A Repartitioning Hypergraph Model for Dynamic Load Balancing ⋆

Umit V. Catalyurek [a,b,*] Erik G. Boman [c,1], Karen D. Devine [c,1], Doruk Bozdağ [b], Robert T. Heaphy [c,1], Lee Ann Riesen [c,1]

[a] *The Ohio State University, Department of Biomedical Informatics*

[b] *The Ohio State University, Department of Electrical and Computer Engineering*

[c] *Sandia National Laboratories, Department of Scalable Algorithms*

Abstract

In parallel adaptive applications, the computational structure of the applications changes over time, leading to load imbalances even though the initial load distributions were balanced. To restore balance and to keep communication volume low in further iterations of the applications, dynamic load balancing (repartitioning) of the changed computational structure is required. Repartitioning differs from static load balancing (partitioning) due to the additional requirement of minimizing migration cost to move data from an existing partition to a new partition. In this paper, we present a novel repartitioning hypergraph model for dynamic load balancing that accounts for both communication volume in the application and migration cost to move data, in order to minimize the overall cost. Use of a hypergraph-based model allows us to accurately model communication costs rather than approximating them with graph-based models. We show that the new model can be realized using hypergraph partitioning with fixed vertices and describe our parallel multilevel implementation within the Zoltan load-balancing toolkit. To the best of our knowledge, this is the first implementation for dynamic load balancing based on hypergraph partitioning. To demonstrate the effectiveness of our approach, we conducted experiments on a Linux cluster with 1024 processors. The results show that, in terms of reducing total cost, our new model compares favorably to the graph-based dynamic load balancing approaches, and multilevel approaches improve the repartitioning quality significantly.

Key words: Dynamic load balancing, hypergraph partitioning, parallel algorithms, scientific computing, distributed memory computers

1 Introduction

An important component of many scientific computing applications is the *assignment* of computational load onto a set of processors. The problem of finding a task-to-processor mapping that minimizes the total execution time is known as the *mapping* problem [5,6,10,12,13,15,25,46,47,52,53,57]. Although efficient optimal solutions exist for certain restricted cases, such as chain- or tree-structured programs [35], the general mapping problem is NP-hard [40]. In this paper we consider the general problem where any task can potentially be assigned to any processor. In the literature, a two-step approach is commonly employed to solve the problem: first tasks are *partitioned* into *load-balanced* clusters of tasks, then these clusters are *mapped* to processors [10,47]. In the partitioning step, a common goal is to minimize the interprocessor communication while maintaining a computational load balance among processors. Partitioning occurs at the start of a computation (*static partitioning*), but often, reassignment of work is done during a computation (*dynamic partitioning* or *repartitioning*) as the work distribution changes over the course of the computation. time. For instance, a computational mesh in an adaptive mesh refinement simulation is updated between time steps. Therefore, after several steps, even an initially balanced assignment of work to processors may suffer serious imbalances. To maintain the balance in subsequent computation steps, a repartitioning procedure that moves data among processors needs to be applied periodically.

Repartitioning is a well-studied problem [19,20,24,27,32,33,42,43,50,51,55,53,58,61,62] that has multiple objectives with complicated trade-offs among them:

(1) balanced load in the new data distribution;
(2) low communication cost within the application (as determined by the new distribution);
(3) low data migration cost to move data from the old to the new distribution; and

⋆ The research was supported in part by the National Science Foundation under Grants #CNS-0643969, #CCF-0342615, #CNS-0426241, #CNS-0403342, NIH NIBIB BISTI #P20EB000591, Ohio Supercomputing Center #PAS0052, and by the Department of Energy's Office of Science through the CSCAPES SciDAC Institute.
* Corresponding author. Tel: 614-292-0914, Fax: 614-688-6600.
Email addresses: umit@bmi.osu.edu (Umit V. Catalyurek), egboman@sandia.gov (Erik G. Boman), kddevin@sandia.gov (Karen D. Devine), bozdagd@ece.osu.edu (Doruk Bozdağ), rheaphy@sandia.gov (Robert T. Heaphy), lafisk@sandia.gov (Lee Ann Riesen).
[1] Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin company, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

(4) short repartitioning time.

Total application execution time is commonly modeled [39,50] as follows to account for these objectives:

$$t_{tot} = \alpha(t_{comp} + t_{comm}) + t_{mig} + t_{repart}. \quad (1)$$

Here, t_{comp} and t_{comm} denote the application's computation and communication times, respectively, in a single iteration of the application; t_{mig} is the data migration time from existing to new partitions; and t_{repart} is the repartitioning time. The parameter α indicates how many iterations (e.g., time steps in a simulation) of the application are executed between each load-balance operation.

The computation time t_{comp} of a parallel application is minimized when the computational load is evenly distributed on the set of processors. Since achieving load balance is the main constraint on repartitioning algorithms, we can safely assume that the computational load will be balanced; hence t_{comp} is inherently minimized by the repartitioning algorithm. Furthermore, the time required by state-of-the-art repartitioning programs to produce a new partitioning is typically much smaller than αt_{comp}. As a result, t_{comp} and t_{repart} in (1) can be ignored; the cost function to be minimized by the repartitioning algorithm reduces to

$$cost_{time} = \alpha t_{comm} + t_{mig} \quad (2)$$

Because time for communication depends on a number of architecture-specific factors (e.g., network topology, message latency), general partitioning models typically assume the time spent in communication is proportional to the "volume" of communication, i.e., the amount of data being sent [29]. Thus, the cost function to be minimized by the repartitioning algorithm becomes

$$cost_{vol} = \alpha b_{comm} + b_{mig} \quad (3)$$

where b_{comm} is the amount of data sent in each iteration of the application and b_{mig} is the amount of data sent during migration.

In this work, we present a *repartitioning-hypergraph model* that minimizes the sum of total communication volume in the application and migration cost to move data, as stated in (3). The repartitioning-hypergraph model is first introduced in our preliminary work in [14]. Hypergraphs accurately model the actual application communication cost and have greater applicability than graphs (e.g., hypergraphs can represent non-symmetric and/or non-square systems) [16]. Therefore, the actual value of b_{comm} is considered in the proposed

model, rather than its approximation as in the case of graph-based models [29]. Furthermore, in our repartitioning-hypergraph model, communication and migration costs are appropriately combined to allow reuse of existing hypergraph partitioners for repartitioning. The new model can be realized effectively with a hypergraph partitioning tool that provides hypergraph partitioning with fixed vertices. Although serial hypergraph partitioners with this feature exist ([3,17]), to the best of our knowledge our implementation in Zoltan [7] is the first parallel hypergraph partitioner that can handle fixed vertices.

The remainder of this paper is organized as follows. In Section 2, we discuss previous work on dynamic load balancing. We present preliminaries for hypergraph partitioning and multilevel partitioning in Section 3. The details of the proposed repartitioning-hypergraph model are presented in Section 4. Section 5 describes the parallel hypergraph-based repartitioning algorithm developed within the Zoltan [63] toolkit. Section 6 includes a detailed empirical comparison of various graph- and hypergraph-based repartitioning approaches. Finally, in Section 7, we give our conclusions and suggest future work.

2 Related Work

Dynamic load-balancing approaches can be classified into three main categories: *scratch-remap*, *incremental* and *repartitioning*. In scratch-remap methods, the computational model representing the modified structure of the application is partitioned *from scratch* without accounting for existing part assignments. Then, old and new partitions are remapped to minimize the migration cost [51,42]. In incremental methods, existing part assignments are used as initial assignments and *incrementally* improved by using a sub-optimal cost function that minimizes either data migration cost (*diffusive* methods) or application communication cost (*refinement* methods). In repartitioning methods, existing part assignments are taken into account to minimize both data migration cost and application communication cost as stated in (3).

Another way of classifying dynamic load balancing methods is with respect to the computational models they use. There are three computational models commonly used in the literature. These are *coordinate-based*, *graph-based* and *hypergraph-based* models. Table 1 summarizes properties of dynamic load balancing approaches in each category.

Some of the early dynamic load-balancing techniques are coordinate-based approaches such as *Recursive Coordinate Bisection* [4] and *Space-Filling Curves* [44,45,60]. These approaches can be applied either from scratch or incrementally. They require geometric coordinates and do not model communication or migration costs explicitly. Still, due to structure of the application data, they often work

Category	Property	Coordinate Based	Graph Based	Hypergraph Based
Scratch-remap	Migration cost	high	high	high
	Communication cost	high	low	low
	Communication model	none	approximate	accurate
Incremental	Migration cost	moderate	low	low
	Communication cost	high	moderate	moderate
	Communication model	none	approximate	accurate
Repartitioning	Migration cost	n/a	low	low
	Communication cost	n/a	low	low
	Communication model	none	approximate	accurate

Table 1
Classification of dynamic load-balancing approaches, with their relative migration costs, application communication costs, and communication model.

reasonably well for mesh partitioning.

Diffusive methods have been one of the most studied incremental dynamic load-balancing techniques in the literature [19,33,34,43,48,58,61]. In diffusive load balancing, extra work on overloaded processors is distributed to neighboring processors that have less than average loads. This strategy inherently limits data migration cost. Some diffusive methods explicitly try to minimize application communication cost using an approximation model (e.g., [48]); however, since each minimization is done independently, these methods are not equivalent to global minimization of total costs in (3).

