Contents

Part I Dynamic Partitioning and Adaptive Runtime Management Frameworks **1**

1 Hypergraph-based Dynamic Partitioning and Load Balancing **3**

U.V. Catalyurek, D. Bozdağ, E.G. Boman, K.D. Devine, R. Heaphy and L.A. Riesen

1.1 Introduction 3
1.2 Preliminaries 6
1.3 Hypergraph-based Repartitioning Methods 7
1.4 Parallel Repartitioning Tool 11
1.5 Experimental Results 13
1.6 Conclusion 22
References 22

Part I

Dynamic Partitioning and Adaptive Runtime Management Frameworks

1 Hypergraph-based Dynamic Partitioning and Load Balancing

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

Dept. of Biomedical Informatics, The Ohio State University, Columbus, Ohio
Discrete Algorithms and Math. Dept., Sandia National Laboratories,[1] Albuquerque, New Mexico

1.1 INTRODUCTION

An important component of parallel scientific computing is the *assignment* of work to processors. This assignment problem is also known as *partitioning* or *mapping*. The goal of the assignment problem is to find a task-to-processor mapping that will minimize the total execution time. Although efficient optimal solutions for certain restricted variations, such as chain- or tree-structured programs exist [21], the general problem is NP-hard [28]. We consider this general version where any task can potentially be assigned to any processor. In the literature, the task-to-processor assignment problem is usually solved by a two-step approach: first tasks are *partitioned* into *load-balanced* clusters of tasks, then these clusters are mapped to processors. 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*)

[1]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.

as the work distribution changes over the course of the computation. Repartitioning usually includes both partitioning and mapping. While distinction between *partitioning*, *load balancing* and *mapping* can be made as described, throughout this chapter we will use the terms interchangeably to refer the assignment problem. We will assume that the mapping step of the two-step approach either is trivial, or inherent in the partitioning approach.

Dynamic partitioning is an important feature in parallel adaptive computations [9]. 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 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. Therefore dynamic load balancing may be required periodically to re-balance the application, that is, move data among processors to improve the load balance.

Dynamic load balancing or repartitioning is a well studied problem [9, 10, 13, 15, 19, 31, 32, 34, 35, 39, 37, 41, 43, 44] that 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 determined by the new distribution);
3. low data migration cost to move data from the old to the new distribution; and
4. short repartitioning time.

Total execution time is commonly modeled [26, 34] as follows to account for these objectives:

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

Here, 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. Furthermore, t_{repart} is typically much smaller than αt_{comp} for most applications thanks to fast state-of-the-art repartitioners. Thus, the objective reduces to minimize $\alpha t_{comm} + t_{mig}$.

Much of the early work in dynamic load balancing focused on diffusive methods [9, 19, 20, 32, 33, 41, 43], where overloaded processors give work to

neighboring processors that have lower than average loads. Another approach, *scratch-remap*, partitions the new problem *from scratch* without accounting for existing partition assignments, and then tries to remap partitions to minimize the migration cost [35].

Diffusive schemes are fast and have low migration cost, but may incur high communication volume. Scratch-remap schemes achieve low communication volume, but are slow 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 [34], 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. [2] 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 [22]. 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 [5] 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 the recent work, Catalyurek et al. [6] proposed a generalized hypergraph model for repartitioning. As will be described in more detail in Section 1.3, the new hypergraph model minimizes the sum of the 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) [7]. Furthermore, the natural combined representation of the two costs in a single hypergraph is more suitable to successful multilevel partitioning schemes than graph-based approaches (see Section 1.2 for details). The new model can be more effectively realized with a parallel repartitioning tool. However, it necessitates hypergraph partitioning with fixed vertices. Although serial hypergraph partitioners with fixed-vertex partitioning exist (PaToH [8]), to the best of our knowledge Zoltan [3] is the first parallel hypergraph partitioner with this feature.

The remainder of this chapter is organized as follows. In Section 1.2, we present some preliminaries for hypergraph partitioning and multilevel partitioning. In Section 1.3, we discuss three approaches for dynamic load balancing using hypergraphs. The parallel repartitioning tool developed within the

Zoltan [45] framework is presented in Section 1.4. Section 1.5 includes empirical results of the hypergraph-based repartitioning approaches compared to graph-based repartitioning approaches. Finally, we give our conclusions and suggest future work.

