Chapter 1

Partitioning and Load Balancing for Emerging Parallel Applications and Architectures

***Karen D. Devine*[†], *Erik G. Boman*[†], *and George Karypis*[‡]**

1.1 Introduction

An important component of parallel scientific computing is partitioning – the assignment of work to processors. This assignment occurs at the start of a computation ("static" partitioning). Often, reassignment also is done during a computation ("dynamic" partitioning) to redistribute work as the computation changes. The goal of partitioning is to assign work to processors in a way that minimizes total solution time. In general, this goal is pursued by equally distributing work to processors (i.e., "load balancing") while attempting to minimize interprocessor communication within the simulation. While distinctions can be made between "partitioning" and "load balancing," in this paper, we use the terms interchangeably.

A wealth of partitioning research exists for mesh-based partial differential equation (PDE) solvers (e.g., finite volume and finite element methods) and their sparse linear solvers. Here, graph-based partitioners have become the tools of choice, due to their excellent results for these applications and the availability of graph-partitioning software [43, 53, 55, 75, 82, 100]. Conceptually simpler geometric methods have proven to be highly effective for particle simulations, while providing reasonably good decompositions for mesh-based solvers. Software toolkits containing

[†]Discrete Algorithms and Math. Dept.; Sandia National Laboratories; P.O. Box 5800; Albuquerque, NM 87185-1111; {kddevin,egboman}@sandia.gov. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under Contract DE-AC04-94AL85000.

[‡]Computer Science and Engineering Dept.; 4-192 EE/CS Building; 200 Union Street S.E.; Minneapolis, MN 55455; karypis@cs.umn.edu.

several different algorithms enable developers to easily compare methods to determine their effectiveness in applications [24, 26, 60]. Prior efforts have focused primarily on partitioning for homogeneous computing systems, where computing power and communication costs are roughly uniform.

Wider acceptance of parallel computing has lead to an explosion of new parallel applications. Electronic circuit simulations, linear programming, materials modeling, crash simulations, and data mining are all adopting parallel computing to solve larger problems in less time. And the parallel architectures they use have evolved far from uniform arrays of multiprocessors. While homogeneous, dedicated parallel computers can offer the highest performance, their cost often is prohibitive. Instead, parallel computing is done on everything from networks of workstations to clusters of shared-memory processors to grid computers. These new applications and architectures have reached the limit of standard partitioners' effectiveness; they are driving development of new algorithms and software for partitioning.

This paper surveys current research in partitioning and dynamic load balancing, with special emphasis on work presented at the 2004 SIAM Conference on Parallel Processing for Scientific Computing. "Traditional" load-balancing methods are summarized in §1.2. In §1.3, we describe several non-traditional applications along with effective partitioning strategies for them. Some non-traditional approaches to load balancing are described in §1.4. In §1.5, we describe partitioning goals that reach beyond typical load-balancing objectives. And in §1.6, we address load-balancing issues for non-traditional architectures.

1.2 Traditional Approaches

The partitioning strategy that is, perhaps, most familiar to application developers is graph partitioning. In graph partitioning, an application's work is represented by a graph $G(V, E)$. The set of vertices V consists of objects (e.g., elements, nodes) to be assigned to processors. The set of edges E describes relationships between vertices in V; an edge e_{ij} exists in E if vertices i and j share information that would have to be communicated if i and j were assigned to different processors. Both vertices and edges may have weights reflecting their computational and communication cost, respectively. The goal, then, is to partition vertices so that each processor has roughly equal total vertex weight while minimizing the total weight of edges "cut" by subdomain boundaries. (Several alternatives to the edge-cut metric, e.g., reducing the number of boundary vertices, have been proposed [41, 42].)

Many graph-partitioning algorithms have been developed. Recursive Spectral Bisection [81, 91] splits vertices into groups based on eigenvectors of the Laplacian matrix associated with the graph. While effective, this strategy is slow due to the eigenvector computation. As an alternative, multilevel graph partitioners [13, 44, 55] reduce the graph to smaller, representative graphs that can be partitioned easily; the partitions are then projected to the original graph, with local refinements (usually based on the Kernighan-Lin method [56]) reducing imbalance and cut-edge weight at each level. Multilevel methods form the core of serial [43, 55, 75, 82, 100] and parallel [53, 100] graph-partitioning libraries. Diffu-

sive graph partitioners [22, 45, 58, 105] operate more locally than multilevel graph partitioners. Diffusive partitioners transfer work from heavily loaded processors to their more lightly loaded neighbors; "neighbors" are defined either by the network in the parallel computer or by a processor graph induced by the application's data dependencies. Diffusive methods are faster than multilevel methods, but can require several iterations to achieve global balance. Diffusive partitioners are also more "incremental" than other graph partitioners; that is, small changes in processor work loads result in only small changes in the decomposition. This incrementality is important in dynamic load balancing, where the cost to move data to a new decomposition must be kept low. Graph partitioners allowing multiple weights per vertex (i.e., multiconstraint or multiphase partitioning) [54, 87, 102] or edge (i.e., multiobjective partitioning) [85] have been applied to a variety of multiphase simulations.

Geometric partitioning methods can be effective alternatives to graph partitioners. Using only objects' weights and physical coordinates, they assign equal object weight to processors while grouping physically close objects within subdomains. While they tend to produce partitions with higher communication costs than graph partitioning, they run faster and, in most cases, are implicitly incremental. Moreover, applications that lack natural graph connectivity (e.g., particle methods) can easily use geometric partitioners.

Geometric recursive bisection uses a cutting plane to divide geometric space into two sets with equal object weight (Figure 1.1). The resulting subdomains are divided recursively in the same manner, until the number of subdomains equals the number of desired partitions. (This algorithm is easily extended from powers of two to arbitrary numbers of partitions.) Variants of geometric recursive bisection differ primarily in their choice of cutting plane. Recursive Coordinate Bisection (RCB) [6] chooses planes orthogonal to coordinate axes. Recursive Inertial Bisection (RIB) [91, 94] uses planes orthogonal to "long directions" in the geometry; these long directions are the principal axes of inertia. (Note that RIB is not incremental.) Unbalanced Recursive Bisection (URB) [48] generates subdomains with lower aspect ratio (and, by implication, lower communication costs) by dividing the geometry in half and then assigning a number of processors to each half that is proportional to the work in that half.

Another geometric method, space-filling curve (SFC) partitioning, uses SFCs to map objects from their position in three-dimensional space to a linear ordering. Objects are assigned a "key" (typically an integer or a real number) representing the point on an SFC that is closest to the object. Sorting the keys creates the linear ordering of the objects. This linear ordering is then cut into equally weighted pieces to be assigned to partitions (Figure 1.2). SFC partitioners can be implemented in a number of ways. Different curves (e.g., Hilbert, Morton) may be used. The sorting step can be replaced by binning strategies [28]. An explicit octree representation of the SFC can be built [34, 63]. Topological connectivity can be used instead of coordinates to generate the SFC [65, 66]. In each approach, however, the speed of the algorithm and quality of the resulting decomposition is comparable to RCB.

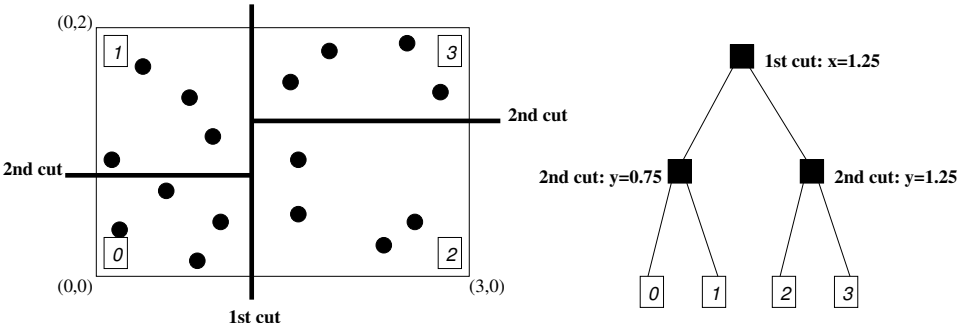


Figure 1.1. *Cutting planes (left) and associated cut tree (right) for geometric recursive bisection. Dots are objects to be balanced; cuts are shown with dark lines and tree nodes.*

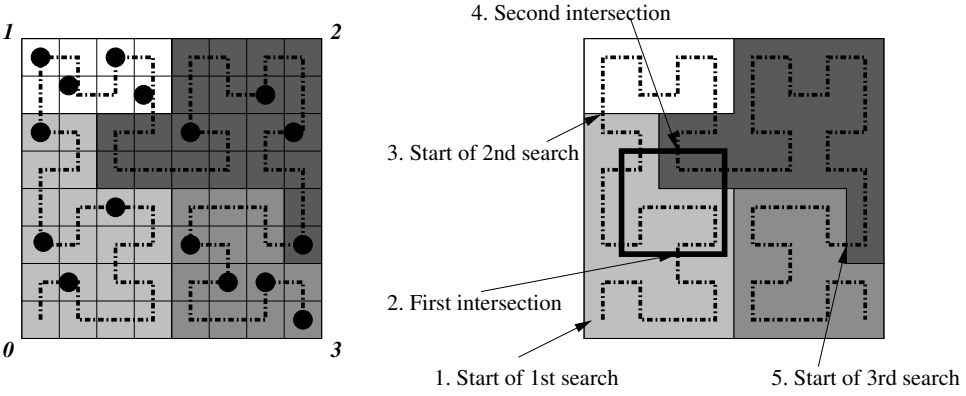


Figure 1.2. *SFC partitioning (left) and box-assignment search procedure (right). Objects (dots) are ordered along the SFC (dotted line). Partitions are indicated by shading. The box for box-assignment intersects partitions 0 and 2.*

1.3 Beyond traditional applications

Traditional partitioners have been applied with great success to a variety of applications. Multilevel graph partitioning is highly effective for finite element and finite volume methods (where mesh nodes or cells are divided among processors). Diffusive graph partitioners and incremental geometric methods are widely used in dynamic computations such as adaptive finite element methods [6, 25, 32, 74, 84]. The physical locality of objects provided by geometric partitioners has been exploited in particle methods [80, 104].

