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

Kirk Schloegel, George Karypis, Vipin Kumar
Department of Computer Science and Engineering
Army HPC Research Center
University of Minnesota, Department of Computer Science

(kirk, karypis, kumar) @ cs.umn.edu

Abstract

Existing state-of-the-art schemes for dynamic repartitioning of adaptive meshes can be classified as either diffusion-based schemes or scratch-remap schemes. We present a new scratch-remap scheme called Locally-Matched Multilevel Scratch-Remap (or simply LMSR). The LMSR scheme tries to compute a partitioning that has a high overlap with the existing partitioning. We show that LMSR decreases the amount of vertex migration required to balance the graph compared to current scratch-remap schemes, particularly for slightly imbalanced graphs. We describe 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 domains. We show that Wavefront Diffusion obtains significantly lower vertex migration requirements while maintaining similar or better edge-cut results compared to existing diffusion algorithms, especially for highly imbalanced graphs. Furthermore, we compare Wavefront Diffusion with LMSR and show that the former scheme results in generally lower vertex migration requirements at the cost of lower quality edge-cuts. Our experimental results on parallel computers show that both schemes are highly scalable. For example, both are capable of repartitioning an eight million node graph in under three seconds on a 128-processor Cray T3E.

Keywords: Dynamic Graph Partitioning, Multilevel Diffusion, Scratch-Remap, Wavefront Diffusion, LMSR, Adaptive Mesh Computations

1 Introduction

Graph partitioning is a well-understood technique for mapping irregular mesh applications to parallel machines. For the growing class of scientific and engineering simulations that utilizes adaptive meshes to model the computation, however, static partitioning is not adequate to maintain a balanced workload among the processors. A graph repartitioning scheme that is able to balance the processor workload, while minimizing both the communications of the application and the amount of data migration necessary to balance the load, is a key component for the successful conduct of these applications.

*This work was supported by NSF CCR-9423082, by Army Research Office contracts DA/DAAG55-98-1-0441 and DA/DAAH04-95-1-0244, by Army High Performance Computing Research Center cooperative agreement number DAAH04-95-2-0003/contract number DAAH04-95-C-0008, the content of which 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 AHPCRC, Minnesota Supercomputer Institute. Related papers are available via WWW at URL: http://www.cs.umn.edu/~karypis

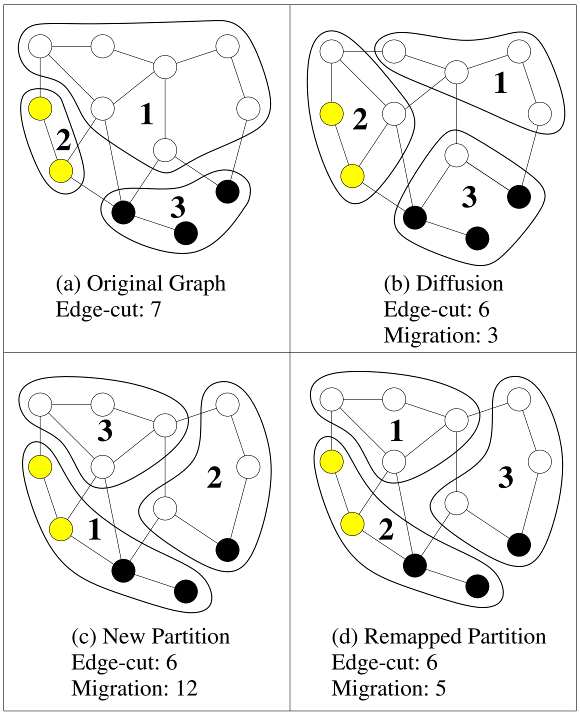

Figure 1: An example of an imbalanced partitioning (a), this partitioning balanced by diffusion (b), partitioning the graph from scratch (c), and remapping this partitioning (d).

Recently, various graph repartitioning techniques [1, 13, 15, 18] have been developed that can quickly compute high-quality repartitionings while minimizing the amount of data that needs to be migrated among processors for large classes of problems. These can be classified as either diffusion-based schemes [15, 18] or scratch-remap schemes [1, 13].

Diffusion-based Repartitioners

Diffusion-based repartitioners attempt to minimize the difference between the original (imbalanced) partitioning and the final repartitioning by making incremental changes in the partitioning to restore balance. Domains that are overweight in the original partitioning export vertices to adjacent domains. These, in turn, may become overweight and further export vertices to other domains in an effort to reach global equilibrium. How the flow of vertices among the domains is computed and which specific vertices within a domain are selected for migration depends on the details of the scheme.

Figures 1(a) and (b) give an example of a diffusive repartitioner. If we assume that the weight of each vertex is one, then every domain should contain four vertices in order for the partitioning to be balanced. However, the partitioning in Figure 1(a) is imbalanced because domain 1 has seven vertices while domain 2 has two and domain 3 has three. In Figure 1(b), a diffusive process has been applied to balance the partitioning. That is, two vertices have migrated from domain 1 to domain 2, and one vertex has migrated from domain 1 to domain 3. (Note, the shading of a vertex indicates the domain on the original partitioning of which that vertex was a member.)

Figures 2(a) and (b) give another example. Here, every domain should contain five vertices in order for the partitioning to be balanced. Again, the partitioning in Figure 2(a) is imbalanced because domain 1

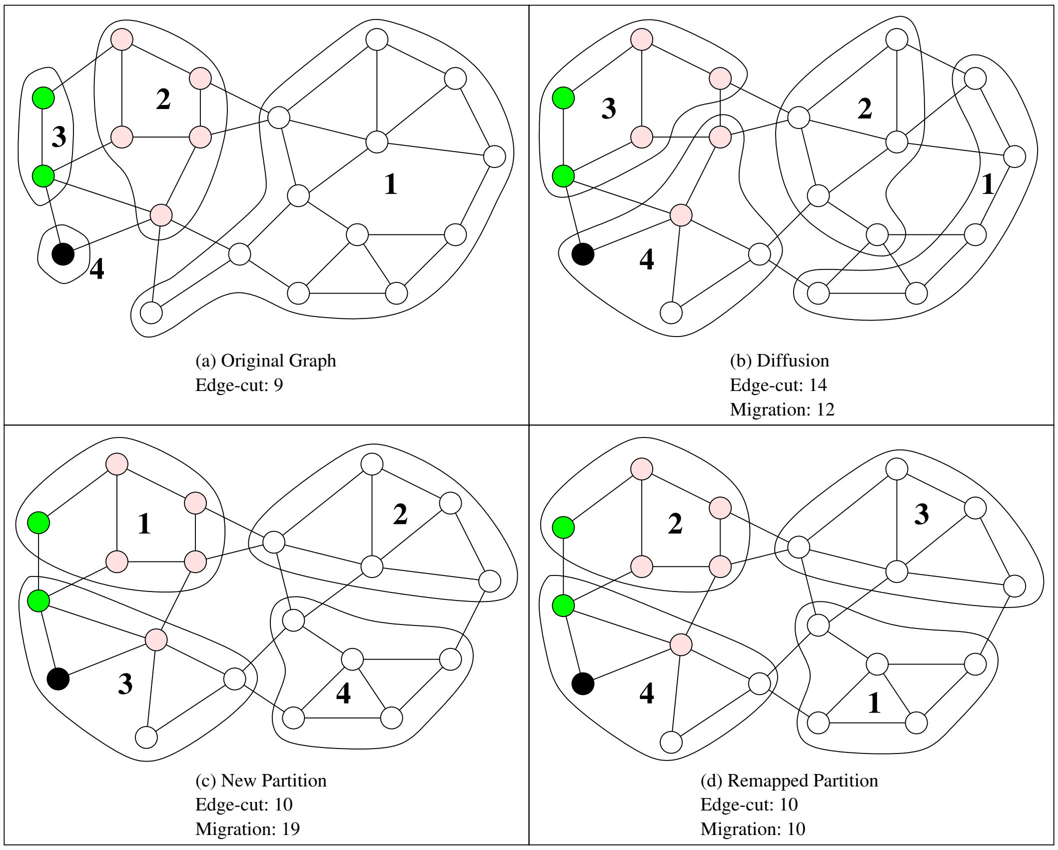

Figure 2: An example of an imbalanced partitioning (a), this partitioning balanced by diffusion (b), partitioning the graph from scratch (c), and remapping this partitioning (d).

has twelve vertices while domain 2 has five, domain 3 has two, and domain 4 has one. In Figure 2(b), a diffusive process has been applied. Here, domain 1 was forced to export seven vertices to domain 2. This was necessary because domain 2 is the only domain adjacent to domain 1 in Figure 2(a). Thus, even though domains 3 and 4 require additional vertex weight in order to balance the partitioning, they can not receive vertices immediately from domain 1 in this example. Instead, a second iteration of diffusion is required. Here, domain 2 (which had become overweight with the import of seven vertices from domain 1) was then able to migrate three vertices to domain 3 and four vertices to domain 4 in order to balance the partitioning. The result is shown in Figure 2(b).