1.2 PRELIMINARIES

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

1.2.1 Hypergraph Partitioning with Fixed Vertices

Hypergraphs can be viewed as a generalization of graphs where each 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 among those vertices, where each net $n_j \in N$ is a non-empty subset of vertices. 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

$$W_p \leq W_{avg}(1 + \epsilon) \text{ for } p = 1, 2, \ldots, k, \tag{1.1}$$

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 cost of a partition P of hypergraph H [30], $cost(H, P)$. The relevant one for our context is known as *connectivity-1* (or k-1) cost, defined as follows:

$$cost(H, P) = \sum_{n_j \in N} c_j(\lambda_j - 1) \tag{1.2}$$

The standard hypergraph partitioning problem [30] can then be defined as the task of dividing a hypergraph into k parts such that the cost (1.2) is minimized while the balance criterion (1.1) is maintained.

The problem of *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 number of parts k, a *fixed-part* function $f(v)$ is also given 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 partition 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 percent of the vertices are fixed, it is expected that 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.

1.2.2 Multilevel Partitioning Paradigm

Although graph and hypergraph partitioning are NP-hard [16, 30], algorithms based on multilevel paradigms [4, 18, 24] have been shown to quickly compute good partitions in practice for both graphs [17, 23, 42] and hypergraphs [8, 25]. Recently the multilevel partitioning paradigm has been adopted by parallel graph [42, 26] and hypergraph [12, 38] 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.

Multilevel hypergraph partitioning algorithms can be adapted to handle fixed vertices. In this chapter, we describe a technique for parallel multilevel hypergraph partitioning with fixed vertices [6]. The implementation is based on the parallel hypergraph partitioner in Zoltan [12].

1.3 HYPERGRAPH-BASED REPARTITIONING METHODS

A typical adaptive application, e.g. time-stepping numerical methods with adaptive meshes, consists of a sequence of iterations. At each iteration the structure of the problem (computation) may change slightly, usually insignificantly. After a certain number of iterations, these changes accumulate and the workload becomes unevenly distributed among the processors. In order to restore the balance, a load balancer is invoked and some of the data are moved (migrated) to establish a new partitioning. Then the computation resumes and this process is repeated until the application is finished.

In dynamic load balancing (repartitioning), communication cost and migration cost are the two main objectives to be minimized. These objectives are typically conflicting, posing a challenge in designing an efficient load balancing algorithm. The majority of the previous work is based on simplifying the problem by setting either of these objectives as a primary one and the other as a secondary, thereby impeding any trade-off between them. Furthermore, none of the previous methods are designed for or applied to hypergraphs. In this section, we first discuss two naive approaches to achieve dynamic load balancing using hypergraph partitioning, and then introduce a new repartitioning hypergraph model that accurately represents the costs associated with the dynamic load balancing problem.

1.3.1 Scratch-Remap Based Repartitioning

One of the trivial approaches for dynamic load balancing is to repartition the modified hypergraph that models the application from scratch after certain number of iterations. Previous partition assignments are ignored during partitioning. However, they are used in a post-processing step that reorders partitions to maximize the overlap between new and previous assignments and, thus, reduce the migration cost.

1.3.2 Refinement Based Repartitioning

Second naive approach is just a simple application of successful move or swap based refinement algorithms; such as Kernighan-Lin [29], Schweikert-Kernighan [36] or Fiduccia-Mattheyses [14]. In this approach, vertices initially preserve their previous assignments and new vertices are assigned to existing parts via a simple, possibly a random, partitioner. Then, a single refinement is applied to minimize the communication cost. Since vertices are allowed to change partitions only during the refinement phase, migration cost is somewhat constrained. However, constrained vertex movement may lead to lower cut quality and, hence, larger communication cost. This approach may be extended to the multilevel partitioning framework to improve cut quality (albeit with increased partitioning time). However, we do not present multilevel refinement results here, as our new model below incorporates both coarsening and refinement in a more accurate way to achieve repartitioning.

1.3.3 A New Repartitioning Hypergraph Model

Our work is based on the composite objective function defined in Section 1.1 to allow trade-off between communication and migration costs. We present a novel hypergraph model to represent the repartitioning problem. Hypergraph partitioning is then directly applied to optimize the composite objective.

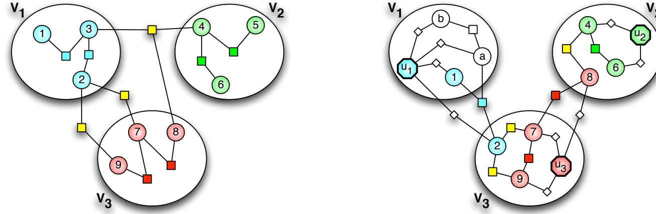

Fig. 1.1 (left) A sample hypergraph for epoch $j-1$, (right) repartitioning hypergraph for epoch j with a sample partitioning.

