En particulier, cette technique est utilisée pour le suivi d'une grandeur scalaire en fonction du temps (courbe 2D). D'autres techniques plus complexes dépendant du temps peuvent également être utilisée (e.g. *particle trace*, *streaklines*).

Interaction

Le pilotage d'une simulation implique plusieurs niveaux d'interaction qu'il convient de distinguer. Le premier niveau d'interaction que nous considérons est *l'interaction de visualisation*, une interaction propre au système de visualisation (UI). Ce type d'interaction permet de naviguer dans la scène 3D, de manipuler les objets graphiques dans l'espace (e.g. rotation, zoom) et de modifier leurs apparences. De manière générale, l'interaction de visualisation permet uniquement de configurer la scène visualisée, elle n'implique pas de communication avec la simulation et ne permet donc pas de modifier son comportement. A l'inverse, *l'interaction de pilotage* permet d'interagir avec la simulation, afin de modifier son comportement. Elle implique donc une étape de communication avec la simulation. Ces interactions dépendent directement des possibilités de l'environnement de pilotage, mais consistent généralement à modifier la valeur de certains paramètres (modifier une condition limite), à déclencher certaines actions (injection d'un fluide ou de particules), à déplacer ou à déformer un objet en lui appliquant une force (e.g. molécule, maillage), *etc*.

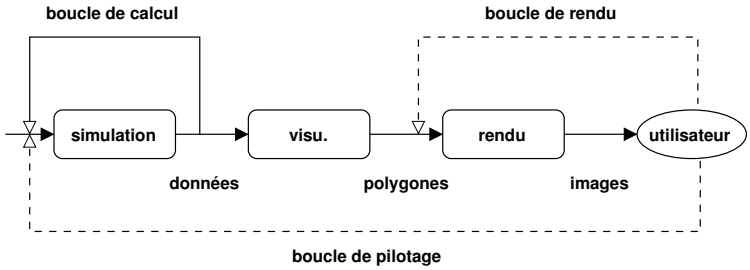


FIG. 3.12 – Les différentes « boucles » dans un environnement de pilotage.

Le pilotage est un processus hautement interactif, nécessitant des temps de réponse suffisamment rapides pour les différentes « boucles » du système (Fig. 3.12). *Brooke et al.* relatent dans [54] les temps de réponse qu'ils ont jugés empiriquement acceptables ou non pour l'utilisateur. Lorsque l'utilisateur navigue dans la scène, celle-ci doit être continuellement redessinée afin de prendre en compte la nouvelle position de la caméra (interaction de visualisation). Pour être acceptable, le temps de réponse de la boucle de rendu ne doit pas chuter en dessous de 10 à 15 mises à jour par secondes (film $\simeq$ 24 Hz), sans quoi la navigation risque d'être dégradée. Remarquons qu'il n'est pas nécessaire au cours de la navigation de recalculer l'ensemble du pipeline de visualisation. En principe, la mise à jour complète du pipeline intervient uniquement lorsque de nouvelles données sont reçues par le système de visualisation. Le temps de post-traitement des données (e.g. le calcul d'une iso-surface) ne doit alors pas dépasser quelques secondes pour être acceptable. En revanche, le temps d'attente acceptable pour une interaction de pilotage (modifiant le comportement de la simulation) est de l'ordre de la minute, ce temps pouvant être augmenté si des résultats intermédiaires (e.g. solveur itératif) sont affichés.

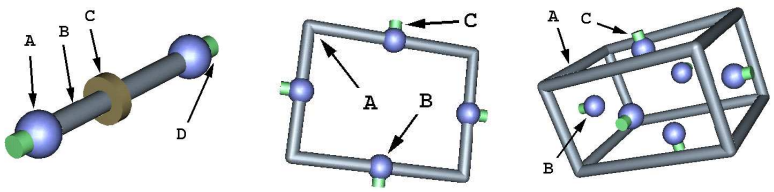


FIG. 3.13 – Exemples de *widgets* 3D dans SCIRun.

Les interactions de visualisation et de pilotage reposent typiquement sur l'utilisation de *widgets*. Un *widget* (Window Object), est un petit composant graphique ayant un aspect et un comportement clairement définis. Les *widgets* sont les briques de construction de toute interface graphique. En 2D, de nombreux widgets sont disponibles : il s'agit de boutons, de cases à cocher, de listes déroulantes, *etc*. En 3D, les *widgets* peuvent posséder une grande variété de géométries (e.g. point, barre, cadre, boîte, sphère) et de comportements, comme

ceux représentés sur la figure 3.13. Ils permettent d'interagir dans la scène 3D par manipulation directe de l'image [61]. Certaines techniques de visualisation sont naturellement combinées avec des *widgets* dans les systèmes de visualisation. Dans ce cas, il s'agit exclusivement d'interaction de visualisation. Par exemple, dans Amira [1], les widgets 3D permettent de positionner des plans de coupes (widget cadre), d'initialiser des *streamlines* (widget point, barre ou cadre), de sélectionner un sous-volume de données (widget boîte ou sphère) afin d'accomplir des traitements supplémentaires (e.g. rendu volumique, iso-surfaces).

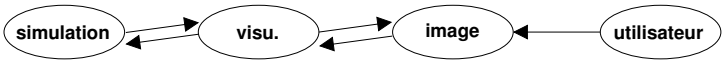


FIG. 3.14 – Interaction par manipulation directe de l'image [59].

Chatzikinos et al. ont proposé dans [59] un modèle permettant de piloter une simulation directement en manipulant l'image. Cette approche favorise l'immersion et l'efficacité de l'utilisateur lors du pilotage. Comme le montre la figure 3.14, la principale difficulté dans cette approche tient au fait qu'il faut être capable « d'inverser » le *pipeline* de visualisation depuis un pixel de l'image affichée jusqu'aux données de la simulation ayant servi à colorier ce pixel. Hors, les systèmes de visualisation *data-flow* classiques ne sont généralement pas prévus pour supporter cette inversion, car la traversée du *pipeline* entraîne une perte d'informations. Pour remédier à ce problème, plusieurs travaux ont proposé d'enrichir le système *data-flow*. AVS/Express [2] et Iris Explorer [85] utilisent une boucle de rétroaction (*feedback loop*) permettant de retrouver la valeur d'une donnée dans l'image (*image probing*). *Felger et al.* [82] ont généralisé cette approche afin de pouvoir inverser l'ensemble des modules formant le *pipeline*. La solution proposée par *Chatzikinos et al.* consiste à enrichir la sémantique du modèle de données qui traversent le *pipeline* afin de conserver les informations relatives aux données originales, ce qui rend possible l'interaction avec les données de la simulation directement en manipulant des points actifs sur l'image. Cette approche a été validée avec une application en chimie cinétique, intégrée sous forme d'un module dans IRIS Explorer. Toutefois, il n'y a aucun résultat dans le cas où la simulation est considérée distante de l'UI.

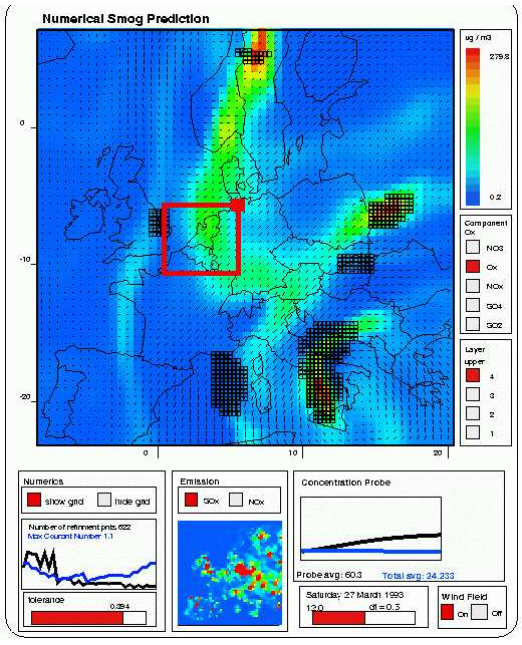


FIG. 3.15 – Utilisation du PGO dans CSE pour une simulation de prédiction du *smog*.

Une dernière technique d'interaction est utilisée dans l'environnement de pilotage CSE [152, 150]. Cette technique se base sur la notion d'objet graphique paramétrée ou PGO (Parametrized Graphics Object), dont

les propriétés sont directement associées aux données de la simulation. L'utilisateur a la possibilité grâce à un éditeur (PGO Editor) de construire et de paramétrer ses propres objets graphiques afin de personnaliser son UI 2D ou 3D (Fig. 3.15). Le point fort du PGO Editor est qu'il permet à la fois de visualiser les données de la simulation et de les modifier simplement en manipulant la représentation graphique de l'objet (déplacement de points actifs). Contrairement au modèle *data-flow*, l'association intrinsèque qu'il existe entre les données et l'objet graphique rend possible dans cet environnement les interactions de pilotage par manipulation directe de l'image. Une telle solution est parfaitement adaptée à la problématique du pilotage, mais ne permet pas de bénéficier de toutes les possibilités offertes par un système de visualisation commercial comme IRIS Explorer (utilisé par ailleurs dans CSE).

Réalité virtuelle

Dans le domaine du pilotage, la définition d'une interface utilisateur intuitive et efficace est un élément clé auquel les techniques de réalité virtuelle (RV) peuvent contribuer. Les environnements virtuels permettent d'améliorer considérablement la sensation d'immersion des utilisateurs et rend les interactions dans l'espace plus intuitive pour la manipulation des ensembles de données 3D. Par ailleurs, l'utilisation des environnements immersifs à base de grands écrans permet, grâce à des résolutions d'images élevées, de favoriser le travail collaboratif, difficilement envisageable sur des stations de travail individuelles (écrans PC). A notre connaissance, il y a encore assez peu de travaux dans ce domaine. On peut citer toutefois les environnements CAVEStudy [185, 186], CSE [151] et CUMULVS [213] qui relatent des expériences de pilotage dans des CAVEs [63]. Nous résumons ici ces résultats.

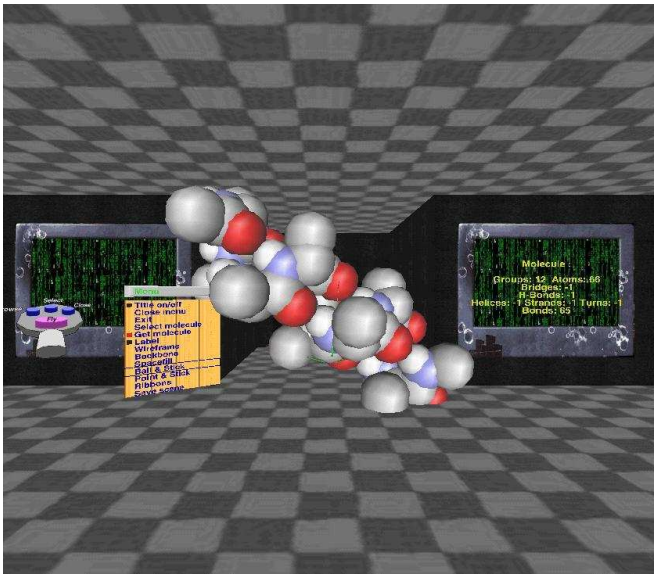


FIG. 3.16 – Pilotage de la simulation NAMD avec CAVEStudy.

CAVEStudy [185, 186] est un environnement de pilotage orienté RV, permettant de piloter une application à partir d'un ou de plusieurs CAVEs distants. Notons que CAVEStudy ne s'intéresse pas exclusivement au pilotage d'applications scientifiques, mais plus largement à tout type d'applications interactives. Contrairement à la plupart des environnements que nous avons présentés jusqu'ici, CAVEStudy s'intéresse à un cycle complet d'exécution d'une simulation pour un jeu de paramètres d'entrée. Dans CAVEStudy, les interactions avec la simulation sont uniquement basées sur l'utilisation de fichiers en entrée et en sortie, sans qu'il soit nécessaire de modifier le code source (i.e. utilisation d'un binaire). En particulier, CAVEStudy exploite les fichiers de sortie de la simulation pour visualiser le résultat des calculs (et éventuellement les résultats intermédiaires). Après quoi l'utilisateur a la possibilité de générer un nouveau jeu de paramètres (fichier d'entrée) et de relancer la simulation. CAVEStudy démontre ses possibilités avec la simulation en dynamique moléculaire NAMD [199] (Fig. 3.16). Les interactions possibles sont toutes accessibles grâce à des menus virtuels 3D et permettent de changer le mode de représentation de la molécule, de sélectionner une nouvelle molécule, le nombre

d'itérations et la température.

Dans CSE [151], les fonctionnalités du PGO Editor ont été étendues pour être utilisées dans des environnements immersifs comme le CAVE. En particulier, la caméra a été intégrée comme un objet paramétré afin d'améliorer la gestion du point de vue, et éventuellement de relier ses caractéristiques (position, orientation) aux données de la simulation. La navigation dans la scène et l'interaction avec les objets sont réalisés grâce à l'utilisation d'un stylet 3D (ou *wand*), à six degrés de liberté, accompagné d'un joystick. Ainsi, en plaçant la caméra à bord d'un bateau dans une simulation de navigation maritime, l'utilisateur a *littéralement* la sensation de piloter le bateau en contrôlant l'orientation et la taille des voiles. Cette impression n'aurait pu être obtenue sur une station graphique traditionnelle, ce qui confirme les bénéfices potentiels de la RV pour le pilotage de ce type d'applications.

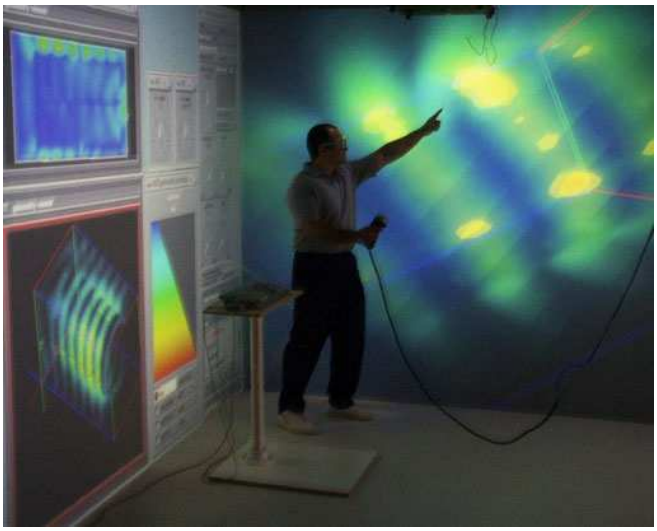


FIG. 3.17 – Interaction de pilotage dans un CAVE dans [196].

Certains systèmes de visualisation comme AVS/Express offrent des extensions pour la RV. Dans [196], *Kuo et al.* utilisent un module appelé VR Viewer [123] permettant de rediriger la sortie graphique d'AVS/Express vers un CAVE (Fig. 3.17). L'interaction avec la visualisation se base comme pour CSE sur un stylet 3D (*wand*), mais un clavier et une souris ont également été conservés à l'intérieur du CAVE pour faciliter l'interaction avec l'UI d'AVS/Express. D'après ces auteurs, l'utilisation d'un grand format d'écran est un avantage considérable, de même que l'utilisation d'interacteur à 6 degrés de liberté pour la manipulation des données. L'utilisation de la stéréo permet d'accentuer l'effet 3D, même si la sensation d'immersion ne leur a pas semblé utile, du moins pour la simulation considérée. Par ailleurs, *Kuo et al.* soulignent le fait que le CAVE est essentiellement utile pour les discussions en groupe et les démonstrations. Dans les autres circonstances, l'utilisation d'une station graphique 3D semble plus appropriée, d'autant que les CAVEs sont un matériel encore relativement rare, imposant généralement de se déplacer. En conclusion, les auteurs soulignent donc l'importance d'utiliser des solutions pratiques permettant de rediriger de manière transparente une application de visualisation classique vers un CAVE.

Des travaux récents dans l'environnement de pilotage CUMULVS [213] ont permis d'étendre la visualisation à des environnements immersifs comme le CAVE [63] ou l'ImmersaDesk [64] (un environnement immersif de taille plus modeste). La solution utilisée par CUMULVS se base sur les technologies VTK [33] et SGI Performer [29]. Un des atouts de Performer est qu'il permet d'effectuer du rendu parallèle (en *multi-pipes*), de même que VTK offre des algorithmes de visualisation parallèle, mais ces possibilités ne sont pas utilisées puisque l'application finale est en fait séquentielle. La figure 3.18 présente la structure générale de l'application. Les données brutes d'une simulation (parallèle) sont d'abord collectées grâce à l'interface CUMULVS. Puis ces données sont converties en structure de données VTK (*vtkStructuredPoints*). A ce niveau, il est possible d'appliquer divers techniques de visualisation aux données afin de décider d'un mode de représentation approprié. Un ou-

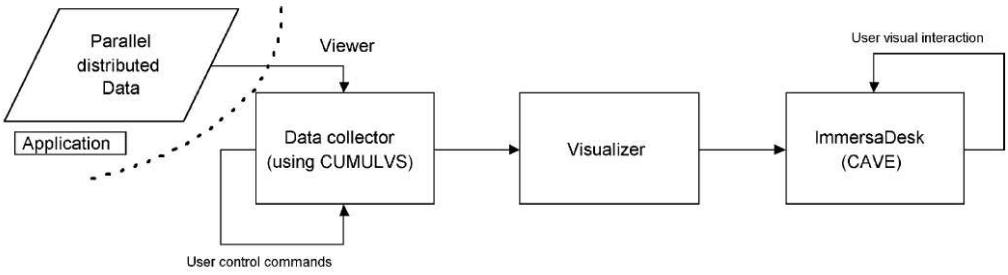


FIG. 3.18 – Pipeline de visualisation de CUMULVS vers le CAVE [213].

til appelé *vtkActorToPF* permet ensuite d'importer l'objet graphique VTK (i.e. *vtkActor*) dans le graphe de scène Performer. Finalement, la scène virtuelle est affichée dans le CAVE grâce à Performer et à la librairie *pfCave* [166]. A l'aide d'un stylet 3D (*wand*), l'utilisateur a la possibilité de manipuler les données (rotation, zoom) dans la scène virtuelle. L'utilisation d'un clavier permet également de configurer l'acquisition des données, par exemple pour changer les limites du domaine ou la fréquence d'envoi. Par ailleurs, un bouton sur le *wand* sert à contrôler manuellement l'acquisition des données. CUMULVS a testé son système sur une simulation d'expansion d'un nuage de gaz (grille structurée $60\times60\times120$). Le rendu volumique n'étant pas supporté par le convertisseur *vtkActorToPF*, la technique de visualisation retenue a été l'iso-surface. Les conversions successives dans le *pipeline* de visualisation pénalisent l'interactivité du système pour le modèle considéré. En particulier, aucune interaction n'est possible lorsque le *pipeline* de visualisation est en cours d'exécution. En conclusion, les auteurs suggèrent d'utiliser le parallélisme afin d'accélérer le rendu.

3.3 Synthèse sur les environnements de pilotage existants

La simulation interactive est un domaine de recherche récent très actif. Au cours des quinze dernières années, beaucoup d'environnements de pilotage sont apparus. Nous avons cité dans la section précédente l'ensemble des travaux qui ont retenu notre attention. Dans cette section, nous revenons plus en détails sur quelques uns de ces environnements, fréquemment cités dans la littérature et représentatifs de l'état de l'art. Parmi ces environnements, nous présentons SCIRun, VIPER, CSE, VASE, Progress & Magellan, DISCOVER, CUMULVS et PAWS. Pour chaque environnement, nous nous efforçons de détailler les caractéristiques principales (intégration, modèle, architecture, système de communication, interface utilisateur, *etc*.) dans les limites des informations disponibles et de notre propre compréhension de ces travaux. Pour plus de détails sur ces environnements, nous renvoyons le lecteur à un article de référence de *Mulder et al.*, intitulé « *A Survey of Computational Steering Environments* » [153], ainsi qu'à l'ensemble des références bibliographiques sélectionnées.

3.3.1 Récapitulatif sur les environnements existants

SCIRun

SCIRun[167, 169, 170, 171] (Scientific Computing and Imaging) est un PSE (Problem Solving Environment) possédant des fonctionnalités de pilotage, et développé à l'université de UTAH à partir de 1995. Il est principalement dédié à l'exploration de modèles et à l'expérimentation algorithmique, avec des exemples d'applications dans des domaines variés tels que la médecine (modules ITK), la géophysique ou la biologie (modules BioPSE). SCIRun se présente comme un environnement de programmation visuelle permettant à un utilisateur de construire, de piloter et d'étudier un large éventail d'applications dans un même temps. Le processus complet de modélisation, simulation et visualisation est entièrement intégré dans le PSE. L'utilisateur construit une simulation pilotable simplement en interconnectant un ensemble de modules (Fig. 3.19). L'ensemble des modules ainsi connectés forme un réseau, qui communiquent selon le paradigme *data flow*. Chaque module se présente sous la forme d'une fonction écrite en C, C++ ou Fortran, à laquelle est associée une interface TCL/TK. Grâce à cette interface, l'utilisateur a la possibilité de modifier la configuration du module ainsi que ses paramètres d'entrée. La modification d'une entrée déclenche automatiquement l'exécution du module et la mise à

jour des sorties, puis du réseau en cascade. Une application dans SCIRun est multi-threads, mais un noyau central coordonne séquentiellement l'exécution des modules. Un tel système permet essentiellement de concevoir de nouvelles applications, par assemblage de modules. L'utilisateur a également la possibilité de construire ses propres modules, sous la forme de classes C++, ce qui permet d'intégrer des applications existantes à condition de les restructurer entièrement en module SCIRun (*recasting*). Par ailleurs, SCIRun offre de nombreux modules de visualisation 2D/3D permettant de collecter les primitives en provenance des autres modules et les afficher dans une fenêtre graphique. En particulier, un module de visualisation, appelé *Salmon*, a la possibilité d'envoyer des messages aux autres modules (*feedback loop*) afin de modifier leurs entrées. Grâce à ce mécanisme, *Salmon* permet de piloter un réseau de modules directement à partir de la visualisation. SCIRun est toujours un projet actif. La version 2 de SCIRun [65] s'oriente vers le modèle de composants logiciels CCA. Dans cette nouvelle approche, les modules sont naturellement remplacés par des composants et la plate-forme SCIRun joue le rôle du *framework* (cf. Sec. 2.2.6)

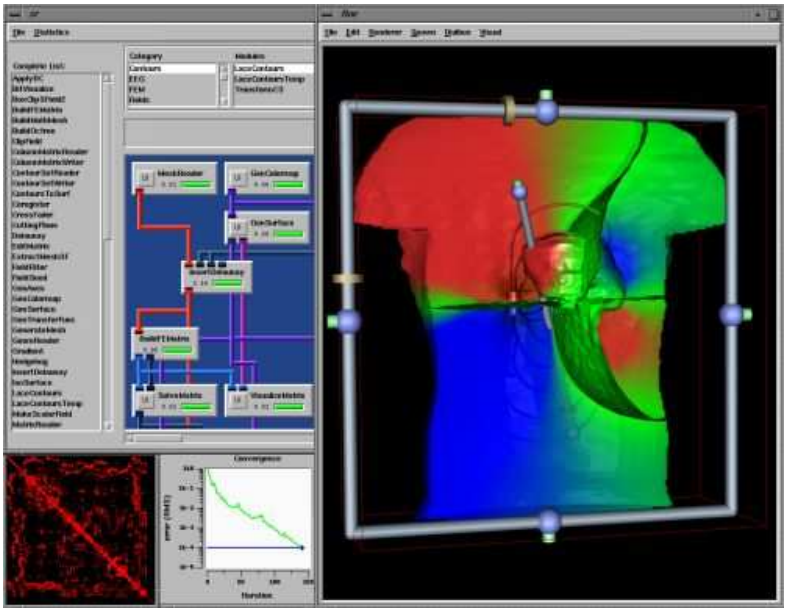


FIG. 3.19 – SCIRun.

VIPER

VIPER (VIsualisation of massively Parallel simulation algorithms for Extended Research) [183] est un environnement de pilotage dédié à l'exploration de modèles reposant sur l'annotation du code source. Il permet de piloter des simulations massivement parallèles (en mémoire partagée ou distribuée) avec des exemples d'applications en dynamique des fluides (CFD). L'architecture de VIPER est basée sur un modèle client/serveur/client, découpé en trois unités logiques : l'unité de calcul (la simulation parallèle), l'unité de visualisation et l'unité de connectivité appelé *serveur dual*. L'unité de calcul et de visualisation sont clientes du serveur dual qui prend en charge l'extraction des données en parallèle et leur transfert vers l'unité de visualisation (mode *on-line*). Si l'unité de visualisation n'est pas connectée, le serveur dual archive les données extraites sous forme de fichiers traces (mode *off-line*). Comme le montre la figure 3.20, le serveur dual est formé de deux serveurs multi-threads, pouvant être distants l'un de l'autre. Les threads permettent d'effectuer le transfert des données en parallèle plutôt que séquentiellement. La communication entre les unités utilise RPC et XDR. A notre connaissance, VIPER ne permet pas de piloter plusieurs applications simultanément, ni même de connecter plusieurs unités de visualisation. L'unité de visualisation est constituée d'une base de données centralisant les données extraites de la simulation et permettant à plusieurs processus de visualisation de se connecter en même temps. Aucune information supplémentaire n'est disponible sur l'interface utilisateur. Le code source de la simulation est annoté pour déclarer les objets de pilotage : paramètres scalaire ou données distribuées de type matrice ou grille. Les objets sont associés dans le code à des points de synchronisation, qui contrôle l'accès aux données. Quand l'exécution de la simulation rencontre un point de synchronisation actif, un signal est envoyé au *dual server* qui

prend en charge l'extraction des données (ou la modification). L'introduction du serveur dual permet de fortement découpler l'unité de calcul et celle de visualisation. Ainsi, VIPER propose 6 modes de communication différents, allant du complètement synchrone au complètement asynchrone. Aucune information n'est fournit sur les mécanismes utilisés pour garantir la cohérence des opérations en parallèle. A notre connaissance, VIPER n'est plus un projet actif.

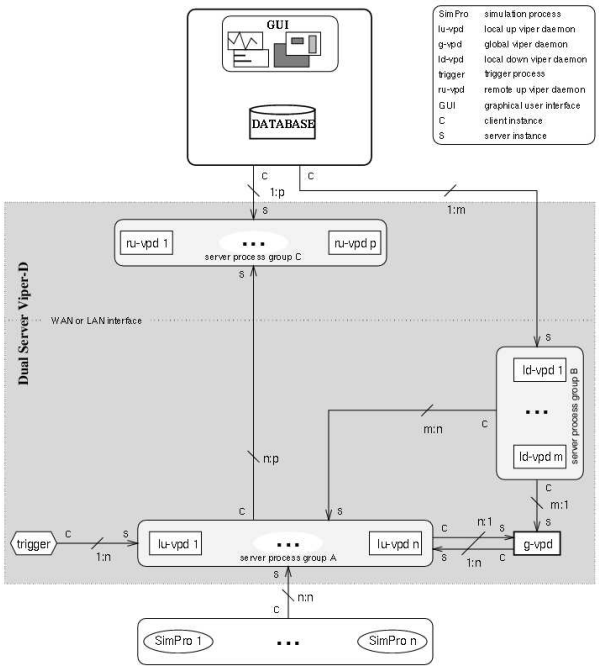


FIG. 3.20 – VIPER.

CSE

CSE (Computational Steering Environment) [201, 202, 203, 204] est un environnement de pilotage, développé au CWI à Amsterdam. Il est principalement dédié à l'exploration de modèles avec des applications dans divers domaines dont une simulation numérique de prédiction du *smog* en Europe. CSE permet l'intégration d'applications numériques par l'annotation du code source. A notre connaissance, aucun résultat n'est publié mentionnant le support de simulations parallèles. L'architecture de CSE repose sur un serveur central, appelé *data manager*, et un ensemble de *satellites* connectés à ce serveur. Comme le montre la figure 3.21, les satellites permettent de représenter la simulation, mais également divers types d'interfaces utilisateurs (PGO, Selection, Calulator). Les processus satellites peuvent être distribués sur différentes machines. Ainsi CSE permet de prendre en compte plusieurs applications et plusieurs UIs simultanément. Le *data manager* héberge une base de données que les satellites utilisent pour s'échanger des données via les primitives de communication *read/write*. De plus, le serveur permet de notifier des événements aux satellites inscrits pour signaler qu'une donnée a été modifiée dans la base. Grâce à ce mécanisme, les satellites peuvent s'échanger des données et ainsi coordonner leurs traitements. Par ailleurs, des variables spéciales (*trigger variable*) dans le *data manager* permettent de synchroniser plusieurs satellites. Afin de minimiser la charge du réseau, le *data manager* est en fait distribué sur l'ensemble des machines satellites. La communication entre le serveur et les satellites utilise TCP/IP. L'intégration d'une simulation implique d'annoter le code source pour déclarer les données au *data manager*. Les données supportées sont des variables scalaires ou des tableaux multi-dimensionnels de type entier, flottant ou chaîne de caractères. CSE dispose de plusieurs UIs intégrées dans l'environnement sous forme de satellites. Parmi ces satellites, on trouve le PGO Editor et le système de visualisation externe IRIS Explorer. En particulier, le PGO Editor [150, 152] permet de construire interactivement une représentation graphique 2D/3D des données. Basé sur la notion d'objet graphique paramétrée (Parametrized Graphic Objects), cet éditeur permet d'associer dynamiquement les variables du *data manager* à des objets graphiques, de telle sorte qu'une modification de la

donnée dans le *data-manager* entraîne une modification de la visualisation et réciproquement (cf. Sec. 3.2.3). Le PGO Editor utilise X11 pour la partie 2D et OpenGL pour la 3D. Il existe également une extension du PGO Editor pour le pilotage dans un CAVE (cf. Sec. 3.2.3). A notre connaissance, CSE n'est plus un projet actif, la dernière date de publication datant de 1998.

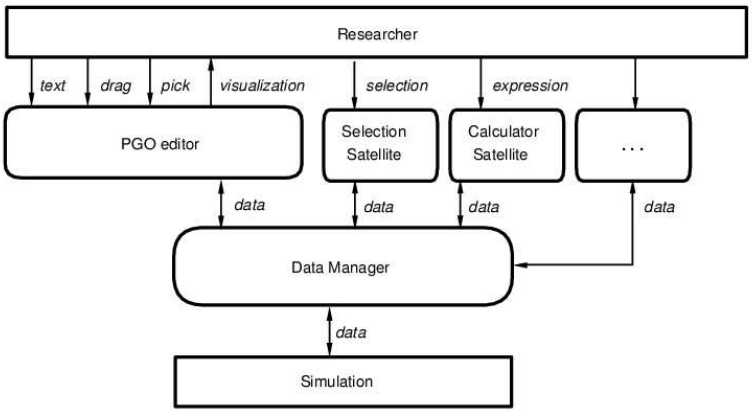


FIG. 3.21 – CSE.

VASE

VASE [101, 113] (Visualization and Application Steering Environment) est l'un des premiers environnement de pilotage (apparu au début des années 90). Il est principalement dédié à l'exploration de modèles et à l'expérimentation algorithmique. L'intégration d'une simulation dans VASE est décomposée en deux étapes. Tout d'abord, l'utilisateur (ou plutôt le programmeur) annote le code source pour déterminer un modèle haut-niveau de l'application, basé sur un *control-flow graph* (CFG). A partir de cette description, l'utilisateur final construit l'application pilotable sans avoir à manipuler le code source, grâce à un outil de configuration. Plusieurs applications peuvent être ainsi connectées simultanément dans l'environnement. A notre connaissance, VASE ne permet de piloter que des simulations séquentielles en Fortran. Dans VASE, la simulation est modélisée par un graphe constitué de blocs de code et d'arcs représentant le flux de contrôle entre ces blocs. L'utilisateur peut insérer des points d'arrêt sur les arcs pour contrôler le flux d'exécution du programme et interagir avec via un langage script proche du C. Ces scripts permettent de lire ou d'écrire des variables de l'application, de contrôler le flux des données entre les applications ou encore d'exécuter à distance des routines de l'application. La communication entre les processus utilise un mécanisme hybride *control-flow-data-flow* basée sur la bibliothèque DTM (Data Transfer Mechanism) qui autorise le transfert des données dans un environnement distribué et hétérogène. Basiquement, les données sont transférées entre un port d'envoi (*input port*) et un port de réception (*output port*), dont la connexion forme ce qu'on appelle un *channel*. Les ports de communication peuvent être associés à un point d'arrêt ou non. Dans le premier cas, le transfert intervient uniquement sur le point ; dans le cas contraire, l'accès aux données est totalement asynchrone. Les données considérées dans VASE sont des scalaires ou des tableaux. VASE propose une interface graphique permettant de configurer et de déployer les applications préalablement instrumentées (Fig. 3.22). Plus précisément, cette interface permet de visualiser le CFG, de spécifier les machines cibles pour le déploiement, de configurer les points d'arrêt pour qu'ils exécutent des scripts, envoient ou reçoivent des données. La visualisation des données repose sur l'utilisation de programmes externes comme IRIS Explorer.

Progress & Magellan

Progress [206] (PROGram and REssource Steering System) a été développé au Georgia Intitute of Technology. Il est dédié à l'exploration de modèles et à l'optimisation des performances. Progress permet d'intégrer des applications parallèles multi-threads en annotant le code source. Progress est l'un des tous premiers environnements à utiliser une abstraction en terme d'objet de pilotage (*steering object model*). Ces objets fournissent

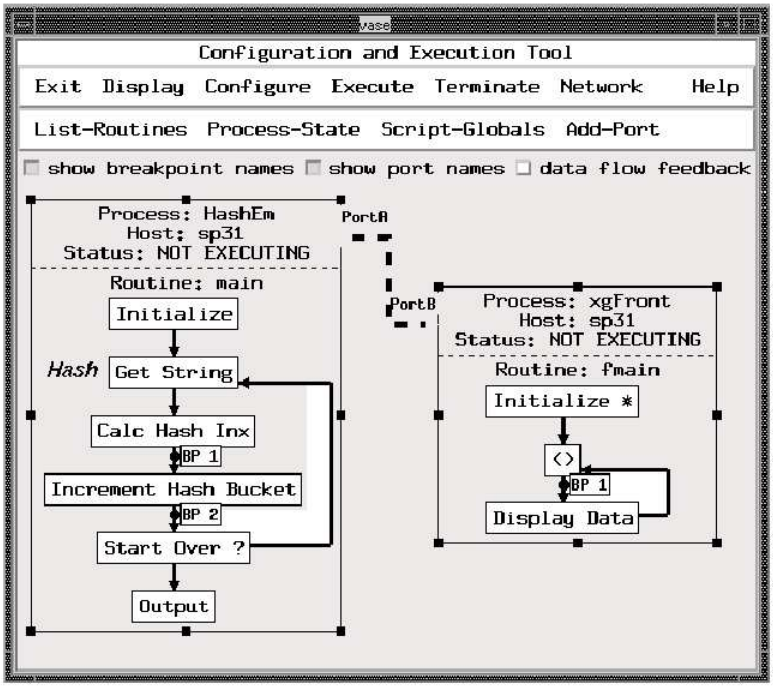


FIG. 3.22 – VASE.

un mécanisme simple permettant d'encapsuler certaines composantes de l'application, de les identifier et de les manipuler. Les objets dans Progress sont essentiellement des données scalaires, que l'utilisateur déclare en annotant le code source. Les opérations disponibles sur ces objets sont l'accès en lecture/écriture asynchrone (*probe read & write*) et synchrone (*sense & actuate*). Les accès synchrones nécessitent d'instrumenter l'application pour signaler précisément les points dans le code où les données seront effectivement consultables et/ou modifiables. Des objets spécialisés permettent de contrôler le flot d'exécution de l'application à l'aide de points de synchronisation (*synch points*), d'exécuter des fonctions déclarées dans le code de l'application (*callback*) ou encore d'exécuter des scripts (côté client) permettant de combiner et de répéter plusieurs opérations. Progress utilise une architecture client/serveur simple. Le serveur s'exécute dans un thread, placé dans le même espace mémoire que l'application. Le client s'exécute typiquement sur une machine distante et joue le rôle de l'interface utilisateur. Client et serveur dialoguent de manière bi-directionnelle via des sockets. Comme le montre la figure 3.23, le serveur possède un registre des objets (*object registry*) déclarés par l'utilisateur. Ce registre est dupliqué chez le client et contient les informations nécessaires pour manipuler les objets depuis l'interface utilisateur. Progress ne fournit aucun support particulier pour le pilotage de plusieurs applications et ne permet pas, à notre connaissance, de connecter simultanément plusieurs utilisateurs. L'interface utilisateur est une application graphique écrite en Motif qui permet de consulter le registre des objets et de réaliser les diverses opérations possibles (consultation d'une valeur). Les données scalaires sont présentées textuellement ou sous forme d'un simple graphique 1D.

Magellan [205, 208, 209] est désigné comme le successeur de Progress. Contrairement à Progress, plusieurs clients peuvent dans Magellan piloter simultanément plusieurs applications. Chaque application est instrumentée de manière comparable à Progress, mais un serveur central (*steering server*) sert à connecter tous les serveurs locaux (Fig 3.24). Chaque processus instrumenté est vu séparément par le serveur central et il n'y a pas distinction faite *a priori* entre le pilotage d'une application parallèle et le pilotage de plusieurs applications. Magellan utilise le même mécanisme de communication par évènement que Progress, mais propose une optimisation intéressante basée sur un langage interprété appelé ACSL (Advanced Computational Steering Language). Ce langage permet à l'utilisateur de programmer dynamiquement des actions de pilotage grâce à l'écriture de règle de la forme : *when <event expression> do action*. Par ailleurs, Magellan a la possibilité de transférer les requêtes de pilotage en ACSL aux serveurs locaux, pour qu'elles y soient interprétées, ce qui permet de limiter le transfert des évènements vers le serveur central (*spatial-locality optimization*). A titre d'exemple, Magellan utilise son sys-

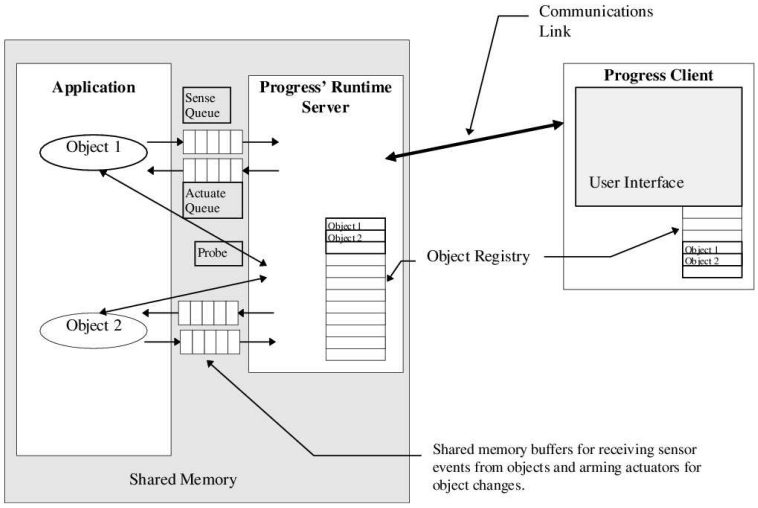


FIG. 3.23 – Progress.

tème pour commander un équilibrage de charge dynamique dans une simulation multi-threads. Ainsi lorsque la charge d'un thread devient inférieur à un certain seuil, son domaine est automatiquement agrandi. Dans ce cas, la commande de pilotage testant le seuil et modifiant les limites du domaine s'exécute localement sur chaque thread, sans qu'il soit nécessaire de communiquer avec le serveur central. Les données supportées sont des scalaires et des tableaux 1D. L'interface utilisateur proposée est un interpréteur de commande en mode texte.

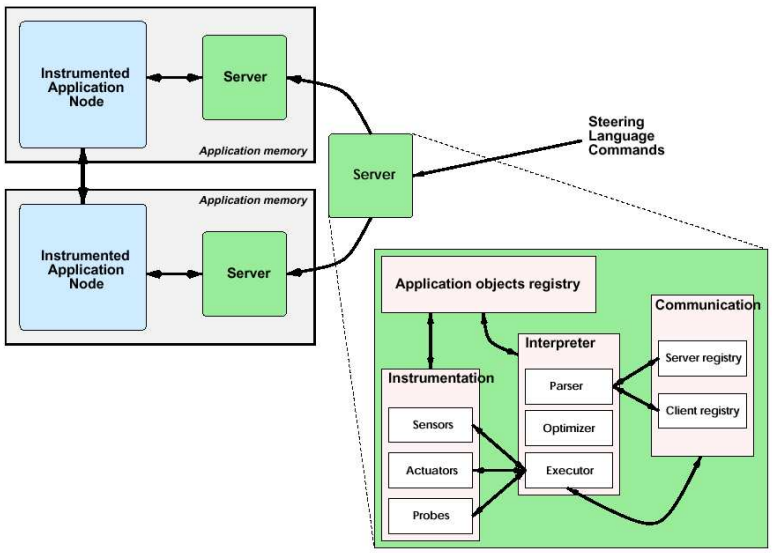


FIG. 3.24 – Magellan.

DISCOVER

DISCOVER [154, 133, 155] est environnement de pilotage, orienté *web*. Il permet à plusieurs utilisateurs de piloter simultanément plusieurs applications grâce à un simple navigateur *web*. L'intégration d'une application dans DISCOVER nécessite d'instrumenter le code source. DISCOVER est principalement dédié au pilotage de codes orientés objets (Java ou C++), mais il est également possible d'intégrer des codes non-objets (C, Fortran) grâce à l'utilisation de *wrapper* C++. Cette approche a été validée sur une simulation parallèle en géophysique, écrite en Fortran et utilisant NetSolve pour distribuer les calculs sur un grille (IPARS [15]). Dans DISCOVER,

l'instrumentation a pour but de transformer les objets de calcul en objets d'interaction. Pour cela, l'objet de calcul doit hériter d'une certaine classe de base et déclarer les méthodes de l'objet utiles au pilotage. L'interface d'un objet d'interaction est décrite en IDL ; elle se compose d'accesseurs (*views*) et de modificateurs (*commands*) permettant d'interagir à distance avec l'objet de calcul. Un objet d'interaction peut être localisé sur un seul nœud de calcul ou distribué sur tout ou partie des nœuds. De plus, au cours de l'exécution, les objets d'interaction peuvent être créés, détruits, ou peuvent encore migrer vers un autre nœud. Dans le cas d'un objet distribué, deux opérations clés, *gather* et *scatter*, permettent à l'utilisateur de spécifier la distribution des données. L'architecture de DISCOVER (Fig.3.25) est basée sur un serveur *web* central, auxquels les utilisateurs se connectent avec des navigateurs *web* classiques (*thin client*). Ce serveur se compose d'un ensemble de *servlets* Java, fournissant les divers services du système (session, archivage, authentification, interaction, visualisation, *etc*.). Les clients dialoguent avec le serveur *web* via le protocole HTTP (requêtes GET & PUT). Sur chaque nœud de l'application, un agent d'interaction (serveur local) est responsable d'exporter l'interface IDL des objets d'interaction. Ces objets se présentent alors comme des objets distribués standards, accessibles depuis le serveur *web* par le protocole IIOP (Java RMI ou CORBA). Les servlets sont responsables de générer dynamiquement des pages HTML présentant les informations associées aux vues des objets. Grâce à l'utilisation de formulaires HTML, l'utilisateur peut poster des requêtes de pilotage qui seront traduits par le serveur en invocation de méthode distantes sur les objets d'interaction. La visualisation 2D/3D des données s'effectue grâce des *applets* Java, directement intégrées dans la page *web*. DISCOVER dispose également de mécanismes de collaboration (cf. Sec. 3.2.2). Au sein d'un groupe d'utilisateurs, un « maître » pilote l'application et tous les utilisateurs membres du groupe reçoivent une vue synchronisée de l'application.

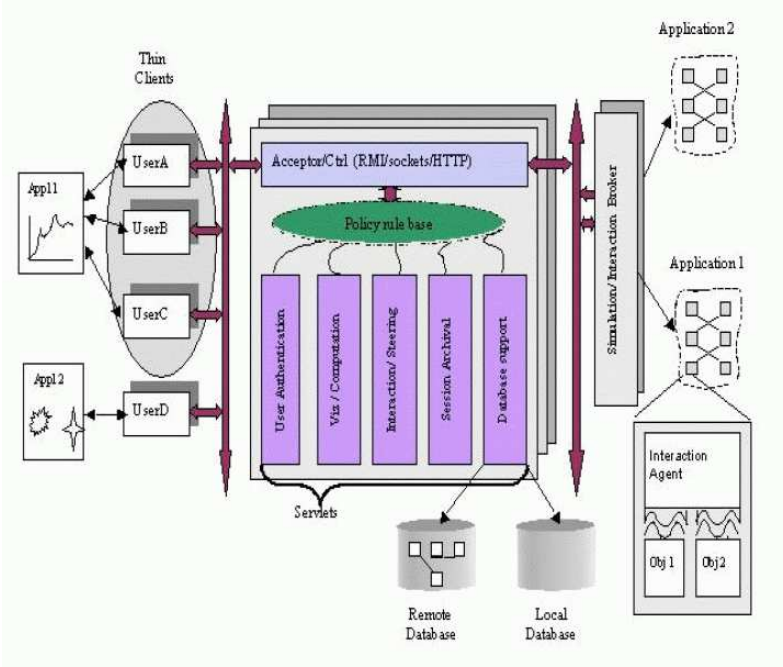


FIG. 3.25 – DISCOVER.

CUMULVS

CUMULVS [122] (Collaborative User Migration, User Library for Visualization and Steering) est un environnement de pilotage basé sur PVM et développé à l'ORNL (Oak Ridge National Laboratory). Dans sa version initiale, CUMULVS se limitait au pilotage de codes PVM avec AVS [2]. Il a par la suite évolué vers une solution plus générique permettant de piloter tout type d'applications parallèles (PVM, MPI, *etc*.). Il est principalement dédié à l'exploration de modèles et à la visualisation en ligne. Plusieurs applications en mécanique des fluides et en sismologie illustrent les possibilités de CUMULVS. Dans CUMULVS, la simulation est vue comme une simple boucle de calcul (SPMD), contrôlée par un point d'instrumentation unique (fonction *sendToFE()*). L'utilisateur déclare sur chaque processus la distribution des données et leur emplacement en mémoire. Les données

supportées dans CUMULVS sont des tableaux denses multidimensionnels distribuées en bloc-cyclique, selon un formalisme emprunté à HPF. Nous détaillons ce modèle dans la section 3.2.1. CUMULVS repose sur une architecture client/serveur simple, où la simulation joue le rôle du serveur et l'interface utilisateur celui du client (*viewer*). Plusieurs utilisateurs peuvent se connecter simultanément à une même application (Fig. 3.26). Chaque utilisateur a la possibilité de requérir un sous-domaine des données à une certaine fréquence (*frame rate*). Il peut également modifier la valeur d'un paramètre scalaire de la simulation. Un mécanisme de jeton (*token*) permet à un utilisateur de verrouiller l'accès à un paramètre afin d'éviter des conflits en cas de modifications concurrentes. A chaque itération, la simulation passe la main à l'environnement de pilotage qui échange avec les clients toutes les informations nécessaires. La communication repose entièrement sur la bibliothèque PVM. Afin de coordonner efficacement les opérations de pilotage en parallèle, CUMULVS utilise une stratégie n'impliquant peu ou pas de synchronisation. Nous renvoyons le lecteur à la section 3.2.2 pour plus de détails sur cette stratégie, appelée *loose synchronization*. CUMULVS se distingue des autres environnements en fournissant un mécanisme de tolérance aux fautes basé sur l'utilisation de points de reprise (*checkpoint*). Les points de reprise sont des points d'instrumentation particuliers permettant de sauvegarder un certain état de la simulation, puis de le restaurer ultérieurement en cas d'erreur. L'utilisateur doit spécifier explicitement l'ensemble des variables à sauvegarder. Ce mécanisme complète avantageusement les fonctionnalités déjà offertes par un environnement de pilotage, dans le sens où il permet à l'utilisateur d'expérimenter plusieurs jeux de paramètres et de restaurer un état précédent en cas d'échec. L'interface utilisateur est construite sur une API *front-end*. CUMULVS propose une interface de pilotage très générique basée sur le système de visualisation AVS. Plus récemment des travaux [213] ont permis d'intégrer la visualisation dans un CAVE grâce à une solution basée sur les technologies VTK et Performer (cf. Sec. 3.2.3). CUMULVS est un projet particulièrement actif, évoluant actuellement dans le contexte des composants CCA. En particulier, CUMULVS participe au sein du projet CCA M×N à la définition d'un composant parallèle de redistribution.

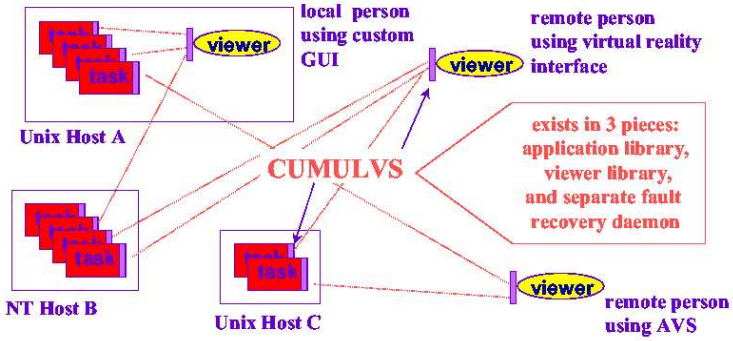


FIG. 3.26 – CUMULVS.

PAWS

PAWS (Parallel Application WorkSpace) [44] est une plate-forme de couplage de codes, permettant de redistribuer des tableaux de données entre deux codes parallèles. PAWS n'est pas à proprement parler un environnement de pilotage, mais offrent des fonctionnalités suffisamment puissante pour permettre de coupler une application de calcul parallèle avec un code de visualisation, éventuellement parallèle (Fig. 3.27). L'API de PAWS permet de spécifier la distribution des données (*layout*) et leur stockage en mémoire (*actual data*). Les applications couplées peuvent posséder des nombres de processeurs différents, utiliser différents types de distribution et être écrits dans différents langages de programmation (C, C++, F77). Les données supportées dans PAWS sont des scalaires et des tableaux denses multidimensionnels distribués en blocs rectilinéaires. Pour plus de détails, nous renvoyons le lecteur à la section 3.2.1. PAWS utilise une abstraction en terme de composant pour représenter les applications parallèles. Chaque composant est constitué d'un ensemble de ports auxquels sont associés la description des données distribuées. Les ports sont dédiés à la communication ; ils servent à connecter les données entre les composants, selon une approche inspirée du modèle *data-flow*. La connexion entre les ports est unidirectionnel, l'un des ports jouant le rôle d'émetteur et l'autre de receveur. Le même port peut être utilisé pour envoyer (ou recevoir) plusieurs données différentes, à condition qu'il partage la même distribution.

L'architecture de PAWS repose sur l'utilisation d'un contrôleur central (*PAWS Controller*). Ce contrôleur tient à jour la liste des applications, des ports et des données. Par ailleurs, il permet de connecter et de déconnecter dynamiquement des applications parallèles durant leur exécution, grâce à une interface script en TCL. Les fonctionnalités du contrôleur sont également accessibles directement à partir de l'API. A la connexion des ports, le contrôleur calcule le schéma de communication entre les composants à partir des informations de distribution (*layouts*). Lors des étapes de communication (*send/recv*), les données sont automatiquement redistribuées d'un code à l'autre, chaque processus n'envoyant ou ne recevant que les données qui lui sont nécessaires. Le transfert s'effectue en parallèle entre les applications couplées, afin d'éviter le goulot d'étranglement provoqué par la sérialisation des messages. Chaque message est envoyé en point-à-point grâce à la bibliothèque Nexus. PAWS distingue plusieurs modes de communication : synchrone, asynchrone ou partiellement synchrone. Dans ce dernier cas, le transfert s'effectue (de manière synchrone) uniquement si la durée du rendez-vous n'excède pas un certain *timeout* défini par l'utilisateur. A notre connaissance, PAWS n'est plus un projet actif, mais il continue d'évoluer au sein du projet CCA M×N [118], où un prototype de composant M×N (basé sur PAWS) a été intégré dans le *framework* CCAFFEINE (cf. Sec. 2.2.6).

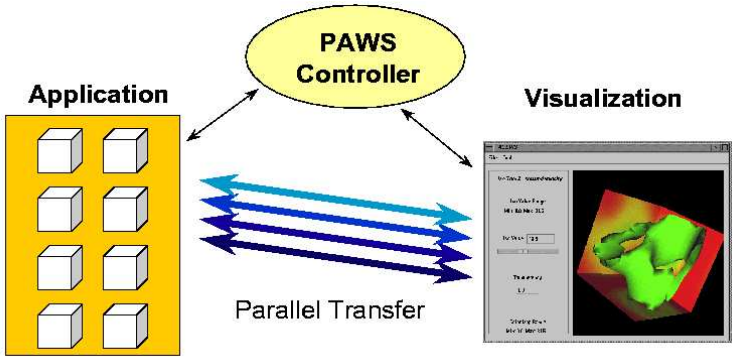


FIG. 3.27 – PAWS.

3.3.2 Discussion

Intégration des simulations parallèles

La plupart des environnements permettent de piloter des simulations parallèles (Tab. 3.1, colonne « Simulation »). Pour les premiers environnements comme Progress, il s'agissait exclusivement d'applications multi-threads. Par la suite, des solutions sont apparues pour le pilotage d'applications parallèles en mémoire distribuée. Tous ces environnements n'abordent pas la problématique au même niveau. En effet, le pilotage de ces simulations posent des difficultés supplémentaires liées à la prise en charge de la distribution des données et de la coordination des traitements en parallèle. Dans ce domaine, seuls les environnements CUMULVS, DAQV, VIPER ou DISCOVER proposent des solutions convaincantes à ces problèmes. Cependant, aucun des projets cités jusqu'ici n'aborde la cas des simulations parallèles distribuées, impliquant plusieurs codes de calculs couplés (e.g. multi-physiques). Notre sentiment est qu'il reste beaucoup de travaux à mener dans le domaine du pilotage pour parvenir à représenter des codes parallèles et parallèles distribués et à les piloter efficacement sur une grille.

Du point de vue de la simplicité d'utilisation et de la flexibilité, les PSEs (SCIRun, COVISE) sont incontestablement la solution idéale. Grâce à une approche « tout intégrée », l'utilisateur modélise, calcule, visualise et interagit dans le même temps. Toutefois, même s'il est possible d'enrichir l'environnement avec de nouveaux modules, les PSEs sont mal adaptés pour intégrer des simulations existantes. Dans ce cas, il est souvent nécessaire de restructurer tout ou partie du code, ce qui peut s'avérer rédhibitoire pour des scientifiques qui ont déjà réalisé des efforts considérables pour mettre au point leurs simulations. De plus, les PSEs actuels ne permettent pas de prendre en compte le parallélisme de données. Notons cependant que certains travaux autour des composants logiciels dans CCA, ont récemment proposé une version parallèle du PSE SCIRun (version 2) basé sur le concept de PRMI (Parallel Remote Method Invocation) [65]. A notre sens,

une solution basée sur l'instrumentation des sources de la simulation s'avère plus appropriée pour l'intégration de simulations parallèles « en vraie grandeur », car elle ne remet pas en cause la structure générale du code. De plus, l'utilisateur est libre d'annoter uniquement les parties du code source qu'il juge intéressantes pour sa problématique, ce qui offre une grande liberté pour l'intégration mais nécessite une bonne expertise du code.

L'intégration d'une simulation dans un environnement de pilotage repose généralement sur une représentation abstraite de la simulation (Tab. 3.1, colonne « Abstraction »). Outre les PSEs qui utilise un abstraction en modules, la plupart des environnements de pilotage se limitent à une représentation de la simulation en points d'instrumentation (points d'arrêt, points d'accès aux données, *etc*). Dans CUMULVS, la simulation est vue comme une simple boucle de calcul SPMD. Cette abstraction peut s'avérer suffisante dans certains cas mais reste très limitée si l'on souhaite piloter *finement* des simulations complexes. D'autres environnements comme VASE [113] ou PathFinder [143] utilisent des abstractions de plus haut-niveau. Dans ces environnements, l'annotation des sources vise à extraire un squelette reflétant la structure essentielle du code. Dans VASE par exemple, la simulation est représentée par un graphe de tâches. PathFinder utilise la notion de transaction, des tâches pouvant être distribuées, pour représenter la simulation. Ces modèles facilitent le repérage dans l'évolution de la simulation. En revanche, les interactions se produisent uniquement sur les points d'instrumentation servant à marquer le début ou la fin des tâches. Dans MOSS et DISCOVER, la simulation est vue comme un ensemble d'objets distribués, pilotés par des invocations de méthodes à distance (RMI). Le point fort de cette approche réside dans la liberté qui est laissée aux utilisateurs pour définir l'interface des objets qu'ils pilotent. Toutefois, cette approche est relativement complexe à mettre en œuvre pour l'utilisateur final et s'applique difficilement aux langages de programmation non-orientés objets.

Architecture, communication et performance

Du point de vue de l'architecture (Tab. 3.1, colonnes « Architecture » et « Communication »), nous avons principalement trouvé deux approches dans la littérature : les environnements utilisant un modèle client/serveur (C/S) simple[1] et ceux utilisant un modèle client/serveur/client (C/S/C). L'avantage de l'utilisation d'un serveur central est qu'il permet de découpler fortement la simulation de l'interface utilisateur. Dans ce cas, la simulation communique uniquement avec le serveur central, et non avec tous les clients comme dans le modèle C/S, ce qui limite les perturbations de la simulation en mode multi-utilisateurs. Par ailleurs, l'archivage des données sur un serveur central permet par exemple dans VIPER de sauvegarder l'historique de la simulation, pour éventuellement effectuer une visualisation *off-line*. En contrepartie, le stockage intermédiaire des données entraîne un goulot d'étranglement (i.e. *gather*), qui pénalise fortement les transferts et donc l'interactivité globale du système de pilotage. A l'inverse, dans les architectures C/S simples, comme celle de CUMULVS, le transfert des données se fait directement entre la simulation et l'UI, ce qui permet d'obtenir de meilleures performances notamment dans le contexte de la visualisation en ligne. VASE ou PAWS utilisent une architecture orientée couplage quelque peu différente des architectures précédentes. Dans ces environnements, les codes couplés utilisent un tiers programme (s'apparentant à un serveur central) pour se contacter et partager des informations de haut-niveau. En revanche le transfert se fait directement entre les codes couplés, comme dans le modèle C/S simple. Ainsi, les codes couplés jouent des rôles symétriques dans ces environnements, ce qui n'est pas tout à fait adapté à la problématique du pilotage, dans la mesure où les interactions entre les codes va principalement se limiter à des échanges de données.

Pour être performant, l'environnement de pilotage doit être peu intrusif et perturber le moins possible le déroulement de la simulation. En particulier, le surcoût doit être minimal en l'absence d'interaction de pilotage. Dans la plupart des environnements orientés flux d'évènements (*event stream*) comme Progress, Magellan ou Falcon, des informations (les évènements) sont continuellement transmises à un serveur central, même en l'absence de clients connectés, ce qui perturbe fortement la simulation. D'autres environnements comme DAQV, CUMULVS, MOSS ou DISCOVER utilisent un modèle de pilotage beaucoup plus économique, basé sur un système de requêtes dirigé par le client. Dans cette approche, les perturbations n'auront lieu *a priori* qu'au moment des requêtes clientes. Un autre critère conditionnant la performance d'un environnement de pilotage

[1] Remarquons que l'interface utilisateur est généralement assimilée au client et la simulation au serveur, même si les rôles peuvent s'inverser au cours de la session de pilotage.

est la réactivité, c'est-à-dire le temps qu'il est nécessaire au système avant de pouvoir répondre à une requête cliente. Dans CUMULVS ou DAQV par exemple, il faut systématiquement attendre que le flot d'exécution de la simulation rencontre un nouveau point d'instrumentation pour que l'environnement de pilotage puisse « prendre la main » et débuter le traitement d'une requête de pilotage, ce qui entraîne une mauvaise réactivité du système. Dans d'autres environnements comme Falcon, la réception des requêtes et l'initialisation des traitements de pilotage est déporté dans un *thread*, ce qui permet d'introduire de l'asynchronisme dans le système de pilotage et donc d'offrir une meilleur réactivité. Du point de vue de l'accès aux données, la plupart des environnements proposent uniquement un accès synchrone et bloquant sur des points d'instrumentation. A notre connaissance, seul Progress & Magellan autorisent un accès asynchrone aux données mais sans garantie de cohérence des données extraites, ce qui limite considérablement l'utilisation d'un tel mécanisme. Dans CUMULVS, une copie des données dans un cache à l'envoi permet également d'introduire de l'asynchronisme dans les transferts tout en garantissant la cohérence, mais entraîne un surcoût de consommation mémoire et des latences non négligeables, en particulier dans le contexte de la visualisation en ligne.

Les premiers environnements de pilotage apparus ont développé leur couche de communication sur des protocoles de bas-niveau comme TCP/IP. Ces environnements doivent ainsi prendre en charge un ensemble de difficultés liés à l'hétérogénéité des réseaux (encodage, alignement, *etc.*), ce qui a donné naissance le plus souvent à des bibliothèques de communication « maison » (NCSA DTM pour VASE, DataExchange pour Falcon, Progress et Magellan, *etc.*). De ce point de vue, il est intéressant de noter que les environnements de pilotage plus récents comme MOSS, DISCOVER ou RealityGrid se sont orientés vers des couches de communication hybrides mixant deux technologies : une technologie de haut-niveau pour la gestion d'un système de requêtes (CORBA, RMI, SOAP) et une technologie de plus bas-niveau pour la réalisation des transferts. Pour terminer, on peut souligner le cas particulier de l'environnement CUMULVS qui utilise les possibilités de la bibliothèque parallèle PVM pour définir sa propre couche de communication.

Coordination efficace des interactions

L'efficacité d'un environnement de pilotage dépend en partie de sa capacité à coordonner des interactions entre les processus de la simulation et de l'interface utilisateur tout en garantissant la cohérence des traitements effectués. Notons que ce problème, pourtant crucial, est rarement pris en compte par les environnements de pilotage existants. Généralement, ces environnements contournent le problème en laissant le soin à l'utilisateur de maintenir la cohérence explicitement. Parmi les stratégies existantes que nous avons étudiées dans ce chapitre (cf. Sec. 3.2.2), nous avons constaté deux défauts importants. Premièrement : la coordination impose une forte synchronisation de tous les processus de la simulation et donc un surcoût important même lorsqu'il n'y a pas de client connecté (e.g. DAQV). Deuxièmement : la coordination nécessite une centralisation systématique des informations sur un serveur et un post-traitement de ces informations (Falcon, PathFinder, Magellan). Cette dernière stratégie évite le premier défaut, car chaque processus de la simulation envoie ses données de manière indépendante, sans consulter les autres processus (i.e. pas de synchronisation). En revanche, cette technique est fortement pénalisée d'une part par les envois systématiques de données, et d'autre part par le stockage intermédiaire des informations, nécessaire pour reconstruire une information globalement cohérente en temps. Seule la stratégie proposée par CUMULVS, baptisée « *loose synchronization* », a la possibilité d'offrir de bonnes performances, en évitant ces deux défauts. Cependant, cette stratégie ne s'applique que dans le cas simplifié d'une simulation parallèle contrôlé par un point d'instrumentation unique (positionné à l'intérieur d'une boucle de calcul). Pour conclure, il faut souligner que la coordination des traitements de pilotage dépend directement de l'abstraction qui est utilisée par l'environnement de pilotage pour représenter la simulation et suivre son évolution dans le temps. Dans ce domaine, il serait intéressant de pouvoir mettre en place un algorithme de coordination permettant de garantir la cohérence temporelle des traitements tout en évitant d'avoir recours à un mécanisme de synchronisation systématique et coûteux.

Interface utilisateur, visualisation et interaction

Du point de vue de l'interface utilisateur (Tab. 3.1, colonne « Interface Utilisateur »), tous les environnements ne sont pas équivalents. Certains comme Magellan se contentent de fournir une interface en ligne de

Environnement	M.-U.	M.-A.	Simulation	Abstraction	Archictecture	Communication	Interface Utilisateur
SCIRun			séquentielle	modules	PSE	data-flow	modules intégrés (Salmon)
COVISE	X		séquentielle	modules	PSE	data-flow	modules intégrés
VASE	X	X	séquentielle	graphe de tâches	couplage	data channels (direct)	GUI (graphe de tâches)
CSE	X	X	séquentielle	points d'instrum.	C/S/C	base de données	PGO, IRIS Explorer
Falcon			multi-threads	points d'instrum.	C/S/C	event stream (centralisé)	GUI (monitoring threads)
Progress			multi-threads	points d'instrum.	C/S	event stream (centralisé)	GUI
Magellan	X	X	multi-threads	points d'instrum.	C/S/C	event stream (centralisé)	interpréteur ACSL
PathFinder	X	X	SPMD	transaction	C/S/C	event stream (centralisé)	
VIPER	X		SPMD	points d'instrum.	C/S/C	RPC + base de données	GUI
DAQV	X	X	SPMD	points d'instrum.	C/S/C	probe/mutate (centralisé)	GUI, MathLab
CUMULVS	X	X	SPMD	boucle de calcul	C/S	send/recv (direct)	AVS, VTK/Performer
MOSS	X	X	SPMD	objets distribués	C/S	requêtes + event stream (direct)	
DISCOVER	X	X	SPMD	objets distribués	C/S/C	requêtes + transfert centralisé	HTML, applet Java3D
PAWS	X	X	SPMD	composants	couplage	send/recv (flux parallèles)	interpréteur TCL
EPSN	X	X	SPMD	MHT	C/S	requêtes + flux parallèles	VTK parallèle

TAB. 3.1 – Synthèse des environnements de pilotage (*M.-U. = Multi-Utilisateurs ; M.-A. = Multi-Applications*) et positionnement de la plate-forme EPSN.

commandes pour piloter la simulation, alors que d'autres comme VASE, VIPER ou DAQV proposent des interfaces graphiques (GUI) plus conviviales dont certaines intègrent des outils de visualisation. La visualisation 2D/3D nous apparaît comme un outil indispensable à l'analyse et à la compréhension des simulations. Dans ce domaine, le PGO Editor (Parametrized Graphics Object) de CSE est une solution intéressante qui permet d'interagir intuitivement par manipulation directe d'un *widget*. Toutefois si nous reconnaissons le bénéfice d'une telle approche, l'utilisation du PGO Editor reste limitée au seul environnement CSE et ne permet de réaliser que des visualisations relativement simples pour un coût en développement relativement important. Dans ce contexte, il nous semble plus approprié d'utiliser des outils de visualisation « standards » plutôt que de re-développer une solution dédiée à un environnement particulier. En effet, la possibilité d'intégrer un ou plusieurs systèmes de visualisation professionnels dans un environnement de pilotage est un atout important pour l'utilisateur qui bénéficie de tout le savoir-faire propre à ce système. Ainsi, à l'instar de CUMULVS, la plupart des environnements ont choisi de reposer sur des systèmes de visualisation externes comme AVS ou IRIS Explorer, proposant des algorithmes pour le traitement de l'information et la visualisation scientifique.

A notre connaissance, il n'existe pas aujourd'hui d'environnements de pilotage prenant en compte des codes de visualisation qui soient parallèles (en mémoire distribuée). Ce dernier point nous semble une limitation importante des environnements existants si l'on considère l'importance du volume de données à traiter et la complexité de certains algorithmes de visualisation et de rendu (e.g. iso-surfaces, rendu volumique, *etc.*). Seul CAVEStudy et CUMULVS relatent des expériences de rendu parallèle (en *multi-pipes*[2]) basé sur la bibliothèque SGI Performer [29]. Beaucoup d'efforts restent à faire aujourd'hui pour que des environnements de pilotage puissent exploiter des systèmes de visualisation immersifs et plus largement des équipements de réalité virtuelle. Dans ce domaine, la tendance actuelle est clairement à l'utilisation de clusters de PCs (plutôt qu'à l'utilisation de machines parallèles à mémoire partagée, beaucoup plus chers), ce qui nécessite de concevoir des environnements de pilotage d'une nouvelle génération capables d'exploiter les capacités de ces installations.

3.4 Objectifs et positionnement de notre étude, apports et concepts fondamentaux

Tentons pour conclure de positionner notre étude concernant l'environnement EPSN. D'une manière générale, EPSN s'oriente vers le pilotage de simulations parallèles SPMD (en mémoire distribuée) et vers la visualisation en ligne des résultats intermédiaires. A ce titre, l'environnement EPSN se positionne comme un environnement pour l'exploration des modèles numériques utilisés dans les simulations. Dans ce contexte, nous pensons que l'utilisation d'environnements immersifs comme des murs d'images ou des CAVEs peut largement bénéficier aux scientifiques – comme tend à le démontrer les expériences décrites dans la section 3.2.3. Toutefois, l'utilisation de « grands écrans » implique de hautes résolutions d'image et donc des temps de calcul accrus. Par ailleurs, la visualisation de simulations complexes en vraie grandeur nécessite de traiter de gros volumes de données qui peuvent largement dépasser les capacités d'une simple station graphique. Partant de ce constat, nous pensons que l'utilisation de la visualisation parallèle et du rendu parallèle pourrait contribuer à améliorer l'interactivité globale du système de pilotage, sinon de la rendre possible. Aucun des environnements de pilotage que nous avons cités jusqu'à présent ne semble avoir exploré cette voie. Afin de permettre la visualisation et l'interaction à des fréquences soutenues entre un code de simulation parallèle et un code de visualisation également parallèle, il s'avère nécessaire de mettre en place un couplage flexible, le plus performant possible. Le positionnement de l'environnement EPSN par rapport aux autres environnements est synthétisé à la dernière ligne du tableau 3.1, dont nous allons maitenant examiner plus en détails les apports et les concepts fondamentaux.

3.4.1 Le pilotage, un problème de couplage ?

Dans cette thèse, nous abordons le problème du pilotage comme un problème de couplage entre deux codes parallèles : le code de simulation numérique d'une part et le code de visualisation incluant l'interface utilisateur d'autre part. La simulation s'exécute typiquement dans un centre de calcul distant où elle peut mobiliser une

[2]Multi-pipes : plusieurs sorties graphiques sur une même machine (e.g. SGI Onyx).

quantité importante de ressources (machines parallèles, grappes de PCs, grappes de grappes, *etc*.) alors que le code de visualisation utilise le plus souvent des ressources locales allant de la simple station graphique à des environnements immersifs basés sur des machines parallèles de type SGI ou des clusters de PCs équipés de périphériques 3D. En effet, les ressources dédiées au calcul ne sont généralement pas adaptées à la visualisation (pas de carte graphique 3D) et ne supportent pas de sessions interactives (utilisation d'un *batch scheduler*). Par conséquent, il est souvent préférable, sinon nécessaire, de déporter la visualisation et l'interaction sur des machines locales mieux adaptées. Ainsi en distribuant ces deux codes, il est possible de mobiliser davantage de ressources et/ou d'utiliser les ressources les mieux adaptées à chaque code. Par ailleurs, le découplage de la simulation et de la visualisation offre une meilleure structuration de l'application globale, ce qui facilite le développement et la maintenance séparée de ces codes par des équipes différentes. Notons ici que nous n'aborderons pas dans la suite de nos propos les aspects liés à la sécurité et au déploiement de ces applications, aspects pour lesquels nous souhaitons reposer sur des solutions existantes.

Toutefois, ce couplage diffère d'un couplage de codes au sens traditionnel pour plusieurs raisons. Tout d'abord, on peut remarquer qu'il n'est pas symétrique dans le sens où la simulation joue un rôle prépondérant par rapport à l'interface utilisateur qui est davantage volatile. Par ailleurs, il présente une forte *dynamicité*. Idéalement, plusieurs utilisateurs peuvent se connecter et/ou se déconnecter au cours de la simulation. Ainsi, le nombre de codes couplés (deux à deux) varient dynamiquement au cours de l'exécution. De plus, l'intervention de « l'humain » dans la boucle de calcul implique la *non-prévisibilité* des actions de pilotage et donc des schémas de communication entre les codes. Tous ces points ne sont pas en contradiction avec les applications classiques du couplage de codes : au contraire, ils soulignent les besoins actuels d'environnements de couplage de plus en plus généralistes, flexibles et dynamiques, une tendance ayant motivé l'émergence de nouveaux standards autour des composants logiciels [4, 189]. Pour finir, notons que le couplage simulation-visualisation présente une certaine *irrégularité*, en partie à cause du fort déséquilibre qu'il existe entre le nombre de processeurs côté simulation (de l'ordre de la centaine et plus) et côté visualisation (de l'ordre de la dizaine voire le plus souvent réduit à un seul processeur). Le transfert des données du code de simulation vers le code de visualisation peut alors entraîner un goulot d'étranglement important. Deux techniques complémentaires sont envisageables pour surmonter ce problème. Une première technique consiste à réduire le volume des données transférées, quitte à dégrader l'information (compression, filtrage). Une deuxième technique consiste à accélérer le transfert en utilisant des flux de communication parallèles (agrégation de la bande passante), ce qui nécessite de mettre en place des algorithmes de redistribution des données entre les codes couplés.

3.4.2 Conception et réalisation de la plate-forme EPSN

Le pilotage des simulations numériques parallèles et le couplage avec des codes de visualisation parallèle soulèvent deux problèmes majeurs que nous étudions dans cette thèse et pour lesquels nous souhaitons apporter une contribution : le problème de la coordination efficace des opérations de pilotage et le problème de la redistribution pour des données complexes (e.g. grilles structurées, ensembles de particules, maillages non structurés, *etc*.). Pour répondre à ces problèmes, nous proposons aux chapitres suivants deux modèles : *un modèle pour le pilotage fin des simulations numériques parallèle* (chapitre 4) et *un modèle pour la redistribution d'objets complexes* (chapitre 5). La conception de la plate-forme EPSN repose intégralement sur la définition de ces deux modèles. La réalisation d'EPSN est décrite au chapitre 6 et repose essentiellement sur la technologie CORBA pour la communication et VTK pour la visualisation (parallèle).

Modèle pour le pilotage fin des simulations numériques parallèle

Dans EPSN, nous avons choisi d'introduire un nouveau modèle de représentation des simulations, appelé « Modèle Hiérarchique en Tâches » (MHT), inspiré des travaux de *Polychronopoulos* et *Girkar* sur les graphes de tâches hiérarchiques (HTG) [92]. Ce modèle repose sur *un arbre de tâches* capable de modéliser le flot d'exécution d'un programme structuré de type SPMD (tâches simples, tâches en boucle, tâches conditionnelles). La construction d'un MHT repose sur l'annotation dans le code source de la simulation du début et de la fin de chaque tâche, ce qui offre à l'utilisateur une très grande liberté pour décrire sa simulation (plus ou moins finement) et donc de la piloter. En outre, le MHT permet de se repérer finement dans l'exécution d'un code parallèle grâce à l'introduction d'un système de datation. Cette propriété fondamentale du MHT va nous permettre de

coordonner efficacement les interactions de pilotage, grâce à un algorithme de planification. Cet algorithme, qui s'inspire de la stratégie « *loose synchronization* » de CUMULVS, permet de garantir la cohérence temporelle des traitements de pilotage en parallèle en évitant toute synchronisation coûteuse. Afin de limiter le surcoût dû au pilotage, nous utilisons dans EPSN un modèle de pilotage entièrement dirigé par le code de visualisation, basé sur un système de requêtes. Ainsi, en l'absence de requêtes de pilotage, la simulation ne souffre *a priori* d'aucune perturbation. Afin d'exécuter efficacement les requêtes de pilotage, nous avons introduit dans EPSN la notion de *plage d'interaction*. En particulier, nous utilisons ce mécanisme pour définir des plages d'accès aux données associées aux tâches du MHT. Cela permet d'une part de spécifier les plages du code où l'accès aux données peut s'effectuer en toute sûreté (cohérence spatiale). D'autre part, l'utilisation de plages d'accès plutôt que de simples points d'accès permet de recouvrir les transferts par les calculs de la simulation, et donc de réduire encore le surcoût du pilotage dans EPSN.

Modèle pour la redistribution d'objets complexes

Afin de transférer efficacement les données entre un code de simulation parallèle et un code de visualisation lui-même parallèle, nous avons développé des algorithmes de redistribution, dont le rôle est de générer les messages qu'il faut échanger entre chaque couple de processus (flux parallèles de communication). A notre connaissance, cette problématique n'a pas encore été étudiée dans le contexte du pilotage, ce qui a motivé nos travaux de recherche dans cette direction. En effet, le couplage simulation/visualisation peut conduire à des redistributions de données plus complexes que celles étudiées classiquement (i.e. tableaux denses, distributions bloc-cycliques). Il est donc important d'étendre les études précédentes pour prendre en compte d'une part des distributions irrégulières (décomposition spatiale [176], structures creuses, *etc.*) et d'autre part des données non structurées (e.g. particules, maillages). Le développement d'une solution pour ce problème repose tout d'abord sur la définition d'un modèle de description des données distribuées en mémoire qui doit être le plus généraliste possible. Notre modèle de description se base sur la notion *d'objet complexe*. Ces objets ne se réduisent pas à de simples tableaux de données distribuées en mémoire, mais ils s'efforcent de décrire les objets physiques tels qu'ils sont présents dans les codes de simulation : grilles structurées, ensembles de particules, maillages non structurés, *etc*. Ces objets sont constitués de plusieurs séries de données partageant la même distribution et encapsulent des informations supplémentaires relatives à la géométrie facilitant leur représentation graphique dans un code de visualisation (e.g. rayon des particules, séries des coordonnées, type de cellule du maillage, *etc.*). Sur la base de ce modèle, nous avons établi plusieurs algorithmes de redistribution, que l'on peut séparer en deux approches : *l'approche spatiale* et *l'approche placement*. L'approche spatiale peut être vue comme une généralisation des algorithmes de redistribution utilisés classiquement en algèbre linéaire. Elle s'applique à des objets complexes comme des matrices, des grilles structurées ou des boîtes d'atomes, qui utilisent des distributions par blocs quelconques (éventuellement « creuses »). L'approche placement de la redistribution est, quant à elle, plus particulièrement adaptée au contexte du pilotage : elle peut s'appliquer à des objets complexes non structurés, comme des ensembles de particules ou des maillages. Cette approche tire profit du fait que la distribution côté visualisation peut être fixée à l'exécution à partir de celle définie dans la simulation. En outre, cette approche possède de bonnes propriétés qui permettent de prendre en compte la dynamicité de ces objets (e.g. migration des particules, filtrage dynamique des données à visualiser, *etc.*).

Réalisation

L'architecture d'EPSN utilise un modèle client/serveur simple, comparable à CUMULVS, mais également proche dans l'esprit du couplage défini dans PAWS, dont nous empruntons l'abstraction en composant. En effet, une des motivations première de notre projet est de mettre en place un couplage performant, ce qui s'accommode mal de l'utilisation d'un serveur de données centralisant les transferts. Afin d'augmenter la réactivité du système de pilotage et de limiter les perturbations induites sur la simulation, la plate-forme EPSN effectue tous les traitements de pilotage dans un thread dédié, ce qui permet également d'introduire de l'asynchronisme dans les traitements. Ce thread est attaché à chaque processus de la simulation : il joue le rôle d'un serveur parallèle à l'écoute de requêtes clientes et peut accéder directement aux données de la simulation via la mémoire partagée pour les envoyer de manière asynchrone aux clients (et inversement). Cette approche permet de recouvrir le transfert des données sans effectuer de copie à l'envoi, mais nécessite la mise en place de mécanismes de synchronisation relativement complexes pour garantir la cohérence des accès. Par

ailleurs, il est possible d'utiliser un pool de threads dans la plate-forme EPSN afin de maximiser la concurrence des traitements en mode multi-utilisateurs. La couche de communication d'EPSN (système de requêtes et transfert des données) est entièrement basée sur la technologie CORBA, un *middleware* orienté objet facilitant le développement d'applications distribuées dynamiques, interopérables et portables (cf. Sec. 2.2.4). De plus, il existe des implantations de CORBA qui offrent de très bonnes performances (e.g. comme OmniORB4), ce qui a justifié le choix de cette technologie dans EPSN. Notons que l'utilisation de CORBA est entièrement masquée aux utilisateurs qui interagissent avec l'environnement EPSN par l'intermédiaire d'une API ou d'une interface utilisateur prédéfinie.

Côté visualisation, nous avons choisi d'explorer les possibilités offertes par VTK [33] qui est bien adaptée à la visualisation scientifique, sans pour autant nous restreindre à cette bibliothèque. VTK possède des widgets 3D, qu'il est possible d'exploiter pour définir (comme dans CSE) des interactions de haut-niveau par manipulation directe de l'image. Par ailleurs, VTK fournit des algorithmes de visualisation parallèle (e.g. calcul d'iso-surfaces en parallèle) qui permettent de traiter efficacement de plus gros volumes de données distribuées en mémoire. D'autre part, il est possible d'accélérer le rendu en combinant les cartes graphiques de plusieurs PCs (rendu parallèle en *sort-last*, cf. Sec. 2.3.2). Au final, il serait intéressant de coupler les possibilités de VTK à la plate-forme EPSN afin de piloter en vraie grandeur une simulation numérique dans un environnement de réalité virtuelle immersif. Plus précisément, notre objectif sera de visualiser en « temps-réel » les résultats intermédiaires d'une simulation parallèle sur un mur d'images et d'interagir avec les paramètres et les données du modèle numérique.