Scratch-Remap Repartitioners

While diffusive repartitioners start at the original partitioning and attempt to balance it, scratch-remap repartitioners start with a newly computed (balanced) partitioning and attempt to minimize its difference from the original partitioning. That is, they utilize a graph partitioning algorithm to compute a balanced partitioning, and then compute a new labeling of the domains that minimizes the difference between the old and new partitionings.

Figures 1(a), (c), and (d) illustrate the scratch-remap process. As stated above, the partitioning in Figure 1(a) is imbalanced. In Figure 1(c), the graph has been partitioned again from scratch. Since, the new partitioning was computed without regard for the original partitioning, a large amount of vertex migration is required here. In this case, all 12 vertices must swap domains. However, by remapping the newly computed partitioning with respect to the original partitioning, the amount of vertex migration can be substantially reduced as shown in Figure 1(d). The number of vertices that are required to swap domains has now dropped from 12 to 5. Notice that since remapping only changes the labels of the domains and not the partitioning itself, the edge-cut is not affected. Thus, intelligent remapping can reduce the amount of data required to balance the graph while maintaining the quality of the edge-cut of the newly computed partitioning.

Figures 2(a), (c), and (d) give another example. In Figure 2(c), 19 out of 20 vertices are required to swap domains after the graph has been partitioned from scratch. Again, this vertex migration is reduced substantially once remapping takes place. Figure 2(d) shows a remapping that requires only 10 vertices to be migrated in order to realize the balanced partitioning.

Issues

Examining Figures 1 and 2 reveals strengths and weaknesses of each scheme. In Figure 1, diffusion does a very good job of balancing the partitioning while keeping both the edge-cut and the amount of vertex migration low. Here, however, the scratch-remap scheme obtains a low edge-cut, but results in higher vertex migration. The reason is that the optimal repartitioning for the graph in Figure 1(a) is quite similar to the original partitioning. Thus, the diffusive repartitioner, which inherits the old partitioning and then attempts to modify it as little as possible so as to meet the balance constraint, performs well here. (In fact, since the sole overweight domain (1) is adjacent to both of the underweight domains (2 and 3), diffusion repartitioning results in the optimally minimal amount of vertex migration required.) On the other hand, the scratch-remap repartitioner, which first computes a new initial partitioning and then attempts to minimize the difference between it and the original partitioning, is not able to obtain as low of vertex migration as the diffusion repartitioner.

In Figure 2, the diffusive repartitioner results in both edge-cut and vertex migration requirements that are higher than those of the scratch-remap repartitioner. Recall that here, diffusion of vertices is required to propagate to the underweight domains (3 and 4) by way of a transient domain (2), and that this is not the case for the example in Figure 1. The results of such propagations of diffusion are i) well-shaped domains are perturbed, tending to increase the edge-cut, and ii) transient domains are pulled into areas formerly belonging to overweight domains. This tends to increase the amount of vertex migration required, as transient domains end up with very few (or none) of their original vertices. Both of these effects can be seen in Figure 2(b). 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.

Results in [13, 16] support our observations from these two examples. They have shown that current 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 graphs and for those in which imbalance occurs globally throughout the graph. For these classes of problems, diffusion-based schemes result in less vertex migration than current scratch-remap schemes. At the same time, the quality of the edge-cut produced is similar between the two schemes (assuming that the original partitioning is of high quality). This is because diffusion-based schemes only minimally perturb the edge-cut here.

Graphs that are highly imbalanced in localized areas require diffusion to propagate over longer distances. For this class of problems, current diffusion-based repartitioners obtain similar or higher vertex migration results than scratch-remap schemes. Also, as the amount of vertex migration increases, so does the resulting edge-cut obtained by diffusion schemes. On the other hand, scratch-remap schemes are able to consistently produce high-quality partitionings, regardless of the weight characteristics of the graph. The result is that the edge-cuts obtained by scratch-remap repartitioners are significantly lower than those obtained by diffusive repartitioners for this class of problems.

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 partitioning that has a high overlap with the existing partitioning. We show that LMSR decreases the amount of vertex migration required to balance the graph compared to current scratch-remap schemes, particularly for slightly imbalanced graphs. We describe 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 domains. We show that Wavefront Diffusion obtains significantly lower vertex migration requirements while maintaining similar or better edge-cut results compared to existing diffusion algorithms, especially for highly imbalanced graphs. Furthermore, we compare Wavefront Diffusion with LMSR and show that the former scheme results in generally lower vertex migration requirements at the cost of lower quality edge-cuts. Our experimental results on parallel computers show that both schemes are highly scalable. For example, both are capable of repartitioning an eight million node graph in under three seconds on a 128-processor Cray T3E.

2 Definitions, Background & Experimental Setup

This section gives definitions that will be used in the remainder of the paper, describes the multilevel graph partitioning paradigm and recent scratch-remap and diffusion-based repartitioners, and explains the experimental setup used to evaluate our new algorithms.

2.1 Definitions

In our discussions, we refer to a partitioning as being composed of k disjoint *domains*. Each of these domains is composed of a number of vertices. Vertices have both *weight* and *size* [13, 17]. Vertex weight is the computational cost of the work represented by the vertex, while size reflects its migration cost. Thus, the repartitioner should attempt to balance the partitioning with respect to vertex weight while minimizing vertex migration 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.

The weight of a domain is the sum of the weights of the vertices of which that domain is composed. A domain is considered *overweight* if its weight is greater than the average domain weight times $1+\epsilon$ (where ϵ is a user specified constant). Likewise, a domain is *underweight* if its weight is less than the average domain weight divided by $1+\epsilon$. A partitioning is *balanced* when none of its domains are overweight (although some domains may be underweight). Two domains are *connected* if there is at least one edge with incident vertices in each of the two domains.

The domain in which a vertex is located on the original partitioning is the *home* domain of that vertex. A vertex is *clean* if its current domain is its home domain. Otherwise, it is *dirty*. TOTALV is defined as the sum of the sizes of vertices that change domains as the result of repartitioning [13]. TOTALV reflects the overall volume of communications needed to balance the partitioning. MAXV is defined as the maximum of the sums of the sizes of those vertices that migrate into or out of any one domain as a result of repartitioning [13]. MAXV reflects the maximum time needed by any one processor to send or receive data.

In our discussions in Sections 3, 4, and 5, we will only focus on TOTALV. Results in [13] show that measuring the MAXV can sometimes be a better indicator of data migration overhead than measuring the TOTALV. However, in general, minimizing TOTALV tends to do a fairly good job of minimizing MAXV. Therefore, the parallel implementations described in this paper attempt to minimize MAXV indirectly by concentrating on minimizing TotalV. In the final set of experiments in Sections 6.1 and 6.2 we present both TotalV and MaxV results for various schemes.

2.2 Multilevel Graph Partitioning

Multilevel graph partitioners are considered state-of-the-art for static graph partitioning [5, 11]. Most existing repartitioning algorithms for adaptive meshes, including the ones presented in this paper, are built upon the multilevel paradigm. Hence, we provide a brief description of the k-way multilevel partitioning scheme for static graphs that will form the basis of all the schemes considered in this paper.

The multilevel graph partitioning paradigm consists of three phases: graph coarsening, initial partitioning, and multilevel refinement. In 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. This newly constructed graph then acts 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, up to the finest (i. e., the original graph). The result is that the refinement algorithm sees multiple views of the graph, from highly global to very local ones. This magnifies the power of refinement so that simple heuristics can be utilized to compute high-quality partitionings quickly.

In the coarsening phase, vertices are matched together by computing a maximal set of independent edges. Most multilevel schemes use some variation of heavy-edge matching [11] to compute this set. Here, vertices are examined in some order. Each vertex is matched with the one of its unmatched adjacent vertices that is connected to it by the edge with the greatest edge weight. This heuristic attempts to match highly connected vertices together (i. e., vertices that are joined by a relatively heavy edge). The result is a sequence of progressively coarser graphs based on the input graph, in which the vertices of the coarse graphs generally consist of highly connected vertices of the finer graphs.

The initial partitioning phase is performed by calling a k-way recursive bisection partitioning algorithm. Since this algorithm is called at the coarsest graph, the partitioning can be computed quickly.

In the multilevel refinement (or uncoarsening) phase, border vertices from the coarsest graph are selected in some order. Each vertex is examined and migrated if doing so will satisfy at least one of the following conditions (in order of importance):

1. decrease the edge-cut while still satisfying the balance constraint, or
2. improve the balance while maintaining the edge-cut.

A small number of iterations through the border vertices is completed at each successively finer level graph. Figure 5 illustrates multilevel k-way graph partitioning.