Some new parallel applications can use enhancements of traditional partitioners with success. Contact detection for crash and impact simulations can use geometric and/or graph-based partitioners, as long as additional functionality for

finding overlaps of the geometry with given regions of space is supported. Data-mining applications often use graph partitioners to identify clusters within data sets; the partitioners' objective functions are modified to obtain non-trivial clusters.

Other applications, however, require new partitioning models. These applications are characterized by higher data connectivity and less homogeneity and symmetry. For example, circuit and density functional theory simulations can have much less data locality than finite element methods. Graph-based models do not sufficiently represent the data relationships in these applications.

In this section, we describe some emerging parallel applications and appropriate partitioning solutions for them. We present techniques used for partitioning contact detection simulations. We survey application of graph-based partitioners to clustering algorithms for data mining. And we discuss hypergraph partitioning, an effective alternative to graph partitioning for less structured applications.

1.3.1 Partitioning for Parallel Contact/Impact Computations

A large class of scientific simulations, especially those performed in the context of computational structural mechanics, involve meshes that come in contact with each other. Examples include simulations of vehicle crashes, deformations, and projectile-target penetration. In these simulations, each iteration consists of two phases. During the first phase, traditional finite difference/element/volume methods compute forces on elements throughout the problem domain. In the second phase, a search determines which surface elements have come in contact with and/or penetrated other elements; the positions of the affected elements are corrected, elements are deformed, and the simulation progresses to the next iteration.

The actual contact detection is usually performed in two steps. The first step, *global search*, identifies pairs of surface elements that are close enough to potentially be in contact with each other. In the second step, *local search*, the exact locations of the contacts (if any) between these candidate surfaces are computed.

In global search, surface elements are usually represented by bounding boxes; two surface elements intersect only if their bounding boxes intersect. In parallel global search, surface elements first must be sent to processors owning elements with which they have potential interactions. Thus, computing the set of processors whose subdomains intersect a bounding box (sometimes called "box assignment") is a key operation in parallel contact detection.

Plimpton *et al.* developed a parallel contact detection algorithm that uses different decompositions for the computation of element forces (phase one) and the contact search (phase two) [80]. For phase one, they apply a traditional multilevel graph partitioner to all elements of the mesh. Recursive coordinate bisection (RCB) is used in phase two to evenly distribute only the surface elements. Between phases, data is mapped between the two decompositions, requiring communication; however, using two decompositions ensures that the overall computation is balanced and each phase is as efficient as possible. Because RCB uses geometric coordinates, potentially intersecting surfaces are likely to be assigned to the same processor, reducing communication during global search. Moreover, the box-assignment operation is very fast and efficient. The RCB decomposition is described fully by the

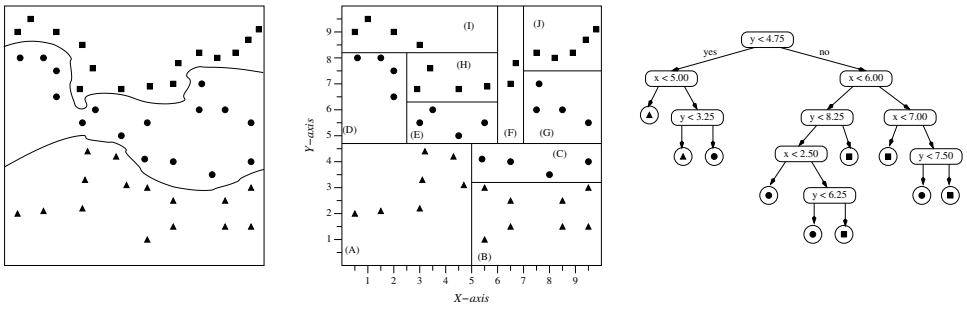
tree of cutting planes used for partitioning (Figure 1.1). The planes are stored on each processor, and the tree of cuts is traversed to determine intersections of the bounding boxes with the processor subdomains.

The use of a geometric method for the surface-element decomposition has been extended to space-filling curve (SFC) partitioners, due in part to their slightly faster decomposition times. Like RCB, SFC decompositions can be completely described by the cuts used to partition the linear ordering of objects. Box-assignment for SFC decompositions, however, is more difficult than for RCB, since SFC partitions are not regular rectangular regions. To overcome this difficulty, Heaphy *et al.* [28, 40] developed an algorithm based on techniques for database query [57, 67]. A search routine finds each point along the SFC at which the SFC enters the bounding box (Figure 1.2); binary searches through the cuts map each entry point to the processor owning the portion of the SFC containing the point.

Multiconstraint partitioning can be used in contact detection. Each element is assigned two weights — one for force calculations (phase one) and a second for contact computations (phase two). A single decomposition that balances both weights is computed. This approach balances computation in both phases, while eliminating the communication between phases that is needed in the two-decomposition approach. However, solving the multiconstraint problem introduces new challenges.

Multiconstraint or multiphase graph partitioners [54, 102] can be applied naturally to obtain a single decomposition that is balanced with respect to both the force and contact phases. These partitioners attempt to minimize interprocessor communication costs subject to the constraint that each component of the load is balanced. Difficulty arises, however, in the box-assignment operation, as the subdomains generated by graph partitioners do not have geometric regularity that can be exploited. One could represent processor subdomains by bounding boxes and compute intersections of the surface-element bounding box with the processor bounding boxes. However, because the processor bounding boxes are likely to overlap, many "false positives" can be generated by box assignment; that is, a particular surface element is said to intersect with a processor, even though none of the processor's locally stored elements identify it as a candidate for local search. To address this problem, Karypis [50] constructs a detailed geometric map of the volume covered by elements assigned to each subdomain (Figure 1.3). He also modifies the multiconstraint graph decomposition so that each subdomain can be described by a small number of disjoint axis-aligned boxes; this improved geometric description reduces the number of false-positives. The boxes are assembled into a binary tree describing the entire geometry. Box assignment is then done by traversing the tree, as in RCB; however, the depth of the tree can be much greater than RCB's tree.

Boman *et al.* proposed a multicriteria geometric partitioning method that may be used for contact problems [12, 28]. Like the multiconstraint graph partitioners, this method computes one decomposition that is balanced with respect to multiple phases. Their algorithm, however, uses RCB, allowing box assignment to be done easily by traversing the tree of cuts (Figure 1.1). Instead of solving a multiconstraint problem, they solve a multiobjective problem: find as good a balance as possible with respect to all loads. While good multicriteria RCB decompositions do not always exist, heuristics are used to generate reasonable decompositions for many



(a) Partitioning of the contact points. (b) Partitioning of the space. (c) Associated Decision Tree

Figure 1.3. *Use of multi-constraint graph partitioning for contact problems: (a) the 45 contact points are divided into three partitions; (b) the subdomains are represented geometrically as sets of axis-aligned rectangles; and (c) a decision tree describing the geometric representation is used for contact search.*

problems. In particular, they pursue the simpler objective

$$\min_s \max(g(\sum_{i \leq s} a_i), g(\sum_{i > s} a_i)),$$

where a_i is the weight vector for object i, and g is a monotonically non-decreasing function in each component of the input vector; typically $g(x) = \sum_j x_j^p$ with $p = 1$ or $p = 2$, or $g(x) = \|x\|$ for some norm. This objective function is unimodal with respect to s; that is, starting with $s = 1$ and increasing s, the objective decreases, until at some point the objective starts increasing. That point defines the optimal bisection value s, and it can be computed efficiently.

1.3.2 Clustering in Data Mining

Advances in information technology have greatly increased the amount of data generated, collected, and stored in various disciplines. The need to effectively and efficiently analyze these data repositories to transform raw data into information and, ultimately, knowledge motivated the rapid development of data mining. Data mining combines data analysis techniques from a wide spectrum of disciplines. Among the most extensively used data mining techniques is *clustering*, which tries to organize a large collection of data points into a relatively small number of meaningful, coherent groups. Clustering has been studied extensively; two recent surveys [39, 47] offer comprehensive summaries of different applications and algorithms.

