Hypergraph-based Dynamic Load Balancing for Adaptive Scientific Computations

Umit V. Catalyurek[†], Erik G. Boman[*], Karen D. Devine[*], Doruk Bozdağ[†], Robert Heaphy[*], and Lee Ann Riesen[*]

[†]Ohio State University
Dept. of Biomedical Informatics
Dept. of Electrical & Computer Eng.
Columbus, OH 43210, USA
{umit,bozdagd}@bmi.osu.edu

[*]Sandia National Laboratories
Discrete Algorithms and Math. Dept.
Albuquerque, NM 87185-1318, USA
{egboman,kddevin}@sandia.gov
{rheaphy,lafisk}@sandia.gov

Abstract

Adaptive scientific computations require that periodic repartitioning (load balancing) occur dynamically to maintain load balance. Hypergraph partitioning is a successful model for minimizing communication volume in scientific computations, and partitioning software for the static case is widely available. In this paper, we present a new hypergraph model for the dynamic case, where we minimize the sum of communication in the application plus the migration cost to move data, thereby reducing total execution time. The new model can be solved using hypergraph partitioning with fixed vertices. We describe an implementation of a parallel multilevel repartitioning algorithm within the Zoltan load-balancing toolkit, which to our knowledge is the first code for dynamic load balancing based on hypergraph partitioning. Finally, we present experimental results that demonstrate the effectiveness of our approach on a Linux cluster with up to 64 processors. Our new algorithm compares favorably to the widely used ParMETIS partitioning software in terms of quality, and would have reduced total execution time in most of our test cases.

1 Introduction

Dynamic load balancing is an important feature in parallel adaptive computations [7]. Even if the original problem is well balanced, e.g., by using graph or hypergraph partitioning, the computation may become unbalanced over time due to the dynamic changes. A classic example is simulation based on adaptive mesh refinement, in which the computational mesh changes between time steps. The difference is often small, but over time, the cumulative change in the mesh becomes significant. An application may therefore periodically re-balance, that is, move data among processors to improve the load balance. This process is known as dynamic load balancing or repartitioning and is a well studied problem [7, 8, 11, 25, 27, 28, 30, 33, 34]. It has multiple objectives with complicated trade-offs among them:

1. good load balance in the new data distribution;
2. low communication cost within the application (as given by the new distribution);
3. low data migration cost to move data from the old to the new distribution; and
4. short repartitioning time.

Much of the early work in load balancing focused on diffusive methods [7, 17, 26, 33], where overloaded processors give work to neighboring processors that have lower than average loads. A quite different approach is to partition the new problem "from scratch" without accounting for existing partition assignments, and then try to remap partitions to minimize the migration cost [25, 28]. These two strategies have very different properties. Diffusive schemes are fast and have low migration cost, but may incur high communication volume. Scratch-remap schemes give low

[*]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. This work was supported by the NNSA's ASC program and by the DOE Office of Science through the CSCAPES SciDAC institute.

[†]Supported by Sandia contract PO283793 and US Department of Energy under Award Number ED-FC02-06ER25775.

communication volume but are slower and often have high migration cost.

Dynamic load balancing schemes should be designed such that the compromise between these extreme choices can be tweaked by the application developer. In [27], Schloegel et al. introduced a parallel adaptive repartitioning scheme based on the multilevel graph partitioning paradigm. In their work, relative importance of migration time against communication time is set using a user-given parameter, and it is taken into account in the refinement phase of the multilevel scheme. Aykanat et al. [1] proposed a graph-based repartitioning model, *RM model*, where the original computational graph is augmented with new vertices and edges to account for migration cost. Then the graph is repartitioned using graph partitioning with fixed vertices using a serial tool *RM-METIS* that they developed by modifying the graph partitioning tool METIS [19]. Although these approaches attempt to minimize both communication and migration costs, their applicability is limited to problems with symmetric, bi-directional dependencies. In a concurrent work, Cambazoglu and Aykanat [4] have recently proposed a hypergraph-partitioning based model for the adaptive screen partitioning problem in the context of image-space-parallel direct volume rendering of unstructured grids. However, in that application, communication occurs only for data replication (migration); hence, their model accounts only for migration cost.

In this work, our approach is to directly minimize the total execution time. We use the common model [27]

$$t_{tot} = \alpha(t_{comp} + t_{comm}) + t_{mig} + t_{repart},$$

where t_{comp} and t_{comm} denote computation and communication times within the application, respectively, t_{mig} is the data migration time, and t_{repart} is the repartitioning time. The parameter α indicates how many iterations (e.g., time steps in a simulation) the application performs between every load-balance operation. Since the goal of load balancing is to minimize the communication cost while maintaining well-balanced computational loads, we can safely assume that computation will be balanced and hence drop t_{comp} term. The repartitioning time is typically significantly smaller than $\alpha(t_{comp} + t_{comm})$ due to fast state-of-the-art repartitioners, so we also ignore t_{repart}. Thus, the objective of our model is to minimize $\alpha t_{comm} + t_{mig}$.

This work has two main contributions:

- We present a new hypergraph model for repartitioning where we minimize the sum of total communication volume in the application plus the migration cost to move data. Hypergraphs accurately model the actual communication cost and have greater applicability than graph models (e.g., hypergraphs can represent non-symmetric and/or non-square systems) [5]. Furthermore, directly incorporating both the communication and migration costs into a single hypergraph/graph model is more suitable to successful multilevel partitioning than accounting for migration costs only in refinement (see Section 2 for details).

- We present a new parallel repartitioning tool based on hypergraph partitioning with fixed vertices. Although serial hypergraph partitioners with fixed-vertex partitioning exist (PaToH [6]), to the best of our knowledge our tool (Zoltan [2]) is the first parallel hypergraph partitioner with this feature.