2.3 Scratch-Remap Algorithms

Oliker and Biswas describe various scratch-remap algorithms in [13]. Here, the imbalanced graph is partitioned from scratch using a multilevel graph partitioning algorithm. The newly computed partitioning is then intelligently mapped to the original partitioning in order to reduce the amount of vertex migration required. Oliker and Biswas describe a simple greedy remapping algorithm that attempts to minimize the sum of the sizes of the vertices that are required to migrate domains (i. e., the TotalV). They also show that this scheme results in remappings that are of near-optimal quality for various application graphs [13].

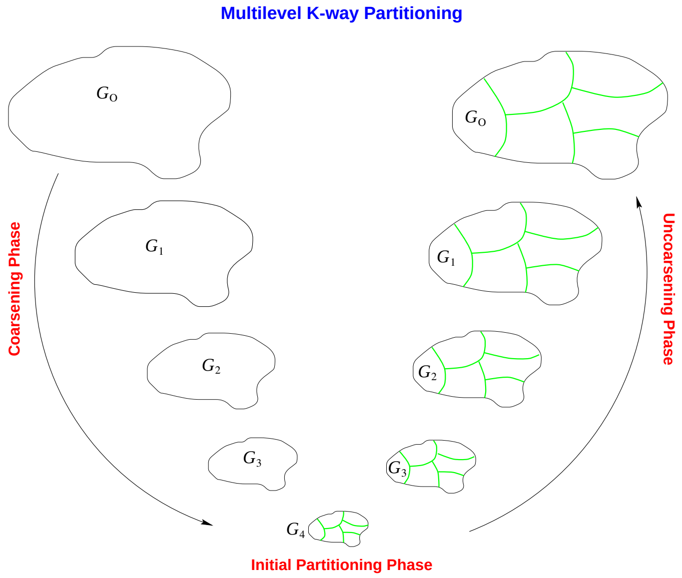

Figure 3: The three phases of multilevel k-way graph partitioning. During the coarsening phase, the size of the graph is successively decreased. During the initial partitioning phase, a k-way partitioning is computed, During the multilevel refinement (or uncoarsening) phase, the partitioning is successively 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.

In this context, partition remapping is a three step process. It requires two input partitionings, the original (or *old*) partitioning and the newly computed (or *new*) partitioning, and it outputs a remapping of the newly computed partitioning. The remapping process is as follows:

1. **Construct a similarity matrix, S, of size $k \times k$.** A similarity matrix is one in which the rows represent the domains of the old partitioning, the columns represent the domains of the new partitioning, and each element, S_{qr}, represents the sum of the sizes of the vertices that are in domain q of the old partitioning and in domain r of the new partitioning.

2. **Select k elements such that every row and column contains exactly one selected element and some objective is satisfied.** For example, if the objective is to minimize the TotalV, then it is necessary to select elements such that the sum of their sizes is maximized. This corresponds to the remapping in which the amount of overlap between the original and the remapped partitionings is maximized, and hence, the total amount of vertex migration required in order to realize the remapped partitioning is minimized.

3. **For each element S_{qr} selected, rename domain r to domain q on the remapped partitioning.**

Figure 4 illustrates such a remapping process for the example in Figure 1. Here, similarity matrix, S, has been constructed. The first row of S indicates that domain 1 on the new partitioning consists of zero vertices from domain 1 on the old partitioning, three vertices from domain 2 on the old partitioning, and four vertices from domain 3 on the old partitioning. Likewise, the second row indicates that domain 2 on the new partitioning consists of two vertices from domain 1 on the old partitioning and zero vertices from either of the other two domains on the old partitioning. The third row is constructed similarly. In this example, we select underlined elements S_{13}, S_{21}, and S_{32}. This combination maximizes the sum of the sizes of the selected elements. Running through the selected elements, domain 3 on the newly computed partitioning is renamed 1 on the remapped partitioning, and domains 1 and 2 are renamed 2 and 3, respectively. Figure 1(d) shows the graph after partition remapping.

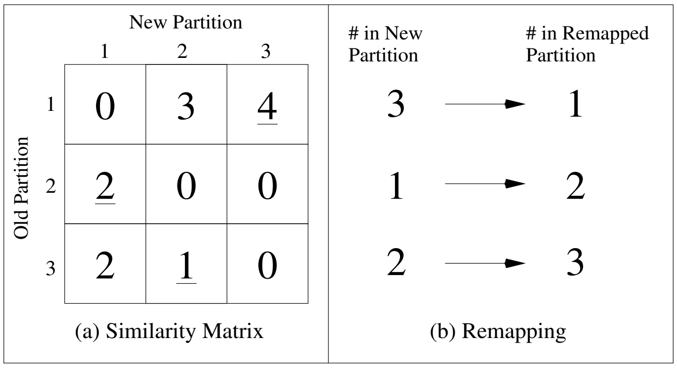

Figure 4: Similarity matrix (a) and remapping (b) for the graph in Figure 1

2.4 Multilevel Diffusion Algorithms

Various multilevel diffusion repartitioning algorithms are described in [15, 18]. As a modification of the multilevel graph partitioning paradigm, these schemes have three phases: a graph coarsening phase, a diffusion phase (in place of the initial partitioning phase), and a multilevel refinement phase.

In the graph coarsening phase, vertex matching is purely local. That is, vertices may be matched together only if they have the same home domain. Otherwise, this phase is identical to that of the multilevel k-way graph partitioner described in Section 2.2. The result of purely local matching is that each successive coarser graph inherits the partitioning of the immediate finer level graph. Therefore, the initial partitioning phase is no longer necessary, as this will be inherited from the original (unbalanced) partitioning. Instead, the inherited partitioning needs to be balanced.

In the diffusion phase, the coarsest (and hence smallest) graph is balanced by incrementally modifying the inherited partitioning. By beginning this process on the coarsest graph, these algorithms are able to move large chunks of highly connected vertices in a single step. Thus, the bulk of the work required to balance the partitioning is done quickly. Eventually, due to the coarseness of the graph, balance may not be able to be improved by an incremental diffusion process. At this point, either refinement is begun on the current graph or the partitioning is projected to the next finer graph and another round of diffusion begins.

In these algorithms, diffusion may be directed by global or local views of the load imbalance. We refer to these two methods as *directed diffusion* and *undirected diffusion* [15]. Comparisons of efficient implementations based on these schemes in [15] have shown the following.

1. Directed diffusion algorithms can efficiently take advantage of the global view of the load imbalance to minimize the amount of vertex migration or the perturbation to the edge-cut.

2. Undirected diffusion algorithms can result in higher vertex migration or edge-cut perturbation than directed diffusion due to the local view of the load imbalance employed to guide diffusion.

3. Undirected diffusion algorithms are highly distributed in nature, and thus, are more scalable than directed diffusion algorithms.

In the case of directed diffusion, border vertices from the coarsest graph are selected in some order. Each vertex is examined and some type of global view of the load imbalance is consulted to determine if this vertex should migrate domains. The authors of [15, 18] utilize a technique described by Hu and Blake [7] that computes a vector, (referred to as the *diffusion solution* in [15]), λ, with p elements, such that the amount of vertex weight that needs to be migrated from domain q to domain r is $\lambda_q - \lambda_r$, where domains q and r are adjacent, and p is the total number of domains. Hu and Blake prove that the vertex flow computed by λ is minimal in the l_2-norm [7]. Thus, the amount of vertex migration required to balance the partitioning is kept low.

For undirected diffusion, migration of vertices is directed by a purely local view of the load imbalance. Border vertices from the coarsest graph are selected in some order. As each vertex is examined, the weight of the domain in which the vertex is currently located is compared to the domain weights of all of the domains that are adjacent to that vertex. The vertex is then assigned to the domain that brings about the greatest reduction in the partition imbalance.

If the original partitioning was computed intelligently, then the edge-cut is likely to be in a local minima of the search space. Hence, any single vertex migration will increase the edge-cut and a series of balancing migrations will tend to increase the edge-cut as the new partitioning is moved further and further away from the initial partitioning in the search space. In the multilevel refinement phase, the objective is to improve the edge-cut that is disturbed during the diffusion phase.

It is important to note that multilevel refinement in the context of repartitioning is a modified version of multilevel refinement in the context of graph partitioning. In repartitioning, not only are edge-cut and balance of concern, but so is the amount of vertex migration required to realize the balanced partitioning. Therefore, multilevel refinement in this context should take all three of these factors into account. In [15] a multilevel refinement scheme is described in the context of repartitioning in which border vertices are migrated if doing so will satisfy one or more of the following conditions (listed in order of importance):

1. decrease the edge-cut while still satisfying the balance constraint,
2. decrease the TotalV while maintaining the edge-cut and still satisfying the balance constraint, or
3. improve the balance while maintaining the edge-cut and the TotalV.

If conflicts occur between possible target domains, then the vertex is migrated to the domain that will satisfy the highest priority condition.