One class of clustering algorithms is directly related to graph partitioning; these algorithms model datasets with graphs and discover clusters by identifying well-connected subgraphs. Two major categories of graph models exist: *similarity-based* models [31] and *object-attributed-based* models [29, 109]. In a similarity-based graph, vertices represent data objects, and edges connect objects that are similar to each other. Edge weights are proportional to the amount of similarity between objects. Variations of this model include reducing the density of the graph by

focusing on only a small number of nearest neighbors of each vertex, and using hypergraphs to allow set-wise similarity as opposed to pair-wise similarity. Object-attribute models represent how objects are related to the overall set of attributes. Relationships between objects and attributes are modeled by a bipartite graph $G(V_o, V_a, E)$, where V_o is the set of vertices representing objects, V_a is the set of vertices representing attributes, and E is the set of edges connecting objects in V_o with their attributes in V_a. This model is applicable when the number of attributes is very large, but each object has only a small subset of them.

Graph-based clustering approaches can be classified into two categories: *direct* and *partitioning-based*. Direct approaches identify well-connected subgraphs by looking for connected components within the graph. Different definitions of the properties of connected components can be used. Some of the most widely used methods seek connected components that correspond to cliques and employ either exact or heuristic clique partitioning algorithms [23, 103]. However, this clique-based formulation is overly restrictive and cannot find large clusters in sparse graph models. For this reason, much research has focused on finding components that contain vertices connected by multiple intra-cluster disjoint paths [5, 36, 38, 62, 88, 89, 93, 98, 108]. A drawback of these approaches is that they are computationally expensive, and, as such, can be applied only to relatively small datasets.

Partitioning-based clustering methods use min-cut graph-partitioning algorithms to decompose the graphs into well-connected components [30, 51, 109]. By minimizing the total weight of graph edges cut by partition boundaries, they minimize the similarity between clusters, and, thus, tend to maximize the intra-cluster similarity. Using spectral and multilevel graph partitioners, high quality decompositions can be computed reasonably quickly, allowing these methods to scale to very large datasets. However, the traditional min-cut formulation can admit trivial solutions in which some (if not most) of the partitions contain a very small number of vertices. For this reason, most of the recent research has focused on extending the min-cut objective function so that it accounts for the size of the resulting partitions and, thus, produces solutions that are better balanced. Examples of effective objective functions are ratio cut (which scales the weight of cut edges by the number of vertices in each partition) [37], normalized cut (which scales the weight of cut edges by the number of edges in each partition) [90], and min-max cut (which scales the weight of cut edges by the weight of uncut edges in each partition) [30].

1.3.3 Partitioning for Circuits, Nanotechnology, Linear Programming and more

While graph partitioners have served well in mesh-based PDE simulations, new simulation areas such as electrical systems, computational biology, linear programming and nanotechnology show their limitations. Critical differences between these areas and mesh-based PDE simulations include high connectivity, heterogeneity in topology, and matrices that are structurally non-symmetric or rectangular. A comparison of a finite element matrix with matrices from circuit and density functional theory (DFT) simulations is shown in Figure 1.4; circuit and DFT matrices are more dense and less structured than finite element matrices. The structure of

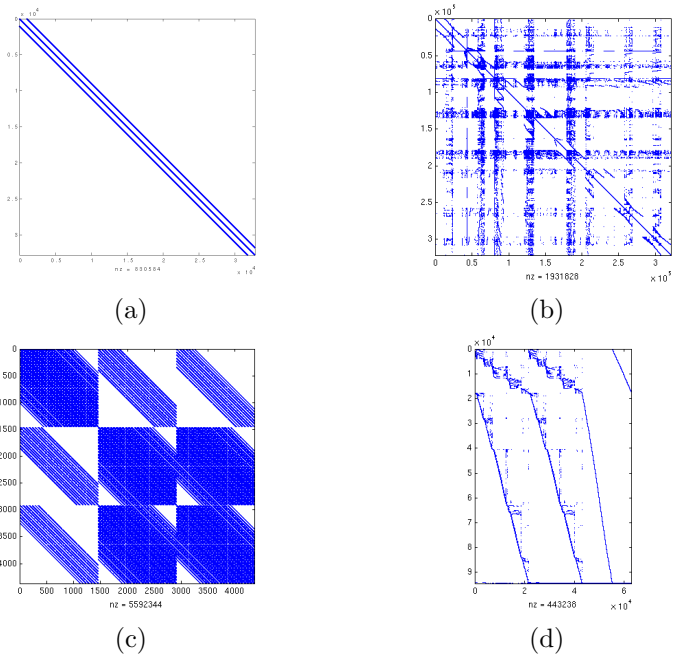


Figure 1.4. *Comparing the non-zero structure of matrices from (a) a hexahedral finite element simulation, (b) a circuit simulation, (c) a density functional theory simulation, and (d) linear programming shows differences in structure between traditional and emerging applications.*

linear programming matrices differs even more; indeed, these matrices are usually not square. In order to achieve good load balance and low communication in such applications, accurate models of work and dependency/communication are crucial.

Graph models are often considered the most effective models for mesh-based PDE simulations. However, the edge-cut metric they use only approximates communication volume. For example, in Figure 1.5 (left), a grid is divided into two partitions (separated by a dashed line). Grid point A has four edges associated with it; each edge (drawn as an ellipse) connects A with a neighboring grid point. Two edges are cut by the partition boundary; however, the actual communication volume associated with sending A to the neighboring processor is only one grid point. Nonetheless, countless examples demonstrate graph partitioning's success in mesh-based PDE applications where this approximation is often good enough.

Another limitation of the graph model is the type of systems it can represent [41]. Because edges in the graph model are non-directional, they imply symmetry in all relationships, making them appropriate only for problems represented by square, structurally symmetric matrices. Structurally non-symmetric systems A must be represented by a symmetrized model, typically $A + A^T$ or $A^T A$, adding new edges to the graph and further skewing the communication metric. While a

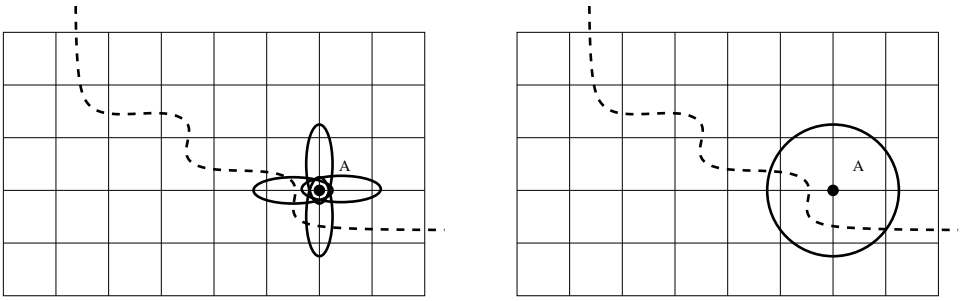


Figure 1.5. *Example of communication metrics in the graph (left) and hypergraph (right) models. Edges are shown with ellipses; the partition boundary is the dashed line.*

directed graph model could be adopted, it would not improve the accuracy of the communication metric.

Likewise, graph models cannot represent rectangular matrices, such as those arising in linear programming. Kolda and Hendrickson [42] propose using bipartite graphs. For an $m \times n$ matrix A, vertices $r_i, i = 1, \ldots, m$ represent rows, and vertices $c_j, j = 1, \ldots, n$ represent columns. Edges e_{ij} connecting r_i and c_j exist for non-zero matrix entries a_{ij}. But as in other graph models, the number of cut edges only approximates communication volume.

Hypergraph models address many of the drawbacks of graph models. As in graph models, hypergraph vertices represent the work of a simulation. However, hypergraph edges (hyperedges) are sets of two *or more* related vertices. A hyperedge can thus represent dependencies between any set of vertices. The number of hyperedge cuts accurately represents communication volume [16, 18]. In the example in Figure 1.5 (right), a single hyperedge (drawn as a circle) including vertex A and its neighbors is associated with A; this single cut hyperedge accurately reflects the communication volume associated with A.

Hypergraphs also serve as useful models for sparse matrix computations, as they accurately represent nonsymmetric and rectangular matrices. For example, the columns of a rectangular matrix could be represented by the vertices of a hypergraph. Each matrix row would be represented by a hyperedge connecting all vertices (columns) with non-zero entries in that row. A hypergraph partitioner, then, would assign columns to processors while attempting to minimize communication along rows. One could alternatively let vertices represent rows and edges represent columns to obtain a row-partitioning.

Optimal hypergraph partitioning, like graph partitioning, is NP-hard, but good heuristic algorithms have been developed. The dominant algorithms are extensions of the multilevel algorithms for graph partitioning. Hypergraph partitioning's effectiveness has been demonstrated in many areas, including VLSI layout [14], sparse matrix decompositions [18, 99], and database storage and data mining [21, 73]. Several (serial) hypergraph partitioners are available (e.g., hMETIS [52], Pa-

ToH [18, 17], MLPart [15], Mondriaan [99]), and two parallel hypergraph partitioners for large-scale problems are under development: Par*k*way [97], which targets information retrieval and Markov models, and Zoltan-PHG [27], part of the Zoltan [11] toolkit for parallel load balancing and data management in scientific computing.

1.4 Beyond traditional approaches

While much partitioning research has focused on the needs of new applications, older, important applications have not been forgotten. Sparse matrix-vector multiplication, for example, is a key component of countless numerical algorithms; improvements in partitioning strategies for this operation can greatly impact scientific computing. Similarly, because of the broad use of graph partitioners, algorithms that compute better graph decompositions can influence a range of applications. In this section, we discuss a few new approaches to these traditional problems.

