Wavefront Diffusion and LMSR: Algorithms for Dynamic Repartitioning of Adaptive Meshes

Kirk Schloegel, *Member*, *IEEE, Computer Society*,
George Karypis, *Member*, *IEEE, Computer Society*, and
Vipin Kumar, *Fellow*, *IEEE*

Abstract—Current multilevel repartitioning schemes tend to perform well on certain types of problems while obtaining worse results for other types of problems. We present two new multilevel algorithms for repartitioning adaptive meshes that improve the performance of multilevel schemes for the types of problems that current schemes perform poorly while maintaining similar or better results for those problems that current schemes perform well. Specifically, we present a new scratch-remap scheme called *Locally-matched Multilevel Scratch-remap* (or simply LMSR) for repartitioning of adaptive meshes. LMSR tries to compute a high-quality partitioning that has a large amount of overlap with the original partitioning. We show that LMSR generally decreases the data redistribution costs required to balance the load compared to current scratch-remap schemes. We present a new diffusion-based scheme that we refer to as *Wavefront Diffusion*. In Wavefront Diffusion, the flow of vertices moves in a wavefront from overweight to underweight subdomains. We show that Wavefront Diffusion obtains significantly lower data redistribution costs while maintaining similar or better edge-cut results compared to existing diffusion algorithms. We also compare Wavefront Diffusion with LMSR and show that these provide a trade-off between edge-cut and data redistribution costs for a wide range of problems. Our experimental results on a Cray T3E, an IBM SP2, and a cluster of Pentium Pro workstations show that both schemes are fast and scalable. For example, both are capable of repartitioning a seven million vertex graph in under three seconds on 128 processors of a Cray T3E. Our schemes obtained relative speedups of between nine and 12 when the number of processors was increased by a factor of 16 on a Cray T3E.

Index Terms—Dynamic graph partitioning, multilevel diffusion, scratch-remap, wavefront diffusion, LMSR, adaptive mesh computations.

1 Introduction

For large-scale scientific simulations, the computational requirements of techniques relying on globally refined meshes become very high, especially as the complexity and size of the problems increase. By locally refining and derefining the mesh either to capture flow-field phenomena of interest [1] or to account for variations in errors [32], adaptive methods make standard computational methods more cost effective. One such example is numerical simulations for improving the design of helicopter blades [1]. (See Fig. 1.) Here, the finite-element mesh must be extremely fine around both the helicopter blade and in the vicinity of the sound vortex that is created by the rotation of the blade in order to accurately capture flow-field phenomena of interest. It should be coarser in other regions of the mesh for maximum efficiency. As the simulation progresses, neither the blade nor the sound vortex remain stationary. Therefore, the new regions of the mesh that these enter need to be refined, while those regions that are no longer of key interest should be derefined. The efficient execution of these simulations on high-performance parallel computers requires redistribution of the mesh elements across the processors because these dynamic adjustments to the mesh result in some processors having significantly more (or less) work than others. Similar issues also exist for problems in which the amount of computation associated with each mesh element changes over time [9]. For example, in particles-in-cells methods that advect particles through a mesh, large temporal and spatial variations in particle density can introduce substantial load imbalance.

Mapping of mesh-based computations onto parallel computers is usually computed by using a graph partitioning algorithm. In the case of adaptive finite-element methods, the graph either corresponds to the mesh obtained after adaptation or to the original mesh with the vertex weights adjusted to reflect error estimates. In the case of particles-in-cells simulations, the graph corresponds to the original mesh with the vertex weights adjusted to reflect the particle density. We will refer to this as the *adaptive graph partitioning problem* to differentiate it from the static graph partitioning problem that arises when the computation remains fixed. Adaptive graph partitioning shares most of the requirements and characteristics of static partitioning (i.e., compute a partitioning such that each subdomain contains a roughly equal amount of vertex weight and such that the edge-cut

• *The authors are with the Department of Computer Science and Engineering, University of Minnesota, Army HPC Research Center, 4-192 EE/CS Building, 200 Union St., S.E., Minneapolis, MN 55455. E-mail: kirk, karypis, kumar@cs.umn.edu.*

Manuscript received 4 Nov. 1998; revised 30 Aug. 2000; accepted 6 Nov. 2000.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 108172.

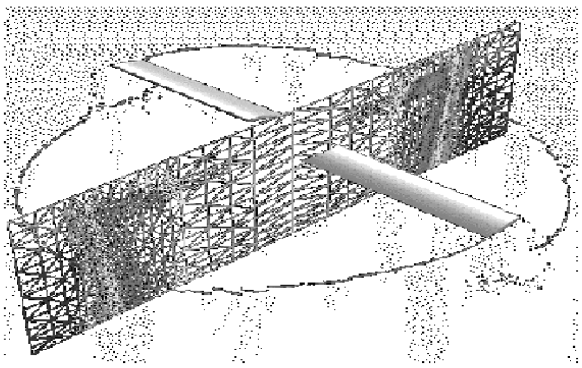

Fig. 1. A helicopter blade rotating through a mesh. As the blade spins, the mesh is adapted by refining it in the regions that the blade has entered and derefining it in the regions that are no longer of interest. (Figure provided by Rupak Biswas, NASA Ames Research Center.)

is minimized), but has an additional minimization objective. That is, the amount of data that needs to be redistributed among the processors in order to balance the load should also be minimized [38].

Recently, scratch-remap [29], [42] and diffusion-based [36], [48] adaptive partitioning techniques have been developed that are able to quickly compute high-quality repartitionings while minimizing the data redistribution costs for large classes of problems. However, there exist classes of problems for which each of these types of schemes perform poorly. Scratch-remap schemes tend to perform poorly when mesh adaptation is slight to moderate or when mesh adaptation is distributed throughout the mesh. Diffusion-based schemes tend to produce poor results when a high degree of adaptation occurs in localized areas of the mesh.

1.1 Our Contributions

This paper focuses on areas of improvement for scratch-remap and diffusion-based repartitioning schemes. We present a new scratch-remap scheme called *Locally-matched Multilevel Scratch-remap* (or simply LMSR). The LMSR scheme tries to compute a repartitioning that has a high overlap with the original partitioning. We show that LMSR decreases the data redistribution costs required to balance the load compared to current scratch-remap schemes for a wide range of problems. We present a new diffusion-based scheme that we refer to as *Wavefront Diffusion*. In Wavefront Diffusion, the flow of vertices moves in a wavefront from overweight to underweight subdomains. We show that Wavefront Diffusion obtains significantly lower data redistribution costs while maintaining similar or better edge-cut results compared to existing diffusion algorithms. We also compare Wavefront Diffusion with LMSR and show that these schemes provide a trade-off between edge-cut and data redistribution costs for a wide range of problems. Finally, we show that both schemes are extremely fast and scalable. For example, both are capable of repartitioning a seven million vertex graph in under three seconds on 128 processors of a Cray T3E. Furthermore, our experimental results show that, for between eight and 128 processors of a Cray T3E and for the range of problems presented, our algorithms exhibit good scaled speedups. That is, they require similar run times as the number of processors is increased by the same factor as the problem size.

The rest of this paper is organized as follows: Section 2 gives definitions and describes previous work in load balancing scientific simulations. Section 3 describes our LMSR algorithm. Section 4 describes our Wavefront Diffusion algorithm. Section 5 gives edge-cut, data redistribution and run time results for our LMSR and Wavefront Diffusion algorithms on a number of synthetic and real test sets. Section 6 gives conclusions.

2 Definitions and Background

This section gives definitions that will be used in the remainder of the paper and describes the multilevel graph partitioning paradigm as well as a number of load-balancing schemes for scientific simulations.

2.1 Definitions

When the vertices of a graph are used to represent the computational requirements of a scientific simulation, it is useful to assign them both *weight* and *size* [29], [41]. The weight of a vertex is its computational cost, while its size reflects its redistribution cost. Therefore, a repartitioner should attempt to balance the partitioning with respect to vertex weight while minimizing data redistribution with respect to vertex size. Depending on the representation of the data, the size and weight of a vertex may or may not be the same.

In our discussions, we refer to a $k-way$ partitioning as being composed of k disjoint *subdomains*. Each of these subdomains is composed of a number of vertices. The weight of a subdomain is the sum of the weights of its vertices. A subdomain is considered *overweight* if its weight is greater than the average subdomain weight times $1+\epsilon$, where ϵ is a user specified constant (and assumed to be 0.05 in this paper). Likewise, a subdomain is *underweight* if its weight is less than the average subdomain weight divided by $1+\epsilon$. A partitioning is *balanced* when none of its subdomains are overweight (although some may be underweight). Two subdomains are *neighbors* if there is at least one edge with incident vertices in each of the two subdomains. The subdomain in which a vertex is located originally is the *home* subdomain of that vertex.

Oliker and Biswas [29] studied various metrics for measuring data redistribution costs. They presented the metrics TOTALV and MAXV. TOTALV is defined as the sum of the sizes of the vertices that change subdomains as the result of repartitioning. TOTALV reflects the overall volume of communications needed to balance the load. MAXV is defined as the maximum of the sums of the sizes of those vertices that migrate into or out of any one subdomain as a result of repartitioning. MAXV reflects the maximum time needed by any one processor to send or receive data.