We call the period between two subsequent load balancing operations an *epoch* of the application. An epoch consists of one or more computation iterations. The computational structure and dependencies of an epoch can be modeled using a computational hypergraph [7]. Since all computations in the application are of the same type but the structure is different across epochs, a different hypergraph is needed to represent each epoch. We denote the hypergraph that models the jth epoch of the application by $H^j = (V^j, E^j)$ and the number of 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. Total cost should be the sum of the communication cost for H^j with the new data distribution, scaled by α_j, and the migration cost 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 $\bar{H}^j = (\bar{V}^j, \bar{E}^j)$ by augmenting H^j with k new vertices and $|V^j|$ new hyperedges using the following procedure:

- Scale each net's cost (representing communication) in E^j by α_j while keeping the vertex weights intact.

- Add a new *partition vertex* u_i with zero weight for each partition 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* between v and u_i if v is assigned to partition i at the beginning of epoch j. Set the migration net's cost to the size of the data associated with v, since this migration net represents the cost of moving vertex v to a different partition.

Figure 1.1 illustrates a sample hypergraph H^{j-1} for epoch $j-1$ (left), and a repartitioning hypergraph $\bar{H}^j$ for epoch j (right). A nice feature of our

model is that no distinction is required between communication and migration vertices as well as nets. However, for clarity in this figure, we represent computation vertices with circles and partition vertices with hexagons. Similarly, nets representing communication during computation are represented with squares, and migration nets representing data to be migrated due to changing vertex assignments are represented with diamonds. In this example, there are nine unit weight vertices partitioned into three parts with a perfect balance at epoch $j-1$. Three cut nets represent data that need to be communicated among the parts. Two of these nets have connectivity of two and one has three. Assuming unit cost for each net, total communication volume is four (Equation 1.2). In other words, each iteration of epoch $j-1$ incurs a communication cost of four.

The computational structure changes in epoch j as displayed in Figure 1.1 (right). In H^j, new vertices a and b are added while vertices 3 and 5 are deleted. To construct the repartitioning hypergraph $\bar{H}^j$ from H^j, three partition 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 seven computation vertices inherited from H^{j-1} is connected to the partition vertex associated with the partition to which the computation vertex was assigned in epoch $j-1$, via a migration net.

Once the new repartitioning hypergraph $\bar{H}^j$ that encodes both communication and migration costs is obtained, the repartitioning problem reduces to hypergraph partitioning with *fixed* vertices. Here, the constraint is that vertex u_i must be assigned, or fixed, to partition i.

Let $P = \{V_1, V_2, \ldots, V_k\}$ be a valid partitioning for this problem. Assume that a vertex v is assigned to partition V_p in epoch $j-1$ and V_q in epoch j, where $p \neq q$. Then, the migration net between v and u_q that represents the migration cost of vertex v's data is cut (note that u_q is fixed in V_q). Therefore, cost of moving vertex v from partition V_p to V_q is appropriately included in the total cost. If a net that represents a communication during computation phase is cut, 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 plus migration cost due to moved data.

In the example given in Figure 1.1, assume that epoch j consists of five iterations, i.e. $\alpha_j = 5$. Then, each of the unit weight communication nets incurs a communication cost of five in epoch j. Further, assume that the size of each vertex is three, i.e. moving a vertex across partitions adds three units to total communication volume. In the repartitioning hypergraph $\bar{H}^j$, this data movement is represented by migration nets of weight three connecting each vertex to the fixed vertex in its partition. In this example, after applying hypergraph partitioning on $\bar{H}^j$, vertices 2 and 8 are assigned to partitions V_3 and V_2, respectively. The cost associated with migration of these vertices is captured by associated migration nets being cut with connectivity of two.

Total migration cost is then $2 \times 3 \times (2-1) = 6$. In this partitioning, two communication nets ($\{1, 2, a\}$ and $\{7, 8\}$) are also cut with connectivity of two, resulting in a total application communication volume of $2 \times 5 \times (2-1) = 10$. Thus, the total cost of epoch j is 16.

1.4 PARALLEL REPARTITIONING TOOL

Effective application of the model presented in the previous section for dynamic repartitioning necessitates a parallel hypergraph partitioning with fixed vertices. As described in Section 1.2, hypergraph partitioning is NP-hard but can be effectively solved in practice using multilevel heuristic approaches. Multilevel hypergraph partitioning algorithms can be adapted to handle fixed vertices [8], and here we describe a technique for parallel multilevel hypergraph partitioning with fixed vertices [6] that is based on the parallel hypergraph partitioner in Zoltan [12].