1.4.1 Partitioning for sparse matrix-vector multiplication

A common kernel in many numerical algorithms is multiplication of a sparse matrix by a vector. For example, this operation is the most computationally expensive part of iterative methods for linear systems and eigensystems. More generally, many data dependencies in scientific computation can be modeled as hypergraphs, which again can be represented as (usually sparse) matrices (see §1.3.3). The question is how to distribute the nonzero matrix entries (and the vector elements) in a way that minimizes communication cost while maintaining load balance. The sparse case is much more complicated than the dense case, and is a rich source of combinatorial problems. This problem has been studied in detail in [17, 18] and in [10, Ch.4].

The standard algorithm for computing $u = Av$ on a parallel computer has four steps. First, we communicate entries of v to processors that need them. Second, we compute local contributions of the type $\sum_j a_{ij}v_j$ for certain i, j and store them in u. Third, we communicate entries of u. Fourth, we add up partial sums in u.

The simplest matrix distribution is a one-dimensional (1D) decomposition of either matrix rows or columns. The communication needed for matrix-vector multiplication with 1D distributions is demonstrated in Figure 1.6. Çatalyürek and Aykanat [17, 18] realized that this problem can be modeled as a hypergraph partitioning problem, where, for a row distribution, matrix rows correspond to vertices and matrix columns correspond to hyperedges, and vice versa for a column distribution. The communication volume is then exactly proportional to the number of cut hyperedges in the bisection case; if there are more than two partitions, the number of partitions covering each hyperedge has to be taken into account. The 1D hypergraph model reduced communication volume by 30–40% on average versus the graph model for a set of sparse matrices [17, 18].

Two-dimensional (2D) data distributions (i.e., block distributions) are often better than 1D distributions. Most 2D distributions used are Cartesian; that is, the matrix is partitioned both along rows and columns in a grid-like fashion and each processor is assigned the nonzeros within a rectangular block. The Cartesian 2D

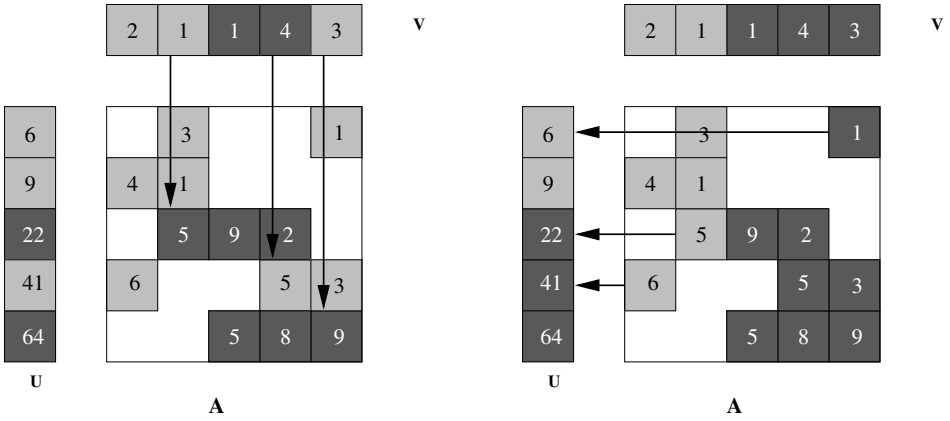


Figure 1.6. *Row (left) and column (right) distribution of a sparse matrix for multiplication $u = Av$. There are only two processors, indicated by dark and light shading, and communication between them is shown with arrows. In this example, the communication volume is three words in both cases. (Adapted from [10, Ch.4].)*

distribution is inflexible and good load balance is often difficult to achieve, so variations like *jagged* or *semi-general block* partitioning have been proposed [61, 77, 83]. These schemes first partition a matrix into p_1 strips in one direction, and then partition each strip independently in the orthogonal direction into p_2 domains, where $p_1 \times p_2$ is the total number of desired partitions. Vastenhow and Bisseling have recently suggested a non-Cartesian distribution called *Mondriaan* [99]. The method is based on recursive bisection of the matrix into rectangular blocks, but permutations are allowed and the cut directions may vary. Each bisection step is solved using hypergraph partitioning. Mondriaan distributions often have significantly lower communication costs than 1D or 2D Cartesian distributions [99].

In the most general distribution, each nonzero (i, j) is assigned to a processor with no constraints on the shape or connectivity of a partition. (See Figure 1.7 for an example.) Çatalyürek and Aykanat [16, 19] showed that computing such general (or fine-grain) distributions with low communication cost can also be modeled as a hypergraph partitioning problem, but using a different (larger) hypergraph. In their fine-grain model, each nonzero entry corresponds to a vertex and each row or column corresponds to a hyperedge. This model accurately reflects communication volume. Empirical results indicate that partitioning based on the fine-grain model has communication volume that is lower than 2D Cartesian distributions [19]. The disadvantage of using such complex data distributions is that the application needs to support arbitrary distributions, which is typically not the case.

After a good distribution of the sparse matrix A has been found, vectors u and v still must be distributed. In the square case, it is often convenient to use the same distribution, but it is not necessary. In the rectangular case, the vector distributions will obviously differ. Bisseling and Meesen [10, 9] have studied this

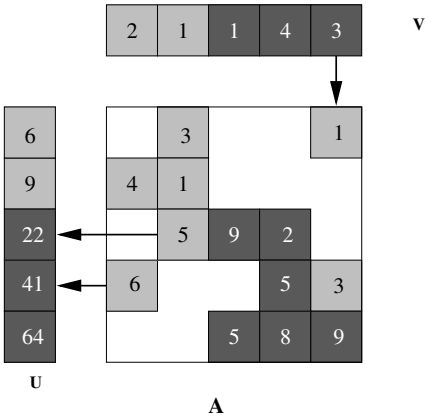


Figure 1.7. *Irregular matrix distribution with two processors. Communication between the two processors (shaded dark and light) is indicated with arrows.*

vector partitioning problem, and suggest that the objective for this phase should be to balance the communication between processors. Note that a good matrix (hypergraph) partitioning already ensures that the total communication volume is small. For computing $u = Av$, no extra communication is incurred as long as v_j is assigned to a processor that also owns an entry in column j of A, and u_i is assigned to a processor that contains a nonzero in row i. There are many such assignments; for example, in Figure 1.7, u_3, u_4, and v_5 can all be assigned to either processor. The vector partitions resulting from different choices for these particular vector entries are all equally good measured by total communication volume. One therefore has flexibility (see also Section 1.5.2 on flexibly assignable work) to choose a vector partitioning that minimizes a secondary objective, such as the largest number of send and receive operations on any processor. (Similar objectives are used in some parallel cost models, like the BSP model [10].) Bisseling and Meesen [10, 9] have proposed a fast heuristic for this problem, a greedy algorithm based on local bounds for the maximum communication for each processor. It is optimal in the special case where each matrix column is shared among at most two processors. Their approach does not attempt to load balance the entries in u and v between processors because doing so is not important for matrix-vector multiplication.

1.4.2 Semidefinite programming for graph partitioning

Although multilevel algorithms have proven quite efficient for graph partitioning, there is ongoing research into algorithms that may give higher quality solutions (but may also take more computing time). One such algorithm uses semidefinite programming (SDP).

The graph partitioning problem can be cast as an integer programming problem. Consider the bisection problem where the vertices of a graph $G = (V, E)$ shall

be partitioned into two approximately equal sets P_0 and P_1. Let $x \in \{-1, 1\}^n$ be an assignment vector such that $x_i = -1$ if vertex $v_i \in P_0$ and $x_i = 1$ if it is in P_1. It is easy to see that the number of edges crossing from P_0 to P_1 is

$$\frac{1}{4} \sum_{(i,j)\in E, i<j} (x_i - x_j)^2 = \frac{1}{4} x^T L x, \tag{1.1}$$

where L is the Laplacian matrix of the graph G. Minimizing (1.1) while maintaining load balance is an NP-hard problem. By allowing x to be a vector of any real numbers, the following relaxation can be solved efficiently:

$$\min_x \frac{1}{4} x^T L x \tag{1.2}$$

$$\text{subject to } x^T x = n, \tag{1.3}$$

$$x^T e = 0. \tag{1.4}$$

The solution to (1.2) is the eigenvector of L with the second smallest eigenvalue of L. (The smallest eigenvalue of L is zero with the eigenvector e.) The *spectral* partitioning algorithm computes the second eigenvector of L and uses a simple rounding scheme to obtain a partitioning from x [81].