2.2 Multilevel Graph Partitioning

A class of partitioning algorithms has been developed [3], [5], [13], [15], [18], [22], [24], [27], [45] that is based on the multilevel paradigm. The multilevel paradigm consists of three phases: graph coarsening, initial partitioning, and uncoarsening/multilevel refinement. In

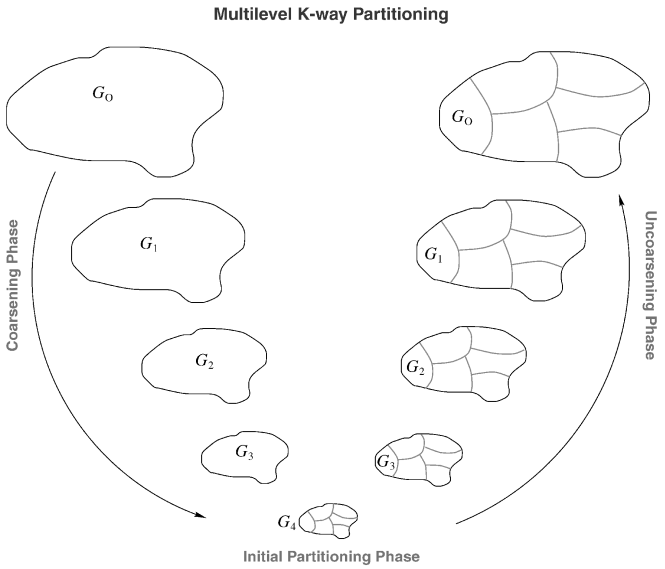

Fig. 2. The three phases of multilevel k-way graph partitioning. During the coarsening phase, the size of the graph is successively decreased. During the initial partitioning phase, a k-way partitioning is computed. During the multilevel refinement (or uncoarsening) phase, the partitioning is successively refined as it is projected to the larger graphs. G_0 is the input graph, which is the finest graph. $G_{i+}1$ is the next level coarser graph of G_i.G_4, is the coarsest graph.

the graph coarsening phase, a series of graphs is constructed by collapsing together selected vertices of the input graph in order to form a related coarser graph. A commonly used method for graph coarsening is to collapse together the pairs of vertices that form a matching. Vertex matchings can be computed by a number methods, such as heavy-edge matching [22], maximum weighted matching [12], and approximated maximum weighted matching [27]. The coarsened graph can then act as the input graph for another round of graph coarsening, and so on, until a sufficiently small graph is obtained. Computation of the initial partitioning is performed on the coarsest (and, hence, smallest) of these graphs and so is very fast. Finally, partition refinement is performed on each level graph, from the coarsest to the finest (i.e., original graph) using a KL/FM-type refinement algorithm [10], [26]. Fig. 2 illustrates the multilevel paradigm. A number of adaptive partitioning algorithms, including the ones presented in this paper, are also built upon the multilevel paradigm.

2.3 Previous Research

One approach that has been used for load-balancing of scientific simulations is to simply use a static partitioning scheme to compute a new partitioning from scratch each time that load balancing is required. We refer to this as simply *partitioning from scratch*. An alternate approach is to perturb the original partitioning just enough so as to balance it. In the context of graph partitioning, vertices are incrementally moved from subdomain to subdomain so as to balance the partitioning. Allowing this movement of vertices to occur only between neighboring subdomains helps to ensure that the edge-cut of the repartitioning is minimized and that the subdomains remain connected. Such schemes were initially motivated by diffusion algorithms used for load balancing independent tasks that are unevenly distributed among processors [2], [6], [8], [19], [20], [21], [49], [50] and so, we will refer to them as *diffusion*-based repartitioners.

2.3.1 Partitioning from Scratch

In principal, any partitioning scheme can be used to compute a repartitioning of an adapted graph from scratch. Geometric schemes [9], [31], [32], [33], [40], [44] have been extensively studied for this purpose. The advantage of these schemes is that they often produce repartitionings that are only minor perturbations of original partitionings (so long as the original partitioning was computed using the same method), especially if the degree of adaptation is not very high. The reason is because these schemes are deterministic and, hence, performing multiple trials on the same input graph result in identical partitionings. Note that a high overlap between subdomains of the original and the new partitionings leads to low data redistribution. The disadvantage of these schemes is that they tend to compute partitionings that are of worse quality than those produced by other methods [37], such as multilevel [3], [5], [13], [15], [18], [22], [24], [27], [45] and spectral [17], [34], [35] methods.

One notable exception is the algorithm described by Simon et al. [39] (and parallelized by Sohn [40]). This scheme uses a spectral method as a preprocessing step to embed the graph into a k-$dimensional$ space based on the connectivity of the graph [14]. The graph is then partitioned (and repartitioned) by a geometric method [28]. This scheme produces partitionings of similar quality to spectral partitioners while also having the advantage of low data redistribution costs described above. The disadvantage of this scheme is that mesh adaptation can only be represented by modifying the weights of the vertices of the graph. It cannot be represented by changing either the number of vertices or the connectivity of the graph. Otherwise, a new geometric embeding of the graph must be computed. This computation can be several orders of magnitude more costly than the time required for most dynamic repartitioning schemes.

Another way to compute a new partitioning from scratch is with a multilevel graph partitioner [3], [5], [13], [15], [18], [22], [24], [27], [45]. Such schemes produce high quality repartitionings (much higher than geometric schemes and slightly higher than spectral methods), but also tend to require large amounts of data redistribution even when very little (or no) adaptation takes place. This is because of two reasons. 1) Unlike geometric schemes, multiple trials from multilevel partitioners generally do not result in partitionings that are similar to each other. (Multiple trials from multilevel schemes do tend to result in partitionings of similar quality, however.) 2) Even when it is the case that the two partitionings are similar to each other, the labels of the corresponding subdomains on each partitioning might be different.

When the partitionings are similar to each other, but the subdomains are not labeled consistently, the data redistribution costs can be substantially reduced by intelligently remapping the subdomain labels of the new partitionings to those of the original partitionings [29], [42]. We use the term

scratch-remap for repartitioners that compute a new partitioning from scratch and then remap the new subdomain labels back to the original partitioning in order to reduce the data redistribution costs. The following method to compute a partition remapping is based on the scheme presented by Sohn and Simon.

1. Construct a similarity matrix, S, of size $k \times k$. A similarity matrix is one in which the rows represent the subdomains of the original partitioning, the columns represent the subdomains of the new partitioning, and each element, S_{qr}, represents the sum of the sizes of the vertices that are in subdomain q of the original partitioning and in subdomain r of the new partitioning.
2. Select k elements such that every row and column contains exactly one selected element and such that some objective is optimized. For example, Oliker and Biswas [29] describe remapping algorithms that attempt to minimize the TOTALV or the MAXV. (They showed that a fast greedy scheme for minimizing TOTALV generally results in good remappings for various application graphs.)
3. For each element S_{qr} selected, rename subdomain r to subdomain q on the remapped partitioning.

Such remapping is particularly interesting in the context of multilevel graph partitioners because these schemes already provide very good edge-cuts while requiring large amounts of data redistribution. We focus on such schemes in Section 3.

2.3.2 Diffusion-Based Repartitioners

Diffusion-based repartitioners attempt to minimize data redistribution even further than scratch-remap schemes by using the original partitioning as an input and perturbing it minimally so as to balance it. Any diffusion-based repartitioning scheme needs to address two questions: 1) *How much work should be transferred between subdomains?* and 2) *Which specific vertices should be transferred?* The answer to the first question tells us how to balance the partitioning, while the answer to the second tells us how to minimize the edge-cut as we do this. Schemes for determining how much work to transfer between subdomains can be grouped into two categories. We refer to diffusion schemes in which the exchange of work among the subdomains is based only upon their respective work loads (and not upon the loads of distant subdomains) as *local* diffusion algorithms [4], [36]. In other schemes [11], [30], [36], [46], [47], [48], global views of the subdomain loads are used to balance the partitioning. We call these *global* diffusion schemes.

Most global diffusion schemes compute *flow solutions* [30], [36], [46], [48] that prescribe the amount of work to be moved between pairs of subdomains. Flow solutions are usually computed in order to optimize some objective. Ou and Ranka [30] present a global diffusion scheme that optimally minimizes the one-norm of the flow using linear programming. Hu et al. [21] present a method that optimally minimizes the two-norm of the flow.

The flow solution indicates how much vertex weight needs to be transferred between each pair of adjacent subdomains. The second problem is to determine exactly which vertices to move so as to minimize the edge-cut of the resulting partitioning. One possibility is to repeatedly transfer layers of vertices along the subdomain boundary until the desired amount of vertex weight has been transferred [11], [30], [44]. A more precise scheme is to move one vertex at a time across the subdomain boundary [46]. Another possibility is to perform diffusion in the multilevel context [36], [48]. Such schemes, called *multilevel diffusion* algorithms, perform both diffusion (for load balancing) and refinement (for improving the edge-cut) in the uncoarsening phase. Often, diffusion takes precedence on the coarse level graphs. Then, once the graph is balanced to a reasonable degree, the focus shifts to refinement in order to improve the edge-cut.

2.4 Which Approach Is Better?

Having two basic approaches for adaptive partitioning (i.e., either compute a new partitioning or balance the original partitioning) leads to the question of which is preferred. Unfortunately, neither type of scheme performs best in all cases. A diffusion scheme that attempts to minimally perturb the original partitioning will naturally perform well when the final solution is close to the original solution. However, when this is not the case, the diffusion process can result in globally nonoptimal partitionings. On the other hand, a scheme that constructs a new solution from scratch will tend to perform better when the original and final solutions are far apart. However, it will often result in excessive data redistribution when this is not the case. Results in [29], [38] support these observations. They have shown that diffusion-based schemes outperform scratch-remap schemes when diffusion is not required to propagate far in order to balance the graph. (This situation occurs for slightly imbalanced partitionings and for those in which imbalance occurs globally throughout the graph.) When diffusion is required to propagate over longer distances, scratch-remap schemes outperform diffusion-based repartitioners. (This occurs when partitionings are highly imbalanced in localized areas of the graph.)