Even though scratch-remap schemes achieve low communication volume, they often result in high migration cost. On the other hand, incremental methods result in low migration cost, but they may incur moderate to high communication volume. In dynamic load balancing, it is desirable that the repartitioning algorithm is sensitive to the iteration parameter α, so that the relative weight of communication cost to migration cost in (3) can be adjusted by the application developer. *Skewed Graph Partitioning* introduced by Hendrickson et al. [32] gives such a control to the application developer, by giving each vertex a *desire* to stay in its current processor. Schloegel et al. [50] proposed a parallel adaptive repartitioning scheme, where relative importance of migration time against communication time is set by a user-provided parameter. Their work is based on the multilevel graph partitioning paradigm, and this parameter is taken into account in the refinement phase of the multilevel scheme. Aykanat et al. [2] proposed a graph-based repartitioning model, called *RM model*, where the original computational graph is augmented with new vertices and edges to

account for migration cost. Then, repartitioning with fixed vertices is applied to the graph using RM-METIS, a serial repartitioning tool that the authors developed by modifying the graph partitioning tool METIS [36]. Although the approaches of Hendrickson et al. [32], Schloegel et al. [50] and Aykanat et al. [2] attempt to minimize both communication and migration costs, their applicability is limited to problems with symmetric, bi-directional dependencies. A hypergraph-based model is proposed in a concurrent work of Cambazoglu and Aykanat [11] for the adaptive screen partitioning problem in the context of image-space-parallel direct volume rendering of unstructured grids. Despite the fact that the limitations mentioned above for graph-based models do not apply, their model accounts only for migration cost since communication occurs merely for data replication (migration) in that problem.

3 Preliminaries

In this section, we present a brief description of hypergraph partitioning with fixed vertices as well as the multilevel partitioning paradigm.

3.1 Hypergraph Partitioning with Fixed Vertices

Hypergraphs can be viewed as generalizations of graphs where an edge is not restricted to connect only two vertices. Formally, a hypergraph $H = (V, N)$ is defined by a set of vertices V and a set of nets (hyperedges) N, where each net $n_j \in N$ is a non-empty subset of vertices. A weight w_i can be assigned to each vertex $v_i \in V$, and a cost c_j can be assigned to each net $n_j \in N$.

$P = \{V_1, V_2, \ldots, V_k\}$ is called a k-way partition of H if each part $V_p, p = 1, 2, \ldots, k$, is a non-empty, pairwise-disjoint subset of V and $\cup_{p=1}^{k} V_p = V$. A partition is said to be *balanced* if

$$W_p \leq W_{avg}(1 + \epsilon) \text{ for } p = 1, 2, \ldots, k, \tag{4}$$

where part weight $W_p = \sum_{v_i \in V_p} w_i$ and $W_{avg} = \left(\sum_{v_i \in V} w_i\right)/k$, and $\epsilon > 0$ is a predetermined maximum tolerable imbalance.

In a given partition P, a net that has at least one vertex in a part is considered to be connected to that part. The *connectivity* λ_j of a net n_j denotes the number of parts connected by n_j under the partition P of H. A net n_j is said to be *cut* if it connects more than one part (i.e., $\lambda_j > 1$).

Let $CutCost(H, P)$ denote the cost associated with a partition P of hypergraph H. There are various ways to define $CutCost(H, P)$ [41]. The relevant one for our context is known as *connectivity-1* (or k-1) metric, defined as follows:

$$CutCost(H, P) = \sum_{n_j \in N} c_j(\lambda_j - 1) \tag{5}$$

We prefer this cost metric because it exactly corresponds to communication volume in parallel computing for important operations like matrix-vector multiplication [16]. The standard hypergraph partitioning problem [41] can then be stated as the task of dividing a hypergraph into k parts such that the cost (5) is minimized while the balance criterion (4) is maintained.

Hypergraph partitioning with fixed vertices is a more constrained version of the standard hypergraph partitioning problem. In this problem, in addition to the input hypergraph H and the requested number of parts k, a *fixed-part* function $f(v)$ is also provided as an input to the problem. A vertex is said to be *free* (denoted by $f(v) = -1$) if it is allowed to be in any part in the solution P, and it is said to be fixed in part q ($f(v) = q$ for $1 \leq q \leq k$) if it is required to be in V_q in the final solution P. If a significant portion of the vertices are fixed, it is expected that the partitioning problem becomes easier. Clearly, in the extreme case where all the vertices are fixed (i.e., $f(v) \neq -1$ for all $v \in V$), the solution is trivial. Empirical studies of Alpert et al. [1] verify that the presence of fixed vertices can make a partitioning instance considerably easier. However, to the best of our knowledge, there is no theoretical work on the complexity of the problem. Experience shows that if only a very small fraction of vertices are fixed, the problem is almost as "hard" as the standard hypergraph partitioning problem.

3.2 Multilevel Partitioning Paradigm

Although graph and hypergraph partitioning are NP-hard [28,41], several algorithms based on multilevel paradigms [9,31,37] have been shown to compute high quality partitions in reasonable time. In addition to serial partitioners for graphs [30,36,56] and hypergraphs [17,38], the multilevel partitioning paradigm has been adopted by parallel graph [56,39] and, quite recently, hypergraph [22,54] partitioners as well.

Multilevel partitioning consists of three phases: *coarsening*, *coarse partitioning* and *refinement*. Instead of partitioning the original hypergraph directly, a hierarchy of smaller hypergraphs that approximate the original one is generated during the *coarsening* phase. The smallest hypergraph obtained at the

end of the coarsening phase is partitioned in the *coarse partitioning* phase. Finally, in the *refinement* phase, the coarse partition is projected back to the larger hypergraphs in the hierarchy and improved using a local optimization method. The same procedure applies to graphs as well.

In Section 5, we describe a technique for parallel multilevel hypergraph partitioning with fixed vertices [14]. The implementation is based on the parallel hypergraph partitioner in Zoltan [22].

4 Repartitioning Hypergraph Model

In this section, we present our novel hypergraph model and explain how it accounts for the trade-off between communication and migration costs due to different values of α. By representing these costs appropriately in a *repartitioning hypergraph*, the proposed approach allows use of existing hypergraph partitioning tools to optimize the composite objective defined in (3).

We call the period between two subsequent load-balancing operations an *epoch* of the application. An epoch consists of one or more computation iterations and the computational structure and dependencies of an epoch can be accurately modeled with a computational hypergraph [16]. Even though computations in the application are of the same type, a different hypergraph is needed to represent each epoch due to changes in the structure of the hypergraph across epochs. We denote the hypergraph that models the jth epoch of the application by $H^j = (V^j, N^j)$ and the number of computation iterations in that epoch by α_j.

Load balancing for the first epoch is achieved by partitioning H^1 using a static partitioner. For the remaining epochs, data redistribution cost between the previous and current epochs should also be included during load balancing. Therefore, the actual cost (3) is the sum of the communication cost b_{comm} for H^j with the new data distribution, scaled by α_j, and the migration cost b_{mig} for moving data between the distributions in epoch $j-1$ and j.

Our new repartitioning hypergraph model appropriately captures both application communication and data migration costs associated with an epoch. To model migration costs in epoch j, we construct a repartitioning hypergraph $\bar{H}^j = (\bar{V}^j, \bar{N}^j)$ by augmenting H^j with k new vertices corresponding to each of the k parts, and $|V^j|$ new hyperedges using the following procedure:

- Scale each net's cost (representing application communication) in N^j by α_j while keeping the vertex weights intact.
- Add a new *part vertex* u_i with zero weight for each part i, and fix those

vertices in respective parts; i.e., $f(u_i) = i$ for $i = 1, 2, \ldots, k$. Hence $\bar{V}^j$ becomes $V^j \cup \{u_i | i = 1, 2, \ldots, k\}$.

- For each vertex $v \in V^j$, add a *migration net* n_v between v and u_i if v is assigned to part i at the beginning of epoch j. Set the migration net's cost c_v to the size of the data associated with v, since this migration net represents the cost of moving vertex v to a different part.

Once the new repartitioning hypergraph $\bar{H}^j$ that encodes both communication and migration costs is constructed, the repartitioning problem reduces to hypergraph partitioning with *fixed* vertices using connectivity-1 metric (5).

Let $\bar{P} = \{\bar{V}_1, \bar{V}_2, \ldots, \bar{V}_k\}$ be a valid partition of $\bar{H}^j$. Since fixed part vertices have zero weights, part weights are equal to the sum of the computational vertices' weights. Therefore, maintaining the balance criterion (4) in this partition corresponds to having a balanced computation in epoch j. Minimizing the connectivity-1 cost metric (5) exactly corresponds to minimizing the repartitioning cost $cost_{vol}$ in (3). That is, for epoch j,

$$cost_{vol} = CutCost(\bar{H}^j, \bar{P}^j). \tag{6}$$

Since we obtained $\bar{H}^j$ by augmenting H^j we can further expand this formula as

$$cost_{vol} = \alpha_j CutCost(H^j, P^j) + \sum_{n_v \in (\bar{N}^j - N^j)} c_v(\lambda_v - 1), \tag{7}$$

where $P^j = \{V_1, V_2, \ldots, V_k\}$ is the same as $\bar{P}^j$ except it does not contain part vertices. In (7), the first term, $\alpha_j CutCost(H^j, P^j)$, corresponds to the amount of data sent in each iteration of the application [16] (i.e., b_{comm} in (3)) and the second term corresponds to the amount of data sent during migration (i.e., b_{mig} in (3)).

Assume that a vertex v is assigned to part p in epoch $j-1$ and part q in epoch j, where $p \neq q$. Then, the migration net n_v between v and u_p that represents the migration cost of vertex v's data is cut with connectivity of $\lambda_v = 2$ (note that u_p is fixed in part p). Therefore, the cost of moving vertex v from part p to q, c_v, is appropriately included in the total cost. If a net that represents communication during the computation phase is cut, the cost incurred by communicating the associated data in all α_j iterations in epoch j is also accounted for since the net's weight has already been scaled by α_j. Hence, our repartitioning hypergraph accurately models the sum of communication during computation phase and migration cost due to moved data.