Over the last decade, much work has been done on semidefinite programming to approximate NP-hard combinatorial problems. SDP can be viewed as a generalization of linear programming where the unknown variable is a symmetric semidefinite matrix. SDP can be solved in polynomial time, but current algorithms require $\Theta(n^3)$ computations per major iteration and $\Theta(n^2)$ memory, and, thus, are quite expensive. One SDP relaxation of the graph partitioning problem is

$$\min_X \frac{1}{4} L \bullet X \tag{1.5}$$

$$\text{subject to } diag(X) = e, \tag{1.6}$$

$$X \bullet (ee^T) = 0, \tag{1.7}$$

$$X \succcurlyeq 0 \quad (X \text{ is semidefinite}), \tag{1.8}$$

where X is a matrix and $\bullet$ denotes the element-wise matrix inner product. The matrix X can be considered a generalization of the vector x in (1.2); each node in the graph is assigned a vector instead of a real number. We remark that the SDP (1.5) becomes equivalent to the spectral problem (1.2) if we impose the additional constraint that rank(X)=1; then $X = xx^T$ for x in (1.2). A decomposition is derived from the matrix X. Since the SDP is a tighter approximation to the discrete problem, SDP solutions should produce higher quality partitions than the spectral algorithm. (Although the continuous SDP solution X will produce a good objective value, the discrete partitioning induced by X does not necessarily have a small cut size due to the rounding step, but randomized techniques can be employed.) The SDP method for graph partitioning has been studied, e.g., in [49, 106]. Oliveira [70] showed that a generalization of graph partitioning, where the vertices have preference values for belonging to a certain partition, can also be modeled as

a SDP. Since algorithms and software for solving SDP are quite slow, faster approximate solvers are needed. A promising approach is subspace iteration, in which X is restricted to lower-dimensional subspaces. Subspace algorithms for SDP are analogous to Lanczos/Arnoldi or Davidson-type algorithms for eigenvalue problems. Recent work [70, 71, 72] indicates that such algorithms for SDP graph partitioning are much faster than full SDP solvers and are competitive with spectral partitioning.

1.5 Beyond traditional models

While the traditional partitioning model (balancing workloads while minimizing interprocessor communication) often minimizes an application's total run time, there are applications for which different models or goals are more important. In applications where the total amount of work in a computation depends on the decomposition, traditional partitioning strategies are insufficient, as they partition with respect to a fixed amount of work; these scenarios, are called "complex objectives." Another load-balancing issue, recently described by Pınar and Hendrickson [79], is "flexibly assignable" work; in some cases, even after the data have been distributed, there is some flexibility to assign work among processors. And for some applications, the cost to move data from an old decomposition to a new one is so high that incorporating migration cost into a dynamic load-balancing strategy yields great benefit. These alternative models and goals for partitioning are described below.

1.5.1 Complex objectives

Traditional models assume that the total work (for a processor) is a linear sum of work loads associated with each task assigned to a processor. Therefore, each task is typically assigned a scalar number (weight) that describes the work (time) associated with that task. In the multiconstraint case, each task may have multiple weights associated with it.

However, there are applications where the total work is *not* a linear sum of the weights associated with the unit tasks. A good example is sparse matrix factorization (LU or Cholesky). Assuming each "task" is a nonzero in the matrix A, we can balance the number of nonzeros in A between processors. Unfortunately, during LU factorization, different processors get different amounts of *fill* in their submatrix. The fill depends on the structure of the sparse matrix and is hard to estimate in advance. Usually, there is no simple relation between the fill in L and U and the number of nonzeros in A.

Pınar and Hendrickson [78] treat such cases as *complex* objectives and use an iterative strategy for load balancing. They start with a simple linear model, and balance with respect to these weights using a traditional partitioner. After the first distribution, they evaluate (or estimate) the true complex objective and iterate using an incremental load-balancing algorithm like diffusion; the balance with respect to the complex objective improves in each iteration.

Many other applications can have "complex objectives." For example, each processor may perform an iterative method where the number of iterations varies and depends on local data. Incomplete factorizations also fall into the category of

complex objectives, except in the level-0 case where the nonzero pattern of $L + U$ is the same as the pattern of A and, thus, can be balanced in the traditional way.

Another load-balancing problem that fits this category is the problem of balancing computation and communication. Models like graph and hypergraph partitioning attempt to balance only computation while minimizing total communication volume. An alternative is to balance both computation and communication simultaneously. Another variation is to balance the weighted sum of computation and communication for each processor, where the weight scaling is machine dependent. This last model may be most realistic since, in practice, the slowest processor (after doing both computation and communication) determines the overall speed. The main difficulty in balancing computation and communication is that the communication requirements are not known until a data distribution has been computed. The iterative approach described above can be applied naturally to this problem [76].

1.5.2 Flexibly assignable work

In the data-parallel programming model, the data distribution implicitly also distributes the computation among processors. However, the data distribution does not always uniquely assign all the computations to processors. In many situations, some work can be performed by any of a set of processors. For example, some portions on the data may be replicated on multiple processors, any of which can perform the work. Alternatively, tasks may involve multiple data items that reside on different processors. These are both examples of what is called *flexibly assignable work* [76, 79]. A simple example is molecular dynamics simulations, where a force computation is required between particles that are close to each other. Typically, geometric space is partitioned into regions and assigned to processors. Processors compute forces between particles in the local region, but when two nearby particles reside on different processors, either processor could perform the computation. This computation is flexibly assignable work. Another example is finite element simulations where some of the computation is node-based while other parts are element-based. A decomposition assigns either nodes or elements to processors; there is then some flexibility in assigning the other entity.

A *task assignment* problem for flexibly assignable work was formally defined in [79]. Their goal is to minimize the number of tasks assigned to the maximally loaded processor. The authors propose two different solution methods. The first approach is based on network flow. Formulating the task assignment problem as a network model in which tasks "flow" from task nodes to processor nodes, they minimize the maximum total flow into any processor node. The solution strategy uses parametric search with probes, where each probe operation attempts to find a complete flow (i.e., a flow assigning every task to some processor) for which the maximum total flow into any processor is less than the search parameter. Each probe involves solving or updating a network flow problem (e.g., using augmenting paths). Since network flow algorithms are difficult to parallelize, this method is suitable when the task assignment may be solved in serial as a preprocessing step.

The second approach is applicable when the task assignment has to be done in parallel. A continuous relaxation of the discrete problem can be formulated as

an optimization problem of the form $\min \|Ax\|_\infty$ s.t. $Bx = d$, where Ax represents flow into processor nodes, Bx represents flow out of task nodes, and d is the vector of task sizes. ($Bx = d$ then enforces the requirement that every task is assigned to some processor.) This formulation can be recast as a standard linear program (LP), which is also difficult to solve in parallel. Instead, it is shown [79] that minimizing the 2-norm is equivalent to minimizing the ∞-norm for the problem in question; thus one has to solve only a constrained least-square problem of the form $\min \|Ax\|_2$ s.t. $Bx = d$. Such problems can be solved efficiently in parallel using iterative methods like Gauss-Seidel. We remark that the 2-norm minimization approach is similar to some diffusive load-balancing schemes [22].

1.5.3 Migration Minimization

The cost of moving data from an old decomposition to a new one is often higher than the cost of computing the new decomposition. Not only must data be communicated from the old to the new decomposition, but data structures must be rebuilt on both the sending and receiving processors to account for data removed or added, respectively. Many applications are interested in reducing this data migration cost, through either clever application development or partitioning algorithms.

Within an application, clever techniques can be used to reduce data migration costs. In adaptive finite element methods, for example, the amount of migrated data can be reduced by balancing coarse meshes rather than fully refined meshes. One technique, called predictive load balancing [35, 69], performs load balancing after error indicators that guide refinement are computed but before actual refinement occurs. Using the error indicators as weights approximating workloads after refinement, the coarse elements are partitioned and migrated; then the mesh is refined. Because data from the smaller mesh are moved, data migration costs are lower than if the refined mesh were partitioned. In addition, the refinement phase can be better balanced, and memory overflows can be avoided in cases where there is sufficient global memory for refinement but insufficient local memory before balancing.

Appropriately selecting load-balancing methods can also reduce migration costs. Incremental partitioners (e.g., RCB, SFC, diffusive graph partitioning) are preferred when data migration costs must be controlled. The unified partitioning strategy in ParMETIS computes both a multilevel graph decomposition ("scratch-remap") and a diffusive decomposition [86]; it then selects the better decomposition in terms of load balance and migration costs.

Still greater reduction of migration costs can be achieved by explicitly controlling them within load-balancing algorithms. For example, the similarity matrix in PLUM [68] represents a maximal matching between an old decomposition and a new one. Old and new partitions are represented by the nodes of a bipartite graph, with edges between old and new partitions representing the amount of data they share. A maximal matching, then, numbers the new partitions to provide the greatest overlap between old and new decompositions and, thus, the least data movement. Similar strategies have been adopted by ParMETIS [53] and Zoltan [24, 26].

Load-balancing objectives can also be adjusted to reduce data migration. Heuristics for selecting objects to move to new processors can select those with

the lowest migration costs. They can also select a few heavily weighted objects to satisfy balance criteria rather than many lightly weighted objects. Hu and Blake compute diffusive graph-based decompositions to achieve load balance subject to a minimization of data movement [46]. Berzins extends their idea by allowing greater load imbalance when data movement costs are high [7, 8]; he minimizes a metric combining load imbalance and data migration to reduce actual time-to-solution (rather than load imbalance) on homogeneous and heterogeneous networks.