The third condition will move a vertex out of a domain that is above the average domain weight (but not necessarily overweight) and into a domain with less domain weight if doing so will not increase the edge-cut and the vertex is not migrating out of its home domain. This has two effects. Not only will the partition balance be improved, but the edge-cut will also tend to be improved. This is because by moving a vertex out of a domain while maintaining the edge-cut, that domain becomes free to accept another vertex from a neighboring domain that can improve the edge-cut. Thus, moving a vertex that satisfies condition three gives a *potential* edge-cut improvement. Note that this heuristic will tend to more effective as the number of dirty vertices increases as this will allow for a greater number of potential edge-cut improving migrations.

Except for the modifications noted above, the multilevel refinement phase is identical to that of the multilevel k-way graph partitioner described in Section 2.2.

Many variations of the multilevel diffusion algorithms described above are possible. For example, both directed diffusion and refinement can be performed on every level graph from the coarsest back up to the original [18]. Another possibility is that directed diffusion is performed on a few of the coarsest level graphs until the partitioning is balanced. At this point, multilevel refinement takes over [15]. A third possibility is that directed diffusion takes place only on the coarsest graph. Undirected diffusion can then be performed on the next few finer level graphs until the partitioning is balanced. Finally, multilevel refinement can be performed. In this paper, we examine the third variation. The advantage is that the bulk of the balancing work is performed by directed diffusion on the coarsest graph. As the graphs increase in size, more scalable

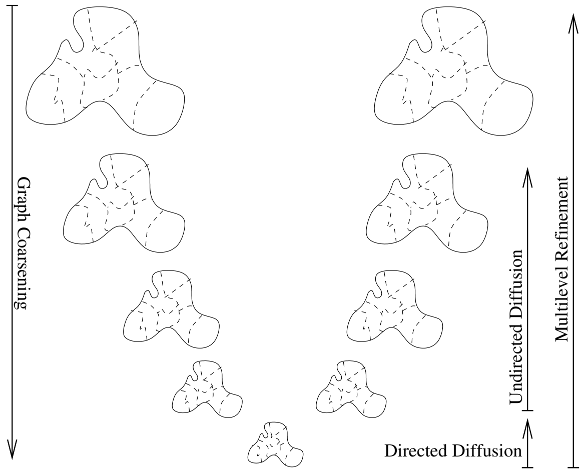

Figure 5: The three phases of multilevel diffusion. During the coarsening phase, the size of the graph is successively decreased. During the diffusion phase, directed diffusion is applied on the coarsest graph. Undirected diffusion is applied on the next few finer level graphs until balance is obtained. During the multilevel refinement (or uncoarsening) phase, the partitioning is successively refined as it is projected to the larger graphs.

undirected diffusion algorithms are then able to efficiently perform the remainder of the balancing work. Figure 5 illustrates this multilevel diffusion scheme.

Note that all variations of directed and undirected diffusion discussed above take the connectivity of the graph into consideration. Therefore, migration of vertices is allowed only between connected domains. This is critical for minimizing the edge-cut of the resulting repartitionings. This property distinguishes these schemes from many existing diffusion-based load-balancing schemes [2, 3, 4, 6, 14, 19] that diffuse work from highly loaded processors to lightly loaded processors. Even though these schemes do take the connectivity of the processors into consideration, such connectivity remains unchanged. In contrast, connectivity of domains located at different processors can change dynamically due to the movement of vertices. We focus this paper on those schemes that do take the graph into consideration when computing a repartitioning.

2.5 Synthetic Graphs Used for Experimental Evaluations

All of the serial experiments described in this paper were conducted on three finite element meshes described in Table 1. Experiments on parallel machines were conducted on larger meshes described in Section 6.1. Experimental 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 1 were set to one. Next, two partitionings were computed for each graph, a 64-way partitioning and a 256-way partitioning. Three adjacent domains were then selected at random from the 256-way partitioning. The weights of all of the vertices in these domains were set to α. The weights of all of the vertices adjacent to these domains were set to $\alpha - 3$. Thus, a ring of vertices of weight $\alpha - 3$ was constructed around the selected domains. Further, rings of $\alpha - 6$, $\alpha - 9$, and so on, were constructed concentrically about the overweight region (while these values were greater than one), so as to moderate the boundary condition. The result is a localized increase in vertex weight. Finally, each 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 domains will be of weight 10. A ring of weight seven vertices will immediately encircle this region. A ring of weight four vertices will then encircle this region. All of the other vertices will have weight one. The weight of the edges inside of the selected domains will be $10^{.667} = 4.65$ (truncated down to four). The weight of an edge with one incident vertex in a selected domain and one vertex in the first encircling ring will be $8.5^{.667} = 4.17$ (also truncated down to four). Finally, the 64-way

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's Saturn
MDUAL2	988,605	1,947,069	dual of a 3D mesh

Table 1: Characteristics of the various graphs used in the experiments.

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.

3 Locally-Matched Multilevel Scratch-Remap

In this section, we describe two enhancements to the scratch-remap scheme. We show that by restricting the matching phase of a multilevel graph partitioner to purely local matching it is possible to decrease the amount of vertex migration required significantly, while increasing the edge-cut only slightly compared to results obtained when global matching is allowed. We describe a scheme that performs remapping in a multilevel context and show that this scheme can reduce the amount of vertex migration required while maintaining the edge-cut compared with schemes that perform remapping after graph partitioning is complete. We refer to our new scratch-remap scheme that implements both local matching and multilevel remapping as *Locally-Matched Multilevel Scratch-Remap* (or simply *LMSR*).

3.1 Local Matching

The effectiveness of the remapping scheme of Oliker and Biswas [13] is dependent on the nature of the similarity matrix. An ideal similarity matrix is one in which there is exactly one non-zero element in each row and column. This corresponds to the situation in which the new partitioning is identical to the old partitioning except with regard to the domain labels. This is infeasible, since the old partitioning is imbalanced and the new partitioning is balanced (to the extent desirable as discussed in Section 2.1). 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 domain of the new partitioning consists of an equal share of every domain of the old partitioning.

Figure 6 illustrates these different types of matrices. Figure 6(a) is an example of an ideal similarity matrix. This is uninteresting because the new partitioning is not balanced. Figure 6(b) is shows a similarity matrix constructed from two partitionings in which there are large amounts of vertex overlap. Figure 6(c) shows an opposite case. Here, each of the domains of the newly computed partitioning share a roughly equal amount of vertex weight of each of the domains of the old partitioning. The underlined entries indicate the selected elements. While both of these remappings were computed using the method described in [13], the TotalV and MaxV are significantly lower for the case in Figure 6(b) than for in Figure 6(c).

One way to increase the effectiveness of remapping is to bias the process of graph partitioning such that the situation illustrated in Figure 6(b) will occur more frequently. That is, the new partitioning is such that there are large areas of overlap between a majority of domains of the old and new partitionings. Existing state-of-the-art multilevel graph partitioners such as MeTiS and Chaco do not provide this bias. More specifically, in these multilevel graph partitioners, a pair of vertices can be matched regardless of whether or not they are

200	0	0	0	0
0	350	0	0	0
0	0	0	0	475
0	0	0	225	0
0	0	250	0	0

TotalV = 0

MaxV = 0

(a)

200	0	0	0	0
100	25	0	175	50
0	75	250	100	50
0	200	0	25	0
0	0	50	0	200

TotalV = 475

MaxV = 225

(b)

40	60	30	40	30
70	50	90	70	70
100	90	95	110	80
40	45	35	45	60
50	55	50	35	60

TotalV = 1130

MaxV = 365

(c)

Figure 6: Examples of ideal (a), good (b), and bad (c) overlap matrices

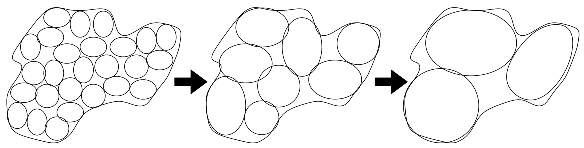

Figure 7: A single domain from a series of successively coarser graphs constructed utilizing local matching

in the same domain of the original partitioning. Hence, the vertices in the coarsest level graph may contain vertices from multiple domains of the original partitioning.

One way to bias the multilevel graph partitioner towards the existing partitioning is to restrict matchings to among vertices that have the same home domain. The result is that vertices of each successively coarser graph correspond to regions within the same domain of the original partitioning. By the time the coarsest graph is constructed, every domain here consists of a relatively small number of well-shaped regions, each of which is a sub-region of a single home domain. Figure 7 illustrates this point. It shows a single domain coarsened via local matching.

Another advantage of this purely local matching is that the edges of the original partitioning still remain visible even in the coarsest graph. Hence, in those portions of the graph that are relatively undisturbed by adaptation, the initial partitioning algorithm will have a tendency to select the same partition boundaries. This can have a positive effect on both edge-cut and TotalV results.