Deuxième partie

Modèle pour un environnement de pilotage

Chapitre 4

Modèle pour un pilotage fin des simulations numériques parallèles

Sommaire

4.1 Introduction

Afin d'être aussi générique que possible, nous établissons le pilotage d'une simulation numérique sur la base de sa représentation dans un *modèle abstrait*. Cette approche est dite *générique* dans la mesure où elle permet de considérer la problématique du pilotage indépendamment d'une simulation particulière. De manière générale, le modèle abstrait a pour but d'indiquer à l'environnement de pilotage *où, quand* et *comment* il peut interagir de manière cohérente avec la simulation. La définition d'un tel modèle est donc de toute première importance pour la conception et la réalisation d'un environnement de pilotage comme EPSN. Même si le modèle que nous présentons dans ce chapitre fait directement référence à l'environnement EPSN, sa définition est tout à fait indépendante d'une implantation particulière. Nous verrons au chapitre 6 un exemple d'implantation de ce modèle basée sur la technologie CORBA. Dans le cadre de cette thèse, nous nous intéressons aux pilotages de simulations numériques parallèles, impliquant un processus de calcul itératif, comme par exemple les schémas de résolution en temps. Cette dernière hypothèse est principalement liée au fait que nous

abordons la problématique de la visualisation en ligne des résultats intermédiaires. Afin de limiter notre étude, nous allons uniquement considérer des applications parallèles (en mémoire distribuée) utilisant le modèle de programmation SPMD (Single Program Multiple Data). Dans ce cas, un seul programme est exécuté en parallèle par un ensemble de processus sur un ensemble de données distribuées. Le modèle de pilotage que nous cherchons à mettre en place ne doit pas se limiter à une vision trop simplifiée du code. Au contraire, nous souhaitons piloter *finement* des simulations numériques, pouvant être constituées d'un ensemble de tâches et de boucles imbriquées arbitrairement complexe. Nous avons étudié dans le chapitre précédent (Sec. 3.2.1) les solutions proposées par les environnements existants pour représenter et piloter les codes de simulation. Aucun des modèles proposés ne nous satisfait tout à fait, car les abstractions utilisées sont souvent trop simplistes pour permettre de coordonner efficacement des opérations de pilotage tout en garantissant la cohérence des traitements effectués (en parallèle). Par ailleurs, aucun des environnements que nous avons étudiés ne permet d'exploiter des *plages d'interaction*. Dans ces environnements, les interactions sont généralement contraintes en des points d'instrumentation (e.g. accès synchrone bloquant aux données). L'utilisation de plage d'interaction peut permettre d'améliorer considérablement les performances lors des transferts, notamment en recouvrant les communications du système de pilotage par des calculs de la simulation. En résumé, le modèle de pilotage que nous proposons est principalement motivé par deux objectifs : (1) offrir à l'utilisateur une abstraction de haut-niveau permettant une représentation fine des simulations parallèles ; (2) offrir un modèle de pilotage performant capable d'exploiter des plages d'interaction tout en garantissant la cohérence des traitements effectués.

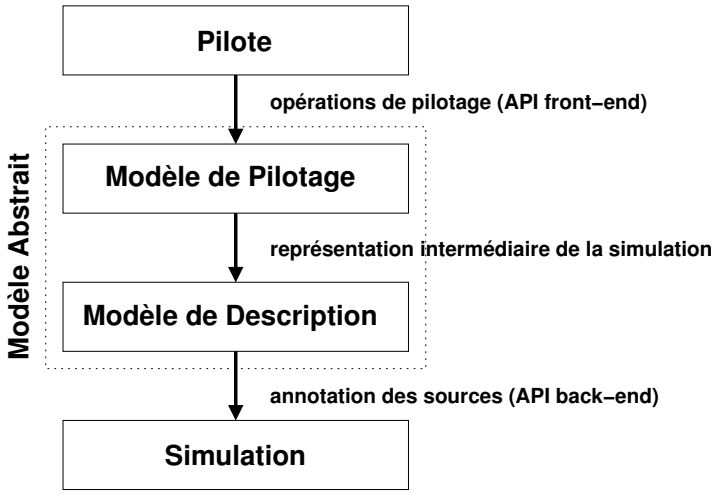


FIG. 4.1 – Modèle abstrait pour le pilotage d'une simulation.

Comme le suggère la figure 4.1, nous avons choisi de clairement séparer la définition de notre modèle abstrait en deux parties : *le modèle de description* et *le modèle de pilotage*.

- Le modèle de description de la simulation a pour but de décrire les éléments clés impliqués dans le processus de pilotage (données distribuées, structure du code, *etc.*). Cette description est généralement obtenue en demandant à l'utilisateur d'instrumenter la simulation, c'est-à-dire d'annoter le code source au moyen d'une API *back-end*. L'originalité de notre approche consiste à décrire la simulation selon un modèle, baptisé « Modèle Hiérarchique en Tâches » (MHT), basé sur un arbre de tâches et inspiré des travaux de *Polychronopoulos* et *Girkar* sur les graphes de tâches hiérarchiques (HTG) [92].

- Le modèle de pilotage a pour but de définir les opérations de pilotage qu'il est possible d'effectuer sur la simulation depuis une application cliente distante. Ce modèle s'appuie sur la représentation intermédiaire du code fournie par le modèle de description. Le pilote déclenche des opérations de pilotages par le biais d'une API *front-end* ou directement à l'aide d'une interface graphique (GUI) construite sur cette API. Le modèle de pilotage que nous proposons se base sur un *système de requêtes* entièrement dirigées par le client.

Dans la suite de ce chapitre, nous présentons plus en détail le modèle de description (Sec. 4.2) et le modèle de pilotage (Sec. 4.4) que nous avons défini. Sur la base de ce modèle, nous introduisons un *système de datation* permettant de se repérer précisément dans une simulation (Sec. 4.3) et de construire un algorithme de coordination résolvant efficacement le problème de la cohérence temporelle (Sec. 4.4.2). A la fin de ce chapitre, nous discuterons des limitations et des extensions possibles de notre modèle pour des simulations non SPMD.

4.2 Modèle de description des simulations

Le modèle de description des simulations SPMD complexes que nous proposons est clairement découpé en trois parties : la description des données distribuées, la description de la structure du programme, et la description des interactions de pilotage possibles. Ce modèle introduit un système de datation permettant de se repérer précisément dans la structure du programme et de suivre son évolution en temps au cours de l'exécution.

4.2.1 Description des données distribuées

Les données présentes dans les simulations numériques ne sont pas toutes pertinentes du point de vue du pilotage : lors de l'instrumentation, il est donc nécessaire de désigner précisément les données que le client pourra consulter ou modifier lors de la phase d'interaction. Comme nous l'avons vu au chapitre précédent (Sec. 3.2.1), cette description implique de décrire d'une part la distribution des données entre les différents processus de calcul et d'autre part de spécifier l'agencement de ces données en mémoire. Pour cela, il ne suffit pas simplement de connaître la plage mémoire allouée sur chaque processus, il faut encore être capable d'accéder à chacun des éléments précisément voire d'accéder à un sous-ensemble de ces éléments. Décrire les données présentes dans les codes de calcul parallèles n'est pas une chose aisée, car il n'existe pas un modèle de description standard qui soit approprié dans tous les contextes. Bien au contraire, les structures de données sont le plus souvent spécifiques à chaque application, conditionnées à la fois par des critères d'organisation logique de l'information et de performance des calculs. Pour ces raisons, nous avons choisi de nous *abstraire* d'un modèle de description précis, afin d'être aussi générique et extensible que possible. Nous proposerons un modèle de description des données au chapitre suivant (chapitre 5 sur la redistribution), car la définition d'une solution au problème de la redistribution des données est intimement liée au modèle de représentation des données utilisé.

Nous nous contenterons donc ici de présenter les éléments de description minimaux permettant de désigner et de manipuler ces données de manière abstraite dans notre modèle de pilotage. Les attributs de description que nous considérons à ce niveau sont au nombre de trois : l'identifiant de la donnée (*ID*), la classe de la donnée (*class*) et sa localisation (*location*). L'identifiant est une simple chaîne de caractères fournie par l'utilisateur permettant de désigner de manière unique une donnée dans le modèle. La classe sert à renseigner sur la nature de la donnée considérée. Cette information dépend directement de la bibliothèque de redistribution utilisée. Usuellement, les classes de données que nous visons sont des scalaires, des tableaux multi-dimensionnels, des grilles structurées, des ensembles de particules ou des maillages non structurés. Le dernier attribut permet d'indiquer si la donnée est distribuée (*distributed*) ou répliquée (*replicated*) sur l'ensemble des processus, ou bien si elle est localisée (*located*) sur un seul processus. Ces informations devront être fournies et complétées par la bibliothèque de redistribution utilisée.

4.2.2 Modèle hièrarchique en tâches (MHT)

L'originalité de notre approche consiste à représenter les simulations parallèles SPMD à l'aide d'un *arbre de tâches* inspiré des *graphes de tâches hiérarchiques* ou HTG. La notion de HTG a été initialement introduite par *Polychronopoulos* et *Girkar*, dans un article publié en 1994 et intitulé « The Hierarchical Task Graph as a Universal Intermediate Representation » [92]. Ces travaux s'inscrivent dans le contexte de la « parallélisation » automatique de codes (compilateur Parafrase-2 [177]). Dans ces travaux, les auteurs démontrent qu'il est possible de remplacer avantageusement le traditionnel CFG (Control Flow Graph) par un HTG, capable d'encapsuler des informations aux différents niveaux de la hiérarchie. Ce type de représentation est particulièrement bien adapté

pour traiter des problèmes de parallélisation automatique à gros grain, ce qui s'avère plus difficile avec des modèles comme les CFGs donnant une vision « à plat » du programme.

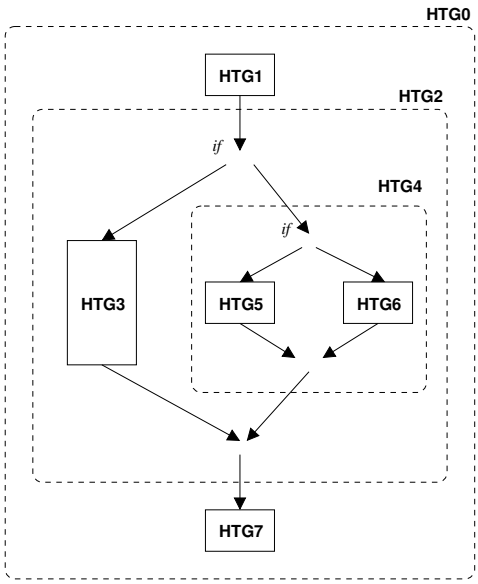


FIG. 4.2 – Exemple de représentation d'un HTG.

Par définition, un HTG est un graphe orienté acyclique (ou DAG), composé de nœuds (ou tâches) et d'arcs modélisant le flot de contrôle entre ces nœuds. Dans la définition initiale, trois types de nœuds sont considérés : les nœuds simples (*single nodes*), les nœuds composés (*compound nodes*) et les nœuds « en boucle » (*loop nodes*). Les nœuds simples servent à encapsuler des blocs de code élémentaires (e.g. suite d'instructions, procédure, *etc.*). Ils sont non hiérarchiques par définition et ne possèdent donc pas de sous-nœuds. Les nœuds composés sont des HTGs, permettant d'encapsuler des sous-nœuds dans la hiérarchie. Ils sont typiquement utilisés pour représenter des structures conditionnelles (*if-then-else, switch-case, etc.*) comme le montre la figure 4.2. Notons qu'au plus haut-niveau de la hiérarchie, le HTG se présente lui-même comme un nœud composé particulier. Les nœuds « en boucle » sont utilisés pour représenter les différents types de boucle (*for, while, etc.*). Un nœud « en boucle » est formé d'un nœud simple marquant le début et la fin de la boucle et d'un nœud composé représentant le corps de la boucle.

Modèle hiérarchique en tâches (MHT)

Dans le contexte de la compilation, le modèle HTG permet de représenter la totalité d'un programme, du point d'entrée (*start node*) jusqu'au point de sortie (*stop node*), de telle sorte que toute instruction appartient au moins à une tâche dans la hiérarchie. En pratique, un HTG s'obtient automatiquement en analysant syntaxiquement le code sur la base d'un AST (Abstract Syntax Tree) ou d'un CFG (recherche des blocs d'instructions insécables, des boucles, des conditionnelles, des appels de routines, *etc.*). Dans le contexte du pilotage, il n'est généralement pas nécessaire de représenter l'ensemble du code. Seules quelques tâches et boucles au plus haut-niveau de la hiérarchie sont suffisantes. Pour ces raisons, nous avons choisi d'utiliser une technique d'instrumentation du code, permettant d'extraire la structure essentielle du code que nous souhaitons piloter. Dans cette approche, l'utilisateur est libre de repérer dans le code source uniquement les tâches qu'il juge intéressantes pour sa problématique. On parle alors de *description partielle* du code. Ainsi, certaines parties du programme peuvent être délibérément laissées à l'écart, si elles ne sont pas instrumentées.

Le principal défaut du HTG à nos yeux est qu'il ne permet pas de se repérer facilement dans le flot d'exécution. En effet, l'utilisation de graphes acycliques orientés (DAGs) pour connecter les différents types de nœuds s'avère dans notre contexte trop complexe. Pour remédier à ce problème, nous avons décidé de

définir un nouveau modèle de représentation des programmes informatiques, appelé « Modèle Hiérarchique en Tâches » (MHT), basé sur un *arbre de tâches*, plus simple à manipuler dans notre contexte qu'un CFG ou qu'un HTG. Ce modèle a l'avantage comparé au HTG de ne pas utiliser de DAGs, qui sont remplacés par la combinaison de deux types de tâche : les *tâches composées* et les *tâches conditionnelles*. Les tâches composées sont formées d'une succession de tâches (linéairement connectées), facilement repérables par un indice dans la séquence. Les tâches conditionnelles permettent de capturer explicitement les structures conditionnelles à deux ou plusieurs branches, chaque branche étant représentée par une sous-tâche (e.g. *bloc-then*, *bloc-else*), facilement repérable par le numéro de la branche. La combinaison de ces deux types de tâche, ainsi que l'utilisation de tâches en boucle, permet de représenter le flot d'exécution de la plupart des programmes que nous souhaitons piloter. A titre d'exemple, notre modèle ne permet pas de représenter des *goto*, des exceptions et plus largement tout mécanisme entraînant des ruptures du flot d'exécution « normal » d'un programme. Par ailleurs, nous introduisons un dernier type de tâche dégénéré, appelé *tâche en point*, exclusivement dédié aux interactions de pilotage avec les clients distants.

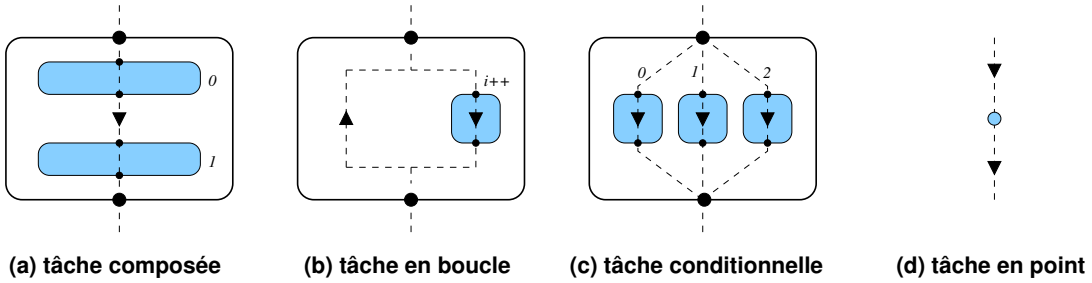


FIG. 4.3 – Les différents types de tâche utilisés dans les arbres de tâches.

En résumé, nous allons considérer quatre types de tâche dans notre modèle : la tâche composée, la tâche en boucle, la tâche conditionnelle et la tâche en point (Fig. 4.3).

1. La *tâche composée* encapsule un bloc de code logique (e.g. suite d'instructions, procédure, *etc.*). Elle est repérée dans le code par deux points d'instrumentation, marquant le début et la fin de la tâche, et contient une séquence de sous-tâches. Notons que cette séquence peut éventuellement être vide. Dans ce cas particulier, nous parlerons de *tâche simple*. L'ensemble des sous-tâches (formant la séquence) sont ordonnées linéairement selon le flot d'exécution du programme, chaque sous-tâche étant repérée par son indice dans la séquence. Il faut remarquer que la totalité du code encapsulé par la tâche composée n'est pas forcément recouvert par l'ensemble des sous-tâches (i.e. description partielle du code).
2. La *tâche en boucle* permet de capturer les structures itératives (*for, while, etc.*), le corps de la boucle étant définie par une et une seule sous-tâche. Deux points d'instrumentation servent à marquer le début et la fin de la boucle dans le code. Un compteur, associé au corps de boucle, tient à jour le numéro de l'itération courante. Ce compteur sera automatiquement incrémenté à chaque passage sur le point d'instrumentation marquant l'entrée dans le corps de boucle (i++).
3. La *tâche conditionnelle* permet de représenter les structures conditionnelles à deux ou plusieurs branches (*if-then-else, switch-case, etc.*), chaque branche étant définie dans une sous-tâche. Les tâches conditionnelles permettent de se repérer simplement dans un flot d'exécution conditionnel, en tenant à jour l'indice de la branche courante.
4. La *tâche en point* est une tâche dégénérée, ne possédant pas de sous-tâches et repérée dans le code par un point d'instrumentation unique. Nous introduisons les tâches en point spécialement pour prendre en compte des interactions de pilotage, comme nous le verrons dans la suite.

La figure 4.4 montre comment il est possible de modéliser un programme à l'aide du modèle hiérarchique en tâches. Cet exemple, baptisé Ω, nous aidera à illustrer nos propos tout au long de ce chapitre. La figure 4.5. représente l'arbre de tâches associé au MHT Ω. Cette dernière représentation insiste sur les relations de parenté entre les tâches (tâche-père, tâche-frère), tandis que la première représentation met en évidence le

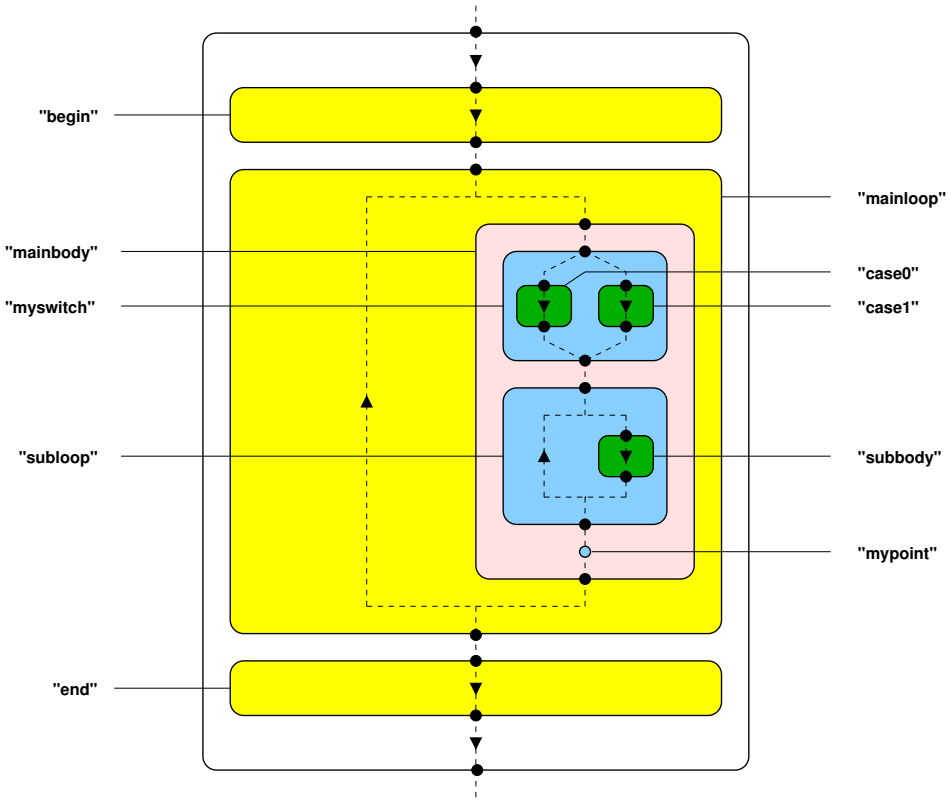


FIG. 4.4 – Exemple d'un MHT Ω. Cette représentation met en évidence la hiérarchie des tâches et le flot d'exécution (en pointillé) du programme modélisé.

flot d'exécution du programme modélisé. Le MHT Ω contient au premier niveau de la hiérarchie une tâche d'initialisation (*begin*), une boucle principale marquant l'évolution en temps de la simulation (*mainloop*) et une tâche de terminaison (*end*). Au deuxième niveau de la hiérarchie, le corps de la boucle principale (*mainbody*) est une tâche composée d'une séquence de trois sous-tâches (*myswitch*, *subloop*, *mypoint*). Tout d'abord, la tâche conditionnelle (*myswitch*) permet d'alterner entre deux tâches simples, *case0* et *case1*, comme par exemple pour calculer un type de pré-conditionneur dans une méthode itérative. La sous-boucle (*subloop*) encapsule le solveur itératif, dont le corps est une tâche simple baptisée *subbody*. Enfin, le corps de la boucle principale s'achève par une tâche en point (*mypoint*).

Ce modèle permet de se repérer intuitivement dans le flot d'exécution d'un programme. Par exemple, il est possible de dire que le flot d'exécution courant se situe dans la boucle *mainloop* au premier niveau de la hiérarchie, à la i-ème itération du corps de boucle *mainbody* au deuxième niveau de la hiérarchie, puis dans la conditionnelle *myswitch* au troisième niveau de la hiérarchie, et enfin dans la tâche *case0* correspondant au quatrième et dernier niveau de la hiérarchie. Dans la section 4.3, nous formaliserons le repérage dans un MHT grâce à l'introduction d'un *système de datation*. En conclusion, le MHT permet de refléter la structure essentielle des codes que nous souhaitons piloter. En effet, dans le contexte du pilotage, quelques niveaux dans la hiérarchie des tâches sont généralement suffisants pour décrire des simulations complexes. Ce modèle permet de représenter naturellement des programmes séquentiels ou parallèles SPMD. Car dans ce dernier cas, tous les processus exécutent le même code et partagent donc la même représentation en MHT. Nous verrons dans les perspectives de cette thèse comment il est possible d'étendre ce modèle pour prendre en compte des applications « multi-codes » (e.g. maître/esclave, MPMD, *etc.*).

4.2.3 Description des interactions de pilotage

Un modèle de description se doit de spécifier *où* et *quand* les diverses interactions de pilotage peuvent s'effectuer au cours de la simulation. Dans cette section, nous présentons deux mécanismes d'interaction complé-

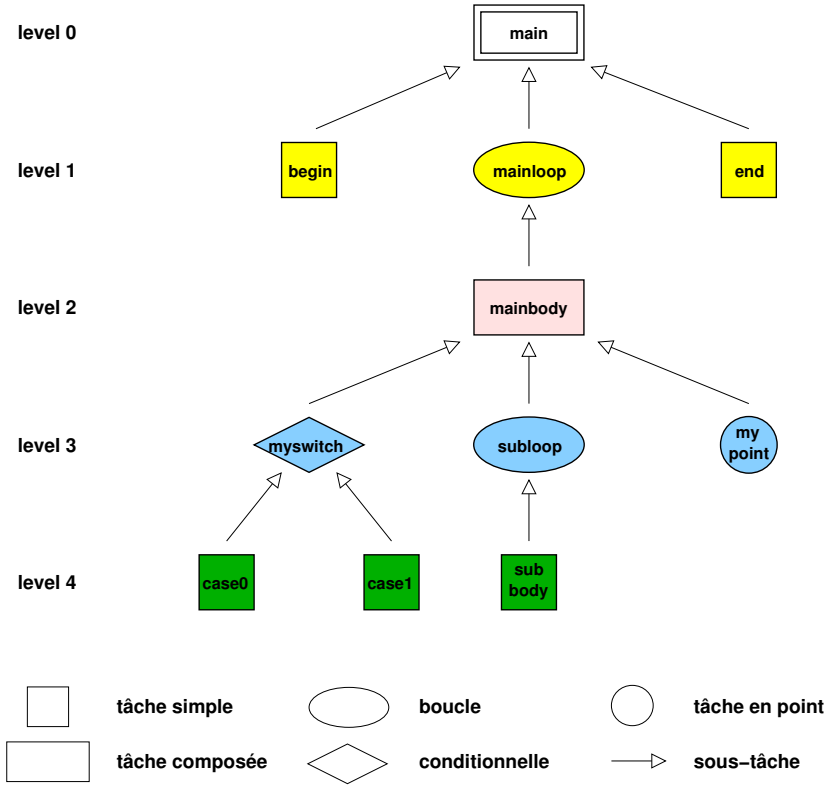


FIG. 4.5 – L'arbre de tâches du MHT Ω.

mentaires : les *points d'interaction* et les *plages d'interaction*. Afin de construire un modèle de description simple et cohérent, nous nous appuyons sur les tâches définies dans le MHT pour spécifier où pourront survenir les différentes interactions possibles. En particulier, nous verrons comment il est possible de configurer précisément l'accès aux données grâce à l'introduction d'un contexte sur les tâches.

Points et plages d'interaction

Par définition, *un point d'interaction* est un mécanisme d'interaction associé à un point d'instrumentation dans le code, permettant d'effectuer des traitements de pilotage intrinsèquement synchrones et bloquants (Fig. 4.6 (a)). Comme nous l'avons déjà suggéré, les tâches sont délimitées par des points d'instrumentation qui peuvent servir de support dans le code de simulation aux diverses interactions avec les programmes clients. En particulier, les tâches en point ont été explicitement introduites dans notre modèle pour servir de point d'interaction. Ces points peuvent être le support d'interactions diverses et variées, comme par exemple : les points d'arrêt, permettant de contrôler le flot d'exécution ; les points d'accès aux données en lecture ou en écriture ; les points d'action, permettant à l'utilisateur de déclencher des actions dans la simulation, *etc*.

Si la notion de point d'interaction est largement exploitée dans tous les environnements de pilotage que nous avons étudiés précédemment, l'utilisation de plages d'interaction est une contribution originale dans nos travaux. Comme le suggère la figure 4.6 (b), nous associons aux tâches du MHT des plages d'interaction. Dans ce cas, l'interaction de pilotage peut s'effectuer de manière concurrente à l'exécution du programme, mais contrainte entre les deux points d'instrumentation marquant le début et la fin de la tâche. Ce mécanisme peut être typiquement utilisé pour envoyer des données de manière asynchrone, ou pour exécuter une action de pilotage concurremment dans un *thread*. Afin de maintenir la cohérence (spatiale), il faut garantir que le traitement de pilotage s'achève avant la fin de la tâche. Si tel est le cas, on dit que le recouvrement calcul/pilotage est *total*. Dans le cas contraire, le point d'instrumentation marquant la fin de la tâche doit temporairement bloquer le flot d'exécution de la simulation et attendre la fin de l'interaction. Le recouvrement calcul/pilotage est alors *partiel*.

La notion de plage d'interaction est suffisamment générale pour s'appliquer à divers types d'interaction de pilotage. Dans la section suivante, nous examinerons plus en détail comment utiliser ce mécanisme pour permettre un accès efficace aux données, mettant en œuvre le recouvrement des transferts de données entre la simulation et l'UI. Ce mécanisme fera l'objet de validation expérimentale dans l'environnement EPSN au chapitre 7.

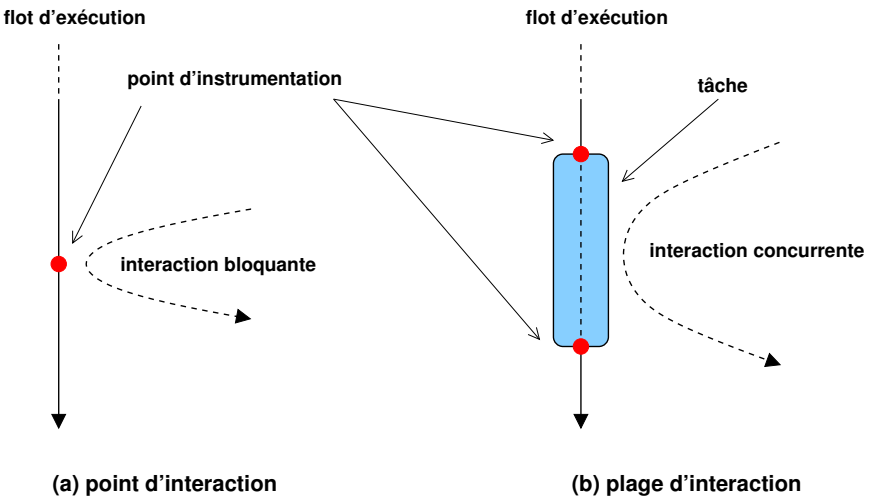


FIG. 4.6 – Les deux mécanismes d'interaction.

Modèle d'accès aux données

L'accès aux données est un problème crucial dans tous les environnements de pilotage. Un modèle d'accès doit permettre à l'utilisateur de décrire simplement les régions du code où l'accès aux données est autorisé afin de permettre à l'environnement de pilotage de maintenir la cohérence des accès (cohérence spatiale). Le modèle d'accès que nous proposons se base simplement sur la description du code en MHT, que nous enrichissons afin de définir des points et des plages d'interaction particuliers pour l'accès aux données. Dans ce cas précis, le point d'interaction joue le rôle d'un *point d'accès* à une ou plusieurs données et la plage d'interaction joue le rôle d'une *plage d'accès*. Afin de configurer les accès aux données dans notre modèle, nous introduisons la notion de *contexte d'accès*, qui permet d'associer des comportements prédéfinis aux points et tâches d'un MHT. Plus précisément, le contexte d'accès permet de spécifier si, au cours d'une tâche ou d'un point, une donnée est consultable (*readable*), modifiable (*writable*), protégée (*protected*) ou modifiée (*modified*). Les deux premiers cas (*readable*, *writable*) permettent de préciser si la donnée est accessible en lecture ou en écriture par le client de pilotage. Dans les deux derniers cas (*protected*, *modified*), l'accès à la donnée est interdit pendant toute la durée de la tâche. De plus, le contexte *modified* signale à l'environnement de pilotage que la donnée est modifiée par la tâche, et donc qu'une nouvelle version (*release*) sera disponible à l'issue de la tâche. Cette information est particulièrement intéressante si l'on cherche à visualiser en ligne les résultats intermédiaires d'une simulation.

L'ensemble des tâches imbriquées dessinent naturellement une hiérarchie de contextes, qu'il est possible de paramétrer finement. Les sous-tâches héritent du contexte de leur père, qu'ils peuvent localement surcharger pour autoriser ou interdire plus précisément l'accès aux données. La figure 4.7 illustre un exemple d'utilisation des contextes pour configurer l'accès à la donnée *mydata* sur le MHT Ω. Dans cet exemple, le contexte initial (associé à la tâche racine du MHT) interdit globalement l'accès à cette donnée. La boucle *mainloop* surcharge localement ce contexte pour ouvrir une grande plage d'accès à la donnée. Toutefois, comme la boucle *subloop* modifie la donnée (contexte *modified*), l'accès à la donnée est interdit durant cette tâche afin de maintenir la cohérence spatiale. A chaque itération de la boucle principale, une nouvelle version de la donnée sera signalée à l'environnement de pilotage. Par ailleurs, nous avons ajouté au point *mypoint* un contexte d'accès en écriture (*writable*), donnant l'opportunité aux clients de pilotage de modifier la donnée sur ce point. Basiquement, l'accès aux données en lecture/écriture est contrôlé par une variable à quatre états : *Read Only* pour un accès en lecture seul, *Write Only* pour un accès en écriture seul, *No Access* pour interdire tout accès et *Read/Write* pour un accès

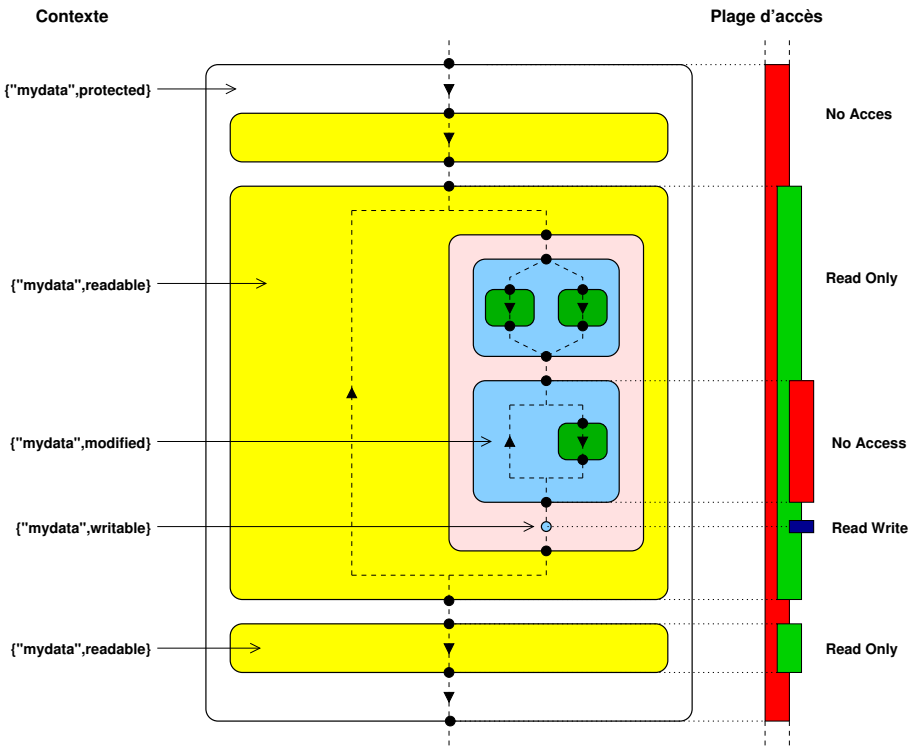


FIG. 4.7 – Contextes d'accès aux données associés au MHT Ω.

en lecture/écriture.

4.2.4 Principe de construction du MHT

Afin d'éclaircir notre propos, nous souhaitons préciser dans cette section comment nous construisons et utilisons le MHT dans un environnement de pilotage comme EPSN.

Instrumentation du code source

Pour construire une simulation pilotable, l'utilisateur doit préalablement annoter le code source au moyen d'une API dédiée (l'API *back-end*). Dans notre contexte, ces annotations visent essentiellement à repérer dans le code source l'ensemble des tâches décrites par le MHT. Plus précisément, l'utilisateur doit marquer le début et la fin de chaque tâche, à l'aide d'un appel de fonction dans l'API (*point d'instrumentation*). A titre d'exemple, la paire de fonctions *beginTask()/endTask()* sert à délimiter le début et la fin d'un tâche simple ou composée. Nous reviendrons plus en détails sur cette API au chapitre 6. La figure 4.8 illustre le principe de l'annotation du code source pour le MHT Ω. On remarquera que les annotations reflètent exactement la structure arborescente du MHT. Au cours de l'exécution, la traversée de ces points d'instrumentation permet de « passer la main » à l'environnement de pilotage pour qu'il effectue les traitements appropriés.

Pour conclure, nous soulignons que le processus d'instrumentation que nous utilisons est entièrement manuel. Certains travaux de *Polychronopoulos et al.* [56, 177] dans le domaine de la compilation ont démontré qu'il était possible d'obtenir automatiquement des représentations intermédiaires du code en HTG grâce à des outils d'analyse syntaxique. On peut imaginer adapter ces techniques pour automatiser (du moins partiellement) la construction d'un MHT. De tels outils ne sauraient complètement remplacer l'instrumentation du code fournie par l'utilisateur. Il se présentent à nos yeux comme une aide à l'instrumentation pour l'utilisateur, qui est le seul à connaître les tâches *a priori* pertinentes pour son problème.

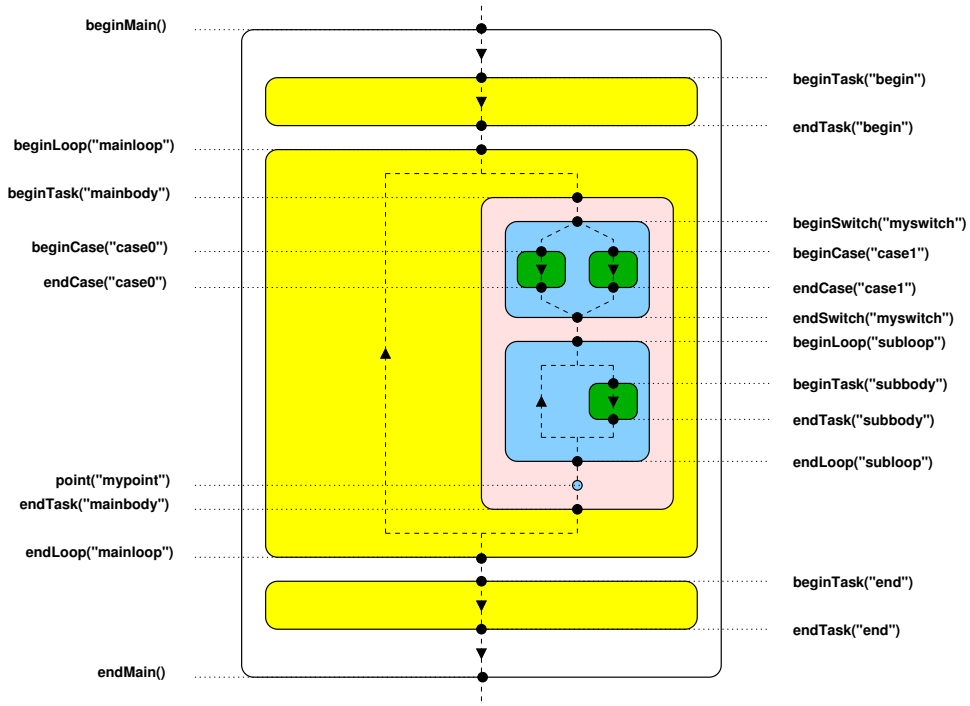


FIG. 4.8 – Annotation des tâches associées au MHT Ω.

Description du MHT en XML

En plus de l'instrumentation classique du code source, nous utilisons un fichier de description externe, dont le rôle est de fournir à l'environnement de pilotage une connaissance *statique* et *globale* du MHT avant l'exécution de la simulation. Ce fichier doit être écrit par l'utilisateur dans le langage XML, à l'aide d'un ensemble de balises élémentaires : *task*, *loop*, *switch*, et *point*. Le langage XML est bien adapté à la structure hiérarchique de notre modèle, dont il permet de refléter parfaitement l'enchevêtrement des tâches, comme l'illustre la figure 4.9. Grâce à l'utilisation de balises ouvrantes et fermantes, comme par exemple *<task id="mainbody">* ... *</task>*, il est possible de déclarer chaque tâche, ainsi que l'ensemble des sous-tâches contenues par cette tâche. L'identifiant relatif à l'attribut *id* renvoie au nom de la tâche utilisé pour l'annotation du code source. La spécification complète de la description XML utilisée dans EPSN est décrite à l'annexe A.3 par un fichier DTD (Document Type Declaration), un format standard du consortium W3C [210].

Même si il serait tout à fait envisageable de découvrir dynamiquement le MHT au moment de l'exécution, nous trouvons plusieurs avantages importants à cette approche. Tout d'abord, la connaissance *a priori* du MHT facilite la détection d'erreurs d'instrumentation au moment de l'exécution du code. Lorsqu'une erreur est découverte (i.e. parcours non valide, défini à la section 4.3.3), il sera toujours possible de la signaler à l'utilisateur pour qu'il effectue la correction appropriée de son instrumentation. La déclaration des tâches dans le XML offre l'opportunité à l'utilisateur de définir des interactions de pilotage, sans qu'il soit nécessaire de modifier l'instrumentation et donc de recompiler l'application. Grâce à l'utilisation de balises comme *data-context* ou *action-context*, il possible d'enrichir le XML en associant des informations contextuelles aux tâches permettant de configurer précisément les interactions de pilotage, comme par exemple l'accès à la donnée *mydata* sur la tâche *mainloop* ou la commande d'une action *myaction* sur le point *mypoint* (Fig. 4.9). Finalement, l'utilisation d'un fichier de description externe permet de réduire le nombre de lignes nécessaires à l'instrumentation du code, en ne conservant que les informations strictement nécessaires et en déportant la configuration des interactions dans ce fichier.

```
<MHT>
   <data-context ref="mydata" context="protected"/>
   <task id="begin"> </task>
   <loop id="mainloop">
      <data-context ref="mydata" context="readable"/>
      <task id="mainbody">
         <switch id="myswitch">
            <task id="case0"> </task>
            <task id="case1"> </task>
         </switch>
         <loop id="subloop">
            <data-context ref="mydata" context="modified"/>
            <task id="subbody"> </task>
         </loop>
         <point id="mypoint">
            <data-context ref="mydata" context="writable"/>
            <action-context ref="myaction" context="authorized"/>
         </point>
      </task>
   </loop>
   <task id="end">
      <data-context ref="mydata" context="readable"/>
   </task>
</MHT>
```

FIG. 4.9 – Description en XML du MHT Ω et des interactions de pilotages associées.

4.3 Les dates

Un des problèmes clés dans le domaine du pilotage consiste à se repérer dans le flot d'exécution d'une application. Dans ce contexte, l'introduction de la notion de « date » devient nécessaire pour plusieurs raisons. Tout d'abord, la notion de date est importante si l'on souhaite suivre et analyser l'évolution « en temps » de la simulation. Cela s'avère particulièrement pertinent pour l'analyse et l'optimisation des performances, ou encore pour contrôler le flot d'exécution de l'application. Par ailleurs, cette notion est primordiale pour coordonner les opérations de pilotage en parallèle et ainsi garantir que les traitements effectués, comme l'extraction des données pour la visualisation, surviennent « à la même date » sur tous les processus et qu'ils sont donc globalement cohérents en temps (cohérence temporelle). La notion de date est donc de première importance pour un environnement comme EPSN, qui s'intéresse plus particulièrement au problème de la visualisation en ligne. Toutefois, ces solutions sont trop simples pour permettre de se repérer précisément dans une simulation complexe telle que nous la représentons avec un MHT. En effet, l'utilisation d'un simple *timestamp* serait nécessairement ambiguë dans notre modèle, qui peut comporter plusieurs boucles imbriquées. Pour surmonter cette difficulté, nous allons introduire la notion de *date de tâche* et de *date de point*, respectivement associés aux tâches et points d'instrumentation d'un MHT.

4.3.1 Les dates de tâche

La définition des dates de tâche utilise des notions empruntés à la *théorie des langages*. Pour plus de détails à ce sujet, nous renvoyons le lecteur à un ouvrage de référence comme [41].

Définition 4.3.1 (Date de tâche) *Une date de tâche d est un mot sur l'alphabet des entiers naturels $\mathbb{N}$, noté $d = d_0.d_1 \dots d_{n-1}$ avec $\|d\| = n$ la longueur du mot.*

L'ensemble des dates de tâche $\mathcal{D}_T$ est l'ensemble des mots sur l'alphabet $\mathbb{N}$, fréquemment noté $\mathbb{N}^*$ en théorie des langages. On dispose sur $\mathcal{D}_T$ d'une loi de composition interne, appelé *produit de concaténation*, dont l'élément neutre est le mot vide ϵ (de longueur nulle). Soient $v = v_0.v_1 \dots v_{m-1}$ et $w = w_1.w_2 \dots w_{n-1}$ deux dates de longueur respective m et n. On note $v.w$ la concaténation de ces deux dates, définie comme le mot $v_0.v_1 \dots v_{m-1}.w_0.w_1 \dots w_{n-1}$.

Les dates de tâche telles que nous venons de les définir sont bien adaptées à l'aspect hiérarchique des MHTs : à chaque niveau de la hiérarchie correspond un indice d_i dans la date. La date d d'une tâche T est construite récursivement par rapport à la date d' de son père T', par ajout d'un indice k supplémentaire à la

date (i.e. $d = d'.k$). Plus précisément, la signification des indices dépend de la nature de la tâche-père. Dans le cas d'une tâche composée, l'indice k désigne le numéro de la sous-tâche considérée dans la séquence. Dans le cas d'une tâche conditionnelle, k renvoie au numéro de la branche choisie. Dans le cas d'une tâche en boucle, k désigne le nombre d'itérations dans la boucle. Ainsi la date d associée à une tâche T est formée par la séquence des indices, obtenues en remontant les niveaux un-à-un dans la hiérarchie des tâches, jusqu'à la tâche principale du MHT correspondant à la date vide ϵ (au niveau 0 de la hiérarchie).

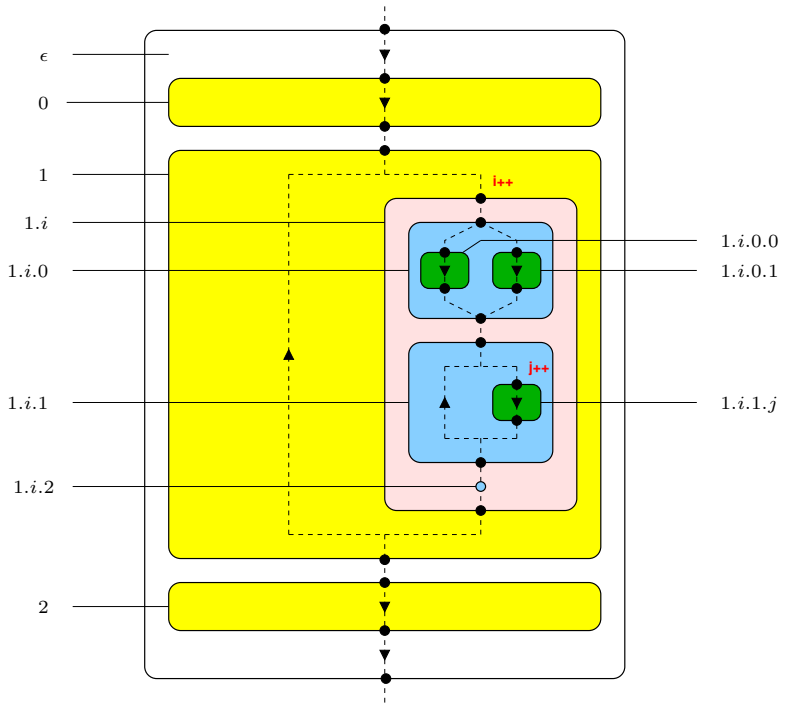


FIG. 4.10 – Dates de tâche associées au MHT Ω.

La figure 4.10 donne un exemple complet de construction des dates de tâche pour le MHT Ω étudié dans ce chapitre. Les tâches *begin*, *mainloop* et *end*, au premier niveau de la hiérarchie sont respectivement représentées par les dates 0, 1 et 2. La tâche *mainbody* est désignée par une date de la forme $1.i$ avec i le numéro d'itération dans la boucle *mainloop*. Les dates de tâche associées aux branches *case0* et *case1* de la conditionnelle *myswitch* sont respectivement $1.i.0.0$ et $1.i.0.1$. On peut remarquer que toutes les tâches descendantes de la boucle principale dépendront naturellement de la valeur du compteur d'itération i. De manière similaire, la date associée au corps de la sous-boucle (*subbody*) s'écrira $1.i.1.j$ avec j le numéro d'itération dans la sous-boucle (*subloop*).

Afin de mieux appréhender les dates de tâches, nous utilisons une représentation des dates en arbre. Les nœuds de l'arbre correspondent aux dates des tâches, et les arêtes correspondent aux indices permettant de passer d'une date de tâche d à la date d'une sous-tâche d', telle que $d' = d.i$ avec $i \in \mathbb{N}$. La racine de l'arbre qui désigne la tâche principale du MHT est associée à la date vide. Cette représentation est proche de l'arbre de tâches utilisé pour décrire le MHT (Fig. 4.5). La principale différence tient à la manière dont nous représentons les boucles : chaque itération dans la boucle induit une nouvelle branche dans l'arbre, ce qui revient simplement à dérouler la boucle dans l'arbre de tâches. La figure 4.11 représente l'arbre des dates de tâche associée au MHT Ω. Cet exemple illustre la construction des dates pour les différents types de tâche. Cette représentation souligne le rôle des indices dans le repérage. En particulier, dans le cas des boucles, le triangle en pointillé est là pour suggérer que nous ne représentons qu'une branche de l'arbre, afin d'éviter de surcharger la représentation de l'arbre.

Nous pouvons remarquer qu'à chaque tâche T d'un MHT Ω correspond un ensemble de dates atteignables dans $\mathcal{D}_T$. Cet ensemble est de taille 1 lorsque la tâche T n'hérite d'aucune boucle et de taille infinie dans le cas contraire, car les boucles entraînent une explosion combinatoire des dates, comme le suggère la figure 4.11.

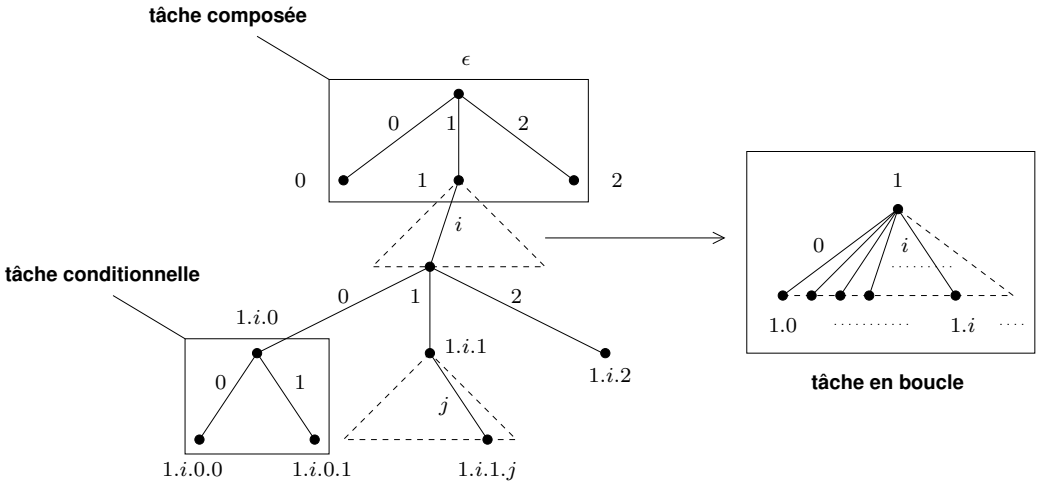


FIG. 4.11 – Arbre des dates de tâche du MHT Ω.

Par exemple, l'ensemble des dates atteignables par la tâche *begin* est le singleton $\{0\}$, de même que l'ensemble des dates atteignables par *mainloop* est $\{1\}$. En revanche, l'ensemble des dates atteignables par le corps de boucle *mainbody* est $\{1.i, i \in \mathbb{N}\}$. Par la suite, nous noterons $\{1.*\}$ cet ensemble. Dans notre nomenclature, $1.*$ représente un *masque de dates*, englobant la liste des dates : $1.0, 1.1, 1.2, \ldots$ Ainsi, les masques de dates des tâches *mypoint* et *subbody* sont respectivement $1.*.2$ et $1.*.1.*$. Finalement, l'ensemble des dates atteignables par le MHT Ω s'écrira $\mathcal{D}_T(\Omega) = \{\epsilon, 0, 1, 1.*, 1.*.0, 1.*.0.0, 1.*.0.1, 1.*.1, 1.*.1.*, 1.*.2, 2\}$. L'ensemble des dates atteintes à l'issue d'une exécution est un sous-ensemble *fini* de l'ensemble des dates atteignables, noté $\hat{\mathcal{D}}_T(\Omega)$. Cet ensemble est fini car on suppose que les boucles n'exécutent qu'un nombre fini d'itérations à chaque exécution. Par ailleurs, il faut remarquer que les branches des conditionnelles s'excluent mutuellement à l'exécution, ce qui signifie que seul une des dates associées aux branches de la conditionnelle sera présente dans $\hat{\mathcal{D}}_T(\Omega)$ (e.g. $1.i.0.0$ ou $1.i.0.1$ pour chaque itération i sur la figure 4.11).

Nous allons maintenant présenter quelques définitions et propriétés sur les dates de tâche, utiles dans la suite de ce chapitre. Tout d'abord, nous disposons trivialement de la relation d'égalité entre les mots. On a $d = d'$, si et seulement si $\|d\| = \|d'\|$ et $d_i = d'_i$ pour tout i compris entre 1 et $\|d\|$. La définition d'une relation d'ordre sur les dates de tâche est moins triviale et nécessite d'abord d'introduire la notion de « plus grande date commune ».

Définition 4.3.2 (Plus grande date commune) *La plus grande date commune à deux dates v et w, noté $v \wedge w$, est le plus grand facteur gauche commun à ces deux mots, c'est-à-dire le plus grand mot z tel qu'il existe deux mots v' et w' vérifiant $v = z.v'$ et $w = z.w'$.*

Considérons trois tâches T, T' et T'' dans un MHT donné. Soient d, d' et d'' trois dates de tâches respectivement associées à T, T' et T'' (Fig. 4.12). Si T est un ancêtre de T', ou de manière équivalente, si T' est un descendant de T, alors $d \wedge d' = d' \wedge d = d$. Par définition, deux tâches T' et T'' sont disjointes si et seulement si T' n'est ni un descendant ni un ancêtre de T''. Dans ce cas, il existe un ancêtre commun dans l'arbre des tâches (au pire la racine), dont la date associée est $d' \wedge d''$, c'est-à-dire la date d de T sur la figure 4.12. Les tâches-frères sont des exemples de tâches disjointes dont l'ancêtre commun le plus proche n'est autre que le père. Par extension, on dira que deux dates de tâches sont disjointes si les tâches associées sont disjointes. De même, on dira qu'une date de tâche est ancêtre (resp. descendant) d'une autre date, si les tâches associées vérifient cette relation.

Définition 4.3.3 (Relation d'ordre) *Considérons un MHT Ω et un ensemble de dates atteintes $\hat{\mathcal{D}}_T(\Omega)$. Il est possible de définir une relation d'ordre $<$ stricte et partielle sur cet ensemble de la manière suivante*[1]*. Soient $v = v_0.v_1 \ldots v_{m-1}$ et $w = w_0.w_1 \ldots w_{n-1}$ deux dates dans $\hat{\mathcal{D}}_T(\Omega)$ respectivement de tailles m et n. On a $v < w$ si et seulement si $v_k < w_k$ avec $k = \|v \wedge w\| + 1$.*

[1] La définition d'une relation d'ordre sur $\mathcal{D}_T$, indépendamment d'un MHT donné, n'a *a priori* pas de sens dans notre contexte.

Soit $d = d_0, d_1 \dots d_{n-1}$ une date de tâche de taille n. Intuitivement, les frères de d d'indice plus petit que d_{n-1} sont strictement inférieurs à d, de même que leurs descendances, et que toutes les dates strictement inférieures aux ancêtres de d. On note d^- cet ensemble de dates strictement inférieures à d, représenté sur la figure 4.12. La relation $<$ permet uniquement de comparer les dates de tâches disjointes entre elles sur $\hat{\mathcal{D}}_T(\Omega)$. Si $v < w$, cela signifie que la tâche associée à v s'est exécutée avant la tâche associée à w. Par exemple, on vérifie dans le MHT Ω que $0 < 1 < 2$, ce qui signifie que la tâche *begin* s'exécute avant la tâche *mainloop*, qui s'exécute avant la tâche *end*. De même, la première itération du corps de boucle *mainbody* s'exécute bien avant la deuxième, ce qui se traduit par la relation $1.0 < 1.1$. En revanche, on ne sait pas comparer les dates non disjointes (descendant ou ancêtre) entre elles, comme par exemple 1 et 1.0. En effet, la définition d'une relation d'ordre entre une tâche et une sous-tâche n'a *a priori* pas de sens, car ces deux tâches s'exécutent en même temps. De même, la comparaison des dates de tâches associées aux branches d'une conditionnelle, comme par exemple $1.i.0.0$ et $1.i.0.1$, n'est pas non plus défini, car ces deux dates s'excluent mutuellement de l'ensemble des dates atteintes $\hat{\mathcal{D}}_T(\Omega)$. Les autres relations $>, \neq, \leq, \geq$ peuvent être définies de manière similaire sur $\hat{\mathcal{D}}_T(\Omega)$.

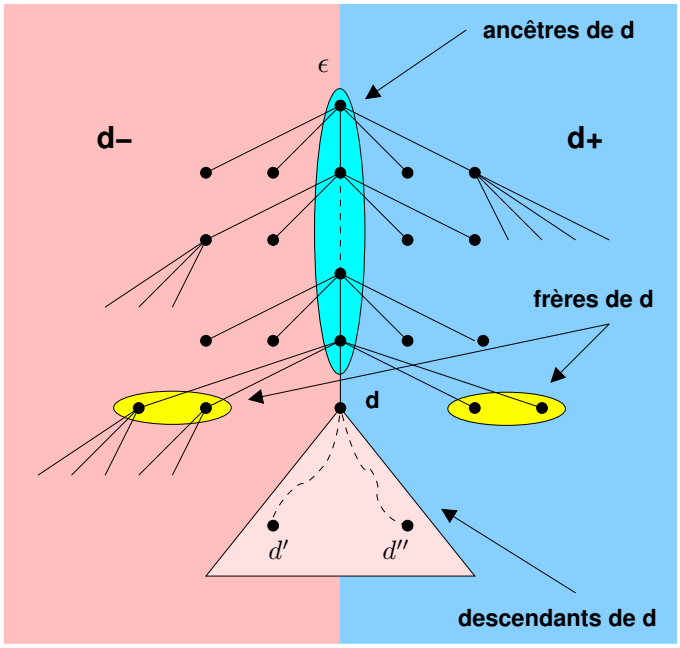


FIG. 4.12 – Frères, ancêtres, descendants, dates inférieures (d^-) et supérieures (d^+).

4.3.2 Les dates de point

Comme nous venons de le voir, les dates de tâche permettent de se repérer simplement dans un MHT, en prenant en compte l'évolution en temps des tâches en boucle. Nous allons maintenant préciser le repérage dans le flot d'exécution d'un programme en introduisant les *dates de points*, qui – comme leur nom l'indique – sont associées aux points d'instrumentation du MHT, marquant le début et la fin de chaque tâche. Dans cette section, nous allons présenter la définition d'une relation d'ordre entre les dates de point. Cette relation d'ordre est fondamentale dans notre modèle, car elle va permettre à la fin de ce chapitre de construire un algorithme de coordination efficace nécessitant de comparer des dates entre les différents processus.

Définition 4.3.4 (Date de point) *Une date de point est une paire de la forme (d, p), constituée d'une date de tâche $d \in \mathcal{D}_T$ et d'un entier $p \in \{0, 1\}$ marquant le début 0 ou la fin 1 de la tâche associée à d.*

On note $\mathcal{D}_P = \mathcal{D}_T \times \{0, 1\}$ l'ensemble des dates de point. De même, on note $\mathcal{D}_P(\Omega) = \mathcal{D}_T(\Omega) \times \{0, 1\}$ l'ensemble des dates de point atteignables par le MHT Ω et $\hat{\mathcal{D}}_P(\Omega)$ l'ensemble des dates de point atteintes à l'issue d'une exécution. Par soucis de clarté, nous noterons b et e, les entiers 0 et 1 marquant respectivement le début (*begin*) et la fin (*end*) des tâches. Deux dates de point sont égales, $(d, p) = (d', p')$, si et seulement si $d = d'$ et $p = p'$.

Définition 4.3.5 (Relation d'ordre) *Considérons un MHT Ω et un ensemble de dates atteintes $\hat{\mathcal{D}}_P(\Omega)$. Il est possible de définir une relation d'ordre $<$ stricte et totale sur cet ensemble de la manière suivante. Soient (d, p) et (d', p') deux dates dans $\hat{\mathcal{D}_\mathcal{P}}$. On a $(d, p) < (d', p')$ si et seulement si une des quatre propositions suivantes est vraie :*

1. *$d = d'$ et $p < p'$ (même tâche),*
2. *$d < d'$ (tâches disjointes),*
3. *d est un ancêtre de d' et $p = b$,*
4. *d est un descendant de d' et $p' = e$.*

Contrairement aux dates de tâche, cette relation est une relation d'ordre *total* sur $\hat{\mathcal{D}}_P(\Omega)$, car il est toujours possible de comparer deux dates de points atteintes pour un MHT donné. Lorsque d'une date de point (d, p) est strictement inférieure à une autre date (d', p'), cela signifie que le point d'instrumentation associé à la première date sera rencontré par le flot d'exécution avant le point d'instrumentation associé à la seconde date. Cela est trivialement vrai pour la première proposition : le point *begin* d'une tâche s'exécute avant le point *end* de cette même tâche. Dans le cas où les tâches sont disjointes (proposition 2), nous disposons déjà d'un ordre sur les dates de tâches, ce qui permet d'affirmer qu'une tâche (ainsi que ses points d'instrumentation) s'exécutera avant une autre. Les deux dernières propositions correspondent aux cas où les tâches ne sont pas disjointes. Dans la proposition 3, il est dit que le point *begin* d'une tâche s'exécutera avant tout autre point associé à une sous-tâche. Et inversement pour le point *end* dans la proposition 4. A titre d'exemple, on peut vérifier les relations suivantes sur le MHT Ω : $(\epsilon, b) < (0, b) < (0, e) < (1, b) < (1.1, b) < \ldots < (1, e) < (2, b) < (2, e) < (\epsilon, e)$. Les autres relations $>, \neq, \leq, \geq$ peuvent être définies de manière similaire dans $\hat{\mathcal{D}_\mathcal{P}}$.

4.3.3 Modèle d'exécution

Nous allons maintenant introduire les notions de *date courante* et de *parcours du MHT*, afin de modéliser l'exécution d'un programme. On se donne un certain MHT Ω représentant un programme parallèle (SPMD) à M processus (notés P_i avec $0 \leq i < M$). Lors de l'exécution du programme, le flot d'exécution de chaque P_i rencontre les différents points d'instrumentation placés dans le code par l'utilisateur autour des tâches qu'il a décidé d'instrumenter.

Définition 4.3.6 (Date courante) *La date courante d'un programme séquentiel est la dernière date de point rencontrée par le flot d'exécution du programme. Dans le cas d'un programme parallèle, nous allons simplement considérer un ensemble de M dates courantes associées à chaque processus P_i. On note $(c_0, c_1, c_2, \ldots, c_{M-1})$ la date parallèle courante avec $c_i \in \mathcal{D}_P(\Omega)$ la date courante du processus P_i.*

Outre la notion de date courante, la notion de version courante d'une donnée (ou *release*) est également importante dans notre modèle. La modification d'une donnée et donc de sa version courante survient à chaque fois que le flot d'exécution traverse un tâche possédant un contexte d'accès de type *modified* pour cette donnée (cf. Sec. 4.2.3). Cette information est particulièrement utile pour contrôler et garantir que les données (distribuées) envoyées à un client sont cohérentes en temps, dans le sens où elles proviennent de la même tâche, à la même date.

Définition 4.3.7 (Version courante d'une donnée) *La version courante d'une donnée est la dernière date de tâche d à laquelle la donnée a été modifiée dans le MHT.*

L'établissement de la date courante (d', p') lors de l'exécution dépend d'une part de la date courante précédente (d, p) et d'autre part du nouveau point d'instrumentation rencontré. L'évolution de la date courante peut alors se résumer par un ensemble de *règles de transition* de la forme $(d, p) \rightarrow (d', p')$, telles que $(d, p) < (d', p')$. Soit $d = d_0.d_1 \ldots d_{n-2}.d_{n-1}$ un date de tâche de taille n. L'écriture de ces règles nécessite l'introduction de quelques opérations élémentaires sur les dates de tâche :

$$\begin{aligned} d^0 &= d_0.d_1 \ldots d_{n-2}.0 \\ d^{+1} &= d_0.d_1, \ldots d_{n-2}.d_{n-1}.0 \\ d^{-1} &= d_0.d_1 \ldots d_{n-2} \\ d + k &= d_0.d_1 \ldots, d_{n-2}.(d_{n-1} + k) \end{aligned}$$

Nous allons maintenant résumer l'ensemble des règles de transition que nous avons établi pour notre modèle (Tab. 4.1) :

- Tout d'abord, le passage d'un point *begin* au point *end* de la même tâche ne modifie pas la date de tâche courante ($d' = d$). On marque simplement la fin de la tâche courante ($p' = e$).
- Le passage d'un point *begin* au point *begin* d'une sous-tâche implique la descente d'un niveau dans la hiérarchie et donc l'ajout d'un indice supplémentaire à la date courante ($d' = d^{+1}$). Typiquement, le flot d'exécution doit commencer par exécuter la première sous-tâche (d'indice 0), sauf dans le cas d'une tâche conditionnelle, où le flot d'exécution saute directement au début de la i-ème sous-tâche. Pour plus de robustesse, nous avons choisi de tolérer dans notre modèle le passage d'une tâche composée directement à la i-ème sous-tâche, car cela ne remet pas en cause l'évolution toujours croissante de la date courante.
- Le passage d'un point *end* au point *begin* d'une tâche-frère d'indice i implique un déplacement dans le MHT à un même niveau de la hiérarchie, typiquement au sein d'une tâche composée. Il s'agit en général de passer de la fin d'une sous-tâche au début de la sous-tâche suivante, c'est-à-dire $i = d_{n-1} + 1$ avec d_{n-1} le dernier indice de la date courante d. Pour plus de robustesse, on autorise le flot d'exécution à sauter directement à la i-ème tâche-frère, sous réserve que cette tâche se situe bien après la tâche courante (i.e. $i > d_{n-1}$).
- Le passage d'un point *end* au point *begin* de la même tâche implique le passage à l'itération suivante dans un corps de boucle. Dans ce cas, le dernier indice de la date courante est automatiquement incrémenté sur ce point ($d + 1$).
- Finalement, le passage d'un point *end* au point *end* de la tâche-père implique la remontée d'un niveau dans la hiérarchie et donc le retrait du dernier indice à la date courante (d^{-1}).

date courante	**point d'instrumentation rencontré**	**nouvelle date courante**
(d, b)	fin même tâche	(d, e)
(d, b)	début i-ème sous-tâche	$(d^{+1} + i, b)$
(d, e)	début i-ème tâche-frère ($i > d_{n-1}$)	$(d^{0} + i, b)$
(d, e)	début même tâche (cas boucle)	$(d + 1, b)$
(d, e)	fin tâche-père	(d^{-1}, e)

TAB. 4.1 – Règles de transition pour l'établissement de la date courante.

L'un des points clés augmentant la robustesse de notre système tient au fait qu'il est possible, grâce à la connaissance statique du MHT, de détecter les transitions interdites. En effet, toutes les transitions non énumérées dans le tableau 4.1 conduisent à des cas d'erreurs qui seront automatiquement détectées à l'exécution et signalées à l'utilisateur. Ces erreurs peuvent survenir pour plusieurs raisons. Tout d'abord, il peut s'agir d'une erreur d'instrumentation du code, non cohérente avec la description du MHT (e.g. oubli d'un point d'instrumentation, inversion de deux points, *etc*.). Une erreur peut également survenir si le flot d'exécution « normal » du programme est rompu, comme par exemple lors du déclenchement d'une exception dans une tâche. Dans ces deux cas, le flot d'exécution va rencontrer un point d'instrumentation non attendu, à partir duquel on pourra continuer l'exécution en corrigeant la date courante (i.e. tolérance aux fautes).

Définition 4.3.8 (Trace d'exécution) *La trace d'exécution dans un MHT correspond à la séquence des dates courantes rencontrées lors de l'exécution du programme. Nous notons cette séquence $(t_0, t_1, t_2, \ldots)$ avec $t_i = (d_i, p_i)$ la i-ème date de point courante.*

Une trace d'exécution débute toujours par la date (ϵ, b) marquant l'entrée dans la tâche principale du MHT et s'achève normalement lorsque la date (ϵ, e) est atteinte. A titre d'exemple, voici une trace d'exécution possible pour le MHT Ω :

$$((\epsilon, b), (0, b), (0, e), (1, b), (1.0, b), (1.0.0, b), (1.0.0.1, b), (1.0.0.1, e), (1.0.0, e), \ldots, (\epsilon, e))$$

L'exécution d'un programme génère dynamiquement une trace d'exécution, qui induit *un parcours* dans le MHT. Ce parcours reflète l'exécution du programme (relativement à l'instrumentation du code) dont on peut suivre l'évolution discrète sur le MHT. Un parcours est dit *valide*, si chaque transition de la forme $t_i \rightarrow$

t_{i+1} dans la trace d'exécution vérifie une des règles de transition énumérées précédemment (cf. Tab. 4.1). En d'autres termes, nous pouvons dire qu'un parcours est valide si les dates de point rencontrées sont toujours strictement croissantes et conformes à un chemin possible dans le MHT. Dans le cas parallèle (SPMD), tous les processus partagent le même code, la même instrumentation, et donc la même représentation en MHT. Cela ne veut pas dire pour autant que tous les processus suivent exactement le même flot d'exécution. En effet, certaines conditions (portant sur le rang du processus) peuvent induire des exécutions différentes, mais aussi des parcours différents dans le MHT si l'instrumentation est suffisamment fine pour capturer ces différences (e.g. cas d'une tâche d'envoi d'un message sur un processus et tâche de réception du message sur un autre processus). Afin de simplifier notre modèle, nous allons poser une hypothèse supplémentaire :

Définition 4.3.9 (Parcours SPMD strict d'un MHT) *Le parcours d'un MHT est dit « SPMD strict » si et seulement si tous les processus P_i possèdent une trace d'exécution identique.*

Intuitivement, cela signifie que tous les P_i vont rencontrer la même séquence de points d'instrumentation aux mêmes dates. Nous insistons sur le fait que cela n'impose pas à l'exécution du programme d'être elle-même strictement SPMD, ni même aux processus d'être synchronisés. Par exemple, la date parallèle courante sur la figure 4.13 est (t_2, t_1, t_2, t_2). En revanche, cela suppose que tous les processus rencontreront la même séquence de dates, dans le même ordre : $t_0, t_1, t_2, t_3, t_4, \ldots$

Par la suite, nous allons uniquement considérer des parcours de MHT « SPMD strict », car cette hypothèse est nécessaire à l'algorithme de coordination que nous proposons à la fin de ce chapitre (cf. Sec 4.4.2). Cette hypothèse est également importante pour l'établissement de la version courante de données distribuées. En pratique, cela suppose que l'instrumentation du code réalisée par l'utilisateur respecte quelques règles simples. Tout d'abord, l'utilisateur doit décider de la bonne granularité de son instrumentation, afin de ne considérer que des « tâches SPMD », c'est-à-dire qui seront exécutées par tous les processus. Dans le cas d'une conditionnelle, tous les processus doivent obligatoirement choisir la même branche à l'exécution (e.g. choix d'une méthode numérique). De même, dans le cas d'une tâche en boucle, tous les processus doivent effectuer le même nombre d'itérations.

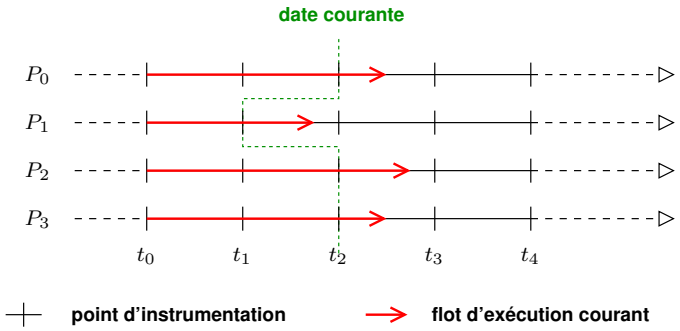


FIG. 4.13 – Schématisation d'un parcours SPMD « strict » : tous les processus P_i rencontrent la même séquence de points d'instrumentation aux dates t_0, t_1, *etc.*

4.4 Modèle de pilotage

Dans cette section, nous proposons la définition d'un modèle de pilotage basé sur le modèle de description précédent. Il permet de clarifier les interactions de pilotage que nous visons dans la plate-forme EPSN. Ce modèle cherche principalement à mettre en place les abstractions nécessaires permettant de piloter un code de simulation parallèle de manière efficace et cohérente, en particulier, afin de réaliser de la visualisation parallèle en ligne et de construire des interactions complexes. Le modèle de pilotage que nous proposons est entièrement basé sur un système de requêtes dirigées par le client via une interface utilisateur (construite grâce à l'API *front-end*). Cette approche se distingue des systèmes de pilotage par flux d'événements dont les interactions sont principalement dirigées par des événements produits par la simulation. Notons que dans cette section, nous

ne cherchons pas à spécifier l'architecture d'un environnement de pilotage, mais plutôt à proposer un modèle abstrait de pilotage répondant à nos objectifs. Nous verrons au chapitre 6 concernant l'environnement EPSN, un exemple d'implantation de ce modèle basée sur la technologie CORBA.

4.4.1 Le pilotage par les requêtes

Le modèle de pilotage que nous proposons est basé sur une relation client/serveur, entre d'une part la simulation jouant le rôle de serveur et d'autre part l'interface utilisateur jouant le rôle de client. Dans nos travaux, la simulation et l'interface utilisateur peuvent tous deux être des programmes parallèles. Toutefois, si les simulations que nous souhaitons piloter respectent un modèle de programmation SPMD, les programmes de visualisation parallèle utilisent généralement une approche maître/esclaves. Les esclaves sont typiquement responsables de calculer une image de la simulation en parallèle, tandis que le maître est un processus interactif avec lequel l'utilisateur dialogue par le biais d'un interacteur (affichage, manipulation de la souris, du clavier, *etc*.). En principe, seul le processus maître a la possibilité de déclencher des actions de pilotage vers la simulation, ce qui va donc simplifier notre modèle. Notons que le maître joue assez fréquemment le rôle d'un esclave en participant également à la visualisation et au rendu en parallèle. En résumé, la simulation joue le rôle d'un *serveur parallèle* à M processus, c'est-à-dire que chaque processus de la simulation joue le rôle d'un serveur à l'écoute de requêtes clientes. Côté interface utilisateur, nous allons considérer un programme de visualisation parallèle à N processus, dont le processus maître (disons le processus 0) jouera la rôle du client de pilotage. Notons que dans cette approche, la visualisation séquentielle est un cas particulier où N sera pris égal à 1.

Dans notre modèle, nous allons distinguer deux types de requêtes : les requêtes de *monitoring* et les requêtes de *steering*. Les requêtes de *monitoring* permettent d'observer l'état de la simulation, tandis que les requêtes de *steering* servent à modifier le comportement de cette simulation. Afin de simplifier notre modèle, nous avons choisi de clairement séparer ces deux types de requêtes. Ainsi, les requêtes de *steering* ne retournent pas de résultats sur les traitements effectués autres qu'un acquittement. Il est donc nécessaire de déclencher une requête de *monitoring* dans un deuxième temps afin d'observer le résultat produit. Nous pouvons formuler plusieurs remarques sur les différences clés qui existent entre ces deux types de requêtes. Tout d'abord, nous pouvons considérer que les requêtes de *steering* sont des événements plutôt rares dans la vie d'une simulation interactive, par opposition aux requêtes de *monitoring* qui sont généralement beaucoup plus fréquentes (e.g. tous les k pas de temps). Autre différence : les requêtes de *steering* sont particulièrement critiques pour la simulation, car elles modifient son comportement et peuvent conduire à des erreurs graves en cas d'échec de la requête[2]. Les requêtes de *monitoring* quant à elles sont *a priori* sans risque pour la simulation, même en cas d'échec. Par conséquent, nous aurons tendance à privilégier la robustesse pour les requêtes de *steering* et la performance pour les requêtes de *monitoring*.

Le modèle client/serveur permet de supporter naturellement plusieurs utilisateurs (serveur multi-threads). Les requêtes sont concurrentes car elles peuvent être émises par plusieurs clients différents et être traitées simultanément par l'environnement de pilotage (côté simulation). Toutefois, la gestion concurrente des requêtes, si elle permet d'obtenir globalement une bonne réactivité du système, pose certaines difficultés, qui nécessitent la mise en place d'une *politique de collaboration*. Nous nous sommes fixés la règle suivante : une requête de *steering* portant sur un certaine donnée du code ne doit être concurrente à aucune autre requête (de *monitoring* ou de *steering*) portant sur cette même donnée. Par exemple, il n'est pas possible de modifier une donnée pendant qu'un autre client la consulte, et inversement. En revanche, deux clients peuvent consulter simultanément une même donnée. Nous verrons dans l'implantation d'EPSN comment il est possible de mettre en place cette politique d'exclusion mutuelle à l'aide d'un jeton.

Le cycle de vie des requêtes

Le système de pilotage est découpé en quatre étapes clés : (1) l'envoi de la requête, (2) une phase de coordination du traitement, (3) le traitement de la requête et (4) l'acquittement de la requête. Nous allons

[2]Erreurs pouvant survenir suite à une erreur d'instrumentation, une « coupure réseau » provoquée par une déconnexion brutale du client distant, *etc*.

maintenant détailler les étapes du cycle, représentées sur la figure 4.14.

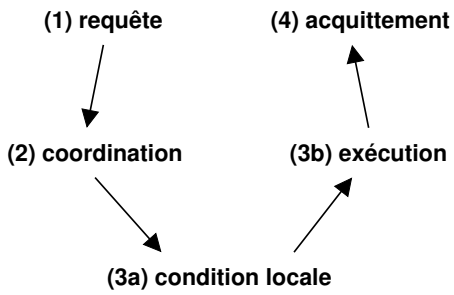


FIG. 4.14 – Le cycle de vie des requêtes : (1) envoi de la requête, (2) coordination des traitements, (3a) évaluation d'une condition locale, (3b) exécution effective de la requête, (4) acquittement de la requête.

Depuis une application cliente, le processus maître envoie une requête (*req*) à tous les processus de la simulation. Les requêtes que nous considérons sont typiquement asynchrones, pour ne bloquer ni le client qui émet la requête, ni la simulation qui reçoit la requête. En effet, on peut assimiler les codes de simulation et de visualisation à des « systèmes bouclés » (boucle de calcul, boucle de rendu), dont les interactions de pilotage perturbent le déroulement normal (calculer côté simulation, afficher et naviguer dans la scène côté visualisation). Ainsi, le système de requêtes que nous souhaitons mettre en place doit être doublement asynchrone. D'une part, il faut permettre à l'utilisateur de conserver le maximum d'interactivité avec l'UI, même si la requête tarde à s'achever, même si la requête échoue. D'autre part, il faut limiter la perturbation côté simulation due à la réception et au traitement des requêtes. Tout ces objectifs nécessitent la plus grande flexibilité de la part du système de requêtes, principalement caractérisé par la « non-prévisibilité » des requêtes déclenchées par l'utilisateur et par la durée des traitements en réponse à ces requêtes. Par la suite, on notera $(req\ \ id)$ une requête simple de type *req* portant sur un objet *id*, cet argument pouvant être optionnel selon le type de la requête. Chaque requête est identifiée par un entier, noté *rid* pour *request ID*. En particulier, le client a la possibilité d'attendre explicitement la fin de la requête grâce à un ordre *wait* ou encore d'annuler explicitement le traitement d'un requête en cas d'échec (ordre *cancel*).

La réception de la requête côté simulation (étape 1) déclenche une phase de coordination visant à planifier une date de traitement[3] (étape 2). La planification repose sur un algorithme de coordination mettant en jeu l'ensemble des processus de la simulation. Cet algorithme permet d'assurer que le traitement de la requête commencera à la même date t_p sur tous les processus. Le point fort de cet algorithme est qu'il permet d'effectuer des traitements parallèles globalement cohérents en temps sans recourir à une synchronisation forte entre les processus de la simulation. Nous présenterons en détail cet algorithme dans la section 4.4.2.

Lorsque la date planifiée (t_p) est atteinte par un processus dans la simulation, débute alors le traitement de la requête de manière tout à fait indépendante des autres processus (étape 3). Ce traitement se décompose en deux étapes élémentaires : une première étape d'évaluation d'une *condition locale* (étape 3a) et une deuxième étape d'exécution effective de la requête (étape 3b). L'utilisation d'une condition locale permet de retarder l'exécution de la requête sur un processus à une date t_e, à partir de laquelle la condition est vérifiée. Cette condition permet typiquement d'assurer la cohérence spatiale des traitements à effectuer, comme par exemple pour vérifier l'accès à une certaine donnée (pour une requête d'extraction des données). Comme nous l'avons vu dans la section 4.2.3, l'exécution de la requête de pilotage pourra s'effectuer soit de manière bloquante sur un point d'instrumentation, soit de manière concurrente tout au long d'une tâche, selon la nature du mécanisme d'interaction utilisée : point ou plage d'interaction. Dans ce dernier cas, l'exécution aura lieu pendant un intervalle de la forme $[t_e, t_{e'}]$, appelé *intervalle d'exécution*, durant lequel la condition locale sera vraie. Notons que dans le cas d'une interaction en point, on a $t_e = t_{e'}$. La figure 4.15 résume bien le principe de traitement

[3]Dans la suite de ce chapitre, le mot « date » renverra à la notion de *date de point* définie en 4.3.4, et pour laquelle nous disposons d'une relation d'ordre total.