Figure 1(a) illustrates a sample computational hypergraph H^j at the beginning of epoch j. The corresponding repartitioning hypergraph $\bar{H}^j$ is displayed in Figure 1(b). A nice feature of our model is that no distinction is required

between communication and migration nets as well as computation and part vertices. However, for clarity in this figure, we represent computation vertices with circles and part vertices with octagons. Similarly, application communication nets are represented with squares, and migration nets are represented with diamonds. In this example, at the beginning of epoch j, there are twelve computation vertices with various computational loads (represented by the numbers inside the circles). Computational load is initially in three highly imbalanced parts. Three cut nets represent data that need to be communicated among the parts. Two of these nets have connectivity $\lambda = 3$ and one has $\lambda = 2$. Assuming unit cost for each net, total communication cost (5) is five. In other words, if the application chooses to continue with this partitioning, each iteration of epoch j incurs a communication cost of five units.

In Figure 1(b), to construct the repartitioning hypergraph $\bar{H}^j$ from H^j, three part vertices u_1, u_2 and u_3 are added and net weights in H^j are scaled by α_j. Then, each of the twelve computation vertices is connected via a migration net to the part vertex associated with the part to which the computation vertex was assigned at the beginning of epoch j.

Two balanced sample solutions for the repartitioning problem are depicted in Figures 1(c) and 1(d). Assume that the sizes of the data associated with each computation vertex and application communication net are the same; i.e., communication and migration nets have unit costs. In Figure 1(c), two vertices with weights three and six are migrated from part 1 to part 2, resulting in migration cost of two and communication cost of four units at each iteration, due to four cut nets with connectivity two. In Figure 1(d), while two vertices with weights three and six are migrated from part 1 to part 3, two vertices of part 3 are migrated to part 2. This distribution results in migration cost of four and communication cost of three units at each iteration. These two solutions present an example of the trade-off between communication and migration costs in the repartitioning problem. Assume that epoch j consists of only one iteration ($\alpha_j = 1$). Then the solution presented in Figure 1(c) is better than the solution presented in Figure 1(d), because the former has a total cost of six, whereas the latter has a total cost of seven. However, if epoch j consists of ten iterations ($\alpha_j = 10$), the solution presented in Figure 1(d) is better because it has a total cost of 34, whereas Figure 1(c) has a total cost of 42. With the user-specified α_j parameter, our repartitioning hypergraph model accurately accounts for this trade-off.

5 Parallel Repartitioning Tool

The dynamic repartitioning model presented in the previous section can be implemented using parallel hypergraph partitioning with fixed vertices. In

such an implementation, the multilevel algorithms commonly used for hypergraph partitioning (as described in Section 3) are adapted to handle fixed vertices [3,17]. In each phase of the multilevel partitioning, the fixed part constraints defined by $f(v)$ must be maintained for each vertex v and its resulting coarse vertices. In this section, we describe our approach for parallel multilevel hypergraph partitioning with fixed vertices [14] using the parallel hypergraph partitioner in Zoltan [22]. We first assume that we partition directly into k parts, and later discuss how fixed vertices are handled when recursive bisection is used to obtain k parts.

5.1 Coarsening Phase

In the coarsening phase of the multilevel algorithms, we approximate the original hypergraph with a succession of smaller hypergraphs with similar connectivity and equal total vertex and edge weight. Coarsening ends when the coarsest hypergraph is "small enough" (e.g., it has fewer than $2k$ vertices) or when the last coarsening step fails to reduce the hypergraph's size by a specified amount (typically 10%). To reduce the hypergraph's size, we merge *similar* vertices, i.e., vertices whose hyperedge connectivity overlaps significantly. In this paper, we use an agglomerative matching technique that has been called as *heavy-connectivity clustering* in PaToH [17,16].

Parallel matching is performed in rounds. In each round, each processor broadcasts a subset of candidate vertices that will be matched in that round. Then, all processors concurrently compute their best match for those candidates and the global best match for each candidate is selected. In agglomerative matching, candidate vertices are allowed to join already matched vertices to form a larger cluster as long as the final cluster's size is not larger than a quarter of a target part size.

For fixed-vertex partitioning, we constrain matching to propagate fixed-vertex constraints to coarser hypergraphs so that coarser hypergraphs truly approximate the finer hypergraphs and their constraints. We do not allow vertices to match if they are fixed to different parts. Thus, there are three scenarios in which two vertices match: 1) both vertices are fixed to the same part, 2) only one of the vertices is fixed to a part, or 3) both are not fixed to any parts (i.e., both are free vertices). In cases 1 and 2, the resulting coarse vertex is fixed to the part in which either of its constituent vertices was fixed. In case 3, the resulting coarse vertex remains free.

5.2 Coarse Partitioning Phase

In the coarse partitioning phase, we construct an initial partition of the coarsest hypergraph available. If the coarsest hypergraph is small enough, we replicate it on every processor. Each processor then runs a randomized greedy hypergraph growing algorithm to compute a different partition into k parts, and the partition with the lowest cost is selected. If the coarsest hypergraph is not small enough, each processor contributes to computation of an initial partition using a localized version of the greedy hypergraph algorithm. In either case, we maintain the fixed part constraints by assigning fixed coarse vertices to their respective parts.

5.3 Refinement Phase

In the refinement phase, we project our coarse partition to finer hypergraphs and improve it using a local optimization method. Our code is based on a localized version of the successful Fiduccia–Mattheyses [26] method, as described in [22]. The algorithm performs multiple pass-pairs and in each pass, each free vertex is considered to move to another part to reduce the cut metric. We enforce the fixed vertex constraints simply; we do not allow fixed vertices to be moved out of their fixed part.

5.4 Handling Fixed Vertices in Recursive Bisection

Zoltan uses recursive bisection (repeated subdivision of parts into two parts) to obtain a k-way partition. This recursive bisection approach can be extended easily to accommodate fixed vertices. For example, in the first bisection of recursive bisection, the fixed vertex information of each vertex can be updated so that vertices that are originally fixed to parts $1 \leq p \leq k/2$ are fixed to part 1, and vertices originally fixed to parts $k/2 < p \leq k$ are fixed to part 2. Then, the multilevel partitioning algorithm with fixed vertices described above can be executed without any modifications. This scheme is applied recursively in each bisection.

6 Experimental Results

In this section we present detailed comparisons of various graph- and hypergraph-based repartitioning approaches using dynamic datasets that are synthetically

generated using real application base cases, as well as real dynamic data from applications in data mining and adaptive mesh refinement simulations. For most experiments, we select square, structurally symmetric data to allow comparisons between graph and hypergraph methods; the data mining application, however, demonstrates the greater applicability of hypergraph methods to non-symmetric, rectangular data — in this case, term-by-document matrices.

6.1 Repartitioning Approaches

We consider three aspects of repartitioning methods and compare different options provided by various algorithms as well as the algorithms themselves.

- *Repartitioning technique:* Following the discussion in Section 2, we classify repartitioning techniques into three categories: scratch-remap, incremental and repartitioning. Repartitioning approaches have been shown to outperform diffusive methods in [50]; therefore, we consider only refinement approaches within the incremental techniques category.
- *Cost model:* Hypergraph models accurately represent communication and migration costs for multi-way interactions, while graph models represent approximate costs. We do not consider coordinate-based models here, since they are not general (e.g., they cannot be applied to data without coordinates) and they do not model communication and migration costs explicitly.
- *Optimization method:* We also make a distinction between *single-level* versus *multi-level* partitioners and compare their performance.

Partitioner	Repartitioning technique	Cost model	Optimization method	Software
Z-repart	repartitioning	hypergraph	multilevel	Zoltan
Z-SL-repart	repartitioning	hypergraph	single level	Zoltan
Z-scratch	scratch-remap	hypergraph	multilevel	Zoltan
Z-SL-refine	iterative	hypergraph	single level	Zoltan
M-repart	repartitioning	graph	multilevel	ParMETIS
M-scratch	scratch-remap	graph	multilevel	ParMETIS

Table 2
Properties of the partitioners used in the experimental evaluation.

We compare six different partitioners given in Table 2 that collectively cover all options with respect to each of the three aspects considered. In our experiments, we use ParMETIS version 3.1 [39] for graph partitioning and Zoltan version 3.0 [7,14,22] for hypergraph partitioning. For the scratch methods, we used a maximal matching heuristic in Zoltan to map part numbers be-

tween old and new partitions to reduce migration cost. We do not expect the partitioning-from-scratch methods to be competitive for dynamic problems, but include them as a useful baseline.

6.2 Dynamically Perturbed Data Experiments

To perform experiments on large numbers of processors, we collected static data from three real applications and dynamically perturbed the data over a series of time-steps. The properties of the application datasets are shown in Table 3. These datasets provide a range of sparsity and regularity representative of different applications.

Two different methods are used to dynamically perturb the data in the experiments. The first method introduces biased random perturbations that change the structure of the data. In this method, a certain fraction of vertices in the original data is randomly deleted along with the incident edges. At each repartitioning iteration, this operation is repeated independently from previous iterations; hence, a different subset of vertices from the original data is deleted. This operation simulates dynamically changing data that can both lose and gain vertices and edges. The results presented in this section correspond to the case where half of the parts lose or gain 25% of the total number of vertices at each iteration. We tested several other configurations by varying the fraction of vertices lost or gained. The results we obtained in these experiments were similar to the ones presented in this section.

The second method simulates adaptive mesh refinement. Starting with the initial data, a certain fraction of the parts at each iteration is randomly selected. Then, the sub-domain corresponding to the selected parts performs a simulated mesh refinement, where the weight and size of each vertex are increased by a constant factor. In the experiments in this section, 10% of the parts are selected at each iteration and the weight and size of each vertex in these parts are randomly increased to between 1.5 and 7.5 of their original value. Similar to the previous method, we tested several other configurations by varying the factor that scales the size and weight of vertices. The results obtained in these experiments were similar to the ones presented here.