The rest of the paper is organized as follows. Section 2 contains some preliminaries about hypergraph partitioning and multilevel partitioning. The details of the proposed hypergraph model for repartitioning are presented in Section 3. Section 4 describes the algorithm for parallel hypergraph partitioning with fixed vertices. Experimental evaluation of the proposed approach is presented in Section 5. Finally, Section 6 contains our conclusions.

2 Preliminaries

The static partitioning problem is often modeled as graph or hypergraph partitioning, where vertices represent the computational load associated with data and the edges (hyperedges) represent data dependencies. The edges (hyperedges) that span more than one partition (so-called cut edges) incur communication cost. We use the hypergraph model because it more accurately reflects communication volume and cost and has greater applicability than graph models [5, 14].

2.1 Hypergraph Partitioning

A hypergraph $H = (V, N)$ is defined by a set of vertices V and a set of nets (hyperedges) N among those vertices, where each net $n_j \in N$ is a non-empty subset of vertices. Non-negative weights (w_i) and costs (c_j) can be assigned to the vertices ($v_i \in V$) and nets ($n_j \in N$) of the hypergraph, respectively. $P = \{V_1, V_2, \ldots, V_k\}$ is called a k-way partition of H if each part is a non-empty, pairwise-disjoint subset of V and the union of all $V_p, p = 1, 2, \ldots, k$, is equal to V. A partition is said to be *balanced* if

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

where part weight $W_p = \sum_{v_i \in V_p} w_i$ is the sum of the vertex weights of part V_p, $W_{avg} = \left(\sum_{v_i \in V} w_i\right)/k$ is the weight of each part under perfect load balance, and ϵ is a predetermined maximum imbalance allowed.

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

There are various ways of defining the cut-size $cuts(H,P)$ of a partition P of hypergraph H [24]. The relevant one for our context is known as connectivity-1 (or k-1) cut, defined as follows:

$$cuts(H,P) = \sum_{n_j \in N} c_j(\lambda_j - 1) \quad (2)$$

The hypergraph partitioning problem [24] can then be defined as the task of dividing a hypergraph into k parts such that the cut-size (Eq. 2) is minimized while the balance criterion (Eq. 1) is maintained.

2.2 Multilevel Partitioning

Although graph and hypergraph partitioning are NP-hard [13, 24], algorithms based on multilevel paradigms [3, 16, 21] have been shown to quickly compute good partitions in practice for both graphs [15, 20, 32] and hypergraphs [6, 22]. Recently the multilevel partitioning paradigm has been adopted by parallel graph [23, 32], and hypergraph [9, 29] partitioners.

In multilevel partitioning, instead of directly partitioning the original large hypergraph (graph), a hierarchy of smaller hypergraphs (graphs) that approximate the original is generated during the *coarsening* phase. The smallest hypergraph (graph) is partitioned in the *coarse partitioning* phase. In the *refinement* phase, the coarse partition is projected back to the larger hypergraphs (graphs) in the hierarchy and improved using a local optimization method.

3 A New Hypergraph Model for Dynamic Load Balancing

Dynamic load balancing (repartitioning) is difficult because there are multiple objectives that often conflict. Thus, some algorithms focus on minimizing communication cost while others focus on migration cost. We propose a novel unified model that combines both communication cost and migration cost. We then minimize the composite objective directly using hypergraph partitioning.

First consider the computational structure of an adaptive application in more detail. A typical adaptive application, e.g., time-stepping numerical methods with adaptive meshes, performs a sequence of iterations. Between each iteration the structure of the problem (computation) may change slightly, but usually not much. After a certain number of iterations, a load balancer is called to rebalance the workloads. The required data is then moved (migrated) between parts to establish the new partitioning, and the computation continues.

We call the period between two subsequent load balancings an *epoch* of the application. A single epoch may consist of one or more iterations of the computation in the application. Let the number of iterations in epoch j be α_j.

It is possible to model the computational structure and dependencies of each epoch using a computational hypergraph [5]. Since each epoch contains computations of the same type, but the structure may change, a different hypergraph is needed to represent each epoch. Let $H^j = (V^j, E^j)$ be the hypergraph that models the jth epoch of the application.

We assume the following procedure. Load balancing of the first epoch is achieved by partitioning the first epoch hypergraph H^1 using a static partitioner. At the end of epoch 1, we need to decide how to redistribute the data and computation for epoch 2. The cost should be the sum of the communication cost for H^2, with the new data distribution, scaled by α_2 (since epoch 2 will have α_2 iterations) plus the migration cost for moving data between the two distributions. This principle holds for H^j at every epoch j, $j > 1$. We assume that the hypergraph H^j describing the task-data dependencies of the computations of epoch j is known at the end of epoch $j-1$. As described below, we construct a *repartitioning hypergraph* by augmenting epoch j's hypergraph H^j with additional vertices and nets to model data migration cost. With these additions, dynamic load balancing (repartitioning) reduces to hypergraph partitioning with fixed vertices. After this repartitioning, the resulting partition can be decoded easily to infer the data-migration pattern and cost.

We propose a new *repartitioning hypergraph model*. The repartitioning hypergraph $\bar{H}^j$ for epoch j is constructed by augmenting epoch j's hypergraph H^j with k new vertices and $|V^j|$ new hyperedges to model the migration cost. In $\bar{H}^j$ we keep the vertex weights intact, but we scale each net's cost (representing communication) by α_j. We add one new *partition vertex* u_i, with zero weight, for each partition i, $i = 1, 2, \ldots, k$. Thus, the vertices in $\bar{H}^j$ are $V^j \cup \{u_i | i = 1, 2, \ldots, k\}$. For each vertex $v \in V^j$, we add a *migration* net between v and u_i if v is assigned to partition i at the end of epoch $j-1$. This migration net represents the data that needs to be migrated for moving vertex v to a different partition; therefore, its cost is set to the size of the data associated with v.