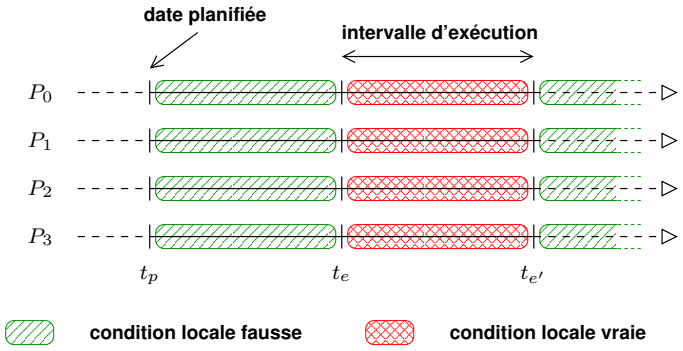


FIG. 4.15 – Traitement d'une requête en parallèle.

d'une requête en parallèle. Afin de conserver la cohérence temporelle acquise pendant la phase de coordination, il est important que tous les processus effectuent un parcours SPMD strict du MHT à partir de t_p et que la condition locale conduise au même intervalle d'exécution $[t_e, t_{e'}]$ sur tous les processus. Dans notre modèle, nous respectons naturellement cette contrainte car nous attachons des informations contextuelles (servant à l'évaluation des conditions locales) directement aux tâches d'un MHT. Par exemple, nous avons vu dans la section 4.2.3 comment il était possible de configurer les droits d'accès aux données dans un MHT. Ainsi lors d'un parcours du MHT, les tâches vont définir des intervalles d'exécution de la forme $[(d_e, b), (d_e, e)]$ avec d_e la date de la tâche considérée, s'étendant du début à la fin de la même tâche. Toutefois, il arrive fréquemment que ces intervalles d'exécution soient tronqués par une ou plusieurs sous-tâches surchargeant le contexte du père[4]. Si le traitement n'est pas achevé à l'issue de l'intervalle d'exécution, le point d'instrumentation marquant la fin de l'intervalle devient bloquant pour la simulation jusqu'à la fin du traitement (e.g. envoi de données au client). Ce mécanisme est susceptible d'introduire un ralentissement de la simulation si le recouvrement du traitement n'est pas total, mais il permet de garantir que le traitement de la requête reste circonscrit à l'intervalle d'exécution prévu à cet effet (cohérence spatiale). La nature du traitement à effectuer varie selon le type de la requête, comme nous allons le voir dans la section suivante. Remarquons que si le traitement d'une requête est systématiquement initié sur les processus de la simulation, il peut naturellement impliquer des processus distants, comme par exemple lors du transfert de données d'un code vers l'autre.

Finalement, la terminaison de la requête (étape 4) est signalée par un acquittement, éventuellement optionnel. Cet acquittement, noté *ack*, vise à renseigner le client et/ou les différents processus impliqués dans les traitements que la requête est globalement terminée.

Requêtes simples

Nous allons maintenant décrire plus en détails les requêtes que nous envisageons. Plus précisément, nous avons choisi de décomposer notre système de pilotage en plusieurs requêtes simples : la requête *get*, la requête *put*, les requêtes de contrôle et d'action.

Requête *get*. La requête *get* est une requête de *monitoring* qui permet d'extraire une donnée de la simulation et de la transmettre au client. Comme le suggère la figure 4.16, cette requête entraîne la circulation d'un flot de données (*data flow*) de la simulation vers le client, typiquement pour construire une représentation graphique des informations extraites. Le traitement de la requête est contrôlé par une condition locale d'accès en lecture à la donnée, qui retarde l'envoi des données tant que l'accès à la donnée n'est pas autorisé. L'exécution de la requête tire ensuite profit des plages d'accès définis dans le MHT pour recouvrir le transfert sur les tâches de calcul de la simulation (Sec. 4.2.3). Cette étape de transfert est relativement élémentaire dans le cas de codes séquentiels ou de données non distribuées. En revanche, le schéma de communication se complique singulièrement

[4]Par exemple, dans le MHT Ω, la tâche *mainloop* ouvre une large plage d'accès $[(1, b), (1, e)]$ qui est fermée périodiquement au niveau de la sous-tâche *subloop* $[(1.i.1, b), (1.i.1, e)]$. D'une itération à l'autre, l'intervalle d'exécution résultant sera de la forme $[(1.i.1, e), (1.(i+1).1, b)]$.

lorsque nous souhaitons visualiser en parallèle une donnée distribuée d'une simulation elle-même parallèle. Dans ce cas, chaque processus de la simulation communique une partie de ces données à un processus de la visualisation, ce qui conduit à un schéma de communication comparable à la routine *all-to-all* de MPI. Au plus, $M \times N$ messages seront échangés « en parallèle » entre ces deux codes (flot de communication parallèle). Le calcul de ces messages dépend directement de la nature des données considérées et de leurs distributions respectives. Nous proposons au chapitre 5 plusieurs solutions pour résoudre le problème délicat de la *redistribution des données*.

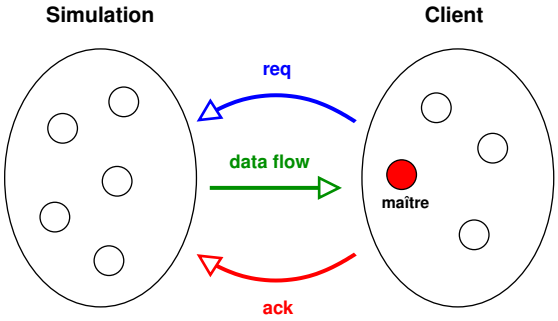


FIG. 4.16 – La requête *get*.

Requête *put*. La requête *put* est une requête de *steering*, qui va jouer un rôle symétrique à la requête *get*. Elle sert à modifier des données dans le code de simulation, en transférant un flot d'informations depuis le code client vers la simulation (Fig. 4.17). Le traitement de la requête est contrôlé par une condition locale d'accès en écriture à la donnée. Lorsque cette condition est vérifiée, la simulation signale explicitement au client (envoi de la requête *ready*) qu'il est prêt à recevoir les données. Le transfert peut alors débuter (*data flow*). Même si – en principe – l'exécution de cette requête pourrait bénéficier d'un mécanisme d'interaction en plage, nous avons choisi de contraindre son exécution sur des points d'interaction. Comme nous l'avons vu, ces points peuvent être simplement configurés grâce à des contextes d'accès dans le MHT (Sec. 4.2.3). Comme dans le cas d'un *get*, l'étape de transfert peut entraîner une redistribution des données entre le code de visualisation et le code de simulation, lorsque ces deux codes sont parallèles. La requête *put* constitue un premier niveau d'interaction simple avec la simulation, permettant par exemple de modifier un paramètre scalaire du modèle (e.g. une condition aux limites) ou une donnée distribuée complexe (e.g. la position des atomes dans un code de dynamique moléculaire).

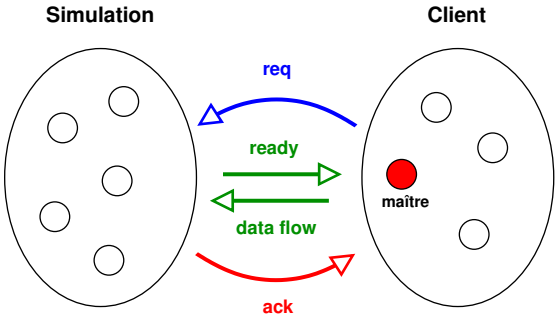


FIG. 4.17 – La requête *put*.

Il est important de noter que, dans notre modèle, le traitement des requêtes est toujours amorcé par les processus de la simulation, et cela pour la simple et bonne raison que l'exécution effective de la requête dépend

d'une condition locale à la simulation. Cela semble évident, si l'on considère le cas d'une requête *get*, car le transfert des données vers la visualisation ne pourra débuter sur les processus de la simulation que lorsque l'accès en lecture à la donnée sera effectivement autorisé. En revanche, cette stratégie est plus discutable dans le cas de la requête *put*. Dans ce cas, nous commençons par vérifier l'accès en écriture à la donnée (côté simulation) avant de déclencher le transfert des données de l'application cliente vers la simulation, grâce à l'envoi d'un ordre *ready* (Fig. 4.17). Une solution alternative consisterait à envoyer les données « au plus tôt », à les stocker temporairement côté simulation, puis finalement à les copier dans la simulation lorsque la condition serait vérifiée. Nous trouvons plusieurs avantages à notre approche. Premièrement, elle permet d'unifier notre modèle de pilotage, en séparant proprement l'étape d'envoi de la requête et de traitement de la requête. Deuxièmement, elle permet d'éviter d'engorger la mémoire de la simulation avec une copie temporaire des données, dont on ne peut prévoir à l'avance combien de temps elle sera stockée. A l'inverse, notre stratégie garantit que les données seront placées en mémoire immédiatement, éventuellement sans aucune copie si la couche de communication le permet.

Requête d'action. La requête *action* est une requête de *steering* qui permet de déclencher l'exécution d'une fonction définie par l'utilisateur dans le code de la simulation, également appelée fonction de rappel ou *callback* (Fig. 4.18). Cette action est associée à un identifiant unique (*id*) enregistré sous forme d'une chaîne de caractères, comme nous l'avons vu à la section 4.2.4 (exemple de l'action *myaction* associée au point *mypoint* dans le MHT Ω). Contrairement aux requêtes *get* et *put*, cette requête implique *a priori* aucun transfert de données entre la simulation et le client. Les *callbacks* peuvent servir à définir des points de sauvegarde/reprise (*checkpoint*) ou encore à commander des opérations comme le raffinement d'un maillage ou encore l'application d'un champs de force. Les actions, tout comme le *put*, sont souvent des requêtes critiques pour la simulation, ne pouvant généralement s'exécuter qu'en des endroits très précis du code. Pour ces raisons, nous avons choisi de contraindre leur exécution uniquement sur des tâches en point dans le MHT, configurées pour recevoir ces actions à l'aide d'un contexte particulier (le contexte *action*).

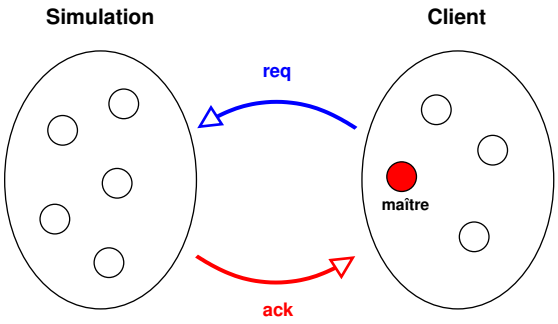


FIG. 4.18 – La requête *action*.

Requêtes de contrôle. Les requêtes de contrôle (*play*, *step*, *stop*) sont des requêtes de *steering* particulières servant à contrôler le flot d'exécution de la simulation, de manière comparable à certaines fonctionnalités utilisées dans un débogueur. La requête *stop* permet de mettre en pause la simulation sur un point d'instrumentation quelconque dans le code, jouant pour l'occasion le rôle de point d'arrêt. Toutefois, nous garantissons que tous les processus de la simulation seront bloqués à la même date t_p, correspondant à la date planifiée lors de la phase de coordination. Cette fonctionnalité est particulièrement utile pour analyser l'état de la simulation à une date fixe, identique sur tous les processus. La requête *play* permet alors de relancer l'exécution de la simulation après un arrêt commandé par *stop*. La requête *step* sert à faire avancer la simulation pas à pas, c'est-à-dire de point d'instrumentation en point d'instrumentation, tout en respectant la contrainte de cohérence en temps.

Les requêtes permanentes et la visualisation en ligne

Les boucles de calcul jouent un rôle de première importance dans le contexte de la visualisation en ligne. En effet, les résultats produits à chaque itération de la boucle représentent dans la plupart des simulations une évolution en temps d'un certain phénomène physique. Cette évolution est discrétisée de telle sorte qu'à chaque itération i, la simulation avance d'un certain pas de temps physique δt, marquant ainsi le calcul pour un nouvel instant t_i dans la simulation. Généralement δt est fixé pour une simulation donnée, et l'on a $t_i = t_0 + i.\delta t$ avec t_0 l'instant initial de la simulation. Le corps de la boucle se décompose traditionnellement en plusieurs tâches de calcul servant à mettre à jour les différentes grandeurs physiques du problème (cas d'une méthode de résolution explicite). Ainsi, à chaque itération de la boucle, une nouvelle version des données est produite, ce que l'on peut signaler dans notre modèle grâce au contexte *modified* (Sec. 4.2.3). La figure 4.19 donne un exemple très simple d'un MHT décrivant une boucle de calcul contenant deux sous-tâches A et B, la première modifiant à chaque itération une certaine donnée *mydata* et la seconde offrant une plage d'accès en lecture à cette donnée. Nous allons utiliser cet exemple pour illustrer notre propos dans cette section.

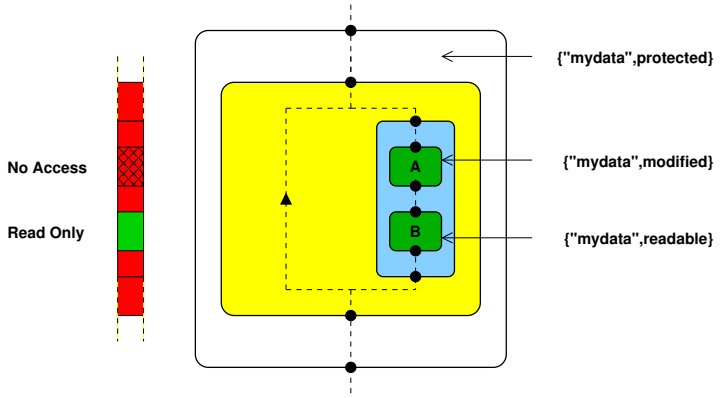


FIG. 4.19 – Exemple d'un MHT représentant une boucle de calcul.

La *visualisation en ligne* consiste à observer les résultats intermédiaires d'une simulation au fur-et-à-mesure qu'ils sont produits. Cela implique d'extraire des données périodiquement et de produire à chaque nouvelle version de ces données transmises au client une représentation graphique adaptée. Ainsi, l'utilisateur suit l'évolution des calculs en temps-réel, image par image, de manière comparable à une animation ou à un film. Remarquons que la visualisation en ligne est naturellement rattachée aux tâches en boucle dans le MHT. Dans le cas d'un programme multi-boucles, l'utilisateur devra sélectionner explicitement la boucle dont il souhaite suivre l'évolution.

Une première approche permettant de produire une visualisation en ligne consiste à répéter à une certaine période en temps (e.g. toutes les secondes) des requêtes de type *get*. L'inconvénient de cette méthode est qu'elle ne permet pas de visualiser tous les pas de temps de la simulation, ni même de garantir que le nombre de pas de temps qui sépare chaque image produite côté visualisation est constant d'une requête à l'autre. On peut trouver plusieurs raisons à cela : par exemple, la durée des pas de temps de la simulation peut varier au cours des calculs, ou encore les temps de transfert peuvent varier en fonction de la charge du réseau de manière non prévisible, *etc*. Afin de remédier à ce problème, nous avons introduit la requête *getp* (*get* permanent), qui s'apparente à une requête *get* ordinaire, mais dont le traitement sera automatiquement répété à une certaine période p, comptée en nombre d'itérations dans la boucle (Fig. 4.20).

Ainsi dans notre exemple, l'envoi de la donnée *mydata*, s'il débute à la $i^{\text{ème}}$ itération, aura lieu durant l'intervalle d'exécution $[(b_i, b), (b_i, e)]$, avec b_i la date de tâche de B à cette itération (Fig. 4.21). La version de la donnée envoyée est alors a_i, ce qui correspond à la date de la dernière tâche ayant modifiée cette donnée. Pendant toute la durée de la plage d'envoi, nous garantissons d'une part que la donnée est accessible en lecture et d'autre part que la donnée n'est pas modifiée (i.e. version de la donnée constante). Une fois la première exécution achevée, l'envoi sera automatiquement répété toutes les p itérations suivantes, c'est-à-dire dans des

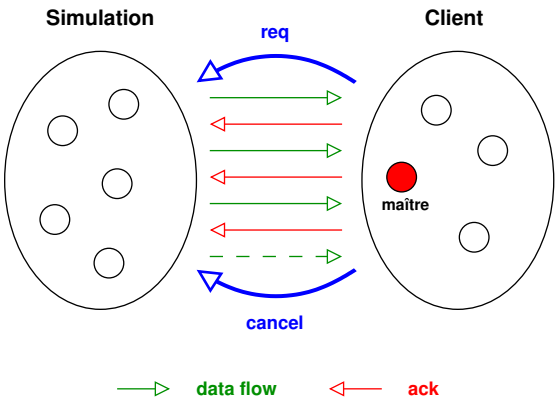


FIG. 4.20 – La requête *getp*.

intervalles d'exécution de la forme $[(b_{i+k.p}, b), (b_{i+k.p+1}, e)]$ avec k un entier strictement positif ($p = 1$ sur la figure 4.20). A chaque réception des données côté client, le système de visualisation déclenche un ensemble de traitements en cascade (i.e. pipeline de visualisation) visant à produire une image de la donnée transmise. Une fois ce traitement accompli, l'utilisateur doit explicitement acquitter la requête comme le suggère la figure 4.20. Cet acquittement (*ack*), comparable à un ordre *continue*, joue un rôle important pour la régulation des transferts. En effet, si le traitement n'est pas achevé à l'issue de l'intervalle d'exécution, le point d'instrumentation marquant la fin de l'intervalle (i.e. la fin de la tâche B) devient bloquant jusqu'à la réception de l'acquittement. Sans l'utilisation d'un tel mécanisme, une simulation rapide aurait tôt fait d'engorger le client de visualisation ne parvenant pas à « digérer » l'ensemble des données reçues. En contrepartie, ce mécanisme de régulation a pour effet de ralentir la simulation, du moins si l'intervalle d'exécution ne permet pas de recouvrir l'ensemble des traitements (envoi de données, visualisation, acquittement). Ce ralentissement peut être relativement conséquent si l'utilisateur souhaite visualiser chaque itération ($p = 1$). En utilisant une période d'envoi plus grande, il est possible de diminuer ce surcoût pour ne pas pénaliser trop fortement la simulation. En pratique, l'utilisateur fixe lui-même la période d'envoi des données qui convient le mieux à ses besoins.

Comparée à la première approche (plusieurs requêtes *get* à la suite), la requête permanente *getp* est un mécanisme efficace basée sur l'envoi d'une seule requête et nécessitant *une seule et unique phase de coordination* pour planifier l'ensemble des traitements. L'exécution de la requête est alors répétée indépendamment sur tous les processus de la simulation à la période p voulue, ce qui permet de garantir que l'image produite (à partir des données collectées sur tous les processus) correspondra aux itérations de la boucle : $i, i+p, i+2.p, i+3.p, \ldots$ L'envoi des données continuera périodiquement jusqu'à la sortie de la boucle ou jusqu'à l'annulation explicite de cette requête par le client grâce à la requête *cancel*.

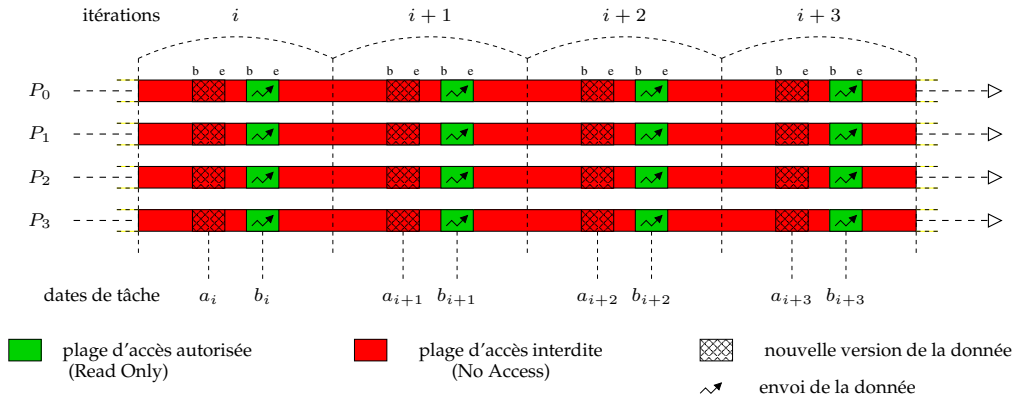


FIG. 4.21 – Traitement d'une requête *getp* ($p = 1$).

En conclusion, nous voulons étendre notre propos en soulignant que la notion de *requête permanente* n'est pas spécifique à l'extraction de données. Nous pouvons imaginer utiliser ce type de requête dans un autre contexte, comme par exemple pour commander une action répétée périodiquement dans la simulation (e.g. *checkpoint*).

Les requêtes complexes

Sur la base de ce modèle, nous avons imaginé la construction d'interactions complexes, basées sur la combinaison de plusieurs requêtes d'interactions simples (*get*, *put*, *action*). L'introduction des requêtes complexes ou *multi-requêtes* dans notre modèle est principalement motivée par deux besoins : l'extraction simultanée de plusieurs données et le déclenchement d'actions paramétrées. Par définition, une multi-requête est une séquence de requêtes simples bénéficiant de la même phase de coordination sur les processus de la simulation, ce qui garantit que l'ensemble des traitements associés aux requêtes simples débuteront à la même date t_p. Par extension, on notera $(req_1\ id_1,\ req_2\ id_2,\ req_3\ id_3,\ \ldots)$ une telle requête composite. L'exécution des traitements commencera uniquement lorsque l'ensemble des conditions locales associées aux requêtes simples sera vérifié (intersection des conditions). Notons que c'est à l'utilisateur qui instrumente le code de garantir que cette condition sera vérifiée au moins en un point de la simulation. Par ailleurs, nous garantissons que l'exécution locale des requêtes simples s'effectuera dans l'ordre défini par la séquence. Ces deux hypothèses sont fondamentales dans notre modèle pour construire des actions complexes ordonnées et globalement cohérentes en temps. Finalement, l'acquittement d'une multi-requête s'effectue lorsque l'ensemble des requêtes ont achevé leurs traitements. Nous allons maintenant présenter deux applications concrètes des *multi-requêtes*.

Un premier type d'application est le *multi-get*, qui consiste à regrouper le traitement de plusieurs requêtes de type *get* sur des données différentes, ce qui peut s'écrire : $(get\ data_1,\ get\ data_2,\ get\ data_3,\ \ldots)$. En effet, il arrive fréquemment dans les codes de simulation que les données à extraire soient constituées de plusieurs séries, variant au cours du temps. De plus, l'utilisation des techniques de la visualisation scientifique encourage la composition de plusieurs séries de données pour construire une image riche en information. Par exemple, la visualisation d'un maillage non structuré se déformant sous un champs de force va nécessiter de connaître (à chaque pas de temps) les coordonnées des nœuds du maillage, outre la liste de connectivité des éléments, ainsi que les différentes grandeurs physiques aux nœuds de ce maillage (e.g. pression, vitesse). Pour construire une image du maillage globalement cohérente en temps, il ne suffit pas d'effectuer plusieurs requêtes *get* indépendantes sur les différentes séries de données. Il faut encore garantir que toutes les séries de données extraites renvoient à une même date dans le code de simulation, et donc à la même version des données associées au maillage. Grâce aux multi-requêtes, nous offrons un élément de réponse à ce problème en coordonnant le traitement de la requête de manière groupée, et en débutant le transfert dès que nous entrons dans une plage d'accès commune à toutes les données $\{data_1,\ data_2,\ data_3,\ \ldots\}$. Cela suppose que l'instrumentation exhibe une telle plage d'accès dans le code. Notons que ce mécanisme peut être défini de manière identique pour le *multi-getp*. Nous insistons sur le fait qu'il s'agit ici d'un élément de réponse, car ce mécanisme ne saurait garantir la cohérence en temps dans un cas plus général où les plages d'accès aux données seraient disjointes. Pour finir, notons qu'il est possible ici de relaxer la contrainte sur l'ordre d'exécution dans le *multi-get* afin d'augmenter la concurrence des transferts dans la plage d'envoi commune à tous les données.

Un deuxième type d'application vise à construire des *actions paramétrées*, c'est-à-dire une extension des actions simples nécessitant le passage préalable à la simulation de un ou plusieurs paramètres. Dans ce cas, la multi-requêtes est de la forme $(put\ param_1,\ put\ param_2,\ \ldots,\ action\ id)$. L'intersection des conditions locales nous conduit à exécuter ce type de requêtes complexes uniquement sur des tâches en point autorisant l'action *id* et vérifiant les droits d'accès en écriture à tous les paramètres sur ce point (contexte *writable*). De plus, l'ordre imposé sur le traitement des requêtes garantit que les paramètres seront transmis avant de débuter l'action. Nous avons imaginé d'utiliser ce type de requête pour déclencher des actions complexes comme par exemple le raffinement d'un maillage, l'application d'un champs de force dans un code de dynamique moléculaire, ou encore le pilotage de l'injection d'un fluide en mécanique des fluides, *etc*. Toutes ces actions aussi diverses soient-elles, dépendent typiquement de plusieurs paramètres scalaires décrivant l'action en elle-même (facteur de raffinement de la maille, vecteur force, intensité, direction et durée de l'injection, *etc*.). De plus, ces actions s'appliquent le plus souvent dans une région de l'espace physique délimitée, ce qui nécessite la définition d'une zone de sé-

lection où l'action sera appliquée. Une manière élégante de paramétrer ces actions consiste à utiliser des *widget 2D/3D* de type rectangle, sphère, pavé ou vecteur directement plongés dans la visualisation (cf. Sec. 3.2.3). L'utilisation de saisie textuelle peut également s'avérer pertinente pour contrôler précisément la valeur de certains paramètres physiques.

4.4.2 Algorithme de coordination

Le pilotage des simulations numériques parallèles, et plus particulièrement la visualisation en ligne des résultats intermédiaires, soulèvent un sérieux problème de cohérence temporelle. En effet, comme nous l'avons déjà souligné plusieurs fois, la collection des données distribuées vers un système de visualisation nécessite que l'envoi des « morceaux de données » survienne à la même date sur l'ensemble des processus, sans quoi aucune interprétation du résultat ne serait possible. De la même façon, une action ou la modification d'un paramètre du code doit s'exécuter précisément à la même date pour tous les processus de la simulation. L'efficacité d'un système de pilotage dépend en partie de sa capacité à coordonner des requêtes de pilotage mettant en jeu l'ensemble des processus de la simulation tout en garantissant la cohérence temporelle du traitement. Dans ce chapitre, nous présentons une nouvelle solution au problème de la cohérence temporelle, s'appliquant à des simulations décrites avec un MHT. Cette solution exploite le système de datation introduit à la section 4.3, qui permet de se repérer précisément dans le flot d'exécution de la simulation. Le principe de l'algorithme consiste à planifier le traitement de la requête, sans qu'il soit nécessaire de synchroniser l'exécution de la simulation en parallèle. Cet algorithme ne nécessite pas une connaissance statique du MHT pour fonctionner, mais repose sur l'hypothèse d'un parcours SPMD strict du MHT (cf. Déf. 4.3.9).

Planification des traitements

L'algorithme de coordination que nous proposons repose sur une stratégie de *planification* des traitements de pilotage. Cet algorithme vise à établir dans « un futur proche » la date[5] à partir de laquelle le traitement de la requête va pouvoir débuter. Comme nous l'avons vu dans la section 4.4.1, nous distinguons dans le cycle de vie d'une requête la date courante à la réception d'une requête, la date t_p planifiée pour le traitement de la requête et la date t_e d'exécution effective de la requête, qui ne débutera qu'après qu'une certaine condition ait été vérifiée localement sur chaque processus (Fig. 4.15). Dans cette section, nous nous intéressons exclusivement au calcul de la date t_p.

Définition 4.4.1 (Date planifiée) *Considérons une simulation parallèle à M processus et un MHT Ω représentant cette simulation. Soit $(c_0, c_1, c_2, \ldots, c_{M-1})$ la date parallèle courante avec $c_i \in \mathcal{D}_P(\Omega)$ la date courante du processus P_i à la réception de la requête. Soit t_{max} la plus grande date couramment atteinte par l'ensemble des P_i, c'est-à-dire $t_{max} = max\{c_0, c_1, \ldots\}$. La date planifiée t_p est la prochaine date atteinte par tous les P_i, strictement supérieure à t_{max} au sens de la définition 4.3.5. En d'autres termes, il s'agit de la première date non encore dépassée par l'ensemble des processus à la réception de la requête.*

Remarquons que s'il est possible de déterminer facilement la date t_{max} au moment de la réception de la requête, il n'en est pas de même pour la date t_p. Car il existe souvent plusieurs dates atteignables à partir d'une date donnée : choix d'une branche ou l'autre dans une conditionnelle ; passage à l'itération suivante ou sortie de la boucle. En pratique, la date t_p sera découverte dynamiquement par chaque P_i lorsqu'il dépassera la date t_{max}. L'hypothèse d'un parcours SPMD strict du MHT sert à nous garantir que cette date sera la même pour tous les processus. Telle que nous la choisissons, la date t_p apparaît comme la plus petite date à partir de laquelle il est possible de débuter le traitement d'une requête en parallèle (planification « au plus tôt »).

Algorithme

La figure 4.22 résume les trois étapes de l'algorithme de coordination que nous avons imaginé, visant à établir la date t_p sans synchroniser les processus de la simulation.

[5] Nous rappelons que, dans cette section, le mot « date » renvoie à la notion de *date de point*, introduite à la section 4.3.2, et pour laquelle nous disposons d'une relation d'ordre total.

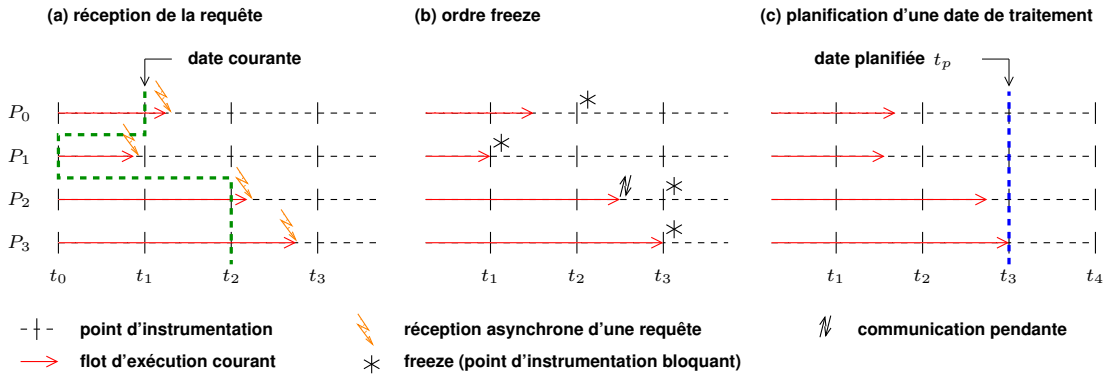


FIG. 4.22 – Algorithme de planification.

1. L'algorithme débute à la réception d'une certaine requête par tous les processus. Dans notre exemple (Fig. 4.22 (a)), la date parallèle courante à la réception de cette requête est (t_1, t_0, t_2, t_2). Comme la réception des requêtes est *a priori* asynchrone dans notre modèle, rien ne garantit que tous les processus reçoivent la requête exactement au même instant. La seule hypothèse que nous faisons est de dire que la simulation doit effectuer un parcours SPMD strict du MHT (cf. Déf. 4.3.9). Ainsi, même si le processus P_1 (en t_0) est en retard par rapport à ces voisins, nous pouvons tout de même garantir qu'il rencontrera la même séquence de points d'instrumentation que tous les autres processus aux dates t_1, t_2, t_3, *etc*.
2. L'étape suivante consiste à « geler » (i.e. rendre bloquant) l'ensemble des points d'instrumentation du code (ordre *freeze*) afin de garantir qu'aucun P_i ne dépassera la date courante c_i (établie à la réception de la requête). En pratique, nous laissons chaque P_i poursuivre son exécution normalement et s'il atteint le point d'instrumentation suivant, nous bloquons automatiquement son exécution à l'entrée de ce point[6], ce que suggère le symbole $*$ sur la figure 4.22 (b).
3. Durant l'étape qui suit, tous les processus échangent leurs dates courantes respectives afin d'établir la date t_{max} prise comme le maximum des dates courantes (équivalent d'une routine de communication *MPI_Allreduce*). Nous insistons sur le fait que les processus ne sont pas nécessairement bloqués durant cette étape. Lorsque ce calcul est effectué, chaque processus a la connaissance de la date t_{max}. On relâche alors l'ordre *freeze* et les processus éventuellement en attente sur des points d'instrumentation gelés peuvent continuer librement. La date t_p sera alors la première date rencontrée strictement supérieure à t_{max} conformément à la définition 4.4.1 (Fig. 4.22 (c)). Lorsqu'un processus rencontre finalement le point d'instrumentation à la date planifiée t_p, il peut débuter le traitement de la requête, dont nous avons rappeler les étapes sur la figure 4.23 : vérification d'une condition locale et exécution effective de la requête à partir de la date t_e.

Discussion

Comme nous venons de le voir, le calcul de la date t_{max} nécessite une opération de communication collective de type *all-reduce* dont la complexité en nombre de messages dépend du logarithme du nombre de processeurs (algorithme parallèle basé sur un arbre binaire). Dans notre contexte, nous utilisons une version asynchrone de cet algorithme *grâce à l'utilisation d'un thread* (cf. architecture d'EPSN au chapitre 6), ce qui nous évite de bloquer entièrement l'exécution des processus, comme il est *a priori* nécessaire pour une opération de ce type. Cette approche nous permet de recouvrir le calcul de t_{max} avec les calculs de la simulation. L'ordre *freeze* est là pour garantir dans cet algorithme qu'aucun processus ne dépassera la date t_{max} pendant son calcul, ce qui évidemment fausserait le résultat. Comme nous l'avons déjà dit, cet ordre peut bloquer temporairement certains processus quand ils atteignent le point suivant. Par ailleurs, dans certains cas, le blocage d'un processus sur un point gelé peut conduire à des inter-blocages *temporaires* avec d'autres processus, par exemple en attente sur des communications pendantes (Fig. 4.22 (b)). Toutefois, si l'ordre *freeze* est relâché avant qu'un processus

[6]Nous supposons que la date courante d'un processus est incrémentée « à la sortie » du point d'instrumentation. Par conséquent, lorsque la simulation est bloquée à l'entrée du point par l'ordre *freeze*, la date courante reste fixée à c_i.

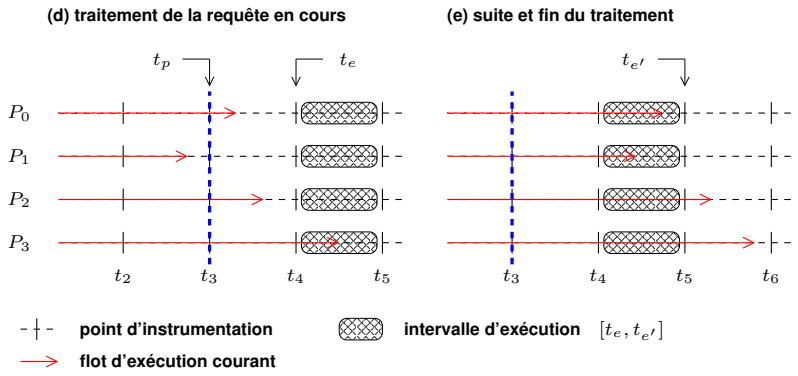


FIG. 4.23 – Traitement de la requête après la planification.

ne rencontre un point gelé, alors le calcul de t_{max} est recouvert, et la planification n'aura induit que peu ou pas de surcoût. Notons qu'il est encore possible d'optimiser le recouvrement de ce calcul en exploitant les résultats intermédiaires de la réduction ainsi calculée à partir d'un arbre binaire. Le principe consiste à relâcher partiellement l'ordre *freeze* pour les processus les plus en retard dans un sous-arbre de réduction, afin qu'ils rattrapent ceux les plus en avance. Si l'on se réfère à la figure 4.22 (b), cela permet par exemple au processus P_1 normalement bloqué à la fin de t_0 (i.e. avant t_1) de rejoindre le processus P_0 à la date t_1 sans attendre le résultat final de la réduction ($t_{max} = t_2$).

Il est facile d'étendre cet algorithme pour mettre en place une *synchronisation* des processus, à la manière d'une barrière MPI. En effet, pour certains types de requêtes, il peut être intéressant de garantir que tous les processus débuteront le traitement à la date t_p de manière synchronisée. Dans ce cas, l'ensemble de processus atteignant la date t_p s'attendent mutuellement avant de poursuivre l'exécution. Cette technique est naturellement utilisée pour les requêtes *step* et *stop*, mais peut également s'appliquer pour toutes requêtes de *steering*. Dans ce dernier cas, il faut conserver la synchronisation acquise en t_p jusqu'à l'entrée dans l'intervalle d'exécution en t_e. De manière générale, l'utilisation d'une synchronisation forte – même s'il induit un surcoût supplémentaire par rapport à la synchronisation faible – peut s'avérer pertinente pour les requêtes de *steering*, car il permet d'augmenter la robustesse des traitements effectués. Par exemple, si l'on considère le cas d'un *put*, un parcours non SPMD strict du MHT peut conduire à l'établissement d'une date t_p qui ne sera pas atteinte par un ou plusieurs processus de la simulation. Avec une synchronisation faible, la simulation peut se trouver dans un état critique où une partie seulement des processus aura effectué l'écriture des données souhaitées. Même si le client a toujours la possibilité d'annuler le traitement de la requête en cours, le traitement partiel d'une requête de *steering* conduira systématiquement à un échec fatal pour la simulation. La synchronisation forte, quant à elle, garantira qu'aucun traitement ne débutera avant que tous les processus n'aient atteint la date t_p. Ainsi, si certains processus n'atteignent jamais la date planifiée ou dépassent cette date (cas d'une désynchronisation partielle), il reste possible d'annuler la requête en toute sécurité voire de renégocier une nouvelle date de traitement pour la requête.

4.5 Conclusion

Dans ce chapitre, nous avons proposé un nouveau modèle pour le pilotage de simulations numériques parallèles, baptisé modèle hiérarchique en tâches (MHT). Ce modèle offre une vision structurée de la simulation s'opposant aux modèles classiquement utilisés dans les environnements de pilotage existants. Grâce à l'introduction d'un système de datation associées aux tâches et aux points du modèle, il est possible de se repérer précisément dans le flot d'exécution d'une simulation. L'introduction d'un ordre sur les dates (de points) et l'hypothèse d'un parcours SPMD strict du MHT nous ont permis de construire un algorithme de coordination efficace basé sur la planification d'une date de traitement. Cet algorithme permet de résoudre le problème de la cohérence temporelle en évitant toute synchronisation coûteuse. Un aspect original de notre modèle est la notion de plage d'interaction et plus particulièrement de plage d'accès aux données que nous associons aux

tâches décrites dans le MHT. Ainsi, en exhibant des régions du code au cours desquelles l'accès aux données est autorisé, il est possible de recouvrir en partie les communications sur ces tâches, tout en garantissant la cohérence des accès. Un autre aspect important dans notre modèle est le système de pilotage par les requêtes, entièrement dirigé par le client. Grâce à un jeu de requêtes réduit (*get*, *getp*, *put*, *action*), nous avons vu comment il était possible de composer des requêtes de pilotage relativement complexes.

Actuellement, notre modèle se limite à la représentation et au pilotage de codes SPMD. En pratique, il est possible de supporter des codes plus complexes à condition que l'instrumentation se limite à une vision SPMD de l'application. Par exemple, dans le cas d'une simulation maître/esclaves, il est possible d'ignorer le processus maître. Pour des simulations muti-codes (MPMD), on peut décider d'instrumenter chaque code séparément. Toutefois, ces solutions ne sont pas très satisfaisantes, car elles ne permettent pas de représenter globalement l'application couplée. Le « vrai » pilotage des applications du couplage de codes comme les simulations multi-physiques reste un problème largement ouvert, pour lequel notre modèle offre certaines perspectives. Ces codes soulèvent des difficultés tant pour leur représentation que pour la définition d'interactions de pilotage globalement cohérentes entre les codes. Nous verrons au chapitre final (chapitre 8) quelques éléments de réponse pour le pilotage de ces codes.

Chapitre 5

Modèle pour la redistribution d'objets complexes

Sommaire

Nous avons présenté au chapitre précédent un modèle de haut-niveau pour le pilotage de simulations numériques parallèles et leurs couplages avec des codes de visualisation parallèles. Toutefois, nous avons délibérément laissé en suspens deux problèmes : la représentation des données distribuées et le transfert de ces données entre les codes couplés. En effet, ces deux problèmes sont intimement liés au problème de la redistribution des données (ou problème M×N) que nous allons à présent étudié dans ce chapitre.

5.1 Introduction

Le problème de la redistribution des données a été beaucoup étudié dans le cadre de la programmation des architectures parallèles à mémoire distribuée, en particulier au titre des travaux menés autour de HPF [121] et de ScaLAPACK [50]. Sur ce type d'architecture, comme par exemple une grappe de PCs, le placement des données sur les processeurs est de première importance. En effet, c'est ce placement qui conditionne à la fois l'efficacité des algorithmes parallèles et la réduction du coût des communications. Le problème de la redistribution survient principalement dans deux contextes. Tout d'abord, il peut s'avérer utile au cours des étapes de calcul d'un programme parallèle de redistribuer les données pour optimiser la suite des calculs. Dans ce cas, la redistribution s'effectue « sur place », au sein du même code parallèle et chaque processeur envoie et reçoit des données. C'est par exemple le rôle de la directive *redistribute* dans le langage HPF. La redistribution des données est également primordiale dans le contexte du couplage de codes. Une application est alors vue comme un assemblage de plusieurs codes, le plus souvent parallèles, collaborant à la réalisation d'un travail commun (e.g. simulations multi-physiques). Ces codes sont typiquement distribués sur une grille de calcul, interconnectés par des réseaux plus ou moins performants. La communication entre les codes parallèles couplés nécessite de passer d'une distribution à une autre et donc de « redistribuer les données ».

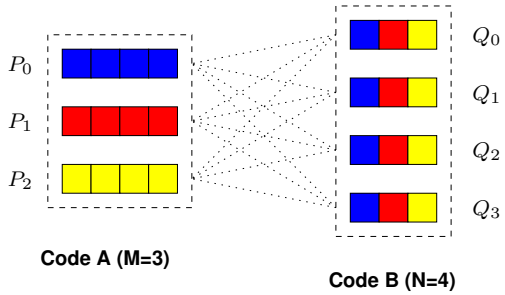


FIG. 5.1 – Le problème de la redistribution des données entre deux codes couplés.

On identifie généralement dans le problème de la redistribution des données quatre sous-problèmes à résoudre : la description des données distribuées, la génération des messages, l'ordonnancement des communications et la réalisation effective de ces communications.

Description des données distribuées. Le développement d'une solution standard au problème de la redistribution nécessite de définir un modèle de description également standard. Cette description doit spécifier d'une part la distribution des données entre les processeurs, et d'autre part l'agencement des données en mémoire sur chaque processeur.

Génération des messages. Le problème de la génération des messages consiste à déterminer, à partir des informations de distribution, les messages qui doivent être échangés entre chaque paire de processeurs. L'ensemble des messages ainsi calculés forme ce qu'on appelle la matrice de communication.

Ordonnancement. Une fois la matrice de communication calculée, il faut encore déterminer dans quel ordre effectuer les communications. En effet, les flux de communication parallèles, s'ils permettent d'agréger la bande-passante, peuvent entraîner des contentions réseaux qui dégradent les performances. Ainsi pour minimiser le temps global de communication, il convient de découper les messages afin de les ordonnancer en étapes de communication.

Communication. Lors de l'étape de communication, les codes couplés échangent l'ensemble des messages générés en respectant l'ordonnancement calculé. Le schéma de communication global s'apparente à un *all-to-all* personnalisé, transférant effectivement les données d'un code à l'autre.

Le pilotage des simulations numériques s'apparente dans notre étude à un problème de couplage entre deux codes parallèles, le code de simulation d'une part et le code de visualisation d'autre part. Afin de tirer profit des techniques de la visualisation parallèle et du rendu parallèle (Sec. 2.3.2), les données doivent être également distribuées côté visualisation, ce qui nous a naturellement conduit à nous intéresser au problème

de la redistribution. L'idée est de pouvoir visualiser rapidement (grâce à un cluster de PCs équipé de cartes graphiques 3D) de grands volumes de données avec une grande résolution (i.e. image de grande taille, mur d'images). Lors d'une session de pilotage, il est alors nécessaire de transférer les données de la simulation vers le client de visualisation et inversement. Cela se traduit dans notre modèle de pilotage par l'utilisation de requêtes *get* ou *put* commandant le transfert des données dans un sens ou l'autre. Du point de vue de la redistribution, ces deux opérations sont rigoureusement symétriques et il ne sera en général pas nécessaire dans ce chapitre de distinguer le code de simulation du code de visualisation. Nous préférons parler de deux codes parallèles A et B, respectivement distribués sur M et N processeurs (Fig. 5.1). Il existe toutefois une différence importante entre couplage et pilotage. Dans un problème de couplage traditionnel, chaque code possède initialement une version de la donnée avec sa propre description. Dans le contexte du pilotage, le programme de visualisation ne possède initialement pas d'information sur les données de la simulation à visualiser (*description vierge*). Par conséquent, le client de visualisation doit pouvoir interroger dynamiquement la simulation pour connaître sa description et construire une description de données compatible. A l'issue de cette étape, nous pouvons nous ramener à un problème de couplage traditionnel, dans le sens où chaque code va posséder sa propre description des données.

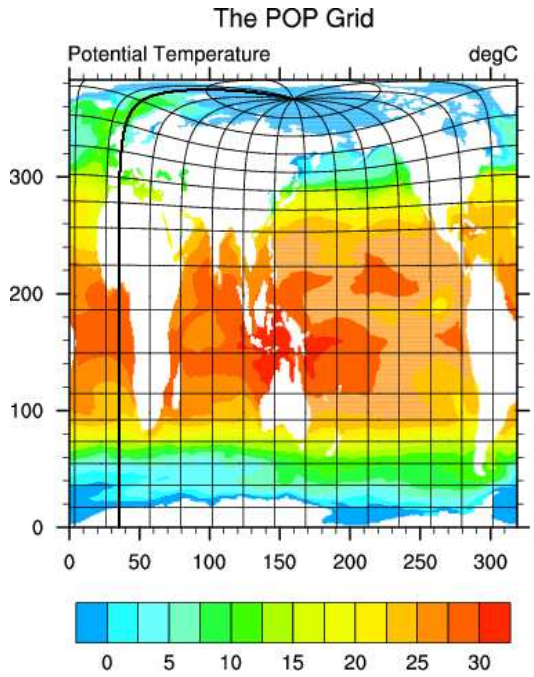


FIG. 5.2 – Exemple de distribution « creuse » dans POP [124]. Les continents (en blanc) ne sont pas pris en compte dans la distribution en bloc.

Le couplage simulation/visualisation peut conduire à des redistributions de données plus complexes que celles étudiées par le passé. En effet, la plupart des résultats existants dans ce domaine s'adressent principalement à des calculs d'algèbre linéaire pour des matrices denses. Dans ce cas, les données sont distribuées de manière bloc-cyclique pour optimiser les calculs numériques, tandis que dans des codes de simulation et/ou de visualisation parallèles plus complexes, les distributions sont souvent beaucoup plus irrégulières. Par ailleurs, il faut prendre en compte le fort déséquilibre qu'il existe entre le nombre de processeurs affectés à la simulation et de ceux utilisés pour la visualisation : de l'ordre de plusieurs centaines pour la simulation à quelques dizaines côté visualisation. La motivation première de nos travaux sur la redistribution est donc d'étendre et de généraliser les travaux existants, afin de pouvoir supporter des distributions de données plus irrégulières : décomposition spatiale, structure creuse, cellules fantômes (*ghost cells*), *overlap* entre les blocs, sous-domaine de visualisation, *etc*. Notre approche s'appuie sur la définition d'un modèle de description des données en termes *d'objets*, tels qu'ils sont véritablement présents dans les codes de simulation : grilles structurées, ensemble de particules, maillages non structurés, *etc*. Ces objets ne sont pas simplement vus comme des tableaux de données en mémoire ; ils encapsulent un ensemble d'informations utiles à leur redistribution et à leur visualisation.

Afin d'illustrer notre propos, reprenons l'exemple de la simulation POP (Parallel Ocean Program) [124] que nous avons présenté au chapitre 1. La discrétisation utilise une grille de calcul de $3600 \times 2400 \times 40$ cellules pour représenter les océans du fond à la surface (Fig. 5.2). La grille est découpée en blocs de taille $16 \times 16 \times 40$, qui sont placés sur un ensemble de processeurs pour équilibrer la charge globale des calculs. La charge affectée à un bloc dépend directement de la surface des océans occupant ce bloc – les continents n'étant pas pris en compte dans le modèle océan. De plus, il faut remarquer que les blocs entièrement recouverts de terre sont éliminés de la distribution, ce qui nous conduit à « une structure creuse ». Ces distributions de données sont également très fréquentes en dynamique moléculaire, où elles sont plus connues sous le nom de *décomposition spatiale* [176]. Dans ces codes, comme par exemple NAMD [199], l'espace physique est découpé en « boîtes d'atomes » placées sur les processeurs pour équilibrer la charge et minimiser le volume de communication. Il s'avère donc important d'étendre les études précédentes sur la redistribution, afin de pouvoir considérer des distributions de données plus générales.

Nous nous intéressons dans ce chapitre au problème général de la redistribution d'objets complexes et plus précisément au calcul de la matrice de communication. Nous n'abordons pas ici le problème de l'ordonnancement des communications, pour lequel nous souhaitons reposer sur des résultats existants [114]. Après un rapide tour d'horizon concernant les travaux antérieurs sur la redistribution, nous proposons dans la section 5.3 une formulation ensembliste du problème de la redistribution, très générale, servant de fil directeur à cette étude. Puis, nous introduisons un modèle de description des données basé sur la notion d'objet complexe (section 5.4). Finalement, nous proposons aux sections 5.6.2 et 5.6.3 deux approches de la redistribution : *l'approche spatiale* et *l'approche placement*. Nous verrons dans la troisième partie de cette thèse une validation des travaux présentés dans ce chapitre et réalisée avec les bibliothèques RedSYM et RedCORBA.

5.2 Travaux existants

Le problème de la génération des messages a principalement été étudié dans le cas des distributions bloc-cycliques de tableaux denses [178, 198, 212]. Ces distributions sont fréquemment utilisées en algèbre linéaire pour des calculs sur les matrices denses car elles possèdent de bonnes propriétés d'équilibrage de charge. Dans ce contexte, il s'agit de passer d'une distribution bloc-cyclique à une autre distribution bloc-cyclique, ce qui conduit à un schéma de communication régulier. On trouve également beaucoup de résultats sur l'ordonnancement des communications pour ce type de distributions [71, 70]. Dans ces travaux, il s'agit essentiellement de générer les messages et de les ordonnancer le plus efficacement possible en exploitant la régularité du schéma de communication. Plus récemment, Jeannot *et al.* [114] ont proposé un algorithme d'ordonnancement général s'appliquant à tout type de distributions en se basant uniquement sur la connaissance de la matrice de communication. De rares travaux abordent le problème de la redistribution dans un contexte aussi général. Nous allons maintenant passer rapidement en revue les travaux existants sur la redistribution. Dans le reste de la section, nous nous intéresserons essentiellement aux modèles de description des données et à la génération des messages. Dans ce domaine, on distingue principalement les travaux portant uniquement sur la *redistribution de tableaux* et ceux indépendants de la structure de données basés sur le *principe de linéarisation*.

5.2.1 La redistribution des tableaux pour le couplage de codes

Conceptuellement, la représentation des tableaux distribués est divisée en deux étapes : la description de la distribution (*layout*), où comment les éléments du tableau sont réparties entre les processeurs, et la description du stockage des données (*data storage*) précisant l'agencement des éléments dans la mémoire de chaque processeur (adresse en mémoire, type de données, alignement, *etc*.). Un point clé motivant cette séparation est que seul la connaissance des distributions est utile pour calculer le schéma de communication entre les processeurs couplés. En effet, comme le suggère la figure 5.3, le calcul de ces communications est indépendant de la connaissance précise du stockage des données en mémoire, même si cette information est au final pour générer les messages physiques. De plus il est possible, grâce à cette séparation formelle, de factoriser le calcul de la matrice de communication pour des tableaux partageant la même distribution. Nous allons maintenant présenter quelques travaux sur la redistribution des tableaux s'inscrivant dans le contexte du couplage de

codes et du pilotage.

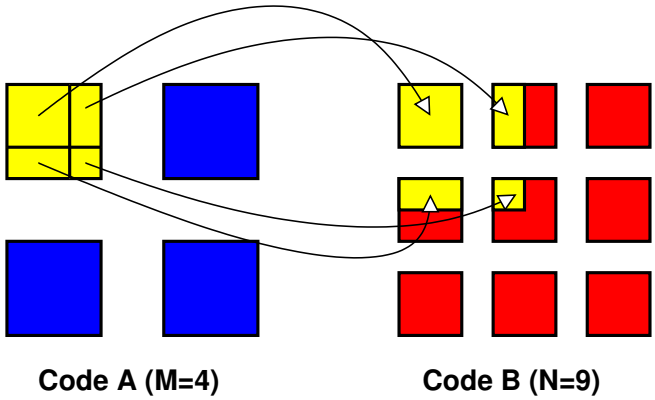


FIG. 5.3 – Le schéma de communication dépend uniquement de la distribution des éléments du tableau et non du stockage.

Dans CUMULVS [122] (cf. chapitre 3), une simulation numérique parallèle peut être couplée à un ou plusieurs programmes séquentiels de visualisation simultanément (plusieurs couplages M×1). Les données considérées côté simulation sont uniquement des tableaux denses utilisant des distributions de données en bloc-cyclique inspirées de HPF. Côté visualisation, chaque client a la possibilité de restreindre sa vue à un sous-domaine d'intérêt, ce qui nous rapproche d'un problème de redistribution classique. CUMULVS supporte uniquement des données contiguës en mémoire, avec éventuellement la présence de cellules fantômes (*ghost cells*) au marge du tableau. Par ailleurs, CUMULVS prend en charge des conversions de types (entiers, flottants) et différentes conventions de stockages (*row/column major*). Notons que CUMULVS propose également un modèle pour la représentation d'ensembles de particules à coordonnées entières. La coordonnée renvoie à la position de la particule dans une distribution de tableau classique. A chaque particule, il est possible d'associer un tableau de valeurs. CUMULVS valide son modèle avec une simulation de pilotage d'un hélicoptère sur une grille 2D, l'hélicoptère se déplaçant de case en case.

Dans PAWS [44, 118] (cf. chapitre 3), il est possible de redistribuer des tableaux denses multidimensionnels entre deux codes couplés, distribués sur des nombres de processeurs différents (couplage M×N). Plus précisément, PAWS supporte des distributions de tableaux très générales, correspondant à une décomposition rectilinéaire d'un domaine global parallélépipédique. En pratique, l'utilisateur déclare explicitement sur chaque processeur un ensemble de blocs appelé sous-domaine. Ensuite, il associe à chaque bloc un emplacement mémoire contiguë avec éventuellement des *strides* entre les éléments et des *ghost cells* au marge du tableau. Notons que l'ensemble des sous-domaines doit recouvrir la totalité du domaine global et ne pas se chevaucher. La matrice de communication est calculée de manière centralisée sur un processus « extérieur » appelé contrôleur. Ce calcul dépend uniquement du *layout* (domaine globale et sous-domaine), qui peut être réutilisé dynamiquement pour différents tableaux. Par ailleurs, PAWS facilite le couplage de codes en prenant en charge certaines difficultés comme des conversions de types, des transpositions (*row/column major*) ou des inversions de dimensions.

Plus récemment, le groupe CCA M×N [5, 48] a proposé une spécification pour la redistribution de données entre des composants logiciels CCA, en s'inspirant largement de la spécification de HPF et en intégrant plusieurs technologies existantes dont celles utilisées dans PAWS et CUMULVS. Une différence importante entre CCA M×N et les travaux précédents tient au fait que CCA M×N ne s'occupe que de la définition d'une interface abstraite et non de son implantation. Notons qu'à ce jour, il existe plusieurs prototypes décrits dans [48], dont aucun n'implante complètement la spécification. Actuellement, CCA M×N se limite au cas de tableaux denses, multidimensionnels et parallélépipédiques. Nous avons résumé sur la figure 5.4 les différents types de distributions qui sont spécifiés. Ces distributions résultent pour la plupart d'une décomposition d'un domaine global, axe par axe, selon différents types. En plus du modèle bloc-cyclique classique regroupant les types *col-*

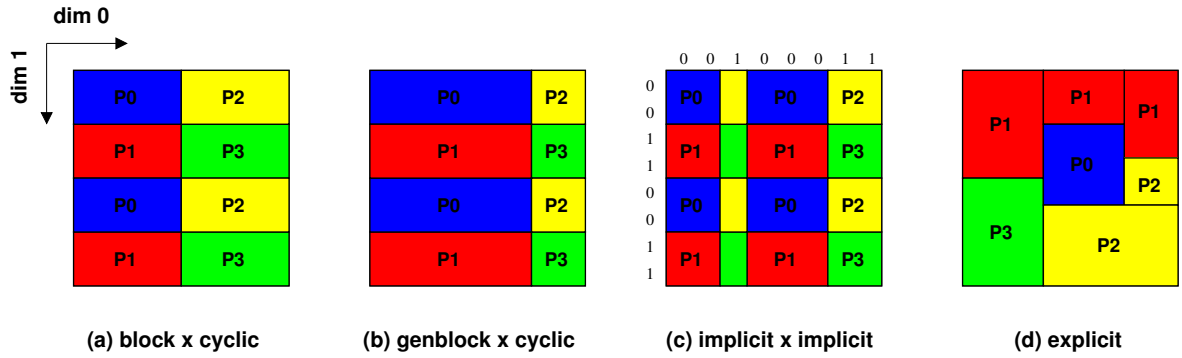


FIG. 5.4 – Les différents types de distribution dans CCA M×N.

lapsed, *block* et *cyclic* (Fig. 5.4 (a)), la proposition introduit deux nouveaux types de distribution : le type *genblock* (bloc généralisé) et le type *implicit*. Le type *genblock* est une variante du type *block* où chaque processeur possède un seul bloc et où les blocs peuvent être de tailles différentes (Fig. 5.4 (b)). Le type *implicit* permet de spécifier entièrement la distribution des éléments, un-par-un, selon chaque axe (Fig. 5.4 (c)). Outre ces distributions, on trouve encore des distributions de type *explicit*, identiques aux distributions rectilinéaires utilisées dans PAWS (Fig. 5.4 (d)). La spécification pour la représentation des tableaux distribués dans CCA se compose essentiellement de deux interfaces. La première interface *DistArrayTemplate* permet de déclarer un *template* de distribution des éléments (vocabulaire emprunté à HPF). La seconde interface *DistArrayDescriptor* permet d'aligner un tableaux réel sur le *template* en prenant en compte les informations liées au stockage (type de données, adresse mémoire).

5.2.2 Le principe de linéarisation

Le principe de linéarisation consiste à ordonner totalement les éléments d'un objet « source » et d'un objet « destination », ce qui permet de mettre implicitement et indirectement en correspondance tous les éléments. Comme le suggère la figure 5.5, les objets considérés sont constitués de plusieurs régions, qui sont linéarisés de manière abstraite dans une structure intermédiaire 1D (i.e. un vecteur). Le point fort du concept de *linéarisation* tient au fait qu'il est indépendant de la nature des objets auxquels il s'applique. Ainsi, il est possible de linéariser et de redistribuer des structures de données *a priori* quelconques « en aplatissant » ces structures (e.g. tableaux, arbres, graphes, maillages, *etc.*). Cependant, il faut remarquer qu'il n'y a généralement pas une manière unique de linéariser un objet. Par exemple, dans le cas d'un arbre, on peut considérer le parcours en profondeur à gauche ou à droite. Par conséquent, les deux applications couplées doivent utiliser le même schéma de linéarisation pour que la redistribution ait un sens. Cela implique typiquement aux programmes couplés d'avoir une connaissance *a priori* l'un de l'autre ou d'échanger des informations supplémentaires.

Le principe de linéarisation a été initialement introduit dans la bibliothèque Meta-Chaos [181, 75] développée à l'université du Maryland. Ce logiciel est principalement dédié au couplage de codes et vise à faire communiquer des bibliothèques et des langages parallèles (HPF, pC++, Multiblock Parti, Chaos, *etc.*) utilisant des structures de données et des distributions différentes. Suite à l'optimisation des algorithmes de redistribution, ce logiciel a récemment évolué vers une nouvelle bibliothèque baptisée InterComm [129]. Dans InterComm, plusieurs méthodes sont fournies pour décrire les éléments au sein d'une région : une première méthode dite *compacte* basée sur des régions agencées sous la forme de tableaux (les éléments sont implicitement ordonnés), et une seconde méthode dite *non-compacte* basée sur une énumération explicite des éléments dans la région. Dans ce dernier cas, la taille de la structure intermédiaire est du même ordre de grandeur que celle des données considérées. Notons que dans Meta-Chaos, seule la méthode non-compacte est disponible. InterComm propose alors divers algorithmes de calcul de la matrice de communication, en parallèle ou de manière centralisée, prenant en compte la compacité des méthodes de linéarisation utilisées. L'idée pour optimiser ce calcul est d'éviter de transférer des descriptions non-compactes, alors qu'il est peu coûteux de transférer voire de répliquer sur les différents processus des descriptions compactes. Cette bibliothèque illustre ces possibilités avec un exemple de redistribution entre une grille structurée 256×256 (distribuée régulièrement

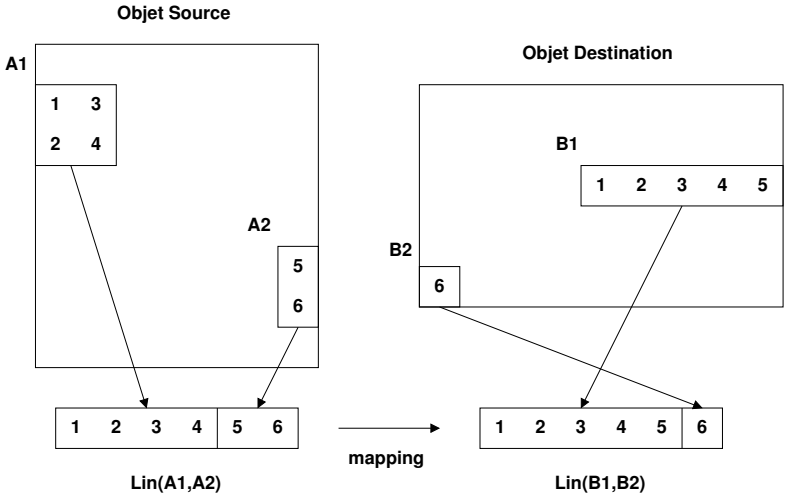


FIG. 5.5 – Principe de linéarisation : les éléments sont mis en correspondance un-à-un via une structure intermédiaire 1D.

avec Multiblock Parti) et un maillage non structuré de 65536 nœuds (distribuée irrégulièrement avec Chaos).

Le principe de linéarisation est également employé dans d'autres travaux autour de la redistribution. Parmi ces travaux, citons en particulier MPI-IO M×N [49] développé à l'université de l'Indiana. Dans ce système, chaque processus du côté « receveur » diffusent à tous les processus côté « émetteur » la liste explicite des éléments dont il a besoin. Cette liste d'éléments fait référence à la linéarisation implicite qui existe lorsqu'un programme parallèle sérialise ces données dans un fichier (Fig. 5.6). Puis, selon le principe des entrées/sorties parallèles de MPI (ou MPI I/O), chaque émetteur écrit dans un *device* virtuel (se substituant au fichier) les données qui sont lues à travers ce même *device* par le receveur. L'un des aspects intéressants de ce travail est que son utilisation est possible de manière transparente avec MPI, soit pour transférer les données via le réseau vers un code distant, soit pour écrire les données dans un fichier.

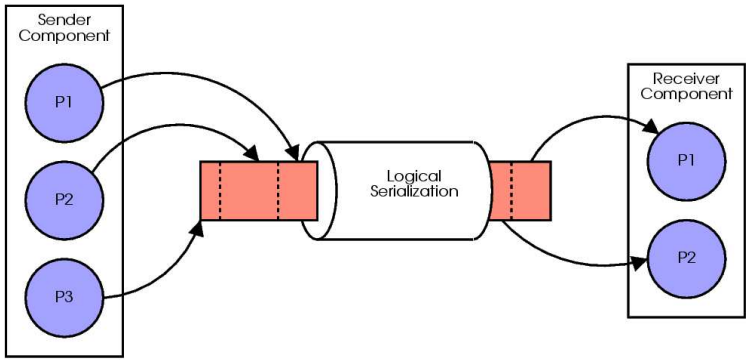


FIG. 5.6 – MPI-IO M×N : les données sont logiquement ordonnées mais transmises en parallèle [49].

5.2.3 Autres travaux reliés

Il existe encore d'autres travaux reliés sur la redistribution qui abordent des problématiques plus spécifiques aux simulations multi-physiques, incluant des problèmes de conversion d'unités, d'interpolation, *etc.*

Parmi ces travaux, il faut citer en particulier MpCCI (Mesh based Parallel Code Coupling Interface) [19] qui permet de coupler des applications parallèles à base de maillages. Le point fort de MpCCI réside dans le fait qu'il utilise des schémas d'interpolation (conservatifs ou non) permettant de transférer les données

même lorsque les maillages ne sont pas identiques. En se basant sur la géométrie des maillages, MpCCI met en relation les nœuds et/ou les éléments des deux maillages. La région de couplage calculée est typiquement une surface 3D comme le montre l'exemple du couplage fluide-structure (Fig. 5.7). Dans cet exemple, un code de mécanique des fluides utilisant une grille structurée 3D est couplé avec un code de structures utilisant un maillage non structuré d'éléments finis. Dans le premier code, la force est calculée au centre des éléments alors que dans le second code, elle est utilisé sur les nœuds.

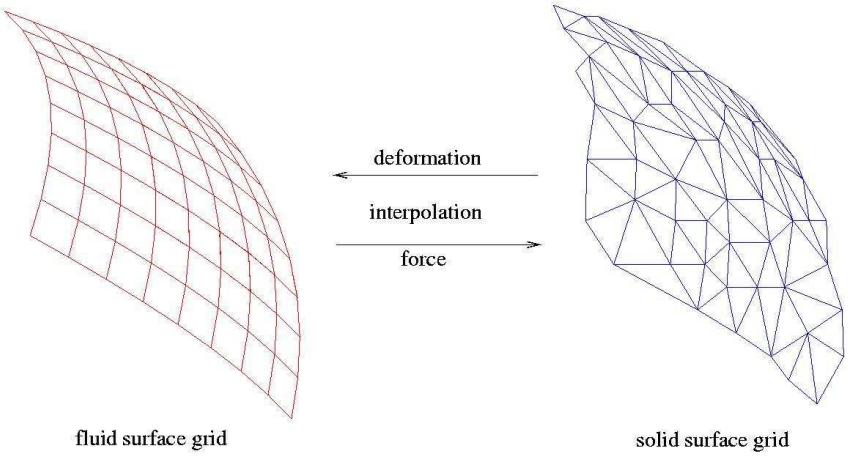


FIG. 5.7 – Exemple d'un couplage fluide-structure avec MpCCI [19].

Un autre logiciel MCT (Model Coupling Toolkit) [16] est utilisé dans le projet CCSM (Community Climate System Model) [6] pour réaliser le couplage entre différents modèles (terre, océan, atmosphère, *etc*.). MCT permet essentiellement de redistribuer des grilles structurées en prenant en compte leur représentation physique (temporelle et spatiale). Dans MCT, la distribution des éléments de la grille est définie dans la structure *GlobalSegmentMap*, qui énumère explicitement les ensembles d'éléments contigus (ou *segments*). Grâce à cette représentation, MCT offre une très grande liberté de distribution des éléments, qui peut être *a priori* quelconque. Les codes associés aux modèles sont couplés entre eux par l'intermédiaire d'un coupleur parallèle (cf. Sec 2.2.1). Ainsi, le problème de la redistribution ne se pose pas directement entre les modèles mais entre chaque modèle et le coupleur. Lors des communications inter-modèles, les données transitent donc via le coupleur, ce qui laisse l'opportunité de réaliser des traitements supplémentaires comme des interpolations (en parallèle) pour adapter la résolution des grilles couplées. Un des inconvénients de MCT tient au fait qu'il impose de travailler avec une structure de données spécifique à ce programme (*AttributeVector*), ce qui oblige un code de simulation à utiliser cette structure ou à effectuer une conversion de structure coûteuse lors des communications.

5.2.4 Synthèse

En conclusion, nous retiendrons que le problème de la redistribution des données a principalement été étudiée dans le cas de tableaux ou de grilles structurées avec des distributions relativement régulières (Tab. 5.1). La plupart des travaux que nous venons de présenter cherchent à décrire les structures de données telles qu'elles sont présentes dans les codes de simulation grâce à un modèle de représentation, plutôt que d'imposer comme dans MCT l'utilisation d'une structure de données interne, entraînant des conversions de données coûteuses (i.e. des recopies). A ce niveau, on constate qu'il y a un compromis à réaliser entre la généricité du modèle de description et la compacité de ces descriptions, ce qui joue également sur l'efficacité des algorithmes de redistribution. Une limitation importante à tous ces travaux, mis à part CCA M×N, est liée au fait qu'ils ne dissocient pas la couche algorithmique responsable du calcul de la matrice de communication, de la couche de communication responsable du transfert des messages, ce qui réduit la réutilisabilité de ces outils. Seul CCA M×N [5] a choisi de clairement séparer à l'aide d'interfaces distinctes, le problème de description des données, de génération des messages et de transfert de ces messages. Du point de vue de la généricité, tous les outils ne sont pas comparables. PAWS et CCA M×N supportent des distributions de tableaux relativement

générales correspondant à une décomposition en blocs d'un domaine parallélépipédique. Toutefois, il n'est pas possible avec ces logiciels de prendre en compte des structures creuses comme celles que nous avons présentées en introduction avec POP, car l'ensemble des blocs doit impérativement recouvrir la totalité du domaine et ne pas se recouper. Les outils basés sur le principe de linéarisation apparaissent plus puissants, car ils peuvent s'appliquer de manière très générale à des structures de données régulières ou irrégulières. Cependant, cette approche a un coût si l'on considère la taille des représentations intermédiaires utilisées pour décrire la distribution des éléments (au pire, du même ordre de grandeur que les données elles-mêmes). De plus, l'absence d'un modèle de description unifié ne permet pas d'assurer l'interopérabilité requise entre les codes couplés. En effet, comme nous l'avons déjà souligné, si deux codes utilisent des linéarisations différentes, le résultat de la redistribution n'aura *a priori* aucun sens, ce qui est une limitation grave de cette approche. Cela est d'autant plus vrai que c'est à l'utilisateur qu'incombe la responsabilité de définir le schéma de linéarisation. Nous soulignons que ce problème est essentiellement lié au fait que Meta-Chaos comme InterComm ne proposent pas de modèle de représentation des objets, ni de schémas de linéarisation standards pour ces objets. Les structures de données sont simplement perçues comme des plages d'éléments en mémoire, sans relation avec la nature réelle de ces objets (ensemble de particules, grille structurée, maillage non structuré, *etc.*).

Bibliothèques	Couplage	Objets	Distributions
HPF, ScaLAPACK	M×M	tableaux	bloc-cyclique
CUMULVS	M×1	tableaux	bloc-cyclique
CUMULVS	M×1	particules dans $\mathbb{Z}^n$	bloc-cyclique
PAWS	M×N	tableaux	explicite
CCA M×N	M×N	tableaux	bloc-cyclique + explicite
Meta-Chaos/InterComm	M×N	quelconques	quelconques
MPI I/O M×N	M×N	quelconques	quelconques
MpCCI	M×N	grilles/maillages	quelconques
MCT	M×N	grilles	quelconques

TAB. 5.1 – Synthèse des techniques de redistribution existantes.

Pour conclure, nous pensons qu'il est indispensable de définir un modèle de description « orienté objet », capable d'encapsuler toutes les informations décrivant ces objets et leurs distributions. Ainsi, en exploitant certaines propriétés relatives à ces objets, nous verrons dans ce chapitre qu'il est possible – sous certaines hypothèses – de redistribuer efficacement des objets irréguliers (e.g. grilles creuses, ensembles de particules, maillages non structurés, *etc.*), sans qu'il soit nécessaire d'effectuer une linéarisation coûteuse. En particulier nous verrons comment, dans le contexte du pilotage, il est possible de relaxer certaines contraintes sur la distribution des éléments visualisés pour faciliter le calcul de la redistribution (approche placement). Dans un autre contexte, un modèle objet permet également de prendre en compte des informations supplémentaires permettant d'effectuer des traitements ou des post-traitements de haut-niveau comme des filtrages, des interpolations, des entrées/sorties ou encore de la visualisation. Un dernier point sur lequel il nous paraît important d'insister est la séparation nécessaire qui doit être faite entre la couche de redistribution et la couche de communication. Une telle séparation laisse supposer l'introduction d'une représentation abstraite des messages, qui est indépendante de la couche de communication utilisée, ce que par la suite nous nommerons les *messages symboliques*.

5.3 Formulation ensembliste du problème de la redistribution

5.3.1 Définitions préliminaires

Soit P un ensemble de processeurs de taille M. Soit E un ensemble d'éléments totalement ordonnés. On associe à chaque élément de E un indice global $k \in \mathbb{N}$ et on note $E[k]$ le k-ème élément de cet ensemble.

Définition 5.3.1 (Décomposition) *Une décomposition de E sur P est une séquence[1] de M sous-ensembles de E, noté $E/P = (A_0, A_1, \ldots, A_{M-1})$, et tel que les éléments du sous-ensemble A_i soient associés au processeur P_i.*

[1] Dans la suite de ce chapitre, nous utiliserons la terminologie « séquence » ou « liste » pour désigner des ensembles ordonnés.

Dans la littérature, les décompositions considérées forment le plus souvent une partition de E, ce qui n'est pas suffisamment général dans notre contexte pour décrire les décompositions complexes rencontrées dans les codes de simulation et de visualisation. Dans notre étude, les ensembles A_i pourront être tout à fait quelconques, se recouper ou ne pas recouvrir l'ensemble E dans sa totalité.

Définition 5.3.2 (Distribution) *Une distribution de E sur P est une fonction α_P qui à l'indice global k d'un élément de E associe un processeur de rang i dans P et un indice local l dans A_i. On dit alors que E est distribué sur P selon α_P et on note $A_i[l] = E[k]$ l'élément d'indice local l placé sur le processeur P_i. La distribution de E sur P selon α_P induit une décomposition des éléments $\{A_0, A_1, \ldots, A_{M-1}\}$ telle que $\alpha_P(k) = (i, l)$ et $E[k] = A_i[l]$.*

Dans le cas particulier d'une distribution bloc-cyclique 1D, la fonction de distribution s'écrit : $k \longrightarrow (\lfloor k/s \rfloor \,\%\, M,\ b.s + k \,\%\, s)$ avec s la taille du bloc, $b = \frac{\lfloor k/s \rfloor}{M}$ l'indice local du bloc relatif à k et M le nombre total de processeurs [71].

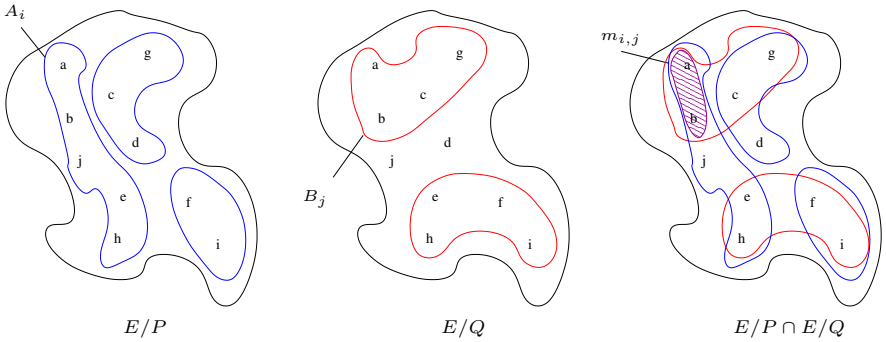


FIG. 5.8 – Décomposition de l'ensemble d'éléments E sur P et Q. Calcul du message $m_{i,j}$ par intersection des ensembles A_i et B_j.

5.3.2 Génération des messages selon le principe d'intersection

Soient E un ensemble d'éléments, P et Q deux ensembles de processeurs respectivement de taille M et N. Considérons la distribution α_P de E sur P et α_Q de E sur Q, avec $E/P = (A_0, A_1, \ldots, A_{M-1})$ et $E/Q = (B_0, B_1, \ldots, B_{N-1})$. Dans cette étude, nous considérons uniquement le cas où les distributions α_P et α_Q sont définies sur le même ensemble d'éléments E[2]. Ce prédicat essentiel permet de garantir que le problème de la redistribution à une solution unique, puisqu'il est alors possible de mettre en correspondance les éléments un à un entre les sous-ensembles A_i et B_j. Pour résoudre le problème de la redistribution, il faut calculer l'ensemble des messages échangés entre P et Q.

Définition 5.3.3 (Message) *Le message $m_{i,j}$ est l'ensemble des éléments qu'il faut échanger entre P_i et Q_j.*

Définition 5.3.4 (Matrice de communication) *L'ensemble des messages échangés entre chaque paire de processeurs (P_i, Q_j) forme la matrice de communication de P vers Q, notée $\mathbb{M} = [m_{i,j}]$ de taille $M \times N$.*

Une ligne i de la matrice $\mathbb{M}$ désigne donc l'ensemble des messages que le processeur P_i doit envoyer vers Q. A l'inverse, une colonne j de la matrice correspond à l'ensemble des messages qui seront reçus par le processeur Q_j.

Définition 5.3.5 (Principe d'intersection) *Le message $m_{i,j}$ échangé entre P_i et Q_j résulte de l'intersection des ensembles d'éléments A_i et B_j (i.e. $m_{i,j} = A_i \cap B_j$).*

Ce principe est illustré par la figure 5.8. Si E/P et E/Q sont des partitions de E, alors l'ensemble des messages $\{m_{i,j} \,|\, 0 \leq i < M, 0 \leq j < N\}$ définit une nouvelle partition de E, plus fine que les partitions E/P et E/Q. Comme l'illustre la figure 5.9, un élément $e = E[k]$ appartient au message $m_{i,j}$, si et seulement si il existe

[2]Dans le cas où α_P serait défini sur un l'ensemble E_P et α_Q sur un ensemble E_Q différent, il serait possible de se ramener au cas précédent en posant $E = E_P \cap E_Q$ et en considérant la restriction de α_P et de α_Q sur E.

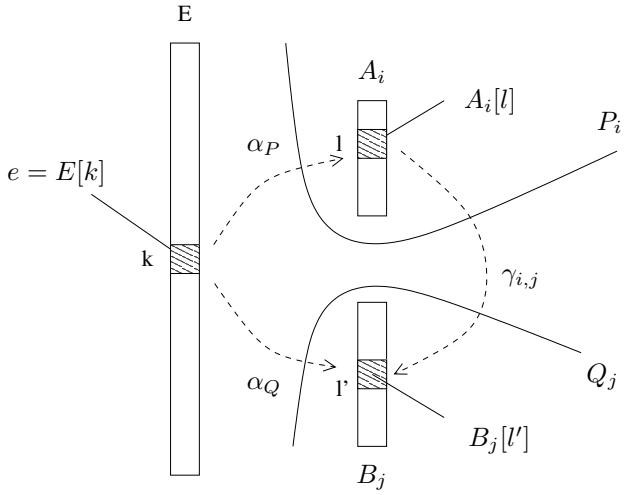


FIG. 5.9 – Redistribution de P vers Q selon γ.

deux indices l et l' tels que $\alpha_P(k) = (i, l)$ et $\alpha_Q(k) = (j, l')$, c'est-à-dire tels que $e = E[k] = A_i[l] = B_j[l']$. On note $\gamma_{i,j}$ la fonction associant l'indice local l d'un élément dans A_i à son indice local l' dans B_j. Par extension, le schéma de redistribution est complètement défini par la fonction $\gamma : (i, l) \longrightarrow (j, l')$ vérifiant $\gamma_{i,j}(l) = l'$. Comme le suggère la figure 5.10, il est possible d'associer à un élément e échangé entre P_i et Q_j un indice z dans le message $m_{i,j}$ ($0 \leq z < |m_{i,j}|$). La fonction $\gamma_{i,j} : l \longrightarrow l'$ se décompose alors en deux fonctions, $\gamma^e_{i,j} : l \longrightarrow z$ côté envoi et $\gamma^r_{i,j} : z \longrightarrow l'$ côté réception, telles que $\gamma_{i,j} = \gamma^r_{i,j} \circ \gamma^e_{i,j}$. La définition de ces fonctions permet de résoudre entièrement le problème de la redistribution, car elles définissent un ordre sur les éléments du message, qui doit être identique entre l'émetteur et le receveur pour que l'étape de communication ait un sens : $z = \gamma^e_{i,j}(l) = {\gamma^r_{i,j}}^{-1}(l')$.