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

1.4.1 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 specified amount (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 [7] (where it was called heavy-connectivity matching), and later adopted by hMETIS [27] and Mondriaan [40]. 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 [12] 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 in which vertices match: 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 adds only an insignificant overhead to the unrestricted IPM matching.

1.4.2 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.

1.4.3 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 [14] method, as described in [12]. The algorithm performs multiple pass-pairs 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 straight-forward. We do not allow fixed vertices to be moved out of their fixed partition.

1.4.4 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.

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

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.

1.5 EXPERIMENTAL RESULTS

Our repartitioning code is based on the hypergraph partitioner in the Zoltan toolkit [11, 12], which is freely available from the Zoltan web site[2]. 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.

Due to the difficulty of obtaining data from real-world simulations, we present results from synthetic dynamic data. The base cases were obtained from real applications, as shown in Table 1.1.

We used two different methods to generate synthetic data. The first method represents biased random perturbations that change 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.

[2] `http://www.cs.sandia.gov/Zoltan`

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 five different algorithms:

1. Z-repart: Our new method implemented within the Zoltan hypergraph partitioner (Section 1.3.3).
2. Z-scratch: Zoltan hypergraph partitioning from scratch (Section 1.3.1).
3. Z-refine: Zoltan refinement-based hypergraph repartitioning (Section 1.3.2).
4. M-repart: ParMETIS graph repartitioning using the *AdaptiveRepart* option.
5. M-scratch: ParMETIS graph partitioning from scratch (*Partkway*).

We used ParMETIS version 3.1 in these experiments [26]. 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 1.2 through 1.13, 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 five bars representing total cost for Z-repart, M-repart, Z-refine, Z-scratch and M-scratch, from left to right respectively. Total cost in each bar is normalized by the corresponding total cost of Z-repart and consists of two components: communication (bottom) and migration (top) costs. First 10 figures, (Figures 1.2–1.11), displays total cost comparison of the five methods, for each one of the five test cases, with perturbed data structure and weights. Figures 1.12 and 1.13 display the average quality results for those five test datasets.

The results show that in the majority of the test cases, our new hypergraph repartitioning method Z-repart outperforms M-repart in terms of minimizing

the total cost. Since minimizing the migration cost is a more deeply integrated objective starting from coarsening, Z-repart trades off communication cost better than M-repart to minimize the total cost. This result is more clearly seen for small and medium α values where minimizing migration cost is as important as minimizing the communication cost. For large α values, 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 observations can be made when comparing Z-repart against scratch-remap based repartitioning methods; Z-scratch and M-scratch. Since the sole objective in Z-scratch and M-scratch is to minimize communication cost, the migration cost is extremely large, especially for small α. The total cost using Z-scratch and M-scratch is comparable to Z-repart only when α is greater than 100. For larger values of α, the objective of minimizing the communication cost dominates; however, Z-repart still performs as well as the scratch methods to minimize the total cost.

Z-refine displays an opposite trend in total cost with increasing α. Due to its constrained initial partitioning, Z-refine attempts to minimize communication volume with relatively fewer vertex movements. Therefore, the cut quality of Z-refine is lower than other partitioners, resulting in a relatively higher total cost for large α values. On the other hand, Z-refine produces lower migration costs compared to scratch methods for small α. However, Z-repart still outperforms Z-refine in such cases.

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

Run times of the tested partitioners normalized by that of Z-repart while changing the data's structure are given in Figures 1.14–1.18. Results for changing vertex weights and sizes are omitted here as they were similar to the ones presented. We have observed 3 different run time profiles on our test set. As shown in the Figure 1.14, multilevel hypergraph partitioning based methods, Z-repart and Z-scratch, are at least as fast as their graph partitioning based counterparts, M-repart and M-scratch, on sparse datasets such as xyce680s. Z-refine is significantly faster than on all others in this dataset, therefore it becomes a viable option for applications that requires a very fast repartitioner for small α values. For 2DLipid (Figure 1.15), although graph based repartitioning approach runs faster for small number of

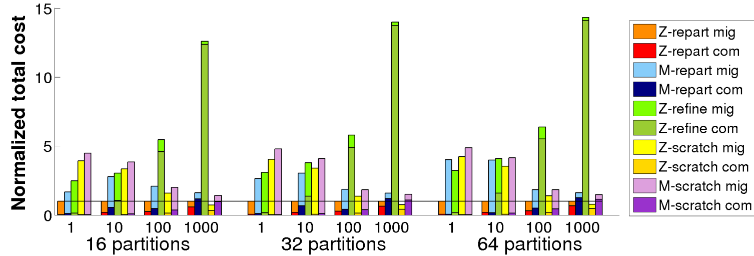

Fig. 1.2 Normalized total cost for xyce680s with perturbed data structure.