We performed the dynamically perturbed data experiments on Sandia's Thunderbird cluster. Each node of Thunderbird has dual 3.6GHz Intel EM64T processors with 6GB of RAM. The nodes are interconnected with an Infiniband network. We use Intel v10.0 compilers with -O0 optimization flag and OpenMPI v1.2.4. All experiments were run on 64, 256, and 1024 processors.

(Note to Reviewers: Several ParMETIS experiments failed on Thunderbird under this configuration. We suspect the problem lies not in ParMETIS but in

Table 3
Properties of the test datasets; $|V|$ and $|E|$ are the numbers of vertices and graph edges, respectively.

Name	$\|V\|$	$\|E\|$	vertex degree			Application Area
			min	max	avg	
xyce680s	682,712	823,232	1	209	2.4	VLSI design
slac6M	5,955,366	11,766,788	2	4	4.0	Finite element mesh
cage15	5,154,859	47,022,346	2	46	18.2	DNA electrophoresis

Thunderbird's compiler/configuration. Before final publication, we will work with the Thunderbird system administrators to obtain the missing ParMETIS results.)

In Figures 2 through 7, the parameter α, the number of iterations in an epoch, is varied from 10 to 1000, and total cost (3) is reported for 64, 256 and 1024 processors (parts). Each result is averaged over a sequence of 20 trials for each experiment. For each configuration, there are six bars representing total cost for Z-repart, Z-SL-repart, Z-scratch, Z-SL-refine, M-repart, and M-scratch, from left to right respectively. The total cost in each bar is normalized by the total cost of Z-repart in the respective configuration and consists of two components: application communication costs (scaled by α) on the bottom (darker shade) and migration costs on the top (lighter shade). Results are shown for both the dynamic structure perturbations and the dynamic weight perturbations.

The results indicate that our new hypergraph repartitioning method Z-repart performs better than M-repart in terms of minimizing the total cost in the majority of the test cases. This can be explained by the fact that the migration cost minimization objective is completely integrated into the multilevel scheme rather than handled in only the refinement phase. Therefore, Z-repart provides a more accurate trade-off between communication and migration costs than M-repart to minimize the total cost. This is more clearly seen for small and moderate α values where these two costs are comparable. On the other hand, for large α values, the migration cost is less important relative to communication cost, and the problem essentially reduces to minimizing the communication cost alone. Therefore, in such cases, Z-repart and M-repart behave similarly to partitioners using scratch methods.

Similar arguments hold when comparing Z-repart against scratch-remap repartitioning methods. Since minimization of migration cost is ignored in Z-scratch and M-scratch, migration cost gets extremely large and dominates the total cost as α gets smaller. Total cost with Z-scratch and M-scratch is comparable to Z-repart only when α is greater than 100, where communication cost starts to dominate. Z-repart still performs as well as the scratch methods in this

range to minimize the total cost.

As the number of parts (processors) increases, there is a noticeable increase in migration cost relative to communication cost when using M-repart. On the other hand, with Z-repart the increase in migration cost is smaller at the expense of a modest increase in communication cost. This shows that Z-repart achieves a better balance between communication and migration costs, and consequently results in a smaller total cost than M-repart with increasing number of parts. This suggests that in addition to minimization of the total cost, Z-repart is superior to M-repart in terms of scalability of the solution quality.

Z-refine and Z-SL-repart attempt to minimize communication volume with relatively fewer vertex movements due to the constrained initial partition. Therefore, the communication cost of these methods is higher than other partitioners, resulting in a relatively higher total cost for large α values. On the other hand, both methods produce lower migration costs compared to scratch methods for small α values. Both Z-refine and Z-SL-repart, however, are outperformed by Z-repart in all of our test cases. Indeed, the benefit of multi-level methods is clearly shown in the comparisons of Z-repart and Z-SL-repart.

Run times of the tested partitioners normalized by that of Z-repart for the perturbed structure and weight experiments are given in Figures 8–13. We observed two different run time profiles in our test cases. The first one is shown in Figures 8 and 9 for the xyce680s dataset, where multilevel hypergraph-based methods Z-repart and Z-scratch are at least as fast as their graph-based counterparts M-repart and M-scratch. In some cases (e.g. perturbed data structure, running on 64 processors) hypergraph-based approaches are up to five times faster than graph-based approaches. Z-SL-repart is significantly faster than most other methods in this dataset with relatively low total cost; therefore, it becomes a viable option for applications that require a very fast repartitioner for small α values. The second run time profile is observed in Figures 10-13 for the slac6M and cage15 datasets. The results show that hypergraph-based repartitioning can be up ten times slower than graph-based approaches. As these results show, there is no clear conclusion on which approach is faster. Furthermore, since the application run time is often far greater than the partitioning time, this enhancement may not be important in practice.

6.3 Adaptive Mesh Refinement Experiments

Adaptive mesh refinement is a decades-old technique used in finite element analysis to obtain desired solution resolution with an optimal number of degrees of freedom. At each time step, the finite element code computes both

the solution and an estimate of the error in the solution. Elements in regions with high error are subdivided into many smaller elements, while elements in regions with low error are coalesced into fewer large elements. Subsequent solves, then, obtain greater resolution in the high-error regions without adding unnecessary degrees of freedom in low-error regions.

In parallel simulations with adaptive mesh refinement, the refinement and coalescing of elements causes significant load imbalance. As processors add or remove elements due to refinement, their workloads change. Dynamic load balancing has played an important role in enabling parallel adaptive mesh refinement simulations, redistributing work to accommodate evolving meshes; see, e.g., [4,18,44,23,27,59,49,42]. Coordinate- and graph-based methods have been used with great success, due to mesh data's relatively regular structure and low vertex degrees. In these experiments, we compare our repartitioning hypergraph model to commonly used graph-based repartitioners.

Our adaptive mesh data is a series of 109 hexahedral meshes from the ALEGRA shock physics explicit finite element code [8]. The series of meshes represents time-steps of the simulation; the mesh refinement tracks the shock moving across the domain and its reflections. (Figure 14 shows the mesh at the time-steps 0, 54, and 108, respectively.) The smallest mesh (time-step 0) has 132,209 nodes and 103,100 elements; the largest (time-step 108) has 1,380,266 nodes and 1,247,000 elements.

We represent mesh nodes with vertices of the graph and hypergraph models, and create a graph edge between nodes that share a mesh element. These graph edges are used directly in the graph methods, and combined into a single hyperedge per node in the hypergraph methods. The smallest mesh has 1,527,841 graph edges; the largest has 17,391,840 graph edges.

In our experiments, we performed an initial partitioning of the initial mesh (time-step 0). Then at each time-step $T > 0$, we assign each node of mesh T to the same part as its closest node in mesh $T-1$ — "closeness" is measured by two nodes' proximity along a space-filling curve through the nodes of both meshes — and repartition mesh T using one of the methods in Table 2.

We ran experiments over 109 meshes with $\alpha = 100$ on 16, 32, and 64 processors of Sandia's Odin cluster. Each node of Odin has two AMD Opteron 2.2GHz processors and 4 GB of RAM. Nodes are connected with a Myrinet network. We used MPICH v1.2.7 and gcc v3.4.3.

Total cost (3) and run times for each method are shown in Figures 15 and 16, respectively. The repartitioning hypergraph method Z-repart produced lower total cost than all other methods, although the graph repartitioning method M-repart was competitive on small numbers of processors. The performance gap between hypergraph and graph methods increases in favor of hypergraph

model with increasing number of processors. Execution time for Z-repart was greater than M-repart, indicating the need for faster heuristics in the hypergraph implementation for applications with relatively low and homogeneous connectivity.

6.4 Term-by-Document Experiments

Our last example is from text analysis and retrieval. Latent Semantic Analysis (LSA) [21] is a popular technique for analysis of large document collections. Given a set of documents, a user can search for specific terms, documents relevant to a specific topic, or find related documents. The method is based on reduced approximations to the term-by-document matrix, where rows represent terms and columns correspond to documents. There is a nonzero matrix entry in position (i, j) if and only if document j contains term i. Note that such matrices are rectangular and non-symmetric, so graph models do not apply. The computationally intensive part of LSA is to compute a truncated singular value decomposition (SVD) of the type $A \approx A_k = U_k \Sigma_k V_k$, where Σ_k is diagonal, and k is the rank of the approximation. It is known that $100 \leq k \leq 300$ is a good range for retrieval. An iterative method based on sparse matrix-vector multiplication by A is used to compute the SVD.

We focus on a parallel strategy for LSA with a dynamic document collection where documents are added over time. (This is motivated by a project at Sandia led by Danny Dunlavy using the LSALIB software.) Our goal is to find an efficient parallel distribution of documents to processors, to ensure load balance and reduce communication. As an example, we use a large term-by-document matrix corresponding to the Citeseer database up to 2004. Each month, a new set of documents are added, and the SVD must be recomputed. The number of documents added will vary from month to month. By default, documents are assigned to processors in a cyclic fashion. There is a cost associated with moving documents between processors. We seek load balance with respect to the number of nonzeros in the term-by-document matrix, which corresponds to memory usage.

We started with all the documents that existed on Jan. 1, 1994, and ran a ten year simulation (120 months). The full matrix has about 700,000 documents and 57 million nonzeros. In this application, α should be in the range $100-600$; we tested $\alpha = 100$. Experiments were run on Sandia's Odin cluster using 16, 32, and 64 processors; results are presented in Figures 17 and 18. We compare only hypergraph-based approaches since graph-based methods (ParMETIS) do not apply directly. We see from Figure 17 that the multilevel methods are clearly performing better than the single-level methods, in terms of solution quality. Since in this application, migration cost becomes very small compared

to application communication cost, there is only a little difference between repartitioning and scratch-remap.