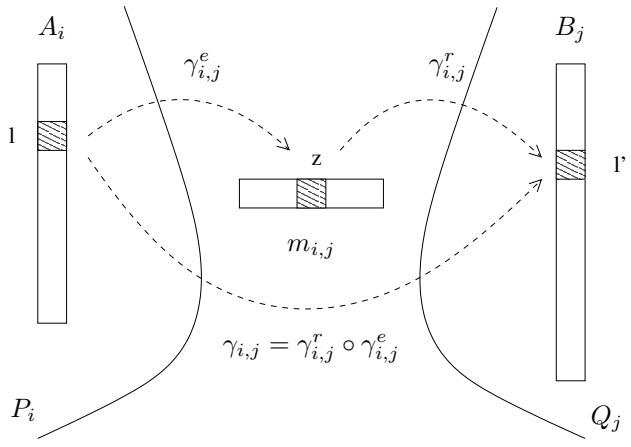


FIG. 5.10 – Envoi et réception d'un message.

Afin d'illustrer notre propos, nous allons maintenant examiner plus en détails l'exemple de la figure 5.11. Dans cet exemple, une tableau 2D (10×6) est distribué sur P ($M = 3$) et Q ($N = 2$) plus ou moins régulièrement selon les décompositions $E/P = (A_0, A_1, A_2)$ et $E/Q = (B_0, B_1)$ représentées sur la figure. Les éléments du tableau sont totalement ordonnés de 0 à 59. En appliquant le principe d'intersection, on trouve la matrice de communication suivante.

$$\mathbb{M} = \begin{bmatrix} m_{0,0} & m_{0,1} \\ m_{1,0} & m_{1,1} \\ m_{2,0} & m_{2,1} \end{bmatrix} \quad \text{avec} \quad \begin{aligned} m_{0,0} &= A_0 \cap B_0 = \{0, \ldots, 4, 10, \ldots, 14, 20, \ldots, 24, 30, \ldots, 34\} \\ m_{0,1} &= A_0 \cap B_1 = \{5, \ldots, 9, 15, \ldots, 19\} \\ m_{1,0} &= A_1 \cap B_0 = \{40, \ldots, 44, 50, \ldots, 54\} \\ m_{1,1} &= A_1 \cap B_1 = \{45, 46, 55, 56\} \\ m_{2,0} &= A_2 \cap B_0 = \emptyset \\ m_{2,1} &= A_2 \cap B_1 = \{25, \ldots, 29, 35, \ldots, 39, 47, 48, 49, 57, 58, 59\} \end{aligned}$$

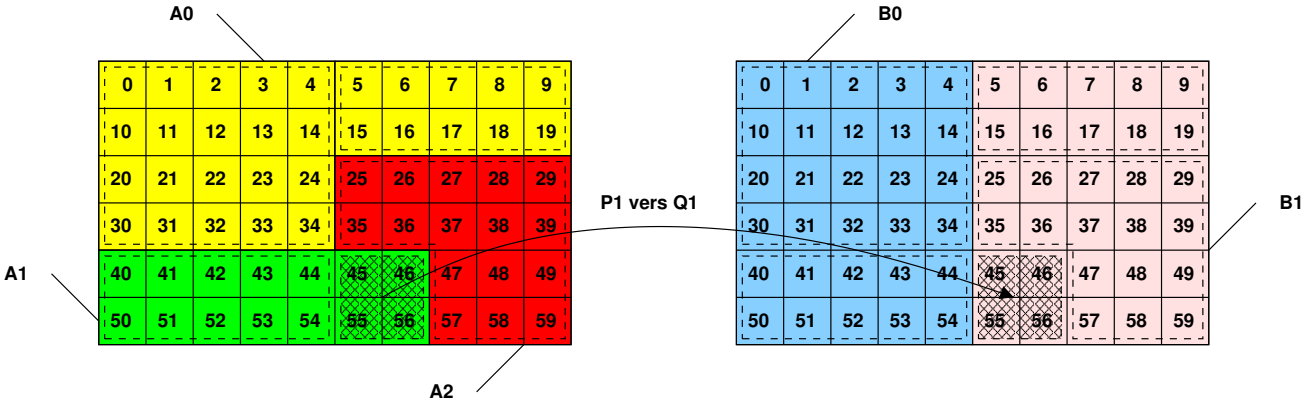


FIG. 5.11 – Exemple de redistribution des éléments d'un tableau 2D entre P et Q. Le message $m_{1,1}$ de P_1 vers Q_1 résulte de l'intersection de A_1 et B_1 (cf. zone hachurée).

Le problème de la redistribution des données, tel que nous l'avons formulé, est symétrique en P et Q. Ainsi, on peut définir de la même manière la matrice de communication de Q vers P, de taille $N \times M$, que l'on note $\mathbb{M}' = [m'_{j,i}]$. D'après le principe d'intersection, la matrice de communication $\mathbb{M}'$ est simplement la transposée de $\mathbb{M}$, ce que traduit l'égalité : $m_{i,j} = m'_{j,i} = A_i \cap B_j$. En d'autres termes, le message que P_i envoie à Q_j est identique à celui que P_i s'attendrait à recevoir de la part de Q_j. Cependant, il peut s'avérer utile dans certains cas de distinguer les ensembles d'éléments servant à *l'envoi* et à *la réception*, ce que l'on peut noter respectivement A^e_i et A^r_i. Pour résoudre le problème de la redistribution dans ce contexte, il est alors nécessaire de calculer deux matrices de communications : une pour l'envoi notée $\mathbb{M}^e = [m^e_{i,j}]$ et une pour la réception notée $\mathbb{M}^r = [m^r_{i,j}]$. Le principe d'intersection s'écrit alors dans sa forme généralisée : $m^e_{i,j} = A^e_i \cap B^r_j$ et $m^r_{i,j} = A^r_i \cap B^e_j$. De plus, il est facile de vérifier que $\mathbb{M}'^e$, la matrice d'envoi de Q vers P, est la transposée de $\mathbb{M}^r$, c'est-à-dire que $m^e_{i,j} = m'^r_{j,i}$ et $m^r_{i,j} = m'^e_{j,i}$.

5.4 Les objets complexes

5.4.1 Définitions

Le modèle de description des données que nous proposons est basé sur la notion *d'objet complexe*. Ce modèle a pour objectif de prendre en compte une grande variété de structures de données présentes dans les codes de calcul scientifique, tout en permettant une génération efficace des messages pour la redistribution. Considérons un ensemble d'éléments E totalement ordonné. Un objet complexe se présente comme une partie de E découpée en sous-ensembles appelées *régions* et composée de plusieurs *séries de données*, tels que chaque élément de l'objet possède une valeur dans chaque série. Nous préciserons dans la section suivante le modèle que nous utilisons pour décrire le stockage des données en mémoire.

Définition 5.4.1 (Objet complexe) *Un objet complexe est un triplet $\mathcal{O} = (E, R, S)$ avec :*

- *E, un ensemble d'éléments totalement ordonnés ;*
- *$R = (R_0, R_1, \ldots)$, une liste de $|R|$ sous-ensembles de E appelés régions ;*
- *$S = (S_0, S_1, \ldots)$, une liste de $|S|$ séries de données S_s de type T_s telles que $S_s[r][x]$ désigne la valeur, dans la série de rang s, de l'élément d'indice x dans la région R_r.*

Dans notre modèle, les données élémentaires que nous considérons sont en fait des n-uplets formés de n composantes de même type ($\mathbb{N}$ ou $\mathbb{R}$). L'utilisation de n-uplets est utile pour prendre en compte des scalaires, des vecteurs 2D, 3D ou encore des tenseurs. En pratique, les séries sont utilisées pour représenter les différentes grandeurs physiques qui sont associées aux éléments d'un objet.

Par exemple, dans une simulation de circulation océanique comme celle réalisée par le code POP, on peut associer aux nœuds d'une grille 3D représentant les océans deux séries de données (S_0, S_1) : la température de type $T_0 = \mathbb{R}$ et la pression également de type $T_1 = \mathbb{R}$. De même dans une simulation de dynamique moléculaire comme celle réalisée par le code NAMD, il est possible d'associer aux atomes trois séries (S_0, S_1, S_2) : le numéro de l'atome dans la molécule de type $T_0 = \mathbb{N}$, sa position dans l'espace de type $T_1 = \mathbb{R}^3$ et sa vitesse de type $T_2 = \mathbb{R}^3$.

Du point de vue de la redistribution, les séries vont permettre de factoriser le calcul de la matrice de communication. En effet les messages générés (en termes d'ensemble d'éléments) sont partagés par toutes les séries, dans la mesure où leurs calculs reposent uniquement sur la distribution des éléments et non sur les valeurs associés aux éléments dans les différentes séries. Les régions, quant à elles, permettent de regrouper logiquement certains éléments dans un sous-ensemble. Cela permet par exemple de séparer les composantes connexes d'un maillage, les blocs d'éléments dans un tableau, des boîtes d'atomes ou encore des espèces de particules différentes. Dans notre modèle, les régions sont des sous-ensembles d'éléments totalement ordonnés, selon un ordre dit *local*. Si cet ordre est simplement induit par l'ordre global des éléments de E, alors cet ordre est dit *local naturel*. Par ailleurs, comme nous le verrons plus tard, les régions permettent de considérer le problème de la redistribution avec une granularité moins fine que l'élément et donc de diminuer la complexité des algorithmes de redistribution sous certaines hypothèses.

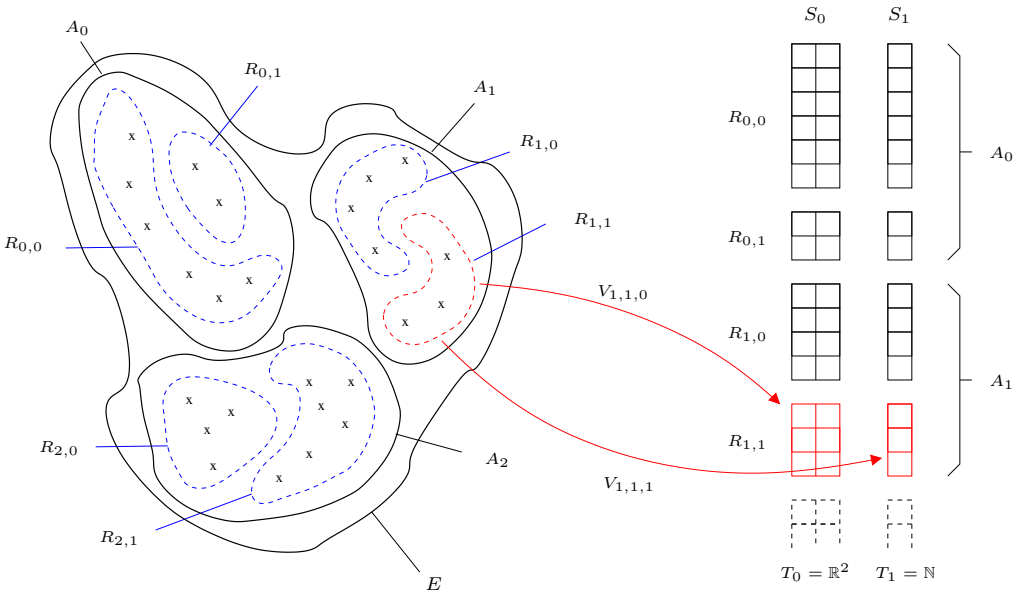


FIG. 5.12 – Exemple d'un objet complexe distribué sur 3 processeurs, possédant deux séries de données (de types S_0 et S_1) et deux régions par processeur (notées $R_{i,r}$).

Soit P un ensemble de processeurs de taille M. Nous allons maintenant introduire la notion *d'objet complexe distribué*. Un objet complexe distribué est une collection de M objets complexes affectés chacun à un processeur P_i. Tous ces objets partagent le même ensemble d'éléments E et les mêmes types de série. La distribution des éléments de E entre les différents objets est défini par une *fonction de distribution par région*, notée σ_P.

Définition 5.4.2 (Distribution par région) *Soit E un ensemble d'éléments découpé en M listes de régions $R_i = (R_{i,0}, R_{i,1}, \ldots)$ (de tailles variables $|R_i|$) affectées chacune au processeur P_i. Une distribution par région de E sur P*

est une fonction $\sigma_P : k \longrightarrow (i, r, x)$ qui à l'indice global k d'un élément de E associe un processeur de rang i dans P, une région d'indice r sur P_i et un indice local x dans la région $R_{i,r}$.

On dit alors que E est distribué sur P selon σ_P et on note $R_{i,r}[x]$ l'élément d'indice local x dans la région $R_{i,r}$ du processeur P_i avec $R_{i,r}[x] = E[k]$. La fonction de distribution par région peut être vue comme un généralisation de la fonction de distribution α_P, que nous avons présentée dans la section précédente (Déf. 5.3.2), prenant en compte la notion de régions. De la même manière que la distribution de E selon α_P induit une décomposition des éléments en des ensembles A_i, la distribution de E selon σ_P induit une décomposition de E en régions $R_{i,r}$ tels que $\bigcup_r R_{i,r} = A_i$. Ainsi, chaque processeur P_i possède localement un ensemble d'éléments A_i découpés logiquement en régions $R_{i,r}$ (Fig. 5.12).

Définition 5.4.3 (Objet complexe distribué) *Un objet complexe distribué sur P selon σ_P est une collection de M objets complexes $\mathcal{O}_i = (E, R_i, S_i)$ affectés chacun à un processeur P_i, notée $\mathcal{O}_P$, telle que :*

- *E est un ensemble d'éléments totalement ordonnés ;*
- *$R_i = (R_{i,0}, R_{i,1}, \ldots)$ est la liste des régions du processeur P_i, induites par une distribution σ_P ;*
- *$S_i = (S_{i,0}, S_{i,1}, \ldots)$ est la liste des séries de données du processeur P_i, telle que $S_{i,s}$ est de même type T_s pour tous les processeurs.*

Si le nombre de régions $|R_i|$ peut varier d'un processeur à l'autre, le type des séries T_s est identique pour tous les processeurs. Ainsi, chaque élément d'un objet $\mathcal{O}_i$ possède systématiquement une valeur dans toutes séries de données. On note $S_{i,s}[r][x]$ la valeur de l'élément $R_{i,r}[x]$ dans la série de rang s du processeur P_i.

5.4.2 Modèle de stockage

Considérons un objet complexe $\mathcal{O}_i = (E, R_i, S_i)$ distribué sur le processus P_i. Pour chaque série de données $S_{i,s}$ de type T_s, il est nécessaire de décrire précisément le stockage des données, afin de pouvoir retrouver efficacement dans la mémoire de chaque P_i l'adresse des éléments à envoyer ou à recevoir. Dans notre modèle, le stockage des éléments est défini à l'aide d'une *fonction d'adressage* pour chaque région et chaque série. Pour définir entièrement le stockage en mémoire de l'objet $\mathcal{O}_i$, il faut donc se donner une matrice de fonctions d'adressage $[V_{i,r,s}]$ de taille $|R_i| \times |S|$.

Définition 5.4.4 (Fonction d'adressage) *Une fonction d'adressage $V_{i,r,s}$ associe à l'indice x d'un élément dans la région $R_{i,r}$ l'adresse de la donnée $S_{i,s}[r][x]$ dans la mémoire de P_i, c'est-à-dire $V_{i,r,s}(x) = \&S_{i,s}[r][x]$.*

Chaque donnée de type T_s possède un emplacement mémoire élémentaire de taille t_s, car nous supposons que les composantes d'un n-uplet sont indissociables et contiguës en mémoire. Le modèle de stockage que nous proposons repose sur la définition d'un *espace de stockage* basé sur une approche classique en *stride*.

Définition 5.4.5 (Espace de stockage) *Soit $R_{i,r}$ une région de m éléments. Un espace de stockage de dimension d pour cette région est un quadruplet de la forme (A, F, L, Z) tel que :*

- *A est l'adresse de base de l'espace de stockage ;*
- *F est l'offset initial associée à l'espace de stockage ;*
- *$L = [0, l_0 - 1] \times [0, l_1 - 1] \times ... \times [0, l_{d-1} - 1]$ est l'ensemble des coordonnées de stockage avec $m = \prod_{k=0}^{d-1} l_k$;*
- *$Z = (z_0, z_1, \ldots, z_{d-1})$ est l'ensemble de saut ou* stride *(en octet) selon chaque dimension.*

Soit $R_{i,r}$ une région de m éléments. Soit (A, F, L, Z) un espace de stockage de dimension d pour cette région. La fonction d'adressage $V_{i,r,s}$ de la région $R_{i,r}$ pour la série de rang s est définie de la manière suivante. Pour tout élément $e \in R_{i,r}$ d'indice x ($0 \leq x < m$), il existe une coordonnée de stockage $c = (c_0, c_1, \ldots)$ dans L vérifiant $x = \sum_{k=0}^{d-1} \left(\prod_{k'=0}^{k-1} l_{k'} \right) . c_k$ et telle que l'adresse mémoire de la donnée associée à cet élément dans la série de rang s est : $V_{i,r,s}(x) = A + F + c_0.z_0 + c_1.z_1 + \ldots + c_{d-1}.z_{d-1}$ (Fig. 5.13).

Le *stride* z_k relatif à la dimension d'indice k dans l'espace de stockage représente le saut en octets qu'il faut effectuer dans la mémoire pour atteindre l'adresse de la donnée suivante selon cette dimension. Plus précisément, si l'on considère un élément e d'indice x et de coordonnée de stockage $c = (c_0, \ldots, c_d, \ldots c_{d-1})$ dans L, alors l'élément suivant selon la k-ème direction de l'espace est l'élément e' d'indice x' et de coordonnée

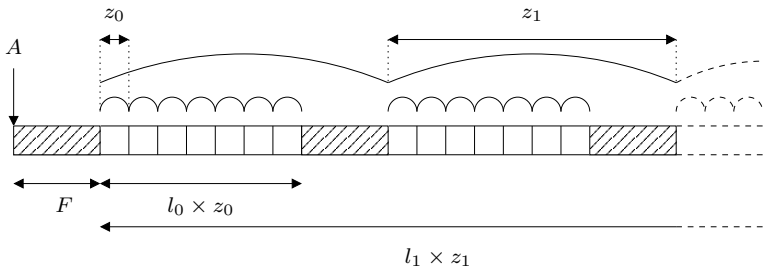


FIG. 5.13 – Modèle de stockage mémoire en *stride*.

de stockage $c' = (c_0, \ldots, c_d + 1, \ldots c_{d-1})$, tel que $x' = x + \prod_{k'=0}^{k-1} l_{k'}$ et $V_{i,r,s}(x') = V_{i,r,s}(x) + z_d$.

Considérons une région de m éléments et une série de type T (de taille t octets). Si les données sont entièrement contiguës en mémoire, alors une seule dimension est suffisante dans l'espace de stockage avec $l_0 = m$ et $z_0 = t$. Notons que si l'on souhaite filtrer un élément sur 2 dans cet espace, il faut alors prendre $l_0 = m/2$ et $z_0 = 2.t$. En jouant sur la valeur de l'offset initial, il est possible de sélectionner les valeurs des éléments d'indice paire ($F = 0$) ou impaire ($F = t$). De manière générale, si on a $z_0 > t$, alors aucune donnée n'est contiguë en mémoire et l'espace de stockage est dit entièrement « stridé ». Dans ce cas, il est souvent préférable d'utiliser un cache pour l'envoi des données formant un message. Considérons maintenant l'exemple de la figure 5.14 représentant un ensemble de 64 éléments, stockés à l'intérieur d'un tableau de taille 10×10, dont on extrait la bande extérieure (i.e. *ghost cells*). Le tableau débute à l'adresse A et contient des données de type T (de taille t octets) avec une composante par élément. On peut simplement représenter ces éléments en mémoire à l'aide de l'espace de stockage 2D de la forme (A, F, L, Z) avec $F = 11.t$, $L = [0, 7] \times [0, 7]$ et $Z = (z_0 = t, z_1 = 10.t)$.

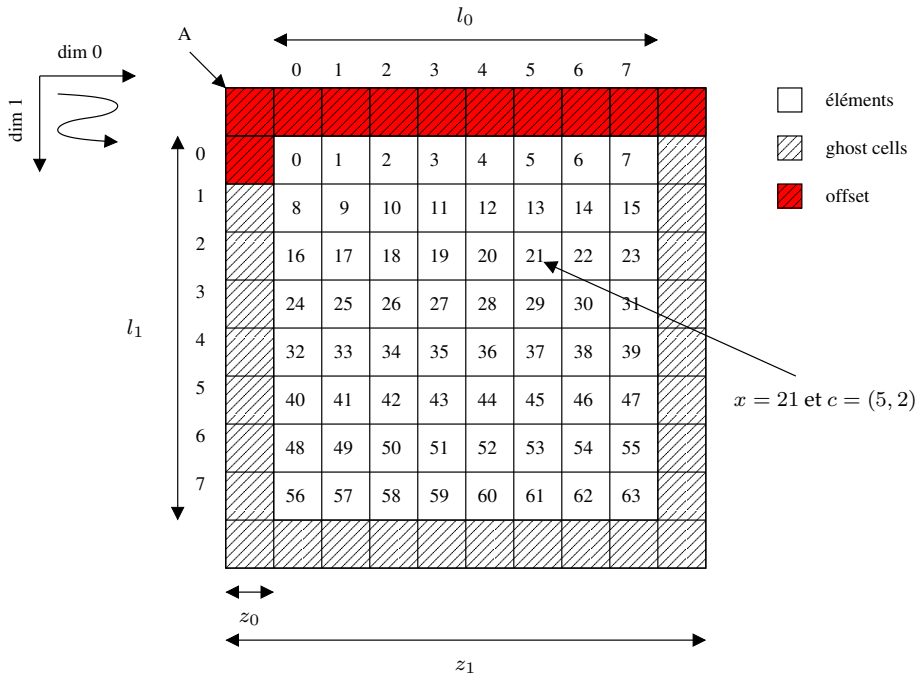


FIG. 5.14 – Exemple d'un espace de stockage 2D avec des cellules fantômes ou *ghost cells* entourant le tableau.

En conclusion, nous soulignerons que ce modèle de stockage ne permet pas d'accéder à une séquence d'adresse mémoire quelconque, toutefois il est bien adapté à notre problématique, car il permet de prendre en compte une grande variété de structures de données (e.g. données contiguës, données entièrement *stridées*, *ghost cells*, *etc.*), tout en offrant un accès aléatoire à la mémoire efficace. La complexité de cet accès est en $\mathcal{O}(d)$ avec d le nombre de dimensions de l'espace de stockage (le plus souvent inférieur à 3).

5.4.3 Les différentes classes d'objets

Sur la base de ce modèle, nous avons proposé la définition de plusieurs classes d'objets complexes (distribués) : les grilles structurées, les boîtes d'atomes, les ensembles de particules, les maillages non structurés ainsi que les paramètres. Tous ces objets, hormis les paramètres, entrent dans la famille des *objets spatiaux*.

Définition 5.4.6 (Objet spatial) *Un objet complexe $\mathcal{O}$ est dit spatial dans $\mathbb{R}^n$ si il existe une fonction coordonnée φ de E vers $\mathbb{R}^n$, telle que $\forall e \in E,\ \ \varphi(e) = (x, y, ...)$ désigne la coordonnée spatiale de e.*

Les objets spatiaux sont très fréquents dans les codes de simulation numérique. Ils présentent un intérêt tout particulier dans le contexte de cette thèse, car il est possible de leur associer naturellement une représentation graphique dans un espace physique à n dimensions, et donc de les visualiser. Le nombre de dimensions n est généralement égal à 2 ou 3 mais peut atteindre 4 ou 6 dans certains codes manipulant l'espace des phases (position, vitesse).

Les objets complexes telles que nous les avons définis sont découpés en régions logiques. Ces régions jouent un rôle important dans notre modèle car elles permettent de décrire la distribution des éléments par « paquet ». Selon la classe de l'objet considéré, les régions prennent des significations différentes. Par exemple, pour les grilles structurées ou les boîtes d'atomes, les régions que nous considérons sont des ensembles d'éléments contenues à l'intérieur de *blocs* parallélépipédiques de l'espace. Pour les ensembles de particules ou les maillages non structurés, les régions représentent simplement des groupes d'éléments, partageant le plus souvent des caractéristiques communes dans la simulation. Dans ce cas, les régions sont identifiées dans la distribution à l'aide d'un *tag* numérique unique. Par ailleurs, les régions définissent dans notre modèle des espaces de stockage autonomes. Un des aspects original de notre modèle est qu'il n'utilise pas de numérotation globale sur les éléments distribués, comme il est en principe nécessaire pour redistribuer des objets irréguliers (cf. principe de linéarisation, Sec. 5.2.2). En effet, il est intéressant de pouvoir se passer de cette information coûteuse à stocker et à manipuler dans un algorithme de redistribution (i.e. un indice global par élément). Notre approche se base uniquement sur *une numérotation locale* des éléments au sein des régions et sur certaines propriétés de ces régions permettant de résoudre le problème de la redistribution, comme par exemple la position des blocs dans l'espace ou les *tags* sur les régions. Nous allons maintenant présenter plus en détails les différentes classes d'objets. Le tableau 5.2 résume les principales caractéristiques de ces objets. Nous décrirons à la section 5.6 des algorithmes de redistribution adaptés à ces objets et nous verrons au chapitre 6 une implantation possible de ce modèle de données dans la bibliothèque *RedSYM*.

Les grilles structurées

Les grilles structurées sont une classe d'objet très répandue dans les codes de simulation, servant le plus souvent à représenter une discrétisation de l'espace physique. Comme nous l'avons vu dans la section 2.1.2, ces objets possèdent une topologie régulière, formée de quadrilatères en 2D et d'hexaèdres en 3D. D'un point de vue informatique, les grilles structurées correspondent à des tableaux de données multidimensionnels. Les éléments que nous considérons pour cet objet sont les nœuds de la grille, auxquels nous associons diverses séries de données[3]. Par ailleurs, les grilles structurées possèdent un système de coordonnées naturel $(i, j, \ldots)$ dans $\mathbb{Z}^n$ (espace topologique), permettant d'identifier de manière unique chaque nœud dans la grille. De plus, chaque nœud possède une coordonnée $(x, y, \ldots)$ dans l'espace physique $\mathbb{R}^n$. Si la grille est régulière (Fig. 5.15 (a)), cette dernière coordonnée peut être calculée simplement à partir de sa coordonnée topologique $(i, j, \ldots)$ en prenant en compte l'espacement entre les nœuds selon toutes les directions de l'espace $(dx, dy, \ldots)$. En revanche, si la grille est irrégulière (Fig. 5.15 (b)), nous utilisons une série de données particulière de type $\mathbb{R}^n$, contenant les coordonnées spatiales de tous les nœuds. En outre, il est possible d'utiliser les séries de données pour associer aux nœuds de multiples informations scalaires ou vectorielles, comme par exemple des champs de pression, de force ou de vitesse.

[3]Notons qu'il est également possible d'étendre ce modèle pour considérer la grille duale où les éléments de l'objet complexe seraient les quadrilatères ou les hexaèdres de la grille. Toutefois, nous nous limiterons dans cette étude au cas où les éléments sont les nœuds, ce qui est plus simple à représenter graphiquement (interpolation des couleurs entre les nœuds).

Objets Complexes	Informations descriptives
grilles structurées	- nombre de dimensions : n - élément de l'objet : nœuds de la grille - région de l'objet : bloc dans $\mathbb{Z}^n$ - distribution : liste de blocs dans $\mathbb{Z}^n$ (sous-domaine) - domaine D dans $\mathbb{Z}^n$ - flag d'envoi/réception pour chaque bloc - espacement entre les nœuds (si grille régulière) - séries de données associées aux nœuds - série des coordonnées dans $\mathbb{R}^n$ (si grille irrégulière) - convention de numérotation : *row major* ou *column major*
boîtes d'atomes	- nombre de dimensions : n - élément de l'objet complexe : atome - région de l'objet : boîte (ou bloc) dans $\mathbb{R}^n$ - distribution : liste de blocs dans $\mathbb{R}^n$ (sous-domaine) - domaine D dans $\mathbb{R}^n$ - nombre courant/maximum d'atomes pour chaque région (bloc) - séries de données associées aux atomes - série des coordonnées dans $\mathbb{R}^n$ - rayon des atomes
ensembles de particules	- nombre de dimensions : n - élément de l'objet : particule - région de l'objet : paquet de particules - distribution : ensemble de régions identifiées par un *tag* - nombre courant/maximum de particules pour chaque région - séries de données associées aux particules - série des coordonnées dans $\mathbb{R}^n$ - rayon des particules
maillages non structurés	- nombre de dimensions : n - élément de l'objet : cellule géométrique - région de l'objet : composante du maillage - distribution : ensemble de régions identifiées par un *tag* - type de la cellule (*tri, quad, hex, tetra, ...*) - nombre de nœuds par élément : p - nombre courant/maximum d'éléments pour chaque région - nombre courant/maximum de nœuds pour chaque région - séries de données associées aux éléments - séries de données associées aux nœuds - série des cellules dans $\mathbb{N}^p$ (liste de connectivités) - série des nœuds dans $\mathbb{R}^n$ (coordonnées) - convention de numérotation : *C* ou *Fortran*

TAB. 5.2 – Description synthétique des différentes classes d'objets.

Dans le cas des grilles structurées, les régions que nous considérons sont des *blocs* d'éléments. Plus formellement, un bloc B est un parallélépipède rectangle de l'espace topologique $\mathbb{Z}^n$, décrit par deux points extrêmes $a = (a_0, a_1, \ldots)$ et $b = (b_0, b_1, \ldots)$, tels que $p = (p_0, p_1, \ldots) \in B$ si et seulement si $\forall i, a_i \leq p_i \leq b_i$. Par la suite, un bloc est noté $B = (a, b)$. Ce bloc joue le rôle d'un conteneur pour les éléments de la grille, et le nombre d'éléments au sein d'un bloc est contraint par le produit de ses dimensions : $(b_0 - a_0 + 1) \times (b_1 - a_1 + 1) \times \ldots$ A titre d'exemple, la partie hachurée sur la figure 5.15 représente le bloc $((2, 1), (3, 3))$ de dimensions 2×3, qui désigne une région de 6 éléments correspondant aux indices suivants $\{(2, 1), (2, 2), (2, 3), (3, 1), (3, 2), (3, 3)\}$. Ainsi, si l'on considère une liste de régions $R_i = (R_{i,0}, R_{i,1}, \ldots)$ affectées au processeur P_i, la distribution des éléments est simplement définie par la liste de blocs $B_i = (B_{i,0}, B_{i,1}, \ldots)$ respectivement associés aux

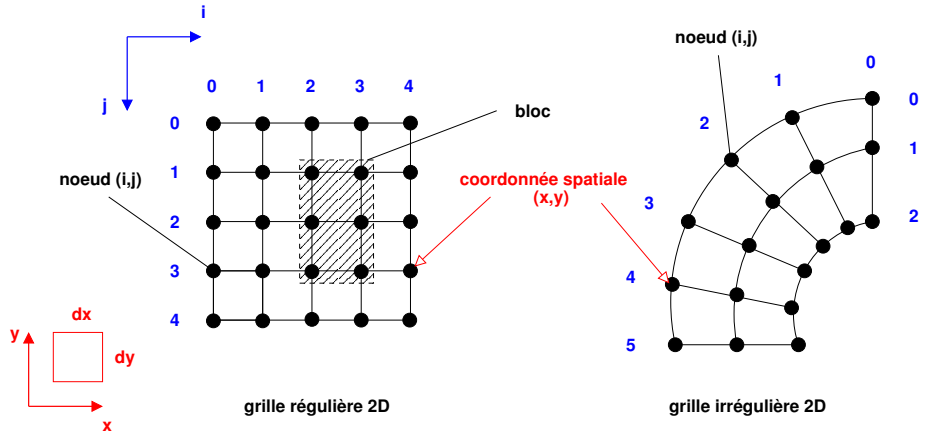


FIG. 5.15 – Exemples de grilles 2D.

régions. Les distributions sont généralement issues de la décomposition d'un bloc D, appelé *domaine global*, en sous-blocs $B_{i,r}$ sur un ensemble de processeurs P. La liste des blocs du processeur P_i forme ce qu'on appelle le *sous-domaine* de P_i. Ainsi, chaque processeur possède un ensemble de blocs tout à fait quelconque, pouvant se recouper ou ne pas couvrir la totalité du domaine D. Ce modèle de description par blocs s'avère très générique et très flexible pour représenter les distributions de données qui sont réellement présentes dans les codes de simulation. Par exemple, il est capable de représenter des *distributions creuses* comme celle qui est utilisée dans la simulation POP pour représenter la surface des océans sans les continents (Fig. 5.16). Contrairement aux travaux précédents (cf. Sec. 5.2.1), nous ne nous limitons pas à des distributions denses rectilinéaires ou simplement bloc-cyliques, qui sont en fait des cas particuliers dans notre modèle. Toutefois, cette approche nécessite de stocker explicitement les coordonnées des points extrêmes de chaque bloc, ce qui peut s'avérer coûteux si la distribution est fine. A titre indicatif, la taille du descripteur pour une distribution bloc-cyclique en HPF est en $\mathcal{O}(3.n)$, alors que dans notre modèle une description équivalente de cette distribution sera en $\mathcal{O}(k \times 2.n)$ avec k le nombre total de blocs dans la distribution.

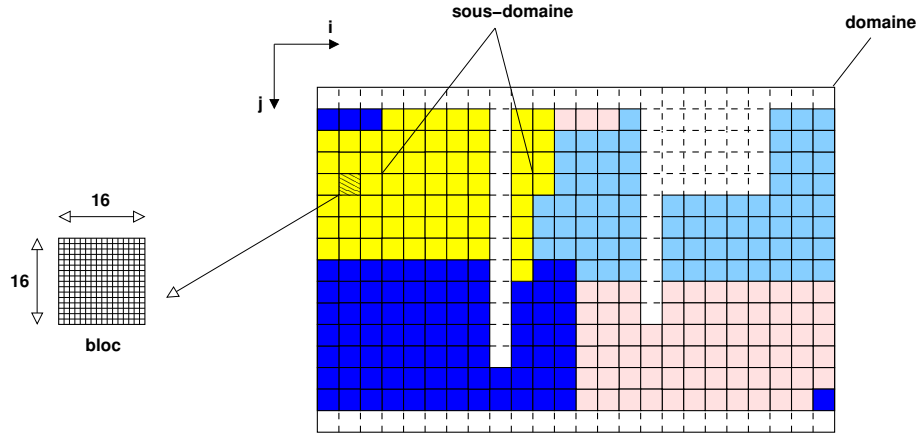


FIG. 5.16 – Exemple d'une distribution creuse par blocs sur 4 processeurs utilisée dans POP : chaque bloc est de dimension 16×16 dans une grille structurée 384×256.

Notre modèle permet également de prendre en compte des *ghost cells* au niveau de la distribution (pas simplement au niveau du stockage). Pour ce faire, nous utilisons un système de *flags* permettant de marquer si un bloc dans la distribution est utilisé pour l'envoi, la réception ou les deux (la valeur par défaut). Dans le cas d'un *ghost*, il faut remarquer que les éléments « fantômes » sont en réalité affectés à un autre processeur. Par conséquent, il est intéressant de les exclure de la région d'envoi (pour éviter de les transférer deux fois). En revanche, il peut être utile de les considérer dans la région de réception afin de mettre à jour leurs valeurs lors de la redistribution. En résumé, le bloc contenant le *ghost* est marqué par un flag de réception, alors que le bloc

privé du *ghost* est marqué avec un flag d'envoi (Fig. 5.17). Dans le premier cas, l'espace de stockage sous-jacent est *a priori* contigu 1D, alors qu'il est 2D avec stride dans le second cas.

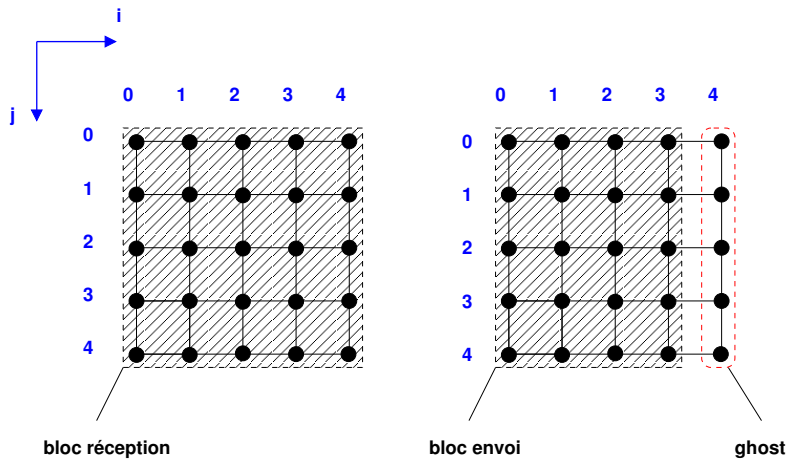


FIG. 5.17 – Exemple d'un *ghost* : les éléments « fantômes » ne sont pas utilisés pour l'envoi, mais sont mis à jour à la réception.

Les boîtes d'atomes

Les *boîtes d'atomes* sont des objets spatiaux représentant des atomes dans l'espace physique $\mathbb{R}^n$ distribués à l'intérieur de boîtes comparables aux distributions par blocs pour les grilles structurées. Dans ce cas, les blocs (ou boîtes) sont définies dans $\mathbb{R}^n$ et non plus dans $\mathbb{Z}^n$ comme pour les grilles structurées. Ils jouent le rôle de conteneurs pour les éléments de l'objet, c'est-à-dire les atomes (Fig. 5.18). Chaque atome est identifié par un indice local dans la boîte et possède une coordonnée spatiale $(x, y, \ldots)$ stockée explicitement dans une série de données particulière, la série des coordonnées. A chaque bloc $B_{i,r}$ correspond une région d'éléments $R_{i,r}$ sur P_i, telle que $e \in R_{i,r}$ si et seulement si $\varphi(e) \in B_{i,r}$. Contrairement aux grilles structurées, le nombre d'éléments contenus par un bloc n'est pas contraint et peut varier au cours de la simulation. On considère alors que les atomes sont mobiles, et qu'ils peuvent migrer d'une boîte à l'autre, apparaître ou disparaître. Afin de supporter ce niveau de dynamicité, nous distinguons dans notre modèle le nombre courant d'atomes dans une région et le nombre maximum d'atomes que peut recevoir l'espace de stockage. Comme pour les grilles structurées, les distributions par blocs que nous considérons sont simplement définies par un ensemble de blocs affectés à un processeur P_i. Cette ensemble *a priori* quelconque forme le sous-domaine de P_i, typiquement inclus à l'intérieur d'un bloc D, formant le domaine global. Afin de faciliter la visualisation de ces objets, l'utilisateur peut éventuellement préciser le rayon de l'atome, ce qui permet une mise à l'échelle de la scène. Ce type d'objet est très fréquent dans les codes de dynamique moléculaire basées sur une technique de décomposition spatiale [176]. Dans la simulation NAMD [199] par exemple, les atomes sont distribués selon un découpage de l'espace en boîtes de $\mathbb{R}^3$ (*patch*), placées plus ou moins régulièrement sur les processus afin d'équilibrer la charge des calculs. On trouve également de tels objets dans des simulations en astrophysique basées sur la méthode SPH (*Smoothed Particule Hydrodynamics* [182]). Cette méthode numérique permet de modéliser un fluide, un gaz ou une galaxie comme un ensemble de particules (formulation Lagrangienne de la dynamique des fluides).

Les ensembles de particules

Les *ensembles de particules* sont des objets spatiaux non structurés représentant des particules dispersées dans l'espace physique $\mathbb{R}^n$. Une série de données particulière sert à décrire la position des particules dans l'espace. Contrairement aux *boîtes d'atomes*, l'ensemble des particules formant une région est *a priori* quelconque (i.e. pas de contrainte sur la position des particules dans la région). Les régions permettent de regrouper des « paquets de particules » ayant le plus souvent une caractéristique commune dans la simulation, comme par exemple des particules chargées positivement ou négativement (Fig. 5.19) ou encore différentes espèces chimiques, *etc*.

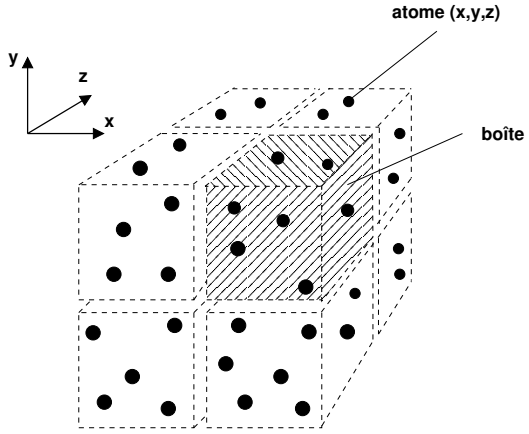


FIG. 5.18 – Exemple de boîtes d'atomes 3D.

Il est également possible d'utiliser cet objet pour décrire des molécules, vues comme des ensembles d'atomes sans liaisons. Ces molécules sont typiquement décomposées en fragments (peptides) qui vont jouer le rôle des régions. Par ailleurs, il faut noter qu'il est possible de reconstruire entièrement la structure d'une molécule (i.e. les liaisons, les potentiels, *etc*.) pour la visualiser à partir des informations sur la position des atomes, du type du fragment et de l'indice de l'atome dans le fragment (cf. VMD [30]). Chaque élément est identifié par un indice local dans sa région et chaque région est identifiés dans la distribution, par une étiquette entière ou *tag*. Par la suite, ces étiquettes vont nous servir à mettre en correspondance les régions partagées entre les codes couplés. Comme pour les boîtes d'atomes, le nombre de particules dans une région peut fluctuer dynamiquement au cours de la simulation. Ainsi nous distinguons pour chaque région le nombre courant de particules et le nombre maximum de particules que peut recevoir l'espace de stockage sous-jacent à chaque région.

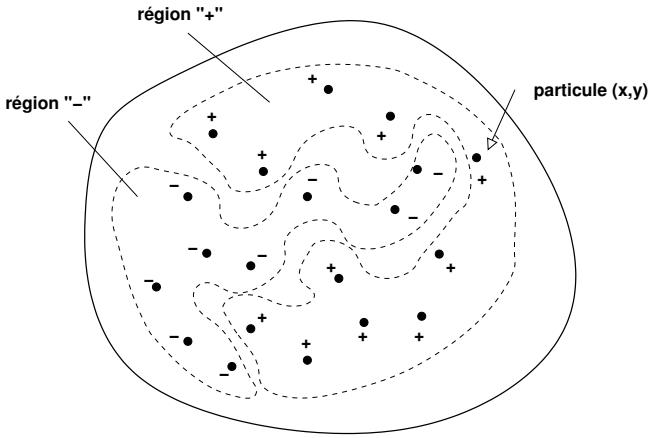


FIG. 5.19 – Exemple d'un ensemble de particules dans $\mathbb{R}^2$ constitué de deux régions : les particules chargées positivement et celles chargées négativement.

Les maillages non structurés

Les *maillages non structurés* sont des objets[4] spatiaux représentant un ensemble de cellules géométriques dans l'espace $\mathbb{R}^n$ ($n = 2$ ou 3). Actuellement, nous ne prenons en compte que le cas de maillages formés d'un seul type d'élément : triangle, quadrilatère, hexaèdre ou tetraèdre. Les éléments de l'objet complexe (que nous considérons du point de vue de la redistribution) sont à proprement parler les cellules du maillage. Nous nous limiterons à ce cas dans nos travaux, même si il est possible d'étendre notre modèle au cas dual où les éléments

[4]Les maillages non structurés sont en fait définis comme une paire d'objet complexe, $\mathcal{O}^{cell}$ et $\mathcal{O}^{node}$, pour les cellules et pour les nœuds. Nous les considérerons abusivement dans notre discours comme un objet complexe classique $\mathcal{O} = \mathcal{O}^{cell}$, car la distribution des nœuds est induite par celle des cellules et réciproquement.

de l'objet sont les nœuds du maillage. Ainsi, chaque région décrit un bout de maillage défini comme un ensemble d'éléments géométriques connectant les nœuds du maillage (Fig. 5.20). Chaque élément géométrique est défini à l'aide d'une série de données particulière dans $\mathbb{N}^p$, représentant la liste des connectivités, avec p le nombre de nœuds par élément (e.g. $p = 3$ pour des triangles, 4 pour des quadrilatères, *etc.*). Cette liste contient les indices locaux des nœuds dans la région servant à former chaque élément. Par exemple, le quadrilatère d'indice local 2 sur la figure 5.20 (a) est formé à partir des quatre nœuds d'indice locaux $\{c, d, h, g\}$. Les coordonnées des nœuds associées à la région sont représentées à l'aide d'une série de données particulière dans $\mathbb{R}^n$. Outre la liste de connectivités et les coordonnées des nœuds, les maillages peuvent posséder deux types de séries de données, les données associées aux éléments géométriques et les données associées aux nœuds du maillage. Dans le cas distribué, le maillage est partitionné en plusieurs composantes (i.e. les régions). Ces régions sont typiquement placés sur différents processeurs pour équilibrer la charge. Comme pour les ensembles de particules, il est possible d'associer un *tag* aux régions pour les identifier précisément au sein d'une distribution. Les distributions que nous considérons sont uniquement des distributions des éléments géométriques (et non des nœuds). Dans ce type de distribution, les éléments ne sont jamais coupés, et chaque élément du maillage est affecté à un et un seul processeur (si l'on suppose qu'il n'y a pas d'éléments « fantômes »). En revanche, les nœuds situés sur les frontières des partitions sont répliqués entre les différents processeurs.

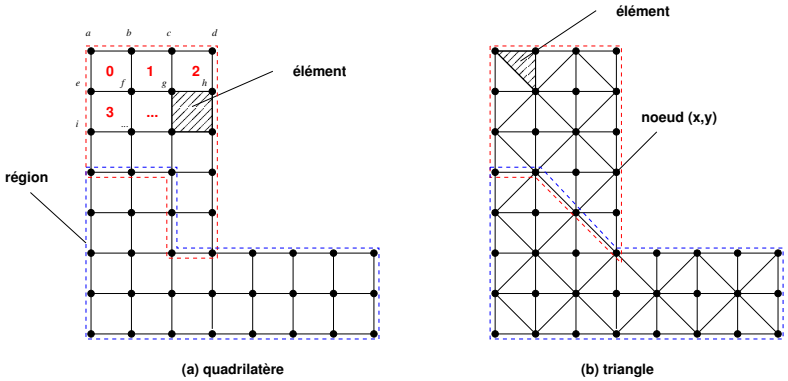


FIG. 5.20 – Exemple d'un maillage 2D formé (a) de quadrilatères et (b) de triangles.

Les paramètres

La dernière classe d'objet que nous considérons sont les *paramètres*. Ces objets sont fréquemment utilisés pour décrire les jeux de paramètres scalaires et vectoriels dans les simulations numériques. Du point de vue de la distribution, nous considérons le cas où le paramètre est dupliqué sur tous les processeurs (*replicated*), et le cas où le paramètre est localisé sur un seul processeur (*located*), en général P_0. Comme tout type d'objet complexe, les paramètres peuvent posséder une ou plusieurs séries de données. En revanche, ils ne possèdent qu'une seule région et qu'un seul élément dans cette région. Ainsi, un objet de classe paramètre peut contenir plusieurs séries de données scalaires et vectorielles (i.e. plusieurs composantes). Ces objets sont importants dans le contexte du pilotage, car ils permettent typiquement à l'utilisateur de surveiller le comportement de la simulation (diagramme temporel 1D) et de la piloter simplement en changeant certaines valeurs des paramètres en des points opportuns du code.

5.5 Les messages symboliques

5.5.1 Définitions

Nous allons maintenant poser quelques définitions et introduire la notion de *message symbolique*, une représentation intermédiaire des messages que nous utiliserons par la suite pour formuler un algorithme de redistribution générique (Sec. 5.6).

Définition 5.5.1 (Masque) *Un masque $\mathcal{M}$ est une liste d'intervalles dans $\mathbb{N}$ de la forme $([u_0, v_0], [u_1, v_1], [u_2, v_2], \ldots)$.*

Définition 5.5.2 (Extraction) *L'opération d'extraction associe à un masque $\mathcal{M}$ et à un ensemble d'éléments R, le sous-ensemble d'éléments $R' \subset R$ tel que $e \in R'$ si et seulement si l'indice x de l'élément e dans R appartient à au moins un des intervalles de $\mathcal{M}$. On note $R' = extract(R, \mathcal{M})$ cette opération.*

La *longueur* d'un masque est égal au nombre d'intervalles qui le compose. Un masque est dit *strictement croissant* si les intervalles vérifient $\forall i,\ v_i < u_{i+1}$. Un masque $\mathcal{M}$ est dit vide s'il ne contient aucun intervalle. A l'inverse, il est dit « plein » s'il possède un intervalle unique tel que $extract(R, \mathcal{M}) = R$. Par la suite, les masques vont s'appliquer exclusivement aux régions d'éléments des objets complexes. Aussi, la définition des intervalles d'éléments utilisera systématiquement les indices locaux à cette région. La figure 5.21 illustre une extraction à l'aide d'un masque strictement croissant.

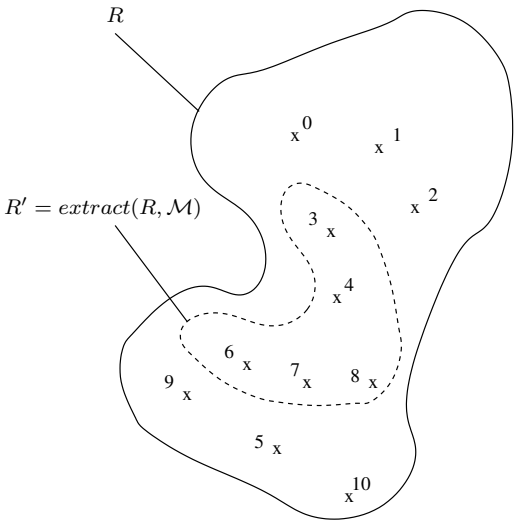


FIG. 5.21 – Extraction d'un sous-ensemble d'éléments R' à l'aide du masque $\mathcal{M} = ([3,4],[6,8])$ sur R.

Définition 5.5.3 (Message symbolique) *Soient $\mathcal{O}_i$ et $\mathcal{O}'_j$ deux objets complexes respectivement distribués sur P_i et Q_j. Un message symbolique $\hat{m}_{i,j}$ de P_i vers Q_j est une séquence de sous-messages, notée $(\hat{m}_{i,j,0}, \hat{m}_{i,j,1}, \hat{m}_{i,j,2}, \ldots)$, tel que le k-ème sous-message symbolique, $\hat{m}_{i,j,k}$, soit un triplet de la forme $(r_k, r'_k, \mathcal{M}_k)$ avec :*

- *r_k le numéro de la région source R_{i,r_k} de P_i,*
- *r'_k le numéro de la région cible R'_{j,r'_k} de Q_j,*
- *$\mathcal{M}_k$ un masque d'extraction sur la région source déterminant la liste ordonnée des éléments relatifs au k-ème sous-message.*

Le k-ème sous-message symbolique, $\hat{m}_{i,j,k}$, représente l'ensemble des éléments qu'il faut communiquer de la région source R_{i,r_k} vers la région cible R'_{j,r'_k}. En accord avec le principe d'intersection, le masque $\mathcal{M}_k$ doit être tel que $R_{i,r_k} \cap R'_{j,r'_k} = extract(R_{i,r_k}, \mathcal{M}_k)$. Ainsi, le message $m_{i,j}$ relatif à $\hat{m}_{i,j}$ est constitué de l'ensemble des éléments résultant de l'extraction successive de tous les sous-messages : $m_{i,j} = \bigcup_k extract(R_{i,r_k}, \mathcal{M}_k)$. L'ordre des masques dans le message symbolique, combiné à l'ordre des intervalles dans chaque masque, définit un ordre sur les éléments du message $m_{i,j}$. Plus précisément, si l'on note $[u_{k,l}, v_{k,l}]$ le l-ème intervalle du k-ème masque dans le message symbolique $\hat{m}_{i,j}$, alors les éléments formant le message sont ordonnés de la manière suivante : $u_{0,0}, u_{0,0}+1, \ldots, v_{0,0}, u_{0,1}, \ldots, v_{0,1}, \ldots, u_{1,0}, \ldots, v_{1,0}, \ldots$

Définition 5.5.4 (Message physique) *Un message physique, noté $\bar{m}$, est une séquence de la forme $((ad_0, sz_0), (ad_1, sz_1), \ldots)$, tel que chaque paire (ad_k, sz_k) désigne une plage de données en mémoire de taille sz_k (en octet), débutant à l'adresse ad_k.*

Un message physique désigne l'emplacement en mémoire des données à envoyer ou à recevoir, d'une manière compréhensible par une couche de communication.

Définition 5.5.5 (Sérialisation) *La sérialisation est l'opération qui consiste à générer pour la série de rang s de l'objet $\mathcal{O}_i$, le message physique noté $\bar{m}_{i,j,s}$ à partir du message symbolique $\hat{m}_{i,j}$. On note serialize$(\hat{m}_{i,j}, S_{i,s})$ cette opération.*

Connaissant le message symbolique $\hat{m}_{i,j} = (\hat{m}_{i,j,0}, \hat{m}_{i,j,1}, \ldots)$ et les fonctions d'adressage $V_{i,r,s}$, il est possible de calculer le message physique $\bar{m}_{i,j,s} = ((ad_0, sz_0), (ad_1, sz_1), \ldots)$ en respectant d'une part l'ordre sur les sous-messages $\hat{m}_{i,j,k}$ défini par le masque dans le message symbolique, et d'autre part l'ordre sur les intervalles dans chaque masque $\mathcal{M}_k$ (Fig. 5.22). Notons que dans le cas d'un espace de stockage contigu, une seule plage mémoire (ad, sz) est associée à chaque intervalle $[u, v]$ du masque $\mathcal{M}_k$, telle que $ad = V_{i,r,s}(u)$ et $sz = (v - u + 1).t_s$ avec T_s le type de la série et t_s sa taille en octets. Les choses sont un peu plus compliquées si l'espace de stockage n'est pas contigu, car dans ce cas, plusieurs plages mémoires disjointes peuvent correspondre au même intervalle d'un masque. Quoiqu'il en soit, les plages mémoire du message physique relatives à n'importe quel sous-message peuvent être calculées analytiquement en temps quasi-constant (complexité en $\mathcal{O}(d)$ avec d généralement inférieur à 3).

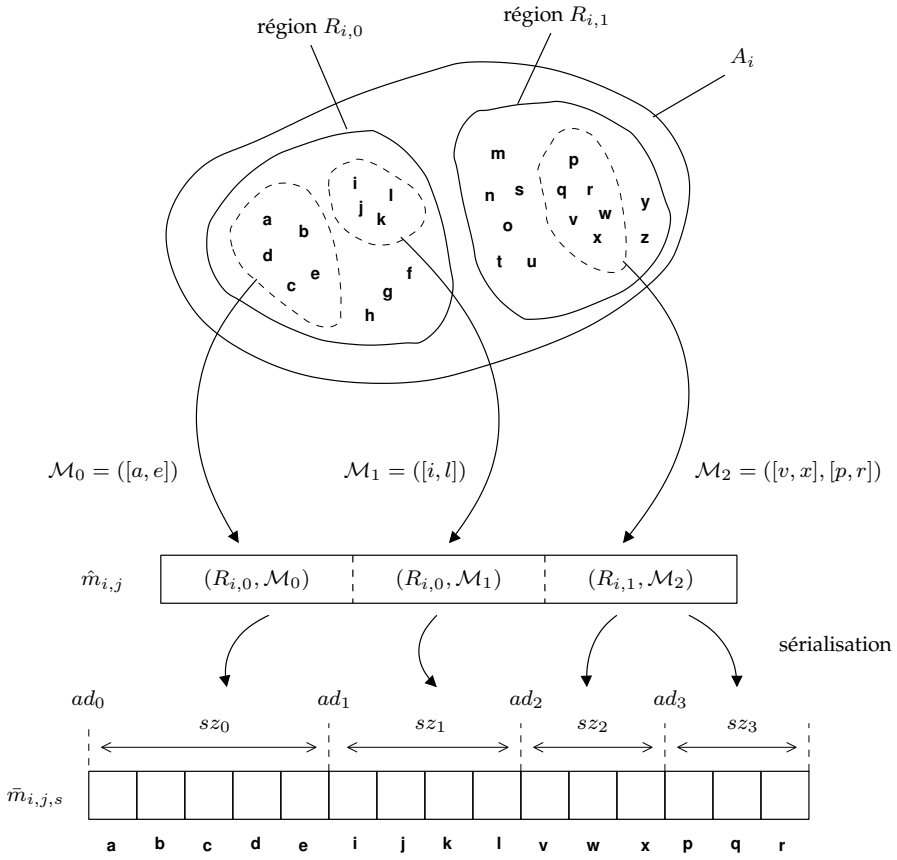


FIG. 5.22 – Sérialisation du message symbolique $\hat{m}_{i,j}$ en messages physique, pour la série de données de rang s.

Même si les messages symboliques peuvent être vus comme une abstraction servant uniquement à calculer les messages physiques, nous soulignons qu'il est intéressant pour plusieurs raisons de conserver leur description en mémoire (*messages symboliques persistants*). Tout d'abord, la couche de communication peut utiliser cette information pour construire ou reconstruire dynamiquement les messages physiques à la demande, ce qui peut s'avérer utile lorsque par exemple l'espace de stockage des données est modifié. La mise à jour des messages physiques est alors très efficace, car elle ne nécessite pas de recalculer les messages symboliques, mais uniquement de recommencer la sérialisation. Un « re-calcul » complet des messages symboliques s'avère nécessaire lorsque la distribution des éléments change entre les codes couplés, un évènement que l'on peut considérer comme rare dans la plupart des codes (i.e. rééquilibrage dynamique de la charge). Le stockage des messages symboliques a un coût mémoire qui dépend directement de la longueur des masques d'extraction calculés. Ce coût doit être relativisé par le fait que les messages symboliques sont une information commune à toutes les séries, plus compacte que les messages physiques. Par ailleurs, il faut souligner que la connaissance des messages symboliques offre une grande flexibilité à la couche de communication. Elle permet d'accéder aux données du message directement en mémoire, ce qui peut servir à effectuer des traitements divers et variés sur les données

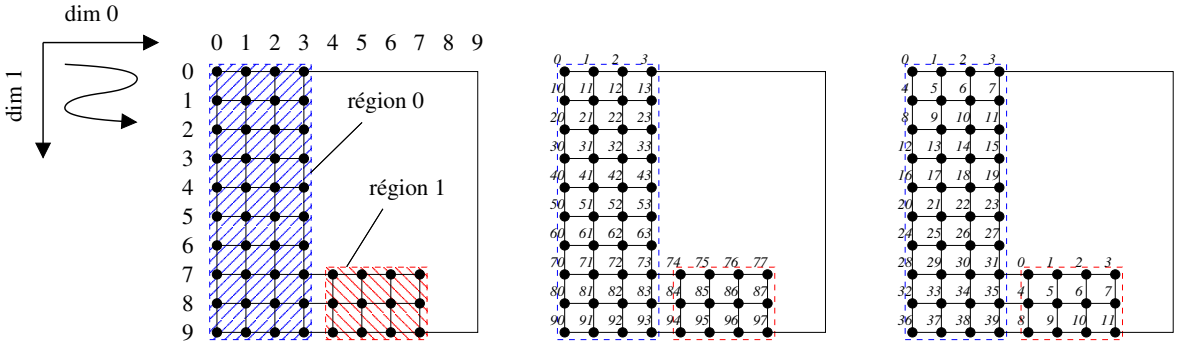
à l'envoi et/ou à la réception. Parmi ces traitements, on peut par exemple envisager de faire des conversions de types, du filtrage et/ou de la compression sur les données, des envois bufferisés (i.e. utilisation d'un cache), *etc.* Il est également possible de mettre en place une politique d'ordonnancement, ce qui va nécessiter de « surdécouper » les messages à envoyer.

5.5.2 Un peu d'ordre dans les messages !

L'un des aspects les plus délicats du problème de la redistribution est le choix de l'ordre des éléments au sein des messages. Plus précisément, pour que P_i parvienne à communiquer « correctement » avec Q_j, il faut que l'ordre des éléments dans le message $m_{i,j}$ soit identique entre l'émetteur P_i et le receveur Q_j. En d'autres termes, cela signifie que le message symbolique $\hat{m}_{i,j}$ calculé localement par P_i désigne la même séquence d'éléments que le message symbolique distant $\hat{m}'_{j,i}$ calculé par Q_j. Dans ce cas, on dit que le message symbolique $\hat{m}_{i,j}$ est *ordonné canoniquement*.

Définition 5.5.6 (Ordre canonique) *Le message $\hat{m}_{i,j}$ est ordonné canoniquement si et seulement si pour tout sous-message $\hat{m}_{i,j,k} = (r_k, r'_k, \mathcal{M}_k)$, on a $\hat{m}'_{j,i,k} = (r'_k, r_k, \mathcal{M}'_k)$ tel que $\mathcal{M}'_k$ soit un masque sur R'_{j,r'_k} qui désigne la même séquence d'éléments que $\mathcal{M}_k$ sur R_{i,r_k}.*

Une stratégie qui peut sembler naturelle pour construire des messages canoniques consiste à respecter *l'ordre global* sur les éléments. L'avantage de cette approche est qu'elle permet de transmettre les éléments dans un ordre *a priori* connu des deux codes couplés. Il faut remarquer cependant que cet ordre n'est pas nécessairement optimal, car il risque de ne pas préserver au maximum la continuité des éléments en mémoire, qui sont stockés région par région[5]. De plus, si les structures de données sont irrégulières, il est nécessaire d'associer explicitement à chaque élément son indice global dans E pour pouvoir accéder à cet ordre. Cela implique donc d'échanger entre chaque code les indices globaux de tous les éléments, ce qui s'avère particulièrement coûteux (cf. principe de linéarisation, Sec. 5.2.2). Par ailleurs, cette stratégie suppose que l'ordre global est connu des codes couplés, ce qui n'est pas toujours le cas. En effet, les codes de simulations parallèles se contentent le plus souvent de manipuler une numérotation locale sur les éléments.



(a) blocs de P0 (b) ordre global (c) ordre local naturel

FIG. 5.23 – Ordre global et ordre local naturel dans le cas des grilles structurées.

Toutes ces remarques nous ont motivé à rechercher une autre approche pour résoudre le problème de la redistribution et plus particulièrement pour ordonner les éléments au sein des messages. Notre stratégie consiste à utiliser *l'ordre local* sur les régions plutôt que *l'ordre global*. Plus précisément, chaque sous-message symbolique $\hat{m}_{i,j,k} = (r_k, r'_k, M_k)$ est défini en se basant sur l'ordre local des éléments dans la région R_{i,r_k}, ce qui revient à construire un masque M_k strictement croissant. Cette approche permet d'ordonner les messages en maximisant la continuité des données en mémoire (stockées selon l'ordre local des régions), ce qui joue favorablement sur la performance des transferts. Toutefois, rien ne garantit *a priori* que les éléments du message $m_{i,j}$ seront ordonnés identiquement sur P_i et Q_j. Ainsi, si l'ordre choisi « localement » pour construire $\hat{m}_{i,j}$

[5]En effet, l'ordre global peut être transversal par rapport aux unités de stockage que sont les régions. Considérer par exemple le cas d'un message correspondant aux deux blocs $((0,7),(3,9))$ et $((4,7),(7,9))$ sur la figure 5.23 (b).

n'est pas rigoureusement le même que celui choisi pour $\hat{m}'_{j,i}$, alors les éléments reçus dans une région peuvent être permutés par rapport aux éléments attendus, ce qui peut poser problème si les codes couplés s'appuient sur cette ordre.

Il existe un cas particulier où cette stratégie peut s'appliquer correctement sans risquer de permuter les éléments, ce que nous appelons *l'ordre local naturel*. Par définition, l'ordre local des éléments sur les régions est dit *naturel* s'il est induit par un ordre global sous-jacent. Notons que cette hypothèse repose sur l'existence d'un ordre global sur les éléments qui soit commun aux objets couplés, mais ne nécessite pas forcément de connaître cet ordre. Dans ce cas, les messages symboliques $\hat{m}_{i,j}$ et $\hat{m}'_{j,i}$ – respectivement calculés par P_i et Q_j – vont désigner la même séquence d'éléments, comme le suggère l'exemple de la figure 5.24. Dans le cas des grilles structurées, cette hypothèse est naturellement respectée si les codes couplés utilisent la même convention de numérotation des éléments : *row major* ou *column major*[6]. La figure 5.23 (c) représente la numérotation des éléments d'une grille 2D selon l'ordre local naturel aux blocs (convention *row major*).

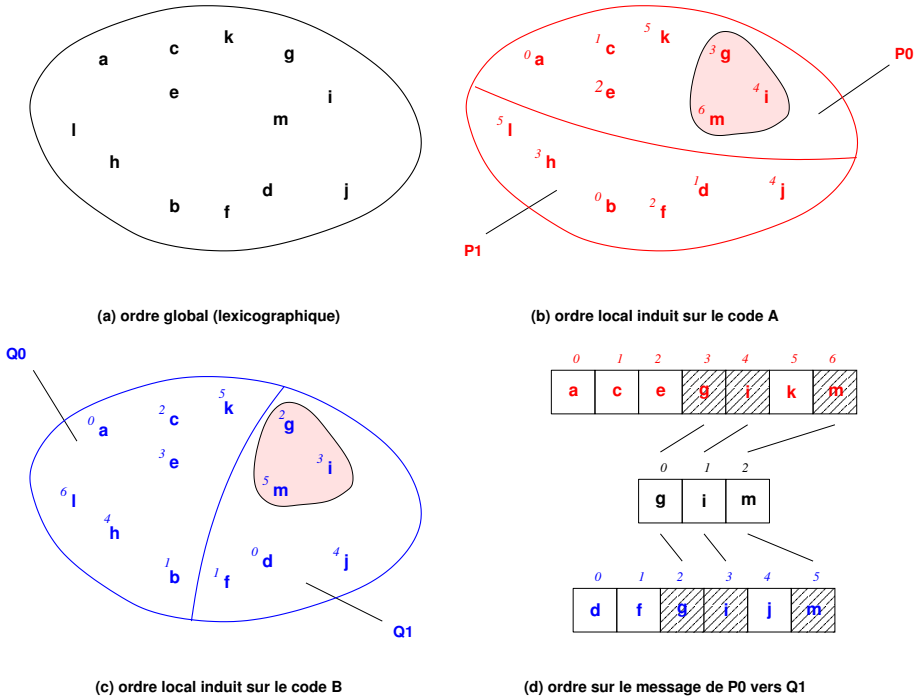


FIG. 5.24 – Construction des messages symboliques $\hat{m}_{0,1}$ et $\hat{m}'_{1,0}$ en se basant sur l'ordre local naturel. Ces deux messages symboliques désignent la même séquence d'éléments (g, i, m), respectivement sur P_0 et Q_1.

Par ailleurs, il est possible d'utiliser la stratégie de l'ordre local pour envoyer un message d'un code A vers un code B, si pour ce dernier l'ordre des éléments qu'il souhaite recevoir n'est pas contraint (*ordre non contraint*). Sous cette hypothèse, l'ordre des éléments défini par les masques de B peut être choisi arbitrairement pour former des messages canoniques. En effet, les éléments reçus par B peuvent être placés librement dans la région de destination, le plus simple et le plus efficace étant de respecter l'ordre des éléments dans le message. Dans notre modèle, nous appliquons essentiellement cette stratégie à deux classes d'objet complexe : les boîtes d'atomes et les ensembles de particules. En effet, il existe de nombreux codes de simulations numériques manipulant ces objets pour lesquels l'ordre de stockage des éléments n'est pas une information critique. Un autre domaine où cette stratégie peut encore s'appliquer est le cadre de la visualisation. En effet, dans les algorithmes de rendu parallèle (cf. Sec. 2.3.2), l'ordre de stockage des éléments dans les régions n'a *a priori* aucune incidence sur l'image calculée. Cette remarque prend toute son importance dans le contexte de cette thèse, car nous cherchons plus précisément à coupler un code de simulation parallèle avec un code de

[6]La convention *row/column major* dépend de l'ordre utilisé sur les dimensions pour parcourir les éléments d'un tableau : en commençant par les colonnes comme en Fortran ou en commençant par les lignes comme en C/C++.

visualisation lui-même parallèle.

En conclusion, nous pouvons souligner qu'il est possible sous certaines hypothèses d'utiliser l'ordre local des éléments dans les régions pour construire efficacement des messages canoniques respectant la continuité des données en mémoire, côté émetteur.

5.5.3 Gestion de la dynamicité des éléments

La gestion de la dynamicité des éléments dans les codes couplés est un problème délicat qui est rarement pris en compte dans les stratégies de redistribution existantes. En général, tout changement de la distribution des éléments en cours d'exécution entraîne un re-calcul complet (ou partiel) de la matrice de communication. Notre idée est de construire un algorithme de redistribution acceptant la dynamicité de ces objets, c'est-à-dire considérant comme « normal » sous certaines hypothèses l'apparition ou la disparition d'éléments dans une région. L'avantage d'une telle approche est qu'il n'est pas utile de recalculer entièrement la matrice de communication à chaque modification de la distribution comme nous allons le voir.

Par définition, un objet complexe distribué est dit *dynamique* s'il est susceptible de modifier sa distribution des éléments au cours de l'exécution. A l'inverse, il est dit *statique* si la distribution des éléments reste fixée au cours du temps. Les grilles structurées sont un exemple classique d'objet statique, car le nombre d'éléments dans chaque bloc est fixé par ses dimensions. Les boîtes d'atomes et les ensembles de particules sont typiquement des objets dynamiques, car dans les codes manipulant ces objets il arrive fréquemment que les éléments se déplacent (modification de la coordonnée de spatiale) et migrent d'une région à l'autre. Nous considérerons également les maillages non structurés comme des objets dynamiques, même si les codes modifiant la distribution du maillage en cours d'exécution sont relativement rares. Afin de supporter une telle dynamicité, nous avons proposé à la section 5.4.3 de définir des objets dynamiques en considérant les régions comme des *conteneurs* d'éléments. Étant donné une région $R_{i,r}$, le conteneur possède un nombre maximum d'éléments $max_{i,r}$ qui renvoie à l'espace de stockage sous-jacent et un nombre courant d'éléments $m_{i,r}$ susceptible de varier au cours de l'exécution ($0 \leq m_{i,r} < max_{i,r}$). L'évolution de la distribution des objets dynamiques peut être repérée simplement grâce à l'utilisation d'un numéro de version entier, tel que ce numéro reste constant si l'objet ne modifie pas sa distribution et augmente dans le cas contraire. Il faut bien distinguer la *version de la distribution* et la *version de la donnée*. Par exemple, un objet statique à une version de distribution toujours constante, alors que la version des données varie typiquement à chaque itération. Dans notre modèle, nous allons uniquement considérer des changements de distribution portant sur le nombre et l'ordre des éléments dans les régions, mais ne modifiant pas le nombre de régions sur chaque processeur.

Niveaux de dynamicité. Dans notre modèle, nous allons distinguer deux *niveaux de dynamicité* : faible ou fort. Un objet est *dynamique faible* si l'ordre local des éléments dans les régions n'est pas contraint et peut changer au cours de l'exécution. Il est *dynamique fort* si l'ordre et le nombre des éléments dans les régions peut changer au cours de l'exécution. Cela signifie que des éléments au sein d'une région sont susceptibles d'apparaître, de disparaître ou de migrer vers une autre région. Notons que dans ces deux cas, nous supposons toujours que le nombre de régions reste fixé. Nous ne considérons pas des changements de distribution plus complexes qui remettraient en cause le découpage en régions logiques. Par ailleurs, il faut préciser qui est susceptible de modifier la distribution d'un objet. Est-ce le code A sous-jacent à l'objet $\mathcal{O}_A$ ou est-ce le code B couplé à cet objet ? Lorsqu'un objet $\mathcal{O}_A$ autorise le code A à modifier sa distribution au cours de l'exécution, il sera dit *dynamique interne*. Il sera dit *dynamique externe* s'il accepte qu'un tiers programme – disons B – modifie sa distribution. Un objet peut être à la fois dynamique interne et dynamique externe. Notons que les objets présents dans les codes de simulation sont rarement dynamiques externes, à moins qu'ils ne soient prévus pour. Ils sont le plus souvent dynamiques internes (avec différents niveaux de dynamicité possible) et nous les considérerons généralement comme tels. Toutefois, dans certains codes de simulation à base d'atomes ou de particules, pour lesquels les éléments sont « anonymes », les objets peuvent être considérés comme dynamiques externes. Il existe un autre domaine où les objets sont naturellement dynamiques externes, c'est le domaine de la visualisation dans le contexte du pilotage. En effet, pour un code B de visualisation, le nombre d'éléments dans les régions ainsi que l'ordre de ces éléments n'a *a priori* aucune incidence sur l'image générée, et il est

tout à fait possible pour un tiers programme couplé – disons la simulation A – de modifier dynamiquement la distribution des éléments que l'on souhaite visualiser.

La principale difficulté liée à la redistribution des objets dynamiques vient du fait que le principe d'intersection que nous avons énoncé à la section 5.3 ne peut plus s'appliquer aussi simplement que pour des objets statiques, car l'ordre ou le nombre des éléments dans les régions est susceptible d'être modifié au cours de l'exécution. En principe, les algorithmes de redistribution qui sont utilisés pour générer les messages sont entièrement *symétriques* pour A et B. Ce type d'algorithme s'applique essentiellement pour des objets statiques dans le cadre du couplage, et toute modification de la distribution d'un des deux objets devra entraîner un recalcul complet de la matrice de communication, ce qui peut s'avérer relativement coûteux. En principe, un tel algorithme de redistribution est requis si les codes couplés s'appuient sur l'ordre des éléments, comme c'est généralement le cas pour des simulations multi-physiques. Dans ce cas, les messages générés doivent être *canoniques* afin de ne pas modifier la distribution des objets couplés lors des communications (bidirectionnelles). Remarquons que les algorithmes basés sur le principe de linéarisation sont canoniques de fait.

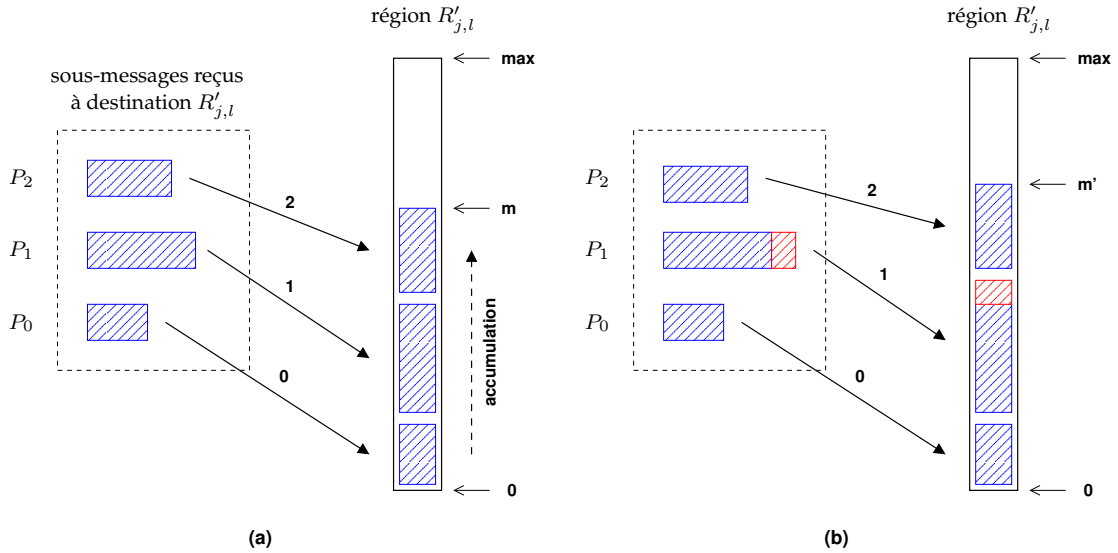


FIG. 5.25 – Mécanisme de réception dynamique de plusieurs sous-messages symboliques $m_{i,j,k}$ en provenance des P_i et à destination de la région $R'_{j,l}$ de Q_j : (a) principe d'accumulation des sous-messages ; (b) apparition dynamique de nouveaux éléments.

Algorithme asymétrique. Afin de supporter la dynamicité interne d'un objet $\mathcal{O}_A$, il est nécessaire de le coupler avec un objet $\mathcal{O}_B$ qui soit *dynamique externe fort*, afin que ce dernier accepte naturellement les changements de distribution relatifs à l'objet $\mathcal{O}_A$. On suppose cependant que $\mathcal{O}_B$ n'est pas lui-même dynamique interne, ce qui signifie qu'il ne modifiera pas sa distribution. Ce type d'algorithme de redistribution est spécialement adapté à la visualisation des objets dynamiques dans *le contexte du pilotage*. Le code A joue alors le rôle de la simulation et le code B celui d'un programme de visualisation parallèle, qui peut modifier les valeurs de $\mathcal{O}_A$ mais pas sa distribution. La prise en compte de ces hypothèses va nécessiter de mettre en place un algorithme de redistribution $\mathcal{R}$ *asymétrique* entre A et B, dont la mise à jour des messages générés ne dépendra que de A. D'un côté, le code A est responsable de générer les messages selon l'algorithme $\mathcal{R}$ et de les mettre à jour dynamiquement à chaque changement de distribution. De l'autre côté, le code B découvrira dynamiquement lors des communications les messages générés par A et mettra à jour sa propre distribution grâce à un mécanisme baptisé *réception dynamique*. Après quoi, le code B peut effectuer des *communications en retour* vers A, sous réserve que la distribution de $\mathcal{O}_A$ n'a pas changé entre temps, ce que nous pouvons contrôler grâce aux numéros de version des distributions. En résumé, les communications de A vers B peuvent survenir librement et modifier la version de la distribution sur B. En revanche, les communications de B vers A ne pourront avoir lieu que si les numéros de version des distributions sont identiques et seront rejetés dans le cas contraire.

Un algorithme asymétrique est dit *canonique* si les communications en retour de B vers A ne modifie pas la distribution de $\mathcal{O}_A$.

Réception dynamique. Si l'objet receveur est dynamique externe fort, il est possible d'utiliser un mécanisme de réception dynamique des messages, car pour ces objets l'ordre des éléments dans les régions n'est pas contraint. Ce mécanisme consiste simplement à accumuler dans la région de destination $R'_{j,l}$ les sous-messages en provenance des P_i dans un ordre déterminé, comme par exemple l'ordre croissant des processeurs émetteurs. Pour chaque sous-message, les éléments sont accumulés dans le conteneur en respectant l'ordre local des éléments dans le message (Fig. 5.25 (a)). La figure 5.25 (b) illustre l'apparition dynamique de nouveaux éléments dans $\mathcal{O}_A$ et la conséquence sur l'ordre des éléments accumulés dans la région $R'_{j,l}$. Par ailleurs, ce mécanisme permet de construire dynamiquement à la réception des messages de P_i un ensemble de masques d'extraction $\mathcal{M}'$ qui pourront être utilisés pour les communications en retour vers A. En conservant l'ordre des éléments reçus grâce aux masques, il est alors possible « d'inverser les communications » et ainsi de transmettre des messages canoniques de B vers A.

5.6 Algorithmes de redistribution

5.6.1 Introduction

Considérons deux codes couplés A et B, respectivement distribués sur deux ensembles de processeurs P et Q, de tailles M et N. Soit E un ensemble d'éléments totalement ordonné (ordre global). Soient $\mathcal{O}_A$ et $\mathcal{O}_B$ deux objets complexes respectivement distribués sur P et Q selon σ_P et σ_Q. On note $\mathcal{O}_i$ et $\mathcal{O}'_j$ les objets complexes respectivement distribués sur chaque P_i et chaque Q_j. Nous présentons dans cette section un algorithme parallèle de redistribution, calculant la matrice de communication $\mathbb{M} = [m_{i,j}]$ de P vers Q et réciproquement. Il ne s'agit donc pas de calculer un ordonnancement des communications ni même d'effectuer ces communications, mais uniquement de générer les messages à partir des informations sur les objets complexes définis dans chaque code. Le tableau 5.3 récapitule les principales notations qui seront utilisées dans la suite de cette section.

Notation	Signification
P	ensembles de M processeurs du code A
Q	ensembles de N processeurs du code B
E	ensemble d'éléments partagé entre les codes couplés
$\mathcal{O}_A$	objet complexe distribué sur P selon σ_P
$\mathcal{O}_B$	objet complexe distribué sur Q selon σ_Q
$\mathcal{O}_i$	objet complexe sur P_i
$\mathcal{O}'_j$	objet complexe sur Q_j
$\hat{\mathcal{O}}_i$	descripteur associé à l'objet complexe $\mathcal{O}_i$
$\hat{\mathcal{O}}'_j$	descripteur associé à l'objet complexe $\mathcal{O}'_j$
$S_{i,s}$	série de données de rang s sur P_i de type T_s
$R_{i,r}$	r-ème région du processeur P_i
$R'_{j,l}$	l-ème région du processeur Q_j
$m_{i,j}$	message entre P_i et Q_j (ensemble d'éléments)
$\hat{m}_{i,j}$	message symbolique de P_i vers Q_j
$\hat{m}_{i,j,k}$	k-ème sous-message symbolique $\hat{m}_{i,j}$
$\bar{m}_{i,j,s}$	message physique de P_i vers Q_j pour la série $S_{i,s}$

TAB. 5.3 – Récapitulatifs des notations utilisées.