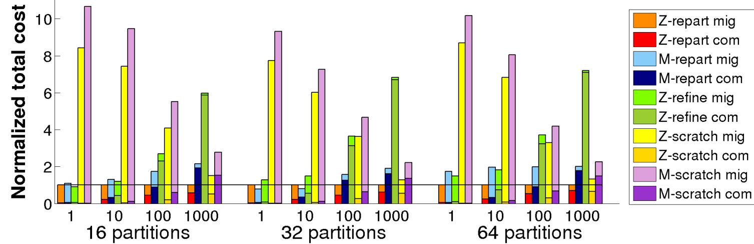

Fig. 1.3 Normalized total cost for xyce680s with perturbed weights.

partitions, with increasing number of processors, their execution time also increases and becomes comparable to that of hypergraph partitioning based repartitioning approaches. This behavior is probably due to direct k-way refinement strategies used in the graph partitioner. When a dense graph such as 2DLipid is partitioned, the number of computations done by the partitioner increases with increasing number of partitions, since it becomes more likely to have an adjacent vertex on every processor. In our tests, the last run time profile occurred in cage14, auto and apoa1-10, hence we omitted results for auto and apoa1-10 and presented the results for largest test graph we have, cage14, in Figure 1.16. This result shows that hypergraph partitioning based repartitioning approach can be up ten times slower than graph based approaches. Average run time results are presented in Figures 1.17 and 1.18. As seen in these figures, on the average, hypergraph partitioning based repartitioning approaches are about 5 times slower than graph based repartitioning approaches. We plan to improve run time performance by using local heuristics in Z-repart to reduce global communication (e.g., using local IPM instead of global IPM).

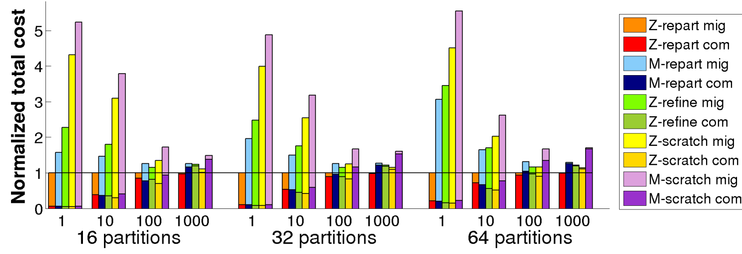

Fig. 1.4 Normalized total cost for 2DLipid with perturbed data structure.

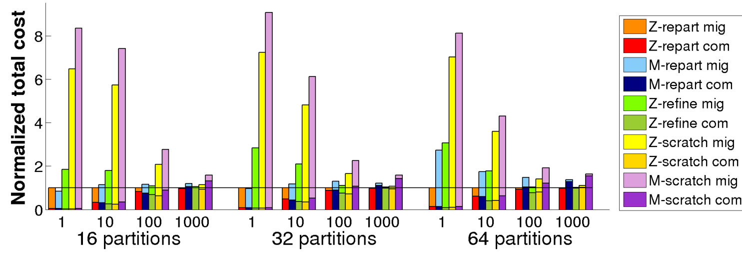

Fig. 1.5 Normalized total cost for 2DLipid with perturbed weights.

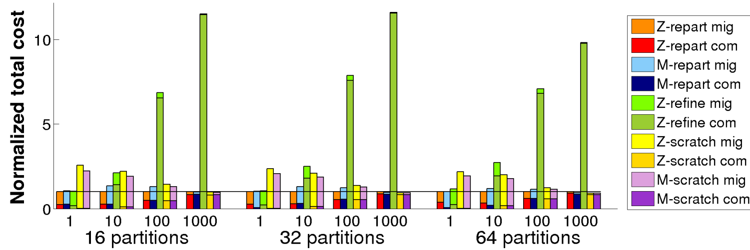

Fig. 1.6 Normalized total cost for auto dataset with perturbed data structure.