Figure 1 illustrates a sample hypergraph H^{j-1} for epoch $j-1$, and a repartitioning hypergraph for epoch j. Our model does not require distinguishing between the two types of vertices and nets; however, for clarity in this figure, we represent computation vertices with circles and partition vertices u_i with hexagons. Similarly, nets modeling

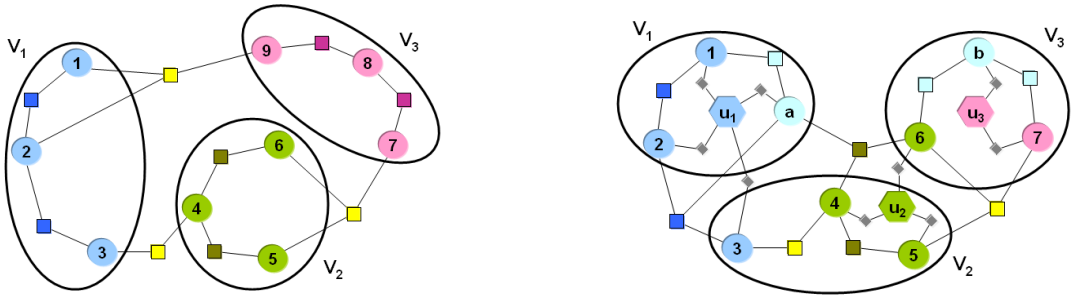

Figure 1. (left) A sample hypergraph for epoch $j-1$; (right) repartitioning hypergraph for epoch j with a sample partitioning.

communication during computation are represented with squares, and migration nets modeling data that must be migrated if a vertex assignment changes are represented with diamonds. At epoch $j-1$ there are nine vertices with, say, unit weights partitioned into three parts with a perfect load balance. There are three cut nets representing data that need to be communicated between three parts. Assuming the cost of each net is one, the total communication volume (Eq. 2) is three, since each cut net has a connectivity of two. In other words, each iteration of epoch $j-1$ incurs a communication cost of three.

In epoch j of Figure 1 (right), the computational structure is changed: vertices 8 and 9 are removed, and new vertices a and b are added. The repartitioning hypergraph $\bar{H}^j$ shown reflects these changes. Additionally, there are three partition vertices u_1, u_2 and u_3. The seven old vertices of H^{j-1} are connected, via migration nets, to the partition vertices for the partitions to which they were assigned in epoch $j-1$. Similarly, new vertices a and b are connected to the partition vertices associated with the partition they were created.

We now have a new repartitioning hypergraph $\bar{H}^j$ that encodes both communication cost and migration cost. By using this novel repartitioning hypergraph with a crucial constraint — vertex u_i must be assigned, or fixed, to partition i — the repartitioning problem reduces to hypergraph partitioning with *fixed* vertices. In Section 4, we describe how partitioning with fixed vertices can be achieved in a parallel multilevel hypergraph partitioning framework.

Let $P = \{V_1, V_2, \ldots, V_k\}$ be a valid partitioning for this problem. We decode the result as follows. If a vertex v is assigned to partition V_p in epoch $j-1$ and to partition V_q in epoch j, where $p \neq q$, then the migration net between v and u_p is cut with connectivity 2, since u_p is fixed in V_p. Hence this migration net will contribute its cost, size of the data associated with v, to the cut-size (Eq. 2) accurately modeling the migration cost of vertex v's data. Recall that cost of a communication net is size of the data item that will be communicated during computation scaled by the number of iterations. If a communication net with connectivity λ is cut, it contributes $(\lambda - 1)$ times its cost to the total cut-size accounting the true communication volume incurred by this net in the computation phase [5]. Thus our repartitioning hypergraph accurately models the sum of communication during computation phase plus migration cost due to moved data.

In Figure 1, assume that the example epoch j has, say, five iterations, i.e. $\alpha_j = 5$. Then the cost of each communication net is five. Further assume that each vertex has size three; i.e. the migration cost of each vertex, hence the cost of each migration net, is three. In this example, vertices 3 and 6 are moved to partitions V_2 and V_3, respectively. Thus, the migration nets connecting them to their previous parts are now cut with connectivity of two. Total migration cost is then $2 \times 3 \times (2-1) = 6$. Furthermore, communication nets $\{2, 3, a\}$ and $\{5, 6, 7\}$ are cut with connectivity of two, as is net $\{4, 6, a\}$ with connectivity of three. They represent a total communication volume of $2 \times 5 \times (2-1) + 1 \times 5 \times (3-1) = 20$, resulting in a total cost of 26 for epoch j.

4 Parallel Multilevel Hypergraph Partitioning with Fixed Vertices

Another contribution of our work is the development of a new technique for parallel hypergraph partitioning with fixed vertices. As described in Section 2, hypergraph partitioning is NP-hard but can be effectively (approximately) solved in practice using multilevel heuristic approaches. Multilevel hypergraph partitioning algorithms can be adapted to handle fixed vertices [6]. Here we describe our technique for parallel multilevel hypergraph partitioning with fixed vertices. Our implementation is based on the parallel hypergraph partitioner in Zoltan [9].

The main idea of partitioning with fixed vertices is to make sure that the fixed partition constraint of each ver-

tex is maintained during phases of multilevel partitioning. We will first describe how this works assuming that we are using a direct k-way multilevel paradigm. Later we will briefly discuss how this is handled when a recursive bisection approach is used.