1.6 Beyond traditional architectures

Traditional parallel computers consist of up to thousands of processors, sharing a single architecture and used in a dedicated way, connected by a fast network. Most often these computers are custom-built and cost millions of dollars, making it difficult for any organizations except large research centers to own and maintain them. As a result, simpler, more cost-effective parallel environments have been pursued. In particular, clusters have become viable small-scale alternatives to traditional parallel computers. Because they are smaller and are built from commodity parts, they are relatively easy to acquire and run. Their computational capabilities can also be increased easily through the addition of new nodes, although such additions create heterogeneous systems when the new nodes are faster than the old ones. On the other extreme, grid computing ties together widely distributed, widely varying resources for use by applications. Accounting for the heterogeneity of both the processors and the networks connecting them is important in partitioning for grid applications. Moreover, both clusters and grids are shared environments in which resource availability can fluctuate during a computation.

We describe two load-balancing strategies for such environments. The first, resource-aware partitioning, uses dynamic information about the computing environment as input to standard partitioning algorithms. The second, loop scheduling, creates a very fine-grained decomposition of a computation, distributing work on an as-needed basis to processors.

1.6.1 Resource-Aware Partitioning

In resource-aware partitioning, information about a computing environment is combined with traditional partitioning algorithms (or variations on traditional algorithms) to dynamically adjust processor workloads in the presence of non-homogeneous and/or changing computing resources. One approach collects information about the computing environment and processes it for use in partitioning algorithms designed for homogeneous systems. Static information about the computing environment (e.g., CPU MHz ratings, network bandwidth, memory capacity per node) can be provided in a file. Dynamic information must be obtained through monitoring of CPU, network and memory usage. The Network Weather Service (NWS) [107], for example, monitors network and CPU resources and uses mathematical models to forecast resource availability over a given time. The Remos system [59] provides similar network monitoring without sending messages to determine network capacity. Sinha and Parashar [92] use NWS to gather data about the state and

capabilities of available resources. They then compute the load capacity of each node as a weighted sum of processing, memory, and communications capabilities, and use standard partitioners to generate partitions with sizes proportional to their load capacity. Similarly, the Dynamic Resource Utilization Model (DRUM) of Faik et al. [28, 33] uses threads to non-intrusively monitor the computing environment; available computational and communication "powers" are computed and used as percentages of work to be assigned to processors by any standard partitioner. Minyard and Kallinderis [64] monitor process "wait times," measuring the time CPUs are idle while waiting for other processors to finish; they use the wait times as weights in an octree partitioning scheme.

A second approach to resource-aware partitioning involves direct incorporation of information about the computing environment into partitioning algorithms. For example, Walshaw and Cross [101] incorporate processor and network information directly into their multilevel graph partitioners. They accept network information through a Network Cost Matrix (NCM), a complete graph with edges representing processor interconnections and edge weights represent the path length between processors. The NCM is incorporated into the cost function used in their multilevel graph partitioners. Teresco et al. [96, 95] use information from DRUM to compute decompositions hierarchically. DRUM models a computing environment as a tree, where the root represents the entire system, children represent high-level subdivisions of the system (e.g., routers, switches), and leaves represent computation nodes (e.g., single processors or shared-memory processors). In hierarchical partitioning, work is divided among children at a given level of the DRUM tree, with percentages of work determined by the powers of the subtrees' resources. The work assigned to each subtree is then recursively partitioned among the nodes in the subtrees. Different partitioning methods can be used in each level and subtree to produce effective partitions with respect to the network; for example, graph or hypergraph partitioners could minimize communication between nodes connected by slow networks while fast geometric partitioners operate within each node.

1.6.2 Load-balancing via Dynamic Loop Scheduling

Load imbalances in scientific applications are induced not only by an application's algorithms or an architecture's hardware, but also by system effects, such as data access latency and operating system interference. The potential for these imbalances to become predominant increases in non-traditional environments such as networks of workstations (NOW), clusters of NOW, and clusters of shared-memory processors. The previously discussed load-balancing approaches rely on application and system characteristics that change predictably during computations. For example, adaptive finite element computations can be effectively load-balanced by existing repartitioning algorithms that account for changes in the mesh and/or architecture that occurred in previous iteration(s). However, these approaches lead to sub-optimal results when the number of data points, the workload per data point, and the underlying computational capabilities cannot be predicted well from an *a priori* evaluation of the computation and architecture.

To address this problem, dynamic work-scheduling schemes can be used to

maintain balanced loads by assigning work to idle processors at run time. By delaying assignments until processors are idle, these schemes accommodate systemic as well as algorithmic variances. A version of this scheme is available in the shared-memory programming model OpenMP [20]. An interesting class of dynamic load-balancing algorithms, well suited to the characteristics of scientific applications, are derived from theoretical advances in scheduling parallel loop iterations with variable running times. For example, Banicescu and her collaborators [1, 2, 3, 4], have recently developed and evaluated dynamic loop-scheduling algorithms based on probabilistic analysis called "factoring" and "fractiling." These schemes accommodate imbalances caused by predictable phenomena (e.g., irregular data) as well as unpredictable phenomena (e.g., data-access latency and operating system interference). At the same time, they maintain data locality by exploiting the self-similarity property of fractals. Loop iterates are executed in "chunks" of decreasing size, such that earlier, larger chunks have relatively little overhead, and their unevenness in execution time can be smoothed over by later, smaller chunks. The selection of chunk sizes requires that chunks have a high probability of completion before the optimal time. These schemes allow the scheduled batches of chunks (where each batch contains P chunks run on P processors) to be fixed portions of those remaining. For highly heterogeneous computing environments, adaptively weighted versions of these approaches account for the variability in processors' performance.

1.7 Conclusion

Partitioning and load balancing continue to be active areas of research, with efforts addressing new applications, new strategies, new partitioning goals, and new parallel architectures. Applications such as clustering and contact detection for crash simulations require enhancements and clever application of existing technologies. Linear programming, circuit simulations, preconditioners, and adaptive methods require richer models. Partitioners that account for processor and network capabilities and availability are needed for effective decompositions on emerging architectures such as clusters and grid computers. As described in this paper, many of these issues are being addressed today, with even greater results expected in the future.

Looking forward, partitioning research will continue to be driven by new applications and architectures. These applications and architectures will be characterized by even less structure than those commonly used today. For example, partitioning extremely large, irregular datasets (e.g., web graphs, social networks, intelligence databases, protein-protein interactions) is only beginning to be addressed. Agent-based computing can have complex, changing relationships between data and only small amounts of computation associated with each agent, making fast, dynamic partitioning with strict control of communication costs important. Differences between computing, memory access, and communication speeds may require partitioners that are sensitive not only to processors and networks, but also to memory hierarchies. In all, partitioning and load balancing are exciting and important areas on the frontiers of scientific computing.

Bibliography

[1] I. Banicescu and R. Carino, *Advances in dynamic load balancing techniques for scientific applications*, in Proc. 12th SIAM Conf. Parallel Processing for Scientific Computing, SIAM, Feb. 2004.

[2] I. Banicescu, S. Ghafoor, V. Velusamy, S. H. Russ, and M. Bilderback, *Experiences from integrating algorithmic and systemic load balancing strategies*, Concurrency and Computation: Practice and Experience, 13 (2001), pp. 121–139.

[3] I. Banicescu and V. Velusamy, *Load balancing highly irregular computations with the adaptive factoring*, in Procs. 16th Int'l Parallel and Distributed Processing Symp., IEEE Computer Society, 2002, p. 195.

[4] I. Banicescu, V. Velusamy, and J. Devaprasad, *On the scalability of dynamic scheduling scientific applications with adaptive weighted factoring*, Cluster Computing, 6 (2003), pp. 215–226.

[5] A. Ben-Dor and Y. Z., *Clustering gene expression patterns*, in Int'l Conf. Computational Molecular Biology, 1999.

[6] M. J. Berger and S. H. Bokhari, *A partitioning strategy for nonuniform problems on multiprocessors*, IEEE Trans. Computers, C-36 (1987), pp. 570–580.

[7] M. Berzins, *A new metric for dynamic load balancing*, Appl. Math. Modelling, 25 (2000), pp. 141–151.

[8] ——, *A new algorithm for adaptive load balancing*, in Proc. 12th SIAM Conf. Parallel Processing for Scientific Computing, SIAM, Feb. 2004.

[9] R. Bisseling and W. Meesen, *Communication balancing in Mondriaan sparse matrix partitioning*, in Proc. 12th SIAM Conf. Parallel Processing for Scientific Computing, SIAM, Feb. 2004.

[10] R. H. Bisseling, *Parallel Scientific Computing: A structured approach using BSP and MPI*, Oxford University Press, 2004.

[11] E. Boman, K. Devine, R. Heaphy, B. Hendrickson, M. Heroux, and R. Preis, *LDRD report: Parallel repartitioning for optimal solver performance*, Tech. Report SAND2004–0365, Sandia National Laboratories, Albuquerque, NM, Feb. 2004.

[12] E. G. Boman, *Multicriteria load balancing for scientific computations*, in Proc. 12th SIAM Conf. Parallel Processing for Scientific Computing, SIAM, Feb. 2004.

[13] T. Bui and C. Jones, *A heuristic for reducing fill in sparse matrix factorization*, in Proc. 6th SIAM Conf. Parallel Processing for Scientific Computing, SIAM, 1993, pp. 445–452.

[14] A. Caldwell, A. Kahng, and J. Markov, *Design and implementation of move-based heuristics for VLSI partitioning*, ACM J. Experimental Algs., 5 (2000).