Dans le cadre du couplage, chaque processeur P_i possède initialement les informations relatives à l'objet local $\mathcal{O}_i$. La résolution du problème de la redistribution nécessite – du moins pour notre approche – que chaque P_i obtienne les informations relatives à tous les objets distants $\mathcal{O}'_j$ et réciproquement (Fig. 5.26 (a)). En

pratique, l'acquisition de ces informations implique une étape de communication bi-directionnelle entre chaque paire de processeurs (P_i, Q_j) en vue d'échanger les *descripteurs* associés aux objets complexes ($2 \times M \times N$ messages au total). Un descripteur d'objet, noté $\hat{\mathcal{O}}_i$, contient des informations concrètes sur l'objet complexe $\mathcal{O}_i$, dépendant de la classe de l'objet et servant principalement à décrire sa distribution (cf. Tab. 5.2). Ce descripteur est privé des informations liés au stockage des séries de données, car ces dernières informations ne sont pas utiles pour le calcul des messages symboliques. Notons que l'échange des descripteurs via le réseau est typiquement sous la responsabilité de la couche de communication. Nous présenterons au chapitre 6 la bibliothèque RedSYM, qui implante les algorithmes de redistribution décrits dans ce chapitre, et la bibliothèque RedCORBA utilisant RedSYM et basée sur la technologie CORBA pour réaliser les échanges de descripteurs et les communications inter-codes. Dans le cadre du pilotage, l'approche est un peu différente. Initialement, seul le code de simulation (code A) possède la description de l'objet complexe $\mathcal{O}_A$. Côté visualisation, le code B n'a *a priori* aucune information sur l'objet $\mathcal{O}_A$ qu'il souhaite visualiser. On dit alors que $\mathcal{O}_B$ est un *objet vierge*. Dans ce cas, la phase d'initialisation de l'algorithme de redistribution s'effectue en trois temps, représentés sur la figure 5.26 (b). Premièrement, la couche de communication doit transmettre la description de $\mathcal{O}_A$ à tous les processeurs distants. A partir de cette information, le code B construit un objet $\mathcal{O}_B$ compatible avec $\mathcal{O}_A$, en choisissant sa propre distribution σ_Q des éléments. Cela revient essentiellement à construire un nouveau découpage des éléments de E en régions et à « allouer la mémoire » pour ces régions. Dans le cas des objets dynamiques, le nombre courant d'éléments dans la région (conteneur) est initialisé à 0. Après quoi, la couche de communication peut transmettre la description de $\mathcal{O}_B$ au code A.

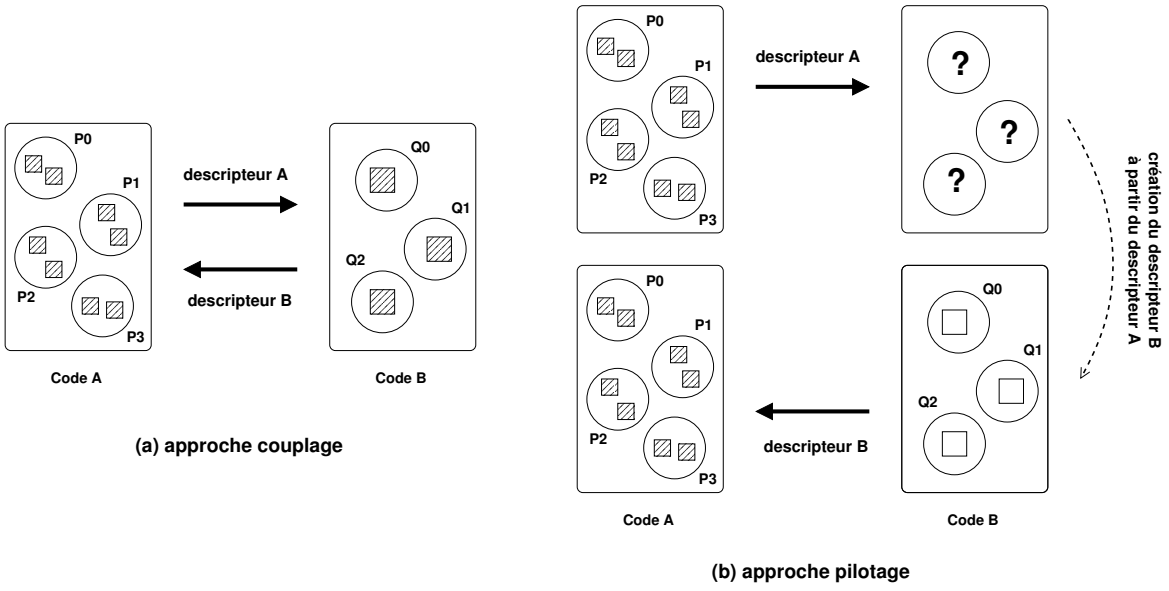


FIG. 5.26 – Initialisation de l'algorithme de redistribution (a) dans le cadre du couplage et (b) dans le cadre du pilotage, montrant l'échange des descripteurs d'objet complexe.

Ainsi, à l'issue de la phase d'initialisation, chaque P_i possède le descripteur de tous les objets distants $\hat{\mathcal{O}}'_j$ ainsi que le descripteur $\hat{\mathcal{O}}_i$ relatif à l'objet local, et réciproquement pour les Q_j. Nous soulignons que dans cet algorithme, les P_i n'ont pas besoin de connaître les descripteurs relatifs aux autres objets locaux. Le problème de redistribution qui se pose est alors symétrique, même si comme nous le verrons il est possible de considérer des algorithmes de redistribution asymétriques permettant de mieux prendre en compte la dynamicité des objets dans le contexte du pilotage. Dans la suite, on se placera du côté du code A (processeurs P_i), qui sera dit *local*, par opposition au code B (processeurs Q_j) qui sera dit *distant*.

L'algorithme que nous proposons est *parallèle*, ce qui signifie que chaque P_i calcule uniquement une ligne de la matrice de communication $\mathbb{M}$, regroupant l'ensemble des messages $\{m_{i,j} \mid 0 \leq j < N\}$ qu'il doit transmettre

Algorithme 5.1
1 Pour chaque processeur P_i faire en parallèle
2 Pour chaque processeur distant Q_j faire
3 **% calcul du message symbolique**
4 $\hat{m}_{i,j} \leftarrow$ *redistribute*$(\hat{\mathcal{O}}_i, \hat{\mathcal{O}}'_j)$
5 **% sérialisation du message symbolique**
6 Pour toutes les séries de rang s faire
7 $\bar{m}_{i,j,s} \leftarrow$ *serialize*$(\hat{m}_{i,j}, S_{i,s})$
8 Fin pour
9 Fin pour
10 Fin pour

FIG. 5.27 – Algorithme parallèle pour la génération des messages sur A.

à tous les Q_j. Par conséquent, il n'est en aucun cas nécessaire de centraliser la matrice de communication, qui est en fait distribuée. Par ailleurs, notre algorithme est dit *symbolique* car il est indépendant d'une couche de communication donnée. Afin d'offrir le maximum de flexibilité vis-à-vis de la couche de communication, nous utilisons la représentation intermédiaire des messages, appelée *message symbolique*, notée $\hat{m}_{i,j}$ (cf. Sec. 5.5). Cet algorithme se décompose en deux étapes principales, décrites sur la figure 5.27 : (1) le calcul des messages symboliques selon une opération abstraite *redistribute* et (2) la génération des messages physiques à partir des messages symboliques grâce à l'opération *serialize*.

La définition concrète de l'opération *redistribute* va dépendre d'une part de la classe de l'objet considérée et d'autre part de l'approche choisie pour redistribuer les objets. Comme nous l'avons vu, un message symbolique $\hat{m}_{i,j}$ est structuré en sous-messages logiques $\hat{m}_{i,j,k}$ construits à partir de masques d'extraction sur les régions. Ce masque désigne les éléments d'une région locale qui seront échangés avec une région distante. Ainsi posé, le problème de la redistribution va principalement consister à calculer des masques d'extraction. Dans notre modèle, le message symbolique remplit deux fonctions primordiales : il permet d'une part de désigner l'ensemble d'éléments échangés entre chaque paire de processeurs, et d'autre part de définir un ordre sur les éléments du message, qui sera utilisé pour l'envoi et la réception des données. Nous avons discuté à la section 5.5.2 du bon choix de cet ordre sous différentes hypothèses. D'une manière générale, notre stratégie consiste simplement à utiliser *l'ordre local* des éléments sur les régions, car elle favorise la continuité en mémoire des données à envoyer ou à recevoir et joue donc favorablement sur les performances des transferts.

La génération des messages physiques à partir des messages symboliques est typiquement sous la responsabilité de la couche de communication, qui réalise une opération de sérialisation, notée *serialize* (Déf. 5.5.5). Cette opération nécessite d'accéder à l'espace de stockage pour chaque série de données $S_{i,s}$.

Nous allons maintenant présenter les deux approches que nous avons définies pour la redistribution des objets complexes : *l'approche spatiale* qui s'applique aux grilles structurées et aux boîtes d'atomes, et *l'approche placement* qui s'applique essentiellement aux ensembles de particules et aux maillages non structurés.

5.6.2 Approche spatiale de la redistribution

Dans cette section, nous allons présenter un algorithme symétrique pour la redistribution des objets spatiaux distribués par blocs, c'est-à-dire les grilles structurées et les boîtes d'atomes. Cette approche se base sur la définition d'un critère d'intersection « spatial » entre les blocs pour construire les messages à échanger entre les codes couplés.

Considérons un objet spatial distribué par bloc sur P_i et sur Q_j. Chaque processeur possède localement une liste de blocs, notés $B_i = (B_{i,0}, B_{i,1}, \ldots)$ pour P_i et $B'_j = (B'_{j,0}, B'_{j,1}, \ldots)$ pour Q_j. Comme nous l'avons vu précédemment, ces ensembles de blocs sont tout à fait quelconques et servent à décrire la distribution des éléments dans l'espace. Dans le cas des grilles structurées, ces blocs sont définis dans $\mathbb{Z}^n$, alors que pour les

boîtes d'atomes, ils sont définis dans $\mathbb{R}^n$. Dans ces deux cas, les blocs vont jouer le rôle de conteneur pour les éléments qui possèdent un coordonnée à l'intérieur du bloc. Afin d'illustrer notre propos, nous allons étudier dans cette section l'exemple de la figure 5.28. Considérons un domaine D dans un espace à 2 dimensions. Dans cet exemple, le processeur local P_i possède deux blocs $B_{i,0}$ et $B_{i,1}$ correspondant aux régions d'éléments $R_{i,0}$ et $R_{i,1}$. Le processeur distant Q_j possède également deux blocs $B'_{j,0}$ et $B'_{j,1}$ correspondant aux régions $R'_{j,0}$ et $R'_{j,1}$.

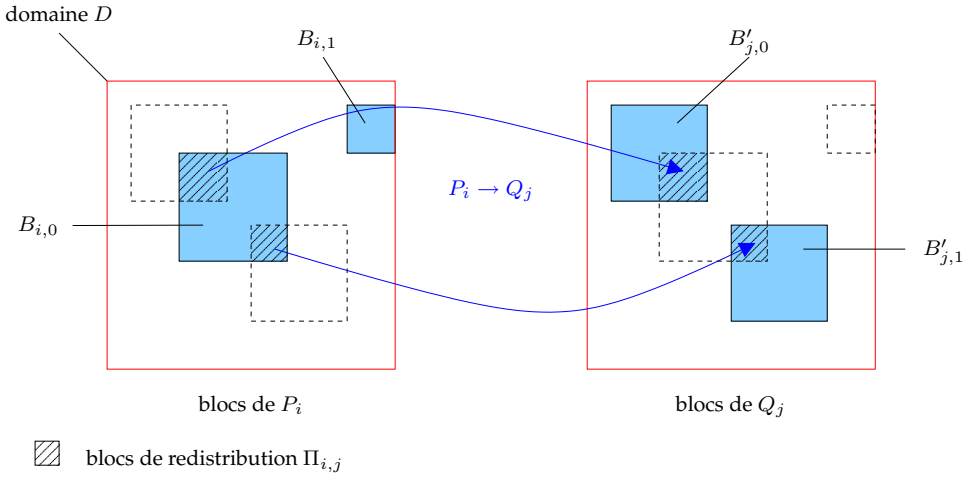


FIG. 5.28 – Approche spatiale : intersection géométrique des blocs de P_i et Q_j.

L'algorithme que nous proposons pour la génération des messages symboliques (Fig. 5.29) se décompose en trois étapes : premièrement, le calcul des blocs de redistribution $\Pi_{i,j,k}$ par intersection géométrique des blocs locaux $B_{i,r}$ et distants $B'_{j,l}$ (ligne 4) ; puis le calcul du masque d'extraction $\mathcal{M}$ relatif au bloc de redistribution $\Pi_{i,j,k}$ grâce à l'opération *find* (ligne 6) ; et finalement la réorganisation des sous-messages symboliques dans un ordre canonique grâce à l'opération *sort* (ligne 12). En accord avec le principe d'intersection formulé en 5.3.5, les éléments appartenant au message $m_{i,j}$ se trouvent à *l'intersection géométrique* des blocs de P_i et de Q_j dans l'espace $\mathbb{Z}^n$ ou $\mathbb{R}^n$ (Fig. 5.28). L'intersection entre le bloc local $B_{i,r}$ (en trait plein) et le bloc distant $B'_{j,l}$ (en pointillé) forme ce qu'on appelle un *bloc de redistribution*, noté $\Pi_{i,j,k} = B_{i,r} \cap B'_{j,l}$. On note $\Pi_{i,j}$ l'ensemble des blocs de redistribution non vides ($|\Pi_{i,j}| \leq |B_i| \times |B'_j|$). A chaque bloc de redistribution $\Pi_{i,j,k}$ non vide correspond un sous-message symbolique $\hat{m}_{i,j,k} = (r, l, \mathcal{M})$, dont il faut déterminer le masque $\mathcal{M}$. Ce masque désigne les éléments de la région $R_{i,r}$ contenus à l'intérieur du bloc $\Pi_{i,j,k} \subset B_{i,r}$.

Algorithme 5.2 (*redistribute*$(\hat{\mathcal{O}}_i, \hat{\mathcal{O}}'_j)$)
1 $k \leftarrow 0$
2 Pour chaque bloc local $B_{i,r}$ de $\hat{\mathcal{O}}_i$ faire
3 Pour chaque bloc distant $B'_{j,l}$ de $\hat{\mathcal{O}}'_j$ faire
4 $\Pi_{i,j,k} \leftarrow B_{i,r} \cap B'_{j,l}$
5 Si $\Pi_{i,j,k} \neq \emptyset$ alors
6 $\mathcal{M} \leftarrow$ *find*$(\Pi_{i,j,k}, R_{i,r})$
7 $\hat{m}_{i,j,k} \leftarrow (r, l, \mathcal{M})$
8 $k++$
9 Fin si
10 Fin pour
11 Fin pour
12 $\hat{m}_{i,j} \leftarrow$ *sort*$(\hat{m}_{i,j})$

FIG. 5.29 – Approche spatiale : génération du message symbolique $\hat{m}_{i,j}$.

La recherche des éléments du bloc de redistribution $\Pi_{i,j,k}$ repose sur une opération abstraite *find*, dont

le rôle est de calculer le masque d'extraction $\mathcal{M}$. Ce masque définit un ordre sur les éléments au sein d'un sous-message, qui se base dans notre approche sur l'ordre local (naturel) de la région source, ce qui revient simplement à construire un masque strictement croissant (cf. Sec. 5.5.2). L'opération *find* est dite abstraite car sa définition va dépendre de la classe de l'objet considéré. Toutefois, il existe deux cas limites que l'on peut résoudre immédiatement : (cas 1) si l'intersection entre les blocs est nulle ($B_{i,r} \cap B'_{j,l} = \emptyset$) alors le masque est vide ; (cas 2) si les deux blocs sont égaux ($B_{i,r} = B'_{j,l}$), alors le masque est plein . Dans le premier cas, cela veut dire qu'il n'y a aucun élément à échanger entre les régions $R_{i,r}$ et $R'_{j,l}$; dans le second cas, cela signifie que tous les éléments de la région $R_{i,r}$ doivent être envoyés à la région $R'_{j,l}$ et réciproquement. Dans les autres cas, l'intersection entre les blocs est dite *partielle* et le calcul du masque va dépendre de la classe de l'objet, grille structurée ou boîte d'atomes.

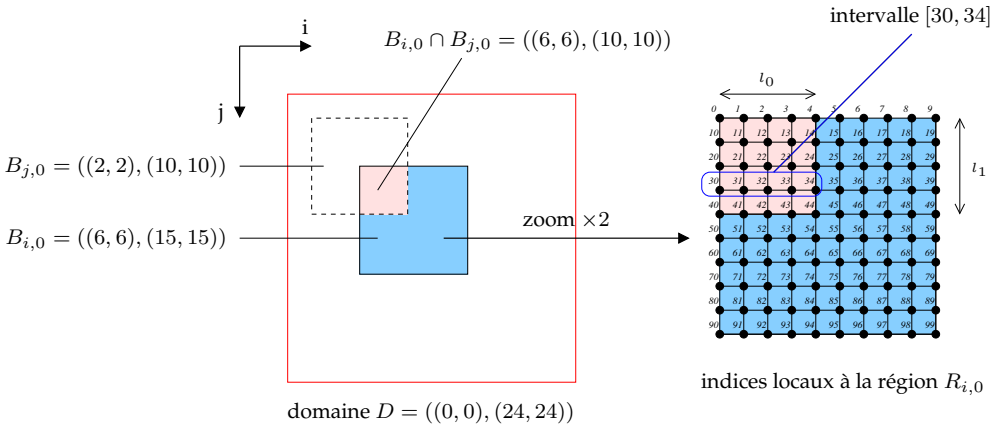


FIG. 5.30 – Approche spatiale : calcul du masque d'extraction associé au bloc de redistribution $B_{i,0} \cap B_{j,0}$ dans le cas des grilles structurées.

Dans le cas des grilles, la construction du masque dépend uniquement de la position relative du bloc de redistribution $\Pi_{i,j,k} = B_{i,r} \cap B'_{j,l}$ par rapport au bloc local $B_{i,r}$ qui le contient (le *bloc source*). Soit $(l_0, l_1, \ldots, l_{n-1})$ les dimensions du bloc de redistribution. Au pire, le masque va donc contenir $l_1 \times \ldots \times l_{n-1}$ intervalles de taille l_0, un nombre qui est d'un ordre de grandeur inférieur au nombre total d'éléments dans la région $(l_0 \times l_1 \times \ldots \times l_{n-1})$. Au mieux, le masque contiendra un seul intervalle de taille $l_0 \times l_1 \times \ldots \times l_{n-1}$ (masque plein). Dans l'exemple de la figure 5.30, le masque permettant d'extraire le bloc de redistribution $B_{i,0} \cap B'_{j,0} = ((6,6),(10,10))$ est formé par un ensemble de $l_1 = 5$ intervalles $([0,4],[10,14],[20,24],[30,34],[40,44])$ de taille $l_0 = 5$.

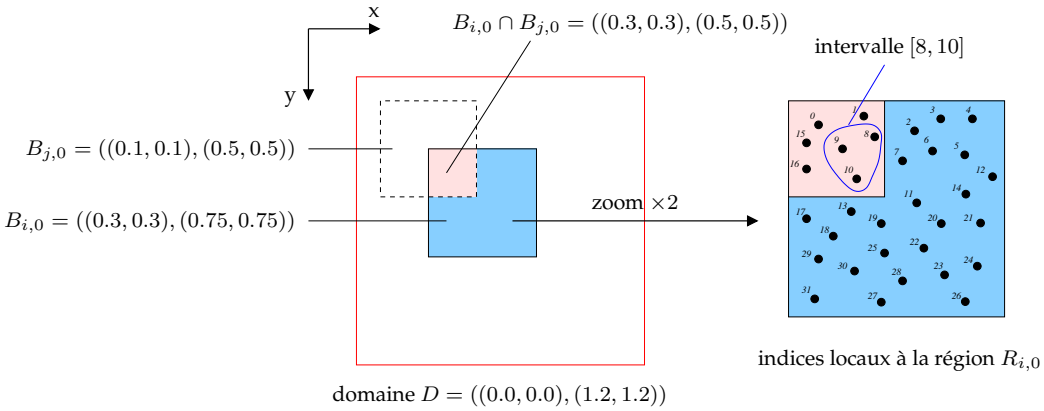


FIG. 5.31 – Approche spatiale : calcul du masque d'extraction associé au bloc de redistribution $B_{i,0} \cap B_{j,0}$ dans le cas des boîtes d'atomes.

Dans le cas des boîtes d'atomes, la construction du masque pour des blocs de redistribution résultant d'une intersection partielle est plus complexe. Elle nécessite de parcourir tous les atomes e de la région locale $R_{i,r}$

et de tester une à une la coordonnée de ces atomes pour déterminer lesquels se trouvent à l'intérieur du bloc de redistribution : $\varphi(e) \in \Pi_{i,j,k}$. Dans l'exemple de la figure 5.31, le masque associé au bloc de redistribution $B_{i,0} \cap B'_{j,0}$ est formé d'un ensemble de 3 intervalles $\{[0,1],[8,10],[15,16]\}$. En respectant l'ordre local de la région lors du parcours des atomes, le masque obtenu sera strictement croissant. Contrairement aux grilles structurées, il n'est pas possible de déterminer analytiquement le nombre d'intervalles dans le masque, ni la taille de ces intervalles. On suppose seulement que la taille des intervalles d'éléments continus ne sera pas trop petite si la position des atomes dans la boîte est plus ou moins corrélée à leurs indices dans la région. Au pire, le nombre d'intervalles est égal au nombre d'atomes dans le bloc de redistribution (i.e. intervalles de taille 1). Notons que lorsque les distributions considérées sont à grain suffisamment fin (cas d'un pavage de l'espace), les intersections partielles entre blocs locaux et distants sont plus rares. Dans ce cas, le calcul des masques devient trivial, dans le sens où il suffit de transférer des boîtes d'atomes entières, sans qu'il soit nécessaire de trier les atomes à l'intérieur d'une boîte (masque plein formé d'un seul intervalle).

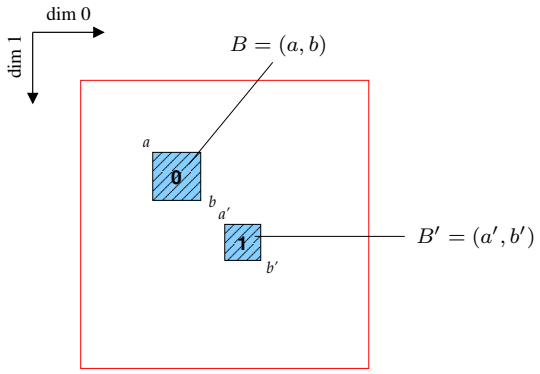


FIG. 5.32 – Opération *sort* : tri des blocs de redistributions B et B'.

Afin d'obtenir un message canonique, il faut encore ordonner les sous-messages symboliques entre eux dans un ordre identique pour P_i et Q_j. C'est précisément le rôle de l'opération *sort* dans notre algorithme. Ce tri des sous-messages peut s'effectuer très simplement si l'on dispose d'un relation d'ordre sur les blocs de redistribution associés à chaque sous-message. La définition d'une relation d'ordre sur les blocs de $\mathbb{Z}^n$ (resp. $\mathbb{R}^n$) peut s'effectuer simplement en comparant la position des points extrêmes relatifs aux blocs [7]. Considérons par exemple deux blocs $B = (a, b)$ et $B' = (a', b')$. Par définition, on a $B < B'$ si et seulement si : $a < a'$ ou si $a = a'$ alors $b < b'$. La figure 5.32 illustre ce propos : la coordonnée de a sur la dimension d'indice 1 (dimension de poids fort) est strictement inférieure à celle de b. Par conséquent, le point a est strictement inférieur à b, et donc le bloc B est strictement inférieur au bloc B'. Ainsi, il est possible de trier les blocs de redistribution dans un ordre canonique, c'est-à-dire identique pour P_i et Q_j et donc de constituer un message symbolique bien ordonné, prêt pour la sérialisation.

La taille des descripteurs échangés entre chaque paire (P_i, Q_j) dans la phase d'initialisation dépend du nombre de blocs locaux détenus par chaque processeur, ainsi que du nombre de dimensions n utilisé pour décrire chaque bloc (i.e. taille en $\mathcal{O}(2.n)$ pour chaque bloc). Ainsi, P_i envoie à chaque Q_j au total $|B_i|$ blocs, et il s'attend à recevoir $|B'_j|$ blocs de la part de chaque Q_j. En totalité, P_i doit posséder dans sa mémoire un descripteur local de taille en $\mathcal{O}(|B_i| \times 2.n)$ et un descripteur parallèle distant de taille $\mathcal{O}(\sum_{j=0}^{N} |B'_j| \times 2.n)$. Du point de vue de la complexité algorithmique, l'étape prépondérante est le calcul des masques par l'opération *find*. Le nombre de masques qu'il faut effectivement calculer est égal au nombre de blocs de redistribution non vides $|\Pi_{i,j}|$. Ce nombre peut être majoré pour chaque P_i par le produit du nombre de blocs locaux et du nombre total de blocs distants, c'est-à-dire par $|B_i| \times \sum_{j=0}^{N} |B'_j|$. La complexité de cette opération dépend de la classe de l'objet considéré. Dans la cas des grilles structurées, ce calcul est simplement linéaire par rapport au nombre d'intervalles dans le masque, un nombre qui au pire est d'un ordre de grandeur inférieur au nombre d'éléments dans la région (i.e. complexité au pire sous-linéaire par rapport au nombre d'éléments). Dans le

[7] On considère la relation d'ordre suivante sur les points de l'espace $\mathbb{Z}^n$ (resp. $\mathbb{R}^n$). Soient $p = (p_0, \ldots, p_{n-1})$ et $p' = (p'_0, \ldots, p'_{n-1})$ deux points de $\mathbb{Z}^n$ (resp. $\mathbb{R}^n$). Par définition, on a $p < p'$ si et seulement si il existe une dimension $c < n$ telle que $p_c < p'_c$ et $\forall i < c,\ p_i = p'_i$.

cas des boîtes d'atomes, ce calcul a une complexité au pire linéaire par rapport au nombre d'éléments dans la région. En pratique, cette complexité sera d'autant plus petite que les distributions des deux codes seront proches (présence de blocs identiques entre ces codes).

En conclusion, nous soulignons que cet algorithme est symétrique canonique, si les régions locales et distantes respectent l'ordre local naturel et si les objets couplés sont statiques. Notons que ces hypothèses sont trivialement vérifiés pour les grilles structurées. Dans le cas des boîtes d'atomes, qui sont généralement des objets dynamiques internes forts, il est possible – dans le contexte du pilotage – d'utiliser une variante asymétrique canonique de l'algorithme permettant de prendre en compte la dynamicité de l'objet $\mathcal{O}_A$ grâce au mécanisme de réception dynamique présenté à la section 5.5.3.

5.6.3 Approche placement de la redistribution

Dans cette section, nous présentons une nouvelle approche de la redistribution, *l'approche placement*, s'appliquant plus spécifiquement dans le contexte du pilotage (Fig. 5.26(b)). Dans cette section, on supposera que le code A joue le rôle d'une simulation parallèle sur M processeurs et B celui d'un programme de visualisation parallèle sur N processeurs ($M \gg N$). Comme nous l'avons vu précédemment, l'objet $\mathcal{O}_B$ est initialement vierge de toute distribution. Il faut alors choisir cette distribution à l'initialisation du couplage, à partir des informations relatives à l'objet $\mathcal{O}_A$. On peut formuler le problème de la redistribution comme un problème plus simple s'apparentant à un *problème de placement* des éléments d'un code vers l'autre. Dans ce cas, la distribution des éléments peut être choisie « au mieux » pour faciliter le couplage et minimiser le coût des communications inter-codes. Nous avons choisi d'appliquer cette approche à des objets complexes non structurés, comme les ensembles de particules ou les maillages non structurés, pouvant être plus ou moins dynamiques selon les codes de simulation. Côté visualisation (code B), nous allons considérer exclusivement des objets *dynamiques externes forts*, afin de tolérer la dynamicité interne du code de simulation couplé. Nous proposons deux variantes de cette approche : l'approche par *placement des régions* (sans découpage) et l'approche *fusion/découpage*.

Placement des régions sans découpage

Le placement des régions sans découpage (*no split & merge*) est une stratégie de redistribution très simple, qui peut s'appliquer de manière très générique à toute classe d'objet complexe distribué par région. Nous avons choisi d'appliquer cette stratégie plus particulièrement pour des ensembles de particules et des maillages non structurés, même si elle peut également s'appliquer aux grilles et aux boîtes d'atomes dont les blocs jouent le rôle des régions. Le principe du placement sans découpage consiste à mettre explicitement en relation les régions de $\mathcal{O}_i$ avec les régions de $\mathcal{O}'_j$, à l'aide d'un *tag* numérique unique servant à identifier les régions dans chaque code. On note $tag(R_{i,r})$ le *tag* de la r-ème région du processeur P_i et $tag(R'_{j,l})$ le *tag* de la l-ème région du processeur Q_j. L'algorithme 5.33 détaille les étapes servant à la génération du message symbolique $\hat{m}_{i,j}$. Ainsi, si la région locale $R_{i,r}$ possède un *tag* identique à la région distante $R'_{j,l}$, cela signifie que ces régions vont partager les même ensemble d'éléments. Dans ce cas, tous les éléments de la région $R_{i,r}$ vont appartenir au sous-message $\hat{m}_{i,j,k} = (r, l, \mathcal{M})$ destiné à la région $R'_{j,l}$ avec $\mathcal{M} = ([0, |R_{i,r}| - 1])$ qui est un masque plein respectant l'ordre local (naturel). On peut ainsi résumer cette stratégie en disant que l'on échange des régions entières (une ou plusieurs) entre chaque paire de processeur (P_i, Q_j). La construction d'un message symbolique canonique nécessite cependant d'ordonner les sous-messages symboliques via l'opération *sort*, ce que nous faisons en respectant simplement l'ordre croissant des *tags* numériques sur les régions.

Cet algorithme peut s'appliquer aussi bien dans le cadre du couplage que du pilotage. Dans ce dernier cas, nous supposons que le code B ne possède pas de distribution (objet vierge). Le placement des régions du code A sur le code B peut être fixé explicitement par l'utilisateur à partir de la description de $\mathcal{O}_A$ ou être calculé automatiquement par *une fonction de placement*. Entre autres informations relatives à la classe de l'objet complexe considéré, les descripteurs contiennent la valeur des *tags* associés à chaque région ainsi que le nombre maximum d'éléments dans chaque région (conteneur). La fonction de placement que nous utilisons répartit l'ensemble des régions de $\mathcal{O}_A$ distribuées sur les M processeurs P_i vers les N processeurs Q_j. Afin d'équilibrer la charge « au mieux », il est possible de prendre en compte une fonction de coût spécifique au code B. Dans le contexte du pilotage, nous considérons en première approximation une fonction de coût linéaire par rapport

Algorithme 5.3 (*redistribute*$(\hat{\mathcal{O}}_i, \hat{\mathcal{O}}'_j)$)

```
1  k ← 0
2  Pour chaque région locale R_{i,r} de Ô_i faire
3      Pour chaque région distante R'_{j,l} de Ô'_j faire
4          Si tag(R_{i,r}) = tag(R'_{j,l}) alors
5              M ← ([0, |R_{i,r}| − 1])
6              m̂_{i,j,k} ← (r, l, M)
7              k + +
8          Fin si
9      Fin pour
10 Fin pour
11 m̂_{i,j} ← sort(m̂_{i,j})
```

FIG. 5.33 – Approche placement des régions : génération du message symbolique $\hat{m}_{i,j}$.

au nombre d'éléments, qui est indépendante de la position des éléments dans l'espace. En effet, la complexité d'une image générée par un code de visualisation (B) dépend directement du nombre de polygones à traiter, qui est proportionnel au nombre d'éléments : un point ou une sphère pour une particule, un polygone pour une cellule de maillage. Cela n'est plus tout à fait vrai si la caméra observe une sous-région de l'espace (*clipping*) ou si nous effectuons des pré-traitements sur les données à visualiser (filtres). Par ailleurs, nous allons supposer que le nombre d'éléments dans les régions de $\mathcal{O}_A$ est globalement équilibré, ce qui est généralement le cas pour la plupart des codes de simulations numériques. Sous ces hypothèses, il est possible de définir une fonction de placement très simple ne dépendant que de M, de N et du nombre de régions $|R_i|$ associé à chaque P_i.

Soient $|R|$ le nombre total de régions et z le reste de la division entière de $|R|$ par M. Le nombre de régions $|R'_j|$ affectés à Q_j est $\lceil |R|/M \rceil$ si $j < z$ et $\lfloor |R|/M \rfloor$ sinon. Si $|R|$ est divisible par M, tous les Q_j possèdent exactement le même nombre de régions, sinon il diffère au pire de 1. Dans l'exemple de la figure 5.34 (a), le code A est distribué sur $M = 4$ processeurs avec 2 régions par processeur, et B est distribué sur $N = 3$ processeurs. Dans ce cas, nous choisissons de placer les $|R| = 8$ régions de la façon suivante : 3 régions sur Q_0 , 3 régions sur Q_1 et 2 régions sur Q_2 (Fig. 5.34 (b)).

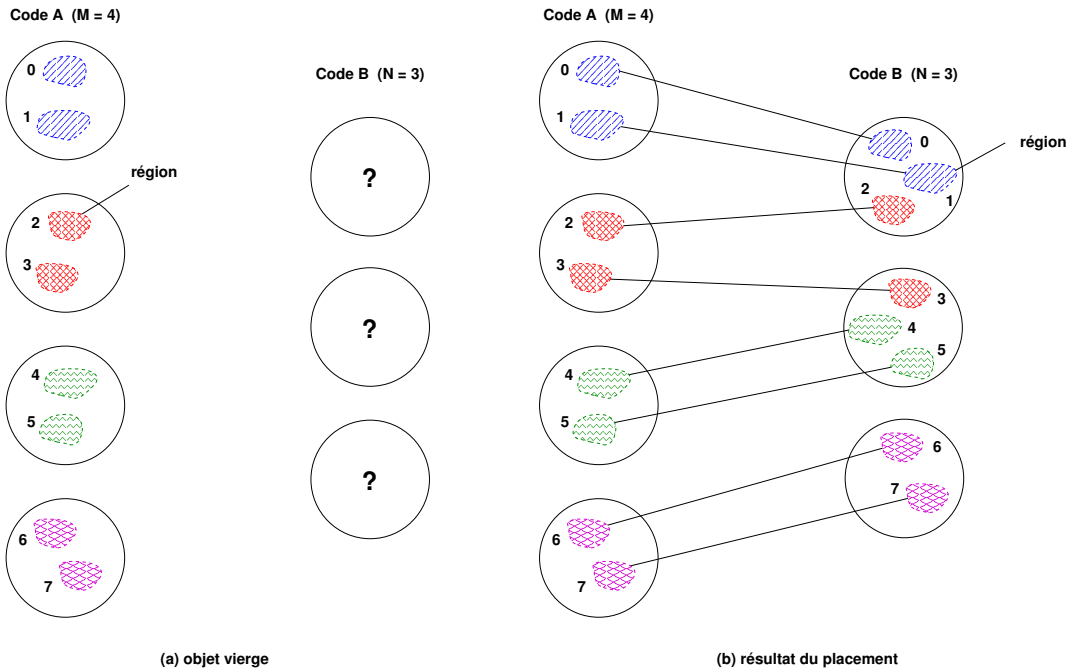


FIG. 5.34 – Exemple de placement de A vers B sans découpage des régions.

En conclusion, il faut souligner que cette stratégie est très flexible et peut s'appliquer de manière très générale à tout type d'objet complexe, à partir du moment où l'on dispose d'un ordre global sur les régions (les *tags*). Cette algorithme de redistribution est asymétrique canonique, ce qui signifie qu'il permet de prendre en compte la dynamicité de l'objet $\mathcal{O}_A$ grâce au mécanisme de réception dynamique. Cela s'avère particulièrement important dans le cas des ensembles de particules, mais aussi pour la visualisation des maillages dynamiques ou non. De plus, le transfert des éléments par région entière permet de garantir la continuité des messages échangés et donc d'offrir de bonnes performances. En revanche, cette approche peut conduire à un mauvais équilibrage de la charge si les régions initiales ne sont pas équilibrées ou si elles ne sont pas à grain suffisamment fin par rapport au nombre total d'éléments. Pis, si le nombre de processeurs N du code B est supérieur au nombre total de régions sur A, alors certains processeurs de B n'auront aucune région affectée lors du placement. Notons que dans le contexte du pilotage, ce problème ne se pose pas car nous considérons généralement $|R| \geq M \gg N$.

Placement avec découpage

Comme nous venons de le dire, le placement des régions peut induire un mauvais équilibrage de la charge, notamment lorsque que M et N sont du même ordre de grandeur. Une idée simple pour surmonter les difficultés de la méthode précédente consiste à autoriser le découpage des régions pour ajuster la taille des messages échangés entre A et B. L'algorithme que nous présentons dans cette section s'applique aux ensembles de particules et aux maillages non structurés. Afin de simplifier notre propos, nous allons nous placer dans le cas où le nombre de régions est fixé à 1 pour chaque processeur P_i (région $R_{i,0}$) et chaque processeur Q_j (région $R'_{j,0}$). De plus, nous supposerons que le nombre d'éléments dans chaque région de P_i est identique et égal à m. Notons qu'il est possible de généraliser cet algorithme sans trop de difficultés au cas où les tailles des régions sont différentes.

Cet algorithme repose sur la définition de l'opération *split*, une opération abstraite de découpage d'une région en n parties de poids $(p_0, p_1, \ldots, p_{n-1})$, tels que la somme des poids p_k soit égal à 1. La k-ème partie est simplement définie par un masque d'extraction $\mathcal{M}_k$ sur la région, désignant un ensemble de $p_k.m$ éléments. L'opération *split* est dite abstraite car elle dépend de la classe de l'objet considéré (Fig. 5.35). Dans le cas des ensembles de particules, il est possible de considérer un découpage en n intervalles d'éléments continus (selon l'ordre local). En revanche, le découpage d'un maillage non structuré en n parties de poids différents nécessite d'utiliser un algorithme de partitionnement, comme ceux implantés dans Scotch [173] ou dans Metis [116] (cf. Sec. 2.1.3). Notons que le partitionnement des cellules d'un maillage induit implicitement un partitionnement des nœuds portées par les cellules. Par conséquent, nous manipulerons en plus du masque classique sur les éléments (i.e. les cellules) noté $\mathcal{M}^{cell}$, un masque sur les nœuds noté $\mathcal{M}^{node}$, simplement calculé à partir de $\mathcal{M}^{cell}$.

Le calcul d'un placement des éléments de A vers B se base sur un découpage en $M \times N$ unités logiques de l'objet $\mathcal{O}_A$ et de l'objet distant $\mathcal{O}_B$. Les unités logiques sont simplement une abstraction qui va nous permettre de calculer plus facilement les poids pour l'opération de découpage. Chaque région locale $R_{i,0}$ est décomposée en N unités logiques et chaque région distante $R'_{j,0}$ est décomposée en M unités logiques. On note $u_{i,k}$ la k-ème unité logique de la région locale $R_{i,0}$ $(0 \leq k < N)$ et $u'_{j,k'}$ la k'-ème unité logique de la région distante $R'_{j,0}$ $(0 \leq k' < M)$. Le schéma 5.36 illustre un découpage logique en 12 unités logiques ($M = 4$ et $N = 3$) et leurs placements sur les processeurs distants selon deux stratégies baptisées *full split & merge* et *split & merge*.

Full Split/Merge. Dans la première stratégie, chaque unité locale $u_{i,k}$ est associée à l'unité distante $u'_{j,k'}$ telle que $j = k$ et $k' = i$. Chaque P_i communique alors avec chaque Q_j un message symbolique $\hat{m}_{i,j}$ formé d'un seul sous-message correspondant à l'unité logique $u_{i,j}$. Ainsi, chaque région $R_{i,0}$ est découpée en N parties égales (de poids $1/N$). Le nombre total de messages générés est alors de $M \times N$, chaque message étant de taille m/N (Fig. 5.36 (a)).

Split/Merge. Dans la seconde stratégie, les unités logiques sont numérotées par ordre de processeurs croissants de 0 à $M.N - 1$ et chaque unité locale $u_{i,k}$ d'indice global $i \times N + k$ est associée à l'unité distante $u'_{j,k'}$ de même indice, c'est-à-dire telle que $i \times N + k = j \times M + k'$. Plus précisément, si l'on note

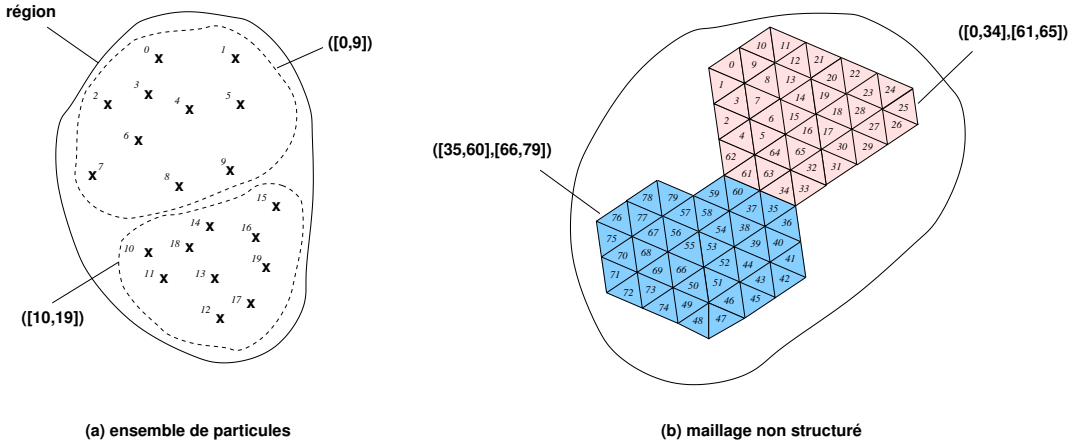


FIG. 5.35 – Découpage d'une région en deux parties égales (poids $1/2$) avec l'opération *split* dans le cas (a) d'un ensemble de particules et (b) d'un maillage non structuré.

$U_i = [i.N, (i+1).N-1]$ l'intervalle des indices globaux relatifs aux unités logiques de P_i et de même pour l'intervalle $U'_j = [j.M, (j+1).M-1]$ sur Q_j, alors le message symbolique $\hat{m}_{i,j}$ est constitué des unités logiques dont l'indice global est à l'intersection de ces deux intervalles : $U_i \cap U'_j$. Lorsque l'intersection est nulle, il n'y a aucun message échangé entre P_i et Q_j. Il est facile de voir que chaque P_i envoie au moins 1 message au processeurs de Q et au plus 2 messages. On peut démontrer que le nombre total de messages échangés est alors de $M + N - \text{PGCD}(M,N)$[8]. Notons que dans le cas particulier où M et N sont des multiples (en supposant $M \geq N$) alors chaque P_i envoie exactement 1 message de N unités logiques consécutives (i.e. pas de découpage) et chaque Q_j en reçoit exactement M/N messages.

Si ces deux stratégies de placement répartissent le nombre total d'éléments de manière équilibrée ($m' = m.M/N$), la seconde stratégie a l'avantage sur la première de générer moins de messages qui sont également plus longs. Dans l'exemple de la figure 5.36 (b), le placement selon la stratégie *split/merge* génère un total de 6 messages pour $M = 4$ et $N = 3$, tels que $U_0 \cap U'_0 = [0,2], U_1 \cap U'_0 = [3,3], U_1 \cap U'_1 = [4,5], U_2 \cap U'_1 = [6,7], U_2 \cap U'_2 = [8,8]$ et $U_3 \cap U'_2 = [9,11]$. Les poids relatifs au découpage des régions des processeurs de P sont alors respectivement $(3/3)$ pour P_0, $(1/3, 2/3)$ pour P_1, $(2/3, 1/3)$ pour P_2 et $(3/3)$ pour P_3.

L'opération *split* permet de calculer effectivement les masques d'extraction relativement à ce découpage, de telle façon que chaque message symbolique est constitué d'un seul sous-message. Les messages que nous construisons respectent simplement l'ordre local des éléments dans la région. Dans le cas des maillages non structurés, certains partitionneurs comme Metis [116] sont uniquement capables de réaliser un découpage en un nombre donné de parties égales. Dans ce cas, nous considérons un découpage en N/q unités logiques avec q le plus grand commun diviseur à M et N et nous construisons des messages symboliques constitués de $p_k.N/q$ sous-messages avec p_k le poids associé à la k-ème partie de la région $R_{i,0}$.

D'une manière générale, la réception des messages pour Q_j se base sur l'opération abstraite *merge*, dont le rôle est de fusionner plusieurs messages différents dans une même région $R'_{j,0}$. Dans le cas des ensembles de particules, cette opération correspond simplement au mécanisme de réception dynamique présenté à la section 5.5.3. Dans le cas des maillages non structurés, nous avons dû adapté ce mécanisme pour prendre en compte la série des connectivités. En effet, les indices des nœuds dans la série des connectivités font référence

[8]Pour établir ce résultat, il faut se ramener au cas où M et N sont premiers entre eux. Si l'on examine « le motif d'intersection » dans ce cas, il est facile de voir que le nombre de messages est exactement de $M + N - 1$. Lorsque M et N ont un multiple commun, il est alors possible de se ramener au cas précédent en posant $M' = M/PGCD(M,N)$ et $N' = N/PGCD(M,N)$. Le motif d'intersection obtenu pour M' et N' se répète autant de fois que le $PGCD$ de M et N. Le nombre de messages total est alors $PGCD(M,N) \times (M' + N' - 1)$. CQFD.

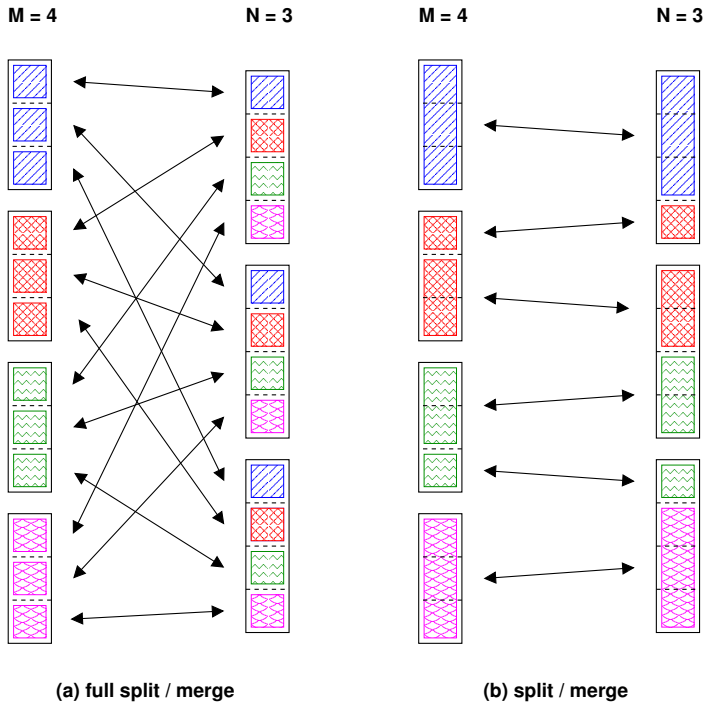


FIG. 5.36 – Stratégie de placement avec découpage (et fusion) des régions.

aux indices locaux dans la région $R_{i,0}$. Par conséquent, ces indices doivent être convertis en une numérotation locale à la région $R'_{j,0}$ au moment de l'accumulation des cellules. Cette conversion s'effectue en deux étapes, une première qui consiste à convertir les indices des nœuds reçus en une numérotation interne au message de rang k s'étendant de 0 à $n_k - 1$, avec n_k le nombre total de nœuds utilisés dans ce message. Notons que cette nouvelle numérotation doit respecter l'ordre croissant des indices de nœuds définis dans la région source. La deuxième étape de la conversion consiste simplement à translater les indices des nœuds de $d = \sum_{k'=0}^{k-1} n_{k'}$ pour prendre en compte l'accumulation des nœuds dans la région $R'_{j,0}$ relative aux $k - 1$ messages précédents. Notons que la seconde étape est également le moment opportun pour effectuer un changement de numérotation C/Fortran entre l'émetteur et le receveur (i.e. numérotation débutant à 0 en C et à 1 en Fortran). A titre d'exemple, on peut considérer deux messages $\hat{m}_{0,j}$ et $\hat{m}_{1,j}$ destinés tous deux à la région $R'_{j,0}$ sur la figure 5.37. Ces messages désignent respectivement les cellules suivantes : $((10, 11, 12), (11, 12, 13), (11, 13, 14))$ et $((6, 10, 20), (5, 6, 20), (5, 9, 20))$. Le premier message sera converti $((0, 1, 2), (1, 2, 3), (1, 3, 4))$ avec n_0 initialisé à 5 ; le second message sera converti en $((1, 3, 4), (0, 1, 4), (0, 2, 4))$ puis translaté en $((6, 8, 9), (5, 6, 9), (5, 7, 9))$ avant d'être accumulé dans la région. Seul la série des connectivités nécessite d'effectuer une telle conversion ; les autres séries (y compris les séries de données associées aux cellules) peuvent être accumulées selon le mécanisme de réception dynamique classique. Notons cependant que cet algorithme ne prend pas en compte les nœuds aux frontières, ce qui signifie que le maillage sur $\mathcal{O}_B$ est divisé en autant de composantes connexes que de messages reçus. Supposons par exemple que les nœuds 13 et 14 de P_0 correspondent aux nœuds 6 et 10 de P_1 sur la figure 5.37. Cela ne pose *a priori* pas de problème pour la visualisation, car le maillage paraîtra visuellement connecté. Pour parvenir à reconnecter correctement les composantes entre elles sur $\mathcal{O}_B$, il faudrait identifier les nœuds aux frontières ainsi que les cellules fantômes, sans quoi ces derniers vont former des doublons dans la région de destination.

Cet algorithme de redistribution est asymétrique canonique dans le cas des ensembles de particules comme des maillages non structurés. Il permet de supporter la dynamicité de l'objet $\mathcal{O}_A$, ce qui s'avère tout à fait pertinent dans le cas des codes particulaires. Cet aspect dynamique présente plus généralement un intérêt pour les codes de visualisation. En effet, il est possible grâce à cette approche de *filtrer dynamiquement* les élements côté simulation (particules ou cellules du maillage) que l'on souhaite visualiser. Cette technique peut être avanta-

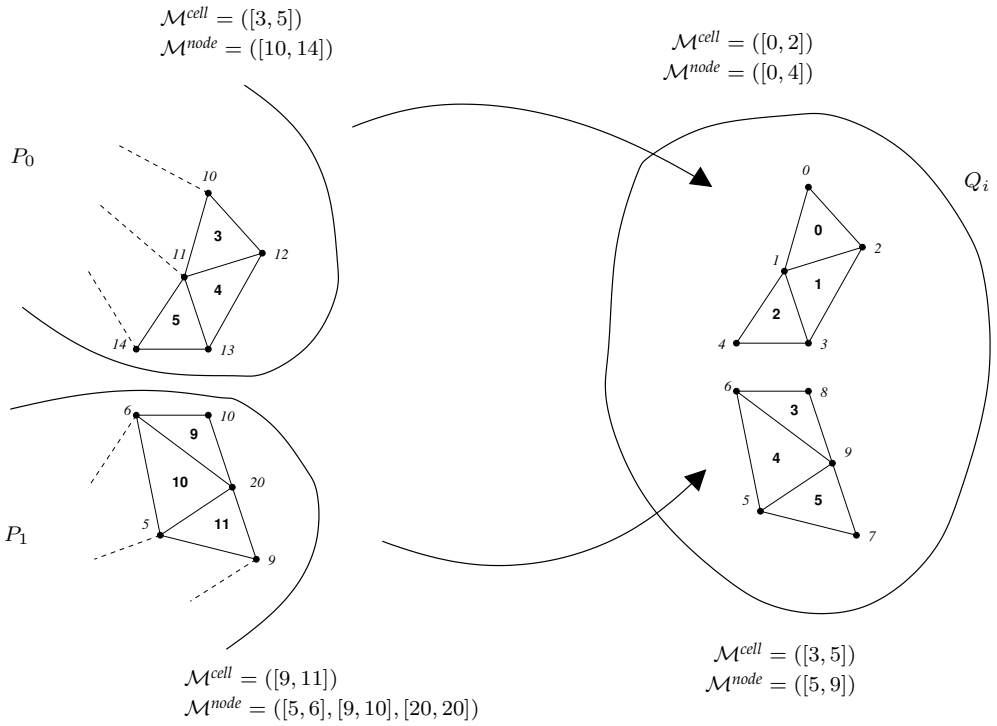


FIG. 5.37 – Opération *merge* dans le cas des maillages non structurés.

geusement exploitée pour isoler un sous-ensemble pertinent des éléments pour la visualisation ou encore pour réduire le volume des communications.

5.7 Conclusion

Dans ce chapitre, nous nous sommes écartés des modèles de redistribution classiques pour nous intéresser à une approche plus générique basée sur la notion d'objet complexe. Ces objets permettent de former une vision cohérente des données telles qu'elles sont distribuées sur chaque code. En outre, il est possible d'enrichir ce modèle pour produire automatiquement une représentation graphique des objets, ce qui nous intéresse plus particulièrement dans le contexte du pilotage comme nous allons le voir dans le chapitre suivant. Pour les deux approches de la redistribution que nous avons proposées, l'approche spatiale et l'approche placement, nous décrivons au chapitre final (chapitre 8) quelques travaux en cours ainsi que plusieurs pistes d'évolution concernant notamment la gestion de structures de données hiérarchiques (grilles multi-niveaux, octree) et la prise en compte des objets dynamiques présents dans les codes AMR (Adaptive Mesh Refinement).

Troisième partie

Réalisation et validation

Chapitre 6

L'environnement EPSN

Sommaire

Dans la partie précédente, nous avons présenté d'une part un modèle pour le pilotage fin des simulations numériques parallèles, et d'autre part des algorithmes pour la redistribution des données selon différentes stratégies adaptées à la problématique du couplage et du pilotage. Dans ce chapitre, nous allons décrire la réalisation de la plate-forme EPSN, un *Environnement pour le Pilotage de Simulations Numériques* parallèles, basé sur la technologie CORBA qui permet d'exploiter les capacités d'un système de visualisation parallèle comme VTK. Nous présenterons des résultats expérimentaux validant nos travaux au chapitre suivant.

6.1 Introduction

EPSN est un environnement logiciel pour le pilotage de simulations numériques parallèles et la visualisation en ligne des résultats intermédiaires de ces simulations. Plus précisément, EPSN permet de piloter des codes de simulations parallèles respectant un modèle de programmation SPMD et utilisant un processus de calcul itératif, comme par exemple les schémas de résolution en temps. Afin d'être aussi générique que possible, nous avons introduit au chapitre 4 un modèle de représentation de ces simulations, basé sur une hiérarchie de tâches (MHT) servant à nous repérer dans l'évolution des calculs et à configurer les interactions possibles. Grâce à un système de dates rattachées aux points d'instrumentation du code, nous avons vu qu'il est possible de suivre précisément l'évolution en temps des calculs et d'opérer efficacement des interactions de pilotage

coordonnées (cf. algorithme de planification). Sur les bases de ce modèle, nous avons pu définir un système de pilotage performant entièrement dirigé par des requêtes clientes et capable d'exploiter les plages d'interaction définies dans le MHT. L'objectif principal de la plate-forme EPSN est de parvenir à piloter efficacement des simulations parallèles en exploitant les capacités des environnements de visualisation immersifs et plus particulièrement des murs d'images (Fig. 6.1). Motivés par cette problématique, nous nous sommes intéressés au chapitre 5 au problème de la redistribution des données entre des codes couplés. Nous avons ainsi proposé un modèle de description des données distribuées s'appuyant sur la notion d'objets complexes. Nous avons distingué plusieurs classes d'objets fréquemment utilisées dans les codes de simulation (grilles structurées, boîtes d'atomes, ensembles de particules, maillages non structurés), pour lesquels nous avons défini différentes stratégies de redistribution adaptées au contexte du pilotage.

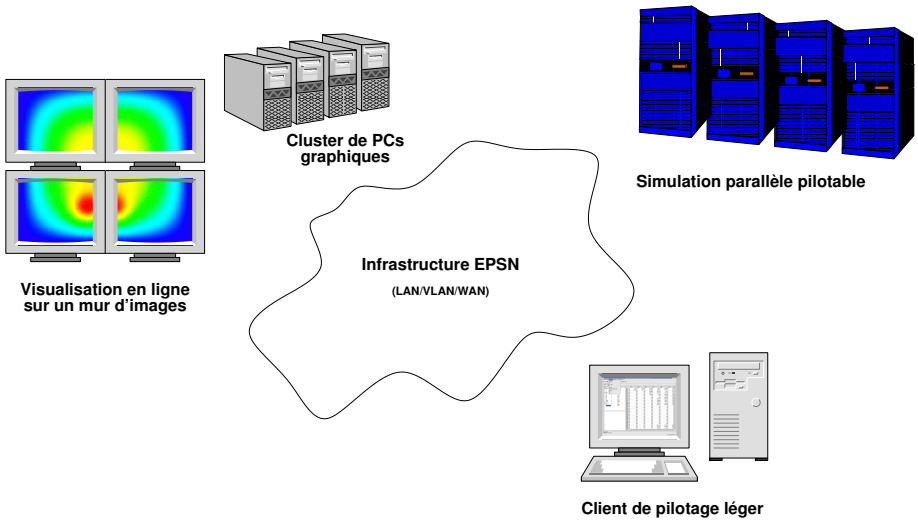


FIG. 6.1 – Vue d'ensemble de la plate-forme EPSN.

Pour parvenir à piloter une simulation avec EPSN, il est nécessaire de respecter quelques étapes, que nous résumons sur la figure 6.2. (1) La première étape consiste à annoter le source avec l'API EPSN *back-end*. Cette étape, baptisée instrumentation, vise à repérer les tâches du MHT modélisant la simulation. Comme nous l'avons vu à la section 4.2.4, l'utilisateur doit compléter son instrumentation avec un fichier de description XML servant essentiellement à décrire le MHT et à configurer les interactions possibles sur les tâches du MHT (plage d'accès aux données, points d'actions, *etc.*). (2) L'étape suivante consiste à recompiler le code source de la simulation pour générer un nouvel exécutable intégrant la plate-forme EPSN, ce que nous appelons la *simulation pilotable*. (3) Cet exécutable peut ensuite être déployé normalement sur un cluster de PCs ou une machine parallèle, c'est-à-dire sans changement apparent pour l'utilisateur (*mpirun*, *batch scheduler*, *etc.*). Une fois déployée, la simulation pilotable s'exécute normalement, mais est prête à recevoir dynamiquement des connexions clientes. (4) Les clients de pilotage sont construits et déployés de manière symétrique à la simulation grâce à l'API EPSN *front-end*. Mais il est également possible d'utiliser un client de pilotage générique comme le programme *Simone* fourni avec l'environnement EPSN (cf. Sec. 6.4). Ce programme permet de se connecter et d'interagir simplement avec toutes simulations instrumentées pour EPSN. (5) Lorsqu'un client de pilotage se connecte à la simulation, alors une session de pilotage débute qui s'achèvera à la déconnexion du client. Durant cette session, le client peut interagir à distance avec la simulation en lui soumettant des requêtes diverses ; il pourra ainsi visualiser en ligne l'évolution de la simulation et éventuellement modifier son comportement en cours d'exécution.

Les principales fonctionnalités offertes par l'environnement EPSN aux utilisateurs pour interagir avec des simulations distantes sont :

- le contrôle à distance du flot d'exécution de la simulation (requêtes *play/step/stop*) ;
- la consultation de la date courante relative au MHT (requête *getDate*) afin de suivre précisément l'évolution de la simulation ;

- l'accès à la volée aux données de la simulation sur des plages d'accès cohérentes configurées dans le MHT (requête *get*) ;
- l'accès périodique aux données produites par la simulation pour la visualisation en ligne des résultats intermédiaires (requête permanente *getp*) ;
- la modification à distance des paramètres et des données de la simulation (requête *put*) ;
- le déclenchement d'actions prédéfinies dans la simulation (requête *action*) ;
- la mesure des performances de la simulation grâce au pilotage de *timers* évaluant le temps passé dans chaque tâche.

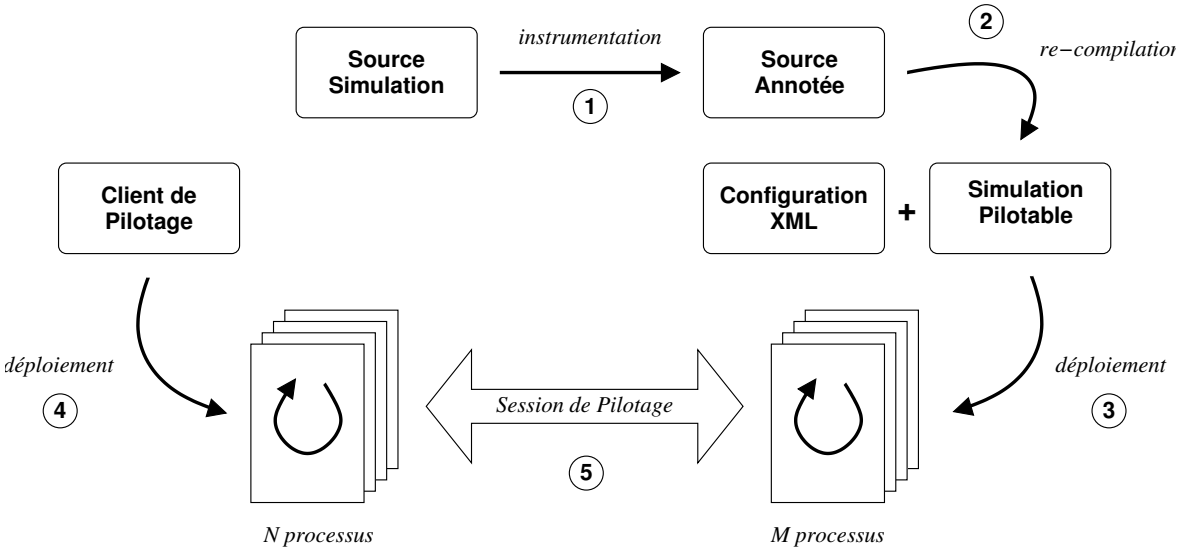


FIG. 6.2 – Le cycle de vie dans EPSN.

L'environnement EPSN est une plate-forme distribuée, portable et interopérable entièrement basée sur la technologie CORBA (cf. Sec. 2.2.4). On peut mettre en avant plusieurs points-clés qui font d'EPSN un environnement de pilotage performant : la faible perturbation des simulations, l'accès efficace et cohérent aux données et l'utilisation de flux parallèles de communication. Nous allons maintenant présenter l'architecture générale de l'environnement EPSN, en détaillant ces points-clés. Puis, nous étudierons à la section 6.3 les différentes couches logicielles qui ont servi à la réalisation d'EPSN, incluant la bibliothèque de redistribution RedSYM. Enfin, nous présenterons comment réaliser des clients de pilotage et de visualisation (pouvant être parallèle) pour EPSN en se basant sur la bibliothèque VTK (The Visualization Toolkit) [33].

6.2 Architecture

6.2.1 Vue d'ensemble

La plate-forme EPSN utilise une représentation des applications parallèles couplées (simulation ou client) en terme de *composant*. Chaque application est vue comme un composant parallèle constitué d'un ensemble de *ports* associés aux différents processus et d'un *proxy* servant de point d'entrée au composant (Fig. 6.3). Le proxy le plus souvent associé au port de rang 0 mais peut également être placé sur un processus à part. EPSN est un environnement distribué basé sur une relation de type *client/serveur* entre les simulations (serveur) et les interfaces de pilotage (client). Notons que cette distinction entre client et serveur est relativement formelle, car elle intervient principalement au moment de la connexion. Au cours d'une session de pilotage, les deux applications couplées joueront simultanément le rôle de client et de serveur. Les clients ne sont pas fortement connectés aux serveurs : ils utilisent un mécanisme de requêtes asynchrones et concurrentes pour interagir à distance avec la simulation. Ces caractéristiques permettent à la fois aux simulations de recevoir concurremment des requêtes de plusieurs clients de pilotage (multi-clients) et à ces clients de se connecter

simultanément à différentes simulations (multi-applications).

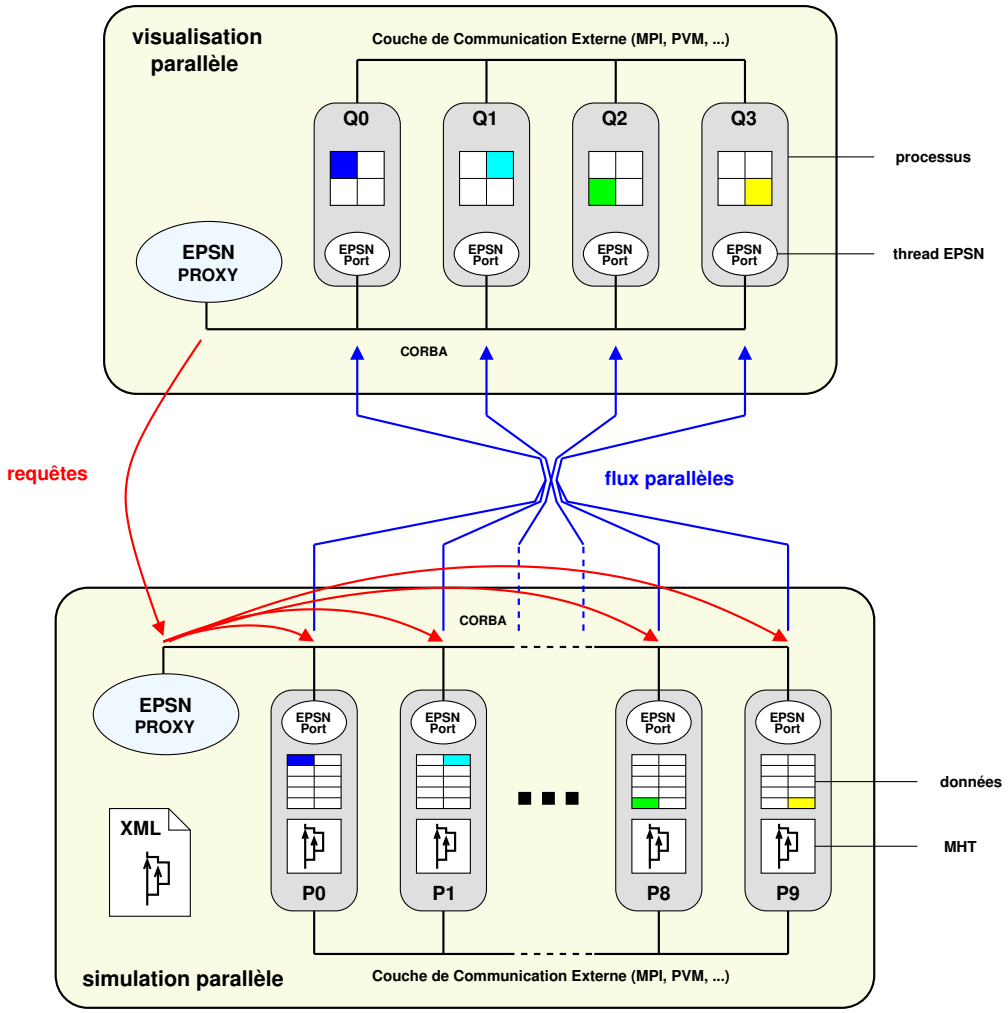


FIG. 6.3 – Architecture de la plate-forme EPSN.

EPSN utilise une couche de communication interne basée sur la technologie CORBA, qui offre l'interopérabilité souhaitée pour coupler des applications réparties sur des machines hétérogènes. Notons que l'utilisation de CORBA dans la plate-forme EPSN est entièrement masquée à l'utilisateur final, qui peut utiliser librement dans son application la couche de communication de son choix (MPI, PVM, *etc.*). Comme le suggère la figure 6.3, l'environnement EPSN est démarré dans un *thread* attaché à chaque processus de la simulation ou du client. Sur chaque processus (port ou proxy), le thread EPSN encapsule un serveur CORBA, en attente de requêtes distantes et en charge de tous les traitements de pilotage. Côté simulation, le *proxy* fournit une vue unifiée de l'application aux clients, regroupant les informations relatives au MHT et à la distribution des données. Il offre un point d'entrée au client qui souhaite se connecter à la simulation et sert de relais pour les requêtes clientes qui sont ensuite transmises aux ports de la simulation. Côté client, le proxy joue le rôle du pilote, en générant des requêtes en réponse aux commandes de l'interface utilisateur (UI) et en les transmettant au proxy de la simulation. Une fois la requête reçue par un port de la simulation, elle est traitée localement. Sur chaque port, le thread EPSN peut accéder aux données de la simulation présentes en mémoire partagée. Ainsi, il peut effectuer des transferts de données vers un client distant. Ce transfert se produit concurremment à l'exécution de la simulation, en respectant les plages d'accès associés aux tâches du code par l'utilisateur. Comme la simulation et le client peuvent tous deux être parallèles, nous utilisons un algorithme de redistribution pour générer les messages à échanger, qui sont envoyés via CORBA de port-à-port (flux parallèles ne nécessitant pas de passer par le proxy). Les ports côté client sont alors utilisés pour traiter les données extraites de la simulation et

produire de la visualisation en parallèle.

La définition d'une architecture performante pour plate-forme EPSN repose principalement sur trois points-clés :

La faible perturbation des simulations. EPSN introduit peu de perturbations dans le déroulement des simulations instrumentées, grâce à l'utilisation d'un système de pilotage entièrement dirigé par le client. Ainsi, lorsque qu'il n'y a pas de requêtes en cours, la simulation n'est pas perturbée et le surcoût est minimal. Par ailleurs, toutes les requêtes de pilotage sont prises en charge côté simulation par un *thread* dédié, le « thread EPSN », embarquant un serveur CORBA. Ce thread est responsable de traiter les requêtes clientes qui sont exécutées concurremment aux calculs de la simulation de manière cohérente grâce à l'algorithme de coordination présenté en 4.4.2.

L'accès efficace et cohérent aux données. La définition de plages d'accès sur les tâches du MHT permet de recouvrir autant que possible le transfert des données entre la simulation et le client. Si la plage d'accès définie dans le code est suffisamment large, alors il est possible de recouvrir quasi totalement le temps de transfert sur les calculs de la simulation. Dans le cas contraire, un mécanisme de synchronisation préserve la cohérence des communications en bloquant la simulation à la fin de la plage d'accès, le temps que les transferts s'achèvent.

Les flux parallèles de communication. EPSN permet de coupler des codes de simulation parallèles avec des codes de visualisation eux-mêmes parallèles grâce à l'utilisation d'algorithme de redistribution, bien adaptés au contexte du pilotage. Ces algorithmes calculent à partir de la description des objets complexes fournie par chaque code l'ensemble des messages nécessaires pour transférer les données d'un code vers l'autre et réciproquement. Ainsi, il n'est pas nécessaire de centraliser les données lors des communications entre les codes couplés, ce qui entraînerait un goulot d'étranglement important. Au contraire, nous utilisons des flux de communications parallèles entre les codes couplés, ce qui permet d'agréger la bande-passante quand le réseau le permet.

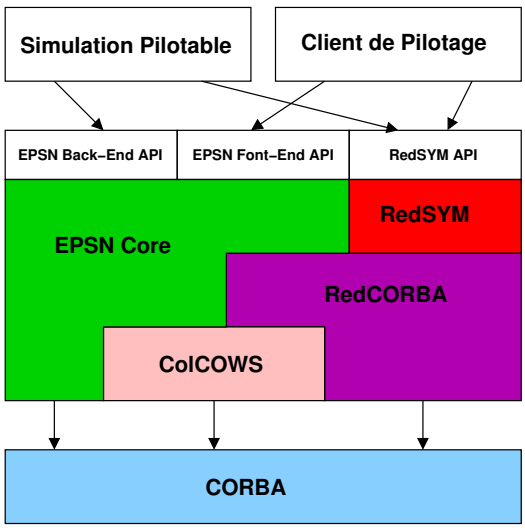


FIG. 6.4 – Les couches logicielles de l'environnement EPSN.

La plate-forme EPSN est découpée en plusieurs couches logicielles : le noyau EPSN Core, et les bibliothèques ColCOWS, RedCORBA et RedSYM (Fig. 6.4). Le noyau EPSN Core implante les deux APIs *back-end* et *front-end* ; elle est responsable du démarrage de la plate-forme dans un thread dédié attaché à chaque processus. Ce thread encapsule un serveur CORBA gérant le système de requêtes d'EPSN coordonnant les traitements de pilotage. La bibliothèque ColCOWS représente la couche de connexion, qui facilite le partage des références d'objets CORBA au sein d'un composant parallèle (activation) et leurs échanges avec un composant distant (connexion). La bibliothèque RedCORBA joue le rôle de la couche de transfert $M \times N$ acheminant les données d'un code parallèle vers l'autre et inversement. Cette bibliothèque utilise la technologique CORBA pour réaliser

les transferts selon un paradigme de communication *one-sided* (get/put). En outre, RedCORBA s'appuie sur la bibliothèque de redistribution RedSYM dont le rôle principal est de générer la matrice de communication à partir de la description des objets complexes fournie par l'utilisateur (via l'API RedSYM).

EPSN a été entièrement développé en C++ sous Linux et possède également une traduction de l'API *back-end* en C et en Fortran 77 (compatible Fortran 90), des langages très répandus dans le domaine du calcul scientifique. La portabilité d'EPSN sous les divers systèmes Unix est assurée par les outils GNU Autotools. EPSN se trouve actuellement à la version 2 de sa génèse. Il hérite d'une version précédente baptisée Epsilon (« petit EPSN »), ayant servi de prototype pour le pilotage de simulations purement séquentielles. Les sources d'EPSN et la documentation sont actuellement disponibles sous le terme de la licence LGPL à l'URL `http://www.labri.fr/epsn`. Nous allons maintenant présenter le fonctionnement de la plate-forme EPSN au travers l'utilisation des deux APIs. Puis nous détaillerons le fonctionnement du système de requêtes dans EPSN Core. Dans la section suivante, nous nous présenterons les différentes couches logicielles de la plate-forme EPSN.

6.2.2 Programmation d'une application de pilotage avec EPSN

Du point de vue utilisateur, EPSN se divise en deux bibliothèques de programmation distinctes : l'API *back-end* servant à annoter le code source de la simulation pour la rendre pilotable et l'API *front-end* permettant aux clients de piloter la simulation à distance (cf. annexes A.1 et A.2). Ces deux APIs sont relativement légères et se présentent sous forme de deux classes C++ `EPSN::SimulationInterface` et `EPSN::ClientInterface`, servant d'interfaces aux ports et au proxy instanciés par la plate-forme EPSN. On peut décomposer les fonctions de ces APIs en quatre groupes logiques servant : (a) à la gestion du cycle de vie de la plate-forme EPSN (côté simulation et côté client) ; (b) à la déclaration des données et des actions (côté simulation et côté client) ; (c) à l'instrumentation du MHT (côté simulation) ; et (d) à la gestion des requêtes lors d'une session de pilotage (côté client). Par ailleurs, EPSN repose sur la bibliothèque RedSYM, dont l'API permet aux utilisateurs de décrire les données distribuées qu'il souhaite piloter, conformément au modèle de description en objet complexe présenté au chapitre 5. Nous présenterons plus précisément la bibliothèque RedSYM à la section 6.3.2.

La programmation d'une application de pilotage EPSN se décompose en plusieurs étapes importantes côté simulation et côté client, dont il faut respecter l'enchaînement. Nous donnons sur les figures 6.5 et 6.6 un exemple type de programmation d'une simulation pilotable et d'un client de pilotage avec les APIs C++ d'EPSN.

Côté simulation

1. La première étape doit être l'initialisation de la plate-forme EPSN dans le code de simulation grâce aux fonctions *initORB()*, *initProxy()* et *initPort()*, dont le rôle est de mettre en place les threads EPSN et les serveurs CORBA sur les différents processus. Cet appel est synchronisant entre tous les processus de la simulation. Le *proxy* est responsable de charger la description du MHT décrite dans le fichier XML ainsi que la configuration des différentes interactions de pilotage relatives aux tâches du MHT.
2. Cette étape doit être complétée par l'ajout des descriptions de données (`RedSYM::Object`) via la méthode *addData()* et l'enregistrement d'éventuels fonctions *callback* pour les actions de pilotage.
3. Une fois ces étapes achevées, l'appel à la fonction *ready()* permet de signaler à la plate-forme EPSN que la simulation est prête à recevoir des connexions clientes et des requêtes de pilotage de la part de ces clients. Cet appel est synchronisant entre tous les processus de la simulation. Il permet d'une part au proxy de diffuser aux ports les informations relatives au MHT, et d'autre part aux ports de fournir au proxy les descripteurs d'objet RedSYM relatifs aux données distribuées.
4. Après quoi, l'utilisateur peut se concentrer sur l'instrumentation à proprement parler de la simulation en annotant dans le code source le début et la fin de chaque tâche (décrites dans le MHT) à l'aide des fonctions *beginMHT()*, *endMHT()*, *beginLoop()*, *endLoop()*, *beginTask()*, *endTask()*, *etc*. Au cours de l'exécution, l'appel à ces fonctions permet de « passer la main » à EPSN pour qu'il effectue les traitements appropriés. Aucune requête de pilotage ne sera traitée en dehors des appels *beginMHT()* et *endMHT()*. Notons qu'il

```
1  // EPSN Simulation Sample
2  #include "mpi.h"
3  #include "EPSN_Simulation.hh"
4  #include "RedSYM.hh"
5
6  int main(int argc, char* argv[]) {
7
8    // (0) Initialize MPI
9    MPI_Init(&argc,&argv);
10   MPI_Comm_size(MPI_COMM_WORLD,&psize);
11   MPI_Comm_rank(MPI_COMM_WORLD,&prank);
12
13   // (1) Initialize EPSN
14   EpsnStatus status;
15   EPSN::SimulationInterface * epsn = new EPSN::SimulationInterface;
16   status = epsn->initORB(argc,argv);
17   if(prank == 0)
18     status = epsn->initProxy("sample-simu","sample.xml", psize);
19   status = epsn->initPort("sample-simu", prank, psize);
20
21   // (2) Add RedSYM objects to EPSN
22   RedSYM::Object * data = new RedSYM::Mesh("data", ...);
23   // ...
24   epsn->addData(object);
25
26   // (3) Set the simulation ready to accept the client connections
27   epsn->ready();
28
29   // (4) The simulation body with intrumentation points
30   epsn->beginMHT();
31   // ...
32   // ...
33   epsn->beginLoop("loop");
34   // ...
35   // ...
36   epsn->endLoop("loop");
37   // ...
38   // ...
39   epsn->endMHT();
40
41   // (5) Finalize EPSN & MPI
42   epsn->finalize();
43   epsn->killORB();
44   delete epsn;
45   MPI_Finalize();
46
47 }
```

FIG. 6.5 – Exemple C++ d'une simulation parallèle pilotable avec EPSN.

```
1  // EPSN Client Sample
2  #include "mpi.h"
3  #include "EPSN_Client.hh"
4  #include "RedSYM.hh"
5
6  int main(int argc, char* argv[]) {
7
8    // (0) Initialize MPI
9    MPI_Init(&argc,&argv);
10   MPI_Comm_size(MPI_COMM_WORLD,&psize);
11   MPI_Comm_rank(MPI_COMM_WORLD,&prank);
12
13   // (1) Initialize EPSN
14   EpsnStatus status;
15   EPSN::ClientInterface * epsn = new EPSN::ClientInterface;
16   status = epsn->initORB(argc,argv);
17   if(prank == 0)
18     status = epsn->initProxy("sample-client","sample-simu", psize);
19   status = epsn->initPort("sample-client", prank, psize);
20
21   // (2a) Get the remote parallel object description
22   RedSYM::ParallelObject * parallel_desc = getRemoteParallelDataDesc("data");
23
24   // (2b) And create the local RedSYM object for the client
25   RedSYM::Object * data = new RedSYM::Mesh("data", ...);
26   // ...
27   epsn->addData(object);
28
29   // (3) The client is ready to start a steering session
30   epsn->ready();
31
32
33   // (4) A sample of steering session
34   if(prank == 0) epsn->play();
35   // ...
36   EpsnRequestID request;
37   epsn->unlockWriteAccess("data");
38   if(prank == 0) request = epsn->get("data", ...);
39   // ...
40   // ...
41   if(prank == 0) epsn->wait(request);
42   epsn->lockWriteAccess("data");
43   // ...
44
45   // (5) Finalize EPSN & MPI
46   epsn->finalize();
47   epsn->killORB();
48   delete epsn;
49   MPI_Finalize();
50
51 }
```

FIG. 6.6 – Exemple C++ d'un client parallèle de pilotage EPSN.

est possible grâce au fichier XML de configurer le point d'entrée du MHT comme étant bloquant (attribut *auto-start="false"*) afin que les clients ne manquent aucune tâche au démarrage de la simulation.