Fig. 3 and Fig. 4 illustrate relatively simple examples for which either scheme performs poorly. In both of these figures, the size and weight of each vertex is one. The weight of each edge is also one. In Fig. 3a, there are 12 vertices and three subdomains. Therefore, every subdomain should contain four vertices in order for the partitioning to be balanced. However, the partitioning is imbalanced because subdomain 1 has seven vertices, while subdomain 2 has two and subdomain 3 has three. In Fig. 3b, the graph has been partitioned from scratch. None of the 12 vertices have been assigned to their home subdomains. Therefore, TOTALV is 12 and MAXV is seven. (Note, the shading of a vertex indicates its home subdomain.) In Fig. 3c, the subdomain labels from Fig. 3b have been remapped with respect to those in Fig. 3a. This has reduced the TOTALV from 12 to five and the MAXV from seven to three without affecting the edge-cut. In Fig. 3d, a diffusive process has been applied to balance the partitioning. That is, two boundary vertices have moved from the overweight subdomain 1 to the neighboring subdomain 2 and one vertex has moved from subdomain

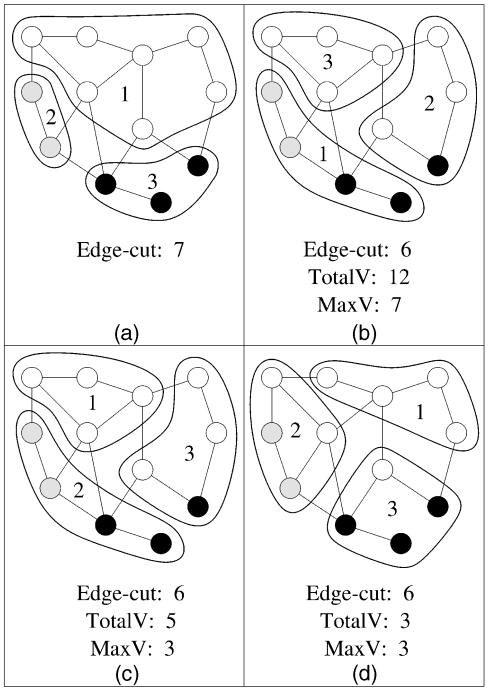

Fig. 3. An example of an imbalanced partitioning and various repartitioning schemes. The partitioning is imbalanced in (a). The graph is partitioned from scratch in (b) and this partitioning is remapped in (c). The partitioning from (a) is balanced by diffusion in (d).

1 to subdomain 3. The result is that both TOTALV and MAXV are three.

In Fig. 4, each of the four subdomains should contain five vertices in order for the partitioning to be balanced. The partitioning in Fig. 4a is imbalanced (because subdomain 1 has 12 vertices, while subdomain 2 has five, subdomain 3 has two, and subdomain 4 has one). In Fig. 4b, 19 out of 20 vertices have changed subdomains after the graph is partitioned from scratch. The MAXV here is 12. Fig. 4c shows a remapping in which TOTALV is 10 and MAXV is seven. Again, both the TOTALV and MAXV are reduced by remapping. In Fig. 4d, a diffusive process has been applied. Here, subdomain 1 was forced to export seven vertices to subdomain 2. This is because subdomain 2 is the only neighbor to subdomain 1 in Fig. 4a. Thus, even though subdomains 3 and 4 require additional vertex weight in order to balance the partitioning, they cannot receive vertices immediately from subdomain 1. Instead, a second iteration was required. In this iteration, subdomain 2 (which had temporarily become overweight) moved three vertices to subdomain 3 and four vertices to subdomain 4. The result is shown in Fig. 4d. In this case, TOTALV is 12 and MAXV is seven.

In Fig. 3, diffusion does a good job of balancing the partitioning while keeping both the edge-cut and data redistribution costs low. However, the scratch-remap scheme obtains a low edge-cut, but results in higher data redistribution costs. The reason is that the optimal repartitioning for the graph in Fig. 3a is quite similar to the original partitioning. Therefore, the diffusive repartitioner is able to balance the partitioning by moving only a few vertices. On the other hand, the repartitioning computed by the scratch-remap scheme is of high quality, but is somewhat structurally different from the original partitioning. Therefore, this scheme obtained a low edge-cut, but higher data redistribution costs, even after remapping.

In Fig. 4, the diffusive repartitioner results in both edge-cut and data redistribution costs that are higher than those of the scratch-remap repartitioner. Here, the diffusion of vertices is required to propagate to the underweight subdomains (3 and 4) by way of a transient subdomain (2). In general, as diffusion is required to propagate over longer distances to balance the partitioning, 1) well-shaped subdomains can become disturbed, increasing the edge-cut, and 2) many subdomains can be forced to export most or all of their original vertices, increasing data redistribution. Both of these effects can be seen in Fig. 4d. The scratch-remap repartitioner, on the other hand, performs well by computing a high-quality partitioning and then mapping it back to the original partitioning.

3 Locally Matched Multilevel Scratch-Remap

In this section, we present a number of enhancements to the scratch-remap scheme. We describe how restricting the coarsening phase of a multilevel graph partitioner to purely local matching can decrease the data redistribution costs by increasing the amount of overlap between subdomains of the original and new partitionings. Next, we describe a scheme that performs partition remapping in a multilevel context and explain how this scheme can be used to explicitly reduce the data redistribution costs while also improving the edge-cut during multilevel refinement.

3.1 Limitations of Scratch-Remap Schemes

Although partition remapping can reduce data redistribution costs (without affecting edge-cuts), scratch-remap schemes still tend to result in higher redistribution costs than schemes that attempt to balance the input partitioning by minimal perturbation. For example, if the newly adapted mesh is only slightly different from the original mesh, partitioning from scratch could produce a new partitioning that is still substantially different from the original, thus requiring a lot of data redistribution even after remapping. Fig. 5 illustrates an example of this. The partitioning in Fig. 5a is slightly unbalanced as the upper-right subdomain has five vertices, while the average subdomain weight is four. In Fig. 5b, the partitioning is balanced by moving only a single vertex from the upper-right subdomain to the lower-right subdomain. Therefore, both TOTALV and MAXV are one. Fig. 5c shows a new partitioning that has been computed from scratch and then optimally remapped to the partitioning in Fig. 5a. Despite optimal remapping, this repartitioning has a TOTALV of seven and a MAXV of two. Note that all three of the partitionings have similar edge-cuts.

The reason that the scratch-remap scheme does so poorly here with respect to data redistribution is because the information that is provided by the original partitioning is not utilized until the final remapping process. At this point, it is too late to avoid high data redistribution costs even if

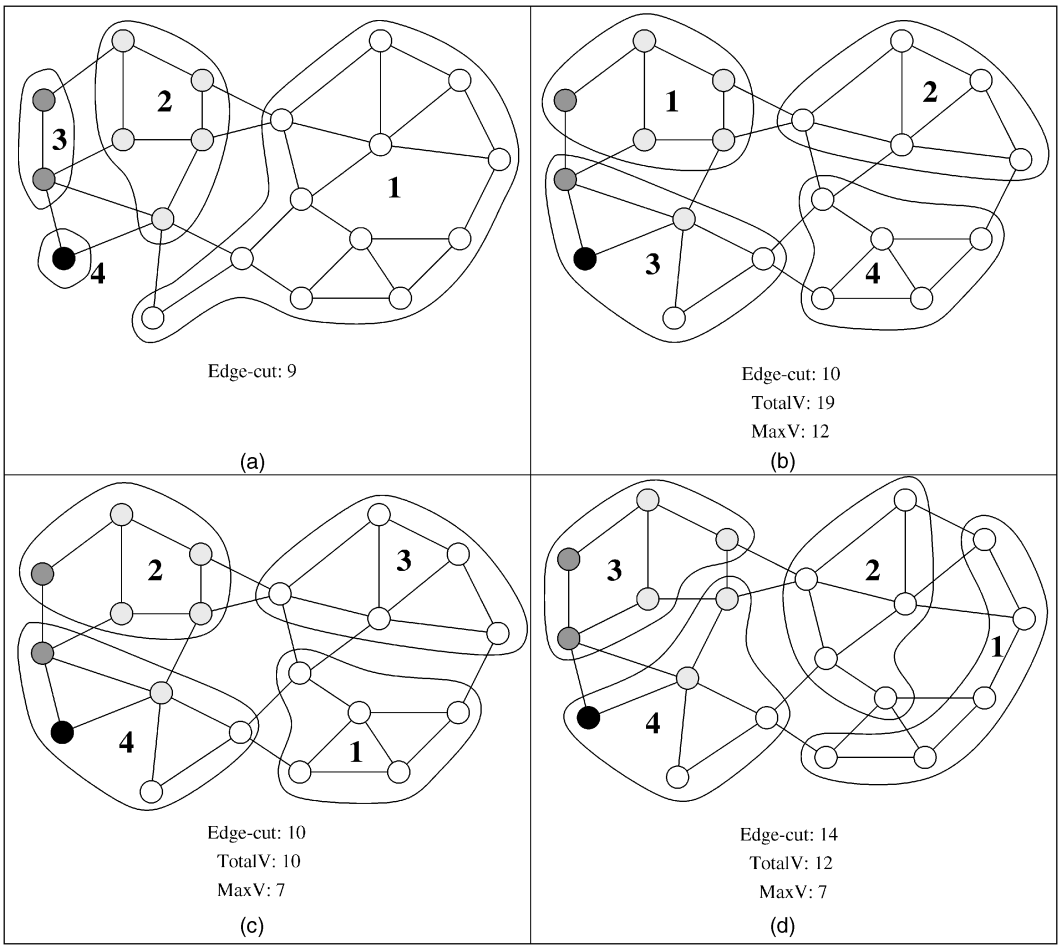

Fig. 4. An example of an imbalanced partitioning and various repartitioning schemes. The partitioning is imbalanced in (a). The graph is partitioned from scratch in (b) and this partitioning is remapped in (c). The partitioning from (a) is balanced by diffusion in (d).

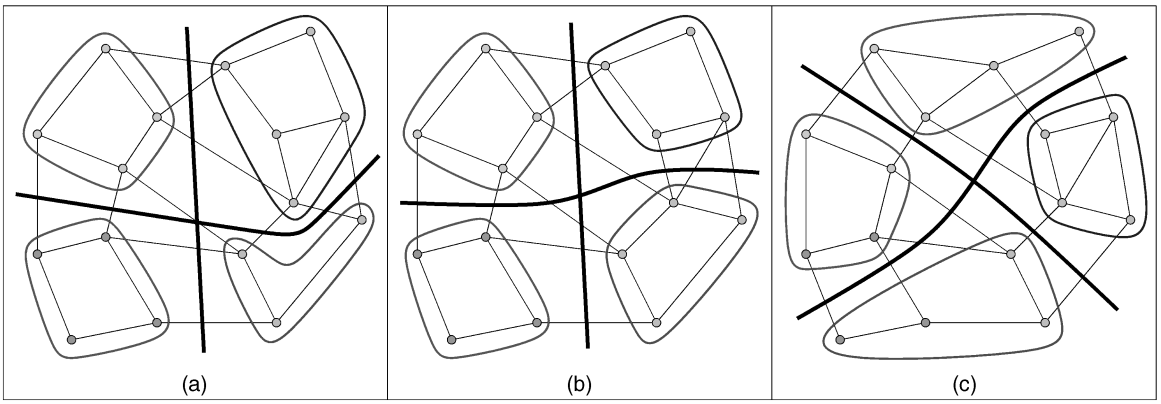

Fig. 5. An unbalanced partitioning and two repartitioning techniques. The partitioning in (a) is imbalanced. It is balanced by an incremental method in (b) and by a scratch-remap method in (c).

we compute an optimal remapping. Essentially, the problem in this example is that the partitioning in Fig. 5a is shaped like a "+" while the partitioning in Fig. 5c forms an "x." Both of these are of equal quality and, so, a static partitioning algorithm is equally likely to compute either one of these. However, we would like the partitioning algorithm used in a scratch-remap repartitioner to drive the computation of the partitioning toward that of the original partitioning whenever possible without affecting the quality. A scratch-remap algorithm can potentially do this if it is able to extract and use the information implicit in the original partitioning during the computation of the new partitioning.