4.1 Coarsening Phase

The goal of the coarsening phase is to approximate the original hypergraph via a succession of smaller hypergraphs. This process terminates when the coarse hypergraph is small enough (e.g., it has less than $2k$ vertices) or when the last coarsening step fails to reduce the hypergraph size by a threshold (typically 10%). In this work we employ a method based on merging *similar* pairs of vertices. We adopted a method called *inner-product matching* (IPM), that was initially developed in PaToH [5] (where it was called heavy-connectivity matching), and later adopted by hMETIS [18] and Mondriaan [31]. The greedy first-choice method is used to match pairs of vertices.

Conceptually, the parallel implementation of IPM works in rounds where in each round, each processor selects a subset of vertices as candidate vertices that will be matched in that round. The candidate vertices are sent to all processors. Then all processors concurrently contribute the computation of their *best* match for those candidates. Matching is finalized by selecting a global best match for each candidate. Zoltan uses a two-dimensional data distribution; hence, the actual inner workings of IPM are somewhat complicated. Since a detailed description is not needed to explain the extension for handling fixed vertices, we have omitted those details. Readers may refer to [9] for more details.

During the coarsening, we do not allow two vertices to match if they are fixed to different partitions. Thus, there are three possible scenarios: 1) two matched vertices are fixed to the same partition, 2) only one of the matched vertices is fixed to a partition, or 3) both are not fixed to any partitions (free vertices). For cases 1 and 2, the resulting coarse vertex is fixed to the part in which either of its constituent vertices was fixed; for case 3, the resulting coarse vertex is free. By constraining matching in this way, we ensure that the fixed vertex information appropriately propagates to coarser hypergraphs, and coarser hypergraphs truly approximate the finer hypergraphs and their constraints.

In order to efficiently implement this restriction, we allow each processor to concurrently compute all match scores of possible matches, including infeasible ones (due to the matching constraint), but at the end when the best local match for each candidate is selected we select a match that obeys the matching constraint. We have observed that this scheme only adds an insignificant overhead to the unrestricted IPM matching.

4.2 Coarse Partitioning Phase

The goal of this phase is to construct an initial solution using the coarsest hypergraph available. When coarsening stops, if the coarsest hypergraph is small enough (i.e., if coarsening did not terminate early due to unsuccessful coarsening) we replicate it on every processor and each processor runs a randomized greedy hypergraph growing algorithm to compute a different partitioning into k partitions. If the coarsest hypergraph is not small enough, then each processor contributes computation of an initial partitioning using a localized version of the greedy hypergraph algorithm. In either case, we ensure that fixed coarse vertices are assigned to their respective partitions.

4.3 Refinement Phase

The refinement phase takes a partition assignment, projects it to finer hypergraphs and improves it using a local optimization method. Our code is based on a localized version of the successful Fiduccia–Mattheyses [12] method, as described in [9]. The algorithm performs multiple passpairs and in each pass, each vertex is considered to move to another part to reduce cut cost. As in coarse partitioning, the modification to handle fixed vertices is quite straightforward. We do not allow fixed vertices to be moved out of their fixed partition.

4.4 Handling Fixed Vertices in Recursive Bisection

Achieving k-way partitioning via recursive bisection (repeated subdivision of parts into two parts) 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 as follows: vertices that are originally fixed to partitions $1 \leq p \leq k/2$, are fixed to partition 1, and vertices originally fixed to partitions $k/2 < p \leq k$ are fixed to partition 2. The partitioning algorithm with fixed vertices then can be executed without any modifications. This scheme is recursively applied in each bisection. Zoltan uses this recursive bisection approach.

5 Results

Our repartitioning code is based on the hypergraph partitioner in the Zoltan toolkit [10, 9], which is freely available from the Zoltan web site[1]. The code is written in C and uses MPI for communication. We ran our tests on a Linux cluster that has 64 dual-processor Opteron 250 nodes interconnected via an Infiniband network.

[1] `www.cs.sandia.gov/Zoltan`

Name	$\|V\|$	$\|E\|$	vertex degree			Application Area
			min	max	avg	
xyce680s	682,712	823,232	1	209	2.4	VLSI design
2DLipid	4,368	2,793,988	396	1,984	1,279.3	Polymer DFT
auto	448,695	3,314,611	4	37	14.8	Structural analysis
apoa1-10	92,224	17,100,850	54	503	370.9	Molecular dynamics
cage14	1,505,785	13,565,176	3	41	18.0	DNA electrophoresis

Table 1. Properties of the test datasets; $|V|$ and $|E|$ are the numbers of vertices and graph edges, respectively.

Due to the difficulty of obtaining data from real-world adaptive simulations, we present results from synthetic dynamic data. The base cases were obtained from real applications, as shown in Table 1. All these problems are structurally symmetric, and can be accurately represented as both graphs and hypergraphs. (We expect our hypergraph approach to have a clear advantage for non-symmetric problems.)

We used two different methods to generate synthetic dynamic data. The first method represents biased random perturbations that change the structure of the data. In this method, we randomly select a certain fraction of vertices in the original data and delete them along with the incident edges. At each iteration, we delete a different subset of vertices from the original data. Therefore, we simulate 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 partitions lose or gain 25% of the total number of vertices at each iteration.

The second method we used to generate synthetic data simulates adaptive mesh refinement. Starting with the initial data, we randomly select a certain fraction of the partitions at each iteration. Then, the sub-domain corresponding to selected partitions performs a simulated mesh refinement, where each vertex increases both its weight and its size by a constant factor. In the results displayed in this section, 10% of the partitions are selected at each iteration and the weight and size of each vertex in these partitions are randomly increased to between 1.5 and 7.5 of their original value.