5. Finalement, l'appel à la fonction *finalize()* permet de terminer l'exécution de la plate-forme EPSN, en déconnectant proprement tous les clients qui seraient encore connectés.

Côté client

1. Comme pour la simulation, la première étape est l'initialisation de la plate-forme EPSN grâce aux fonctions *initORB()*, *initProxy()* et *initPort()* de manière relativement symétrique. Cette étape commande la connexion du client à une simulation distante identifiée par son ID. Cet appel est synchronisant entre tous les processus du client et ne s'achèvera correctement que si la simulation est *ready*. En cas d'échec, il est possible de recommencer une nouvelle tentative de connexion.
2. Une fois connectée, le client de pilotage peut interroger dynamiquement le proxy de la simulation pour obtenir la description complète du MHT et des données distribuées. En particulier, la fonction *getRemoteParallelDataDesc()* permet d'obtenir le *descripteur parallèle* des données de la simulation (`RedSYM::ParallelObject`). Cette étape préliminaire (2a) permet au client de visualisation de construire dynamiquement des objets complexes (`RedSYM::Object`) compatibles avec la simulation, mais ayant leur propre distribution. Après quoi, cet objet est transmis à la plate-forme EPSN via la méthode *addData()* (étape 2b).
3. L'appel à la fonction *ready()* signale à la plate-forme EPSN que le client est prêt à démarrer une session de pilotage. Cet appel est synchronisant entre tous les processus du client. Le client rassemble alors sur le proxy client les descripteurs relatifs aux données clientes distribuées sur tous les ports, et forme un descripteur parallèle à transmettre à la simulation. C'est à ce moment précis que débute l'algorithme parallèle de redistribution (symétrique ou asymétrique) calculant sur chaque port, côté client et côté simulation, les messages à envoyer et à recevoir.
4. A la fin du *ready()* commence la session de pilotage. Le client déclenche alors les requêtes de son choix via le proxy : *play()*, *step()*, *stop()*, *get()*, *put()*, *etc*. Les ports clients quant à eux doivent gérer l'envoi ou la réception d'éventuelles données en configurant explicitement les plages d'accès en lecture/écriture avec les fonctions *unlockReadAccess()*, *lockReadAccess()*, *unlockWriteAccess()*, *lockWriteAccess()*. Les fonctions *wait()*, *test()*, *cancel()* permettent au proxy de contrôler le cycle de vie des requêtes en utilisant l'identifiant de la requête (`RequestID`) retourné au moment de sa soumission.
5. Finalement, l'appel à la fonction *finalize()* termine l'exécution de la plate-forme EPSN dans le programme client en le déconnectant proprement de la simulation distante.

6.2.3 La gestion des requêtes dans EPSN

Le noyau de la plate-forme EPSN (EPSN Core) supervise les autres couches et interagit avec les codes via les APIs côté simulation et côté client. Il gère d'une part la représentation de la simulation en MHT et la configuration des interactions grâce au *parsing* du fichier XML ainsi que l'établissement de la date courante lors de la traversée des points d'instrumentations dans le code de simulation, et d'autre part il est responsable de la gestion des requêtes de pilotage qui sont déclenchées à distance par le client et exécutées en parallèle côté simulation. La couche EPSN Core implante plusieurs objets CORBA – côté simulation et côté client – pour permettre l'échange des informations entre les codes couplés à la connexion, le transport des requêtes de pilotage et des acquittements, la coordination des traitements, *etc*. Dans cette section, nous allons nous intéresser plus spécifiquement à la gestion des requêtes.

Le gestionnaire de requêtes est l'élément central de la plate-forme EPSN coordonnant toutes les interactions de pilotage entre le client et la simulation. Dans le modèle présenté à la section 4.4.1, nous avons décomposé le cycle de vie d'une requête en quatre étapes clés : (1) l'envoi de la requête, (2) la coordination du traitement, (3) le traitement de la requête et (4) l'acquittement de la requête. La figure 6.7 résume schématiquement ces étapes pour les différents types de requête que nous considérons : *get*, *put*, *play/stop* et *action*.

1. La première étape est systématiquement initiée par le proxy client qui soumet une requête au proxy de la simulation.
2. Puis, dans un second temps intervient l'algorithme de coordination (cf. Sec 4.4.2) qui établit une date de planification t_p pour cette requête et la transmet aux ports de la simulation. L'établissement de cette date suppose une phase de communication préliminaire où le proxy soumet un ordre *freeze* aux ports et les interroge pour obtenir la date courante parallèle. Après quoi, le proxy peut planifier une date de traitement et transmettre la requête à chaque port, enrichie de cette date. L'ordre *freeze* sera automatiquement relâché à la réception de la requête sur les ports.
3. Le traitement de la requête sur les ports de la simulation est ensuite décomposé en deux sous-étapes : l'évaluation d'une condition locale retardant l'exécution et l'exécution effective de la requête. Si l'algorithme de coordination permet de garantir globalement la cohérence en temps des interactions de pilotage, l'évaluation de la condition permet quant à elle de garantir localement la cohérence de ces interactions.
4. Pour plus de flexibilité, le système d'acquittement (étape 4) est entièrement paramétrable. En effet, pour certaines requêtes comme *play/stop*, l'acquittement des traitements n'est pas forcément pertinent. Pour les requêtes de type *get* ou *getp*, cet acquittement joue un rôle important dans la régulation des transferts : il part des ports du client recevant les données jusqu'aux ports de la simulation en passant par les proxys. L'acquittement entre le proxy du client et le proxy de la simulation peut éventuellement être contrôlé explicitement par l'utilisateur (fonction *ack()* de l'API) afin de laisser le temps au code de visualisation de traiter les données reçues. Dans le cas de la requête *put* ou *action*, nous utilisons un système de double acquittement : (4a) pré-acquittement des ports sur le proxy de la simulation et (4b) post-acquittement du proxy sur les ports de la simulation signalant la fin *globale* du traitement parallèle. L'acquittement final du proxy client (et éventuellement des ports) permet de signaler la fin du traitement à l'utilisateur, ce qu'il peut contrôler grâce aux fonctions *test()* et *wait()* de l'API.

Comme nous venons de le voir, le cycle de vie d'une requête dans EPSN fait intervenir le proxy et les ports côté simulation et côté client qui chacun possède un gestionnaire de requêtes (classe `RequestManager`) stockant la table des requêtes en cours. Chaque requête (classe `Request`) est identifiée de manière unique sur le réseau par un *RequestID*, initialement délivré par le proxy client. Les gestionnaires communiquent entre eux grâce à un objet CORBA de type `RequestInterface` fournissant des opérations élémentaires comme la soumission d'une requête, l'annulation d'une requête ou l'acquittement d'une requête. Sur les ports de la simulation, le `RequestManager` joue un rôle particulier *d'exécuteur des requêtes*, que nous allons détaillé au paragraphe suivant.

Les exécuteurs de requêtes et le pool de *workers*

Nous associons à chaque requête reçue par un port de la simulation un *exécuteur* de type abstrait `RequestExecutor`. Cette classe contient la méthode abstraite *execute()* qui devra être invoquée lorsque la requête sera prête à être exécutée, ainsi que la méthode abstraite *isReady()* qui permet de tester la condition locale d'exécution. L'implantation d'un exécuteur (sous-classe de `RequestExecutor`) dépend directement du type de la requête considérée *get*, *put*, *action*, *etc*. Ceci étant posé, nous allons examiner plus en détails le fonctionnement du gestionnaire de requêtes sur les ports de la simulation. Les requêtes dans ce gestionnaire sont triées en trois listes différentes en fonction de leur statut : *NotReady*, *Ready* et *Blocking* (Fig. 6.8).

NotReady. Une requête est présente dans cette liste si elle n'est pas encore prête, soit parce qu'elle n'a pas encore dépassé la date t_p planifiée pour débuter son traitement, soit parce que la condition locale associée à ce traitement n'est pas encore vérifiée. C'est en principe le statut d'une requête lors de sa réception. Rien n'arrivera aux requêtes dans cette liste tant qu'elles n'auront pas changé de statut (et donc de liste).

Ready. Les requêtes dans cette liste sont prêtes à être exécutées ou en cours d'exécution ou dans l'attente d'un acquittement achevant le traitement. La condition primordiale pour être dans cette liste et d'être à l'intérieur de l'intervalle d'exécution $[t_e, t_{e'}]$, mais de ne pas avoir atteint la date $t_{e'}$ au delà de laquelle l'exécution de la requête n'est plus autorisée.

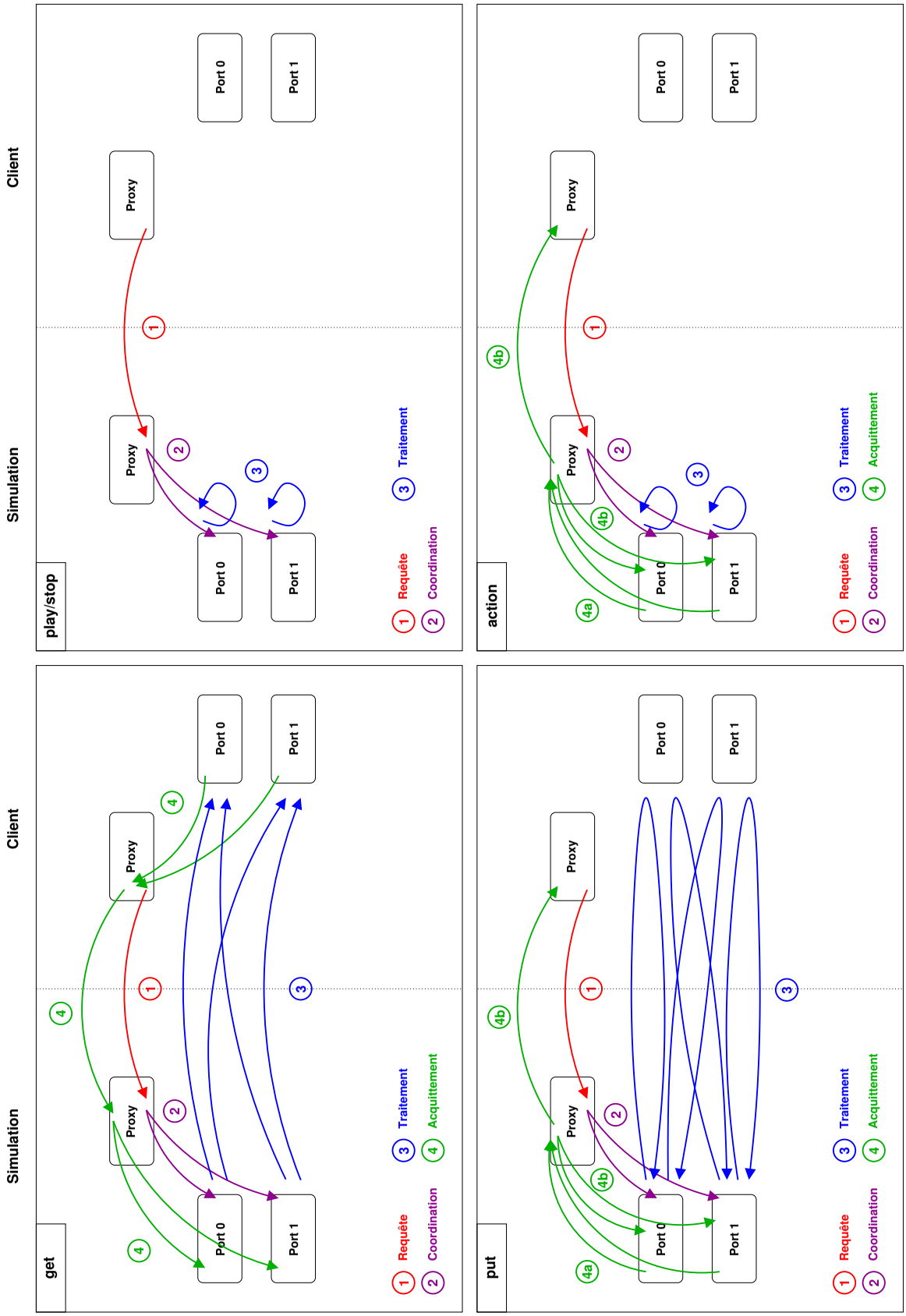


FIG. 6.7 – Gestion des requêtes dans EPSN.

Blocking. Les requêtes présentes dans cette liste ont atteint la date $t_{e'}$ à la fin de l'intervalle d'exécution, mais n'ont pas complètement achevé leur traitement. En fait, ces requêtes peuvent être dans l'attente d'un acquittement, en cours d'exécution ou même ne pas avoir commencé leur exécution, exactement comme si elles étaient *Ready*. La principale différence vient du fait que ces requêtes doivent nécessairement s'achever sur le point d'instrumentation courant, afin de maintenir la cohérence spatiale. Dans ce cas, EPSN bloque l'avancement de la simulation sur le point d'instrumentation courant, le temps qu'il sera nécessaire pour achever de traiter toutes les requêtes présentes dans cette liste. Après quoi, le point d'instrumentation est débloqué et la simulation reprend son cours. Le statut *Blocking* est donc un statut transitoire qui permet d'achever le traitement des requêtes (i.e. recouvrement partiel) ou d'effectuer des interactions bloquantes sur des tâches en point. Notons que cette liste doit nécessairement être vide entre deux points d'instrumentation.

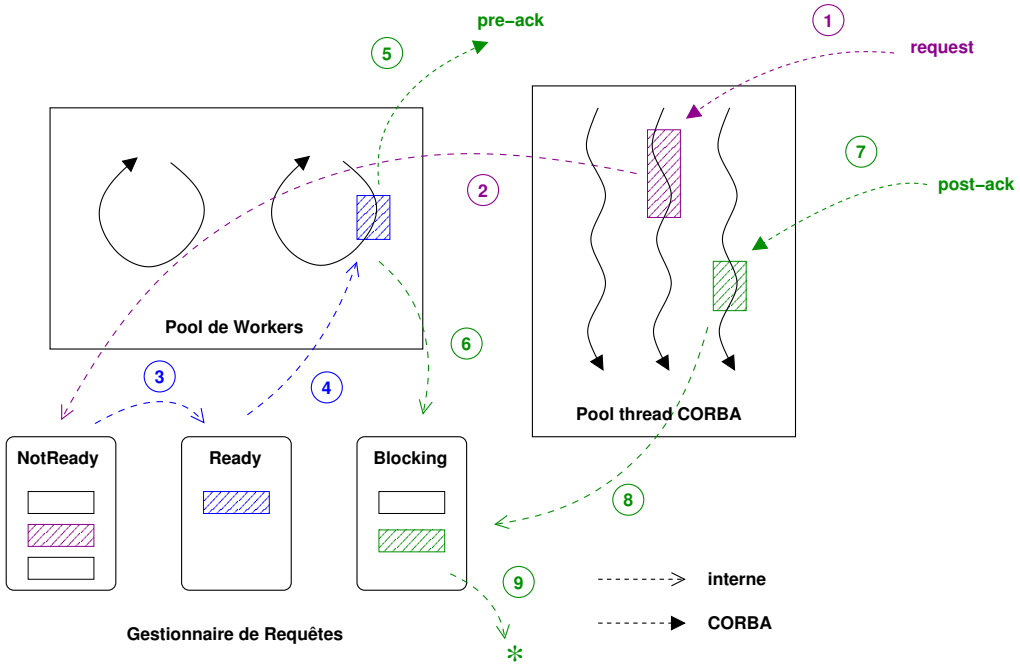


FIG. 6.8 – La gestion des requêtes sur ports de la simulation : de la réception de la requête (1-2) à l'exécution (3-4), jusqu'à l'acquittement complet (5-8). A l'issue du pré-acquittement (5), la requête peut passer dans la liste *Blocking* (6) dans l'attente d'un post-acquittement (7-8), avant d'être finalement détruite (9).

L'exécution des requêtes sur les ports de la simulation est prise en charge par un *pool de threads* appelés *workers*. Les *workers* traitent en boucle et de manière concurrente les requêtes dès qu'elles sont prêtes, de statuts *Ready* ou *Blocking*, en commençant prioritairement par les requêtes *Blocking* car elles sont susceptibles de ralentir la simulation. Lorsque le *worker* traite une requête, il fait appel à la méthode *execute()* de son exécuteur. L'exécuteur possède également un statut qui renseigne sur l'état d'avancement de son exécution. Initialement, le statut de l'exécuteur est *NotExecuted*, puis passe à *OnGoing* lorsque l'exécution a commencé et devient finalement *Executed* lorsque le traitement de la requête s'achève. La requête est alors retirée de sa liste courante. Certaines requêtes comme *put* ou *action* transmettent à la fin de leur exécution locale un acquittement au *proxy* de la simulation, et d'autres sont susceptibles comme *get* ou *action* de recevoir un acquittement global. Dans ce dernier cas illustré sur la figure 6.8, l'exécuteur prend un statut intermédiaire de *Zombie* jusqu'à la réception de cet acquittement. Ce mécanisme combiné aux requêtes *Blocking* est particulièrement important pour la régulation des transferts entre la simulation et le client. Dans le cas d'une requête permanente (*getp*), l'exécuteur prend le statut particulier *Recycling* à la fin de sa première exécution, et la requête associée retourne automatiquement dans la liste *NotReady* du gestionnaire de requêtes. A chaque fois que la requête est de nouveau *Ready*, elle sera ré-exécutée par un *worker* et ainsi de suite, à moins qu'une requête *cancel* vienne interrompre explicitement le recyclage. Notons qu'il est possible de contrôler la période à laquelle la requête est répétée. Comme pour les requêtes simples, la cohérence en temps des requêtes permanentes est

assurée par la planification de la date initiale t_p et l'hypothèse d'un parcours SPMD strict du MHT (cf. Sec. 4.4.1).

Le nombre de *workers* peut être contrôlé par la variable d'environnement `EPSN_NB_WORKERS` positionnée par défaut à 1. L'utilisation de plusieurs *threads* permet de maximiser la concurrence des traitements, notamment lorsqu'il y plusieurs clients connectés simultanément. Afin de minimiser les perturbations dans la simulation, les *workers* s'endorment lorsqu'il n'y a plus de requêtes prêtes à traiter. Il seront réveillés automatiquement par le gestionnaire de requêtes lorsqu'une nouvelle requête sera prête, ce qui peut arriver à la réception d'une requête via CORBA ou à la rencontre d'un point d'instrumentation dans la simulation modifiant le statut d'une requête *NotReady*. Il est également possible de configurer à l'aide de la variable d'environnement `EPSN_WORKER_WAKE_UP` un *timeout* (en ms) à grâce auquel le *worker* se réveille spontanément, ce qui augmente la réactivité du gestionnaire de requêtes. Dans le cas des architectures SMP, il est possible d'améliorer considérablement les performances du système en configurant le placement des threads de la plate-forme EPSN sur les CPUs disponibles grâce à la variable d'environnement `EPSN_AFFINITY_MODE`. Nous avons imaginé plusieurs modes de placement adaptés aux machines bi-processeurs : le processus principal de la simulation sur la CPU 0 et tous les workers sur le CPU 1, tous les threads sur le CPU 0 ou 1, *etc.*. Par défaut, nous ne définissons aucune affinité, ce qui revient à laisser le choix à l'ordonnanceur du système d'exploitation de placer les différents threads et de les faire migrer selon sa politique. Ces solutions sont en pratique suffisantes dans le cas des bi-processeurs, mais devront être étendues pour des quadri-pros et plus. En outre, nous laissons la gestion du pool de *threads* CORBA à la discrétion de l'ORB utilisé.

6.3 Les couches logicielles de la plate-forme EPSN

6.3.1 La couche de connexion ColCOWS

ColCOWS (Collective CORBA Object Workspace) est une bibliothèque développée au sein du projet EPSN afin de faciliter le partage de références d'objet CORBA (ou IOR) au sein d'un groupe de processus, formant ce que nous appelons un *workspace*. Plus précisément, un *workspace* est constitué d'un ensemble de nœuds, c'est-à-dire des threads ou des processus distincts qui jouent le rôle de serveurs CORBA. Chaque nœud dispose d'un ensemble d'objets CORBA qu'il souhaite mettre en partage (i.e. les références locales). Chaque *workspace* est identifié par un ID sous forme d'une chaîne de caractères. Chaque nœud est identifié par son rang dans le *workspace*. Chaque objet est identifié sur chaque nœud par une chaîne de caractères servant de clé dans une table de hachage stockant les IORs.

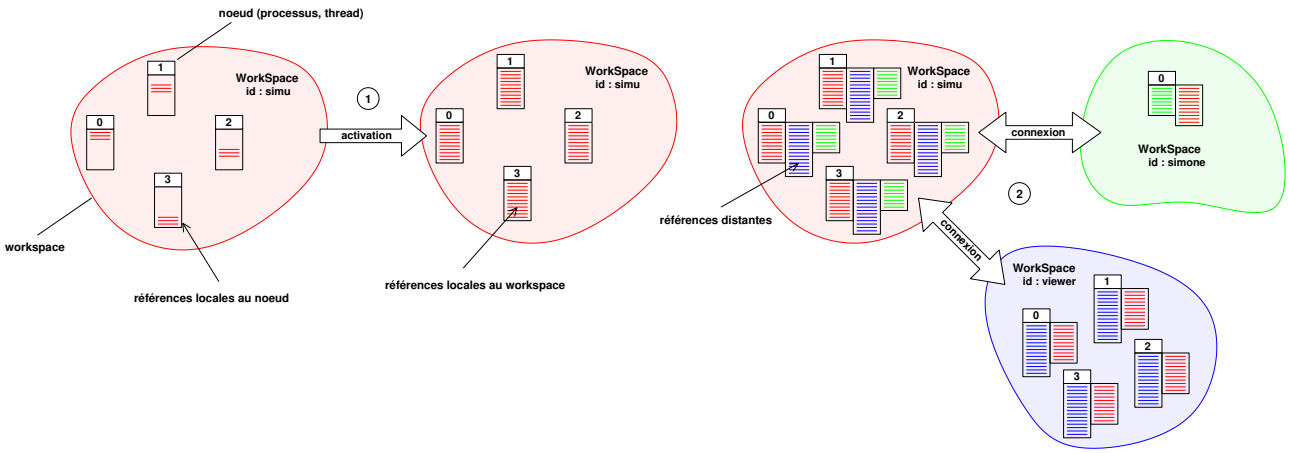


FIG. 6.9 – ColCOWS : activation et connexion entre *workspaces*.

Initialement, chaque nœud possède un ensemble d'IORs locales qu'il souhaite mettre en partage dans le *workspace*. ColCOWS offre principalement deux fonctionnalités : l'activation et la connexion (Fig 6.9).

L'activation est une opération collective comparable à un *all-to-all* consistant à diffuser à tous les nœuds du *workspace* l'ensemble des IORs présentes sur chaque nœud. A l'issue de cette opération, chaque nœud possède une connaissance globale de toutes les références locales au *workspace*. La connexion intervient après l'activation et permet à deux *workspaces* de mettre en partage l'ensemble des IORs présentes dans chaque *workspace*. Chaque connexion est identifiée dans ColCOWS par un numéro de connexion unique, commun aux deux *workspaces*. En outre, ColCOWS permet de gérer les opérations inverses : déconnexion et désactivation.

ColCOWS se présente comme une bibliothèque C++ essentiellement basée sur la classe `ColCOWS::Node`, qui contient une structure de données logique pour stocker et accéder à l'ensemble des références CORBA associées au *workspace* local ainsi qu'à tous les *workspaces* distants connectés à celui-ci. L'implantation de ColCOWS utilise un objet CORBA « en interne » fournissant les opérations élémentaires pour la diffusion des références CORBA au sein d'un *workspace* ou entre *workspaces*. ColCOWS repose en partie sur l'utilisation du service de nommage de CORBA, une sorte d'annuaire des IORs, dont nous nous servons pour localiser les *workspaces* sur le réseau. Du point de vue de la plate-forme EPSN, ColCOWS représente la *couche connexion*, car elle va permettre d'interconnecter les *workspaces* de la simulation et du client de pilotage, et donc de partager les différents objets CORBA utilisés dans la couche EPSN Core et dans RedCORBA. Cette couche est fondamentale car elle permet de mettre en relation les ports et les proxys du client de la simulation qui vont communiquer via des appels de méthodes sur des objets distribués CORBA. Notons que ColCOWS permet de gérer la connexion entre plusieurs *workspaces*, ce qui en fait le socle de la plate-forme distribuée EPSN. Dans EPSN, l'activation et la connexion sont effectuées de manière transparente lors des appels aux fonctions *initPort()* et *initProxy()*. La connexion est initiée par le client qui doit indiquer la localisation du service de nommage CORBA sur le réseau (nom de machine, port) dans lequel le *workspace* de la simulation s'est inscrit. En outre, ColCOWS permet d'informer les différentes couches logicielles de la plate-forme EPSN des évènements relatifs au cycle de vie d'un *workspace* (activation, connexion, *etc*.) afin de réaliser les traitements appropriés.

6.3.2 La couche de redistribution RedSYM

La bibliothèque RedSYM implante le modèle de description des données et les algorithmes de redistribution présentés au chapitre 5. Plus précisément, RedSYM permet de calculer sur chaque processeur P_i l'ensemble des *messages symboliques* pour tous les processeurs Q_j à partir de la description de l'objet complexe local et du descripteur des objets complexes distants. RedSYM a été développé en C++, mais propose également une API C et Fortran 77 (compatible Fortran 90). Ce *framework* est prévu pour être extensible selon deux axes principaux : ajout de nouveaux objets complexes, ajout de nouvelles stratégies de redistribution. La bibliothèque RedSYM est dite *symbolique* dans le sens où son interface est entièrement indépendante de la couche de communication. Nous verrons dans la section suivante avec la bibliothèque RedCORBA une implantation possible d'une couche de communication basée sur CORBA.

La description des données est très succincte dans l'environnement EPSN, car nous avons choisi de déporter entièrement le modèle de données dans la bibliothèque RedSYM. La représentation manipulée par EPSN se résume principalement aux informations suivantes : l'identifiant de l'objet (sous la forme d'une simple chaîne de caractères), la classe de l'objet complexe, le nombre de séries de données et un identifiant pour chaque série. Toutes ces informations sont détenues par la classe mère des objets RedSYM, la classe `RedSYM::Object`, qui est aussi la seule classe utilisée par EPSN. En pratique, l'utilisateur décrit les objets complexes qu'il souhaite coupler à l'aide de l'API de RedSYM et fournit à l'environnement EPSN cet objet grâce à la fonction *addData()*. Par la suite, l'API EPSN manipule uniquement les identifiants associés aux objets et aux séries de RedSYM. La fonction *updateData()* de l'API EPSN permet de signaler à RedSYM qu'un objet a été localement modifié et ainsi de demander la mise à jour locale des messages générés. Cela permet essentiellement de prendre en compte des modifications liées au stockage et à la dynamicité des objets : *swap* d'un buffer d'une itération à l'autre, modification du nombre courant d'atomes ou de particules dans une région, *etc*. Toute modification plus profonde de l'objet complexe, comme par exemple un changement de distribution, nécessitera un nouveau calcul complet de la matrice de communication, ce qui n'est pas supporté dans la version courante d'EPSN.

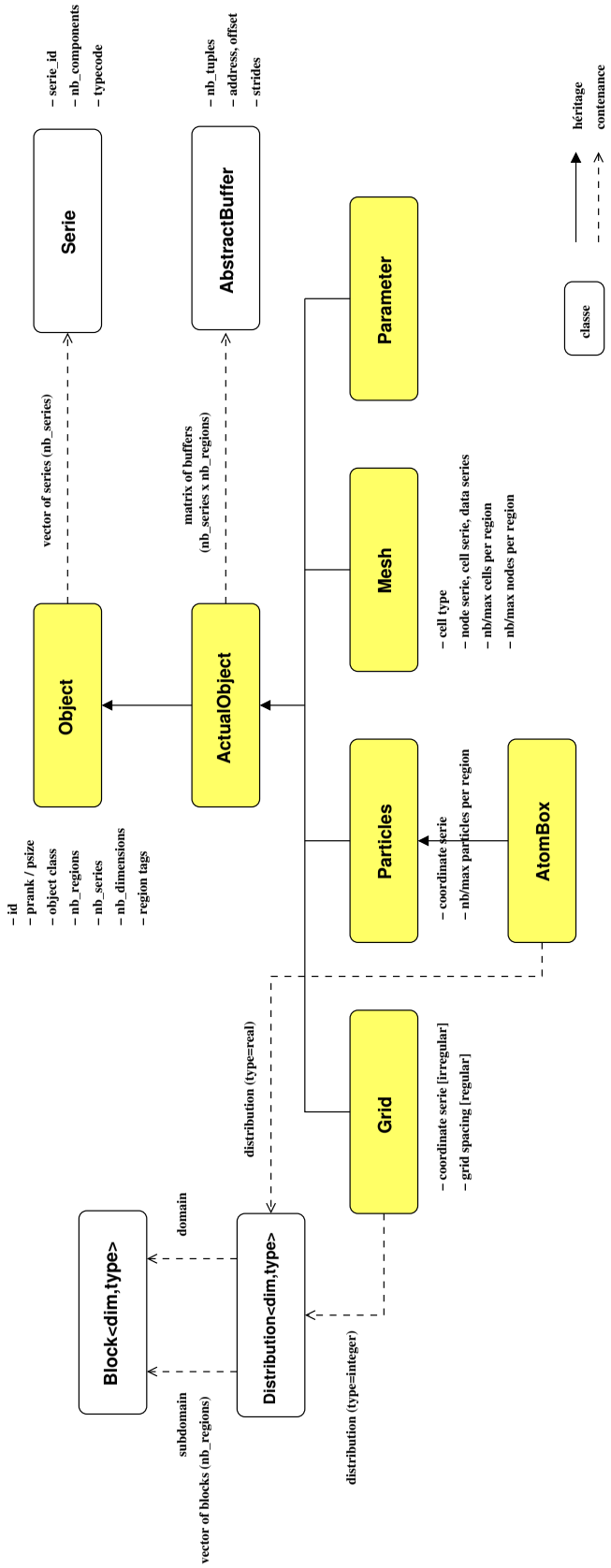


FIG. 6.10 – La hiérarchie des objets RedSYM.

La hiérarchie des objets RedSYM

RedSYM supporte actuellement cinq classes d'objets complexes que nous avons détaillées à la section 5.4.3 : les paramètres scalaires (`RedSYM::Parameter`), les grilles structurées régulières ou irrégulières (`RedSYM::Grid`), les boîtes d'atomes (`RedSYM::AtomBox`), les ensembles de particules (`RedSYM::Particles`) et les maillages non structurés (`RedSYM::Mesh`). La hiérarchie des objets RedSYM est représentée sur la figure 6.10. On peut remarquer que les grilles régulières ou irrégulières sont prises en compte par la même classe `RedSYM::Grid`, qui permet également de modéliser des matrices (creuses par bloc) et plus généralement des tableaux multi-dimensionnels distribués par bloc. La principale différence entre les différentes types de grilles tient aux informations fournies pour représenter graphiquement ces objets : coordonnées des nœuds calculées implicitement ou stockées explicitement dans une série de données. La distribution par bloc des grilles est simplement représentée par une collection de blocs à coordonnées entières (`Block<dim,integer>`). Notons que dans le cas des boîtes d'atomes, la distribution par bloc utilise des coordonnées réelles (`Block<dim,real>`). On peut également remarquer que les boîtes d'atomes sont implantées comme une classe dérivée des ensembles de particules associant à chaque région un bloc dans la distribution. Tous ces objets héritent de la classe mère `RedSYM::Object`, qui contient les informations de haut-niveau décrivant un objet : identifiant (chaîne de caractères), classe de l'objet, nombre de régions, nombre de séries, description des séries (identifiant, type de données, nombre de composantes), *tags* sur les régions, *etc*. Le stockage en mémoire des données associées à l'objet complexe est géré par la classe `RedSYM::ActualObject` dérivant de `RedSYM::Object`. Cette classe contient une matrice de buffers pour chaque série et chaque région. Chaque buffer est représenté par une instance de la classe `RedSYM::AbstractBuffer`, qui implante le modèle de stockage avec *stride* décrit à la section 5.4.2. A terme, cette classe pourra être étendue pour supporter d'autre type de stockage. La classe `RedSYM::ActualObject` fournit principalement deux méthodes aux classes filles : la méthode *wrap()* qui permet « d'envelopper » les données telles qu'elle sont présentes en mémoire (pas de copie), et la méthode *allocate()* qui permet à RedSYM d'allouer lui-même les données en mémoire, ce qui s'avère plus particulièrement utile pour les codes de visualisation. En particulier, nous utilisons deux variantes de la fonction *wrap()* que sont *wrapStride()* et *wrapGhost()* lorsque les données possèdent un espace de stockage qui n'est pas contigu. Nous fournirons au chapitre 7 plusieurs exemples d'utilisation de l'API RedSYM pour déclarer des objet complexe que seront redistribués et pilotés par EPSN.

Calcul des messages symboliques

Le calcul parallèle des messages symboliques sur un processeur P_i repose dans RedSYM sur la classe abstraite `AbstractCommunicationMatrix` (cf. Sec. 5.5). Cette classe est suffisamment générique pour prendre en compte une grande variété de stratégies de redistribution (approche spatiale, placement des régions ou *split/merge*) et de classes d'objets complexes (y compris les paramètres). En pratique, l'implantation d'une stratégie de redistribution nécessite de définir une sous-classe définissant les méthodes abstraites *compute()* et *update()*. L'implantation des méthodes *compute()* et *update()* repose entièrement sur la description de l'objet complexe local ainsi que sur le *descripteur parallèle* de l'objet complexe distant. La connaissance de ce descripteur suppose que son transport soit pris en charge par la couche de communication. La figure 6.11 représente l'organisation dans RedSYM des différents algorithmes de redistribution que nous proposons. Par exemple, la classe `CommunicationMatrixGrid` implante l'algorithme de redistribution des grilles structurées (régulière ou non) selon l'approche spatiale. La classe `CommunicationMatrixNoSplitMerge` implante l'algorithme de redistribution basé sur une approche placement sans découpage, qui peut s'appliquer de manière très générique à toutes les classes d'objet complexe. En revanche, la stratégie de redistribution « placement avec découpage » est uniquement disponible pour les maillages non structurées et les ensembles de particules (classes filles de `CommunicationMatrixSplitMerge`). Notons que RedSYM est dépendant de la bibliothèque Metis [17] pour le partitionnement des maillages non structurés. A terme, il sera également possible d'utiliser d'autres partitionneurs comme par exemple Scotch [173].

La classe `AbstractCommunicationMatrix` contient une matrice de `CommunicationBuffer` pour chaque processeur distant Q_j et chaque série de données S_s, encapsulant les informations relatives aux différents messages symboliques $\hat{m}_{i,j}$. Plus précisément, la classe `CommunicationBuffer` est un simple vecteur de `PieceOfCommunicationBuffer`, servant à représnter les sous-messages symbolique. Chaque sous-message

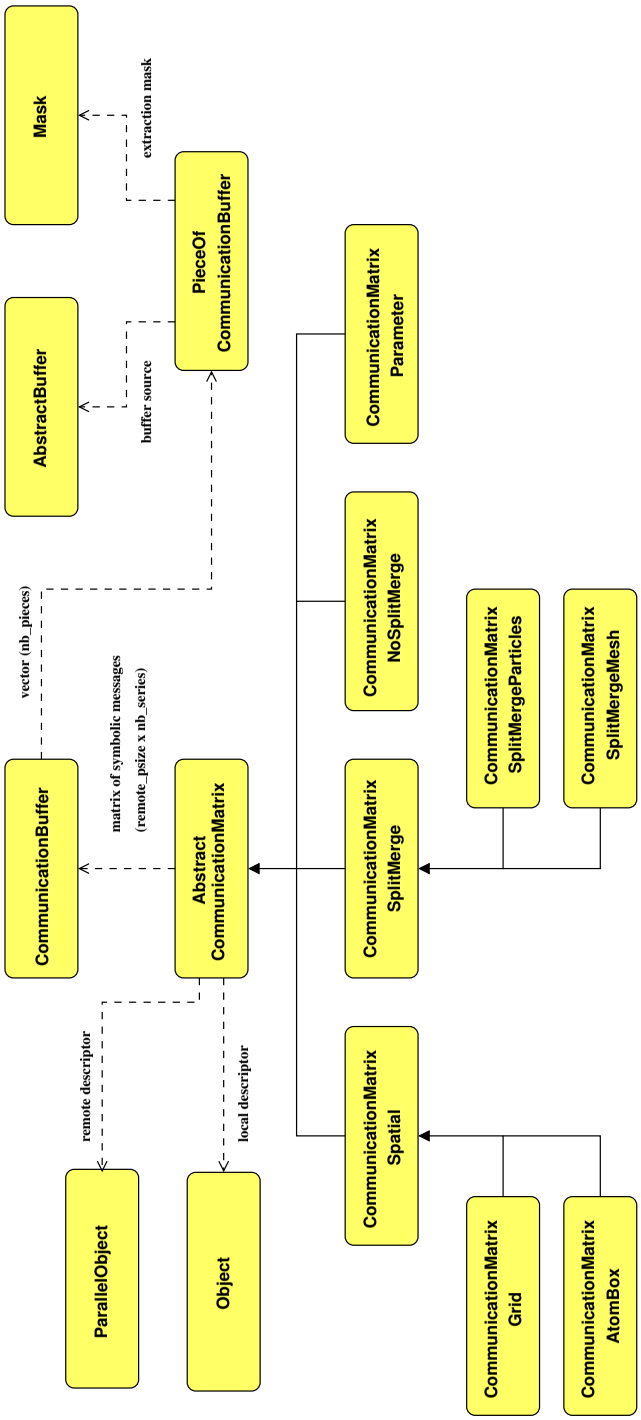


FIG. 6.11 – Matrices de communication dans RedSYM.

est défini par une référence vers un `AbstractBuffer` désignant le buffer de la région source et une référence sur un masque (`Mask`) décrivant la liste ordonnée des éléments constituant le sous-message. Notons que les masques sont le plus souvent partagés pour toutes les séries, sauf dans le cas des maillages où nous distinguons des masques différents pour les cellules et les nœuds. La classe `AbstractCommunicationMatrix` fournit plusieurs services pour faciliter le développement d'une *couche de transfert*, comme RedCORBA. Tout d'abord, elle fournit l'ensemble des accesseurs nécessaires à l'exploration des messages symboliques calculés, ainsi que des *itérateurs* permettant de parcourir efficacement les données en mémoire constituant les messages physiques. En outre, cette classe fournit des primitives de bas-niveau permettant de copier les données du message en mémoire (i.e. décrites par un `CommunicationBuffer`) vers un tampon[1] et inversement. Ces primitives sont typiquement utilisées pour gérer un cache à l'envoi si les données sont « trop stridées » ou encore pour implanter le mécanisme de réception dynamique présenté à la section 5.5.3. En effet, dans ce mécanisme, il est nécessaire de recevoir les données dans un tampon séparé, car l'emplacement en mémoire des données à recevoir n'est pas déterminé à l'avance. La classe `AbstractCommunicationMatrix` fournit également des informations de haut-niveau comme la *table des communications*, indiquant pour chaque paire de processeurs s'il faut échanger ou non un message. En particulier, cette table permet de connaître le nombre exact de messages à envoyer et à recevoir, en ignorant les messages vides (schémas de communication partiels). La connaissance de cette table est nécessaire par exemple dans le système de requêtes d'EPSN pour savoir combien il faut recevoir de messages avant d'acquitter localement le transfert, et combien d'acquittements il faut attendre au total avant d'acquitter la requête. Par ailleurs, la table des communications contient les informations sur la taille en octets des messages échangés. Cette information peut être utilisées pour calculer un ordonnancement des communications. RedSYM propose une stratégie d'ordonnancement classique en *round-robin*, sans découpage des messages à transmettre. Cette stratégie – qui sera utilisée dans RedCORBA – consiste pour chaque processeur P_i à envoyer son premier message non vide au processeur distant Q_j tel que $j = i\%N$ et ainsi de suite. Cette stratégie permet d'éviter que tous les processeurs P_i communiquent initialement avec le même Q_j (i.e. collisions à la réception).

6.3.3 La couche de transfert RedCORBA

RedCORBA est une bibliothèque de couplage de codes, qui permet d'interconnecter dynamiquement plusieurs codes parallèles (MPI, PVM, *etc*.) et de partager des données ayant des distributions différentes entre ces codes. Son architecture repose sur une approche composant – inspirée librement de PAWS [44] – basée sur une collection de ports et sur un proxy de manière similaire à EPSN. RedCORBA s'appuie sur la bibliothèque de redistribution RedSYM pour calculer les messages à échanger et se base sur la technologie CORBA pour transférer ces messages sur un réseau potentiellement hétérogène. RedCORBA utilise la bibliothèque ColCOWS pour gérer de multiples connexions entre les codes : couplage entre deux simulations ou couplage d'une simulation avec plusieurs clients de visualisation, *etc*. Au sein de la plate-forme de pilotage EPSN, RedCORBA joue le rôle de la *couche de transfert*, mais il faut souligner que cette bibliothèque peut être utilisée dans le contexte du couplage de manière tout à fait indépendante d'EPSN. Du point de vue de l'architecture logicielle, cela signifie que RedCORBA possède sa propre API utilisateur ainsi que son propre système de requêtes qui est sensiblement différent de celui que nous avons présenté pour EPSN (cf. Sec. 6.2.3). La principale différence tient au fait que les codes couplés jouent un rôle symétrique dans RedCORBA, ce qui n'est pas le cas dans EPSN. En pratique, cela nous a permis de valider les algorithmes de redistribution implantés dans RedSYM indépendamment de la plate-forme EPSN.

D'une manière générale, RedCORBA assure trois fonctionnalités principales : (1) la traduction des objets complexes RedSYM en descripteur IDL échangeables sur le réseau[2], (2) la génération des messages CORBA à partir des messages symboliques calculés par RedSYM et (3) le transfert des messages générés via CORBA. RedCORBA utilise une abstraction en terme de *channel* pour virtualiser le transfert des données entre les composants interconnectés. Pour chaque connexion et chaque objet complexe RedSYM, un *channel* particulier est construit, qui sert à commander le transfert des messages relatifs aux différentes séries de données associées à cet objet. Ce

[1] Notons que ce tampon est également représenté dans RedSYM par une instance de la classe `AbstractBuffer` ou de la classe `CommunicationBuffer`. Dans ce dernier cas, la copie prend en compte un masque sur les éléments du buffer destination.

[2] Notons que la traduction des objets RedSYM en IDL est extériorisée dans une bibliothèque baptisée RedSYM2IDL. En fait, nous souhaitons généraliser cette approche pour traduire les objets complexes RedSYM vers différents formats, comme par exemple des sources de visualisation VTK ou des fichiers de stockage HDF5.

channel est représenté sur le proxy et sur chaque port par une instance de la classe `RedCORBA::ProxyChannel` ou `RedCORBA::PortChannel`. Le channel contient la matrice de communication RedSYM ainsi que les messages physiques CORBA et sert à commander la redistribution des données entre les codes couplés. Notons que pour un même objet complexe RedSYM, il y aura autant de *channels* créés que de connexions ouvertes. En effet, il faut bien considérer que les messages générés diffèrent pour chaque code connecté, car la distribution des données dans ces codes voire la stratégie de redistribution choisie est potentiellement différente. Dans EPSN, la création de tous les *channels* associés à une connexion cliente s'effectue automatiquement au moment de l'appel par le client à la fonction *ready()* de l'API EPSN. C'est à ce moment précis que RedCORBA effectue l'échange des descripteurs et qu'il calcule sur chaque port les messages CORBA à partir des messages symboliques RedSYM.

Le transfert des messages

RedCORBA propose un modèle de communication *one-sided* basé sur des requêtes de type *get/put*. Contrairement au modèle de communication *two-sided* (i.e. *send/recv*), le transfert des données dans un sens ou l'autre est commandé par « un seul côté ». L'envoi et la réception sont alors contrôlés par la déclaration de plage d'accès en lecture & écriture pour les séries de données de chaque objet complexe. Dans RedCORBA, les requêtes *get* et *put* sont utilisées de manière symétrique par les codes couplés selon deux modes différents : le *mode centralisé* et le *mode parallèle*. Dans le premier mode, le transfert est commandé de manière centralisée à partir du `ProxyChannel` qui transmet la requête aux `PortChannel`. A l'inverse, dans le second mode, le transfert des données est commandé en parallèle directement à partir de tous les `PortChannel`. C'est ce dernier mode qui est utilisé pour implanter la couche transfert de la plate-forme EPSN. Ainsi, lorsqu'un client de pilotage soumet une requête de type *get* à la simulation, EPSN invoque la requête *put* de RedCORBA sur les ports de la simulation pour envoyer les données aux ports clients, et inversement dans le cas d'une requête EPSN de type *get* (cf. Fig. 6.7).

Le transfert des messages dans RedCORBA repose en interne sur un objet CORBA de type `DataInterface` associé à chaque port (Fig. 6.12). Cette interface est partagée pour toutes les connexions clientes et toutes les séries de données relatives aux objets complexes. On trouve principalement deux méthodes dans cette interface : la méthode *requestMessage()* et la méthode *sendMessage()*. La première permet de commander à un port local ou distant de transférer ses messages vers leurs destinataires, ce qui est ensuite réalisé avec la seconde méthode. Ainsi, une requête de type *get* se traduit par un appel à la méthode *requestMessage()* de l'objet `DataInterface` distant ; à l'inverse une requête de type *put* se traduit par un appel à cette méthode sur l'objet local. La nature de la requête est décrite par un descripteur de type `RequestDescriptor` qui contient des informations de haut-niveau associées à cette requête : identifiants de la requête, de l'objet complexe, de la série, numéro de connexion, *etc*. Le message s'accompagne d'un descripteur de type `MessageDescriptor` qui reprend les informations précendentes et les complète avec notamment le numéro du port émetteur du message et la *release* de la donnée au moment de l'envoi. Le type du message dépend directement du type de la série de données considérées. Les types IDL actuellement supportés sont tous les types primitifs classiques : *octet*, *short*, *unsigned short*, *long*, *unsigned long*, *float* et *double*. Notons qu'il est nécessaire pour prendre en compte de nouveaux types structurés d'étendre cette interface[3]. Les messages RedCORBA respectent la structure logique en sous-messages définie à la section 5.5. Ces messages sont définis comme un vecteur CORBA (i.e. *sequence*) de sous-messages (i.e. de type `PieceOfMessage`). Chaque sous-message identifie clairement les éléments d'une région source à transmettre vers une région cible. Les données relatives à un sous-message se présente alors sous la forme d'une séquence de buffers, chaque buffer représentant un tableau de données contigu en mémoire (e.g. `sequence<double>`).

Afin d'optimiser les performances lors des transferts successifs, les messages symboliques RedSYM et les messages physiques CORBA sont pré-calculés pour toutes les séries et stockés dans les `PortChannel`. Ainsi, il n'est pas nécessaire de recalculer les messages à envoyer à chaque transfert. Les messages symboliques permettent de regénérer les messages physiques à la demande lorsque ces derniers ne sont plus à jour (i.e. modification espace de stockage d'un objet ou cas d'un objet dynamique interne). La construction des messages

[3]Une alternative possible consisterait à utiliser le type CORBA *any*, permettant d'encapsuler n'importe quel type IDL défini par l'utilisateur. Nous n'avons pas exploré cette voie davantage car en pratique l'utilisation des seuls types primitifs s'avère suffisante.

```
1  // CORBA Object Interface of RedCORBA Port
2  interface DataInterface {
3
4    // Buffer
5    typedef sequence<double> BufferDouble;
6
7    // Piece of message
8    typedef sequence<DoubleBuffer> PieceOfMessageDouble;
9
10   // Message
11   typedef sequence<DoublePieceOfMessage> MessageDouble;
12
13   // Request to send message(s)
14   void requestMessage(in RequestDescriptor descriptor) raises RequestException;
15
16   // Send a message
17   oneway void sendMessageDouble(in MessageDescriptor descriptor, in MessageDouble message);
18
19   // ...
20 };
```

FIG. 6.12 – Extrait de l'interface IDL responsable du transfert des messages CORBA (cas du type *double*).

physiques repose sur l'utilisation de la fonction *replace()* des séquences CORBA qui permet « d'envelopper » en mémoire une plage de données contiguë. Si l'ORB choisi le permet, comme c'est le cas pour OmniORB4 [23], alors l'envoi des messages s'effectuera de manière performante sans copie (stratégie *zéro-copie*). Lorsque les messages sont trop découpés en mémoire (i.e. stride), il vaut mieux activer l'utilisation d'un cache dans RedCORBA. Ce cache permet de construire un message contigu en mémoire, mais au prix d'une recopie mémoire avant chaque transfert. En outre, l'utilisation d'un cache permet de maximiser la plage de recouvrement possible. Afin d'optimiser l'envoi des messages CORBA, la méthode *sendMessage()* de `DataInterface` est déclarée *oneway*[4]. L'utilisation de ce qualificatif permet de rendre l'appel à la méthode distante non boquant, du moins pendant l'exécution du corps de la méthode, car l'appel à une méthode *oneway* reste cependant bloquant durant le transfert des arguments vers le serveur, c'est-à-dire pendant le transfert des messages. En outre, la réception d'un message CORBA comme argument de la méthode *sendMessage()* se fait dans une plage mémoire réservée par l'ORB. Il est alors nécessaire de recopier les données reçues vers la plage mémoire du code utilisateur déterminée par RedSYM. C'est aussi le moment opportun pour effectuer des post-traitements divers : conversions de type (float/double), changement de convention C/Fortran, renumérotation des maillages, accumulation dynamique, *etc*.

RedCORBA contrôle l'accès aux données à envoyer ou à recevoir grâce à des mécanismes de synchronisation inter-threads classiques entre le processus principal de l'application et les threads de communication. Dans RedCORBA, les plages d'accès en lecture/écriture associées à chaque objet complexe (incluant toutes ses séries de données) sont explicitement déclarées via les appels aux fonctions : *unlockReadAccess(), lockReadAccess(), unlockWriteAccess(), lockWriteAccess()* de l'API. Notons que dans EPSN, l'appel à ces fonctions est masqué côté simulation, car les plages d'accès sont implicitement définies sur les tâches grâce à un *contexte d'accès* défini dans le XML. Les contextes *modified* et *protected* verrouillent l'accès en lecture/écriture pendant toute la durée de la tâche. Le contexte *readable* autorise l'accès en lecture et le contexte *writable* celui en écriture, en tenant compte du contexte hérité des tâches mères. Par défaut, aucun accès n'est autorisé pour aucune donnée dans le MHT. De manière générale, il faut souligner que la définition d'une plage d'accès en écriture n'interdit pas dans notre modèle la définition d'une plage d'accès en lecture au même endroit ! La cohérence des accès aux données est en fait maintenue dynamiquement au moment des transferts en interdisant la concurrence des accès contradictoires (deux écritures par exemple, une lecture et une écriture). Cela implique d'utiliser pour chaque donnée un mécanisme de *fenêtre d'envoi et de réception* pour protéger ces transactions. Cette approche nous permet (côté simulation) de maximiser les plages d'accès en lecture qui ne sont pas inutilement coupées par des plages d'accès en écriture.

[4]Notons que les méthodes CORBA traditionnelles sont *two-way*, ce qui signifie qu'elles envoient des arguments en entrée et attendent des résultats en sortie, de manière bloquante.

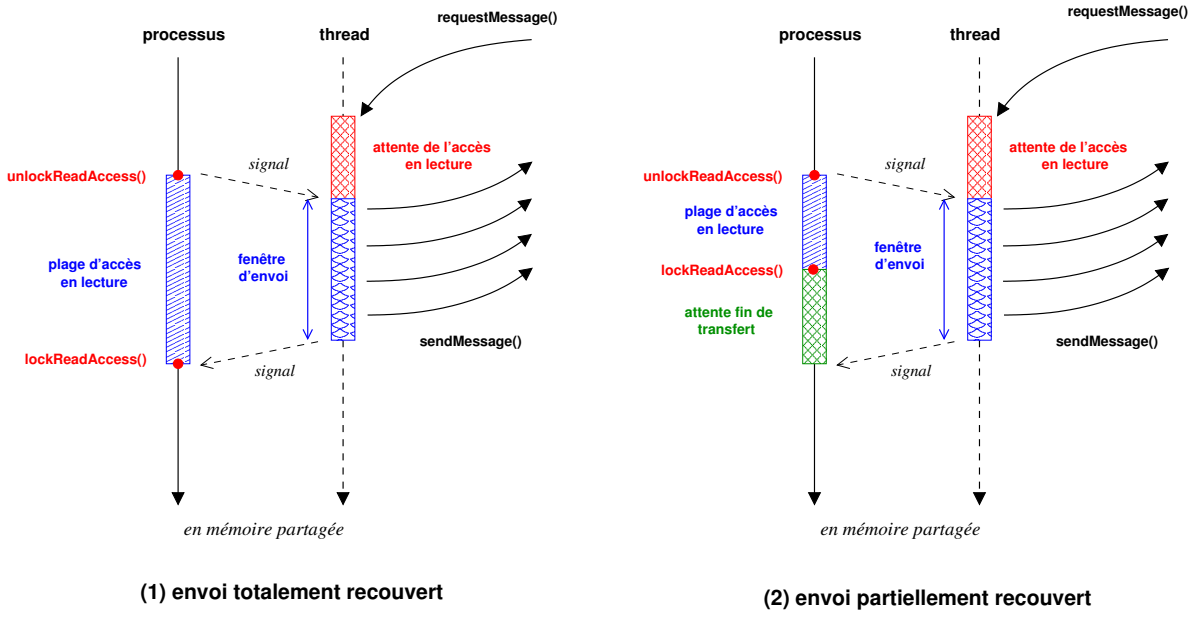


FIG. 6.13 – Mécanismes de synchronisation utilisés pour garantir la cohérence des accès à l'envoi : (1) envoi partiellement recouvert ou (2) totalement recouvert.

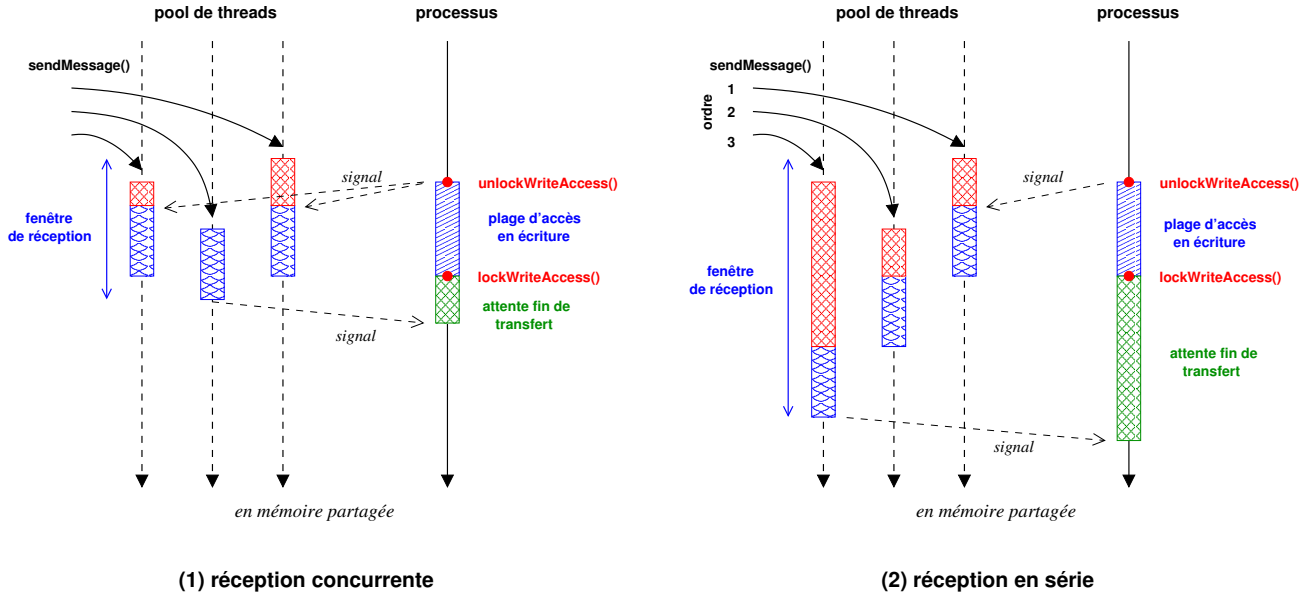


FIG. 6.14 – Mécanismes de synchronisation utilisés pour garantir la cohérence des accès à la réception : (1) réception concurrente ou (2) réception en série.

La cohérence des accès avec les fenêtres repose sur deux règles simples : (a) une fenêtre d'envoi peut être ouverte si la plage d'accès en lecture est ouverte et si il n'y a aucune autre fenêtre de réception ouverte ; (b) une fenêtre de réception peut être ouverte si la plage d'accès en écriture est ouverte et si il n'y a aucune autre fenêtre d'envoi ou de réception ouverte. Lorsqu'une fenêtre d'envoi ou de réception n'est pas prête, elle se met dans l'attente d'un accès et sera réveillé par l'envoi d'un *signal* inter-threads à la fin de la fenêtre précédente ou à l'ouverture de la plage d'accès souhaitée. La programmation d'une fenêtre d'envoi est relativement simple et consiste simplement à verrouiller la fenêtre durant la boucle d'envoi des messages. L'ordre des messages envoyés par RedCORBA respecte l'ordonnancement en *round-robin* calculé par RedSYM. La figure 6.13 met en évidence les mécanismes de synchronisation basés sur la fenêtre d'envoi dans deux cas : (a) le cas où la plage d'accès est suffisamment longue pour recouvrir totalement l'envoi, et (b) le cas inverse où ce recouvrement est partiel. Pour la réception, les choses sont un peu plus complexes car il faut coordonner la réception de plusieurs messages (à destination d'un même port) dans la même fenêtre. Ces messages sont reçus par le pool de threads CORBA de manière asynchrone et concurrente. Nous distinguons dans RedCORBA deux stratégies de réception différentes : *en concurrence* et *en série* (Fig. 6.14). Dans la première stratégie tous les messages associés à une fenêtre sont reçus de manière concurrente, ce qui suppose que l'emplacement en mémoire des données reçues est connu par avance comme c'est le cas pour des objets statiques (i.e. les grilles). Pour des objets dynamiques forts, nous utilisons la stratégie de réception des messages *en série*, excluant toute concurrence lors de la réception comme il est nécessaire pour effectuer l'accumulation des messages selon le mécanisme de *réception dynamique* (cf. Sec. 5.5.3). De plus, nous imposons un ordre de réception des messages afin de garantir que les données reçues ne seront pas permutées d'un transfert à l'autre ou simplement d'une série de données à l'autre au sein d'un même objet complexe. L'ordre de réception des messages est établi en prenant en compte l'ordonnancement en *round-robin* calculé par RedSYM, afin de limiter les latences et la consommation mémoire. Les fenêtres d'envoi et de réception que nous venons de présenter sont des mécanismes strictement locaux aux ports, ce qui laisse supposer que plusieurs fenêtres concurrentes peuvent s'entrelacer entre les différents ports. Cela peut poser des difficultés par exemple dans le cas où deux clients effectuent des requêtes *put* concurrentes vers une même simulation. En pratique, ce cas est simplement interdit au niveau du proxy de RedCORBA ou d'EPSN, qui attend la fin de la requête *put* courante avant d'autoriser une autre requête sur les mêmes données.

6.4 Clients de visualisation et d'interaction

Une fois l'instrumentation d'une simulation réalisée, l'utilisateur se sert d'un client de pilotage pour interagir avec une simulation distante. Les clients de pilotage sont le plus souvent des interfaces graphiques (GUI) « clef-en-main » dont le rôle est de faciliter l'interaction avec les simulations. Ces clients intègrent typiquement des fonctionnalités de visualisation (en ligne) pour suivre en temps réel l'évolution des calculs et permettre de piloter le déroulement des calculs en modifiant certains paramètres ou données du modèle et en déclenchant des actions dans le code. EPSN propose trois stratégies complémentaires pour interagir avec une simulation pilotable.

Client spécifique. La première stratégie consiste à développer son propre client de pilotage à partir de l'API *front-end* d'EPSN (uniquement disponible en C++). Cette approche demande un coût de développement relativement important, mais offre aussi une plus grande liberté pour définir un client de pilotage dédié à une simulation particulière. En outre, cette approche permet de réutiliser des codes de visualisation déjà existants en intégrant EPSN de manière symétrique à l'instrumentation qui est faite côté simulation. Nous avons validé cette approche avec des applications diverses écrites en OpenGL/GLUT et en VTK.

Le client générique Simone. Une autre stratégie consiste à utiliser le client générique *Simone* (Simulation Monitoring for EPSN) fourni avec la plate-forme EPSN (Fig. 6.16). Simone est un client de pilotage séquentiel, entièrement développé en C++ à partir de l'API *front-end* d'EPSN et possédant une interface graphique (GUI) en QT. Cet outil permet de se connecter à une ou plusieurs simulations simultanément et de les piloter de manière tout à fait générique. Simone offre aux utilisateurs la plupart des fonctionnalités de la plate-forme EPSN : contrôle du flot d'exécution, consultation des données au travers de simples « feuilles de données », modification des paramètres et des données, déclenchement d'actions, *etc*. Simone inclut

également des « plug-ins » VTK pour la visualisation en ligne des différentes classes d'objets complexes. Elle inclut également un plug-in de visualisation QWT pour le suivi temporel de paramètres scalaires (e.g. minimum, maximum d'une série). En outre, Simone offre à l'utilisateur certaines facilités comme l'affichage du MHT et de la date courante, une gestion avancée des requêtes (contrôle de la fréquence, annulation, *etc.*). Finalement, elle permet d'afficher des rapports de *benchmarks*, détaillant le temps moyen passé dans les tâches du MHT.

Client semi-générique. Une dernière stratégie, dite semi-générique, consiste à intégrer EPSN dans un système de visualisation programmable comme AVS/Express [2] ou Paraview [25]. Cette approche repose sur la possibilité qu'offre ces environnements d'ajouter de nouveaux modules. Les modules EPSN doivent permettre de se connecter à une simulation distante, de contrôler son exécution et d'acquérir les séries de données associées aux objets complexes de la simulation. L'utilisateur peut ensuite connecter un module d'acquisition des données avec d'autres modules de traitement et d'affichage disponibles dans le système de visualisation. Nous avons réalisé un prototype de client semi-générique avec AVS/Express, un outil très utilisé pour le post-traitement des données scientifiques. La figure 6.15 représente notre prototype avec à gauche le panneau de connexion et de contrôle de la simulation, et à droite la zone d'affichage des données extraites de la simulation FluidBox [159]. Il s'agit d'un maillage non structuré modélisant l'écoulement de l'air autour du cockpit d'une navette pénétrant ici l'atmosphère.

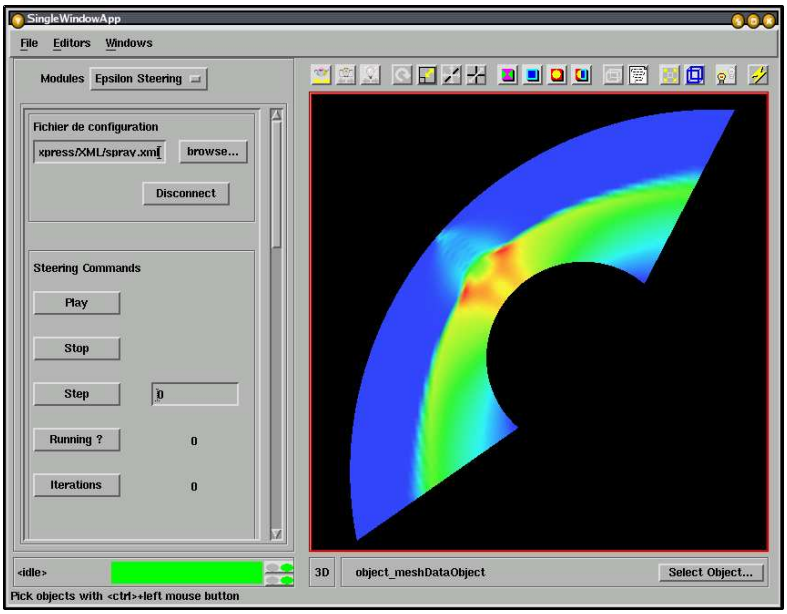


FIG. 6.15 – Pilotage d'un code de mécanique des fluides (FluidBox) dans AVS/Express.

Nous travaillons actuellement à l'intégration d'EPSN dans le système de visualisation Paraview, basé sur le framework VTK. L'avantage de Paraview sur AVS/Express est qu'il est non commercial (*open source*) et qu'il permet d'effectuer de la visualisation parallèle. L'intégration dans Paraview passe par la définition de modules sources VTK réutilisables. Dans les sections suivantes, nous allons détailler la programmation d'un client de visualisation VTK pour EPSN avec ces modules sources. Nous consacrerons également une section à la problématique de la visualisation parallèle dans VTK. Pour plus d'informations concernant VTK et les techniques de la visualisation parallèle, nous renvoyons le lecteur à la section 2.3, ainsi qu'à l'ouvrage de référence sur VTK [191].

6.4.1 Programmation d'un client de visualisation VTK pour EPSN

D'une manière générale, la programmation des clients de pilotage pour la *visualisation en ligne* est un exercice relativement complexe. Cela tient au fait que les systèmes de visualisation ne sont pas initialement prévus pour accepter la modification dynamique des données à l'entrée du pipeline, c'est-à-dire au niveau des

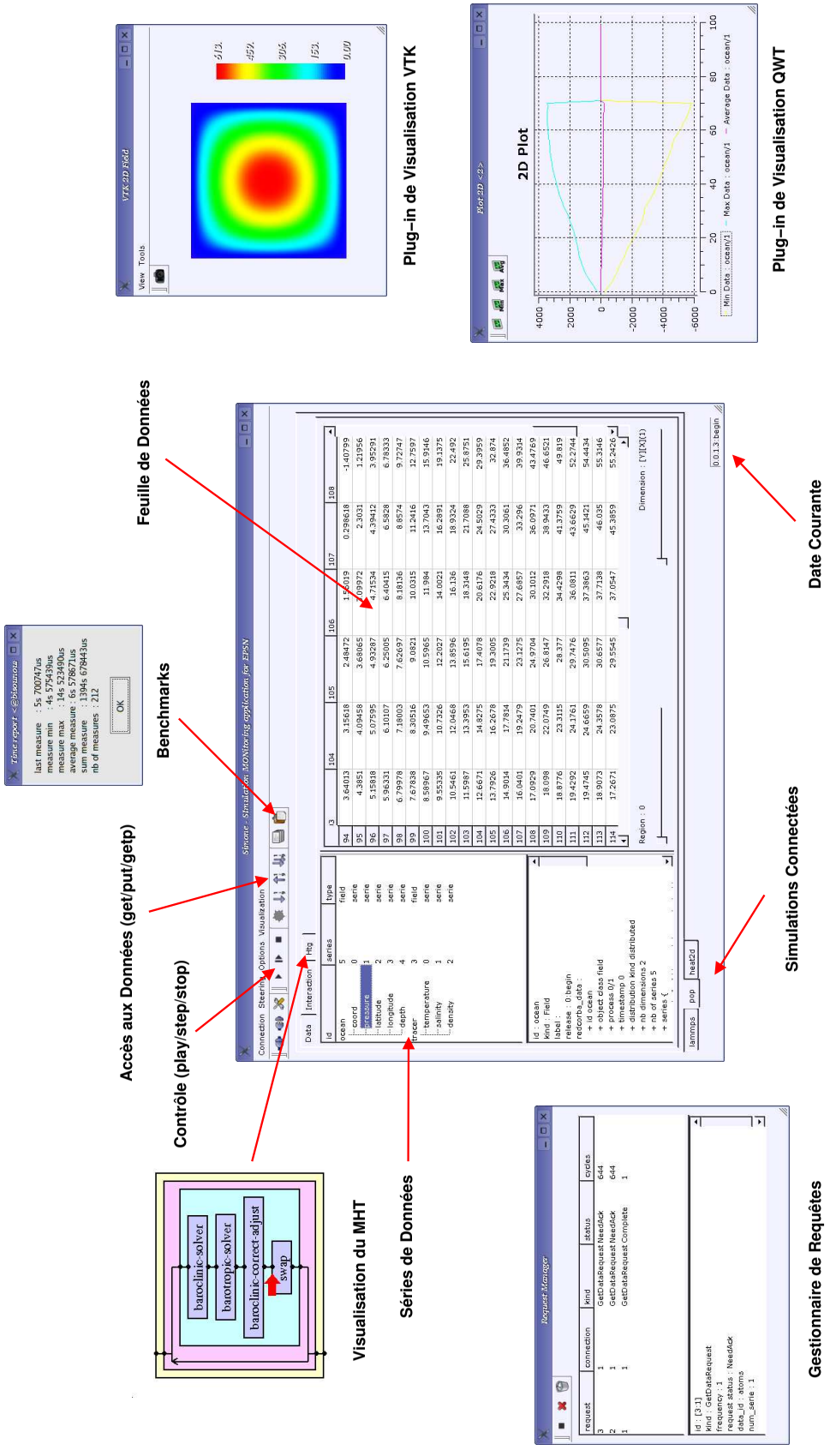


FIG. 6.16 – Le client générique Simone®.

modules *source*. En effet, la plupart de ces systèmes fonctionnent selon un boucle d'évènements « fermée » commandant la mise à jour de l'affichage en réponse aux interactions clavier & souris de l'utilisateur. La seule opportunité pour un programmeur de modifier dynamiquement la scène est d'utiliser un évènement en temps (*timer*) produit périodiquement par le système de visualisation. Cet évènement se traduit par l'appel d'une fonction *callback* que le programmeur peut définir. Une autre solution consisterait à déclencher directement le rendu de la scène au moment de la réception des données dans le thread CORBA, ce qui améliorerait la réactivité du système. Malheureusement, les systèmes de visualisation ne sont généralement pas *thread-safe*, ce qui interdit d'emblée cette solution. Nous nous contenterons donc d'utiliser le *timer* de la boucle d'évènements du système de visualisation.

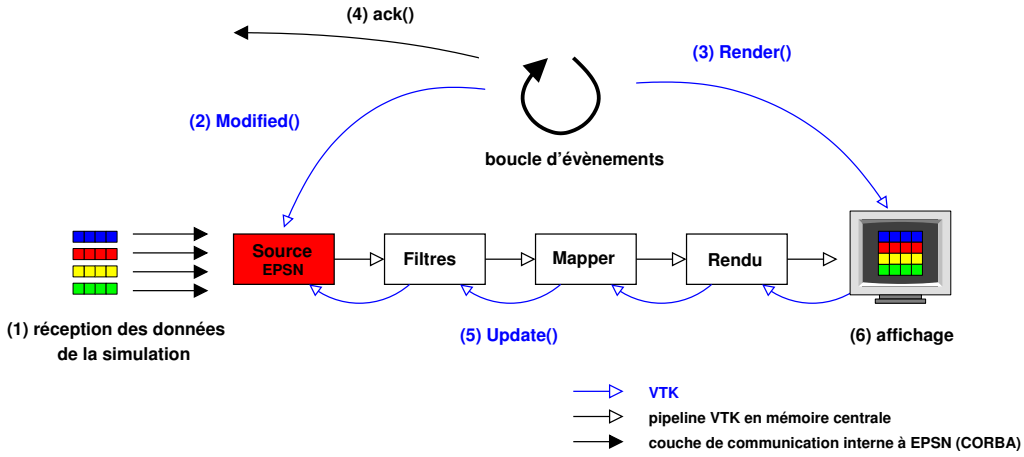


FIG. 6.17 – Intégration d'une source EPSN dans le pipeline de VTK et gestion des évènements en temps pour commander la mise à jour de l'affichage à la réception d'une nouvelle version des données (requête *getp*).

La programmation d'un client VTK pour EPSN repose tout d'abord sur l'intégration d'une *source EPSN* à l'entrée du pipeline de VTK. Cette source sert à recevoir les données de la simulation et à les traduire en un objet VTK compréhensible par la suite du pipeline. Nous reviendrons plus en détails sur les sources EPSN dans la section suivante. Afin de visualiser en ligne (requête *getp*) la nouvelle version des données reçues par la source EPSN, il faut respecter quelques étapes élémentaires représentées sur la figure 6.17. A chaque évènement en temps, la fonction *callback* du *timer* doit commencer par tester si une nouvelle version des données a été reçue (fonction *test()* de l'API EPSN). Dans ce cas précis, il faut marquer explicitement les données de la source EPSN comme étant modifié (*Modified()*). On peut alors commander la mise à jour du pipeline avec la fonction *Render()*, ce qui entraîne automatiquement la remontée du pipeline et l'affichage de la nouvelle version des données. Après quoi, il ne faut pas oublier d'acquitter la simulation (*ack()*). Cet acquittement permet de garantir que les données reçues seront traitées avant la réception d'une nouvelle version. Ce mécanisme permet de réguler le couplage entre la simulation et le code de visualisation. Si la simulation est « plus rapide » que la visualisation, alors les ports de la simulation vont automatiquement se bloquer (à la fin de la plage d'envoi) dans l'attente de l'acquittement du client. Ce mécanisme ralentit la simulation, mais évite l'engorgement du client avec de multiples messages qu'il n'aurait pas le temps de traiter. Afin de limiter le surcoût (en moyenne) induit par EPSN sur la simulation, on peut simplement baisser la fréquence d'envoi associée à la requête *getp*. A l'inverse, si la simulation est « plus lente » que le code de visualisation, alors il est possible d'anticiper l'acquittement en le déclenchant avant le calcul du rendu. Dans ce cas, il est nécessaire de verrouiller l'accès en écriture autour de la fonction *Render()* afin de garantir la cohérence spatiale des traitements (car un nouveau message pourrait être reçu à ce moment). Cette approche évite de ralentir la simulation inutilement en recouvrant le calcul du rendu par les calculs de la simulation.

6.4.2 Les sources EPSN pour VTK

Afin de faciliter la programmation des clients de visualisation VTK, nous avons développé un ensemble de *sources* permettant d'intégrer très simplement EPSN à l'entrée du *pipeline* de VTK. Une source EPSN permet simplement de traduire un objet complexe de la simulation en objet VTK visualisable par le client. Notons que les sources EPSN sont *intrinsèquement parallèles* et peuvent donc servir à la fois dans le contexte classique de la visualisation séquentielle, mais aussi dans le contexte du parallélisme de données. Dans ce dernier cas, le pipeline de visualisation est entièrement répliqué sur tous les processus et les données sont distribuées entre les différentes sources EPSN. Une source VTK est simplement une classe C++ héritant de la classe `vtkSource` et implantant la méthode abstraite *RequestData()* qui sera invoquée à chaque demande de mise à jour du pipeline VTK (*Update()*). Notons qu'à chaque classe d'objet complexe RedSYM va correspondre un type de source différent dans VTK.

La construction d'une source EPSN se décompose en deux étapes principales. Notons que ces étapes sont entièrement transparentes pour l'utilisateur final qui se contente de fournir l'identifiant de l'objet distant qu'il souhaite visualiser. La première étape consiste à construire un objet RedSYM local au client, prêt à recevoir les données de la simulation distante. La création de cet objet repose sur la connaissance du descripteur parallèle relatif à l'objet côté simulation. Ce descripteur est accessible via la fonction *getRemoteDataDesc()* de l'API EPSN. Dans le cas séquentiel, le choix de la distribution cliente est relativement trivial. Nous discuterons du choix la distribution cliente dans le cas parallèle à la section suivante. La deuxième étape consiste à traduire l'objet RedSYM créé en un objet VTK compréhensible par le pipeline. Il existe une correspondance relativement naturelle entre les objets RedSYM et VTK que nous résumons dans le tableau ci-dessous. Nous soulignons que cette traduction est rendue possible grâce aux informations supplémentaires que nous encapsulons dans notre modèle de description (cf. Sec. 5.4.3) concernant en particulier la géométrie des objets complexes : le pas d'une grille régulière, la série des coordonnées pour les particules ou les grilles irrégulières, le type des cellules pour les maillages non structurés, *etc*.

Objet	**RedSYM**	**VTK**
Série de données	`AbstractBuffer`	`vtkDataArray`
Grille régulière (2D ou 3D)	`Grid`	`vtkImageData`
Grille irrégulière (2D ou 3D)	`Grid`	`vtkStructuredGrid`
Particules	`Particles`	`vtkPoints`
Boîtes d'atomes	`AtomBox`	`vtkPoints`
Maillage non structuré (cellules 2D)	`Mesh`	`vtkPolyData`
Maillage non structuré (cellules 3D)	`Mesh`	`vtkUnstructuredGrid`

TAB. 6.1 – Tableau des correspondances entre RedSYM et VTK.

Le choix de la visualisation qui sera faite pour ces objets dépend entièrement de la suite du pipeline. VTK offre une très large palette de *filtres* et de *mappers* pour traiter et afficher les données de la source : plan de coupe, calcul d'iso-surface, rendu polygonal, rendu volumique, *etc*. A terme, l'objectif est d'intégrer nos sources dans Paraview [25], ce qui permettra également d'effectuer de la visualisation parallèle. Avec un tel outil, l'utilisateur sera libre de programmer sa propre application de visualisation en interconnectant « à la souris » les sources EPSN avec les filtres de son choix disponibles dans Paraview.

6.4.3 Visualisation parallèle avec VTK

La visualisation parallèle dans VTK [128] utilise un modèle maître/esclaves classique. Le maître est responsable de l'affichage et de l'interaction clavier/souris qui s'effectue de manière centralisée. Chaque esclave est un processus qui réplique entièrement le pipeline VTK (source, filtre, mapper, rendu) sur des données distribuées en mémoire (modèle SPMD) et calcule un rendu partiel de la scène. Les images produites par les différents escalves sont ensuite combinées (algorithme *sort-last*) pour produire l'affichage final sur le maître. La mise à jour de la scène est explicitement commandé par le maître qui envoie un ordre *Update()* à tous les

escalves. Cette ordre a pour effet de commander la mise à jour du pipeline. Les communications entre le maître et les esclaves sont basés sur MPI. Notons que le maître joue fréquemment le rôle d'un esclave en calculant lui-même une partie du rendu. Dans l'approche *sort-last* (cf. Sec. 2.3.2), la minimisation du temps de calcul va nécessiter de distribuer les données de manière équilibrée entre les esclaves. Notons que le respect d'une certaine localité des données n'a *a priori* pas d'influence sur le temps de calcul d'un rendu partiel, mais permet lors de la composition de réduire « au mieux » la taille des images échangées[5].

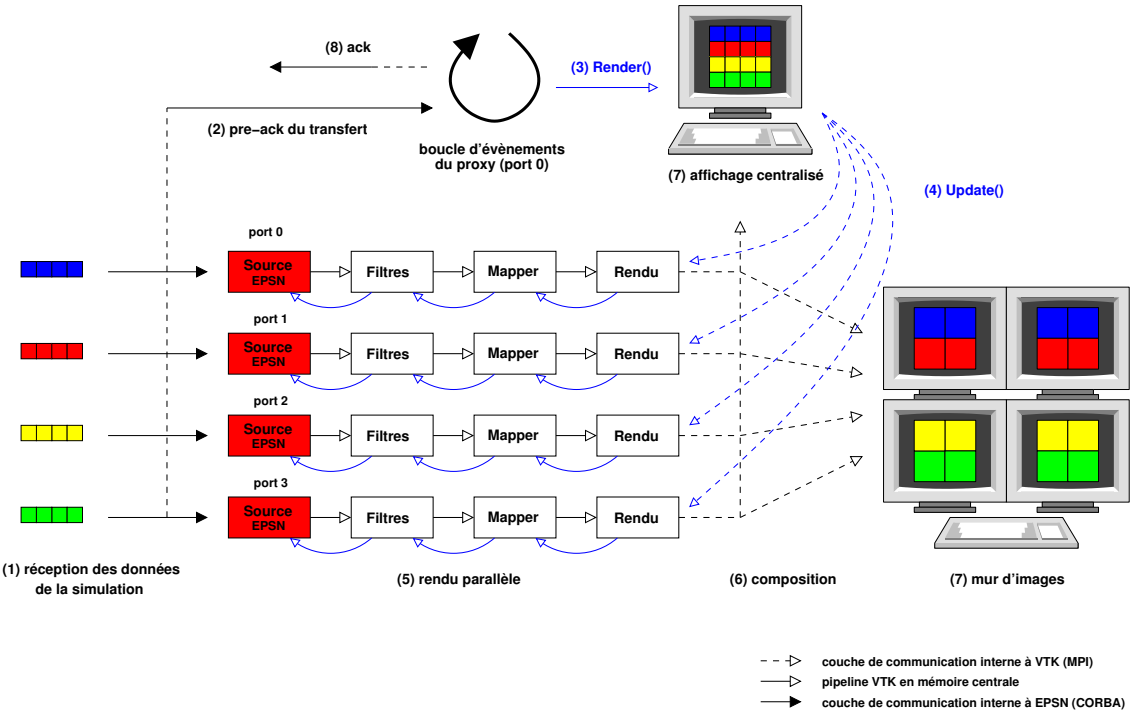


FIG. 6.18 – EPSN et la visualisation parallèle dans VTK.