3.2 Local Matching

The effectiveness of the greedy remapping scheme described in [29] is dependent on the nature of the similarity matrix. An ideal similarity matrix is one in which there is exactly one nonzero element in each row and column. This corresponds to the situation in which the new partitioning

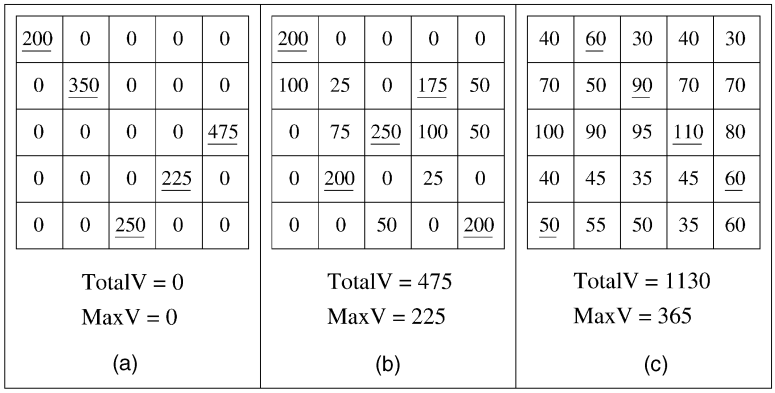

Fig. 6. Examples of ideal (a), good (b), and bad (c) overlap matrices.

is identical to the original partitioning except with regard to the subdomain labels. This is infeasible since the original partitioning is imbalanced and the new partitioning is balanced. A good similarity matrix is one in which most of the rows contain a small number of large values. The worst case similarity matrix is one in which all of the elements of a given row have identical values. This corresponds to the situation in which every subdomain of the new partitioning consists of an equal share of every subdomain of the original partitioning.

Fig. 6 illustrates different types of matrices. Fig. 6a is an example of an ideal similarity matrix. This is uninteresting because the new partitioning is not balanced. Fig. 6b is shows a similarity matrix constructed from two partitionings in which there are large amounts of overlap between the subdomains of the original and new partitionings. Fig. 6c shows an opposite case. Here, each of the subdomains of the newly computed partitioning share a roughly equal amount of vertex weight of each of the subdomains of the original partitioning. The underlined entries indicate the selected elements. While both of these remappings were computed using the greedy method described in [29], the TOTALV and MAXV are significantly lower for the case in Fig. 6b than for in Fig. 6c.

One way to increase the effectiveness of remapping is to bias the process of graph partitioning such that the situation illustrated in Fig. 6b will occur more frequently. Essentially, we would like to drive the computation of the new partitioning towards the original partitioning. This will result in large regions of overlap between a majority of subdomains of the original and new partitionings. Existing multilevel graph partitioners such as METIS [23] and Chaco [16] do not provide this bias.

It is possible to bias a multilevel graph partitioner toward the original partitioning during the graph coarsening phase. This can be done by restricting the matching of vertices to those that have the same home subdomain. The result is that vertices of each successively coarser graph correspond to regions within the same subdomain of the original partitioning. By the time the coarsest graph is constructed, every subdomain here consists of a relatively small number of well-shaped regions, each of which is a subregion of a single home subdomain. Therefore, when the new initial partitioning is computed on the coarsest graph, it will have a high degree of overlap with the original partitioning. Fig. 7 illustrates this point. It shows a single subdomain coarsened locally.

Another advantage of purely local matching is that the boundaries of the original partitioning remain visible on every level graph down to the coarsest graph. When two matched vertices are collapsed together, it is necessary to assign the new coarse vertex a home subdomain. If matching is purely local, then both of the matched vertices will always have the same home subdomain and, so, this assignment is straightforward. However, when global matching is performed, the home subdomains of the matched vertices may be different. If this is the case, the coarsened vertex can be assigned to only one of the matched vertices' home subdomains. Regardless of which subdomain is selected, the original partitioning becomes obscured on the coarse graph. Essentially, a portion of the subdomain boundary becomes hidden within a single vertex in these cases. Since local matching ensures that the original partitioning remains visible, even on the coarsest graph, in those portions of the graph that are relatively undisturbed by adaptation, the initial partitioning algorithm often has a tendency to select the same subdomain boundaries. This can have a positive effect on both edge-cut and data redistribution results.

3.3 Multilevel Scratch-Remap

A second potential improvement to the scratch-remap algorithm is to apply remapping on the coarsest graph after the new initial partitioning is computed, but before multilevel refinement is begun. This allows the partition refinement algorithm to explicitly minimize both edge-cuts and data redistribution costs during the uncoarsening phase. If partition remapping is performed only after multilevel refinement, the data redistribution cost cannot be minimized in this way as there is no way to accurately determine it. The reason is that the destination processors for the subdomains are finalized only after remapping.

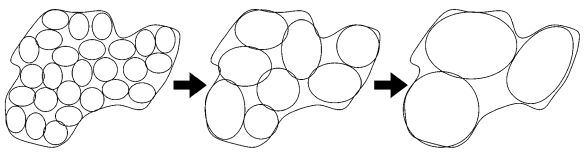

Fig. 7. A single subdomain from a series of successively coarser graphs constructed utilizing local matching.

3.4 TOTALV-*Aware* Refinement

In order to realize the benefit obtained by Multilevel Scratch-remap, the partition refinement algorithm must be modified in order to minimize not only the edge-cut, but also the data redistribution cost associated with load-balancing. Typically, refinement algorithms move vertices among the subdomains in order to either: 1) decrease the edge-cut while maintaining the balance constraint or 2) improve the balance while maintaining the edge-cut.

We have developed a refinement scheme that has a third objective: 3) decrease the TOTALV while maintaining the edge-cut and the balance constraint. In addition, the second objective above is changed to: 2) improve the balance while maintaining both the edge-cut and the TOTALV [36].

The second objective (of both schemes) will result in vertices being moved out of subdomains whose weights are above the average subdomain weight (even though these are not overweight with respect to the imbalance tolerance) if doing so does not increase the edge-cut. (Note that our scheme must not increase the TOTALV either in order to make these moves.) Such refinement moves have two effects. Not only will the partition balance be improved, but the edge-cut of the final partitioning will also tend to be improved. This is because, by moving a vertex out of a subdomain while maintaining the edge-cut, that subdomain becomes free to later accept another vertex from a neighboring subdomain that can improve the edge-cut.

If TOTALV is not considered (as is the case in current scratch-remap schemes), the refinement scheme is not restricted as to when it can make such moves. Our refinement algorithm, however, will only make these moves if they do not increase the TOTALV (i.e., the selected vertex is not moving out of its home subdomain). The result is that, as the TOTALV is further minimized, our refinement algorithm becomes more constrained as to the balance-improving moves that can be made. This can result in slightly worse edge-cuts for the final partitioning by a few percent. Essentially, the problem is that the two objectives of minimizing both the edge-cut and the data redistribution cost are in conflict here. On the positive side, our refinement algorithm can be easily modified to accept a user-supplied parameter that controls whether or not such moves should be made. This modification provides the user with control over the trade-off between the edge-cut and the data redistribution cost.

Our LMSR algorithm is a modification of the scratch-remap (SR) algorithm [29] that incorporates all three of these techniques (i.e., local matching, Multilevel Scratch-remap, and TOTALV-aware refinement).

4 WAVEFRONT DIFFUSION

Diffusion-based repartitioning schemes determine *how much* work should be moved between subdomains and *which* specific tasks to move in order to minimize interprocessor communications. A third question is: *When is the best time to move each task*? Existing diffusion schemes determine the order in which vertices are selected to be moved either by a greedy or a random scheme. Greedy schemes select the next vertex to move according to which will result in the lowest edge-cut. (We refer to this as *greedy global diffusion* (GGD).) Random schemes compute a random ordering of the vertices and examine them in this order. (We refer to this as *random-order global diffusion* (RO).) Both schemes often produce partitionings of similar quality. However, random schemes are more amenable to parallelization. The disadvantage of either of these schemes is that subdomains may simultaneously be both recipients and donors of vertices during diffusion. This means that subdomains are often forced to move out vertices before they have received all of the vertices that they are supposed to receive from their neighbors. Hence, these will have a limited choice for selecting good vertices to move out in order to minimize the edge-cut and data redistribution costs.

A better method to determine when to move vertices in order to satisfy the flow solution is to begin the diffusion of vertices at those subdomains that have no required flow of vertices into them. Then, the next iteration is begun on the set of subdomains whose required flow of vertices into them was satisfied during the previous iteration, and so on, until all of the subdomains are balanced. This method guarantees that all subdomains will contain the largest selection of vertices possible when it is their turn to export vertices. Thus, subdomains are able to select those vertices for migration that will best minimize the edge-cut and data redistribution cost.

A disadvantage of this scheme is that it requires more iterations to balance the graph than the RO and GGD schemes, and hence, is less scalable. We have implemented a modification that retains the spirit of this scheme while requiring fewer iterations to balance the partitioning. We maintain two arrays, $inflow$ and $outflow$, with one element per subdomain. $inflow\,[i]$ contains the sum of the vertex weight that subdomain i is required to receive in from other subdomains and $outflow\,[i]$ contains the sum of the vertex weight that subdomain i is required to send out to other subdomains. In each iteration, only those subdomains are allowed to move vertices out of their home subdomains for which the ratio of $outflow\,[i]/inflow\,[i]$ is above a threshold. All subdomains are allowed to move vertices that are not currently in their home subdomains. By setting the threshold to infinity, we obtain the algorithm described above. By setting the threshold to zero, we obtain random-order global diffusion. In our experiments, we set this threshold to be equal to the third highest $outflow\,[i]/inflow\,[i]$ ratio for $i = 0, 1, 2, \ldots, k-1$. This heuristic seems to strike a good balance between scalability and improved effectiveness for a moderate number of processors (e.g., up to 128 processors of a Cray T3E).