[15] A. E. Caldwell, A. B. Kahng, and I. L. Markov, *Improved algorithms for hypergraph bipartitioning*, in Proc. Asia and South Pacific Design Automation Conf., IEEE Press/ ACM Press, 2000, pp. 661–666.

[16] Ü. Çatalyürek, *Hypergraph models for load balancing irregular applications*, in Proc. 12th SIAM Conf. Parallel Processing for Scientific Computing, SIAM, Feb. 2004.

[17] Ü. Çatalyürek and C. Aykanat, *Decomposing irregularly sparse matrices for parallel matrix-vector multiplications*, Lecture Notes in Computer Science, 1117 (1996), pp. 75 – 86.

[18] ——, *Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication*, IEEE Trans. Parallel Dist. Systems, 10 (1999), pp. 673–693.

[19] ——, *A fine-grain hypergraph model for 2d decomposition of sparse matrices*, in Proc. IPDPS 8th Int'l Workshop on Solving Irregularly Structured Problems in Parallel (Irregular 2001), April 2001.

[20] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, and J. McDonald, *Parallel Programming in OpenMP*, Morgan Kaufmann, Elsevier, 2000.

[21] C. Chang, T. Kurc, A. Sussman, Ü. Çatalyürek, and J. Saltz, *A hypergraph-based workload partitioning strategy for parallel data aggregation*, in Proc. 11th SIAM Conf. Parallel Processing for Scientific Computing, SIAM, March 2001.

[22] G. Cybenko, *Dynamic load balancing for distributed memory multiprocessors*, J. Parallel Distrib. Comput., 7 (1989), pp. 279–301.

[23] S. de Amorim, J. Barthelemy, and C. Ribeiro, *Clustering and clique partitioning: Simulated annealing and tabu search approaches*, Journal of Classification, 9 (1992), pp. 17–42.

[24] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, and C. Vaughan, *Zoltan data management services for parallel dynamic applications*, Computing in Science and Engineering, 4 (2002), pp. 90–97.

[25] K. Devine and J. Flaherty, *Parallel adaptive* hp*-refinement techniques for conservation laws*, Appl. Numer. Math., 20 (1996), pp. 367–386.

[26] K. Devine, B. Hendrickson, E. Boman, M. St. John, and C. Vaughan, *Zoltan: A Dynamic Load Balancing Library for Parallel Applications; User's Guide*, Sandia National Laboratories, Albuquerque, NM, 1999. Tech. Report SAND99-1377 `http://www.cs.sandia.gov/Zoltan/ug_html/ug.html`.

[27] K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and U. V. Catalyurek, *Parallel hypergraph partitioning for scientific computing*, in Proc. of IPDPS'06, IEEE, 2006. To appear.

[28] K. D. Devine, E. G. Boman, R. T. Heaphy, B. A. Hendrickson, J. D. Teresco, J. Faik, J. E. Flaherty, and L. G. Gervasio, *New challenges in dynamic load balancing*, Appl. Num. Math., 53 (2005), pp. 133–152.

[29] I. S. Dhillon, *Co-clustering documents and words using bipartite spectral graph partitioning*, Tech. Report TR-2001-05, Dept. Computer Science, University of Texas, Austin, 2001.

[30] C. H. Q. Ding, X. He, H. Zha, M. Gu, and H. D. Simon, *A min-max cut algorithm for graph partitioning and data clustering*, in Proc. ICDM 2001, 2001, pp. 107–114.

[31] R. Duda, P. Hart, and D. Stork, *Pattern Classification*, John Wiley & Sons, 2001.

[32] H. C. Edwards, *A Parallel Infrastructure for Scalable Adaptive Finite Element Methods and its Application to Least Squares C^∞ Collocation*, PhD thesis, The University of Texas at Austin, May 1997.

[33] J. Faik, *Dynamic load balancing on heterogeneous clusters*, in Proc. 12th SIAM Conf. Parallel Processing for Scientific Computing, SIAM, Feb. 2004.

[34] J. Flaherty, R. Loy, M. Shephard, B. Szymanski, J. Teresco, and L. Ziantz, *Adaptive local refinement with octree load-balancing for the parallel solution of three-dimensional conservation laws*, J. Parallel Distrib. Comput., 47 (1998), pp. 139–152.

[35] J. E. Flaherty, R. M. Loy, M. S. S. an d Boleslaw K. Szymanski, J. D. Teresco, and L. H. Ziantz, *Predictive load balancing for parallel adaptive finite element computation*, in Proc. PDPTA '97, H. R. Arabnia, ed., vol. I, 1997, pp. 460–469.

[36] J. Garbers, H. J. Promel, and A. Steger, *Finding clusters in VLSI circuits*, in Proc. IEEE Int'l Conf. Computer Aided Design, 1990, pp. 520–523.

[37] C. Hagen and A. Kahng, *New spectral methods for ratio cut partitioning and clustering*, IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, 11 (1992).

[38] L. Hagen and A. Kahng, *A new approach to effective circuit clustering*, in Proc. IEEE Int'l Conf. Computer Aided Design, 1992, pp. 422–427.

[39] J. Han, M. Kamber, and A. K. H. Tung, *Spatial clustering methods in data mining: A survey*, in Geographic Data Mining and Knowledge Discovery, H. Miller and J. Han, eds., Taylor and Francis, 2001.

[40] R. Heaphy, *Load balancing contact deformation problems using the Hilbert space filling curve*, in Proc. 12th SIAM Conf. Parallel Processing for Scientific Computing, SIAM, Feb. 2004.

[41] B. Hendrickson, *Graph partitioning and parallel solvers: Has the emperor no clothes?*, Lecture Notes in Computer Science, 1457 (1998), pp. 218 – 225.

[42] B. Hendrickson and T. G. Kolda, *Graph partitioning models for parallel computing*, Parallel Computing, 26 (2000), pp. 1519 – 1534.

[43] B. Hendrickson and R. Leland, *The Chaco user's guide, version 2.0*, Tech. Report SAND94–2692, Sandia National Laboratories, Albuquerque, NM, October 1994.

[44] ——, *A multilevel algorithm for partitioning graphs*, in Proc. Supercomputing '95, ACM, December 1995.

[45] Y. Hu and R. Blake, *An optimal dynamic load balancing algorithm*, Tech. Report DL-P-95-011, Daresbury Laboratory, Warrington, WA4 4AD, UK, Dec. 1995.

[46] Y. F. Hu, R. J. Blake, and D. R. Emerson, *An optimal migration algorithm for dynamic load balancing*, Concurrency: Practice and Experience, 10 (1998), pp. 467 – 483.

[47] A. K. Jain, M. N. Murty, and P. J. Flynn, *Data clustering: A review*, ACM Computing Surveys, 31 (1999), pp. 264–323.

[48] M. T. Jones and P. E. Plassmann, *Computational results for parallel unstructured mesh computations*, Computing Systems in Engineering, 5 (1994), pp. 297–309.

[49] S. Karisch and F. Rendl, *Semidefinite programming and graph equipartition*, vol. 18, AMS, 1998, pp. 77–95.

[50] G. Karypis, *Multi-constraint mesh partitioning for contact/impact computations*, in Proc. SC2003, Phoenix, AZ, 2003, ACM.

[51] G. Karypis, *Graph partitioning based data clustering: Problems, models, and algorithms*, in Proc. 12th SIAM Conf. Parallel Processing for Scientific Computing, SIAM, Feb. 2004.

[52] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, *Multilevel hypergraph partitioning: application in VLSI domain*, in Proc. 34th Conf. Design Automation, ACM, 1997, pp. 526 – 529.

[53] G. Karypis and V. Kumar, *Parmetis: Parallel graph partitioning and sparse matrix ordering library*, Tech. Report 97-060, Dept. Computer Science, University of Minnesota, 1997. `http://www.cs.umn.edu/~metis`.

[54] ——, *Multilevel algorithms for multiconstraint graph paritioning*, Tech. Report 98-019, Dept. Computer Science, University of Minnesota, 1998.

[55] G. Karypis and V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, SIAM J. Sci. Comput., 20 (1999).

[56] B. W. Kernighan and S. Lin, *An efficient heuristic procedure for partitioning graphs*, Bell System Technical Journal, 49 (1970), pp. 291–307.

[57] J. Lawder and P. King, *Using space-filling curves for multi-dimensional indexing*, Lecture Notes in Computer Science, 1832 (2000).

[58] E. Leiss and H. Reddy, *Distributed load balancing: design and performance analysis*, W.M. Keck Research Computation Laboratory, 5 (1989), pp. 205–270.

[59] B. Lowekamp, N. Miller, D. Sutherland, T. Gross, P. Steenkiste, and J. Subhlok, *A resource query interface for network-aware applications*, in Proc. 7th IEEE Symp. on High-Performance Distributed Computing, 1998.

[60] B. Maerten, D. Roose, A. Basermann, J. Fingberg, and G. Lonsdale, *DRAMA: A library for parallel dynamic load balancing of finite element applications*, in Proc. Ninth SIAM Conf. Parallel Processing for Scientific Computing, San Antonio, TX, 1999.

[61] F. Manne and T. Sørevik, *Partitioning an array onto a mesh of processors*, in Proc. Para'96, Workshop on Applied Parallel Computing in Industrial Problems and Optimization, vol. 1184, Lecture Notes in Computer Science, Springer, 1996, pp. 467–477.