Comme nous l'avons dit précédemment, les sources EPSN sont intrinsèquement parallèles. En pratique, cela signifie qu'il faut choisir une distribution des objets RedSYM côté client qui sera affectée aux sources. Comme plusieurs choix sont évidemment possibles, nous avons sélectionné une distribution par défaut qui pourra éventuellement être modifiée par l'utilisateur en fonction de ses besoins. Pour les ensembles de particules et les maillages non structurés, nous utilisons la stratégie de placement *split/merge* qui procure un bonne équilibrage des données et tend à conserver la localité des données présente dans la simulation. Dans le cas des grilles et des boîtes d'atomes, le choix de la distribution que nous faisons diffère selon que la distribution par bloc est dense ou creuse. Dans le premier cas, nous utilisons une décomposition classique du domaine global de type bloc 1D, 2D ou 3D en fonction du nombre de processeurs côté client (N) et de la dimension des données (2D ou 3D). Dans le cas creux, nous utilisons actuellement la stratégie de placement des régions sans découpage (i.e. des blocs), qui procurera un bon équilibrage de la charge si le nombre des blocs est relativement grand comparé à N (cf. Sec 5.6.3).

La figure 6.18 résume les étapes principales suivies pour effectuer de la visualisation en ligne dans le cas parallèle. A la réception des données sur les ports (côté client), l'envoi d'un pré-acquittement au proxy client marque la fin du transfert, ce qui permet de déclencher un ordre de rendu parallèle avant de transmettre un acquittement global à la simulation. En plus de la stratégie de composition classique vers un seul écran, nous avons également expérimenté les possibilités d'effectuer une composition en parallèle vers un *mur d'images*.

[5]En déplaçant la caméra, il est toujours possible de mettre en défaut le choix d'une certaine distribution des données. Il suffit simplement de placer les données d'un seul esclave (ou d'un sous-ensemble) dans le champs de vision de la caméra . Dans ce cas, les autres escalves ne participent pas au rendu de la scène puisque leurs données sont hors caméra (*clipping*).

Pour ce faire, nous avons utilisé la bibliothèque ICE-T (Image Composition Engine for Tiles) développée au laboratoire de Sandia par *Moreland et al.* [147, 148]. Cette bibliothèque a été récemment intégrée dans VTK & Paraview et son utilisation avec EPSN est entièrement transparente. La figure 6.19 illustre nos expériences de visualisation parallèle avec VTK et ICE-T. Il s'agit du calcul d'une iso-surface en parallèle sur les données distribuées d'une IRM (150 tranches de taille 255×255). EPSN est simplement utilisé dans cette expérience pour changer la distribution initiale des données (en tranches, bloc 1D) en une distribution de type bloc 3D.

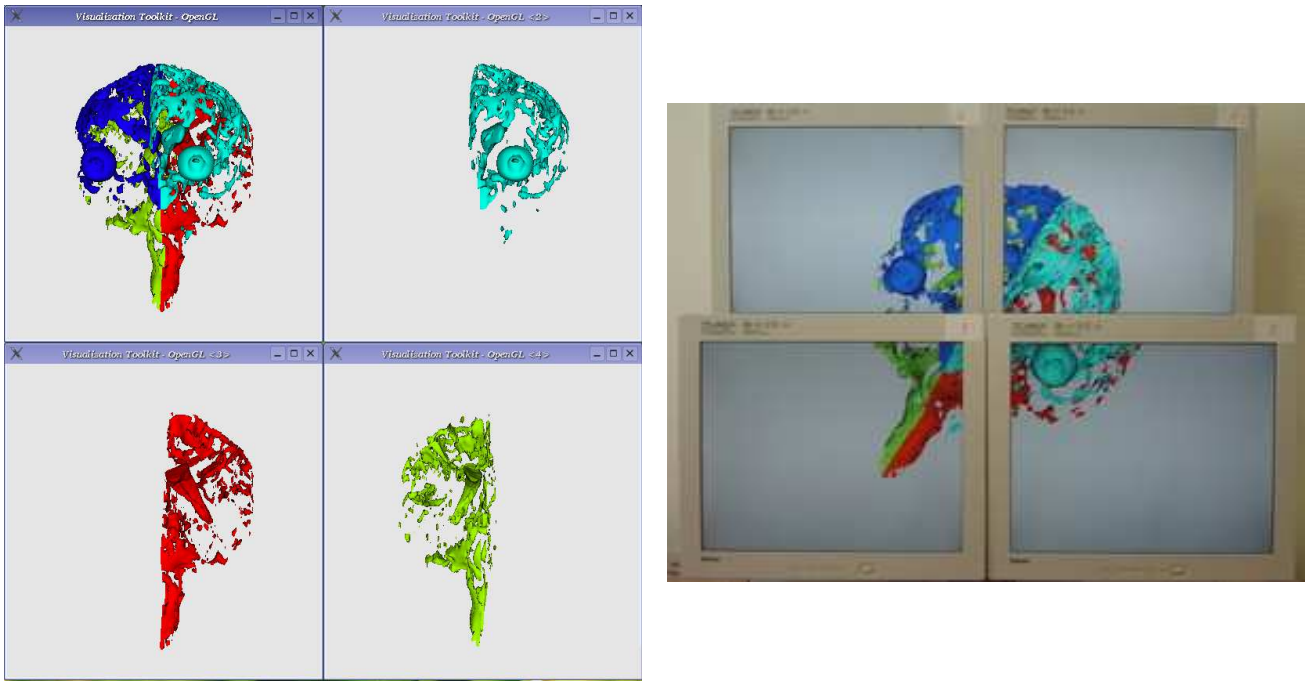


FIG. 6.19 – Calcul d'une iso-surface en parallèle et rendu parallèle avec VTK : à gauche, approche classique avec affichage centralisé (montrant les différents rendus partiels) ; à droite, affichage avec ICE-T sur un mur d'images « artisanal ». Les différentes couleurs représentent la distribution des données choisies (en bloc 3D) pour les 4 processus de visualisation.

6.5 Conclusion

La plate-forme EPSN fournit une approche générique, flexible et dynamique pour le pilotage des simulations. Elle propose une architecure originale basé sur des choix de technologies résolument modernes : XML pour la description de la simulation, CORBA pour la couche de communication et VTK pour la visualisation. L'architecture de la plate-forme EPSN respecte les deux modèles définis aux chapitres précédents concernant le pilotage fin des simulations numériques et la redistribution de données complexes. Nous avons choisi de séparer la bibliothèque de redistribution de la plate-forme EPSN afin qu'elle puisse être réutilisable dans d'autres contexte, notamment dans le contexte du couplage de codes. A ce titre, nous collaborons avec le projet GridCCM pour intégrer RedSYM dans les objets parallèles CORBA (PaCO++ [180]). D'une manière générale, RedSYM et EPSN ont été conçues pour être évolutifs et servir de support à de futures recherches, notamment concernant la mise en œuvre de nouveaux algorithmes de redistribution, mais aussi pour expérimenter des techniques d'interaction de pilotage plus complexe basés sur l'utilisation de *widget* 3D dans VTK. Nous verrons au chapitre suivant des exemples d'utilisation de cette plate-forme mettant en évidence ces différentes potentialités. Au chapitre final (chapitre 8), nous donnerons quelques perspectives d'évolutions pour la plate-forme EPSN, concernant notamment l'optimisation de la performance des transferts en s'orientant vers une couche de communication hybride (CORBA + MPI-2).

Chapitre 7

Résultats & Applications

Sommaire

7.1 Préambule

Avant de discuter plus en avant des performances de la plate-forme EPSN, nous souhaitons présenter dans ce préambule les configurations matérielles ayant servi à réaliser nos expériences. La figure 7.1 représente les deux principaux clusters utilisés, le cluster du site *Grid'5000* [14] à Bordeaux et le cluster « local » de visualisation configuré en mur d'images 2×2. Dans les expériences de pilotage en vraie grandeur que nous décrirons à la section 7.4, le site *Grid'5000* Bordeaux jouera typiquement le rôle du cluster de calcul sur lequel une simulation parallèle s'exécute, tandis que le cluster local au LaBRI sera plus particulièrement dédié à la visualisation interactive et au pilotage.

Les deux premières sections seront consacrées à l'évaluation de la plate-forme EPSN et des bibliothèques de redistribution RedSYM & RedCORBA. Les expériences seront alors réalisées exclusivement sur le cluster *Grid'5000*, car nous commençerons par analyser les performances de la plate-forme indépendamment

de la visualisation. Nous allons essentiellement nous intéresser aux performances de la requête *getp* qui nous permet de commander les transferts périodiques nécessaires à la visualisation en ligne. L'impact des traitements de visualisation sur la plate-forme EPSN pour des applications réelles sera étudié dans la section 7.4.

Le cluster du site *Grid'5000* à Bordeaux se compose de cinquante bi-Opterons (AMD, 64 bits) cadencés à 2.2 GHz (2 Go de mémoire partagée). Le cluster de visualisation, quant à lui, est formé de 4 bi-Xeons (Intel, 32 bits) cadençés à 2.8 GHz avec 2 Go de mémoire partagée. Ce cluster est équipé de cartes graphiques NVidia GeForce 4 (AGP $8\times$) avec 128 Mo de RAM. Chaque PC possède un écran plat 19 pouces configuré avec une résolution 1280×1024, ce qui fait une résolution totale de 2560×2048 pour notre mur d'images (Fig. 7.1). Nous utilisons *Synergy* pour mettre en partage un clavier et une souris entre ces 4 PCs. Les PCs de chaque cluster sont interconnectés par un réseau Giga-Ethernet grâce à un *switch* (bande-passante théorique à 128 Mo/s). La connexion entre ces deux clusters est réalisée par un VLAN (Virtual Local Area Network) reposant sur un lien unique de type Giga-Ethernet. Ces deux clusters possèdent un système d'exploitation de type Linux (noyau 2.6). Du point de vue logiciel, nous utilisons le compilateur C/C++ GCC 4 et le compilateur Fortran G95. Nous utilisons également LAM/MPI version 7.1 ainsi que OmniORB [23] version 4.0.6, une implantation de CORBA réputée pour ses bonnes performances.

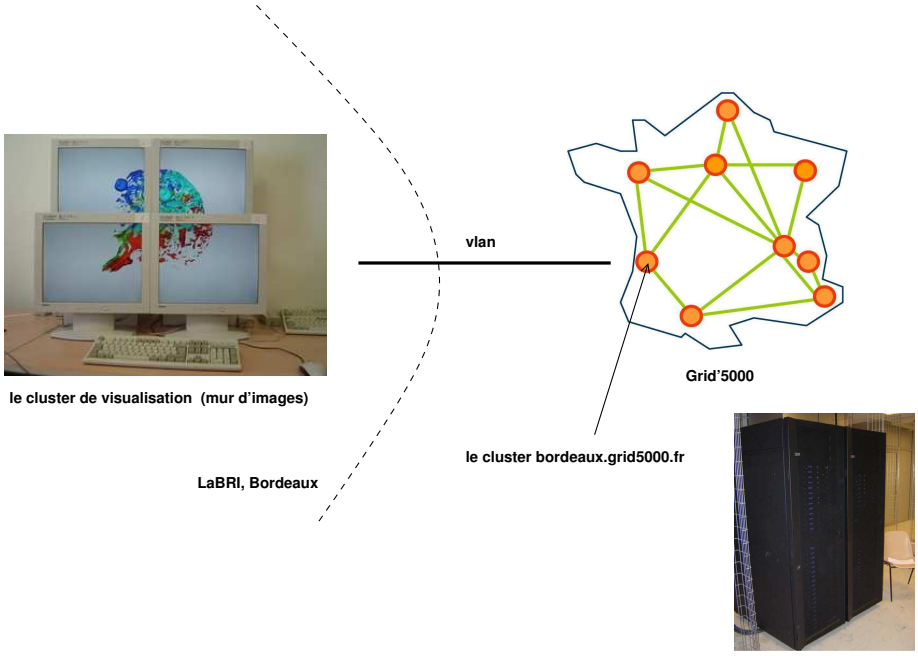


FIG. 7.1 – Présentation des clusters ayant servis à nos expérimentations.

Pour finir, ce préambule, nous souhaitons présenter quelques courbes de référence utiles à l'analyse des résultats qui suivront. La figure 7.2 présente les débits (en Mo/s) mesurés expérimentalement en point-à-point sur le cluster *Grid'5000* Bordeaux pour différentes tailles de données allant de quelques octets jusqu'à plus de 100 Mo. Notons que les débits mesurés sur le VLAN sont sensiblement identiques. D'une manière générale, nous nous intéressons au comportement de la plate-forme EPSN pour des volumes de données supérieurs au kilo-octet, ce qui correspond à son utilisation normale dans le contexte de la visualisation en ligne. Nous avons mesuré un débit maximal pour OmniORB4 à 110 Mo/s (latence 107 μs) en point-à-point sur ce cluster contre 112 Mo/s pour MPI/LAM7 (latence 103 μs). On peut ainsi remarquer que OmniORB4 est pratiquement aussi performant que LAM/MPI, ce qui vient du fait qu'il utilise une stratégie zéro-copie qui le rend très efficace. Nous avons également ajouté la courbe de référence « OmniORB4 avec copie » à la réception, ce qui correspond à l'utilisation normale que nous en faisons dans RedCORBA et EPSN. En effet, comme nous l'avons vu à la section 6.3.3, à cause de certaine limitation de CORBA, nous payons le prix d'une recopie mémoire dans RedCORBA à la réception des messages. Dans ce cas, le débit maximal atteint n'est plus que de 97.5 Mo/s (-13%), ce qui va donc correspondre à notre vraie courbe de référence, même si par la suite nous comparerons

systématiquement nos résultats au débit obtenu avec MPI (notre « étalon réseau » en quelque sorte).

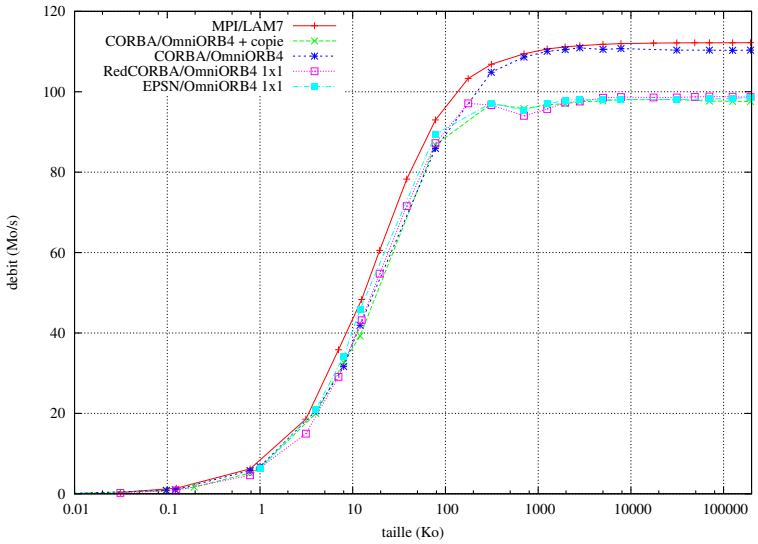


FIG. 7.2 – Courbes de référence CORBA, MPI, RedCORBA et EPSN en 1×1.

Au risque d'anticiper un peu, nous avons ajouté les courbes de débit de RedCORBA et d'EPSN pour la configuration 1×1 (i.e. une simulation séquentielle et un client séquentiel distant). RedCORBA commande le transfert à l'aide d'une requête *put* (approche centralisée), ce qui va correspondre à une requête de type *get* dans EPSN. Les courbes ainsi obtenues sont sensiblement identiques à la courbe « OmniORB4 avec copie » et montrent que les surcoûts liés aux traitements des requêtes sont négligeables dans cette configuration[1], du moins pour de gros messages. Les performances sont essentiellement affectées au niveau de la latence qui est alors de $195\ \mu s$ pour RedCORBA et EPSN en 1×1.

7.2 Évaluation de la plate-forme EPSN

Dans cette première série d'expériences, nous évaluons les performances de la plate-forme EPSN par rapport au mécanisme de recouvrement des transferts et de la coordination des requêtes en parallèle. Afin de faciliter l'évaluation de notre plate-forme, nous avons développé une simulation-modèle entièrement paramétrable à laquelle nous connectons un client servant uniquement à soumettre des requêtes et à recevoir des données. Notons que ce client n'effectuera ici aucun post-traitement sur les données reçues.

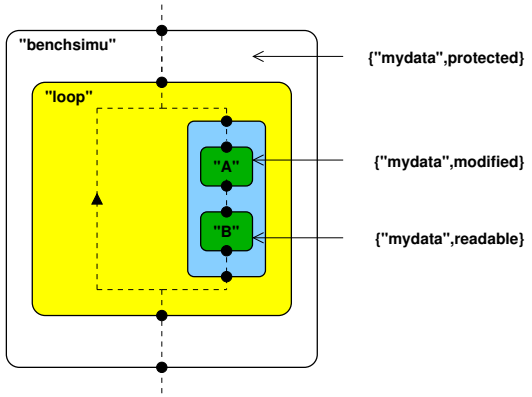


FIG. 7.3 – Le MHT de la simulation test *benchsimu*.

[1]Notons que le proxy est placé dans nos expériences sur le processus de rang 0. Par conséquent, le tranfert d'une requête du proxy vers le port 0 se traduit en CORBA par un simple appel de fonction, ce qui explique qu'il n'y a pas de surcoût réseau à ce niveau.

7.2.1 La simulation de test

Notre simulation test, baptisée *benchsimu*, se présente comme une simple boucle en temps, effectuant à chaque itération deux tâches de calcul, A et B, la première modifiant une certaine donnée *mydata* (contexte *modified*) et l'autre autorisant l'accès en lecture à cette donnée (contexte *readable*). La figure 7.3 représente le MHT relatif à cette simulation (cf. Sec. 4.2.2). En résumé, l'accès à la donnée *mydata* est interdit durant tout le temps de la tâche A et une nouvelle version est produite à l'issue de cette tâche ; la donnée est globalement accessible le reste du temps (tâche B). L'avantage de cette simulation pour les expériences qui vont suivre est qu'elle est entièrement paramétrable, ce qui n'est pas le cas d'une application réelle. Nous énumérons dans le tableau 7.1 les différents paramètres de cette simulation. Outre le nombre de processeurs M de la simulation et N du client, il est possible de régler la taille totale D de la donnée *mydata* (un simple tableau 1D, une grille structurée au sens des objets RedSYM). Nous pouvons également paramétrer le temps total (en ms) d'une itération de la simulation en jouant sur la quantité de calcul effectuées par les tâches A et B[2]. De plus, on pourra régler la période p d'envoi des données pour la requête *getp*, la principale requête que nous allons étudier par la suite. Les expériences concernant la redistribution des données seront étudiées à la section 7.3. Aussi, pour la plupart des expériences que nous allons réaliser dans cette section, nous considérerons le comportement de la plate-forme EPSN pour une simulation séquentielle et un client séquentiel, ce que nous appelons la configuration 1×1 par opposition à la configuration $M \times N$.

paramètres	**description**
M	nombre de processus côté simulation
N	nombre de processus côté client
K	nombre de clients
W	nombre de threads *workers* dans EPSN
T_A	durée de la tâche A (en ms)
T_B	durée de la tâche B (en ms)
p	période d'envoi associée à la requête *getp* ($p = 0$ si pas d'envoi)
D	taille totale des données distribuées (en Ko)

TAB. 7.1 – Les paramètres de nos expériences avec la simulation *benchsimu*.

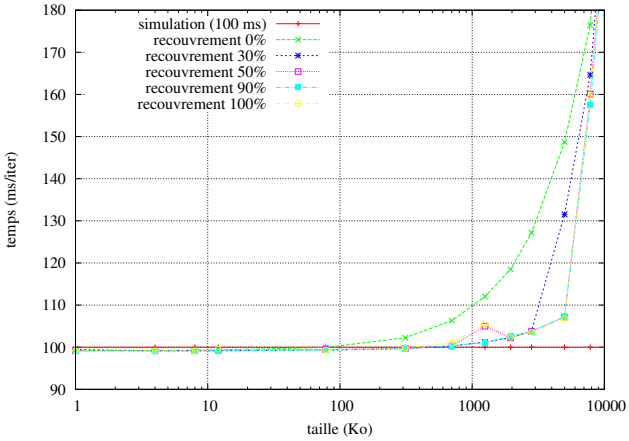
Il faut noter que toutes nos expériences utilisent des machines bi-processeurs. D'une manière générale, nous avons choisi d'exécuter un seul processus de la simulation (ou du client) par machine afin de faciliter l'analyse des débits réseaux. Ainsi, chaque processus dispose d'une interface réseau propre (i.e. non partagée). Par défaut, ce processus sera placé sur le CPU 0. Selon les expériences, les threads du pool de workers pourront être placés sur le CPU 0 (le même que la simulation), le CPU 1 ou encore de manière hybride entre les deux CPUs (variable d'environnement `EPSN_AFFINITY_MODE`, cf. Sec. 6.2.3).

7.2.2 Mise en œuvre du recouvrement des transferts

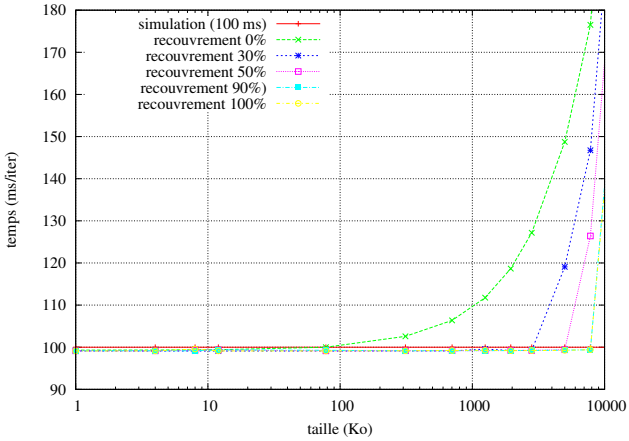
Dans cette section, nous étudions la capacité de la plate-forme EPSN à exploiter des plages d'accès plus ou moins grandes en lecture à des données pour effectuer un recouvrement des communications par les calculs de la simulation. Nous considérons une simulation séquentielle ($M = 1$) et client séquentiel ($N = 1$), effectuant une requête de type *getp* ($p = 1$) sur des données de différentes tailles (en Ko). Nous avons choisi une simulation séquentielle plutôt que parallèle pour cette expérience, car ce mécanisme d'accès aux données est clairement local à chaque processus de la simulation. Pour chaque expérience, nous mesurons le temps moyen d'une itération de la simulation lorsque les transferts se produisent, et nous le comparons au temps de la simulation « à vide » afin d'évaluer le surcoût. Les figures 7.4 et 7.5 illustrent les cas d'une simulation de 100 ms et de 1000 ms par itération. Dans chaque cas, nous faisons varier la durée de la plage d'accès en lecture (T_B) de telle façon que le temps totale d'une itération ($T_A + T_B$) reste constant. Dans la mesure où l'envoi est réalisé par un thread *worker* de manière concurrente à l'exécution de la simulation, le temps T_B correspond aussi à la *plage maximale de recouvrement* dont nous disposons pour tenter de masquer le surcoût des transferts

[2]Pour ce faire, nous évaluons le nombre de fois qu'il faut répéter une certaine expression arithmétique pour obtenir 1 ms de calcul.

avec EPSN. Nous dirons que la plage de recouvrement est de 100% lorsque que les données sont accessibles en lecture tout le temps de l'itération (i.e. T_A=0 ms). Lorsque la plage de recouvrement est de 50%, cela signifie que $T_A = T_B = 50$ ms pour une simulation de 100 ms par itération. Le cas limite, où la plage de recouvrement est de 0%, correspond en fait à l'utilisation d'un point d'envoi pour lequel les transferts sont bloquants sur ce point ($T_B = 0$ ms, surcoût maximal). Par ailleurs, il faut noter que comme nous effectuons une requête de type *getp* qui est permanente (cf. Sec 4.4.1), la planification des envois successifs est faite uniquement la première fois et sera donc négligeable dans nos mesures.

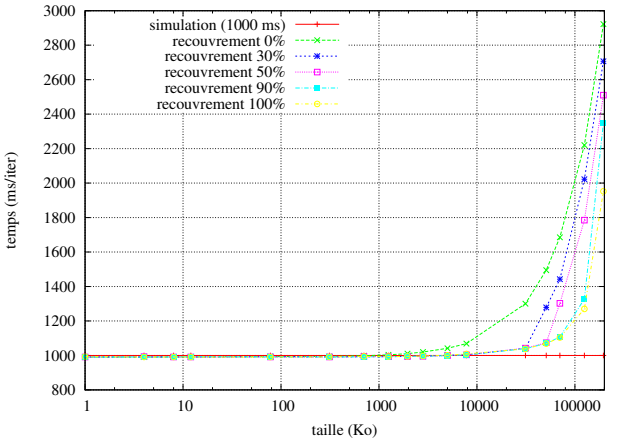


(a) simulation et worker sur le CPU 0.

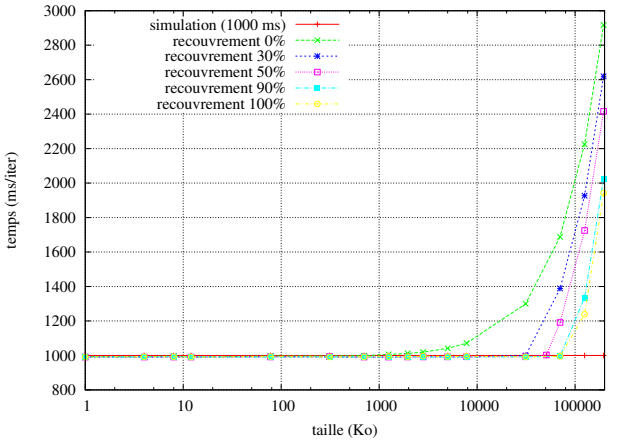


(b) simulation sur le CPU 0 et worker sur le CPU 1.

FIG. 7.4 – Différentes expériences sur le recouvrement dans EPSN en 1×1 pour une simulation de 100 ms/iter. Les différentes courbes correspondent au cas d'une requête *getp* ($p = 1$) pour des plages d'accès en lecture comprises entre 0% et 100% du temps total de l'itération.



(a) simulation et worker sur le CPU 0.



(b) simulation sur le CPU 0 et worker sur le CPU 1.

FIG. 7.5 – Même expérience que ci-dessus mais pour une simulation de 1000 ms/iter.

On constate que l'exploitation d'une plage d'accès en lecture permet de diminuer efficacement le surcoût des transferts, et cela d'autant mieux que cette plage est grande. En deçà d'un certaine taille donnée (dépendant de T_B), le surcoût est quasi-nulle et le recouvrement est alors parfait. Au delà de ce seuil, le temps nécessaire au transfert devient supérieur à T_B et le recouvrement ne sera partiel (cf. Sec. 6.2.3, requête *blocking*). Nous avons également étudié l'impact du placement d'un thread *worker* ($W = 1$) sur le recouvrement des communications. On vérifie expérimentalement que le cas où le thread *worker* est placé sur un CPU différent du processus de la simulation est le cas idéal pour la plate-forme EPSN, comparé au cas où il partage le même CPU que la simulation (Fig. 7.4 (b) et Fig. 7.5 (b)). Nous avons également expérimenté le cas où nous n'attribuons aucune affinité entre les processus/threads et les CPUs. Dans ce cas, l'ordonnanceur du système d'exploitation est libre de placer les threads comme il le souhaite. On observe alors que les mesures oscillent entre les deux modes

précédents avec des surcoût en temps ectopiques lors des migrations du thread worker d'un CPU à l'autre.

7.2.3 Surcoût du système de requêtes dans la phase de coordination

L'expérience qui suit vise à établir le surcoût induit sur les simulations parallèles par l'algorithme de planification pour coordonner le traitement des requêtes (cf. Sec. 4.4.2). Cet algorithme vise à établir une date de traitement commune à tous les processus de la simulation, sans les synchroniser et en évitant au maximum de les ralentir. L'établissement de cette date s'effectue dans un *thread* CORBA de manière concurrente à l'exécution de la simulation et nécessite au proxy de communiquer avec tous les processus. Notre algorithme n'étant pas optimisé à ce jour, le temps absolu qu'il faut pour calculer cette date est clairement linéaire par rapport au nombre M de processus de la simulation, ce que nous vérifions expérimentalement sur la figure 7.6. Nous avons évalué deux implantations différentes de cet algorithme basées sur des requêtes CORBA de type statique (SII) ou dynamique (DII). Même si la seconde stratégie est plus performante que la première ($\sim$ 13 sec. pour 40 processus), elle n'en reste pas moins linéaire. Nous avons présenté à la fin de la section 4.4.2 une optimisation possible de notre algorithme basée sur l'utilisation d'un arbre binaire de communication inspiré de la routine *MPI_Allreduce*.

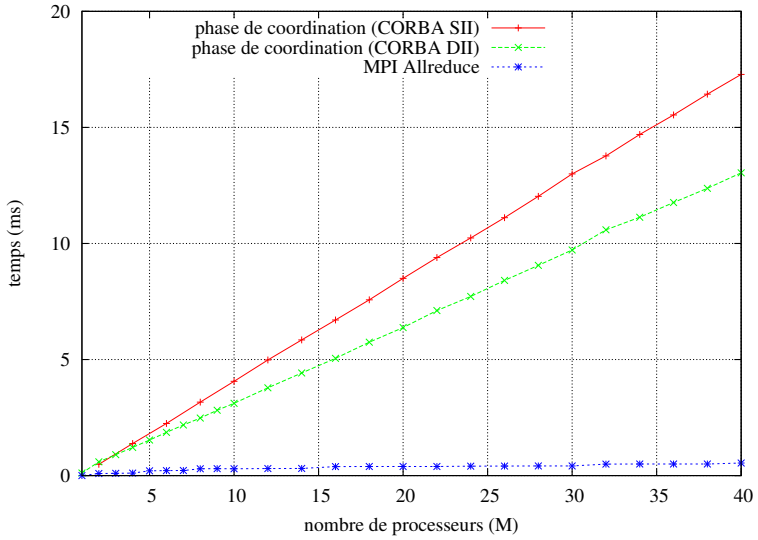


FIG. 7.6 – Surcoût absolu du système de requêtes mesuré en interne dans la plate-forme EPSN pour une requête test n'effectuant aucun traitement.

Pour obtenir ces résultats, nous avons simplement mesuré le temps en interne dans le thread EPSN du proxy, ce qui est clairement indépendant du type de la requête et du client. Notons que ce temps inclut l'envoi de la requête (initialement envoyée par le client) à tous les processus de la simulation, le déclenchement de l'ordre *freeze*, l'établissement de la date de planification sur le proxy et la diffusion de cette date à tous les processus, et enfin le relâchement de l'ordre *freeze*. Nous rappelons que l'ordre *freeze* ne ralentira la simulation que si un processus tente de dépasser le point d'instrumentation suivant, ce qui invaliderait le calcul de la date de planification en cours.

La seconde expérience mesure le temps moyen d'une itération de la simulation lorsqu'on lui envoie des requêtes test (une par itération) n'effectuant aucun traitement et ne demandant pas d'acquittement (Fig. 7.7). Les autres paramètres de l'expérience sont $T_A = T_B$, $W = 1$, $N = 1$ et un placement du thread worker sur le même CPU que le processus de la simulation. Même si le surcoût absolu paraît relativement élevé, nous observons expérimentalement que le *surcoût réel* mesuré sur la simulation *benchsimu* est beaucoup plus faible : quasi-nulle pour une simulation de 1000 ms/iter. et de l'ordre de 3 ms pour une simulation de 100 ms/iter. s'exécutant en parallèle sur 40 processeurs. Cela est dû au fait que la phase de coordination s'effectue de manière concurrente au déroulement de la simulation et que le surcoût absolu induit par cette phase aura d'autant plus de chance

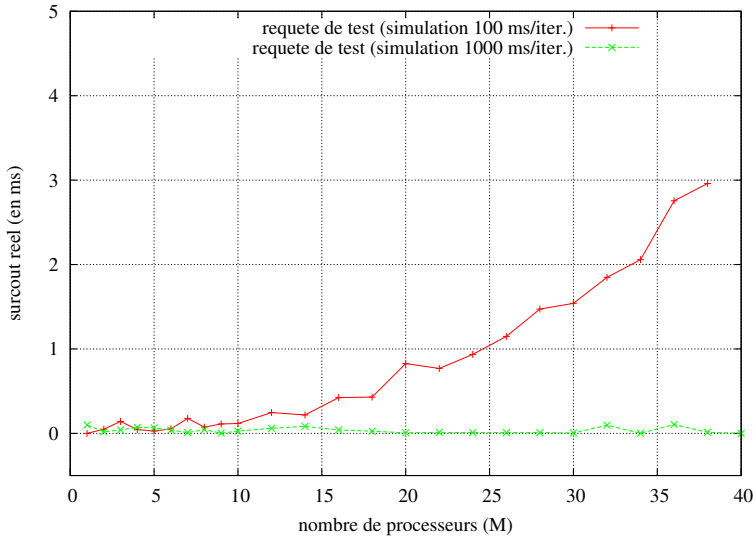


FIG. 7.7 – Surcoût réel (en ms) du système de requêtes d'EPSN mesuré pour une simulation de 100 ms/iter. et de 1000 ms/iter.

d'être recouvert que la requête est reçue par un processus de la simulation au début d'une tâche longue.

7.2.4 Clients séquentiels multiples

Nous allons à présent étudier un autre aspect de la plate-forme EPSN : le cas de clients séquentiels multiples connectés simultanément à une simulation séquentielle ($M = 1$, $N = 1$, $K \geq 1$). Nous cherchons à évaluer l'impact sur la simulation de la gestion de ces multiples clients, lorsqu'une requête de type *getp* est déclenchée par tous les clients. L'envoi des données vers tous les clients doit alors s'effectuer à chaque itération ($p = 1$). Pour cette expérience, nous avons considéré une simulation séquentielle ($M = 1$) effectuant une itération de calcul toute les 400 ms pour une plage d'accès en lecture de 200 ms, soit une plage de recouvrement maximale des transferts de 50%. Nous avons choisi pour cette expérience une donnée de taille $D = 5000$ Ko dont nous avons évalué le temps de transfert à environ 50 ms pour OmniORB4 (avec copie à la réception). Ce temps représente $1/4$ de la plage de recouvrement maximale, ce qui signifie qu'au mieux on peut espérer transférer cette donnée vers 4 clients séquentiels avant de ralentir effectivement la simulation.

Nombre de clients (K)	**0**	**1**	**2**	**3**	**4**	**6**	**10**
1 worker sur le CPU 0	400.0	407.1	414.9	422.2	429.8	598.1	840.4
1 worker sur le CPU 1	400.0	400.1	400.1	400.2	400.2	474.4	647.9
2 workers sur le CPU 1	400.0	400.1	400.1	400.2	400.2	468.1	641.1
2 workers (1 sur chaque CPU)	400.0	404.0	407.4	412.6	419.2	567.0	791.8

TAB. 7.2 – Temps moyen (en ms) d'une itération d'une simulation séquentielle (CPU 0) effectuant un transfert des données à chaque itération vers plusieurs clients séquentiels connectés simultanément (requête *getp* avec $p = 1$, $T_A = 200$ ms, $T_B = 200$ ms, D=5000 Ko).

Les résultats de nos expériences pour le cas multi-clients sont retranscrits dans le tableau 7.2 pour différents nombres de *workers* et différents types de placements. Le processus de la simulation est placé sur le CPU 0. Les *workers* prennent en charge l'envoi des données vers les clients et leur configuration a donc un impact direct sur les performances de la plate-forme EPSN, même s'ils partagent la même interface réseau. On remarque que dans le cas idéal où 1 *worker* est placé sur le CPU 1, le recouvrement est presque parfait puisque le surcoût apparent pour 4 clients connectés est très faible ($+0.2$ ms). Lorsqu'il y a plus de 4 clients connectés, le temps de transfert ne peut plus être recouvert et s'ajoute brutalement au temps de la simulation. Les résultats sont encore plus satisfaisants dans le cas où 2 *workers* sont placés sur le CPU 1, car ces derniers se répartissent les envois de

messages, ce qui tend à améliorer les performances même s'ils partagent la même interface réseau. En revanche, lorsque le *worker* est placé sur le même CPU que la simulation, le recouvrement n'est pas parfait et la simulation est ralentie de 7 ms incompressibles par client avant que le recouvrement ne soit plus possible (au dessus de 4 clients). Dans la configuration hybride où un *worker* est placé sur chaque CPU ($W = 2$), les résultats sont plus mitigés et de manière un peu surprenante moins bons que dans le cas où un seul worker est placé sur le CPU 1. Cela tient au fait que les envois de données sont répartis (en moyenne) entre ces deux *workers* et que les envois effectués sur le CPU 0 vont induire un ralentissement de la simulation.

7.2.5 Régulation des transferts

Afin de limiter le surcoût lié aux transferts de gros messages ne pouvant pas être recouverts, une solution simple consiste à augmenter la période p de la requête *getp*, ce qui signifie que l'envoi de la donnée se fera toutes les p itérations. Ainsi, le surcoût occasionné par le transfert devient moins fréquent et va donc diminuer en moyenne lorsque p augmente. Le tableau 7.3 relate le résultat de nos expériences effectuées pour une simulation de 400 ms par itération avec une plage d'accès de 200 ms. Nous étudions différentes configurations de type $M \times N$ avec $M = N$, en prenant des distributions identiques pour le client et la simulation afin que chaque processus de la simulation ne communique qu'avec son vis-à-vis (i.e. pas de redistribution). La taille des données est fixée à 50000 Ko par processus de la simulation, de telle façon que le transfert ne puisse pas être recouvert entièrement à $p = 1$. Notons que dans ces expériences, nous avons choisi de placer le processus de la simulation et le *worker* sur des CPUs différents. La lecture d'une ligne du tableau permet d'observer que le temps moyen d'une itération (incluant les envois périodiques) diminue effectivement en fonction de p.

Période du *getp* (p)	**0**	**1**	**2**	**3**	**4**	**6**	**10**
1×1	400.0	689.9	545.0	497.6	471.0	448.8	429.4
4×4	400.0	692.9	549.2	499.6	472.8	449.7	430.0
16×16	400.0	704.2	552.5	501.6	475.3	451.6	431.5

TAB. 7.3 – Temps moyen (en ms) d'une itération de la simulation en fonction de la période p du *getp* ($T_A = 200$ ms, $T_B = 200$ ms, D/M=50000 Ko).

Comme le volume des données est *identique pour chaque processus* dans les différentes configurations $M \times M$ (D/M=50000 Ko), on s'attend à obtenir des surcoûts identiques à p fixé (dans chaque colonne). Le surcoût supplémentaire que l'on observe (entre 2 et 15 ms pour 16×16 par rapport à 1×1) est dû au temps nécessaire pour acheminer les acquittements vers les processus de la simulation (Fig. 6.7). Cet acquittement envoyé par le client à tous les processus de la simulation permet de garantir que la réception des données est achevée ainsi que d'éventuels traitements de visualisation. Sans ce mécanisme ralentissant artificiellement la simulation, une simulation trop rapide par rapport au client aurait tôt fait de le saturer de messages qu'il ne pourrait traiter.

7.3 Évaluation de la redistribution dans RedCORBA

Nous allons maintenant consacrer une section à l'évaluation des performances de la bibliothèque RedCORBA sur laquelle EPSN s'appuie pour réaliser les transferts. Cette bibliothèque s'appuie elle-même sur la bibliothèque RedSYM responsable de la génération des messages (cf. chapitre 6). Dans le contexte du pilotage ou du couplage, il faut souligner que la génération des messages est typiquement une opération ponctuelle survenant à l'initialisation du couplage, alors que les transferts sont répétés continuellement, le plus souvent à chaque itération du code. C'est pourquoi nous allons principalement étudier les temps de transfert et plus particulièrement les débits mesurés (en Mo/s) entre les codes couplés. Les expériences pour l'évaluation de la redistribution ont entièrement été réalisées sur le cluster *Grid'5000* à Bordeaux. Dans ces expériences, nous examinons le problème de la redistribution pour différentes configurations $M \times N$ et différentes distributions d'objets complexes plus ou moins régulières. Nous évaluons le transfert de données de type *double* entre deux codes parallèles (LAM/MPI) couplés avec RedCORBA, l'un jouant le rôle d'émetteur (M processeurs) et l'autre

de récepteur (N processeurs). Plus précisément, nous mesurons le débit cumulé moyen (taille total des données / temps total) obtenu après répétition de plusieurs requêtes *put* pour des données de taille totale variant de quelques kilo-octets à plus de 100 Mo. Nous effectuons principalement des mesures de débits cumulés (en Mo/s), car elles permettent de mettre en évidence l'agrégation possible de la bande passante lors des flux parallèles de communication. Dans toutes les expériences qui vont suivre, nous avons choisi de placer un seul processus émetteur ou récepteur par bi-processeur afin qu'il dispose d'une interface réseau déiée, ce qui facilite l'analyse des résultats. Nous comparons nos résultats au débit cumulé obtenu avec MPI pour une configuration $M \times M$, sans redistribution, ce qui nous sert de en quelque sorte « d'étalon réseau ».

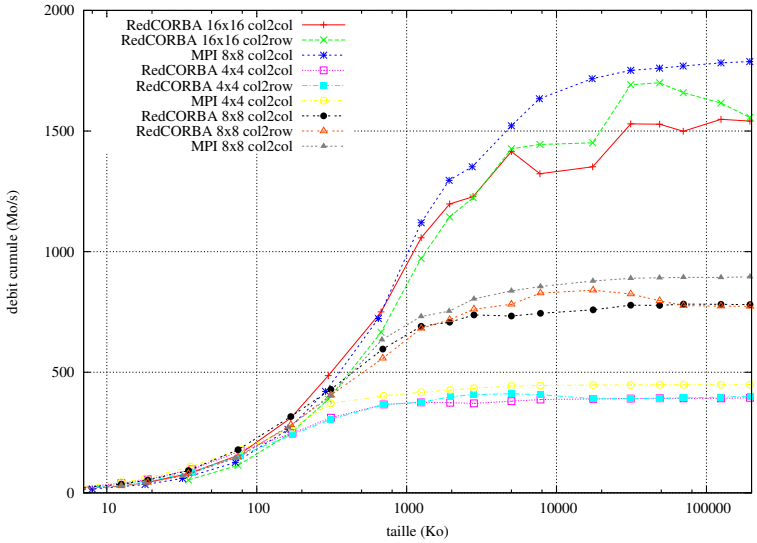


FIG. 7.8 – Redistribution des grilles structurées dans les configurations 8×8 et 16×16. Mesure du débit cumulé (en Mo/s) pour différents motifs de redistribution (*col2col* ou *col2row*) en fonction de la taille totale des données distribuées.

7.3.1 Cas des grilles structurées

Nous avons choisi d'étudier le problème de la redistribution pour une grille structurée régulière (carré) 2D avec une série de données réelles associées aux nœuds de la grille. Nous considérons deux « motifs » de redistribution : *col2col* et *col2row*. Pour le premier motif, *col2col*, les deux codes possèdent une distribution bloc-colonne 1D. Si M et N sont égaux alors les distributions sont identiques et il n'y aura donc pas de redistribution à faire (1 seul message envoyé pour chaque émetteur à son vis-à-vis). Dans le second motif, *col2row*, le code émetteur possède une distribution bloc-colonne 1D et le code récepteur une distribution bloc-ligne 1D. Nous avons choisi ce dernier motif, car il implique un schéma de communication total entre les deux codes et donc un découpage des données dans la mémoire de chaque émetteur en N messages. La figure 7.8 montre le débit cumulé obtenu pour des configurations de type $M \times M$ (avec $M = 4, 8$ ou 16) alors que la figure 7.9 s'intéresse à des configurations de type $M \times N$ avec $M = 16$ et N variant entre 1 et 16. Dans chaque figure, nous examinons les performances pour les motifs de redistribution *col2col* et *col2row*.

Les résultats obtenus pour les grilles structurées sont globalement satisfaisants, car ils montrent nettement l'agrégation de la bande-passante obtenue avec RedCORBA. Ces résultats sont d'autant plus satisfaisants que ces mesures prennent en compte le temps de gestion des requêtes et des acquittements de RedCORBA, ainsi que la copie à la réception des données. Il faut noter que l'écart de débit entre RedCORBA et MPI reste toujours du même ordre de grandeur que celui mesuré en 1×1, c'est-à-dire d'environ -13%. Toutefois, de manière assez surprenante on observe que RedCORBA avec un motif du type *col2row* obtient le plus souvent de meilleures performances qu'avec le motif *a priori* idéal *col2col*. Notre explication est que la réception de multiples « petits » messages s'effectue de manière concurrente dans RedCORBA (pool de threads CORBA), ce qui diminue très certainement le temps de recopie des données à la réception en *col2row* comparé au cas

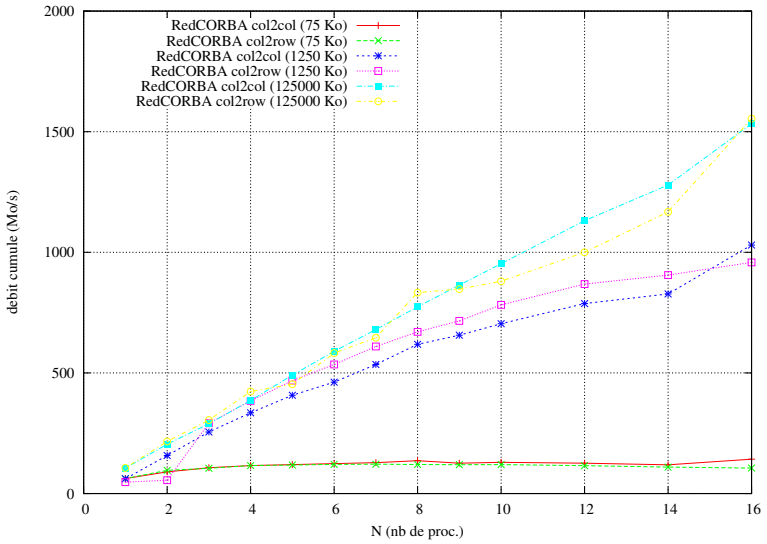


FIG. 7.9 – Redistribution des grilles structurées dans les configurations $16 \times 1,\ 16 \times 2,\ \ldots\ 16 \times 16$. Mesure du débit cumulé (en Mo/s) pour différents motifs de redistribution (*col2col* ou *col2row*) et différentes tailles de données distribuées.

col2col où un seul « gros » message est reçu. Dans la figure 7.9, nous étudions quelques cas plus irréguliers où le nombre de processeurs émetteurs et récepteurs ne sont pas identiques ($M \times N$ avec $M \neq N$). Le comportement linéaire en fonction du nombre de processeurs N traduit clairement le fait que le nombre d'interfaces réseau est un facteur limitant pour la réception. Les débits sont tout aussi satisfaisants que dans l'expérience précédente, débits qui dépendent clairement du volume des données échangées et du motif de redistribution choisi.

$M \times N$	**col2col**	**col2row**
8×8	13.0 ms	14.8 ms
16×16	27.1 ms	32.2 ms
16×8	23.3 ms	24.1 ms

TAB. 7.4 – Temps de génération des messages symboliques et physiques avec RedSYM & RedCORBA pour une grille structurée 400×400.

Concernant le temps nécessaire pour générer les messages symboliques et physiques avec RedSYM & RedCORBA (Tab. 7.4), nous constatons que celui-ci augmente en fonctions du nombre total de blocs de redistribution, résultant de l'intersection des blocs appartenant aux deux distributions (cf. Sec 5.6.2). A titre d'exemple, si l'on considère la configuration 8×8, le nombre de blocs de redistribution sera de 8 dans le cas *col2col* contre 64 dans le cas *col2row*. En outre, ce temps s'avère constant par rapport à la taille de la grille dans tous les cas triviaux (e.g. 8×8 en *col2col*). Dans les cas plus irréguliers (e.g. *col2row*), le temps de génération des messages est alors sous-linéaire par rapport au nombre d'éléments dans la grille (cf. Sec. 5.6.2). Ce temps est principalement dû à la traduction des messages symboliques en messages CORBA, dont la complexité dépend directement du nombre total d'intervalles à l'intérieur des masques d'extraction associés à chaque bloc de redistribution.

7.3.2 Cas des ensembles de particules et des boîtes d'atomes

Nous allons maintenant présenter quelques résultats expérimentaux pour les ensembles de particules et les boîtes d'atomes, qui sont des objets complexes relativement proches. La principale différence tient au fait que les boîtes d'atomes sont décrites à l'aide d'une distribution des éléments par bloc dans $\mathbb{R}^n$, très fréquentes dans les codes utilisant une technique de décomposition spatiale. Nous avons proposé à la section 5.6 deux approches différentes pour la redistribution de ces objets, une approche spatiale comparable aux grilles structurées dans le cas des boîtes d'atomes et une approche placement (avec ou sans découpage) pour les ensembles de particules.

La figure 7.10 présentent plusieurs expériences de redistributions plus ou moins régulières en 8×8 et en 8×7, pour un nombre de particules ou d'atomes variant de quelques milliers à plusieurs millions. Pour chaque cas, nous mesurons le débit cumulé (en Mo/s) relatif au transfert de de la série des coordonnées en 2D. Les particules ou atomes sont initialement distribuées de manière équilibrée entre tous les processeurs émetteurs.

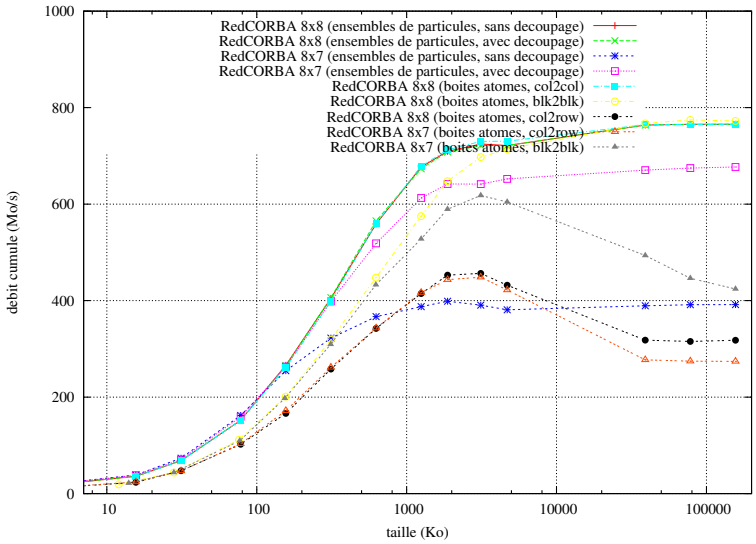


FIG. 7.10 – Redistribution avec RedCORBA des ensembles de particules et des boîtes d'atomes dans $\mathbb{R}^2$.

Commençons par analyser les résultats sur les ensembles de particules. Notons que la stratégie de placement avec découpage donne des résultats identiques à la stratégie sans découpage pour des cas triviaux (8×8), mais offre de bien meilleures performances lorsque M et N sont différents. Dans le cas 8×7, le débit est même divisé par 2, ce qui s'explique par le fait que le premier processeur côté réception se voit attribuer lors du placement deux régions contre une seule pour les autres récepteurs. Dans le cas des boîtes d'atomes, les performances dépendent directement du motif de redistribution, ce motif résultant de l'intersection des distributions choisi pour les deux codes couplés. Nous étudions les performances pour plusieurs types de distributions : bloc-colonne 1D (*col*), bloc-ligne 1D (*row*) et un pavage de l'espace en blocs 2D (*blk*). Plus précisément, la distribution *blk* correspond à une décomposition du domaine global 2D en $M \times M$ blocs (côté émetteur), encore raffiné en 2×2 sous-blocs, ce qui nous donne un pavage relativement fin du domaine : en tout $4.M^2$ blocs côté émetteur et $4.N^2$ blocs côté récepteur. Lorsque les distributions sont « bien ajustées » (*col2col* ou *blk2blk* en 8×8, les performances sont aussi bonnes que pour les ensembles de particules. En revanche, lorsque les distributions sont « pathologiques » (*col2row* ou *blk2blk* en 8×7 et *col2row* en 8×8), les atomes à envoyer vers un certain destinataire doivent être extraits des blocs-source en fonction de leurs coordonnées spatiales (cf. Sec. 5.6.2), ce qui induit un découpage important des données en mémoire et l'utilisation d'un cache à l'envoi dans RedCORBA. Ce comportement est d'autant plus critique lorsque le nombre d'atomes augmente, entraînant une chute importante des performances au delà d'un certain seuil.

Le temps nécessaire pour générer les messages CORBA avec RedSYM et RedCORBA (incluant les échanges de descripteurs) est constant par rapport à la taille des données dans le cas des ensembles de particules et ne dépend que du nombre de processeurs M et N. Ce temps est de l'ordre de 10 ms dans les expériences précédentes. Dans le cas des boîtes d'atomes, le comportement est identique tant que les motifs de redistribution ne sont pas pathologiques. En revanche dans les cas pathologiques, la génération des messages devient linéaire par rapport au nombre d'atomes dans les boîtes (parcours des atomes pour le calcul des différents masques d'extraction) : environ 200 ms pour 100000 atomes en *col2row* 8×8 ou 8×7, et 50 ms en *blk2blk* 8×7.

Pour conclure, notons que la dynamicité des éléments (i.e. fluctuation du nombre d'éléments par région) n'induit aucun surcoût apparent dans le cas des ensembles de particules, car il suffit dans ce cas de remettre à jour localement (côté émetteur) l'adresse et de la taille des messages physiques. En revanche, cette remise à jour

des messages entraîne un chute de débit importante pour les cas pathologiques avec des boîtes d'atomes, car il faut alors re-trier les atomes se déplaçant au sein des boîtes (ou migrant vers une autre boîte) avant chaque nouvel envoi. Dans le contexte du pilotage, nous préférerons toujours utiliser la stratégie de placement des ensembles de particules plutôt que l'approche spatiale avec des boîtes d'atomes, car elle est plus simple à mettre en œuvre et offre de meilleures performances.

7.3.3 Cas des maillages non structurés

Nous reproduisons des expériences similaires aux ensembles de particules mais pour des maillages non structurés. Il s'agit de comparer le comportement des redistributions de maillages en 8×8 et en 8×7 pour les stratégies de placement avec ou sans découpage. La figure 7.11 montre les débits cumulés obtenus avec RedCORBA pour transférer une série de données associées aux nœuds du maillage (de type *double*). Le maillage est en fait une grille de quadrilatères définie comme un maillage non structuré (i.e. liste de connectivités et coordonnées spatiales des nœuds), initialement découpée en colonne sur chaque processeur côté émetteur. Notons que selon la stratégie de redistribution utilisée (avec ou sans découpage), le résultat du placement ne sera pas le même.

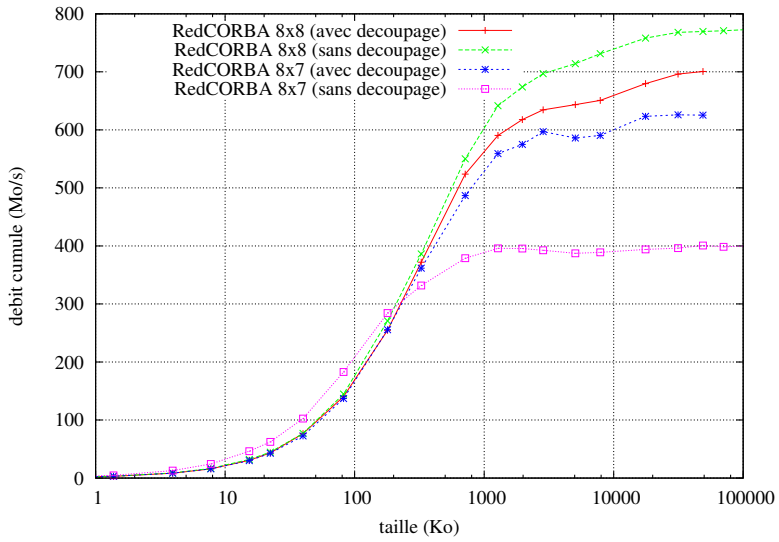


FIG. 7.11 – Redistribution des maillages non structurés avec RedCORBA : débit cumulé en fonction de la taille totale des données distribuées pour les stratégies de placement avec ou sans découpage.

Dans le cas idéal 8×8, le placement calculé est identique pour les deux stratégies et l'on pourrait s'attendre à obtenir d'aussi bonnes performances avec découpage que sans découpage. En fait, la différence de résultat tient au fait que nous n'avons pas optimisé notre algorithme de génération des messages pour prendre en compte les cas particuliers où M et N sont multiples entre eux voire égaux. Il en résulte que le maillage est artificiellement découpé en 8 unités qui seront toutes transmises dans le même message CORBA au processeur vis-à-vis. La différence de performance observée est essentiellement liée au fait que nous utilisons un cache à l'envoi[3]. En revanche, le temps nécessaire pour générer ces messages est très supérieure pour la stratégie avec découpage où nous avons recours à un algorithme plus complexe de partitionnement des maillages actuellement basé sur Metis (Tab. 7.5). D'une manière générale, le temps de génération des messages est constant (dépendant de M et N uniquement) dans la stratégie sans découpage alors qu'il est sur-linéaire par rapport au nombre de cellules du maillage dans la stratégie avec découpage.

Le cas 8×7 est là pour mettre en évidence l'intérêt de la stratégie avec découpage pour équilibrer les flux de communication et obtenir de bonnes performances. En effet, on constate que le débit est divisé par 1.5 pour la

[3]Quelques expériences plus récentes nous ont permis d'observer qu'OmniORB4 gérait relativement bien les messages non contigus en mémoire, ce qui permet d'éviter l'utilisation d'un cache à l'envoi et augmente considérablement les performances dans ce cas.

stratégie sans découpage. Cela tient au fait que le premier récepteur reçoit exactement deux régions complètes du maillage. Le volume de communication est alors multiplié par deux pour ce processeur, ce qui pénalise les performances globales du transfert. Notons que le débit serait exactement divisé par 2 si l'on ne prenait pas en compte la copie dans le cache d'envoi pour la stratégie avec découpage.

$M \times N$	**sans découpage**	**avec découpage**
8×8	11.2 ms	256.5 ms
8×7	10.2 ms	252.0 ms

TAB. 7.5 – Temps de la génération des messages symboliques et physiques avec RedSYM & RedCORBA pour un maillage de 160 000 quadrilatères.

7.4 Quelques applications réelles de pilotage avec EPSN

Nous relatons dans cette section un ensemble d'expériences de pilotage menées avec EPSN sur de « vrais » codes de simulations numériques. Nous avons choisi pour illustrer les possibilités de notre plate-forme quatre codes parallèles différents, tous basés sur MPI, écrits en C ou en Fortran. Le premier de ces codes (*heat2d*) est un « cas d'école » ayant servi à valider nos algorithmes et à mettre en place la plate-forme EPSN tout au long de son développement. Les autres codes correspondent à des simulations développées par différentes équipes de chercheurs dans divers domaines et auxquels nous avons eu accès librement par Internet (Gadget2, POP) ou au titre d'une collaboration plus étroite (FluidBox).

Même si les résultats que nous présentons ici permettent de valider notre approche du pilotage, il faut cependant souligner qu'ils sont encore préliminaires et qu'une plus fine analyse des codes décrits dans cette section nécessiterait une véritable collaboration avec les équipes de chercheurs concernées.

7.4.1 Heat2d : équation de la chaleur 2D

La première simulation que nous présentons est « un cas d'école » résolvant l'équation de la chaleur en deux dimensions par la méthode des différences finies. Le code source est écrit en C/C++, mais nous disposons également d'une version Fortran 77. La simulation utilise la bibliothèque MPI (Message Passing Interface) selon un modèle de programmation maître/esclaves ; cependant, nous nous contentons de piloter les esclaves qui respectent un modèle de programmation SPMD. Le domaine de calcul est une grille régulière 2D de dimension 4000×4000, représentant la température (réels en simple précision, 62500 Ko). Les données sont initialement distribuées en bloc-ligne 1D sur tous les esclaves. La simulation calcule en boucle la température en chaque point de la grille à partir des valeurs précédentes de son voisinage (stockées dans un second tableau) et ainsi de suite en inversant le rôle des tableaux à chaque itération (tâche *swap*). La complexité de cet algorithme est ainsi linéaire en fonction du nombre de points dans la grille (cas d'une « simulation rapide »). Le fichier XML (Fig. 7.12) décrit la modélisation que nous faisons de ce code en MHT. L'accès aux températures est autorisé durant tout le temps du corps de boucle, hormis au moment de la tâche *swap*. De plus, nous avons ajouté en fin de boucle une tâche en point permettant d'effectuer deux types d'interactions : la modification des températures via une requête cliente *put* et le déclenchement d'une action *raz* remettant la grille des températures à sa valeur initiale.

La figure 7.13 illustre l'évolution des températures au fil des itérations telle que nous la suivons avec le client générique Simone. Les conditions aux limites étant initialement fixées à une température froide, la température globale va progressivement baisser comme le montre les trois première étapes (température froide en bleu). Le client Simone nous permet d'interagir directement avec la simulation en modifiant certaines valeurs des températures (introduction d'un point chaud à l'étape 4). La réponse de la simulation peut alors être suivie visuellement aux étapes 5 et 6.

```
<simulation id="heat2d">
  <data id="heat" class="grid">
    <serie id="temperature" />
  </data>
  <action id="raz"/>
  <MHT>
    <data-context ref="heat" context="readable"/>
    <loop id="main">
      <task id="body">
        <task id="swap">
          <data-context ref="heat" context="modified"/>
        </task>
        <point id="interaction">
          <data-context ref="heat" context="writable"/>
          <action-context ref="raz" context="allowed"/>
        </point>
      </task>
    </loop>
  </MHT>
</simulation>
```

FIG. 7.12 – Description en XML du MHT de la simulation *heat2d*.

Afin d'établir l'influence d'une session de pilotage sur la durée d'une simulation, nous avons réalisé une série d'expériences sur une simulation s'exécutant sur $M = 4$ ou 8 processeurs. La grille de température étant de taille relativement importante (62500 Ko), le temps de transfert des données vers un client séquentiel est d'environ 550 ms, ce qui est largement supérieur au temps d'exécution de la simulation « à vide » (i.e. sans EPSN). Par conséquent, il n'y a aucune chance de recouvrir les communications par les calculs de la simulation. Le tableau 7.6 donne le temps moyen d'une itération mesuré expérimentalement pour une requête *getp* avec ou sans visualisation dans le cas d'un client séquentiel. Même si ce ralentissement est relativement important (du fait de l'importance des transferts), l'utilisation d'une période d'envoi $p = 10$ permet de diminuer efficacement ce surcoût, y compris dans le cas de la visualisation séquentielle (résolution 1280×1024). Nous avons également réalisé des expériences de visualisation en ligne sur notre mur d'images avec ICE-T. Nous avons testé deux distributions des données clientes sur les 4 machines de visualisation : la distribution bloc-colonne 1D (*col*) et la distribution bloc 2D (*blk*). De manière assez intuitive, ce dernier choix de distribution s'avère le plus pertinent et donne de meilleurs résultats que la visualisation séquentielle pour une résolution d'image 4 fois plus grandes (2560×2048). Nous avons évalué ce temps de rendu à 410 ms contre 740 ms dans le mode *col*. Cette différence est

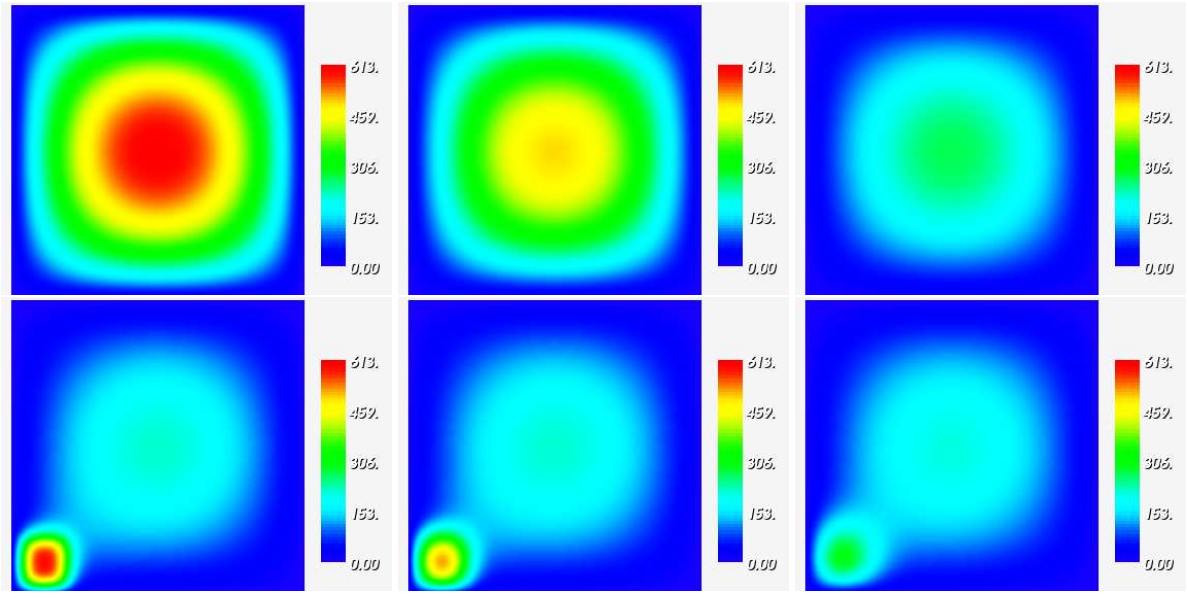


FIG. 7.13 – Visualisation avec VTK de l'évolution des températures dans la simulation *heat2d* et modification interactive de la température.

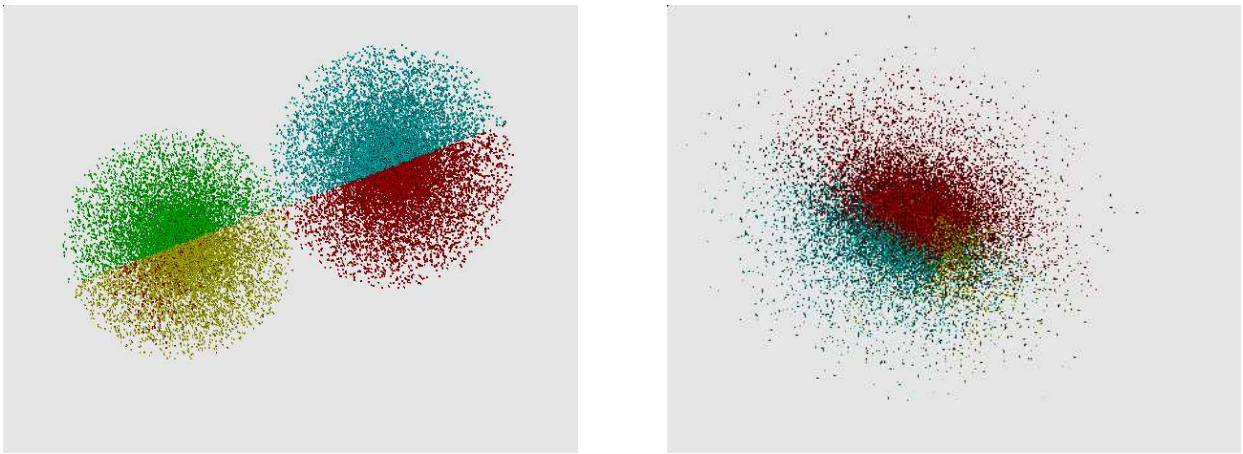
liée au fait que la composition des rendus partiels avec ICE-T nécessite un plus gros volume de communications dans le cas *col* (cf. Sec 2.3.2).

nb proc. simu. (M)	$M = 4$	$M = 8$
simulation à vide	78 ms/iter.	40 ms/iter
getp (N=1, p=1)	760 ms/iter.	739 ms/iter
getp (N=1, p=10)	150 ms/iter.	111 ms/iter
visu. séq. en ligne (p=10)	166 ms/iter.	174 ms/iter
visu. paral. en ligne ICE-T (N=4, p=10, col)	225 ms/iter	220 ms/iter
visu. paral. en ligne ICE-T (N=4, p=10, blk)	147 ms/iter	143 ms/iter

TAB. 7.6 – Diverses expériences de pilotage réalisées avec EPSN sur la simulation *heat2d* pour un cas test 4000×4000 (62500 Ko).

7.4.2 Gadget2 : simulation en astrophysique

Gadget2 [12] simule la formation des structures cosmologiques, comme par exemple la collision de galaxies ou l'effondrement gravitationnel d'une galaxie sur elle-même. Ce code, écrit en C et parallélisé avec MPI, a été développé par Springel *et al.* [193] à l'Institut Max-Planck d'Astrophysique en Allemagne. Gadget2 permet de résoudre des problèmes à N-corps en calculant les forces gravitationnelles sur la base d'une décomposition hiérarchique de l'espace en octree. Les galaxies sont modélisées comme un fluide ou un gaz de particules régies par les lois de l'hydrodynamique (cf. méthode SPH [182]).



(a) Distribution initiale sur 4 processeurs. (b) Collision des deux galaxies.

FIG. 7.14 – Visualisation en ligne avec VTK de la simulation *Gadget2*.

Le cas test que nous étudions simule la collision de deux galaxies, modélisées par un total de 60000 particules distribuées sur 16 processeurs dans notre expérience (Fig. 7.14). L'ensemble des particules est décrit par un objet complexe de type `RedSYM::Particles` avec une seule région par processeur. Nous considérons plusieurs séries de données indiquant la position des particules dans l'espace, leurs vitesses, leurs masses ainsi que l'énergie potentielle. Comme ces informations sont stockées dans un tableau de structures (incluant d'autres informations), chaque série de données est complètement stridée en mémoire (utilisation d'un cache pour l'envoi). La simulation est modélisée dans EPSN par un MHT contenant une simple boucle de calcul incluant trois sous-tâches : (1) le déplacement des particules, (2) le calcul de la nouvelle décomposition du domaine et (3) le calcul de l'accélération. La dernière étape de complexité $\mathcal{O}(n.log(n))$ représente 75% des calculs d'une itération et offre un large plage d'accès en lecture pour les clients distants. Par ailleurs, il faut souligner que ce code est fortement dynamique puisque nous constatons que de nombreuses particules migrent entre les processeurs à chaque itération. En effet, lors de la simulation, les galaxies se rapprochent, entraînant un

recalcul de la décomposition du domaine à chaque itération. La figure 7.15 illustre le suivi en ligne dans Simone (à l'aide du plug-in QWT) du nombre de particules par processeur (avec une moyenne de 3700 particules par processeur).

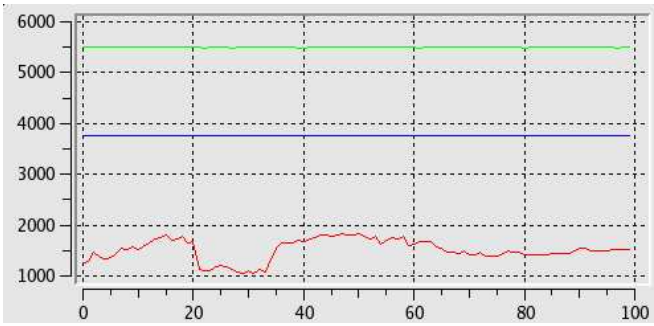


FIG. 7.15 – Suivi en ligne dans Simone du nombre minimum, moyen et maximum de particules dans *Gadget2* mettant en évidence la migration des particules au cours des calculs.

Les résultats de nos expériences sont détaillées dans le tableau 7.7. Pour une simulation s'exécutant en parallèle sur 16 processeurs, le temps d'une itération sans EPSN est en moyenne de 258 ms (après 200 itérations). Le transfert des données (série des positions, 1400 Ko) vers un client séquentiel est de l'ordre de 40 ms mais n'est pas très bien recouvert dans le cas du *getp* ($p = 1$) certainement à cause des communications MPI internes au code qui sont relativement intensives. La visualisation en ligne avec une technique classique à base de *glyphs* implique un temps de rendu de l'ordre de ~ 8 s en séquentiel et de 2 s en parallèle sur le mur d'images avec ICE-T. Dans cette technique, chaque particule est représentée par un objet polygonale (un tetraèdre dans nos expériences, ce qui fait un total de 240000 polygones). Notons que nous utilisons pour la redistribution une stratégie de placement des régions sans découpage : 4 régions placées sur chaque processeur de visualisation. En remplaçant les glyphs par une technique de visualisation à base de *sprites* (i.e. des textures 2D représentant une sphère en trompe-l'œil), les temps de rendus séquentiels et parallèles sont plus raisonnables (environ 264 ms avec ICE-T) et le surcoût devient acceptable pour effectuer de la visualisation en ligne sur le mur d'images (inférieur à 10% pour $p = 10$).

nb proc. simu. (M)	$M = 16$
simulation à vide	258 ms/iter.
getp (p=1)	342 ms/iter.
getp (p=10)	279 ms/iter.
visu. séq. en ligne (p=1, glyph)	8250 ms/iter.
visu. séq. en ligne (p=10, glyph)	1441 ms/iter.
visu. paral. en ligne ICE-T (N=4, p=1, nosplit, glyph)	2370 ms/iter.
visu. paral. en ligne ICE-T (N=4, p=10, nosplit, glyph)	379 ms/iter.
visu. paral. en ligne ICE-T (N=4, p=1, nosplit, sprite)	370 ms/iter.
visu. paral. en ligne ICE-T (N=4, p=10, nosplit, sprite)	280 ms/iter.

TAB. 7.7 – Diverses expériences de pilotage réalisées avec EPSN sur la simulation Gadget2 pour un cas test de 60000 particules.

7.4.3 FluidBox : simulation en mécanique des fluides

FluidBox [159] est un code parallèle (MPI, Fortran) développé au MAB au sein du projet ScAlApplix et modélisant les équations classiques de Navier-Stokes compressible en 2D ou 3D. Le cas test que nous présentons modélise l'écoulement hypersonique de l'air (considéré comme un gaz parfait) autour d'un demi-cylindre représentant le cockpit d'une navette pénétrant dans l'atmosphère. Ce cas test s'appuie sur un maillage non structuré 2D à base de triangles. La version de FluidBox que nous utilisons pour simuler ce cas s'appuie sur

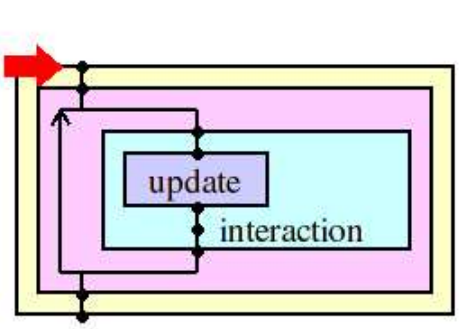


FIG. 7.16 – MHT représentant la simulation *FluidBox*.

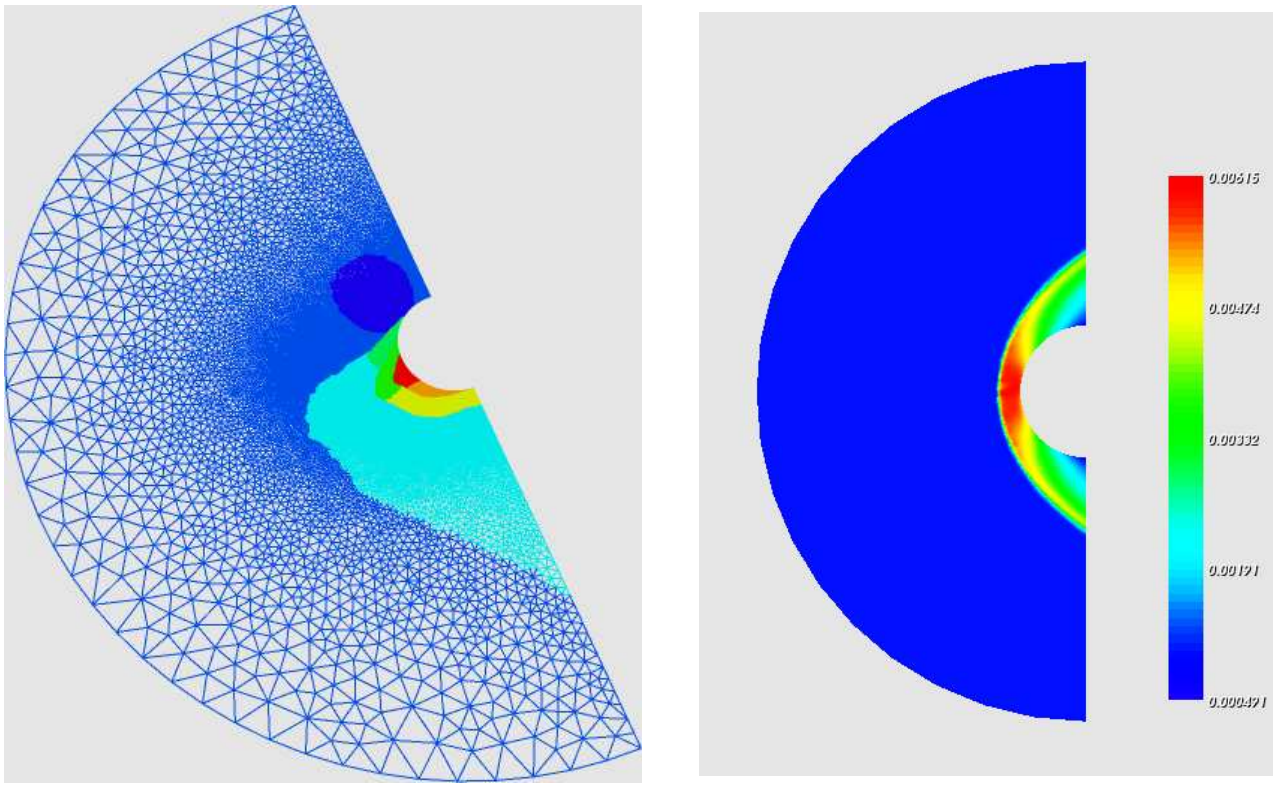
une méthode explicite (i.e. pas de formulation matricielle du problème) de type « volumes finis ». A chaque itération de la boucle en temps, la simulation calcule les échanges de flux entre les cellules duales et met à jour sur tous les nœuds les variables conservatives (densité, quantité de mouvement et énergie totale), puis les variables thermodynamiques locales (pression, vitesse du son, température).

Nous représentons cette simulation d'une manière relativement grossière en considérant une simple tâche en boucle contenant une sous-tâche *update* durant laquelle les variables conservatives sont mises à jour à partir de variables intermédiaires (Fig. 7.16). En résumé, les données sont accessibles pratiquement tout le temps d'une itération (plage d'accès maximale). Nous avons exécuté cette simulation sur un maillage de 16760 triangles et 8595 nœuds distribués sur 8 processeurs (Fig.7.17 (a)). Le temps moyen d'une itération est alors de 2.35 secondes. Le transfert de la série de données « pression » (∼86 Ko) vers un client séquentiel met environ 40 ms et se trouve totalement recouvert (pas de surcoût apparent). Comme illustré sur la figure 7.17 (b), nous effectuons la visualisation en ligne de la pression sur le maillage à chaque itération (requête *getp* avec $p = 1$). Le temps de la simulation mis pour chaque itération est alors de 2.65 s pour un client de visualisation séquentiel contre 2.41 s pour la visualisation en parallèle sur le mur d'images avec ICE-T. Dans ce dernier cas, nous effectuons une redistribution de type « placement sans découpage », ce qui revient simplement à placer deux parties du maillage de la simulation sur chaque processeur de visualisation ($N = 4$).

Une fois l'état stationnaire atteint (Fig. 7.17 (b)), il est alors possible de perturber « à la main » l'écoulement du fluide en modifiant simplement sa densité aux abords du cockpit. Cette expérience correspond à un exemple concret d'interaction effectué par les chercheurs sur ce code. Auparavant, il était nécessaire de stopper la simulation en sauvegardant toutes les informations courantes dans un fichier. Puis dans un deuxième temps, il fallait modifier à la main les valeurs de la densité aux endroits souhaités, avant de relancer les calculs de simulation grâce à un mécanisme de reprise. Ces interactions sont souvent lourdes et fastidieuses à mettre en application, alors qu'EPSN fournit une approche à la fois plus conviviale et plus efficace. Pour ce faire, nous avons ajouté un point d'interaction dans la simulation à la fin de la boucle afin de permettre de modifier les différentes variables associées au maillage (Fig. 7.16). Cependant, nous soulignerons que l'interaction est d'autant plus conviviale que la simulation calcule rapidement, car il est ainsi possible d'obtenir un retour visuel immédiat sur les actions effectuées.

7.4.4 POP : simulation de la circulation océanique

POP (Parallel Ocean Program) [124] a été développé à Los Alamos, il y a une dizaine d'années et participe activement aux efforts actuels concernant la prédiction des changements climatiques de la Terre. Notons que POP est en fait un composant qui peut être couplé à d'autres modèles dont notamment le composant CICE modélisant le comportement de la glace aux pôles. POP est une simulation de la circulation océanique héritant d'une longue tradition de modèles (Bryan, Cox, Semtner, Chervin) et servant aujourd'hui de code de référence dans ce domaine. Ce code résout les équations tridimensionnelles de la dynamique des fluides sur une sphère en rotation sous l'approximation de Boussinesq [124].