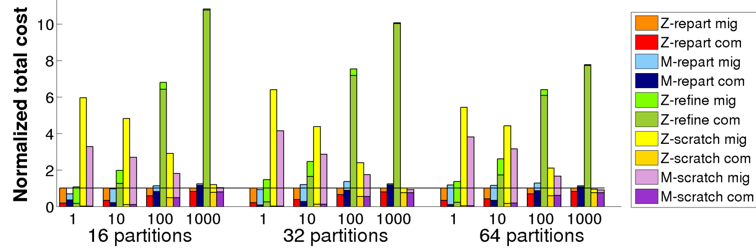

Fig. 1.7 Normalized total cost for auto dataset with perturbed weights.

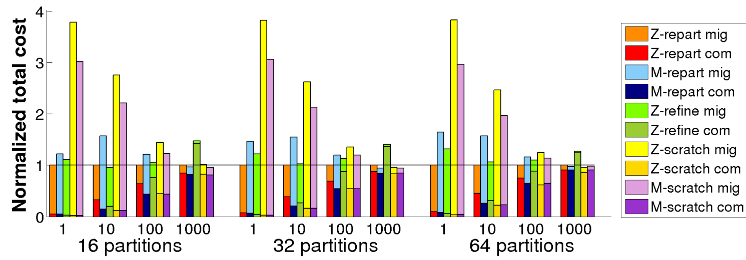

Fig. 1.8 Normalized total cost for apoa1-10 with perturbed data structure.

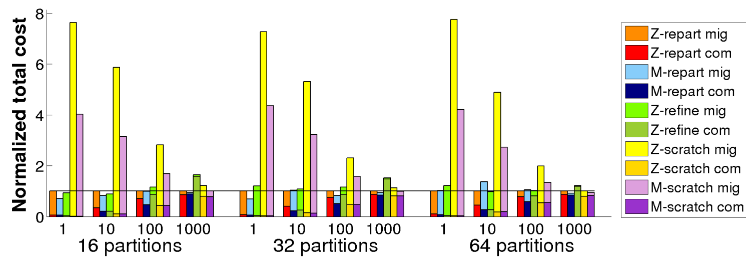

Fig. 1.9 Normalized total cost for apoa1-10 with perturbed weights.

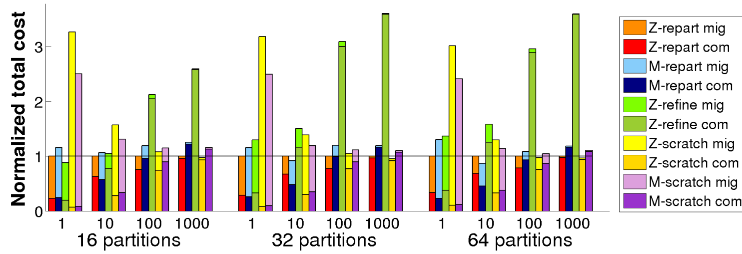

Fig. 1.10 Normalized total cost for cage14 with perturbed data structure.

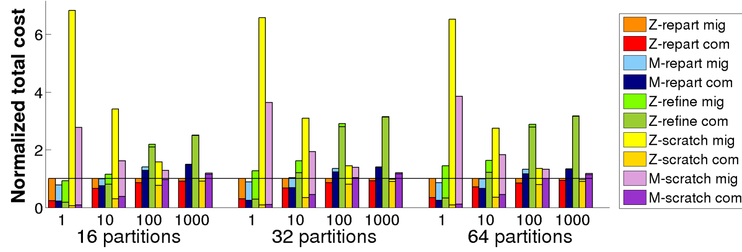

Fig. 1.11 Normalized total cost for cage14 with perturbed weights.

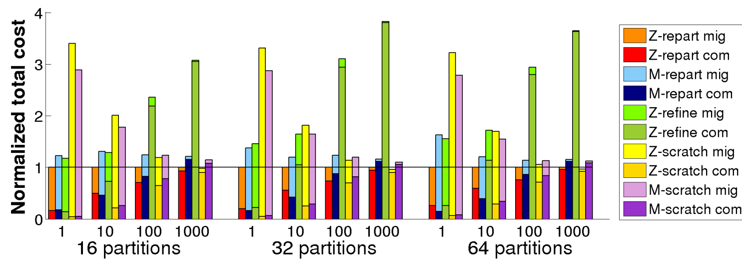

Fig. 1.12 Normalized total cost for average of five test graphs with perturbed data structure.

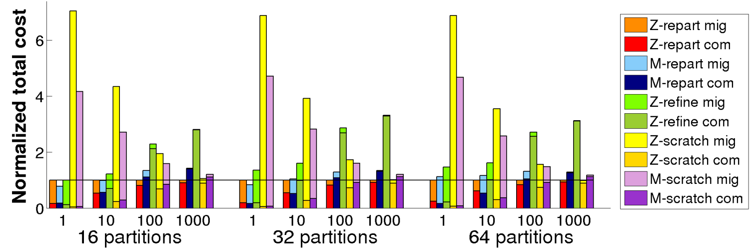

Fig. 1.13 Normalized total cost for average of five test graphs with perturbed weights.