3.2 Experimental Results for Local Matching

To judge the effectiveness of local matching, four experiments were conducted on each of the three graphs from Table 1. These were constructed via the process described in Section 2.5 by setting α to 2, 5, 10, and 20. The experimental graphs were repartitioned by two implementations of the scratch-remap algorithm described in [13] one utilizing global matching, and the other using only local matching. Figure 8 compares the edge-cut and TotalV results of these algorithms. For each experiment, Figure 8 contains two bars. The first bar indicates the edge-cut obtained by the scratch-remap algorithm utilizing local matching normalized by the edge-cut obtained by the scratch-remap algorithm utilizing global matching. The second bar indicates the TotalV obtained by the scratch-remap algorithm utilizing local matching normalized by the TotalV

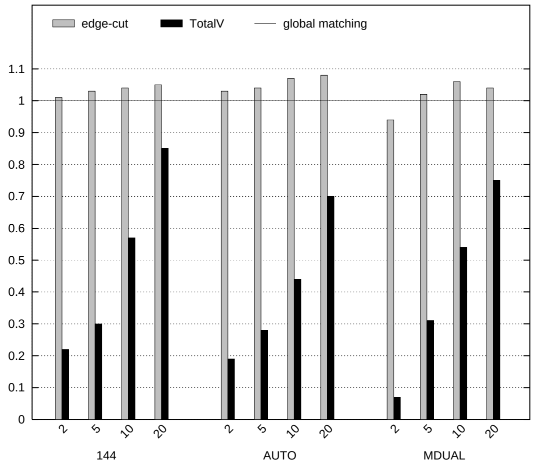

Figure 8: A comparison of edge-cut and TOTALV results obtained from scratch-remap algorithms utilizing local and global matching.

obtained by the same algorithm utilizing global matching. Therefore, a result below the index line indicates that the local matching results are better than the global matching results.

For all of the experiments in Figure 8, local matching resulted in TOTALV levels that are 7% to 85% of those obtained by global matching. This means that utilizing local matching can decrease the amount of data that is required to be migrated among the processors by 15% to over 90% compared to global matching. Figure 8 shows that the edge-cuts of the two schemes are similar. However, local matching results in generally worse edge-cuts by about 2% to 6%. This is because global matching is more free than local matching to collapse vertices with very heavy edges between them. Collapsing such vertices, as shown in [11], has a beneficial effect on edge-cut for multilevel refinement algorithms. Notice that as the values for α decrease, the differences in TOTALV results increase, and the differences in edge-cut decrease. Figure 8 shows that the locally matched algorithm does a better job of keeping vertex migration low while maintaining the edge-cut compared to the globally-matched algorithm. This is especially true for slightly to moderately imbalanced graphs.

The above results are consistent with those presented by Oliker and Biswas in [13]. They show that remapping is more effective for partitionings computed from scratch by the parallel k-way multilevel graph partitioner implemented in PARMETIS [8, 12] than by the serial k-way multilevel graph partitioner implemented in METIS [9, 10]. The PARMETIS graph partitioner utilizes a matching scheme that favors local matching over global matching,[1] while the METIS graph partitioner utilizes global matching.

3.3 Multilevel Scratch-Remap

A potential improvement to the scratch-remap algorithms described in [13] is to apply remapping on the coarsest graph after the new initial partitioning is computed, but before multilevel refinement is begun. As discussed in Section 2.4, multilevel refinement in the context of repartitioning takes three criteria into consideration when determining whether to migrate vertices. Here, border vertices are examined and migrated if doing so will satisfy one or more of the following conditions (in the order of importance):

1. decrease the edge-cut while still satisfying the balance constraint,

[1] PARMETIS allows non-local matchings for vertices that could not be matched by a purely local edge.

2. decrease the TotalV while maintaining the edge-cut and still satisfying the balance constraint, or
3. improve the balance while maintaining the edge-cut and the TotalV.

Including criterion 2 allows the TotalV resulting from the remapped partitioning to be minimized as a secondary priority during multilevel refinement. Thus, the power of multilevel refinement can be focused on reducing both edge-cut and TotalV. When partition remapping is performed only after multilevel refinement, criterion 2 does not apply (i. e., there is no way to determine the TotalV). Therefore, it cannot effectively be minimized in this way. We refer to this new scheme as Multilevel Scratch-Remap.

3.4 Experimental Results for Multilevel Scratch-Remap

In order to test the effectiveness of the Multilevel Scratch-Remap scheme we repartitioned the same experimental graphs described in Section 3.2 with two implementations of the scratch-remap algorithm. In one implementation, remapping is performed during the initial partitioning phase, immediately after the partitioning is computed and before multilevel refinement begins. Also, the multilevel refinement algorithm is modified as described in Section 3.3 so as to minimize TotalV as a secondary priority. In the second implementation, remapping is performed after multilevel refinement is complete. Global matching is performed in both implementations. Figure 9 compares the edge-cut and TotalV results obtained by the scratch-remap algorithms performing pre- and post- refinement remapping. The results obtained by the algorithm in which remapping is performed prior to multilevel refinement are normalized by those obtained by the algorithm in which remapping is performed after multilevel refinement.

Figure 9 shows that the edge-cuts obtained from the two schemes are virtually identical if we discount noise. However, the pre-refinement remapping scheme obtained somewhat better TotalV results on average than the scheme in which remapping is performed after refinement. In one case, the pre-refinement remapping scheme resulted in TotalV as little as 41% of the other scheme. Seven out of 12 experiments resulted in better TotalV results for the pre-refinement remapping scheme compared with the scheme in which remapping is performed after multilevel refinement, three out of 12 resulted in higher TotalV, and two out of 12 resulted in similar TotalV. Figure 9 indicates that utilizing remapping in a multilevel context can generally decrease the amount of TotalV while maintaining a high-quality edge-cut compared with performing remapping after refinement.

4 Wavefront Diffusion

An effective and robust implementation of a directed diffusion scheme requires the consideration of two issues. The first of these is the problem of *dynamic domain connectivity*. As stated in Section 2.1, two domains are connected if there is at least one edge with incident vertices in each of the two domains. Computation of the diffusion solution depends on knowledge of the current domain connectivity. (As discussed in Section 2.4, this is also critical for obtaining good edge-cuts.) However, as vertices are migrated during diffusion, the connectivity of domains may change. The result is that a domain may lose connectivity with another domain, and thus, be unable to migrate the required amount of vertex weight.

Figure 10 illustrates this problem. Figure 10(a) shows a graph with an imbalanced partitioning. Figure 10(b) gives a graphic depiction of the diffusion solution for this partitioning. It shows that domain 1 must migrate two vertices to each of domains 2 and 3 and that no migration of vertices is required between domains 2 and 3. Figure 10(c) shows the state of the partitioning during directed diffusion, immediately after domain 1 has migrated two vertices to domain 2. Figure 10(d) gives a graphic depiction of the diffusion solution at this state. Here, domain 1 is still required to migrate two vertices to domain 3 (and no migration of vertices

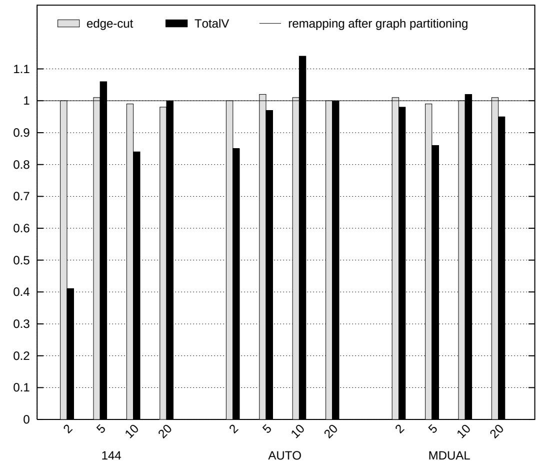

Figure 9: A comparison of edge-cut and TotalV results obtained from scratch-remap algorithms utilizing pre-refinement remapping and remapping after multilevel refinement.

is required between domains 1 and 2 nor between domains 2 and 3.) However, since these domains are no longer connected, this is not possible. At this point, some measure must be taken to correct the situation or else directed diffusion will fail to balance the partitioning.

The second issue arises for partitionings that are highly imbalanced in localized areas. This class of problems requires diffusion to propagate over long distances. As a result, many domains may be simultaneously both recipients and donors of vertices during diffusion. For these transient domains, current directed diffusion algorithms interleave the outgoing flow of vertices with the incoming flow of vertices from neighboring domains. Such a domain is often forced to move out vertices before it has gotten all of the vertices that it is supposed to receive from its neighbors. Hence, it will have only a limited choice for selecting good outgoing vertices with respect to minimizing the edge-cut and the required vertex migration. In the worst case, a domain could even inadvertently export all of its vertices during directed diffusion. The result is that this domain would lose connectivity with all other domains in the partitioning. Thus, it would be unable to ever receive incoming vertices, and again, directed diffusion would fail to balance the partitioning. Figure 11 illustrates this problem. In Figure 11(a), domain 1 is overweight. Figure 11(b) gives a graphic depiction of the diffusion solution at this state. Here, domain 1 is required to migrate six vertices to domain 2, and domain 2 is required to migrate three vertices to each of domains 3 and 4. However, domain 2 contains only four vertices. Hence, it is required to export more vertices to domains 3 and 4 than it currently contains.