We tested several other configurations by varying the fraction of vertices lost or gained and the factor that scales the size and weight of vertices. The results we obtained in these experiments were similar to the ones presented in this section.

We compare four different algorithms:

1. Zoltan-repart: Our new method implemented within the Zoltan hypergraph partitioner,

2. Zoltan-scratch: Zoltan hypergraph partitioning from scratch.

3. ParMETIS-repart: ParMETIS graph repartitioning using the *AdaptiveRepart* option.

4. ParMETIS-scratch: ParMETIS graph partitioning from scratch (*Partkway*).

We used ParMETIS version 3.1 in these experiments. For the scratch methods, we used a maximal matching heuristic in Zoltan to map partition numbers to reduce migration cost. We do not expect the partition-from-scratch methods to be competitive for dynamic problems, but include them as a useful baseline.

In Figures 2 through 6, experimental results for total cost while varying the number of processors and α are presented. In our experiments we varied the number of processors (partitions) between 16 and 64, and α from 1 to 1000. (Our α corresponds to the ITR parameter in ParMETIS.) We report the average results over a sequence of 20 trials for each experiment. For each configuration, there are four bars representing total cost for Zoltan-repart, ParMETIS-repart, Zoltan-scratch and ParMETIS-scratch, from left to right respectively. Total cost in each bar is normalized by α and consists of two components: communication (bottom) and migration (top) costs. In order to improve the readability of the charts, we limited the y-axis for $\alpha = 1$ where total costs for Zoltan-scratch and ParMETIS-scratch were much larger than the costs for Zoltan-repart and ParMETIS-repart.

The results show that in the majority of the test cases, our new hypergraph repartitioning method Zoltan-repart outperforms ParMETIS-repart in terms of minimizing the total cost. Since minimizing the migration cost is a more deeply integrated objective starting from coarsening, Zoltan-repart trades off communication cost better than ParMETIS-repart to minimize the total cost. This is more clearly seen for small α values where minimizing migration cost is as important as minimizing the communication cost. As α grows, migration cost decreases relative to communication cost and the problem essentially reduces to minimizing the communication cost alone. Due to increased emphasis on communication volume, the partitioners find smaller communication cost with increasing α.

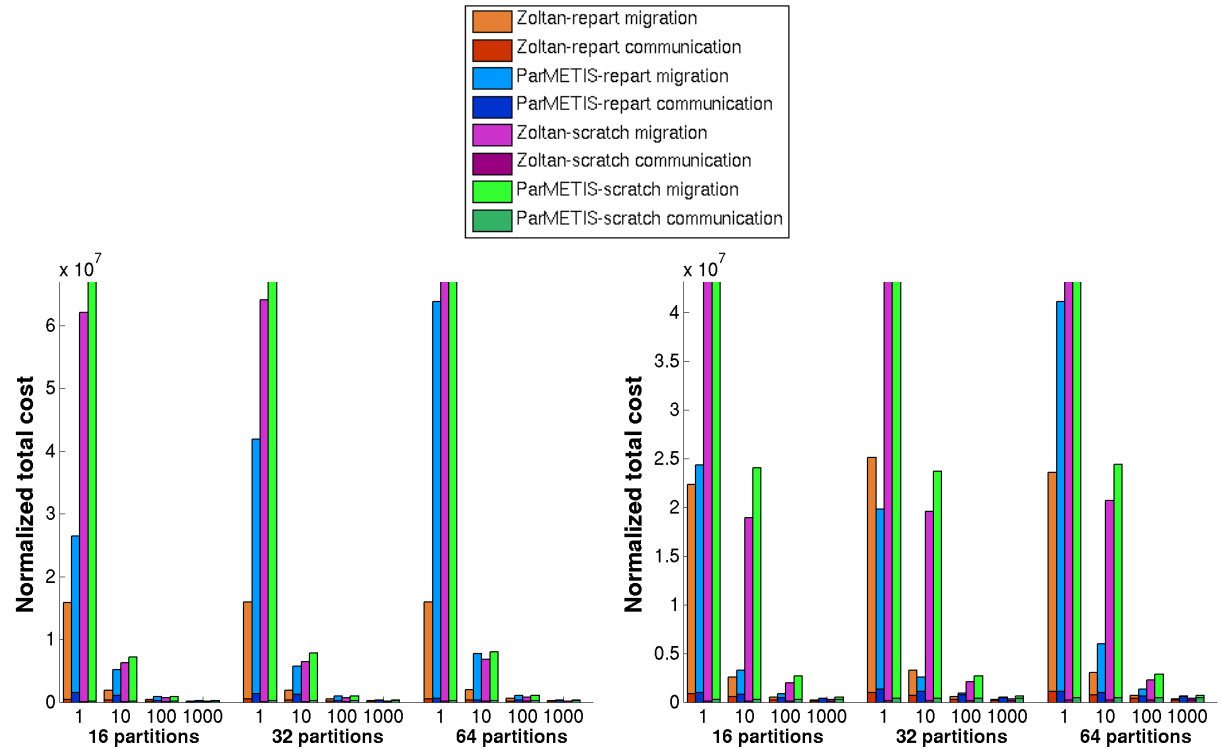

Figure 2. Normalized total cost (communication volume + (migration volume)/α) for xyce680s with (a) perturbed data structures and (b) perturbed weights. Each group of four bars represents results for Zoltan-repart, ParMETIS-repart, Zoltan-scratch and ParMETIS-scratch, from left to right respectively.