[62] D. W. Matula, *k-components, clusters and slicings in graphs*, SIAM Journal on Applied Mathematics, 22 (1972), pp. 459–480.

[63] T. Minyard and Y. Kallinderis, *Octree partitioning of hybrid grids for parallel adaptive viscous flow simulations*, Int. J. Numer. Meth. Fluids, 26 (1998), pp. 57–78.

[64] ——, *Parallel load balancing for dynamic execution environments*, Comput. Methods Appl. Mech. Engrg., 189 (2000), pp. 1295–1309.

[65] W. F. Mitchell, *Refinement tree based partitioning for adaptive grids*, in Proc. Seventh SIAM Conf. Parallel Processing for Scientific Computing, SIAM, 1995, pp. 587–592.

[66] ——, *The refinement-tree partition for parallel solution of partial differential equations*, NIST Journal of Research, 103 (1998), pp. 405–414.

[67] D. Moore, *Fast hilbert curve generation, sorting and range queries.* `http://www.caam.rice.edu/~dougm/twiddle/Hilbert`.

[68] L. Oliker and R. Biswas, *PLUM: Parallel load balancing for adaptive unstructured mesh es*, J. Parallel Distrib. Comput., 51 (1998), pp. 150–177.

[69] L. Oliker, R. Biswas, and R. C. Strawn, *Parallel implementaion of an adaptive scheme for 3D unstructured grids on the SP2*, in Proc. 3rd Int'l Workshop on Parallel Algorithms for Irregularly Structured Problems, Santa Barbara, 1996.

[70] S. Oliveira, D. Stewart, and S. Toma, *A subspace semidefinite programming for spectral graph partitioning*, Lecture Notes in Computer Science, 2329 (2002), pp. 1058–1067. Proc. Int'l Conf. Computational Science; April 2002; Amsterdam, Netherlands.

[71] ——, *Semidefinite programming for graph partitioning with preferences in data distribution*, Lecture Notes in Computer Science, 2565 (2003), pp. 703–716. Proc. VECPAR 2002; June 2002; Oporto, Portugal.

[72] S. P. Oliveira and S.-C. Seok, *Semidefinite approaches to load balancing*, in Proc. 12th SIAM Conf. Parallel Processing for Scientific Computing, SIAM, Feb. 2004.

[73] M. Ozdal and C. Aykanat, *Hypergraph models and algorithms for data-pattern based clustering*, Data Mining and Knowledge Discovery, (2004). Accepted for publication.

[74] A. Patra and J. T. Oden, *Problem decomposition for adaptive* hp *finite element methods*, J. Computing Systems in Engg., 6 (1995).

[75] F. Pelligrini, *SCOTCH 3.4 user's guide*, Research Rep. RR-1264-01, LaBRI, Nov. 2001.

[76] A. Pinar, *Alternative models for load balancing*, in Proc. 12th SIAM Conf. Parallel Processing for Scientific Computing, SIAM, Feb. 2004.

[77] A. Pinar and C. Aykanat, *Sparse matrix decomposition with optimal load balancing*, in Proc. Int'l Conf. High Performance Computing, Dec. 1997, pp. 224–229.

[78] A. Pinar and B. Hendrickson, *Graph partitioning for complex objectives*, in Proc. 15th Int'l Parallel and Distributed Processing Symp. (IPDPS), San Francisco, CA, April 2001.

[79] ——, *Exploiting flexibly assignable work to improve load balance*, in Proc. ACM 14th Symp. Parallel Algorithms and Architectures (SPAA), Winnipeg, Canada, August 2002, pp. 155–163.

[80] S. Plimpton, S. Attaway, B. Hendrickson, J. Swegle, C. Vaughan, and D. Gardner, *Transient dynamics simulations: Parallel algorithms for contact detection and smoothed particle hydrodynamics*, J. Parallel Distrib. Comput., 50 (1998), pp. 104–122.

[81] A. Pothen, H. Simon, and K. Liou, *Partitioning sparse matrices with eigenvectors of graphs*, SIAM J. Matrix Anal., 11 (1990), pp. 430–452.

[82] R. Preis and R. Diekmann, *The PARTY partitioning library, user guide version 1.1*, Tech. Rep. tr-rsfb-96-024, Dept. Computer Science, University of Paderborn, Paderborn, Germany, Sept. 1996.

[83] L. F. Romero and E. L. Zapata, *Data distributions for sparse matrix multiplication*, Parallel Comput., 21 (1995), pp. 585–605.

[84] K. Schloegel, G. Karypis, and V. Kumar, *Multilevel diffusion algorithms for repartitioning of adaptive meshes*, J. Parallel Distrib. Comput., 47 (1997), pp. 109–124.

[85] ——, *A new algorithm for multi-objective graph partitioning*, Tech. Report 99-003, University of Minnesota, Dept. Computer Science and Army HPC Center, Minneapolis, 1999.

[86] ——, *A unified algorithm for load-balancing adaptive scientific simulations*, in Proc. Supercomputing, Dallas, 2000.

[87] ——, *Parallel static and dynamic multiconstraint graph partitioning*, Concurrency and Computation – Practice and Experience, 14 (2002), pp. 219–240.

[88] F. Shahrokhi and D. Matula, *The maximum concurrent flow problem*, Journal of the ACM, 37 (1990), pp. 318–334.

[89] R. Shamir and R. Sharan, *Click: A clustering algorithm for gene expression analysis*, in the 8th Int'l Conf. Intelligent Systems for Molecular Biology, 2000.

[90] J. Shi and J. Malik, *Normalized cuts and image segmentation*, IEEE Trans. Pattern Analysis and Machine Intelligence, 22 (2000), pp. 888–905.

[91] H. D. Simon, *Partitioning of unstructured problems for parallel processing*, Comp. Sys. Engng., 2 (1991), pp. 135–148.

[92] S. Sinha and M. Parashar, *Adaptive system partitioning of AMR applications on heterogeneous clusters*, Cluster Computing, 5 (2002), pp. 343–352.

[93] S. Tamura, *Clustering based on multiple paths*, Pattern Recognition, 15 (1982), pp. 477–483.

[94] V. E. Taylor and B. Nour-Omid, *A study of the factorization fill-in for a parallel implementation of the finite element method*, Int. J. Numer. Meth. Engng., 37 (1994), pp. 3809–3823.

[95] J. D. Teresco, *Hierarchical partitioning and dynamic load balancing for scientific computation*, Tech. Report CS-04-04, Williams College Dept. Computer Science, 2004.

[96] J. D. Teresco, M. W. Beall, J. E. Flaherty, and M. S. Shephard, *A hierarchical partition model for adaptive finite element computation*, Comput. Methods Appl. Mech. Engrg., 184 (2000), pp. 269–285.

[97] A. Trifunovic and W. J. Knottenbelt, *Parkway 2.0: A parallel multilevel hypergraph partitioning tool*, in Proc. 19th International Symposium on Computer and Information Sciences (ISCIS 2004), vol. 3280 of LNCS, Springer, 2004, pp. 789–800.

[98] S. van Dongen, *A cluster algorithm for graphs*, Tech. Report INS-R0010, National Research Institute for Mathematics and Computer Science in the Netherlands, 2000.

[99] B. Vastenhouw and R. H. Bisseling, *A two-dimensional data distribution method for parallel sparse matrix-vector multiplication*, SIAM Review, 47 (2005), pp. 67–95.

[100] C. Walshaw, *The Parallel JOSTLE Library User's Guide, Version 3.0*, University of Greenwich, London, UK, 2002.

[101] C. Walshaw and M. Cross, *Multilevel Mesh Partitioning for Heterogeneous Communication Networks*, Future Generation Comput. Syst., 17 (2001), pp. 601–623.

[102] C. Walshaw, M. Cross, and K. McManus, *Multiphase mesh partitioning*, Appl. Math. Modelling, 25 (2000), pp. 123–140.

[103] H. Wang, B. Alidaee, and G. Kochenberger, *Evaluating a clique partitioning problem model for clustering high-dimensional data mining*, in 10th Americas Conf. Information Systems, 2004, pp. 1946–1960.

[104] M. S. Warren and J. K. Salmon, *A parallel hashed oct-tree n-body algorithm*, in Proc. Supercomputing '93, Portland, OR, Nov. 1993.

[105] S. Wheat, *A fine-grained data migration approach to application load balancing on MP MIMD machines*, PhD thesis, The University of New Mexico, Dept. Computer Science, Albuquerque, NM, 1992.

[106] H. Wolkowicz and Q. Zhao, *Semidefinite programming relaxations for the graph partitioning problem*, Discrete Applied Mathematics, 96–97 (1999), pp. 461–79.

[107] R. Wolski, N. T. Spring, and J. Hayes, *The Network Weather Service: A distributed resource performance forecasting service for metacomputing*, Future Generation Comput. Syst., 15 (1999), pp. 757–768.

[108] C. W. Yeh, C. K. Cheng, and T. T. Lin, *Circuit clustering using a stochastic flow injection method*, IEEE Trans. Computer Aided Design of Integrated Circuits and Systems, 14 (1995), pp. 154–162.

[109] H. Zha, X. He, C. Ding, H. Simon, and M. Gu, *Bipartite graph partitioning and data clustering*, in CIKM, 2001.