One way to address this problem is to begin the diffusion of vertices at those domains that have no required flow of vertices into them. Then, after these domains reach balance, the diffusion solution is recomputed and the next iteration is begun on the set of domains whose required flow of vertices into them was satisfied during the previous iteration, and so on, until all of the domains are balanced. This method guarantees that all domains will contain the largest selection of vertices possible when it is their turn to export vertices. Thus, domains are able to select those vertices for migration that will best minimize edge-cut and vertex migration requirements, while reducing the likelihood of losing connectivity to a neighboring domain.

A disadvantage of this scheme is that it requires a large number of iterations to balance the graph, and hence, is not scalable. We implemented a modification that retains the spirit of this scheme, while requiring fewer iterations to balance the partitioning as follows. We maintain two arrays, $inflow$ and $outflow$, with one element per domain. $inflow\,[i]$ contains the sum of the vertex weight that domain i is required to receive in from other domains, and $outflow\,[i]$ contains the sum of the vertex weight that domain i is required to send

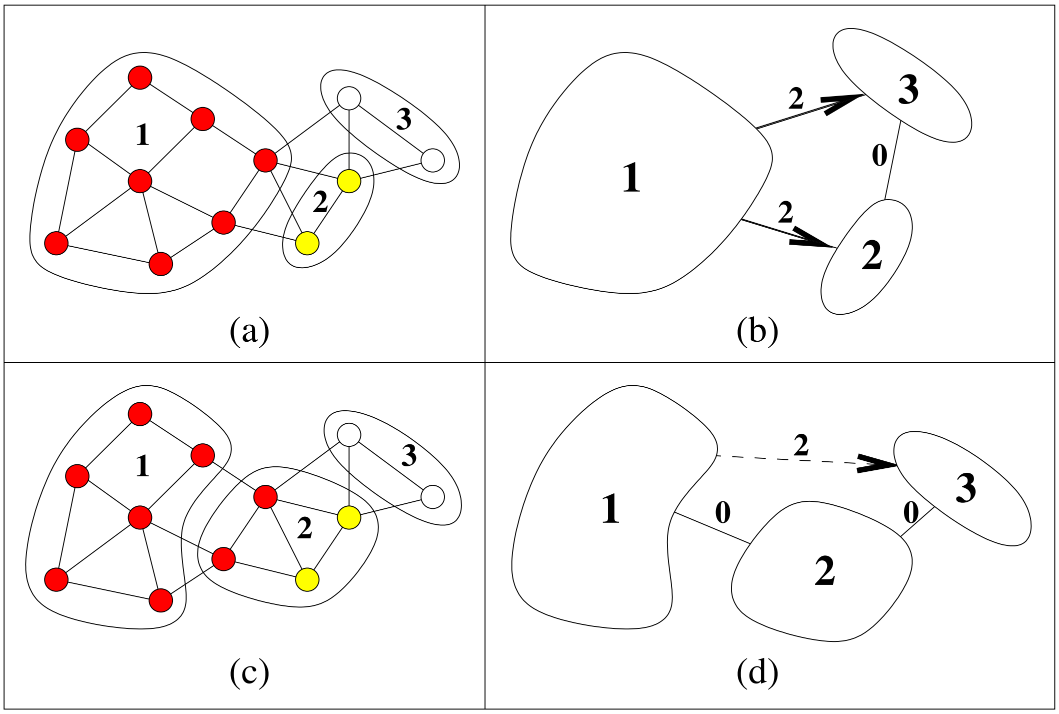

Figure 10: This figure shows an imbalanced partitioning (a), a graphic depiction of the diffusion solution (b), the state of the graph after domain 2 has received two vertices (c), and the diffusion solution at this time (d).

out to other domains. In each iteration, only those domains are allowed to migrate clean vertices for which the ratio of $outflow\,[i]\,/inflow\,[i]$ is above a threshold. In addition, all domains are allowed to migrate dirty vertices. By setting the threshold to infinity, we obtain the algorithm described above. By setting the threshold to zero, we obtain the directed diffusion algorithm described in [15]. In our experiments, we compared the ratios of every domain and set this threshold to be slightly less than the maximum finite ratio found. Thus, only those domains with a ratio equal to infinity and a few of the domains with the highest non-infinity ratios were allowed to migrate clean vertices on any given iteration.

When the threshold is set to a suitably high number (as such), this scheme achieves an important effect. That is, dirty vertices are able to be migrated multiple times, reducing TotalV and possibly MaxV. The reason is vertices that migrate from overweight domains in the first iterations are able to be migrated again in later iterations as diffusion continues to propagate. Furthermore, the potential for this reuse of dirty vertices increases as diffusion is required to propagate further distances. Our experimental results (not included in this paper) have shown that this is a very significant effect. In fact, we have obtained results in which as much as 85% of all vertex migrations are made by dirty vertices on extremely imbalanced graphs. Such dirty vertex reuse is tremendously beneficial in obtaining low vertex migration results.

A further modification of the scheme described above that decreases the edge-cut of the resulting repartitioning is to sort vertices with respect to the amount of edge weight that is cut by the current partitioning prior to each diffusion iteration. Thus, vertices that are highly connected to vertices in different domains are selected first for migration. This modification tends to decrease the edge-cut at the cost of a minor increase in vertex migration.

We refer to the algorithm that incorporates all of these modifications as *Wavefront Diffusion* (or simply WD), as the flow of vertices move in a wavefront from overweight domains to underweight domains. For the sake of clarity, we will refer to the directed diffusion algorithm described in [15] (i. e., the multilevel diffusion

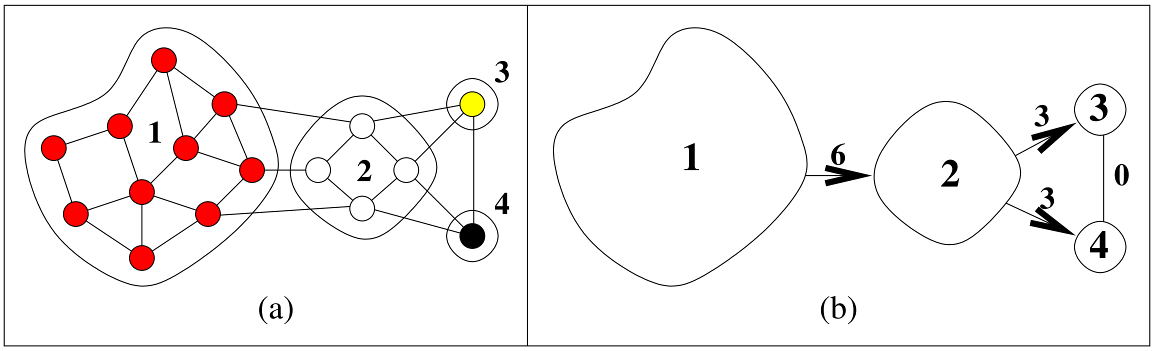

Figure 11: This figure shows an imbalanced partitioning (a) and a graphic depiction of the diffusion solution (b).

phase of the MLDD algorithm) as MLDD. Note that the coarsening and multilevel refinement phases of the WD and MLDD algorithms are identical. These two schemes differ only in the way diffusion is performed at the coarsest graph.

4.1 Experimental Results for Wavefront Diffusion

In order to test the effectiveness of our Wavefront Diffusion algorithm, we repartitioned the experimental graphs described in Section 3.2 with both our Wavefront Diffusion algorithm and the directed diffusion algorithm, MLDD, described in [15]. These experiments give the results of the diffusion phase only. That is, the graphs were coarsened identically 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. Figure 12 gives experimental results comparing WD with MLDD. The bars indicate the edge-cut (and TotalV) results obtained by our WD algorithm normalized by those obtained by the MLDD algorithm. Thus, a bar below the index line indicates that the WD algorithm obtained better results than the MLDD algorithm.

Figure 12 shows that the WD algorithm produced better results than the MLDD algorithm across the board for both edge-cut and TotalV. Specifically, the WD algorithm obtained edge-cuts that are 93% to 53% of those obtained by the MLDD algorithm. The WD algorithm obtained TotalV results that are 64% to 36% of those obtained by the MLDD algorithm. Thus, the results indicate that our Wavefront Diffusion algorithm is more effective at computing high-quality repartitionings, while minimizing the amount of TotalV than the MLDD algorithm, especially for meshes with high levels of localized adaptation.