7 Conclusion

In this paper, we presented a new approach to dynamic load balancing based on a single hypergraph model that incorporates both communication volume in the application and data migration cost. Detailed comparison of graph- and hypergraph-based repartitioning using datasets from a range of application areas showed that hypergraph-based repartitioning produces partitions with similar or lower cost than the graph-based repartitioning. The full benefit of hypergraph partitioning is realized on non-symmetric and non-square problems that cannot be represented easily with graph models [16,22].

Our hypergraph-based repartitioning model uses a single user-defined parameter α to control trade-off between communication cost and migration cost. Experiments show that the approach works particularly well when migration cost is more important, and does not degrade quality when communication cost is more important. Therefore, we recommend the presented approach as a universal method for dynamic load balancing. The best choice of α will depend on the application, and can be estimated easily. Reasonable values are in the range $1 - 1000$.

The experiments showed that the hypergraph-based repartitioning approach implemented in Zoltan is scalable. However, in many cases it required more time than its graph-based counterpart due to the greater richness of the hypergraph model. We will further investigate exploiting locality given by the data distribution in order to improve the execution time of the hypergraph-based repartitioning implementation. However, since the application run time is often far greater than the partitioning time, this enhancement may not be important in practice.

Acknowledgments

We thank Danny Dunlavy for providing the Citeseer data; Richard Drake and Johan Steensland for the adaptive mesh refinement data; and Rich Lee, Mark Shephard, and Xiaojuan Luo for the slac6M data.

References

[1] C. J. Alpert, A. E. Caldwell, A. B. Kahng, and I. L. Markov. Hypergraph partitioning with fixed vertices [vlsi cad]. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 19(2):267–272, 2000.

[2] C. Aykanat, B. B. Cambazoglu, F. Findik, and T. Kurc. Adaptive decomposition and remapping algorithms for object-space-parallel direct volume rendering of unstructured grids. *J. Parallel Distrib. Comput.*, 67(1):77–99, Jan 2007.

[3] C. Aykanat, B. B. Cambazoglu, and B. Uçar. Multi-level direct k-way hypergraph partitioning with multiple constraints and fixed vertices. *J. Parallel Distrib. Comput.*, 2008. In Press, Available online 7 October 2007.

[4] M. J. Berger and S. H. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Trans. Computers*, C-36(5):570–580, 1987.

[5] S. H. Bokhari. On the mapping problem. *IEEE Trans. Computers*, 30(3):207–214, 1981.

[6] S. H. Bokhari. *Assignment Problems in Parallel and Distributed Computing*. Kluwer Academic Publishers, Norwell, MA, USA, 1987.

[7] E. Boman, K. Devine, R. Heaphy, B. Hendrickson, V. Leung, L. A. Riesen, C. Vaughan, U. Catalyurek, D. Bozdag, W. Mitchell, and J. Teresco. *Zoltan 3.0: Parallel Partitioning, Load Balancing, and Data-Management Services; User's Guide*. Sandia National Laboratories, Albuquerque, NM, 2007. Tech. Report SAND2007-4748W `http://www.cs.sandia.gov/Zoltan/ug_html/ug.html`.

[8] E. A. Boucheron, K. H. Brown, K. G. Budge, S. P. Burns, D. E. Carroll, S. K. Carroll, M. A. Christon, R. R. Drake, C. G. Garasi, T. A. Haill, J. S. Peery, S. V. Petney, J. Robbins, A. C. Robinson, R. Summers, T. E. Voth, and M. K. Wong. *ALEGRA: User Input and Physics Descriptions Version 4.2*. Sandia National Laboratories, Albuquerque, NM, 2002. Tech. Report SAND2002-2775.

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

[10] T. Bultan and C. Aykanat. A new mapping heuristic based on mean field annealing. *J. Parallel Distrib. Comput.*, 16:292–305, 1992.

[11] B. B. Cambazoglu and C. Aykanat. Hypergraph-partitioning-based remapping models for image-space-parallel direct volume rendering of unstructured grids. *IEEE Trans. Parallel Distrib. Syst.*, 18(1):3–16, Jan 2007.

[12] B. B. Cambazoglu and C. Aykanat. Remapping models for scientific computing via graph and hypergraph partitioning. In *Proc. of the SIAM Workshop on Combinatorial Scientific Computing (CSC07)*, Costa Mesa, CA, 2007.

[13] W. Camp, S. J. Plimpton, B. Hendrickson, and R. W. Leland. Massively parallel methods for engineering and science problems. *Communication of ACM*, 37(4):31–41, April 1994.

[14] U. V. Catalyurek, E. G. Boman, K. D. Devine, D. Bozdağ, R. Heaphy, and L. A. Fisk. Hypergraph-based dynamic load balancing for adaptive scientific computations. In *Proc. of 21st International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2007.

[15] U. V. Çatalyürek and C. Aykanat. A hypergraph model for mapping repeated sparse matrix-vector product computations onto multicomputers. In *Proc. of International Conference on High Performance Computing*, Dec. 1995.

[16] U. V. Çatalyürek and C. Aykanat. Hypergraph-partitioning based decomposition for parallel sparse-matrix vector multiplication. *IEEE Trans. Parallel Distrib. Syst.*, 10(7):673–693, 1999.

[17] U. V. Çatalyürek and C. Aykanat. *PaToH: A Multilevel Hypergraph Partitioning Tool, Version 3.0.* Bilkent University, Department of Computer Engineering, Ankara, 06533 Turkey. PaToH is available at `http://bmi.osu.edu/~umit/software.htm`, 1999.

[18] N. Chrisochoides. Multithreaded model for dynamic load balancing parallel adaptive PDE computations. ICASE Report 95-83, ICASE, NASA Langley Research Center, Hampton, VA 23681-0001, Dec. 1995.

[19] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *J. Parallel Distrib. Comput.*, 7:279–301, 1989.

[20] H. deCougny, K. Devine, J. Flaherty, R. Loy, C. Ozturan, and M. Shephard. Load balancing for the parallel adaptive solution of partial differential equations. *Appl. Numer. Math.*, 16:157–182, 1994.

[21] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman. Indexing by latent semantic analysis. *J. of the American Society for Information Science*, 41(6):391–407, 1990.

[22] K. Devine, E. Boman, R. Heaphy, R. Bisseling, and U. Catalyurek. Parallel hypergraph partitioning for scientific computing. In *Proc. of 20th International Parallel and Distributed Processing Symposium (IPDPS'06)*. IEEE, 2006.

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

[24] P. Diniz, S. Plimpton, B. Hendrickson, and R. Leland. Parallel algorithms for dynamically partitioning unstructured grids. In *Proc. 7th SIAM Conf. Parallel Processing for Scientific Computing*, pages 615–620. SIAM, 1995.

[25] J. Faik, L. G. Gervasio, J. E. Flaherty, J. Chang, J. D. Teresco, E. G. Boman, and K. D. Devine. A model for resource-aware load balancing on heterogeneous clusters. Technical Report CS-03-03, Williams College Dept. Computer Science, 2003. `http://www.cs.williams.edu/drum/`.

[26] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proc. 19th IEEE Design Automation Conf.*, pages 175–181, 1982.

[27] 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(2):139–152, 1998.

[28] M. R. Garey and D. S. Johnson. *Computers and Intractability.* W.H. Freeman and Co., New York, New York, 1979.

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

[30] B. Hendrickson and R. Leland. *The Chaco user's guide, version 2.0.* Sandia National Laboratories, Alburquerque, NM, 87185, 1995.

[31] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. In *Proc. Supercomputing '95.* ACM, December 1995.

[32] B. Hendrickson, R. Leland, and R. Van Driessche. Skewed graph partitioning. In *Proc. Eighth SIAM Conf. Parallel Processing for Scientific Computation*, March 1997.

[33] Y. F. Hu and R. J. Blake. An optimal dynamic load balancing algorithm. Technical Report DL-P-95-011, Daresbury Laboratory, Warrington, UK, 1995.

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

[35] M. A. Iqbal and S. H. Bokhari. Efficient algorithms for a class of partitioning problems. *IEEE Trans. Parallel Distrib. Syst.*, 6(2):170–175, 1995.

[36] G. Karypis and V. Kumar. *MeTiS A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices Version 4.0.* University of Minnesota, Department of Comp. Sci. and Eng., Army HPC Research Center, Minneapolis, 1998.

[37] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1), 1999.

[38] G. Karypis, V. Kumar, R. Aggarwal, and S. Shekhar. *hMeTiS A Hypergraph Partitioning Package Version 1.0.1.* University of Minnesota, Department of Comp. Sci. and Eng., Army HPC Research Center, Minneapolis, 1998.

[39] G. Karypis, K. Schloegel, and V. Kumar. Parmetis: Parallel graph partitioning and sparse matrix ordering library, version 3.1. Technical report, Dept. Computer Science, University of Minnesota, 2003. `http://www-users.cs.umn.edu/~karypis/metis/parmetis/download.html`.

[40] H. Kasahara and S. Narita. Practical multiprocessor scheduling algorithms for efficient parallel processing. *IEEE Transactions on Computers*, C-33(11):1023–1029, Nov 1984.

[41] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout.* Willey–Teubner, Chichester, U.K., 1990.

[42] L. Oliker and R. Biswas. PLUM: Parallel load balancing for adaptive unstructured mesh es. *J. Parallel Distrib. Comput.*, 51(2):150–177, 1998.