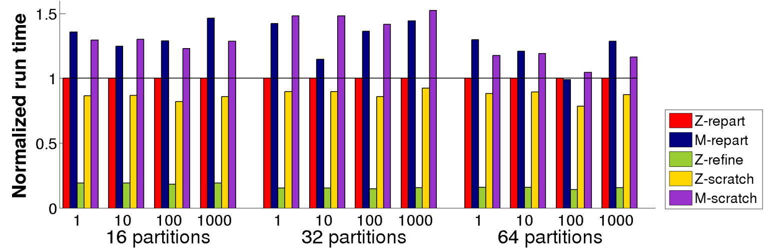

Fig. 1.14 Normalized run time with perturbed data structure for xyce680s.

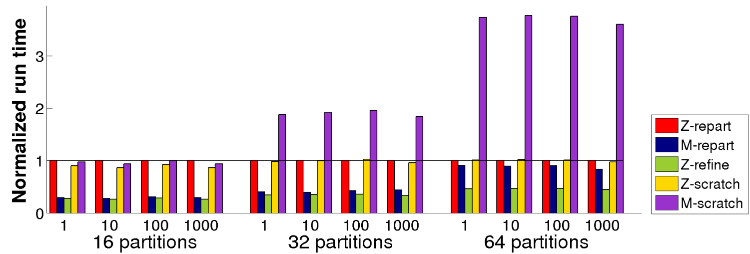

Fig. 1.15 Normalized run time with perturbed data structure for 2DLipid.

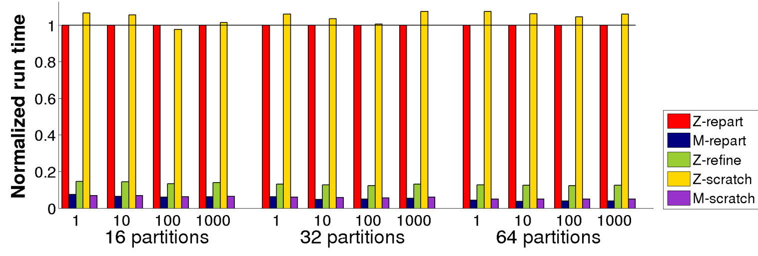

Fig. 1.16 Normalized run time with perturbed data structure for cage14.

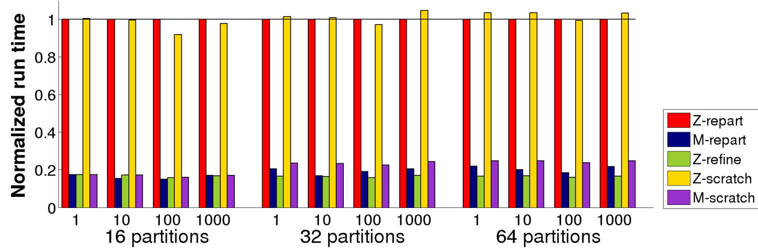

Fig. 1.17 Normalized run time for average of five test graphs with perturbed data structure.

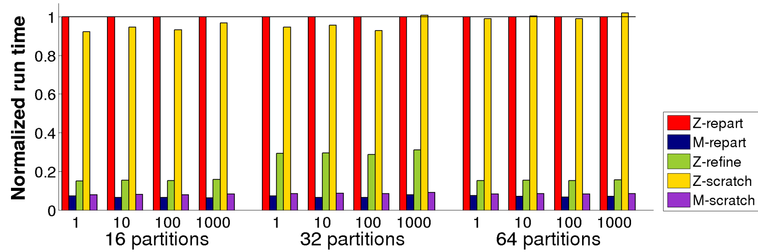

Fig. 1.18 Normalized run time for average of five test graphs with perturbed weights.

1.6 CONCLUSION

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 partitioner generally required more time than ParMETIS, mostly due to the greater richness of the hypergraph model. The full benefit of hypergraph partitioning is realized on unsymmetric and non-square problems that cannot be represented easily with graph models. So that we could provide comparisions with graph repartitioners, we restricted our tests here to square, symmetric problems; unsymmetric and non-square problems have been studied elsewhere [7, 12]. 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 to 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.