5 Comparing LMSR and Wavefront Diffusion

We compared our WD and LMSR algorithms on a set of experiments that includes a spectrum of slightly to extremely imbalanced partitionings. The graphs are the same as described for Figure 8 with the addition of some highly imbalanced graphs (i. e., α values of 30, 40, 50, and 60). Figure 13 compares the edge-cut and TotalV results of our two new schemes, Wavefront Diffusion and LMSR. The bars indicate the edge-cut (and TotalV) results obtained by our WD algorithm normalized by those obtained by our LMSR algorithm. Thus, a bar below the index line indicates that the WD algorithm obtained better results than the LMSR algorithm.

Figure 13 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 WD algorithm are up to 42%

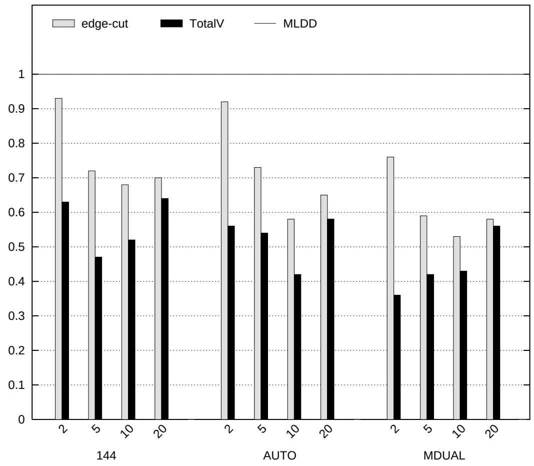

Figure 12: A comparison of edge-cut and TotalV results obtained from the WD and MLDD algorithms.

higher than those obtained by the LMSR algorithm. Figure 13 also shows that the WD algorithm was able to obtain TotalV results that are significantly better than those obtained by the LMSR algorithm across the board. In particular, the WD algorithm obtained TotalV results that are 24% to 95% of those obtained by the LMSR algorithm. Figure 13 shows that except for the case of very slightly imbalanced partitionings, there is a clear tradeoff between edge-cut and TotalV with respect to the two new 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, the WD scheme is strictly better than the LMSR scheme, as it obtains similar edge-cuts and better TotalV.

6 Parallel Implementations

In this section, we describe the parallel implementations of our LMSR and WD algorithms and present experimental comparisons of these schemes with scratch-remap and multilevel diffusion algorithms.

The coarsening phases of our LMSR and WD algorithms are identical. The vertices are initially assumed to be distributed across the processors. This division of vertices corresponds to the original partitioning. In the coarsening phase of both schemes, purely local matching is performed. This ensures that all matched vertices are present on the processor, and hence, the coarsening phase is embarrassingly parallel. The multilevel refinement phases for both our LMSR and WD algorithms are likewise identical. These are the same as the multilevel refinement phase of the MLDD algorithm described in [15]. Here, vertices are selected for migration if doing so will satisfy at least one of the following conditions (in order of importance).

1. decrease the edge-cut while still satisfying the balance constraint,
2. decrease the TotalV while maintaining the edge-cut and still satisfying the balance constraint, or
3. improve the balance while maintaining the edge-cut and the TotalV.

Also in this implementation, adjacent domains migrate vertices in alternating phases in order to avoid thrashing [8].

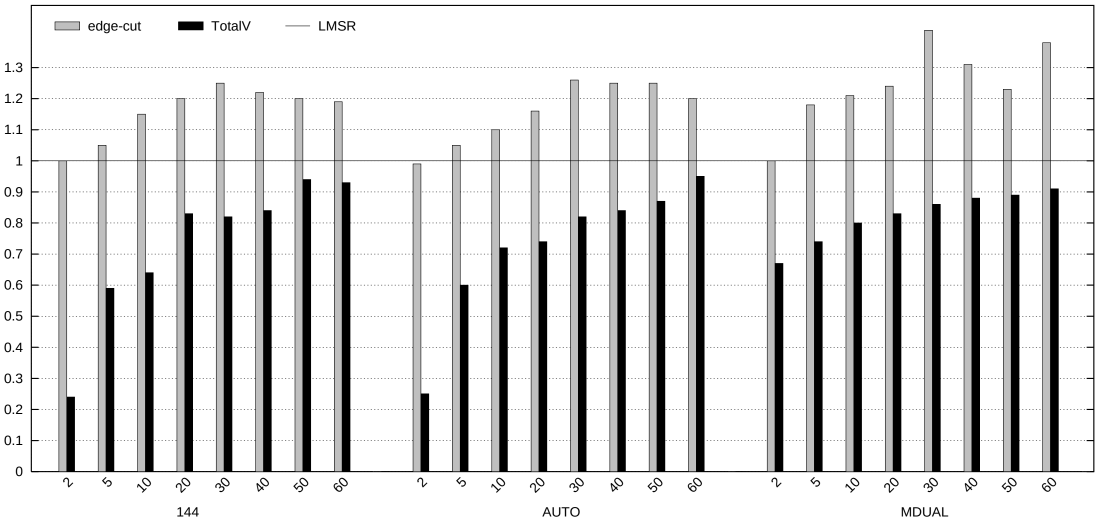

Figure 13: A comparison of edge-cut and TotalV results obtained from the WD and LMSR algorithms.

Our LMSR and WD algorithms differ only in the initial partitioning phase. The initial partitioning phase of our LMSR algorithm is identical to that used in parallel multilevel k-way partitioning algorithm described in [8]. It is parallelized by recursive decomposition. That is, every processor computes (the same) bisection of the graph. Then half of the processors compute a bisection of either sub-graph. This continues recursively until the number of unique domains equals the number of processors. Next, the LMSR algorithm performs pre-refinement remapping. (Note, the computation of partition remapping for all schemes is performed serially on a single processor, as its run time tends to be much smaller than the time to move the vertices to their destinations.)

The initial partitioning phase of our WD algorithm is as follows. After graph coarsening, the coarsest graph is assembled and broadcast to all of the processors. Next, one processor performs the sorted version of Wavefront Diffusion and all of the others perform the unsorted version of Wavefront Diffusion. Since the unsorted version of the WD algorithm examines vertices for migration in a random order, all of the processors are likely to explore a different solution path. After at least one processor has balanced the graph to within 10%, then all of the computed partitionings are compared and the one that has the lowest value for (edge-cut × balance) is selected. This partitioning is then broadcast to all of the processors and multilevel refinement begins.

In the following sections, we compare the results of our LMSR algorithm with the multilevel scratch-remap repartitioner implemented in ParMETIS as the routine PARPAMETIS. PARPAMETIS is essentially identical to the scratch-remap scheme described by Oliker and Biswas in [13] except that its matching has built in bias for local edges. That is, every vertex is matched locally, if this is possible. Then, all of the vertices unable to be matched locally are matched globally. We will refer to this scheme as *SR*. The initial partitioning and multilevel refinement phases of the SR algorithm are similar to those described above for the LMSR algorithm with the following two exceptions. First, pre-refinement remapping is not performed during the initial partitioning phase of the SR algorithm. Instead, remapping is performed following the multilevel refinement phase. Second, the multilevel refinement phase does not consider criterion 2 above when examining a vertex for migration since there is no original partitioning here to compare against.

The parallel formulation for the MLDD algorithm is identical to that of WD with the following exception. During the initial partitioning phase, instead of calling the serial Wavefront Diffusion algorithm to balance

	SR	LMSR	WD	MLDD
Graph Coarsening				
Matching scheme	global	local		
Initial Partitioning				
Partitioning scheme	partitioned from scratch	partitioned from scratch	inherited	inherited
Pre-refinement remapping	no	yes	no	no
Pre-refinement diffusion	no	no	WD	MLDD
Multilevel Refinement				
Refinement scheme	edge-cut aware	edge-cut and TOTALV aware		
Post-refinement remapping	yes	no		

Table 2: A summary of the parallel implementations of the four repartitioning schemes compared in the following experimental results.

the partitioning, all of the processors call the multilevel diffusion phase of the MLDD algorithm described in [15] and summarized in Section 2.4.

Table 2 summarizes the differences between the four algorithms.

6.1 Experimental Results on Synthetic Graphs

In this section, we present experimental evaluations of the LMSR, WD, SR, and MLDD algorithms on a 128-processor Cray T3E. The experimental results shown in Figures 14 through 17 are from a set of test graphs that are larger, but otherwise very similar in nature to those used for the serial experiments. The method we used to construct these test graphs is a slight modification of that described in Section 2.5. This modification allowed us to efficiently construct the test graphs in parallel.

Each figure contains eight sets of three results. Each set contains results on 32, 64, and 128 processors for a given graph and value for α. The first set is for mrng.C, a four million node finite element graph, with α set to five. The next three sets are on mrng.C with α set to 10, 20, and 30. Sets five through eight are on mrng.D, an eight million node finite element graph. Again α increases from five to 30 (moving left to right). For all of the experiments, a partitioning in which no domain contains 105% of the average domain weight is considered to be well-balanced. All of the repartitioning schemes were able to compute well-balanced partitionings for every experiment.