[43] C.-W. Ou and S. Ranka. Parallel incremental graph partitioning using linear programming. Technical report, Syracuse University, Syracuse, NY, 1992.

[44] A. Patra and J. T. Oden. Problem decomposition for adaptive *hp* finite element methods. *J. Computing Systems in Engg.*, 6(2), 1995.

[45] J. R. Pilkington and S. B. Baden. Partitioning with spacefilling curves. CSE Technical Report CS94–349, Dept. Computer Science and Engineering, University of California, San Diego, CA, 1994.

[46] P. Sadayappan and F. Ercal. Nearest-neigbour mapping of finite element graphs onto processor meshes. *IEEE Transactions on Computers*, 36(12):1408–1424, 1987.

[47] P. Sadayappan, F.Ercal, and J. Ramanujam. Cluster partitioning aproaches to mapping parallel programs onto hypercube. *Parallel Computing*, 13:1–16, 1990.

[48] K. Schloegel, G. Karypis, and V. Kumar. Multilevel diffusion algorithms for repartitioning of adaptive meshes. *J. Parallel Distrib. Comput.*, 47(2):109–124, 1997.

[49] K. Schloegel, G. Karypis, and V. Kumar. Multilevel diffusion algorithms for repartitioning of adaptive meshes. *J. Parallel Distrib. Comput.*, 47(2):109–124, 1997. Also available on WWW at URL http://www.cs.umn.edu/~karypis.

[50] K. Schloegel, G. Karypis, and V. Kumar. A unified algorithm for load-balancing adaptive scientific simulations. In *Proc. Supercomputing*, Dallas, 2000.

[51] K. Schloegel, G. Karypis, and V. Kumar. Wavefront diffusion and LMSR: Algorithms for dynamic repartitioning of adaptive meshes. *IEEE Trans. Parallel Distrib. Syst.*, 12(5):451–466, 2001.

[52] S. Sinha and M. Parashar. Adaptive system partitioning of AMR applications on heterogeneous clusters. *Cluster Computing*, 5(4):343–352, October 2002.

[53] 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.*, 184:269–285, 2000.

[54] 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)*, volume 3280 of *LNCS*, pages 789–800. Springer, 2004.

[55] R. Van Driessche and D. Roose. Dynamic load balancing with a spectral bisection algorithm for the constrained graph partitioning problem. In *High-Performance Computing and Networking*, number 919 in Lecture Notes in Computer Science, pages 392–397. Springer, 1995. Proc. Int'l Conf. and Exhibition, Milan, Italy, May 1995.

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

[57] C. Walshaw and M. Cross. Multilevel mesh partitioning for heterogeneous communication networks. *Future Generation Computer Systems*, 17:601–623, 2001.

[58] C. Walshaw, M. Cross, and M. Everett. Parallel dynamic graph-partitioning for adaptive unstructured meshes. *J. Par. Dist. Comput.*, 47(2):102–108, 1997.

[59] C. Walshaw, M. Cross, and M. G. Everett. Parallel dynamic graph partitioning for adaptive unstructured meshes. *J. Parallel Distrib. Comput.*, 47(2):102–108, 1997.

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

[61] M. Willebeek-LeMair and A. P. Reeves. Strategies for dynamic load balancing on highly parallel computers. *IEEE Trans. Parallel Distrib. Syst.*, 4(9):979–993, 1993.

[62] R. Williams. Performance of dynamic load balancing algorithms for unstructured mesh calculations. *Concurrency*, 3:457–481, October 1991.

[63] Zoltan: Data management services for parallel applications. `http://www.cs.sandia.gov/Zoltan/`.

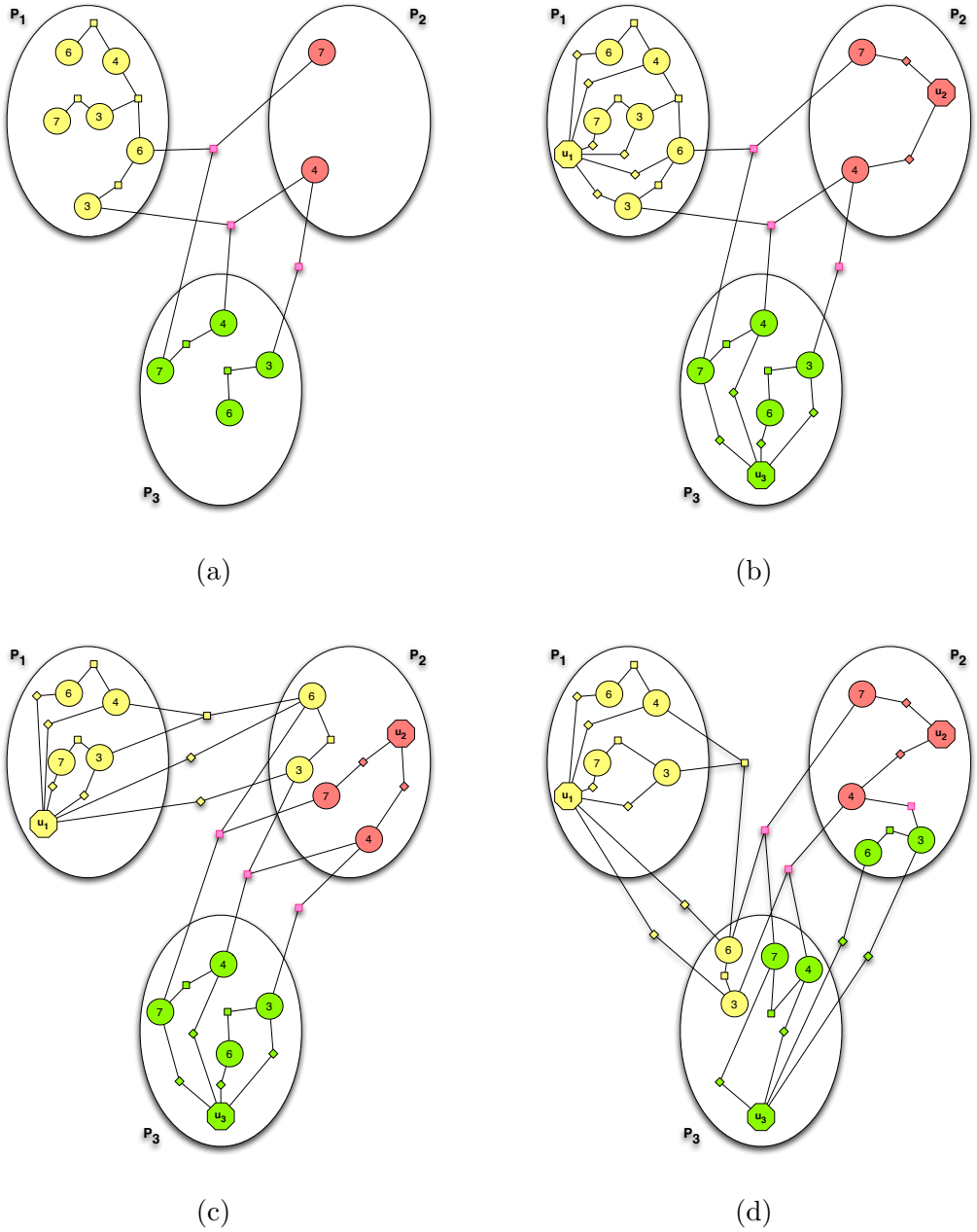


Fig. 1. (a): A sample computational hypergraph representation at the beginning of epoch j. Nets are depicted as squares and vertices are depicted as circles. The numbers inside the circles are the computational loads of each vertex. (b): Repartitioning hypergraph for epoch j; for simplicity in the presentation, migration nets are depicted as diamonds and part vertices are depicted as octagons. (c) and (d): Two sample solutions with $b_{comm} = 4, b_{mig} = 2$, and $b_{comm} = 3, b_{mig} = 4$, respectively, under the assumption that the migration cost of each computation vertex and the application communication cost per net are one (i.e., each net's cost is one).

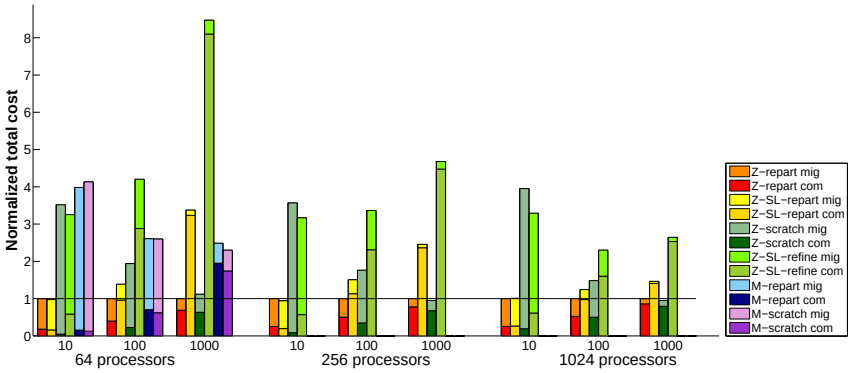


Fig. 2. Normalized total cost for xyce680s with perturbed data structure with $\alpha = 10, 100, 1000$ on 64, 256, and 1024 processors; colors indicate which repartitioning method was used.