(a) Distribution sur 8 processeurs.

(b) Densité de l'air aux abords du cockpit.

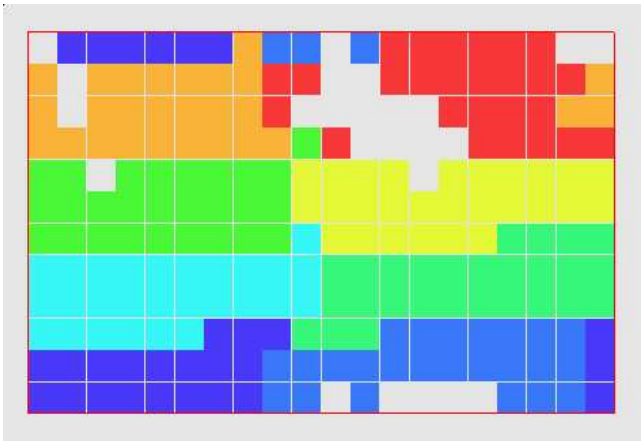
FIG. 7.17 – Simulation *FluidBox* et visualisation en ligne avec VTK : cas d'un maillage à 16760 triangles.

Ce code est l'un des exemples clés ayant motivés nos travaux sur la redistribution de données complexes. POP décrit la surface des océans à l'aide d'une grille irrégulière (Fig. 7.18). Nous utilisons un objet RedSYM de type `Grid` pour représenter cette grille 2D, dont les coordonnées cartésiennes (en 3D) de chaque nœud sont explicitement définies par la série *coord*. L'utilisation de cette série nous permet de reconstruire automatiquement le modèle sphérique 3D côté visualisation (Fig. 7.20). Les autres séries contiennent différentes grandeurs physiques comme la pression à la surface des océans, la température ou encore la profondeur des océans (Fig. 7.19 (b)). Du point de vue de la distribution, cette grille est explicitement découpée en blocs 2D de taille identique répartie sur les différents processeurs pour équilibrer la charge. Par ailleurs, cette distribution est creuse, car les blocs au niveau des continents ne sont pas pris en compte dans le modèle océan (Fig. 7.19 (a)). Du point de vue du stockage, chaque bloc de la distribution est en fait contenu à l'intérieur d'un bloc alloué de taille plus grande incluant un *halo* de `nghost` cellules fantômes. Il en résulte que les données en mémoire représentant un « bloc réel » sont complètement stridées en mémoire (*wrapGhost()*). Dans ce cas, nous utilisons un cache à l'envoi pour reconstruire un buffer contigu mais au prix d'une copie.

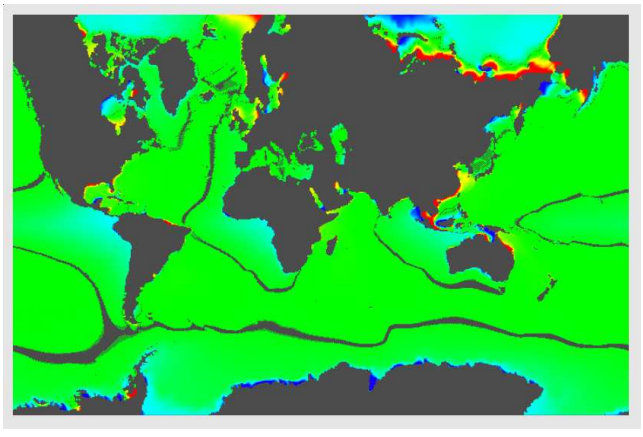
Les résultats que nous avons obtenus pour le pilotage de ce code sont encore préliminaires, mais nous avons tout de même souhaité les présenter dans cette thèse. En effet, à cause d'un problème de portage du code Fortran de POP sur l'architecture 64 bits du cluster *Grid'5000*, nous utilisons le cluster de visualisation (4 bi-processeurs, 32 bits) comme un cluster de calcul. L'expérience que nous avons effectuée s'exécute sur une grille 720×468, ce qui fait un total de 2632 Ko pour la série de données des pressions et de 7897 Ko pour les coordonnées cartésiennes 3D de la grille. Le temps d'une itération de la simulation à vide sur 8 CPUs est de 11.8 secondes ; par conséquent la visualisation en ligne n'a pas un intérêt majeur dans ce contexte. Nous préférons parler de « visualisation de surveillance » (*monitoring*). Dans ce cas, l'utilisateur se connecte et se déconnecte ponctuellement pour surveiller l'évolution des calculs et éventuellement interagir en corrigeant certains paramètres. Le transfert des données vers une tierce machine connectée au réseau giga-ethernet de notre cluster s'effectue rapidement en quelques dizaine de millisecondes, ce qui est favorisé par une large plage d'accès en lecture (quasiment toute la durée de l'itération). En revanche, la reconstruction du modèle sphérique tridimensionnel prend un temps considérable de l'ordre de 22 secondes en séquentiel. Ce temps accru s'explique par le fait que le pipeline de

```
1  !*** global domain
2  call RedSYM_BlockInt2D_create(domain,0,0,nx_global-1,ny_global-1)
3
4  !*** create the distribution for the surface ocean model
5  call RedSYM_DistributionInt_create(dist,domain)
6  call RedSYM_DistributionInt_create(ghost_dist,domain)
7  do num_block = 1,nblocks
8
9     !*** real block (in absolute coordinates)
10    rblock = get_block(blocks_clinic(num_block),num_block)
11    call RedSYM_DistributionInt2D_addExplicitBlock(dist,          &
12         rblock%i_glob(rblock%ib)-1, rblock%j_glob(rblock%jb)-1, &
13         rblock%i_glob(rblock%ie)-1, rblock%j_glob(rblock%je)-1)
14
15    !*** ghost block (larger allocated block with ghost cells)
16    call RedSYM_DistributionInt2D_addExplicitBlock(ghost_dist,                  &
17         rblock%i_glob(rblock%ib)-1-nghost, rblock%j_glob(rblock%jb)-1-nghost, &
18         rblock%i_glob(rblock%ie)-1+nghost, rblock%j_glob(rblock%je)-1+nghost)
19 end do
20
21 !*** create grid object for the surface ocean model
22 call RedSYM_Grid_create(ocean, "ocean", prank, psize)
23 call RedSYM_Grid_setNumberOfDimensions(ocean, 2)                    ! nb dimensions
24 call RedSYM_Grid_setDistribution(ocean, dist)                       ! real distribution
25 call RedSYM_Grid_addSerie(ocean, "coord", REDSYM_DOUBLE, 3)         ! serie 0
26 call RedSYM_Grid_addSerie(ocean, "pressure", REDSYM_DOUBLE, 1)      ! serie 1
27 call RedSYM_Grid_addSerie(ocean, "temperature", REDSYM_DOUBLE, 1)   ! serie 2
28 call RedSYM_Grid_setCoordinateSerie(ocean, 0)                      ! the coordinate serie
29
30
31 !*** wrap buffers for all blocks and all series (without ghost cells)
32 do num_block = 1,nblocks
33    call RedSYM_DistributionInt_getBlock(ghost_dist, num_block-1, gblock)
34    call RedSYM_Grid_wrapGhost(ocean, 0, num_block-1, UCOORD(:,:,:,num_block), gblock)
35    call RedSYM_Grid_wrapGhost(ocean, 1, num_block-1, PSURF(:,:,curtime,num_block), gblock)
36    call RedSYM_Grid_wrapGhost(ocean, 2, num_block-1, TRACER(:,:,1,1,curtime,num_block), gblock)
37 end do
```

FIG. 7.18 – Exemple de déclaration avec l'API RedSYM en Fortran d'une grille irrégulière modélisant la surface des océans dans le code POP.



(a) Distribution par bloc « creuse » sur 8 processeurs.



(b) Pression à la surface des océans.

FIG. 7.19 – La simulation POP pilotée par la plate-forme EPSN.

traitement des données en VTK s'avère relativement long et complexe (i.e. prix de la conversion d'un objet structuré par blocs en un maillage VTK non structuré), même si nous pensons pouvoir à terme l'optimiser. A titre comparatif, le temps du rendu calculé en parallèle avec ICE-T et affiché sur notre mur d'images (Fig. 7.20) n'est plus que de 400 ms (avec la simulation en pause). Ainsi, en optimisant le pipeline de visualisation VTK et en déployant cette simulation avec plus de processeurs sur *Grid'5000*, on peut espérer réaliser de la visualisation en ligne.

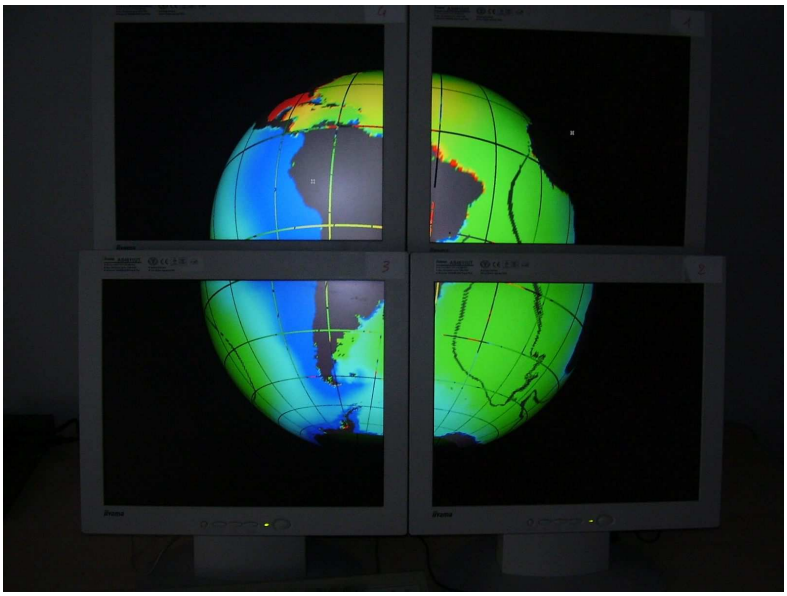


FIG. 7.20 – Reconstruction 3D et visualisation sur le mur d'images avec ICE-T de la simulation POP.

7.5 Conclusion

Tous ces résultats mettent en évidence la pertinence d'une approche $M \times N$ dans le contexte du pilotage. Nous avons vu qu'il était possible grâce à l'utilisation de multiples stratégies (envois zéro-copie, flux parallèles, plage de recouvrement, visualisation parallèle, *etc.*) de réduire « au mieux » le surcoût du pilotage dans les simulations même si certains temps de transfert ou de calcul pour la visualisation sont parfois incompressibles. Plus précisément, nous pensons que la pertinence de la visualisation en ligne dépend principalement de la « géométrie » des codes pilotés : temps d'une itération, volume des données à transférer, plages d'accès en lecture, complexité des algorithmes de visualisation, *etc*. Lorsque la simulation est « trop rapide », il est nécessaire de la ralentir – même artificiellement, en mode pas à pas – pour suivre les étapes intermédiaires du calcul. A l'inverse, si la simulation est « trop lente », on peut envisager de piloter un cas test plus petit ou d'utiliser davantage de processeurs pour autant que la simulation soit *scalable*. Pour des simulations en temps de calculs très longs, nous préférons parler de surveillance ou de contrôle à distance (*monitoring*) plutôt que de visualisation en ligne. A ce titre, l'outil Simone est tout à fait adapté pour effectuer des accès ponctuels aux données distantes, du suivi temporel de grandeurs scalaires et des interactions de pilotage légères comme le changement de paramètres en cours d'exécution. Cette approche est confortée par le fait que l'environnement EPSN est peu intrusif dans une simulation. En effet, il n'y a *pas de surcoût apparent* lorsqu'aucun client n'est connecté ou lorsque que ces clients ne soumettent pas de requêtes.

Notons pour conclure que nous avons réalisé des expériences de pilotage non relatés dans ce chapitre avec d'autres codes de simulation numérique, essentiellement en mécanique des fluides (e.g. ESMF [11], Clawpack [7]) et en dynamique moléculaires (e.g. LAMMPS [190] et NAMD [199]). A titre d'exemple, la figure 7.21 illustre une expérience de pilotage sur un cas test de ESMF (Earth System Modeling Framework). Ce *framework* nous intéresse plus particulièrement car il permet de coupler plusieurs composants/modèles physiques (dont le code POP) pour l'étude du comportement climatique de la Terre. Comme nous le verrons dans les perspectives de cette thèse (chapitre 8), nous envisageons de faire évoluer la plate-forme EPSN afin de piloter de telles simulations multi-physiques. Nous verrons également dans ce chapitre quelques perspectives pour l'optimisation des performances de la plate-forme EPSN.

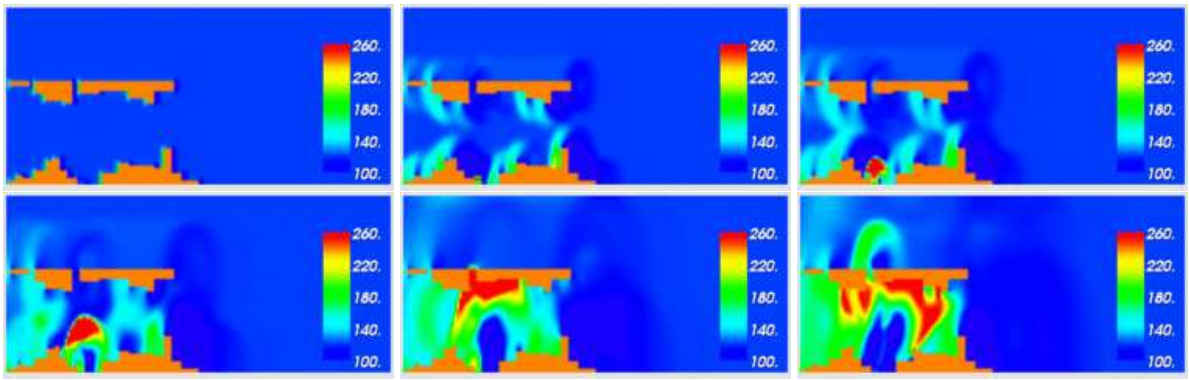


FIG. 7.21 – ESMF dans EPSN : visualisation en ligne de l'écoulement d'un premier fluide perturbé par des obstacles et pilotage de l'injection d'un second fluide à forte pression (en rouge, à la troisième étape).

Chapitre 8

Conclusion générale et perspectives

En guise de conclusion, nous souhaitons donner un ensemble de perspectives sur les trois principales contributions de cette thèse : le modèle pour le pilotage de simulations parallèles, le modèle pour la redistribution des objets complexes et la réalisation de la plate-forme EPSN.

8.1 Vers le pilotage des simulations parallèles distribuées

Actuellement, notre modèle se limite à la représentation et au pilotage de codes SPMD. En pratique, il est possible de supporter des codes plus complexes à condition que l'instrumentation se limite à une vision SPMD de l'application. Par exemple, dans le cas d'une simulation maître/esclaves, il est possible d'ignorer le processus maître. Pour des simulations muti-codes (MPMD), on peut décider d'instrumenter chaque code séparément. Toutefois, ces solutions ne sont pas très satisfaisantes, car elles ne permettent pas de représenter globalement l'application couplée. Le « vrai » pilotage d'applications couplant des codes différents comme les simulations multi-physiques reste un problème largement ouvert, pour lequel notre modèle offre certaines perspectives. Ces codes soulèvent des difficultés tant pour leur représentation que pour la définition d'interactions de pilotage globalement cohérentes entre les codes. Nous pouvons typiquement distinguer deux types d'interactions pour ces applications : les *interactions intra-codes* au sein d'un seul code et les *interactions inter-codes* mettant en jeu tous les codes. Par ailleurs, les applications du couplage de codes possèdent généralement deux modes d'exécution : individuel ou couplé. Dans le mode individuel, un code s'exécute seul (i.e. pas de couplage) avec des conditions aux limites fixées de manière *ad-hoc* pour compenser les informations normalement transmises dans le mode couplé par le code distant.

Dans ce contexte, il nous paraît intéressant de pouvoir piloter ces applications dans l'un ou l'autre de ces deux modes sans qu'il soit nécessaire de modifier l'instrumentation. Cette remarque nous incite donc à représenter chaque code séparément avec son propre MHT. Toutefois, une telle représentation ne permet pas de résoudre simplement le problème de la coordination des requêtes de pilotage inter-codes. En effet, l'absence d'un système de dates commun entre les MHTs couplés empêche de planifier une date de traitement cohérente, et même de comparer les versions des données entre les codes. La solution que nous envisageons consiste à introduire la notion de *tâches inter-codes*, en associant explicitement des tâches entre les MHTs couplés, comme par exemple T et T' sur la figure 8.1. Ainsi, grâce à la connaissance des deux MHTs et des tâches inter-codes, il doit être possible d'établir une date de traitement cohérente entre ces deux codes en étendant l'algorithme de planification que nous avons défini à la section 4.4.2 Grâce à la connaissance de cette date, il doit être possible d'effectuer des traitements globalement cohérents en temps entre tous les processus couplés, comme un *stop* ou un *multi-get* sur des données distribuées séparément sur les codes A et B. Il ne s'agit ici que d'un premier

élément de réponse à ce problème encore peu étudié, du moins dans le contexte du pilotage. Il nous invite à imaginer un environnement de couplage encore plus générique capable de prendre en compte les spécificités de ce type de pilotage.

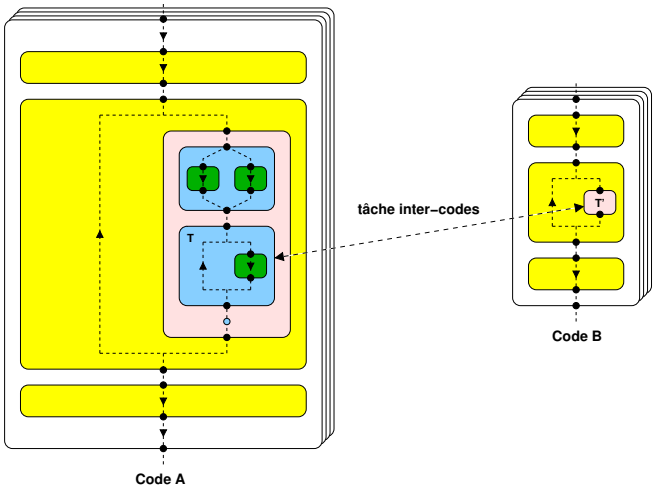


FIG. 8.1 – Définition d'une tâche inter-codes entre deux MHTs couplés.

8.2 Perspectives pour la redistribution des données

8.2.1 Approche spatiale

En nous appuyant sur une formulation ensembliste du problème de la redistribution, nous avons vu comment il était possible d'étendre le principe d'intersection pour des objets spatiaux distribués par blocs. En particulier, cela nous a permis d'établir des algorithmes de redistribution pour des grilles structurées ou des boîtes d'atomes. Cette approche peut être vue comme une généralisation des algorithmes de redistribution classiquement utilisés en algèbre linéaire pour des matrices denses à des cas plus irréguliers. Nous pensons également pouvoir étendre cette approche pour redistribuer des maillages non structurés dont la distribution serait issue d'une décomposition spatiale. Par exemple, il serait intéressant de prendre en compte la hiérarchie des blocs décrivant la distribution, comme on la trouve dans la technique de décomposition spatiale [176] ou dans les méthodes multipôles rapides [96] (utilisation d'un *octree* pour calculer des interactions particulaires). Actuellement, nous considérons deux niveaux dans la hiérarchie : le domaine global au premier niveau et les blocs distribués au second niveau (i.e. sous-domaine). La description d'une hiérarchie de blocs plus fine permettrait de réduire la complexité de l'algorithme de redistribution lors du calcul des intersections (élagage de sous-arbres). Notons que les blocs de redistribution ainsi calculés représentent eux-mêmes un niveau supplémentaire dans la hiérarchie. La gestion de la hiérarchie est une première évolution qui pourrait nous permettre à terme de traiter des cas plus complexes, dont notamment le cas des grilles multi-niveaux fréquemment utilisées en mécanique des fluides pour adapter la précision des calculs dans des sous-grilles raffinées.

En outre, il serait intéressant de pouvoir supporter à moindre coût le raffinement dynamique de la grille grâce à des opérations élémentaires d'ajout, de déplacement ou de suppression de blocs. Pour être efficace, ces opérations doivent être prises en compte par l'algorithme de redistribution qui effectue un recalcul partiel de la matrice de communication (et non total). Par ailleurs, la prise en compte d'une telle dynamicité pourrait également être intéressante dans le contexte de la visualisation en ligne. L'idée serait de mettre en place des techniques efficaces de « visualisation à la demande » afin de prendre en compte dynamiquement le champs de vision défini par l'utilisateur (Fig. 8.2). Ainsi, si l'utilisateur n'observe qu'un sous-domaine de la simulation (i.e. zoom), il apparaît tout à fait inutile de transmettre les données en dehors de son champs de vision.

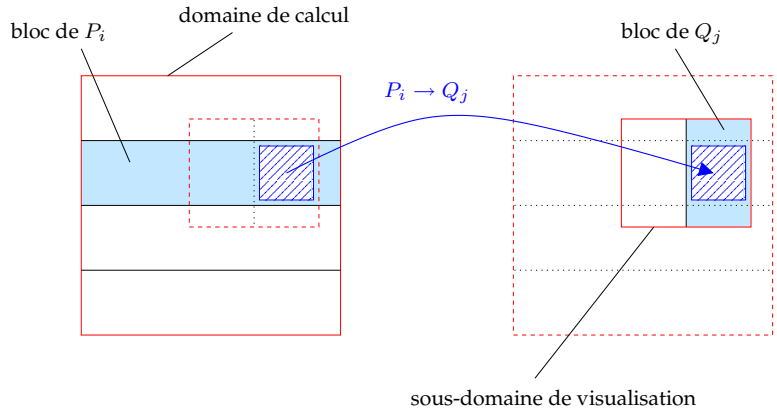


FIG. 8.2 – Fenêtre dynamique de visualisation.

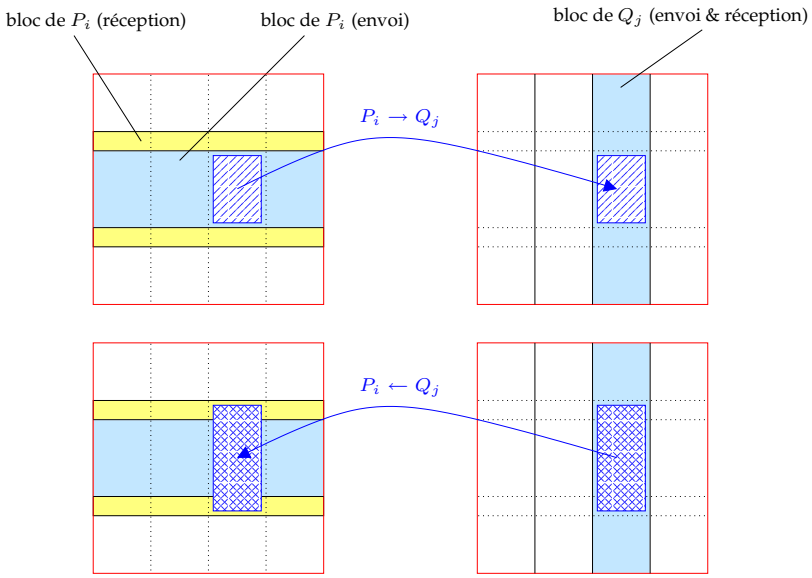


FIG. 8.3 – Prise en compte des « éléments fantômes » dans la redistribution des grilles structurées.

Une extension possible de nos travaux concerne aussi la gestion d'une matrice de communication différente pour l'envoi et la réception. En effet, cela peut s'avérer utile si les éléments que l'on souhaite partager (pour l'envoi) différent de ceux que l'on souhaite recevoir. Dans ce cas, chaque P_i calcule en fait deux messages symboliques un pour l'envoi et l'autre pour la réception. En particulier, nous avons appliqué cette stratégie dans le cas des grilles structurées en permettant à l'utilisateur de distinguer dans la distribution les blocs utilisés pour l'envoi uniquement, ceux utilisés pour la réception uniquement et ceux utilisés pour les deux (cf. Sec. 5.4.3). Cela nous permet, par exemple, de prendre en compte nativement des « éléments fantômes » dans la redistribution des grilles structurées. Ainsi ces éléments ne sont pas utilisés pour l'envoi, mais seront mis à jour à la réception, ce qui est en général laissé sous la responsabilité de l'utilisateur (Fig. 8.3). On peut également envisager d'appliquer cette stratégie dans le contexte du pilotage, afin de définir un sous-bloc de sélection pour modifier les données de la simulation, tout en continuant la visualisation du domaine global.

8.2.2 Approche placement

Nous avons également exploré une autre approche de la redistribution, basée sur un critère de placement plutôt que sur un critère d'intersection, et pouvant s'appliquer de manière très générale aux objets complexes distribués en régions logiques (blocs, composantes de maillages, ensembles de particules, *etc.*). Nous avons vu

que cette approche était particulièrement bien adaptée dans le contexte du pilotage. Dans ce cas, la distribution du code de visualisation peut être choisie dynamiquement en fonction de la distribution de la simulation, en réalisant un *placement* des régions d'origine vers le code distant, avec ou sans découpage. Cette approche possède de bonnes propriétés vis-à-vis de la dynamicité des éléments dans les régions sources, mais nécessite d'avoir recours à des mécanismes de réception des messages plus complexes que dans le cas des objets statiques (e.g. grilles structurés) où l'emplacement mémoire des données à recevoir peut être déterminé à l'avance. Le placement avec découpage permet de mieux équilibrer la charge des calculs. Si l'opérateur de découpage est *a priori* trivial dans le cas des ensembles de particules, il repose sur des techniques complexes de partitionnement pour les maillages non structurés. En contrepartie d'un meilleur équilibrage, les messages générés dans ce dernier cas sont relativement discontinus en mémoire, ce qui pénalisera les temps de transfert. La fonction de coût que nous avons proposée pour paramétrer le découpage est relativement naïve, puisqu'elle dépend linéairement du nombre d'éléments dans les régions sources. Cette fonction nous permet d'équilibrer simplement le nombre d'éléments placés sur le code distant (en supposant que les régions du code source sont à peu près équilibrées). Il serait intéressant de généraliser ce principe en permettant à l'utilisateur de définir sa propre fonction de coût, afin d'améliorer la qualité du placement.

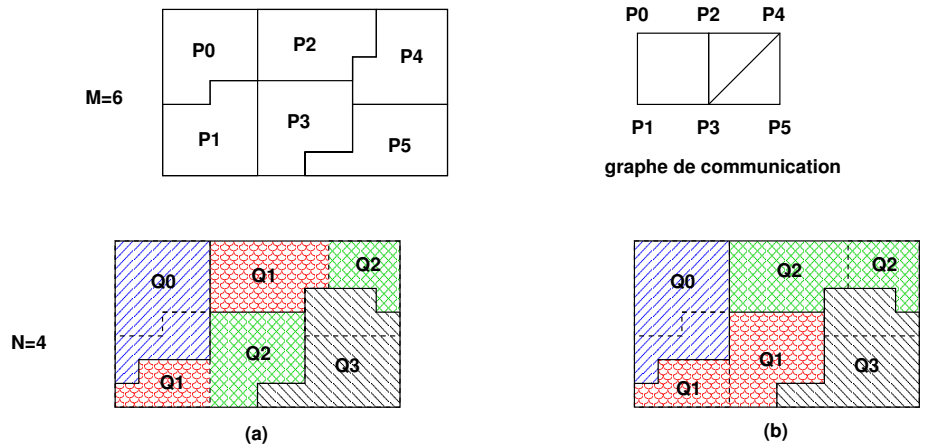


FIG. 8.4 – Prise en compte du graphe de communication dans le calcul du placement. Le placement (b) utilise les relations de voisinage entre régions induites par ce graphe pour améliorer la qualité du placement. Le placement (a) correspond à la stratégie actuelle, obtenu en supposant que le graphe de communication est une chaîne.

Un autre aspect permettant d'améliorer la qualité du placement serait de prendre en compte le graphe de communication dans le code afin de reconstruire une relation de voisinage (ou de connexité) entre les régions distribuées d'un objet complexe. Cette connaissance permettrait de respecter « une certaine localité » lors du placement de plusieurs sous-régions vers le même processeur (Fig 8.4). Actuellement, le placement que nous calculons ne prend pas en compte cette information et le graphe de communication est considéré comme une simple chaîne de processeurs (ordre croissant). Par ailleurs, il faut souligner que la définition de ce graphe peut être déduite à partir de la description des zones de recouvrement entre les régions du code de simulation, une information qui est également pertinente pour effectuer une gestion correcte des éléments fantômes dans la redistribution (i.e. cas des doublons). Pour conclure, notons que le calcul du placement pourrait avantageusement exploiter une information de plus haut-niveau conditionnant le graphe de communication et décrivant les régions selon une structure hiérarchique (octree, arbre de partitionnement).

8.3 Évolution de la plate-forme EPSN

Pour conclure sur ce chapitre, nous donnons quelques pistes d'évolutions possibles de la plate-forme EPSN. Une première évolution consisterait à remplacer la couche de transfert d'EPSN, c'est-à-dire RedCORBA par une couche de communication plus performante. L'idée est de développer une nouvelle couche de communication au dessus de RedSYM pour effectuer le transfert des données entre les codes couplés. Même

si EPSN est actuellement dépendant de RedCORBA, cette évolution ne devrait poser *a priori* aucune difficulté, car cela reviendrait essentiellement à définir une nouvelle implantation des méthodes *get* et *put* associées aux *PortChannels* en utilisant une autre technologie de communication comme par exemple MPI. Afin de valider l'indépendance de RedSYM vis-à-vis de la couche de communication, nous avons réalisé un prototype appelé RedMPI : ce prototype utilise une couche de communication hybride basée sur (Red)CORBA pour l'échange des descripteurs IDL et la gestion des requêtes, et sur MPI pour la réalisation des transferts. Les messages symboliques générés par RedSYM sont directement traduits dans le type `MPI_DataType`. Les messages MPI ainsi construits peuvent être utilisés pour effectuer des communications point-à-point (*send/recv*), mais il serait également possible de définir des routines de communications collectives de redistribution comparables à un *all-to-all* personnalisé. L'inconvénient majeur de cette approche est qu'elle nécessite de disposer d'un communicateur MPI global incluant statiquement tous les processus des codes couplés. Si cette contrainte peut être acceptable dans le cadre du couplage, elle ne l'est pas dans le cadre du pilotage (connexion dynamique des codes clients). La solution que nous envisageons dans le futur serait d'utiliser les possibilités dynamiques de MPI-2 (notion d'inter-communicateurs). Par ailleurs, MPI-2 dispose du paradigme de communication *one-sided*, tout à fait adapté à l'utilisation que nous en faisons dans EPSN, prenant nativement en compte la gestion relativement complexe des plages d'accès aux données. Nous espérons qu'une telle évolution pourrait considérablement améliorer la performance des transferts. Une autre piste possible pour améliorer les performances d'EPSN serait d'introduire dans CORBA une nouvelle sémantique permettant à l'utilisateur de gérer lui-même l'adressage en mémoire des arguments reçus lors des invocations de méthodes distantes. Une telle approche permettrait d'effectuer des réceptions de données en mode zéro-copie. La plate-forme EPSN est actuellement indépendante du choix de l'ORB qui peut être changé. Même si nous avons privilégié OmniORB4 dans nos discussions pour ses bonnes performances, il est tout à fait possible d'en utiliser d'autres et nous avons déjà testé Mico2 et Orbacus. A ce titre, il serait intéressant d'utiliser PadicoTM [69], dont le principal avantage à nos yeux est qu'il va permettre d'exploiter CORBA[1] sur des réseaux rapides comme Myrinet.

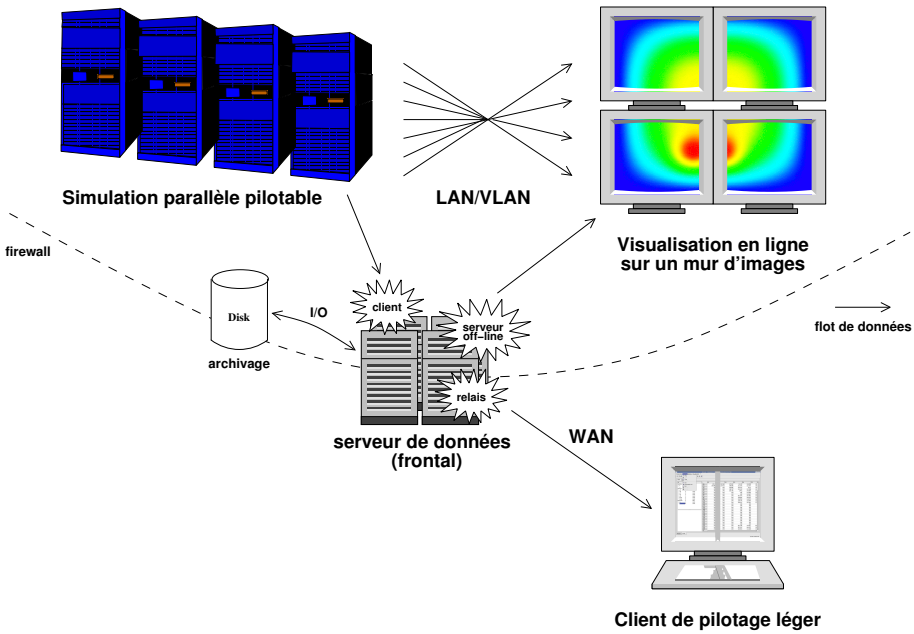


FIG. 8.5 – Mise en place d'un serveur de données (parallèle) dans la plate-forme EPSN, jouant le rôle double d'un client d'archivage et d'une « fausse » simulation capable de « rejouer » hors-ligne l'historique de la vraie simulation.

La deuxième évolution envisagée pour la plate-forme EPSN serait de développer un client particulier, jouant le rôle d'un serveur central de données (éventuellement parallèle) auquel d'autres clients pourraient se connecter pour « rejouer » l'historique de la simulation hors-ligne (Fig 8.5). Cette fonctionnalité nous apparaît tout aussi importante que la visualisation en ligne, en particulier lorsque la simulation est « longue ».

[1] Les implantations de CORBA traditionnelles n'exploitent que des réseaux de type Ethernet.

Dans ce cas, EPSN permettrait à l'utilisateur d'effectuer de la surveillance : il se connecte, se déconnecte périodiquement et observe l'état d'avancement des calculs qu'il souhaite éventuellement piloter. Dans ce contexte, il serait intéressant de pouvoir consulter l'historique des calculs effectués, en faisant « rejouer » fictivement l'exécution de la simulation. Cela nécessite d'une part de mémoriser les traces d'exécution du MHT, et d'autre part d'archiver automatiquement les données sur disque. Le serveur de données serait ainsi capable de recharger en mémoire les étapes précédentes du calcul pour les transmettre à un client de pilotage (visualisation hors-ligne). Dans ce cas, aucune interaction de pilotage n'est *a priori* possible, mais on pourrait envisager de coupler le serveur de données à un mécanisme de *checkpoint/restart* de la simulation, une possibilité déjà étudiée dans CUMULVS [122]. Pour réaliser l'archivage des données, nous comptons développer la bibliothèque RedSYM2HDF5 (comparable à RedSYM2IDL) qui serait capable de traduire les objets complexes RedSYM en fichier au format HDF5 [156]. L'avantage d'un tel format est qu'il est standard, hiérarchique et « auto-descriptif ». Ainsi, il doit être possible de stocker un objet complexe dans un fichier à ce format puis de le recharger en mémoire de manière tout à fait générique. Un tel serveur de données présente également un autre intérêt pour rediriger les données vers un client de pilotage situé à l'extérieur du cluster de calcul, typiquement protégé par un *firewall* (Fig. 8.5). Ce serveur, placé sur le frontal du cluster, jouera alors le rôle d'un relais permettant aux clients connectés de piloter la simulation, d'effectuer de la visualisation en-ligne ou hors-ligne. Cependant, cette solution ne permet pas de régler tous les problèmes classiques liés à la sécurité et pour lesquels nous souhaitons reposer sur des solutions existantes (e.g. tunnel SSH, *etc*.). D'une manière générale, il reste beaucoup de travail pour permettre de déployer simplement EPSN sur une grille de calcul. Pour la réalisation de communication longue distance (WLAN), il serait tout à fait pertinent d'implanter au niveau de ce serveur des schémas de compression et de filtrage des données exportées.

Une dernière évolution importante de la plate-forme EPSN serait de coupler les simulations avec des équipements de réalité virtuelle. Nous avons déjà étudié la possibilité de coupler une simulation à un mur d'images, grâce notamment à VTK et à ICE-T. Mais, il serait également intéressant de réaliser des expériences avec d'autres équipements de RV [103, 86] comme un WorkBench ou un CAVE (Fig. 8.6), et d'autres technologies comme par exemple SGI Performer [29] ou NetJuggler [38]. Un autre aspect qui mériterait encore beaucoup de travail est la réalisation d'interaction de haut-niveau par manipulation directe de l'image. Cet aspect est d'autant plus fondamental qu'il favorise l'immersion des utilisateurs dans un environnement virtuel. A court terme, nous envisageons de développer de telles interactions grâce à l'utilisation de *widgets* 3D dans VTK (cf. Sec. 3.2.3). A plus long terme, il serait intéressant de commander les interactions de pilotage à partir de périphériques de réalité virtuelle : stylet 3D (*wand*), périphériques haptiques à retour d'effort, *etc*.

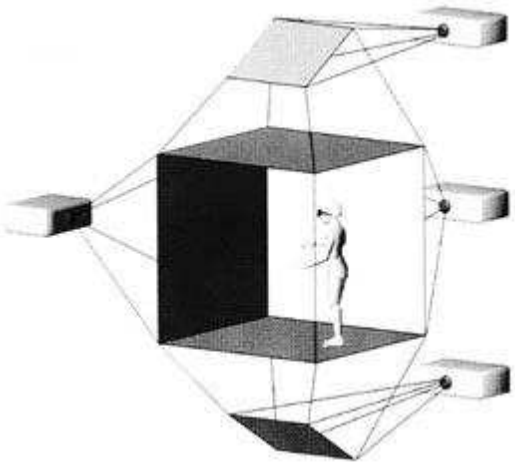


FIG. 8.6 – The CAVE

Annexe A

Les APIs d'EPSN

A.1 Extrait de l'API back-end pour l'instrumentation d'une simulation

```
1  // The EPSN Simulation Interface
2  class SimulationInterface {
3
4    /* ****************** Life Cycle Management ****************** */
5
6    // Initialize CORBA
7    Status initORB(int argc, char* argv[]);
8    // Initialize an EPSN port on simulation side
9    Status initPort(char * simulation_name, int num_port, int nb_ports);
10   // Initialize the EPSN proxy on simulation side
11   Status initProxy(char * simulation_name, char * xml_filename, int nb_ports);
12   // Set EPSN ready
13   Status ready();
14   // Terminate EPSN
15   Status finalize();
16   // Kill CORBA
17   Status killORB();
18
19   /* ****************** Data/Action Management ****************** */
20
21   // Add a new data
22   Status addData(RedSYM::Object * redsym_object);
23   // Update a data
24   Status updateData(char * data_id);
25   // Set the callback function of an action
26   Status setCallback(char * action_id, Callback * callback);
27   // Set the callback argument of an action
28   Status setCallbackArg(char * action_id, void * callback_arg);
29
30   /* ****************** Instrumentation Points ****************** */
31
32   // Begin the simulation MHT
33   Status beginMHT();
34   // End the simulation MHT
35   Status endMHT();
36   // Begin a simple task
37   Status beginTask(char * task_name);
38   // End a simple task
39   Status endTask(char * task_name);
40   // Begin a loop task
41   Status beginLoop(char * task_name);
42   // End a loop task
43   Status endLoop(char * task_name);
44   // Begin a switch task
45   Status beginSwitch(char * task_name);
46   // End a switch task
47   Status endSwitch(char * task_name);
48   // Cross a task point
49   Status point(char * task_name);
50
51 };
```

A.2 Extrait de l'API front-end pour le pilotage d'une simulation

```
1  // The EPSN Client Interface
2  class ClientInterface {
3
4    /* ***************** Life Cycle Management ***************** */
5
6    // Initialize CORBA
7    static Status initORB(int argc, char* argv[]);
8    // Initialize the EPSN proxy on client side and connect a remote simulation
9    Status initProxy(char* client_name, char* simulation_name, int nb_ports);
10   // Initialize an EPSN port on client side
11   Status initPort(char* client_name, int num_port, int nb_ports);
12   // Set the client ready to start a steering session
13   Status ready();
14   // Terminate EPSN
15   Status finalize();
16   // Kill CORBA
17   static Status killORB();
18
19   /* ***************** Data Management ***************** */
20
21   // get the simulation data description
22   RedSYM::ParallelObject * getRemoteDataDesc(char* data_id);
23   // add a new redsym data object
24   Status addData(RedSYM::Object * redsym_object);
25   // update the redCORBA data according to the redsym data object
26   Status updateData(char* data_id);
27   // lock read access to the data object (including all series)
28   Status lockReadAccess(char* data_id);
29   // unlock read access to the data object (including all series)
30   Status unlockReadAccess(char* data_id);
31   // lock write access to the data object (including all series)
32   Status lockWriteAccess(char* data_id);
33   // unlock write access to the data object (including all series)
34   Status unlockWriteAccess(char* data_id);
35
36   /* ***************** Request Management ***************** */
37
38   // Play the simulation
39   Status play();
40   // Play the simulation for some steps
41   Status step(int nb_steps);
42   // Stop the simulation
43   Status stop();
44   // Get the current date of all simulation processes
45   Date * getDate();
46   // Get a data serie from the simulation
47   RequestID get(char* data_id, char* serie_id, bool explicit_ack);
48   // Get periodically a data serie from the simulation
49   RequestID getp(char* data_id, char* serie_id, int period, bool explicit_ack);
50   // Put a data serie to the simulation
51   RequestID put(char* data_id, char* serie_id);
52   // Trigger an action on the simulation
53   RequestID action(char* action_id, int period);
54   // Wait for the end of a request
55   bool wait(RequestID request_id);
56   // Test if a request is ended
57   bool test(RequestID request_id);
58   // Ack explictly a request
59   Status ack(RequestID request_id);
60   // Cancel a request
61   Status cancel(RequestID request_id);
62
63 };
```

A.3 Format DTD des fichiers description XML

```
<?xml version="1.0" encoding="UTF-8" ?>

<!ELEMENT simulation (data*,action*,MHT)>
<!ATTLIST simulation id ID #REQUIRED>

<!ELEMENT data (serie)*>
<!ATTLIST data id ID #REQUIRED
          class (parameter|grid|atombox|particles|mesh) #REQUIRED>

<!ELEMENT serie EMPTY>
<!ATTLIST serie id ID #REQUIRED>

<!ELEMENT action EMPTY>
<!ATTLIST action id ID #REQUIRED>

<!ELEMENT MHT (task|loop|switch|point|data-context|action-context|bench-context)*>
<!ATTLIST MHT auto-start (true|false) "true" >

<!ELEMENT task (task|loop|switch|point|data-context|action-context|bench-context)*>
<!ATTLIST task id ID #REQUIRED
          synchronize (no|auto|block) "no">

<!ELEMENT loop (task|loop|switch|point|data-context|action-context|bench-context)*>
<!ATTLIST loop id ID #REQUIRED
          synchronize (no|auto|block) "no">

<!ELEMENT switch (task|loop|switch|point|data-context|action-context|bench-context)*>
<!ATTLIST switch id ID #REQUIRED
          synchronize (no|auto|block) "no">

<!ELEMENT point (data-context|action-context|bench-context)*>
<!ATTLIST point id ID #REQUIRED
          synchronize (no|auto|block) "no">

<!ELEMENT data-context EMPTY>
<!ATTLIST data-context ref IDREF #REQUIRED
          context (readable|writable|protected|modified) #REQUIRED>

<!ELEMENT action-context EMPTY>
<!ATTLIST action-context ref IDREF #REQUIRED
          context (allowed|forbidden) #REQUIRED>

<!ELEMENT bench-context EMPTY>
<!ATTLIST bench-context frequency NMTOKENS "1">
```

Bibliographie

[1] Amira : An Advanced 3D Visualization and Volume Modeling System. http ://www.amiravis.com.

[2] AVS : Advanced Visual Systems Inc. http ://www.avs.com.

[3] CACTUS. http ://www.cactuscode.org.

[4] CCA : Common Component Architecture Forum. http ://www.csm.ornl.gov/cca.

[5] CCA M×N. http ://www.csm.ornl.gov/cca/mxn.

[6] CCSM : Community Climate System Model. http ://www.ccsm.ucar.edu.

[7] Clawpack : Conservation Law Package. http ://www.amath.washington.edu/ claw.

[8] Distributed Interactive Engineering Toolbox (DIET). http ://graal.ens-lyon.fr/DIET.

[9] DynInst : An Application Program Interface (API) for Runtime Code Generation. http ://www.dyninst.org.

[10] Ensight. http ://www.ceintl.com/products/ensight.html.

[11] ESMF : Earth System Modeling Framework. http ://www.esmf.ucar.edu.

[12] Gadget-2 : A Code for Cosmological Simulations of Structure Formation. http ://www.mpa-garching.mpg.de/gadget.

[13] Genome@Home. http ://www.stanford.edu/group/pandegroup/genome.

[14] Grid'5000. http ://www.grid5000.org.

[15] IPARS-NetSolve-DISCOVER. http ://www.ices.utexas.edu/ ut/webipars.

[16] MCT : The Model Coupling Toolkit. http ://www-unix.mcs.anl.gov/mct.

[17] Metis. http ://www-users.cs.umn.edu/ karypis/metis.

[18] MICO : MICO is CORBA. http ://www.mico.org.

[19] MpCCI : Mesh-based parallel code coupling Interface. http ://www.mpcci.org.

[20] NetCDF : Network Common Data Form. http ://my.unidata.ucar.edu/content/software/netcdf/index.html.

[21] OASIS. http ://www.cerfacs.fr/globc/software/oasis/oasis.html.

[22] OMG : Object Management Group. http ://www.omg.org.

[23] OmniORB. http ://omniorb.sourceforge.net.

[24] OpenInventor. http ://oss.sgi.com/projects/inventor.

[25] ParaView. http ://www.paraview.org.

[26] PDB : Protein Data Bank. http ://www.rcsb.org/pdb.

[27] RMI over IIOP. http ://java.sun.com/products/rmi-iiop.

[28] Seti@Home. http ://setiathome.ssl.berkeley.edu.

[29] Silicon Graphics Computer Systems, IRIS Performer. http ://www.sgi.com/software/performer.

[30] VMD : Visual Molecular Dynamics. http ://www.ks.uiuc.edu/Research/vmd.

[31] VRML : Virtual Reality Modeling Language. http ://www.w3.org/MarkUp/VRML.

[32] VTHD : Vraiment Très Haut Débit. http ://www.vthd.org.

[33] VTK : The Visualization Toolkit. http ://www.vtk.org.

[34] Data Parallel CORBA Specification. OMG : Object Management Group, November 2001.

[35] J. Ahrens, C. Law, W. Schroeder, K. Martin, and M. Papka. A Parallel Approach for Efficiently Visualizing Extremely Large, Time-Varying Datasets. Technical Report LAUR-001630, Los Alamos National Laboratory, 2000.

[36] Alcatel and FPX. MicoCCM. http ://www.fpx.de/MicoCCM.

[37] B. A. Allan, R. C. Armstrong, A. P. Wolfe, J. Ray, D. E. Bernholdt, and J. Arthur Kohl. The CCA Core Specification in a Distributed Memory SPMD Framework. *Concurrency and Computation : Practice and Experience*, 14(5) :323–345, 2002.

[38] J. Allard, B. Raffin, and F. Zara. Coupling Parallel Simulation and Multi-display Visualization on a PC Cluster. In *Euro-Par 2003*, Klagenfurt, Austria, August 2003.

[39] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Toward a Common Component Architecture for High-Performance Scientific Computing. In *HPDC '99 : Proceedings of the The Eighth IEEE International Symposium on High Performance Distributed Computing*. IEEE Computer Society, 1999.

[40] O. Aumage, L. Bouge, J.-F. Mehaut, and R. Namyst. Madeleine II : a Portable and Efficient Communication Library for High-Performance Cluster Computing. *Parallel Computing*, 28(4) :607–626, 2002.

[41] J.M. Autebert. *Langages algébriques*. Études et Recherches en Informatique. Masson, 1987.

[42] D. M. Beazley. Simplified Wrapper and Interface Generator (SWIG). http ://www.swig.org.

[43] D. M. Beazley and P. S. Lomdahl. Lightweight Computational Steering of Very Large Scale Molecular Dynamics Simulations. In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing (CDROM)*. ACM Press, 1996.

[44] P. H. Beckman, P. K. Fasel, W. F. Humphrey, and S. M. Mniszewski. Efficient Coupling of Parallel Applications Using PAWS. *The Tenth IEEE International Symposium on High Performance Distributed Computing (HPDC)*, 1998.

[45] T. Beisel, E. Gabriel, and M. Resch. An Extension to MPI for Distributed Computing on MPPs. In *Proceedings of the 4th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 75–82. Springer-Verlag, 1997.

[46] J. K. Bennett. The Design and Implementation of Distributed Smalltalk. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 22, pages 318–330, New York, NY, 1987. ACM Press.

[47] P. A. Bernstein. Middleware : A Model for Distributed System Services. *Commun. ACM*, 39(2) :86–98, 1996.

[48] F. Bertrand, R. Bramley, K. B. Damevski, J. A. Kohl, D. E. Bernholdt, J. W. Larson, and A. Sussman. Data Redistribution and Remote Method Invocation in Parallel Component Architectures. In *Proceedings of the 19th International Parallel and Distributed Processing Symposium : IPDPS 2005*, 2005.

[49] F. Bertrand, Y. Yuan, K. Chiu, and R. Bramley. An Approach to Parallel MxN Communication. In *12th High Performance Distributed Computing (HPDC)*, Seattle, WA, June 2003.

[50] L.S. Blackford, J. Choi, A. Cleary, E. D'Azevado, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R.C. Whaley. ScaLAPACK Users'Guide. Technical report, SIAM, 1997.

[51] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, and H. Nielsen. Simple Object Access Protocol (SOAP) 1.1. http ://www.w3.org/TR/SOAP, may 2000.

[52] K. Brodlie and J. Wood. Recent Advances in Volume Visualization. In *Computer Graphics*, 2001.

[53] J. M. Brooke, P. V. Coveney, J. Harting, S. Jha, S. M. Pickles, R. L. Pinning, and A. R. Porter. Computational Steering in RealityGrid. In *Proceedings of the UK e-Science All Hands Meeting*, 2003.

[54] J. M. Brooke, T. Eickermann, and U. Woessner. Application Steering in a Collaborative Environment. In *SC'03, Phoenix, Arizona, USA*, november 2003.

[55] G. Burns, R. Daoud, and J. Vaigl. LAM : An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[56] C. Calidonna, M. Giordano, and M. Furnari. A Graphic Parallelizing Environment for User-Compiler interaction. In *ICS '99 : Proceedings of the 13th International Conference on Supercomputing*, pages 238–245, New York, NY, USA, 1999. ACM Press.

[57] H. Casanova and J. Dongarra. NetSolve : A Network Server for Solving Computational Science Problems. Technical Report CS-96-328, Knoxville, TN 37996, USA, 1996.

[58] E. E. Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, Dept. of CS, U. of Utah, December 1974.

[59] F. Chatzinikos and H. Wright. Computational Steering by Direct Image Manipulation. *VMV*, pages 455–462, 2001.

[60] D. Cheng and R. Hood. A Portable Debugger for Parallel and Distributed Programs. In *Proceedings of the 1994 conference on Supercomputing*, pages 723–732. IEEE Computer Society Press, 1994.

[61] B. D. Conner, S. S. Snibbe, K. P. Herndon, D. C. Robbins, R. C. Zeleznik, and A. van Dam. Three-Dimensional Widgets. In *SI3D '92 : Proceedings of the 1992 symposium on Interactive 3D graphics*, pages 183–188. ACM Press, 1992.

[62] T.W. Crokett. An Introduction to Parallel Rendering. In *Parallel Computing*, volume 23, pages 819–843, july 1997.

[63] C. Cruz-Neira, D. J. Sandin, T. A. DeFanti, R. V. Kenyon, and J. C. Hart. The CAVE : Audio Visual Experience Automatic Virtual Environment. *Commun. ACM*, 35(6) :64–72, 1992.

[64] M. Czernuszenko, D. Pape, D. Sandin, T. DeFanti, G. L. Dawe, and M. D. Brown. The ImmersaDesk and Infinity Wall Projection-Based Virtual Reality Displays. *SIGGRAPH Comput. Graph.*, 31(2) :46–49, 1997.

[65] K. Damevski and S. G. Parker. Parallel Remote Method Invocation and M-by-N Data Redistribution. High-performance Distributed Computing. In *12th High-Performance Distributed Computing Conference (HPDC)*, 2003.

[66] A. Denis. CORBA et réseaux haute performance. In *13es Rencontres Francophones du Parallélisme (RenPar 13)*, pages 189–194, Paris, France, April 2001.

[67] A. Denis, C. Pérez, and T. Priol. Portable Parallel CORBA Objects : An Approach to Combine Parallel and Distributed Programming for Grid Computing. In *Proc. of the 7th Intl. Euro-Par'01 Conference (EuroPar'01)*, pages 835–844, 2001.

[68] A. Denis, C. Pérez, and T. Priol. Towards High Performance CORBA and MPI Middlewares for Grid Computing. In *Proceedings of 2nd IWGC*, pages 14–25, 2001.

[69] A. Denis, C. Pérez, and T. Priol. PadicoTM : An Open Integration Framework for Communication Middleware and Runtimes. *Future Generation Computer Systems*, 19 :575–585, 2003.

[70] F. Desprez, S. Domas, J.J. Dongarra, A. Petitet, C. Randriamaro, and Y. Robert. More on Scheduling Block-Cyclic Array Redistribution. In *Proceedings of 4th Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers (LCR98)*, volume 1151 of *LNCS*, pages 275–287. Springer-Verlag, 1998.

[71] F. Desprez, J.J. Dongarra, A. Petitet, C. Randriamaro, and Y. Robert. Scheduling Block-Cyclic Array Redistribution. *IEEE Transaction on Parallel and Distributed Systems*, 9(2) :192–205, 1998.

[72] G.O. Domik. Scientific Visualization : An introduction. Technical report, University of Colarodo at Boulder, november 1995.

[73] J. Dongarra, R. Geijn, and D. Walker. LAPACK Working Note 43 : A Look at Scalable Dense Linear Algebra Libraries. Technical report, University of Tennessee, 1992.

[74] R. A. Drebin, L. Carpenter, and P. Hanrahan. Volume Rendering. In *Computer Graphics, Proceedings of SIGGRAPH 88*, volume 22, pages 65–74, 1988.

[75] G. Edjlali, A. Sussman, and J. H. Saltz. Interoperability of Data Parallel Runtime Libraries. In *IPPS '97 : Proceedings of the 11th International Symposium on Parallel Processing*, pages 451–459, Washington, DC, USA, 1997. IEEE Computer Society.

[76] G. E.Fagg and J. J. Dongarra. PVMPI : An Integration of the PVM and MPI Systems. Technical report, University of Tennessee, 1996.

[77] T. Eickmann and W. Frings. VISIT : a Visualization Interface Toolkit. http ://www.kfa-juelich.de/zam/visit, 2000.

[78] G. Eisenhauer, B. Plale, and K. Schwan. DataExchange : High-Performance Communications in Distributed Laboratories. *Parallel Computing*, 24(12–13) :1713–1733, 1998.

[79] G. Eisenhauer and K. Schwan. An Object-Based Infrastructure for Program Monitoring and Steering. In *2nd SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT'98)*, pages 10–20, 1998.

[80] J. M. Favre. Large Data and Distributed Visualization with the Visualization Toolkit (VTK). *EPFL Supercomputing Review*, pages 7–11, january 2004.

[81] G. Fedak, C. Germain, V. Neri, and F. Cappello. XtremWeb : A Generic Global Computing System. In *CCGRID '01 : Proceedings of the 1st International Symposium on Cluster Computing and the Grid*. IEEE Computer Society, 2001.

[82] W. Felger and F. Schroder. The Visualization Input Pipeline - Enabling Semantic Interaction in Scientific Visualization. *Computer Graphics Forum*, 11(3) :139–139, 1992.

[83] I. Foster and C. Kesselman. Globus : A Metacomputing Infrastructure Toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2) :115–128, Summer 1997.

[84] I. Foster and C. Kesselman. *The Grid : Blueprint for a new Computing Infrastructure*. Morgan Kaufman Publishers, 1998.

[85] D. Foulser. IRIS Explorer : A Framework for Investigation. *SIGGRAPH Comput. Graph.*, 29(2) :13–16, 1995.

[86] P. Fuchs, G. Moreau, B. Arnaldi, J.M. Burkhardt, A. Chauffaut, S. Coquillart, S. Donikian, T. Duval, J. Grosjean, F. Harrouet, E. Klinger, D. Lourdeaux, D. Mellet d'Huart, A. Paljic, J. P. Papin, P. Stergiopoulos, J.Tisseau, and I. Viaud-Delmon. *Le traité de la réalité virtuelle*. Les presses de l'école des Mines de Paris, 2001.

[87] M. R. Garey and D. S. Johnson. *Computers and Intractablility : A Guide to the Theory of NP-Completeness*. W. H. Freeman, San Francisco, 1979.

[88] T. Gautier, O. Coulaud, and H. Hamid Reza. *Etat des lieux en architecture, parallélisme et système*, chapter Couplage de codes parallèles. Hermes Science Publications, march 2004.

[89] T. Gautier and H. Hamid Reza. HOMA : un compilateur IDL optimisant les communications de données pour la composition d'invocation de méthodes CORBA. In *15ème Rencontres francophones du parallélisme (RenPar'15)*, pages 127–134, 2003.

[90] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. S. Sunderam. *PVM : Parallel Virtual Machine : A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, MA, USA, 1994.

[91] W. L. George, J. G. Hagedorn, and J. E. Devaney. IMPI : Making MPI Interoperable. *Journal of Research of the National Institute of Standands and Technology*, 105(3) :343–428, 2000.

[92] M. Girkar and C. Polychronopoulos. The Hierarchical Task Graph as a Universal Intermediate Representation. *International Journal of Parallel Programming*, 22(5) :519 – 551, 1994.

[93] C. M. Goral, K. E. Torrance, D. P. Greenberg, and B. Battaile. Modeling the Interaction of Light between Siffuse Surfaces. In *SIGGRAPH '84 : Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, pages 213–222. ACM Press, 1984.

[94] H. Gouraud. Continuous Shading of Curved Surfaces. 20(6) :623–628, 1971.

[95] M. Govindaraju, S. Krishnan, K. Chiu, A. Slominski, D. Gannon, and R. Bramley. Merging the CCA Component Model with the OGSI Framework. In *CCGRID '03 : Proceedings of the 3st International Symposium on Cluster Computing and the Grid*. IEEE Computer Society, 2003.

[96] L. Greengard and V. Rokhlin. A Fast Algorithm for Particle Simulations. *Journal of Computational Physics*, 73 :325–348, 1987.

[97] A. Grimshaw, A. Ferrari, F. Knabe, and M. Humphrey. Legion : An Operating System for Wide-Area Computing. Technical report, 1999.

[98] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6) :789–828, September 1996.

[99] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI (2nd ed.) : Portable Parallel Programming with the Message-Passing Interface*. MIT Press, Cambridge, MA, USA, 1999.

[100] W. Gu, G. Eisenhauer, K. Schwan, and J.Vetter. Falcon : On-line Monitoring for Steering Parallel Programs. *Concurrency : Practice and Experience*, 10(9) :699–736, 1998.

[101] R. Haber, B. Bliss, D. Jablonowski, and C. Jog. A Distributed Environment for Run-Time Visualization and Application Steering in Computational Mechanics. *Computing Systems in Engineering*, pages 501–515, 1992.

[102] R.B. Haber and D.A. McNabb. Visualization Idioms : A Conceptual Model for Scientific Visualization Systems. *IEEE Computer Society Press*, pages 74–93, 1990.

[103] M. Hachet, J.-B. de la Rivière, and P. Guitton. Interaction in Large-Display VR Environments. In *Proceedings of Virtual Concept*, November 2003.

[104] S. Hackstadt, C. Harrop, and A. Malony. A Framework for Interacting with Distributed Programs and Data. In *HPDC*, pages 206–214, 1998.

[105] C. Hansen and P. Hinker. Massively Parallel Isosurface Extraction. In IEEE Computer Society Press, editor, *Proceedings of Visualization '92*, pages 77–83, 1992.

[106] D. Hart and E. Kraemer. Consistency Considerations in the Interactive Steering of Computations. In *International Journal of Parallel and Distributed Networks and Systems*, 1999.

[107] P. S. Heckbert and M. Garland. Surface Simplification using Quadric Error Metrics. In *ACM SIGGRPAH '97*, pages 209–216, 1997.

[108] P. S. Heckbert and M. Garland. Survey of Polygonal Simplification Algorithms. In *SigGraph '97*, 1997.

[109] P. S. Heckbert and P. Hanrahan. Beam Tracing Polygonal Objects. In *Computer Graphics, SIGGRAPH '84 Proceedings*, volume 18, pages 119–127, July 1984.

[110] P. Hinker and C. Hansen. Geometric Optimization. In IEEE Computer Society Press, editor, *Proceedings of Visualization '93*, 1993.

[111] G. Humphreys, M. Houston, Y. Ng, R. Frank, S. Ahern, P. Kirchner, and J. Klosowski. Chromium : A Stream Processing Framework for Interactive Graphics on Clusters. SIGGRAPH, San Antonio, Texas, 2002.

[112] IBM. OpenDX. http ://www.research.ibm.com/dx.

[113] D. Jablonowski, J. Bruner, B. Bliss, and R. Haber. VASE : The Visualization and Application Steering Environment. In *Proceedings of Supercomputing'93*, pages 560–569, 1993.

[114] E. Jeannot and F. Wagner. Two Fast and Efficient Message Scheduling Algorithms for Data Redistribution through a Backbone. *IPDPS*, 2004.

[115] N. Karonis, B. Toonen, and I. Foster. MPICH-G2 : A Grid-Enabled Implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing (JPDC)*, 63(5) :551–563, may 2003.

[116] G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. Technical Report 95-035, University of Minnesota, june 1995.

[117] G. Karypis and V. Kumar. A Coarse-Grain Parallel Formulation of Multilevel k-way Graph-partitioning Algorithm. In *Proc. 8th SIAM Conference on Parallel Processing for Scientific Computing*, 1997.

[118] K. Keahey, P. K. Fasel, and S. Mniszewski. PAWS : Collective Interactions and Data Transfers. *The Tenth IEEE International Symposium on High Performance Distributed Computing (HPDC), San Francisco, California*, pages 47–54, 7–9 August 2001.

[119] K. Keahey and D. Gannon. PARDIS : A Parallel Approach to CORBA. In *HPDC*, pages 31–39, 1997.

[120] K. Keahey and D. Gannon. PARDIS : CORBA-Based Architecture for Application-Level Parallel Distributed Computation. In *Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–14. ACM Press, 1997.

[121] C. Koebel, D. Loveman, R. Schreiber, G. Steele Jr., and M. Zosel. *The High Performance Fortran Handbook*. MIT Press, Cambridge, 1994.

[122] J. A. Kohl and P. M. Papadopoulos. CUMULVS : Providing fault-tolerance, Visualization, and Steering of Parallel Applications. *Int. J. of Supercomputer Applications and High Performance Computing*, pages 224–235, 1997.

[123] E. Kuo, M. Lanzagorta, R. Rosenberg, and S. Julier. The avs vr viewer.

[124] Los Alamos National Laboratory. POP : Parallel Ocean Program. http ://climate.lanl.gov/Models/POP.

[125] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7) :558–565, 1978.

[126] U. Lang, J.P. Peltier, P. Christ, S. Rill, D. Rantzau, H. Nebel, A. Wierse, R. Lang, S. Causse, F. Juaneda, M. Grave, and P. Haas. *COVISE : Perspectives of Collaborative Supercomputing and Networking in European Aerospace Research and Industry*, volume 11. Future Generation Computer Systems (FGCS) Elsevier Science, 1995.

[127] C. Law. A Multithreaded Streaming Pipeline Architecture for Large Structured Meshes. *IEEE Visualization Proceedings*, 1999.

[128] C. C. Law, A. Henderson, and J. Ahrens. An application architecture for large data visualization : A case study.

[129] J.-Y. Lee and A. Sussman. Efficient Communication Between Parallel Programs with InterComm. Technical Report CS-TR-4557 and UMIACS-TR-2004-0, University of Maryland, Department of Computer Science and UMIACS, January 2004.

[130] S. Lo. The Implementation of a High Performance ORB over Multiple Network Transports. Middleware 98, 1998.

[131] W.E. Lorensen and H.E. Cline. Marching Cube : A High Resolution 3D Surface Reconstruction Algorithm. In *Computer Graphics*, volume 21, pages 163–169, 1987.

[132] A. Malony and S. Hackstadt. Performance of a System for Interacting with Parallel Applications. In *International Journal of Parallel and Distributed Systems and Networks*, 1999.

[133] V. Mann, V. Matossian, R. Muralidhar, and M. Parashar. DISCOVER : An environment for Web-based interaction and steering of high-performance scientific applications. *Concurrency and Computation : Practice and Experience*, 13(8-9) :737–754, 2001.

[134] R. Marshall, J. Kempf, S. Dyer, and C.-C. Yen. Visualization methods and simulation steering for a 3D turbulence model of Lake Erie. In *Proceedings of the 1990 symposium on Interactive 3D graphics*, volume 24, pages 89–97. ACM Press, 1990.

[135] B.H. McCormick, T.A. DeFanti, and M.D. Brown. Visualization in scientific computing. *Computer Graphics*, 21(6), november 1987.

[136] Message-Passing Interface Forum. *MPI-2.0 : Extensions to the Message-Passing Interface*. MPI Forum, june 1997.

[137] Microsoft. COM : Component Object Model Technologies. http ://www.microsoft.com/com.

[138] Microsoft. .NET. http ://www.microsoft.com/net.

[139] Sun Microsystems. eXternal Data Representation standard (XDR). http ://www.faqs.org/rfcs/rfc1014.html, june 1987.

[140] Sun Microsystems. Remote Procedure Call specification version 2 (RPC). http ://www.faqs.org/rfcs/rfc1057.html, june 1988.

[141] Sun Microsytems. EJB : Enterprise Java Beans. http ://java.sun.com/products/ejb.

[142] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn Parallel Performance Measurement Tool. *Computer*, 28(11) :37–46, 1995.

[143] D. Miller, J. Guo, E. Kraemer, and Y. Xiong. On-the-Fly Calculation and Verification of Consistent Steering Transactions. In *ACM/IEEE SC2001 Conference*, 2001.

[144] A. Modi, N. Sezer-Uzol, and L. N. Long. Scalable Computational Steering System for Visualization of Large-Scale CFD Simulations. In *Proceedings of AIAA Fluid Dynamics Meeting*, 2002.

[145] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A Sorting Classification of Parallel Rendering. *IEEE Comput. Graph. Appl.*, 14(4) :23–32, 1994.

[146] J. E. Morel. Massively Parallel Multiphysics Code Development. Los Alamos National Laboratory, 2003. number 28.

[147] K. Moreland and D. Thompson. From cluster to wall with VTK. *IEEE symposium on Parallel and Large-Data Visualization and Graphics*, pages 25–31, 2003.

[148] K. Moreland, B. Wylie, and C. Pavlakos. Sort-Last Parallel Rendering for Viewing Extremely Large Data Sets on Tile Displays. In *PVG '01 : Proceedings of the IEEE 2001 Symposium on Parallel and Large-Data Visualization and Graphics*, pages 85–92. IEEE Press, 2001.

[149] C. Morin, P. Gallard, R. Lottiaux, and G. Vallée. Towards an Efficient Single System Image Cluster Operating System. *Future Generation Computer Systems*, 20(2), January 2004.

[150] J. D. Mulder. *Computational steering with parametrized geometric objects*. PhD thesis, University of Amsterdam, june 1998.

[151] J. D. Mulder, R. van Liere, and J. J.van Wijk. Computational steering in the CAVE. *Future Generation Computer Systems*, 14(3–4) :199–207, 1998.

[152] J. D. Mulder and J. J. van Wijk. 3D computational steering with parametrized geometric objects. In *1523*, page 14. Centrum voor Wiskunde en Informatica (CWI), ISSN 0169-118X, 30 1996.

[153] J. D. Mulder, J. J. van Wijk, and R. van Liere. A survey of computational steering environments. *Future Generation Computer Systems*, 15(1) :119–129, 1999.

[154] R. Muralidhar, S. Kaur, and M. Parashar. An Architecture for Web-Based Interaction and Steering of Adaptive Parallel Distributed Applications. *Lecture Notes in Computer Science*, 1900, 2001.

[155] R. Muralidhar and M. Parashar. An interactive Object Infrastructure for Computational Steering of Distributed Simulations. Proceedings of the ninth IEEE International Symposium on High Performance Distributed Computing (HPDC), 2000.

[156] NCSA and University of Illinois. HDF5 : Hierarchical Data Format 5. http ://hdf.ncsa.uiuc.edu/HDF5.

[157] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global Arrays : A Portable Programming Model for Distributed Memory Computers. In *Supercomputing*, pages 340–349, 1994.

[158] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global Arrays : A Nonuniform Memory Access Programming Model for High-Performance Computers. *The Journal of Supercomputing*, 10(2) :169–189, 1996.

[159] B. Nkonga and P. Charrier. Generalized Parcel Method for Dispersed Spray and Message Passing Strategy on Unstructured Meshes. *Parallel Computing*, 28 :369–398, 2002.

[160] ObjectWeb. OpenCCM, The Open CORBA Component Model Platform. http ://openccm.objectweb.org.

[161] OMG. CORBA 2.6, The Common Object Request Broquer Architecture (CORBA), Specification 01-12-01, december 2001.

[162] OMG. CORBA Components, v3.0 (full specification), Specification 02-06-65, june 2002.

[163] OMG : Object Management Group. Common Object Request Broker Architecture Specification. http ://www.corba.org.

[164] The OpenGroup. Distributed Computing Environment (DCE). http ://www.opengroup.org/dce.

[165] P. Papadopoulos, J. Kohl, and B.D. Semeraro. CUMULVS : Extending a Generic Steering and Visualization Middleware for Application Fault-Tolerance. In *Proceedings of the 31st Hawaii International Conference on System Sciences (HICSS-31)*, 1998.

[166] D. Pape. CAVELib. http ://www.evl.uic.edu/pape/CAVE, 1996.

[167] S.G. Parker. *The SCIRun problem solving environment and computational steering software system*. PhD thesis, University of Utah, august 1999.

[168] S.G. Parker, D.M. Beazley, and C.R. Johnson. Computational Steering Software Systems and Strategies. *IEEE Computational Science and Engineering*, 4(4) :50–59, 1997.

[169] S.G. Parker and C.R. Johnson. SCIRun : a Scientific Programming Environment for Computational Steering. In *Proceedings of Supercomputing'95*, 1995.

[170] S.G. Parker, M. Miller, C. Hansen, and C.R. Johnson. An Integrated Problem Solving Environment : the SCIRun Computational Steering System. In *Hawaii International Conference of System Sciences*, pages 147–156, 1998.

[171] S.G. Parker, D.M. Weinstein, and C.R. Johnson. The SCIRun Computational Steering Software System. *Modern Software Tools for Scientific Computing*, pages 1–40, 1997.

[172] F. Pellegrini. Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs. In *Proc. SHPCC'94, Knoxville*, pages 486–493. IEEE, May 1994.

[173] F. Pellegrini. Graph Partitioning based Methods and Tools for Scientific Computing. *Parallel Computing ETPSC'3*, 23 :153–164, 1997.

[174] C. Pfister and C. Szyperski. Why Objects are Not Enough. In *Proceedings, International Component Users Conference*, Munich, Germany, 1996. SIGS.

[175] B. Tuong Phong. Illumination for Computer Generated Pictures. *Commun. ACM*, 18(6) :311–317, 1975.

[176] S. Plimpton and B. Hendrickson. Parallel Molecular Dynamics Algorithms for Simulation of Molecular Systems. In *Parallel Computing in Computational Chemistry*, 592, pages 114–135. American Chemical Society, 1995.

[177] C. Polychronopoulos, M. Gikar, M. Haghighat, C. Lee, B. Leung, and D. Schouten. The Structure of Parafrase-2 : an Advanced Parallelizing Compiler for C and FORTRAN. In *Selected Papers of the Second Workshop on Languages and Compilers for Parallel Computing*, pages 423–453. Pitman Publishing, 1990.

[178] L. Prylli and B. Tourancheau. Efficient Block Cyclic Data Redistribution. In *Euro-Par*, volume 1, pages 155–164. LNCS Springer Verlag, August 1996.

[179] C. Pérez, T. Priol, and A. Ribes. A Parallel CORBA Component Model for Numerical Code Coupling. *The International Journal of High Performance Computing Applications (IJHPCA)*, 17(4) :417–429, 2003.

[180] C. Pérez, T. Priol, and A. Ribes. PaCO++ : A Parallel Object Model for High-Performance Distributed Systems. In *Distributed Object and Component-based Software Systems Minitrack in the Software Technology Track of the 37th Hawaii International Conference on System Sciences (HICSS-37)*, Big Island, Hawaii, USA, January 2004. IEEE Computer Society Press.

[181] M. Ranganathan, A. Acharya, G. Edjlali, A. Sussman, and J. Saltz. Runtime Coupling of Data-Parallel Programs. In *ICS '96 : Proceedings of the 10th international conference on Supercomputing*, pages 229–236, New York, NY, USA, 1996. ACM Press.

[182] F. A. Rasio. Particle Methods in Astrophysical Fluid Dynamics. In *5th International Conference on Computational Physics (ICCP5)*, Kanazawa, Japan, November.

[183] S. Rathmayer and M. Lenke. A Tool for On-line Visualization and Interactive Steering of Parallel HPC Applications. In *Proceedings of the 11th IPPS'97*, pages 181–186, 1997.

[184] E. Reinhard, A.G. Chalmers, and F.W. Jansen. Overview of Parallel Photo-Realistic Graphics. In *Proceedings of Eurographics*, 1998.

[185] L. Renambot, H. E. Bal, D. Germans, and H. J. W. Spoelder. CAVEStudy : An Infrastructure for Computational Steering and Measuring in Virtual Reality Environments. *Cluster Computing*, 4(1) :79–87, 2001.

[186] L. Renambot, H. E. Bal, D. Germans, and H. J. W. Spoelder. CAVEStudy : An Infrastructure for Computational Steering and Measuring in Virtual Reality Environments. *Cluster Computing*, 4(1) :79–87, 2001.

[187] C. René and T. Priol. MPI Code Encapsulating Using Parallel CORBA Object. *Cluster Computing*, 3(4) :255–263, 2000.

[188] R. L. Ribler, J. S. Vetter, H. Simitci, and D. A. Reed. Autopilot : Adaptive Control of Distributed Applications. In *HPDC*, pages 172–179, 1998.

[189] D. Sevilla Ruiz. The CORBA & CORBA Component Model (CCM) Page. http ://ditec.um.es/ dsevilla/ccm.

[190] S. Plimpton et al. LAMMPS (Large-scale Atomic/Molecular Massively Parallel Simulator). http ://www.cs.sandia.gov/ sjplimp/lammps.html.

[191] W. Schroeder, K. Martin, and B. Lorensen. *The Visualisation ToolKit*. Kitware, 2002.

[192] K. Seymour, H. Nakada, S. Matsuoka, J. Dongarra, C. Lee, and H. Casanova. GridRPC : A Remote Procedure Call API for Grid Computing, june 2002.

[193] V. Springel. The cosmological simulation code GADGET-2. 2005.

[194] J. Squyres, A. Lumsdaine, W. George, J. Hagedorn, and J. Devaney. The Interoperable Message Passing Interface (IMPI) Extensions to LAM/MPI. MPI Developer's Conference, Ithica, NY, 2000.

[195] J. Stone, J. Gullingsrud, P. Grayson, K. Schulten, J. F. Hughes, and C. H. Séquin. A system for interactive molecular dynamics simulation (IMD). In *ACM Symposium on Interactive 3D Graphics (ACM SIGGRAPH)*, pages 191–194, 2001.

[196] J. E. Swan, M. Lanzagorta, D. Maxwell, E. Kuo, J. Uhlmann, W. Anderson, H.-J. Shyu, and W. Smith. A computational steering system for studying microwave interactions with missile bodies.

[197] C. Szyperski. *Component Software : Beyond Object-Oriented Programming*. ACM Press/Addison-Wesley Publishing Co., 1998.

[198] R. Thakur, A. Choudhary, and J. Ramanujam. Efficient Algorithms for Array Redistribution. *IEEE Transactions on Parallel and Distributed Systems*, 7(6) :587–594, 1996.

[199] Theoretical and Computational Biophysics Group. NAMD : Scalable Molecular Dynamics. http ://www.ks.uiuc.edu/Research/namd.

[200] C. Upson, T. Faulhaber, D. Kamis, D. Schlegel, D. Laidlaw, F. Vroom, R. Gurwitz, and A. vanDam. The Application Visualization System : A Computational Environment for Scientific Visualization. *IEEE Computer Graphics and Applications*, 9 :30–42, 1989.

[201] R. van Liere, J. D. Mulder, and J. J. van Wijk. Computational steering. *Future Generation Computer Systems*, 12(5) :441–450, 1997.

[202] R. van Liere and J. J. van Wijk. CSE : A Modular Architecture for Computational Steering. In M. Gobel, J. David, P. Slavik, and J. J. van Wijk, editors, *Virtual Environments and Scientific Visualization '96*, pages 257–266. Springer-Verlag Wien, 1996.

[203] J. van Wijk and R. van Liere. An Environment for Computational Steering. In Hans Hagen Gregory M. Nielson and Heinrich Mueller, editors, *Scientific Visualization : Overviews, Methodologies, and Technique*. IEEE Computer Society Press, 1997.

[204] J. J. van Wijk, R. van Liere, and J. D. Mulder. Bringing Computational Steering to the User. Centrum voor Wiskunde en Informatica (CWI), ISSN 1386-369X, 31 1997.

[205] J. Vetter. Experiences using computational steering on existing scientific applications. In *Ninth SIAM Conf. Parallel Processing*, 1999.

[206] J. Vetter and K. Schwan. Progress : A Toolkit for Interactive Program Steering. In *Proceedings of the 1995 International Conference on Parallel Processing*, pages 139–142, 1995.

[207] J. Vetter and K. Schwan. Models for Computational Steering. Technical Report GIT-CC-95-39, Georgia Institute of Technology, 1996.

[208] J. Vetter and K. Schwan. High Performance Computational Steering of Physical Simulations. In *Proceedings of the 11th IPPS'97*, pages 128–132, 1997.

[209] J. Vetter and K. Schwan. Techniques for High-Performance Computational Steering. In *IEEE Concurrency*, pages 63–74, 1999.

[210] W3C. Extensible Markup Language (XML) 1.0 (Third Edition). http ://www.w3.org/TR/REC-xml.

[211] W3C. Web Services. http ://www.w3.org/2002/ws.

[212] D. W. Walker and S. W. Otto. Redistribution of Block-Cyclic Data Distributions using MPI. *Concurrency : Practice and Experience*, 8(9) :707–728, 1996.

[213] T. Wilde, J. A. Kohl, and R. E. Flanery Jr. Immersive and 3D viewers for CUMULVS : VTK/CAVE and AVS/Express. *Future Generation Computer Systems (FGCS) Elsevier Science*, 19 :701–719, 2003.

[214] C. Wittenbrink. Survey of Parallel Volume Rendering Algorithms. In *Proceedings of Parallel and Distributed Processing Techniques and Applications*, pages 1329–1336, july 1998.

[215] A. Wollrath, R. Riggs, and J. Waldo. A Distributed Object Model for the Java System. In *2nd Conference on Object-Oriented Technologies & Systems (COOTS)*, pages 219–232. USENIX Association, 1996.

[216] O.C. Zienkiewicz and R.L. Taylor. *The finite element method*, volume 1. McGraw-Hill Bokk Company, fourth edition.

Liste des publications

[1] A. Esnard *Modèle pour la redistribution de données complexes*. RenPar'16, Croisic (Presqu'Île de Guérande), France, 2005, pages 25–36.

[2] A. Esnard, M. Dussere and O. Coulaud. *A Time-coherent Model for the Steering of Parallel Simulations*. Euro-Par 2004 Parallel Processing, Springer LNCS, pages 90–97, 2004.

[3] A. Esnard and M. Dussere. *Vers le pilotage des simulations numériques sur la grille*. RenPar'15, La colle-sur-loup, France, 2003, pages 196–203.

[4] O. Coulaud, M. Dussere and A. Esnard. *Toward a Computational Steering Environment based on CORBA*. Parallel Computing : Environments and Tools for Parallel Scientific Computing, Advances in Parallel Computing, Elsevier, Dresden, 2003, 13 :151–158.