When the threshold is set to a suitably high number (as such), this scheme achieves an important effect. That is, vertices tend to hop across multiple subdomains to balance the partitioning. This reduces TOTALV and often MAXV as well. The reason is because any vertices that have moved from their overweight home subdomains in the first iterations can move in all subsequent iterations without increasing the TOTALV. Our experimental results indicate

TABLE 1
A Summary of the Parallel Implementations of the Four Repartitioning Schemes Compared in the Experimental Results Section

	SR	LMSR	RO	WF
Graph Coarsening				
Matching scheme	global	purely local		
Initial Partitioning				
Partitioning scheme	partition from scratch	partition from scratch	inherited	inherited
Pre-refinement remapping	no	yes	no	no
Pre-refinement diffusion	no	no	RO	WF
Multilevel Refinement				
Refinement scheme	edge-cut aware	edge-cut and TOTALV aware		
Post-refinement remapping	yes	not required		

TABLE 2
Characteristics of the Graphs Used to Create the Synthetic Test Sets

Graph	Num of Verts	Num of Edges	Description
144	144,649	1,074,393	3D mesh of a parafoil
auto	448,695	3,314,611	3D mesh of GM Saturn
*mdual*2	988,605	1,947,069	dual of a 3D mesh
*mrng*3	4,039,160	8,016,848	dual of a 3D mesh
*mrng*4	7,533,224	14,991,280	dual of a 3D mesh

that the potential for this effect increases as diffusion is required to propagate over further distances. In fact, we have obtained results (not presented in this paper) in which as much as 85 percent of all balancing moves are made by vertices outside of their home subdomains. This is tremendously beneficial in obtaining low data redistribution results.

We refer to this algorithm as *Wavefront Diffusion* (or simply WF), as the flow of vertices move in a wavefront from overweight to underweight subdomains. Essentially, it differs from existing global diffusion schemes (i.e., random-order and greedy) only in the order in which it selects vertices for movement. Note that, in order to increase the effectiveness of the WF algorithm, it can be performed in the multilevel context (i.e., after graph coarsening), similarly to other global diffusion schemes. In this paper, we focus on the case in which WF is performed only on the coarsest graph.

5 Experimental Results

In this section, we present experimental results comparing the edge-cuts, data redistribution costs, and run time results from the SR, LMSR, RO, and WF algorithms on synthetically generated test sets, as well as test sets derived from the simulations of a diesel engine and helicopter blade. Experimental results were performed on a Cray T3E, an IBM SP, and a cluster of Pentium Pro workstations connected via a Myrinet switch. Table 1 summarizes the algorithms compared in this section.

5.1 Test Sets Used for Experimental Evaluations

In this section, we describe the setup for the experiments that were performed in order to evaluate the repartitioning schemes.

5.1.1 Synthetic Test Sets

A set of synthetic experiments were constructed using five graphs derived from finite-element meshes. These graphs are described in Table 2. The synthetic test sets were constructed as follows: The sizes and weights of all of the vertices and the weights of all of the edges of the graphs from Table 2 were set to one. Next, two partitionings were computed for each graph, a 256-way partitioning and a k-*way* partitioning (where k is the number of processors used in the experiment). Three subdomains were selected from the 256-way partitioning. The weights of all of the vertices in these subdomains were set to α (where α was set to 2, 5, 10, 20, 30, and 60). This results in localized increases in vertex weight. Finally, each local edge was multiplied by the average weight of its two incident vertices raised to the $2/3$ power. For example, if $\alpha = 10$, then each vertex in the selected subdomains will be of weight 10. All of the other vertices will have weight one. The weight of the local edges inside of the selected subdomains will be $10^{.667} = 4.65$ (truncated down to four). The weight of a local edge with one incident vertex in a selected subdomain and one vertex outside of the subdomain will be $5.5^{.667} = 3.12$ (truncated down to three). Finally, the k-*way* partitioning was used as the original partitioning for the repartitioning algorithms. These experiments were designed to simulate adaptive mesh applications in which changes in the mesh are localized in nature. By modifying α, we can simulate slight to extreme levels of localized adaptation. If we set α low (e.g., two or five), then this results in experimental test sets in which the degree of adaptation is low. These are the type of problems for which scratch-remap schemes perform poorly. If we set α high (e.g., 20 through 60), this results in experimental test sets in which the degree of adaptation is high in localized regions of the graph. These

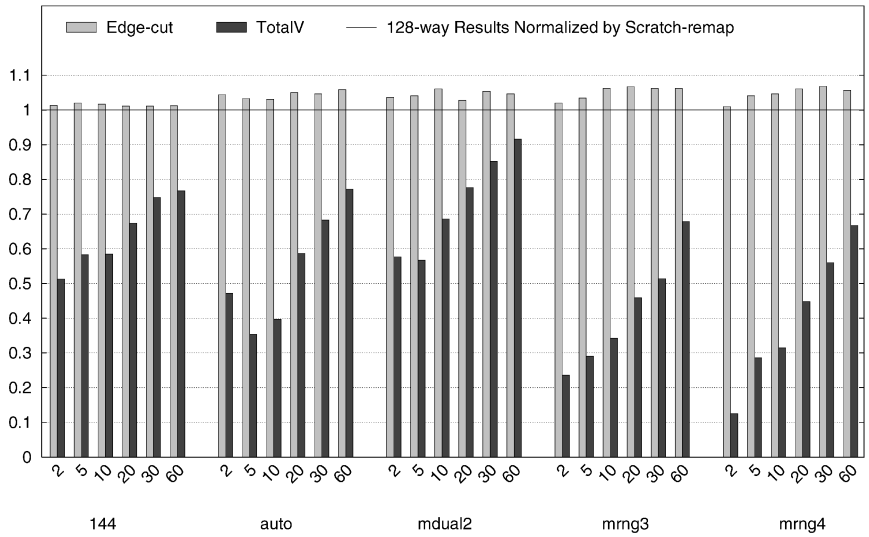

Fig. 8. A comparison of edge-cut and TOTALV results obtained from the SR and LMSR algorithms using synthetic test sets on a 128 processor of Cray T3E.

are the type of problems for which diffusion-based schemes perform poorly. Finally, we set k (and the number of processors) equal to 32, 64, and 128. However, we present only the 128-processor results here as the trends are similar.

5.1.2 Repartitioning for a Particles-in-Cells Simulation

Further experiments were performed on a problem set originating in a simulation of a diesel internal combustion engine.[1] This is a particles-in-cells computation. The mesh consists of 175,000 mesh elements. At first, no fuel particles are present in the combustion chamber. As the computation progresses, fuel particles are injected into the chamber at a single point and begin to spread out. Thus, they may enter regions of the mesh belonging to different processors. Load imbalance occurs as processors are required to track different numbers of particles.

5.1.3 Repartitioning for a Helicopter Blade Simulation

Finally, experiments were performed on a test set derived from three-dimensional mesh models of a rotating helicopter blade. These meshes are examples of applications in which high levels of adaptation occurs in localized areas of the mesh and so are quite difficult problems for diffusion-based repartitioners. They were provided by the authors of [29].

Here, the first of a series of six graphs, $G_1, G_2, \ldots, G_6$ was originally partitioned into eight subdomains with the multilevel graph partitioner implemented in PARMETIS [25]. The partitioning of graph G_1 acted as the original partitioning for graph G_2. Repartitioning the imbalanced graph, G_2, resulted in the experiment named *first* and the original partitioning for graph G_3. Similarly, the repartitioning of graph G_3 resulted in experiment *second*, the original partitioning for G_4, and so on, through experiment *fifth*.

1. These test sets were provided to us by Boris Kaludercic, HPC Product Coordinator, Computational Dynamics Ltd, London, England.

The last set of results is marked *SUM*. This is the sum of the raw scores of all five experiments and was included because these experiments consist of a series of repartitioning problems. That is, all of the repartitioning schemes used their own results from the previous experiments as inputs for the next experiment. Hence, only the first experiment in which all repartitioning schemes used the same input is directly comparable. However, by focusing on the sum of the results, we can obtain the average difference in repartitioning schemes across the five experiments.

5.2 Experimental Results for SR and LMSR

Fig. 8 compares the edge-cut and TOTALV results of the SR and LMSR algorithms on a number of test synthetic problems with varying degrees of adaptation. Six test problems were constructed for every graph by setting the value of α to 2, 5, 10, 20, 30, and 60. Fig. 8 gives the results on a 128-processor Cray T3E. For every experiment, the figure contains two bars. The first bar indicates the edge-cut obtained by LMSR normalized by the edge-cut obtained by SR. The second bar indicates the TOTALV obtained by LMSR normalized by the TOTALV obtained by SR. Therefore, a result below the 1.0 index line indicates that LMSR obtained better results than SR. All of the experiments resulted in partitionings that are balanced to within an imbalance tolerance of 5 percent.

For all of the experiments presented in Fig. 8, LMSR resulted in TOTALV results that are lower than those obtained by SR. The difference is as great as factor of 8 with α set to 2 on $mrng4$. Typically, LMSR resulted in less than half of the TOTALV costs compared to SR. Also, notice that the LMSR algorithm performed especially well when the degree of adaptation was low. If we consider the trend in the other direction, it appears that the TOTALV results of the LMSR and SR schemes will eventually converge for extremely high degrees of adaptation.

Fig. 8 shows that the edge-cuts of the two schemes are similar. However, LMSR results in generally worse edge-cuts by up to 8 percent compared to SR. This is because

TABLE 3
Edge-Cut, TOTALV, MAXV, and Runtime Results of the Adaptive Graph Partitioners for a Particles-in-Cells Simulation on a Cray T3E.