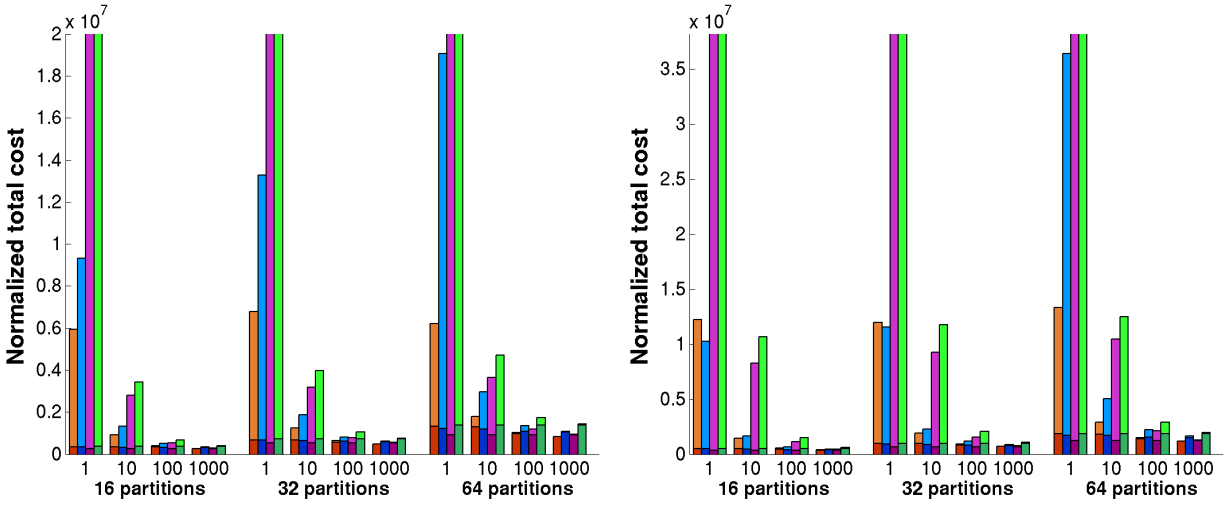

Figure 3. Normalized total cost (communication volume + (migration volume)/α) for 2DLipid with (a) perturbed data structures and (b) perturbed weights. Each group of four bars represents results for Zoltan-repart, ParMETIS-repart, Zoltan-scratch and ParMETIS-scratch, from left to right respectively.

Similar observations can be made when comparing Zoltan-repart against Zoltan-scratch and ParMETIS-scratch. Since the sole objective in Zoltan-scratch and ParMETIS-scratch is to minimize communication cost, the

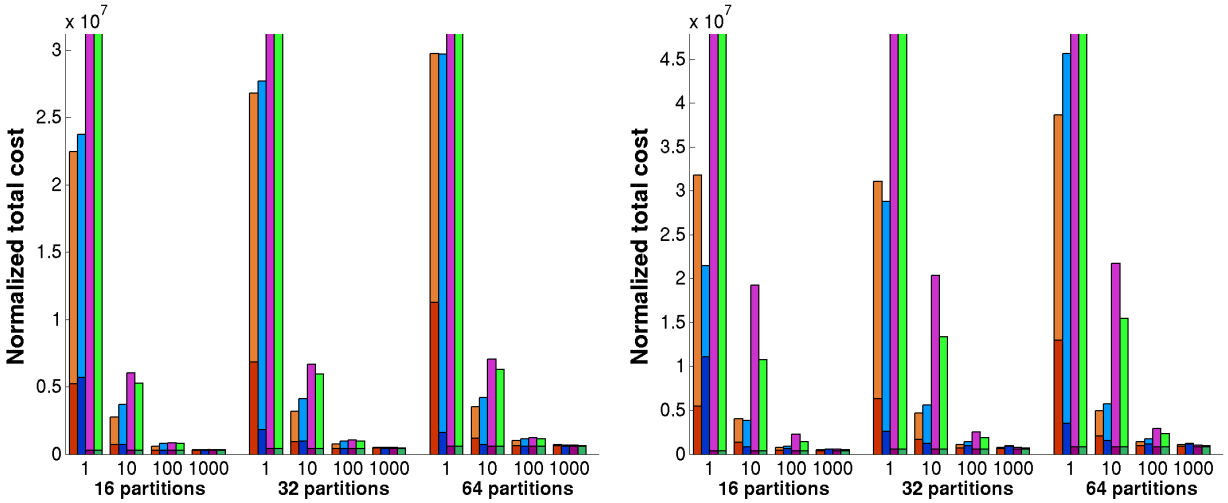

Figure 4. Normalized total cost (communication volume + (migration volume)/α) for auto dataset with (a) perturbed data structures and (b) perturbed weights. Each group of four bars represents results for Zoltan-repart, ParMETIS-repart, Zoltan-scratch and ParMETIS-scratch, from left to right respectively.