REFERENCES

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

2. Cevdet Aykanat, B. Barla Cambazoglu, Ferit Findik, and Tahsin 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.

3. Erik Boman, Karen Devine, Lee Ann Fisk, Robert Heaphy, Bruce Hendrickson, Courtenay Vaughan, Umit Catalyurek, Doruk Bozdag, and William Mitchell. *Zoltan 2.0: Data Management Services for Parallel Applica-*

tions; User's Guide. Sandia National Laboratories, Albuquerque, NM, 2006. Tech. Report SAND2006-2958 `http://www.cs.sandia.gov/Zoltan/ug_html/ug.html`.

4. 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.

5. B. Barla Cambazoglu and Cevdet 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.

6. Umit V. Catalyurek, Erik G. Boman, Karen D. Devine, Doruk Bozdağ, Robert Heaphy, and Lee Ann Fisk. Dynamic load balancing for adaptive scientific computations via hypergraph partitioning. In *Proceedings of 21st International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2007. to appear.

7. Ü. 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.

8. Ü. 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/$\sim$umit/software.htm`, 1999.

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

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

11. Karen Devine, Erik Boman, Robert Heaphy, Bruce Hendrickson, and Courtenay Vaughan. Zoltan data management services for parallel dynamic applications. *Computing in Science and Engineering*, 4(2):90–97, 2002.

12. K.D. Devine, E.G. Boman, R.T. Heaphy, R.H. Bisseling, and U.V. Catalyurek. Parallel hypergraph partitioning for scientific computing. In *Proc. of 20th International Parallel and Distributed Processing Symposium (IPDPS'06)*. IEEE, 2006.

13. 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.

14. 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.

15. J.E. Flaherty, R.M. Loy, M.S. Shephard, B.K. Szymanski, J.D. Teresco, and L.H. 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.

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

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

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

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

20. 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.

21. Mohammad Ashraf Iqbal and Shahid H. Bokhari. Efficient algorithms for a class of partitioning problems. *IEEE Trans. Parallel Distrib. Syst.*, 6(2):170–175, 1995.

22. 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.*

23. 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.

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

25. 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.

26. 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`.

27. George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. Multilevel hypergraph partitioning: application in VLSI domain. In *Proc. 34th Design Automation Conf.*, pages 526 – 529. ACM, 1997.

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

29. B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49(2):291–307, February 1970.

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

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

32. Chao-Wei Ou and Sanjay Ranka. Parallel incremental graph partitioning using linear programming. Technical report, Syracuse University, Syracuse, NY, 1992.

33. Kirk Schloegel, George Karypis, and Vipin Kumar. Multilevel diffusion algorithms for repartitioning of adaptive meshes. *J. Parallel Distrib. Comput.*, 47(2):109–124, 1997.

34. Kirk Schloegel, George Karypis, and Vipin Kumar. A unified algorithm for load-balancing adaptive scientific simulations. In *Proc. Supercomputing*, Dallas, 2000.

35. Kirk Schloegel, George Karypis, and Vipin Kumar. Wavefront diffusion and LMSR: Algorithms for dynamic repartitioning of adaptive meshes. *IEEE Trans. Parallel Distrib. Syst.*, 12(5):451–466, 2001.

36. D. G. Schweikert and B. W. Kernighan. A proper model for the partitioning of electrical circuits. In *Proceedings of the 9th ACM/IEEE Design Automation Conference*, pages 57–62, 1972.

37. James D. Teresco, Mark W. Beall, Joseph E. Flaherty, and Mark S. Shephard. A hierarchical partition model for adaptive finite element computation. *Comput. Methods Appl. Mech. Engrg.*, 184:269–285, 2000.

38. Aleksandar Trifunovic and William 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.

39. Rafael Van Driessche and Dirk 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.
40. Brendan Vastenhouw and Rob H. Bisseling. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM Review*, 47(1):67–95, 2005.
41. C. Walshaw, M. Cross, and M. Everett. Parallel dynamic graph-partitioning for adaptive unstructured meshes. *J. Par. Dist. Comput.*, 47(2):102–108, 1997.
42. Chris Walshaw. *The Parallel JOSTLE Library User's Guide, Version 3.0.* University of Greenwich, London, UK, 2002.
43. Marc Willebeek-LeMair and Anthony P. Reeves. Strategies for dynamic load balancing on highly parallel computers. *IEEE Trans. Parallel Distrib. Syst.*, 4(9):979–993, 1993.
44. R. Williams. Performance of dynamic load balancing algorithms for unstructured mesh calculations. *Concurrency*, 3:457–481, October 1991.
45. Zoltan: Data management services for parallel applications. `http://www.cs.sandia.gov/Zoltan/`.