Scheme	Edge-cut	TOTALV	MAXV	Run Time
8 processors				
SR	7559	73400	22105	0.39
LMSR	8116	50627	21385	0.45
RO	11041	39941	20955	0.45
WF	11301	27721	13558	0.45
16 processors				
SR	15314	101983	12488	0.28
LMSR	14546	75698	12413	0.28
RO	20228	55894	11266	0.28
WF	19551	39091	10916	0.28
32 processors				
SR	21010	114229	7904	0.28
LMSR	21780	80177	6312	0.24
RO	27987	77602	6219	0.32
WF	27633	41380	5714	0.23
64 processors				
SR	30161	107719	4038	0.26
LMSR	28548	83296	3446	0.23
RO	39159	90495	3530	0.23
WF	39954	44200	2754	0.22

of two reasons. First, the global matching scheme used by the SR algorithm is more free than the local matching scheme to collapse vertices with very heavy edges between them. Collapsing such vertices, as shown in [22], can improve the effectiveness of multilevel refinement and, so, can result in lower edge-cuts. The second reason is because the two objectives of minimizing both the edge-cut and the data redistribution cost are often in conflict with one another (as discussed in Section 3). Here, the LMSR algorithm does so well at minimizing the TOTALV that the edge-cut suffers somewhat.

Note that MAXV results for the synthetic test sets are not presented. This is because these were usually within 10 percent of each other. The reason is that, due to the nature of the test sets (i.e., localized adaptations), the MAXV was dominated by the amount of vertex weight that was required to be moved out of the most overweight subdomain. Therefore, no scheme was able to improve upon this while still balancing the partitioning. Selected run time results for these experiments are reported in Section 5.5.

Table 3 shows the results for each of the repartitioning schemes on the diesel combustion engine test sets for eight, 16, 32, and 64 processors. Here, we see the same general trends as in the experiments presented in Fig. 8. Once again, LMSR outperformed SR with respect to TOTALV, while obtaining similar edge-cut results. Table 3 shows that LMSR obtained somewhat better MAXV results than SR. The run times of the two schemes are similar.

Fig. 9 gives a comparison of the edge-cut, TOTALV, and MAXV results of the five helicopter blade experiments, (followed by the sum of these) for SR, LMSR, RO, and WF. The results obtained by LMSR, RO, and WF are normalized by those obtained by SR. Hence, a bar below the index line indicates that the corresponding algorithm obtained better results than those obtained by the SR algorithm.

Fig. 9 shows trends that are similar to those for the synthetic and diesel engine results. The two scratch-remap schemes obtained similar edge-cuts and MAXV results, while LMSR obtained somewhat better TOTALV results than SR. The LMSR algorithm obtained TOTALV results that are on average 20 percent less than of those obtained by SR.

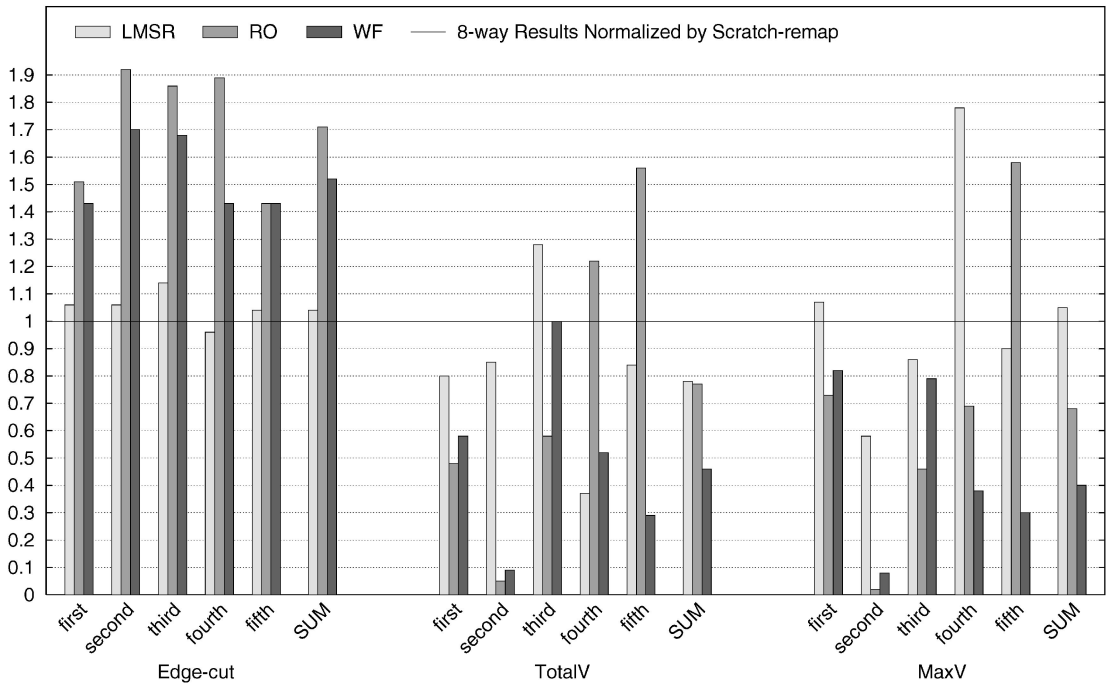

Fig. 9. A comparision of edge-cut, TOTALV, and MAXV results obtained by the SR, LMSR, RO, and WF algorithms on a series of application meshes of a helicopter blade rotating through space.

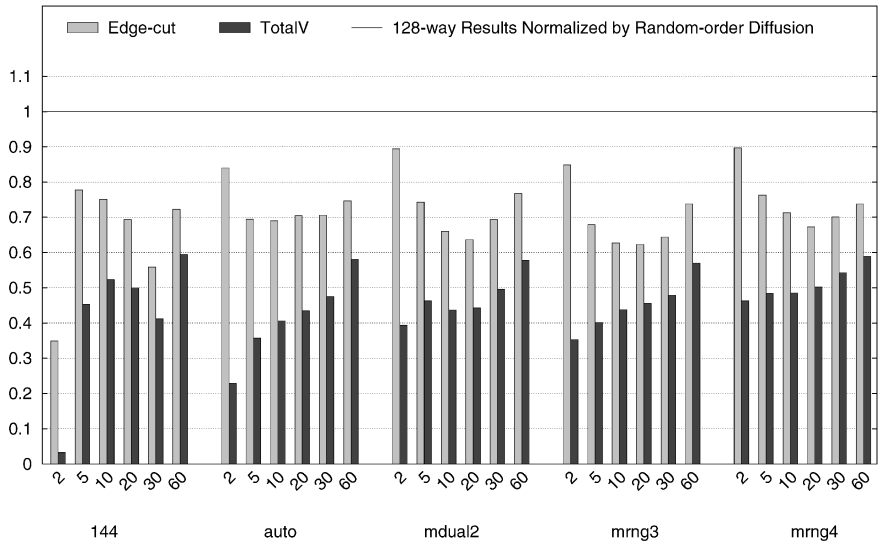

Fig. 10. A comparison of edge-cut and TOTALV results obtained from the RO and WF diffusion algorithms on synthetic test sets for a 128-processor of Cray T3E.

5.3 Experimental Results for WF and RO Diffusion

In order to test the effectiveness of our Wavefront Diffusion scheme, we repartitioned the experimental graphs described in Section 5.1 with both the RO and WF algorithms.

Fig. 10 presents the edge-cut and TOTALV results for the synthetic test sets on 128 processors of a Cray T3E. Again, MAXV results were quite similar for both of these algorithms and so are not presented here. Run time results for selected experiments are presented in Section 5.5. These experiments give the results of the multilevel diffusion phase only. That is, the graphs were coarsened identically (via local matching) and multilevel refinement was not conducted. This allows us to focus our attention on the diffusion algorithm and not on the effects of the multilevel paradigm. In Fig. 10, the bars indicate the edge-cut and TOTALV results obtained by the WF algorithm normalized by those obtained by the RO algorithm. Thus, a bar below the 1.0 index line indicates that the WF algorithm obtained better results than the RO algorithm. All of the experiments resulted in partitionings that are balanced to within a tolerance of 5 percent.

Fig. 10 shows that the WF algorithm obtained similar or better results across the board than the RO algorithm for both edge-cut and TOTALV. Typically, the WF algorithm obtained TOTALV results that are 40 to 60 percent of those obtained by RO and edge-cuts that are 60 to 90 percent of those obtained by the RO algorithm. These results show that the WF algorithm is more effective at computing high-quality repartitionings while minimizing the amount of TOTALV than the RO algorithm. Note that these figures compare the edge-cut results of the two schemes *prior to* multilevel refinement. The edge-cuts of the two schemes after multilevel refinement are usually comparable. This is because the effectiveness of multilevel refinement is able to make up the difference in edge-cut in many cases. The exception is when the partitioning is extremely imbalanced. In this case, WF tends to compute higher-quality partitionings than RO even after multilevel refinement.

Table 3 shows that WF resulted in significantly lower data redistribution costs (both TOTALV and MAXV) than RO. However, here, the edge-cuts are similar. This is because Table 3 (as well as Fig. 9) presents the results obtained after multilevel refinement. The run times of the two schemes are similar.

Fig. 9 shows similar trends. WF generally obtained better edge-cut, TOTALV, and MAXV results than RO. Specifically, WF obtained edge-cut results that are on average 20 percent of those obtained by RO and TOTALV and MAXV results that are on average 40 percent of those obtained by RO. These results confirm that WF is able to obtain lower edge-cuts and data redistribution costs than RO when mesh adaptation occurs to a high degree in localized regions.

5.4 Trade-Offs between LMSR and WF

Fig. 11 presents experimental results comparing the edge-cut and TOTALV results obtained by the WF and LMSR algorithms on the same experimental test sets described in Section 5.1. In these figures, the bars indicate the edge-cut (and TOTALV) results obtained by WF normalized by those obtained by LMSR. Thus, a bar below the 1.0 index line indicates that WF obtained better results than LMSR.

Fig. 11 shows that the Wavefront Diffusion algorithm obtained edge-cut results similar to or higher than the LMSR algorithm across the board. Specifically, the edge-cuts obtained by the WF algorithm are up to 42 percent higher than those obtained by the LMSR algorithm. Fig. 11 also shows that the WF algorithm was able to obtain TOTALV results that are significantly better than those obtained by the LMSR algorithm across the board. In particular, the WF algorithm obtained TOTALV results that are as little as 5 percent and generally less than half of those obtained by the LMSR algorithm.