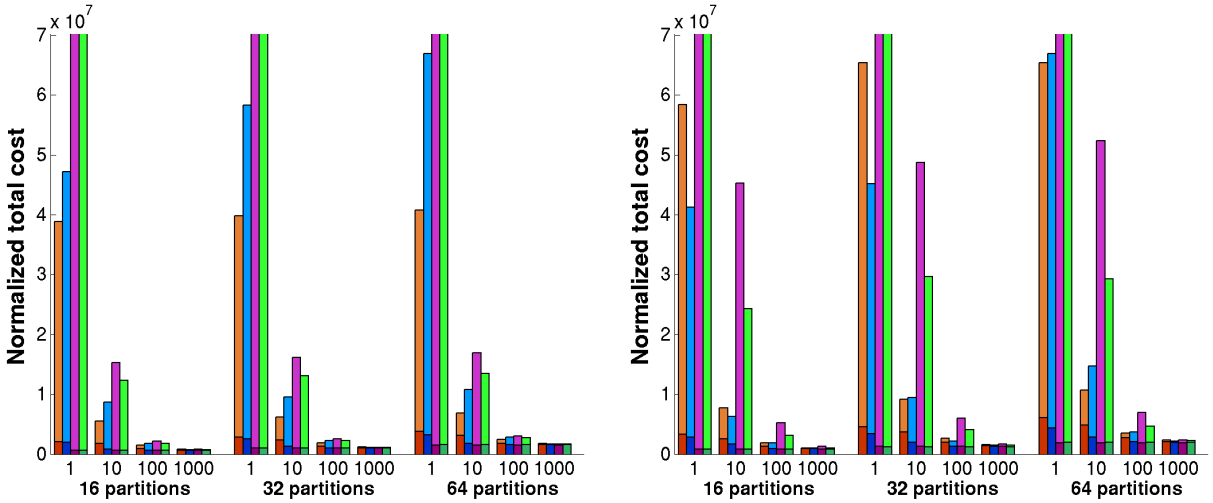

Figure 5. Normalized total cost (communication volume + (migration volume)/α) for apoa1-10 with (a) perturbed data structures and (b) perturbed weights. Each group of four bars represents results for Zoltan-repart, ParMETIS-repart, Zoltan-scratch and ParMETIS-scratch, from left to right respectively.

migration cost is extremely large, especially for small α. The total cost using Zoltan-scratch and ParMETIS-scratch is comparable to Zoltan-repart only when α is greater than 100. For larger values of α, the objective of minimizing the communication cost dominates; however, Zoltan-repart still performs as well as the scratch methods to minimize the total cost.

When using ParMETIS-repart, migration cost increases noticeably compared to communication cost with increasing number of partitions (processors). On the other hand, with Zoltan-repart, the increase in migration cost is kept small at the expense of a modest increase in communication cost. Consequently, Zoltan-repart achieves a better balance between communication and migration costs; hence, the total cost gets better compared to ParMETIS-repart as the number of partitions increases. This shows that Zoltan-repart is superior in minimizing the total cost objective as well as in scalability of the solution quality compared to

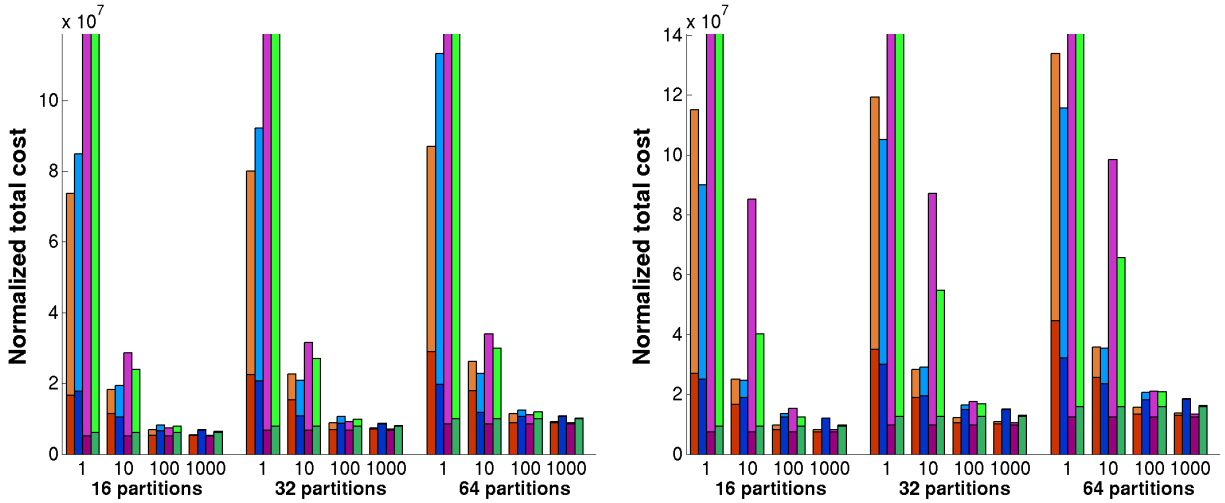

Figure 6. Normalized total cost (communication volume + (migration volume)/α) for cage14 with (a) perturbed data structures and (b) perturbed weights. Each group of four bars represents results for Zoltan-repart, ParMETIS-repart, Zoltan-scratch and ParMETIS-scratch, from left to right respectively.

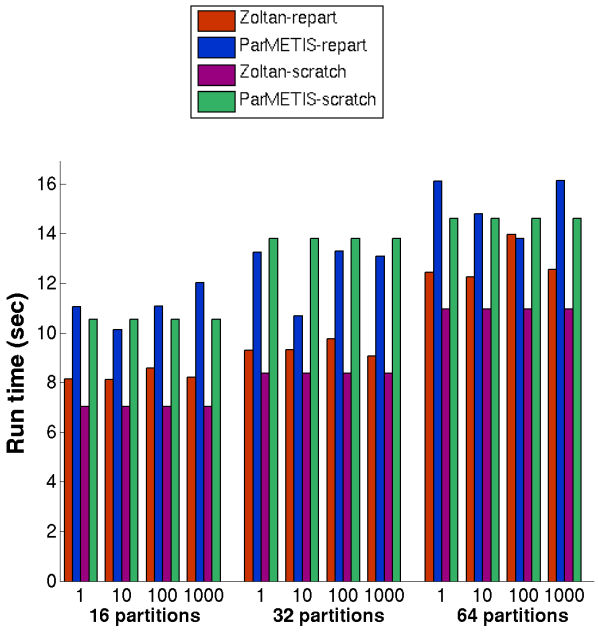

Figure 7. Run time with perturbed data structure for xyce680s. Each group of four bars represents results for Zoltan-repart, ParMETIS-repart, Zoltan-scratch and ParMETIS-scratch, from left to right respectively.