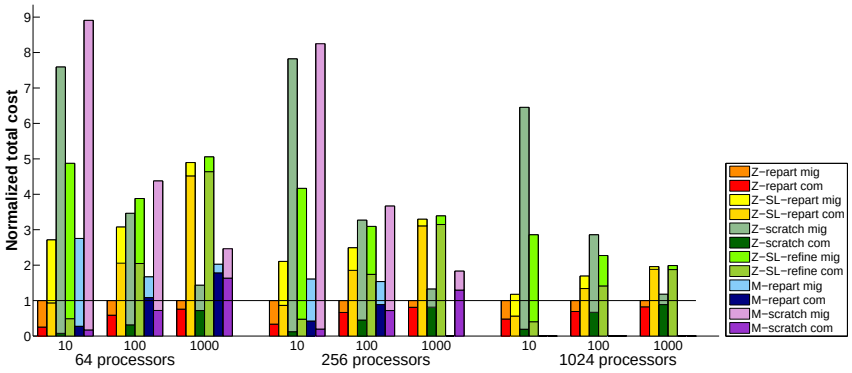


Fig. 3. Normalized total cost for xyce680s with perturbed weights with $\alpha = 10, 100, 1000$ on 64, 256, and 1024 processors; colors indicate which repartitioning method was used.

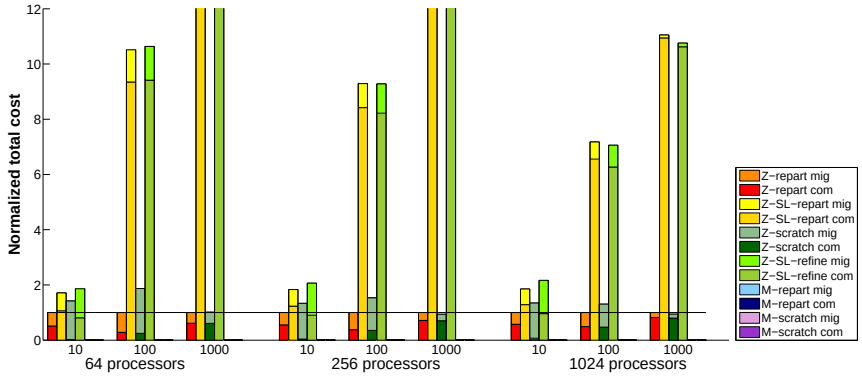


Fig. 4. Normalized total cost for slac6M with perturbed data structure with $\alpha = 10, 100, 1000$ on 64, 256, and 1024 processors; colors indicate which repartitioning method was used. Z-SL-repart and Z-SL-refine bars are truncated to enhance readability.

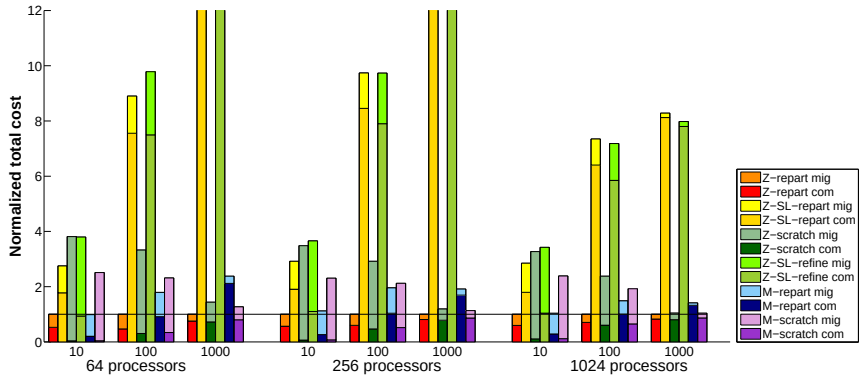


Fig. 5. Normalized total cost for slac6M with perturbed weights with $\alpha = 10, 100, 1000$ on 64, 256, and 1024 processors; colors indicate which repartitioning method was used. Z-SL-repart and Z-SL-refine bars are truncated to enhance readability.

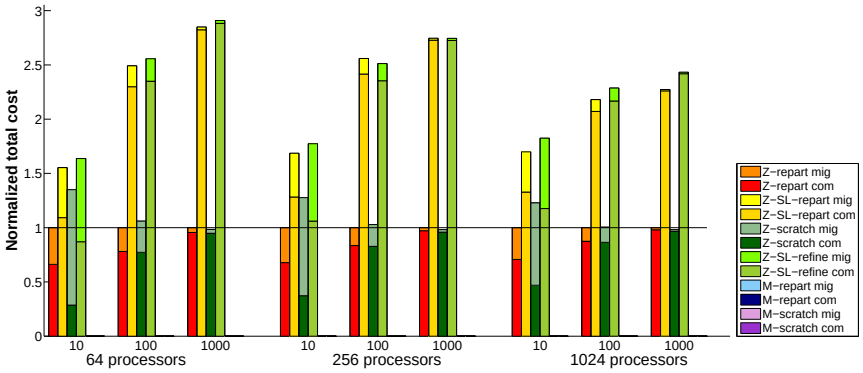


Fig. 6. Normalized total cost for cage15 with perturbed data structure with $\alpha = 10, 100, 1000$ on 64, 256, and 1024 processors; colors indicate which repartitioning method was used.

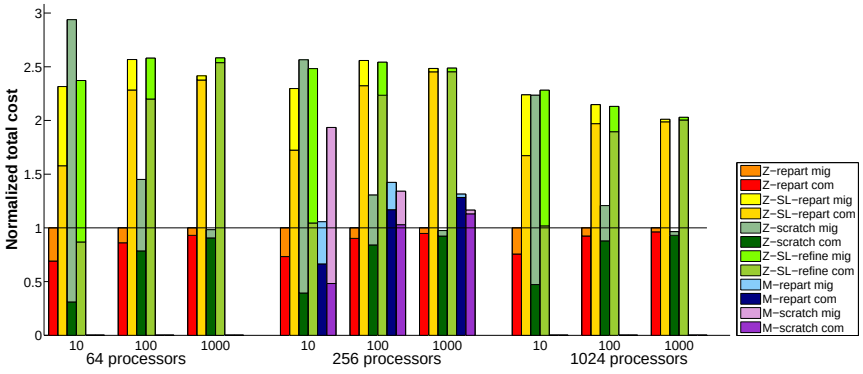


Fig. 7. Normalized total cost for cage15 with perturbed weights with $\alpha = 10, 100, 1000$ on 64, 256, and 1024 processors; colors indicate which repartitioning method was used.

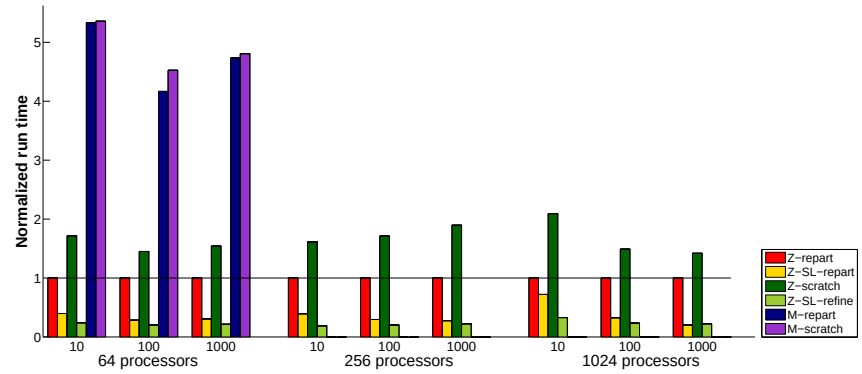


Fig. 8. Normalized run time with perturbed data structure for xyce680s with $\alpha = 10, 100, 1000$ on 64, 256, and 1024 processors; colors indicate which repartitioning method was used.

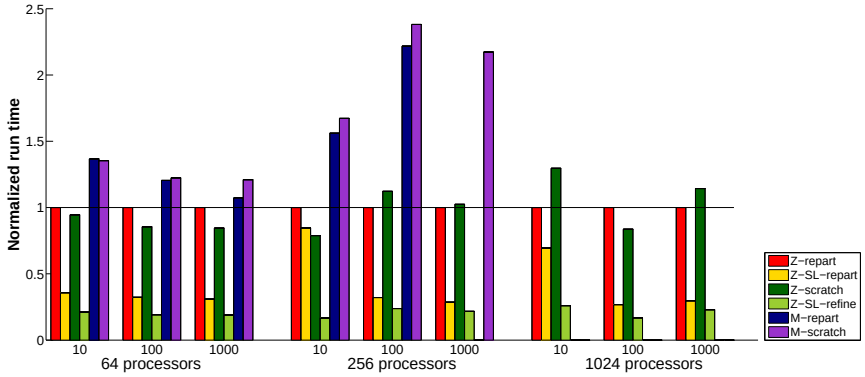


Fig. 9. Normalized run time with perturbed weights for xyce680s with $\alpha = 10, 100, 1000$ on 64, 256, and 1024 processors; colors indicate which repartitioning method was used.

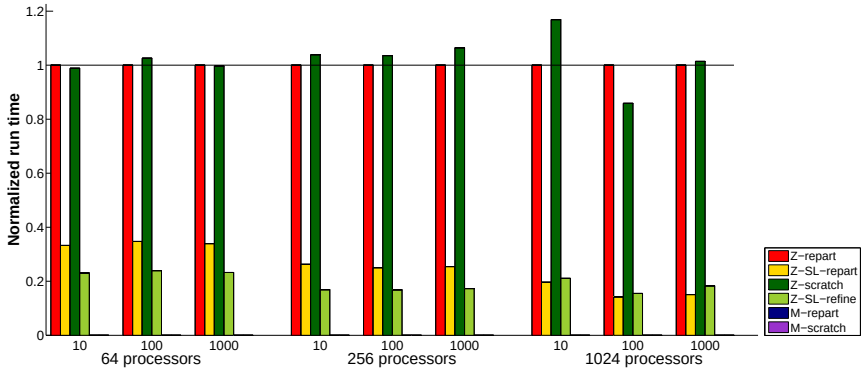


Fig. 10. Normalized run time with perturbed data structure for slac6M with $\alpha = 10, 100, 1000$ on 64, 256, and 1024 processors; colors indicate which repartitioning method was used.