Fig. 11 shows that, except for the case of very slightly imbalanced partitionings, there is a clear trade-off between the edge-cut and the TOTALV with respect to the two new

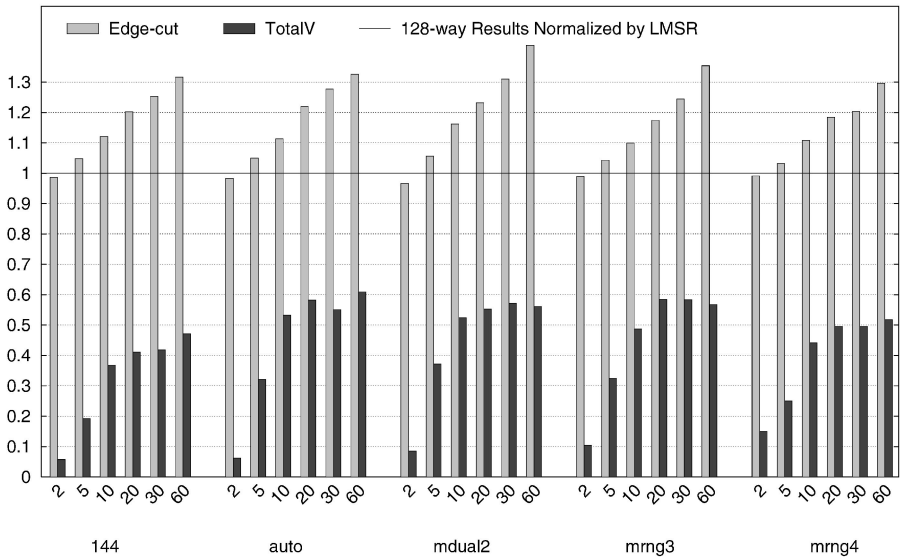

Fig. 11. A comparison of edge-cut and TOTALV results obtained from the LMSR and WF algorithms on synthetic test sets for a 128-processor of a Cray T3E.

TABLE 4
Parallel Runtimes of the Adaptive Graph Partitioners on a Cray T3E on the Synthetic Test Set with a $\alpha = 10$

Graph	Scheme	8 processors	16 processors	32 processors	64 processors	128 processors
144	SR	0.61	0.43	0.37	0.47	1.05
144	LMSR	0.63	0.40	0.31	0.33	0.74
144	RO	0.63	0.44	0.28	0.34	0.79
144	WF	0.64	0.43	0.31	0.34	1.16
auto	SR	2.29	1.22	0.70	0.50	0.61
auto	LMSR	2.45	1.25	0.67	0.62	0.48
auto	RO	2.47	1.30	0.73	0.61	0.55
auto	WF	2.47	1.28	0.70	0.55	0.59
*mdual*2	SR	2.84	1.43	0.83	0.58	0.69
*mdual*2	LMSR	3.22	1.61	0.85	0.68	0.51
*mdual*2	RO	3.25	1.62	0.92	0.74	0.59
*mdual*2	WF	3.25	1.79	1.17	0.75	0.58
*mrng*3	SR	9.69	4.92	2.65	1.48	1.56
*mrng*3	LMSR	11.01	5.54	2.86	1.59	1.17
*mrng*3	RO	11.04	5.58	2.93	2.02	1.42
*mrng*3	WF	11.03	5.58	2.91	1.67	1.32
*mrng*4	SR	18.34	9.24	4.83	2.70	1.97
*mrng*4	LMSR	21.11	10.52	5.34	2.79	1.79
*mrng*4	RO	21.28	10.58	5.40	2.95	1.88
*mrng*4	WF	21.28	10.60	5.46	2.84	2.22

TABLE 5
Parallel Runtimes of the Adaptive Graph Partitioners on an IMB SP on the Synthetic Test Sets for Graph $mrng3$ with $\alpha = 10$

Scheme	8 processors	16 processors	32 processors	64 processors
SR	10.58	5.12	2.95	1.72
LMSR	12.63	6.59	3.64	1.74
RO	12.89	6.95	3.66	1.99
WF	13.11	6.93	3.60	1.85

algorithms. That is, the LMSR algorithm minimizes the edge-cut at the cost of TOTALV, and Wavefront Diffusion minimizes TOTALV at the cost of edge-cut. For slightly imbalanced partitionings, WF is strictly better than LMSR, as it obtains similar edge-cuts and better TOTALV.

Comparing the results in both Table 3 and Fig. 9, we see a similar trade-off between edge-cut and data redistribution costs. Table 3 shows that the run times of the schemes are similar.

TABLE 6
Parallel Runtimes of the Adaptive Graph Partitioners on a Cluster of Pentium Pro Workstations Connected by a Myrinet Switch for Graph *Auto* with a $\alpha = 10$

Scheme	2 processors	4 processors	8 processors
SR	3.40	1.79	1.03
LMSR	3.57	1.83	1.03
RO	3.49	1.82	0.97
WF	3.59	1.81	0.99

5.5 Parallel Runtime Results

Table 4, Table 5, and Table 6 give run time results of selected experiments presented in the previous sections. Specifically, these tables show the times required for the SR, LMSR, RO, and WF algorithms to compute repartitionings for the series of synthetic experiments in which α is set to 10. Table 4 gives the results obtained for all five synthetic test graphs on up to 128 processors of a Cray T3E. Table 5 gives the results obtained for $mrng3$ with α set to 10 on up to 64 processors of an IBM SP. Table 6 gives the results obtained for $auto$ with α set to 10 on up to eight processors of a cluster of Pentium Pro workstations connected by a Myrinet switch. Table 4, Table 5, and Table 6 show that the repartitioning algorithms studied in this paper are very fast. For example, they are all able to compute a 128-way repartitioning of a 7.5 million vertex graph in under three seconds on 128 processors of a Cray T3E. These results also show that our parallel algorithms obtain similar run times as you increase the number of processors by the same factor as the problem size increases. That is, they exhibit good *scaled speedups*. For example, the time required to repartition $mdual2$ (with approximately one million vertices) on eight processors of the Cray T3E is similar to that of partitioning $mrng3$ (four million vertices) on 32 processors and $mrng4$ (seven and one-half million vertices) on 64 processors of the same machine. Finally, Table 4 shows that our algorithms obtained speedups of 11.8 (LMSR) and 9.6 (WF) when the number of processors was increased by a factor of 16 (from eight to 128) for the largest graph, $mrng4$. Note that all of the reported runtimes were obtained on nondedicated machines. Therefore, these results contain a certain amount of noise.

6 Conclusions

In this paper, we have presented two new repartitioning algorithms, LMSR and Wavefront Diffusion. We have shown that LMSR obtains lower data redistribution results than current scratch-remap repartitioners and that the difference between the schemes tends to increase when the degree of adaptation to the mesh is small. We have also shown that Wavefront Diffusion obtains lower edge-cut and data redistribution results than random-order diffusion repartitioners, especially when adaptation occurs to a high degree in localized regions of the mesh.

Compared against each other, these two schemes present a clear trade-off between edge-cut and data redistribution costs. That is, the Wavefront Diffusion algorithm minimizes the data redistribution required to balance the load, while LMSR minimizes the edge-cut of the repartitioning.

We have shown that our LMSR and WF schemes are extremely fast. (For example, they can compute repartitionings for a million-element mesh in about half of a second on 128 processors of a Cray T3E.) Other repartitioning methods, especially geometric schemes, can be somewhat faster. However, experimental results [7], [43] have shown that the time required to compute repartitionings can be significantly less than the time required to actually perform the data redistribution. This means that the ability of a repartitioning scheme to minimize the data redistribution is more important than the runtime of the scheme in many cases. None of the simple and fast methods [9], [31], [32], [33], [40], [44] explicitly minimize data redistribution costs, while we have shown that our schemes both obtain extremely low data redistribution costs and compute high-quality repartitionings.

The parallel repartitioning algorithms described in this paper are publically available in the PARMETIS [25] graph partitioning library at http://www.cs.umn.edu/~metis.

Acknowledgments

This work was supported by the US Department of Energy, contract number LLNL B347881, by the US National Science Foundation, grant CCR-9972519, by Army Research Office contracts DA/DAAG55-98-1-0441 and DA/DAAH04-95-1-0244, by the Army High Performance Computing Research Center cooperative agreement number DAAH04-95-2-0003/ contract number DAAH04-95-C-0008. The content of this work does not necessarily reflect the position or the policy of the government and no official endorsement should be inferred. Additional support was provided by the IBM Partnership Award and by the IBM SUR equipment grant. Access to computing facilities was provided by the AHPCRC, and the Minnesota Supercomputer Institute. Related papers are available via WWW: www.cs.umn.edu/~karypis.

References

[1] R. Biswas and R. Strawn, "A New Procedure for Dynamic Adaption of Three-Dimensional Unstructured Grids," *Applied Numerical Math.* vol. 13, pp. 437-452, 1994.
[2] J. Boillat, "Load Balancing and Poisson Equation in a Graph," *Concurrency: Practice and Experience,* vol. 2, pp. 289-313, 1990.
[3] T. Bui and C. Jones, "A Heuristic for Reducing Fill in Sparse Matrix Factorization," *Proc. Sixth SIAM Conf. Parallel Processing for Scientific Computing,* pp. 445-452, 1993.
[4] J. Castanos and J. Savage, "Repartitioning Unstructured Adaptive Meshes," *Proc. Int'l. Parallel and Distributed Processing Symp.,* 2000.
[5] J. Cong and M. Smith, "A Parallel Bottom-Up Clustering Algorithm with Applications to Circuit Partitioning in VSLI Design," *Proc. ACM/IEEE Design Automation Conf.,* pp. 755-760, 1993.
[6] G. Cybenko, "Dynamic Load Balancing for Distributed Memory Multiprocessors," *J. Parallel and Distributed Computing,* vol. 7, no. 2, pp. 279-301, 1989.
[7] K. Devine, B. Hendrickson, E. Boman, M. St. John, and C. Vaughan, "Design of Dynamic Load-Balancing Tools for Parallel Applications," *Proc. Int'l. Conf. Supercomputing,* 2000.
[8] R. Diekmann, A. Frommer, and B. Monien, "Efficient Schemes for Nearest Neighbor Load Balancing," *Parallel Computing,* vol. 25, pp. 789-812, 1999.