ParMETIS-repart.

Run times of the tested partitioners while changing the data's structure are given in Figures 7 and 8. Results for changing vertex weights and sizes are omitted since they were similar to the ones presented here. As shown in the figures, Zoltan-repart is at least as fast as ParMETIS-repart on the sparse dataset xyce680s. For the dense 2DLipid data, although graph partitioning approaches run faster for small number of partitions, their execution times increase and become comparable to that of hypergraph partitioning approaches as the number of partitions increases. For medium-dense graphs (e.g., auto, cage14, apoa1-10), Zoltan-repart is significantly slower than ParMETIS-repart. Here, we present run time results for only the auto data. The results for cage14 and apoa1-10 were very similar to that of auto, displaying 10 to 15 times faster execution for graph-based approaches compared to hypergraph-based ones. We plan to improve this performance by using local heuristics in Zoltan-repart to reduce global communication (e.g., using local IPM instead of global IPM).

6 Conclusions

We have 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. Our experiments, using data from a wide range of application areas, show that our method produces partitions that give similar or lower cost than the adaptive repartitioning scheme in ParMETIS. Our code generally required longer time than ParMETIS but that is mostly due to the greater richness of the hypergraph model.

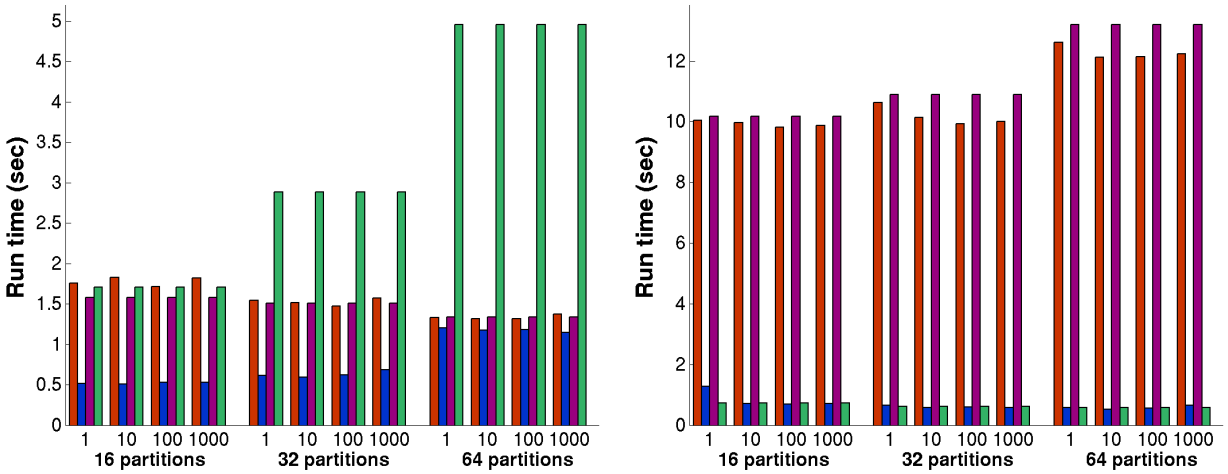

Figure 8. Run time with perturbed data structure for (a) 2DLipid and (b) auto. Each group of four bars represents results for Zoltan-repart, ParMETIS-repart, Zoltan-scratch and ParMETIS-scratch, from left to right respectively.

The full benefit of hypergraph partitioning is realized on unsymmetric and non-square problems that cannot be represented easily with graph models. To provide comparisions with graph repartitioners, we did not test such problems here, but they have been studied elsewhere [5, 9]. The experiments showed that our implementation is scalable.

Our approach uses a single user-defined parameter α to trade between communication cost and migration cost. Experiments show that our method works particularly well when migration cost is more important, but without compromising quality when communication cost is more important. Therefore, we recommend our algorithm as a universal method for dynamic load balancing. The best choice of α will depend on the application, and can be estimated. Reasonable values are in the range $1 - 1000$.

In future work, we will test our algorithm and implementation on real adaptive applications. We will also attempt to speed up our algorithm by exploiting locality given by the data distribution. We believe the implementation can be made to run faster without reducing quality. 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 Bruce Hendrickson and Vitus Leung for their contributions to the Zoltan project.

References

[1] 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. *Journal of Parallel and Distributed Computing*, 67(1):77–99, Jan 2007.

[2] E. Boman, K. Devine, L. A. Fisk, R. Heaphy, B. Hendrickson, C. Vaughan, U. Catalyurek, D. Bozdag, and W. Mitchell. *Zoltan 2.0: Data Management Services for Parallel Applications; User's Guide*. Sandia National Laboratories, Albuquerque, NM, 2006. Tech. Report SAND2006-2958 `http://www.cs.sandia.gov/Zoltan/ug_html/ug.html`.

[3] 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.

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

[5] U. V. Çatalyürek and C. Aykanat. Hypergraph-partitioning based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):673–693, 1999.

[6] 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.

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

[8] 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.

[9] 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.

[10] 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(2):90–97, 2002.

[11] 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.

[12] 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.

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

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

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

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

[17] 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.

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

[19] G. Karypis and V. Kumar. METIS 3.0: Unstructured graph partitioning and sparse matrix ordering system. Technical Report 97-061, Dept. Computer Science, University of Minnesota, 1997. *http://www.cs.umn.edu/˜metis*.

[20] 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.

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

[22] 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.

[23] 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`.

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

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

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

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

[28] 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.

[29] 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.

[30] 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.

[31] B. Vastenhouw and R. H. Bisseling. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM Review*, 47(1):67–95, 2005.

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

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

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