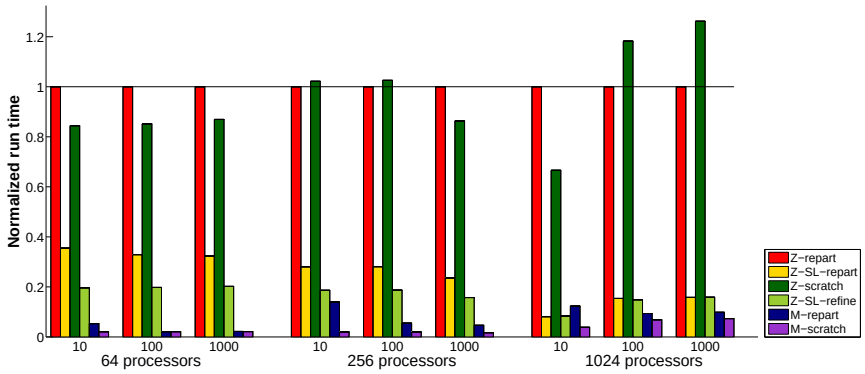


Fig. 11. Normalized run time with perturbed weights for slac6M with $\alpha = 10, 100, 1000$ on 64, 256, and 1024 processors; colors indicate which repartitioning method was used.

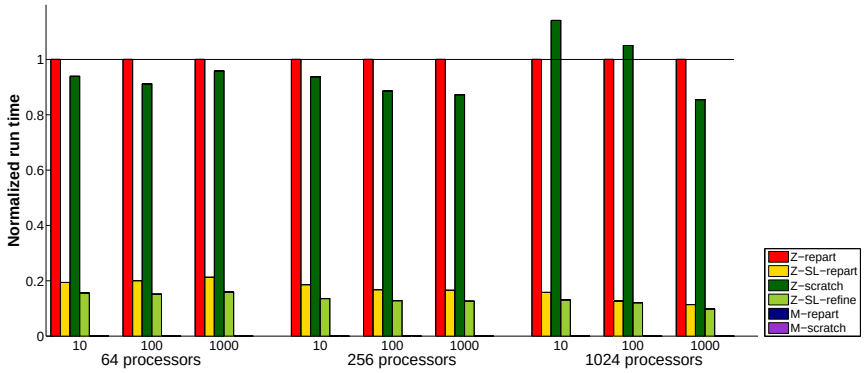


Fig. 12. Normalized run time with perturbed data structure for cage15 with $\alpha = 10, 100, 1000$ on 64, 256, and 1024 processors; colors indicate which repartitioning method was used.

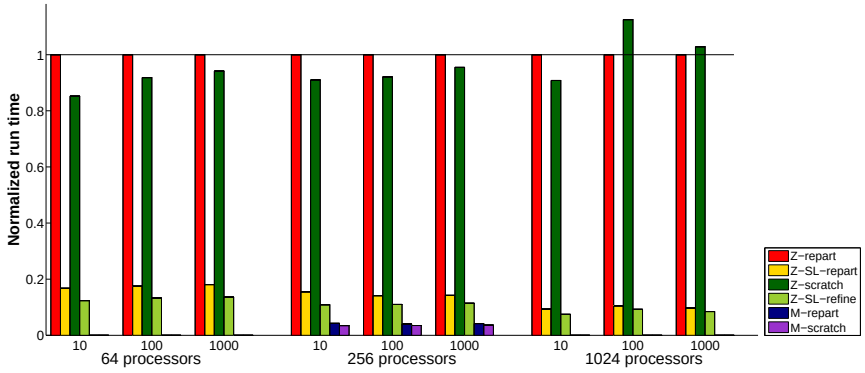
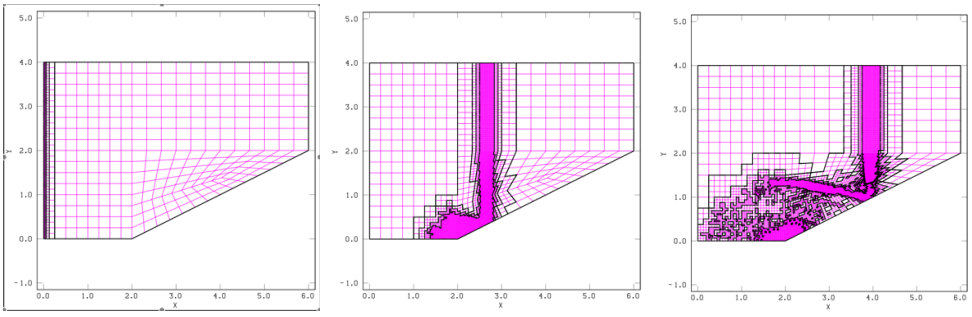


Fig. 13. Normalized run time with perturbed weights for cage15 with $\alpha = 10, 100, 1000$ on 64, 256, and 1024 processors; colors indicate which repartitioning method was used.



(a) (b) (c)

Fig. 14. Hexahedral finite element meshes with adaptive mesh refinement at time-steps 0, 54, and 108, respectively.

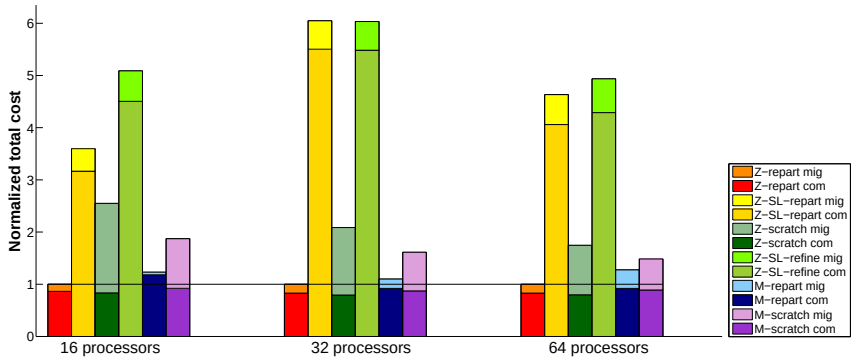


Fig. 15. Normalized total cost for adaptive mesh refinement experiments with $\alpha = 100$ on 16, 32 and 64 processors; colors indicate which repartitioning method was used.

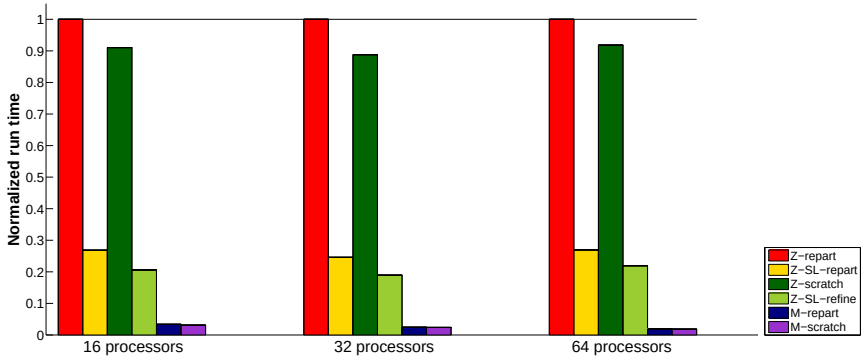


Fig. 16. Normalized run time for adaptive mesh refinement experiments with $\alpha = 100$ on 16, 32, and 64 processors; colors indicate which repartitioning method was used.

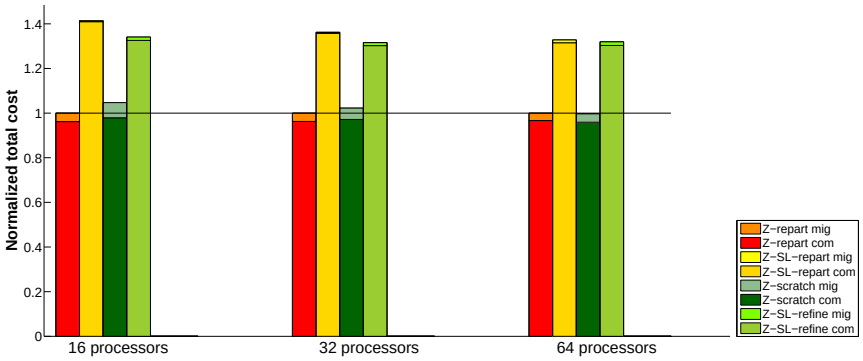


Fig. 17. Normalized total cost for term-by-document with $\alpha = 100$ on 16, 32 and 64 processors; colors indicate which repartitioning method was used.

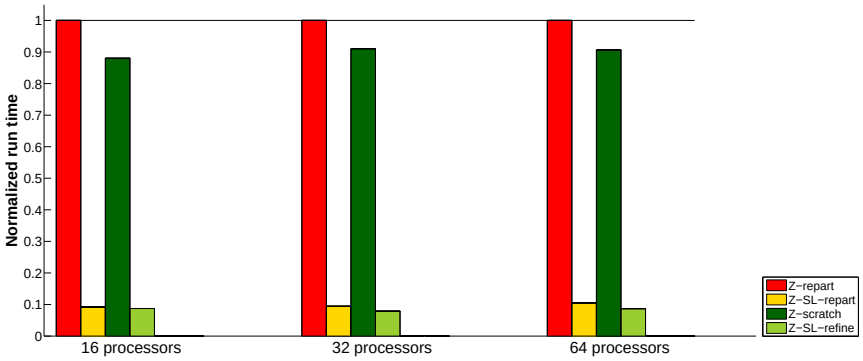


Fig. 18. Normalized run time for term-by-document with $\alpha = 100$ on 16, 32, and 64 processors; colors indicate which repartitioning method was used.