[9] P. Diniz, S. Plimpton, B. Hendrickson, and R. Leland, "Parallel Algorithms for Dynamically Partitioning Unstructured Grids," *Proc. Seventh SIAM Conf. Parallel Procedures,* 1995.

[10] C. Fiduccia and R. Mattheyses, "A Linear Time Heuristic for Improving Network Partitions," *Proc. 19th IEEE Design Automation Conf.,* pp. 175-181, 1982.

[11] J. Flaherty, R. Loy, C. Ozturan, M. Shephard, B. Szymanski, J. Teresco, and L. Ziantz, "Parallel Structures and Dynamic Load Balancing for Adaptive Finite Element Computation," *Applied Numerical Math,* vol. 26, pp. 241-263, 1998.

[12] H. Gabow, "Data Structures for Weighted Matching and Nearest Common Ancestors with Linking," *Proc. First Ann. ACM-SIAM Symp. Discrete Algorithms,* pp. 434-443, 1990.

[13] A. Gupta, "Fast and Effective Algorithms for Graph Partitioning and Sparse Matrix Reordering," *IBM J. Research and Development,* vol. 41, nos. 1/2, pp. 171-183, 1996.

[14] K. Hall, "An $r-Dimensional$ Quadratic Placement Algorithm," *Management Science,* vol. 17, no. 3, pp. 219-229, 1970.

[15] S. Hauck and G. Borriello, "An Evaluation of Bipartitioning Technique," *Proc. Conf. Advanced Research in VLSI,* 1995.

[16] B. Hendrickson and R. Leland, "The Chaco User's Guide, Version 2.0.," Technical Report SAND94-2692, Sandia Nat'l Laboratories, 1994.

[17] B. Hendrickson and R. Leland, "An Improved Spectral Graph Partitioning Algorithm for Mapping Parallel Computations," *SIAM J. Scientific Computing,* vol. 16, no. 2, pp. 452-469, 1995.

[18] B. Hendrickson and R. Leland, "A Multilevel Algorithm for Partitioning Graphs," *Proc. Supercomputing,* 1995.

[19] G. Horton, "A Multi-Level Diffusion Method for Dynamic Load Balancing," *Parallel Computing,* vol. 9, pp. 209-218, 1993.

[20] Y. Hu and R. Blake, "An Improved Diffusion Algorithm for Dynamic Load Balancing," *Parallel Computing,* vol. 25, pp. 417-444, 1999.

[21] Y. Hu, R. Blake, and D. Emerson, "An Optimal Migration Algorithm for Dynamic Load Balancing," *Concurrency: Practice and Experience,* vol. 10, pp. 467-483, 1998.

[22] G. Karypis and V. Kumar, "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs," *SIAM J. Scientific Computing,* vol. 20, no. 1, pp. 359-392, 1998.

[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.," Technical Report, Dept. of Computer Science and Eng., Univ. of Minnesota, 1998.

[24] G. Karypis and V. Kumar, "Multilevel $k-Way$ Partitioning Scheme for Irregular Graphs," *J. Parallel and Distributed Computing,* vol. 48, no. 1, 1998.

[25] G. Karypis, K. Schloegel, and V. Kumar, "PARMETIS: Parallel Graph Partitioning and Sparse Matrix Ordering Library," technical report, Dept. of Computer Science and Eng., Univ. of Minnesota, 1997.

[26] B. Kernighan and S. Lin, "An Efficient Heuristic Partitioning Graphs," *The Bell System Technical J.,* vol. 49, no. 2, pp. 291-307, 1970.

[27] B. Monien, R. Preis, and R. Diekmann, "Quality Matching and Local Improvement for Multilevel Graph-Partitioning," technical report, Univ. of Paderborn, 1999.

[28] B. Nour-Omid, A. Raefsky, and G. Lyzenga, "Solving Finite Element Equations on Concurrent Computers," *Am. Soc. Mechancial Eng.* A.K. Noor, ed. pp. 291-307, 1986.

[29] L. Oliker and R. Biswas, "PLUM: Parallel Load Balancing for Adaptive Unstructured Meshes," *J. Parallel and Distributed Computing,* vol. 52, no. 2, pp. 150-177, 1998.

[30] C. Ou and S. Ranka, "Parallel Incremental Graph Partitioning Using Linear Programming," *Proc. Supercomputing,* pp. 458-467, 1994.

[31] C. Ou, S. Ranka, and G. Fox, "Fast and Parallel Mapping Algorithms for Irregular and Adaptive Problems," *J. Supercomputing,* vol. 10, pp. 119-140, 1996.

[32] A. Patra and D. Kim, "Efficient Mesh Partitioning for Adaptive hp Finite Element Meshes," technical report, Dept. of Mechanical Eng., State University of New York, Buffalo, 1999.

[33] J. Pilkington and S. Baden, "Dynamic Partitioning of Non-Uniform Structured Workloads with Spacefilling Curves," technical report, Dept. of Computer Science and Eng., Univ. of California 1995.

[34] A. Pothen, H. Simon, and K. Liou, "Partitioning Sparse Matrices with Eigenvectors of Graphs," *SIAM J. Matrix Analysis and Applications,* vol. 11, no. 3, pp. 430-452, 1990.

[35] A. Pothen, H. Simon, L. Wang, and S. Bernard, "Towards a Fast Implementation of Spectral Nested Dissection," *Proc. Supercomputing,* pp. 42-51, 1992.

[36] K. Schloegel, G. Karypis, and V. Kumar, "Multilevel Diffusion Schemes for Repartitioning of Adaptive Meshes," *J. Parallel and Distributed Computing,* vol. 47, no. 2, pp. 109-124, 1997.

[37] K. Schloegel, G. Karypis, and V. Kumar, "Graph Partitioning for High Performance Scientific Simulations," *CRPC Parallel Computing Handbook,* Morgan Kaufmann, 2000.

[38] K. Schloegel, G. Karypis, V. Kumar, R. Biswas, and L. Oliker, "A Performance Study of Diffusive vs. Remapped Load-Balancing Schemes" *ISCA 11th Int'l Conf. Parallel and Distributed Computing Systems,* pp. 59-66, 1998.

[39] H. Simon, A. Sohn, and R. Biswas, "HARP: A Fast Spectral Partitioner," *Proc. Ninth ACM Symp. Parallel Algorithms and Architectures,* pp. 43-52, 1997.

[40] A. Sohn, "S-HARP: A Parallel Dynamic Spectral Partitioner," technical report, Dept. of Computer and Information Science, New Jersey Institute of Technology, 1997.

[41] A. Sohn, R. Biswas, and H. Simon, "Impact of Load Balancing on Unstructured Adaptive Grid Computations for Distributed-Memory Multiprocessors," *Proc. Eighth IEEE Symp. Parallel and Distributed Processing,* pp. 26-33, 1996.

[42] A. Sohn and H. Simon, "JOVE: A Dynamic Load Balancing Framework for Adaptive Computations on an SP-2 Distributed-Memory Multiprocessor," Technical Report 94-60, Dept. of Computer and Information Science, New Jersey Institute of Technology, 1994.

[43] N. Touheed, P. Selwood, P. Jimack, and M. Berzins, "A Comparison of Some Dynamic Load-Balancing Algorithms for a Parallel Adaptive Flow Solver," *Parallel Computing,* vol. 26, no. 1, pp. 535-554, 2000.

[44] A. Vidwans, Y. Kallinderis, and V. Venkatakrishnan, "Parallel Dynamic Load-Balancing Algorithm for Three-Dimensional Adaptive Unnstructured Grids," *AIAA J.,* vol. 32, pp. 497-505, 1994.

[45] C. Walshaw and M. Cross, "Mesh Partitioning: A Multilevel Balancing and Refinement Algorithm," *SIAM J. Scientific Computing,* vol. 22, no. 1, pp.63-80, 2000.

[46] C. Walshaw, M. Cross, and M. Everett, "Dynamic Mesh Partitioning: A Unified Optimisation and Load-Balancing Algorithm," Technical Report 95/IM/06, Centre for Numerical Modelling and Process Analysis, Univ. of Greenwich, 1995.

[47] C. Walshaw, M. Cross, and M. Everett, "Mesh Partitioning and Load-Balancing for Distributed Memory Parallel Systems," *Proc. Parallel and Distbuted Computing for Computer Mechanics,* 1997.

[48] C. Walshaw, M. Cross, and M. Everett, "Parallel Dynamic Graph Partitioning for Adaptive Unstructured Meshes," *J. Parallel and Distributed Computing,* vol. 47, no. 2, pp. 102-108, 1997.

[49] J. Watts and S. Taylor, "A Practical Approach to Dynamic Load Balancing," *Trans. Parallel and Distributed Systems,* vol. 9, pp. 235-248, 1998.

[50] C. Xu and F. Lau, "The Generalized Dimension Exchange Method for Load Balancing in k-Ary ncubes and Variants," *J. Parallel and Distributed Computing,* vol. 24, pp. 72-85, 1995.

Kirk Schloegel has a PhD in computer science from the University of Minnesota. He is a research associate at the University of Minnesota. His research interests include graph partitioning, load balancing, and parallel computing. He is a member of the IEEE Computer Society and SIAM.

George Karypis has a PhD in computer science from the University of Minnesota. He is an assistant professor of computer science at the University of Minnesota. His research interests include data mining, bio-informatics, parallel computing, graph partitioning, and scientific computing. He is a member of the IEEE Computer Society and the ACM.

Vipin Kumar is the director of the Army High Performance Computing Research Center and professor of computer science at the University of Minnesota. His current research interests include high performance computing and data mining. Dr. Kumar serves on the editorial boards of *IEEE Concurrency*, *Parallel Computing*, the *Journal of Parallel and Distributed Computing*, and he has served on the editorial board of the *IEEE Transactions of Data and Knowledge Engineering* during 1993-1997. He is a fellow of the IEEE, a member of SIAM and ACM, and a fellow of the Minnesota Supercomputer Institute.