6.1.1 SR vs. LMSR

Figure 14 shows the TOTALV and MAXV results of SR and LMSR. In this figure, the results from our LMSR algorithm are normalized by those obtained from the SR algorithm. Hence, a bar below the 1.0 index line indicates that our LMSR algorithm obtained better results than SR.

Figure 14 shows that the LMSR algorithm obtained TOTALV results that are generally lower than those obtained by SR by up to 48%. Furthermore, as the number of processors increases or as the partitionings become more imbalanced, the TOTALV results of the LMSR algorithm tend to decrease relative to those of the SR algorithm. Comparing these results with those given in Figure 8 shows that the TOTALV improvement is significantly less for the LMSR algorithm compared to the SR algorithm in Figure 14 than for the local matching algorithm compared to the global matching algorithm in Figure 8. This is because the SR algorithm

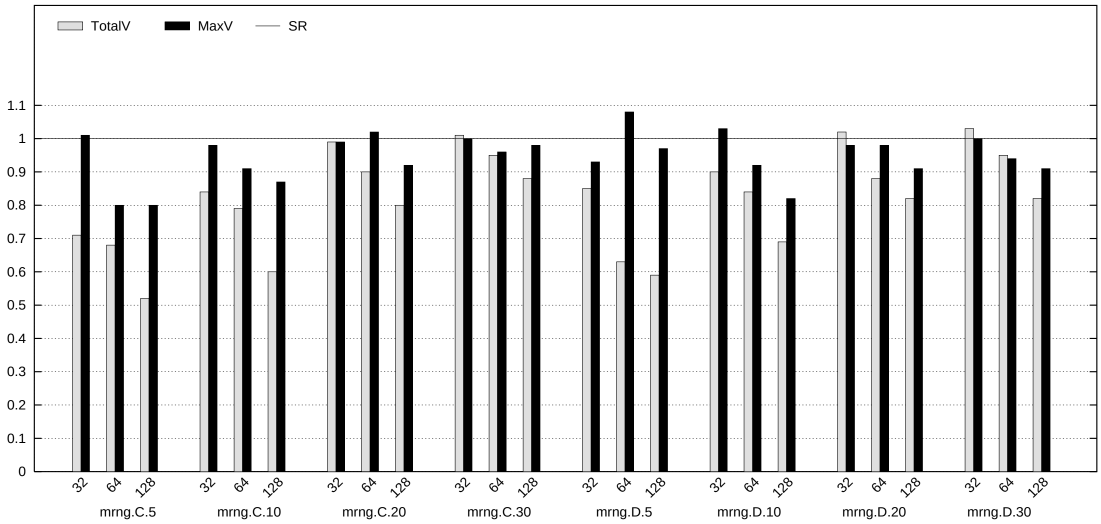

Figure 14: A comparison of TotalV and MaxV results obtained from the LMSR and SR algorithms.

used here (PARPAMETIS) utilizes a matching scheme that favors local matching over global matching. Therefore, it performs better than the SR algorithm in Figure 8 that uses a purely global matching algorithm.

Figure 14 also shows that LMSR generally obtains better MaxV results than SR. Specifically, in 18 out of 24 cases the LMSR algorithm obtained better MaxV results than the SR algorithm. In 4 out of 24 cases, the SR algorithm obtained better MaxV, and in other two cases, the MaxV results were similar.

Furthermore, the edge-cut and run time results of both of these schemes (not presented here) are similar. Thus, our LMSR algorithm is able to obtain generally better TotalV and MaxV results than the scratch-remap algorithm while computing high-quality, well-balanced partitionings.

6.1.2 WD vs. LMSR vs. MLDD

Figures 15, 16, and 17 compare the TotalV, edge-cut, and MaxV results obtained by the multilevel directed diffusion algorithm (MLDD) described in [15] and implemented as PARDAMETIS in ParMETIS [12], with our LMSR and Wavefront Diffusion algorithms. In these figures, the results from MLDD and Wavefront Diffusion are normalized against those obtained from our LMSR algorithm. Hence, a bar below the 1.0 index line indicates that the LMSR algorithm obtained worse results than the indicated algorithm.

TotalV Results Figure 15 compares the TotalV results of the three schemes. WD obtained consistently better results compared to LMSR and MLDD. The relative difference between the TotalV for WD and MLDD became increasingly higher as the level of imbalance increased. MLDD obtained generally better results than the LMSR scheme with the exception of a few highly imbalanced experiments. These results tend to converge for higher levels of imbalance.

Edge-cut Results Edge-cut results are compared in Figure 16. Figure 16 shows that our LMSR algorithm obtained edge-cut results that are better than both of the other schemes across the board. In most cases, the difference is within 20%. This shows that our LMSR algorithm is able to compute very high-quality

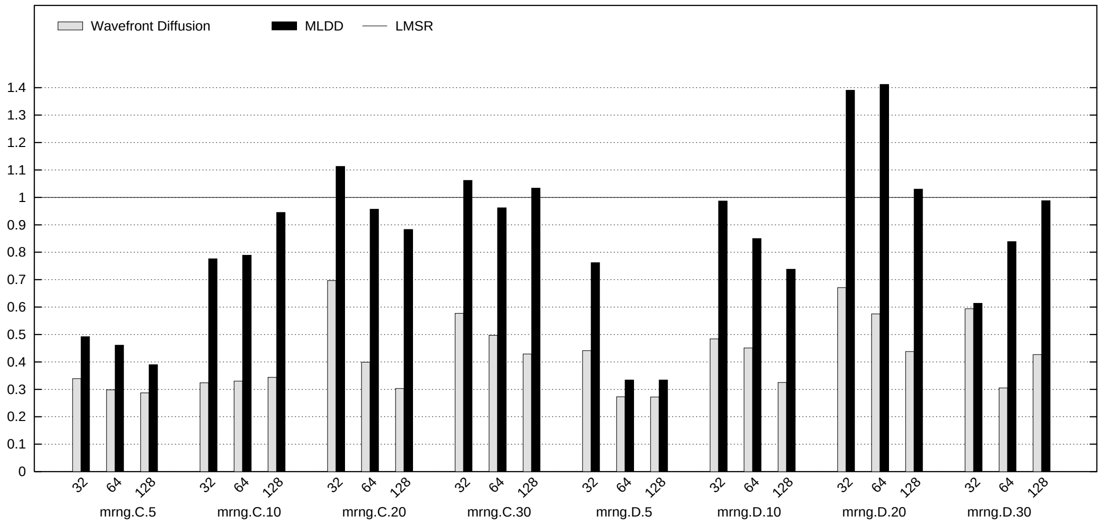

Figure 15: A comparison of TOTALV results obtained from the WD, MLDD and LMSR algorithms.

repartitionings regardless of the level of imbalance of the graph. Both of the diffusion-based schemes obtained results that are generally within 10% of each other.

Figure 16 shows that the two diffusion-based schemes obtained similar edge-cut results. This was not the case prior to multilevel refinement. That is, immediately following the multilevel diffusion phase of both schemes, the WD algorithm generally had lower edge-cut results than MLDD (similar to the results shown in Figure 12). However, the power of multilevel refinement allowed the MLDD algorithm, which had perturbed the edge-cut significantly, to catch up to the Wavefront Diffusion results. In fact, MLDD obtained slightly better edge-cut results than WD in the majority of the cases. However, by examining the TOTALV results from Figure 15, we see that for the cases in which MLDD obtained slightly better edge-cut results than WD, the WD algorithm tended to obtain significantly better TOTALV results. For example, in the mrng.D.10 results on 32 processors, the WD algorithm resulted in an edge-cut about 10% higher than MLDD but TOTALV that is almost 50% lower. Likewise for experiment mrng.C.20 on 128 processors, the WD algorithm resulted in an edge-cut 5% higher, but TOTALV that is 65% lower, than MLDD. The reason, as discussed in Section 2.4, is that the multilevel refinement algorithm will migrate only dirty vertices to improve the partition balance (and hence, give a potential decrease in edge-cut) if doing so does not also decrease the edge-cut. Thus, minimizing the number of dirty vertices can somewhat hinder the effectiveness of multilevel refinement to minimize edge-cut.

MAXV **Results** In Figure 17, MAXV results are compared. Wavefront Diffusion obtained generally better MAXV results than either of the other schemes across the board. However, this improvement is not as dramatic as it is for the TOTALV results. MLDD obtained mixed results compared with the LMSR scheme. Specifically, the MAXV results obtained from MLDD are 60% to 140% of those obtained by LMSR. In 14 out of 24 cases, MLDD obtained lower MAXV results than LMSR and in 10 out of 24 cases, the situation is reversed.

Run Times The run time results (not shown) of all four of the parallel schemes compared in this section are *extremely* fast. None of the run times for any of the schemes are over two seconds for the four million

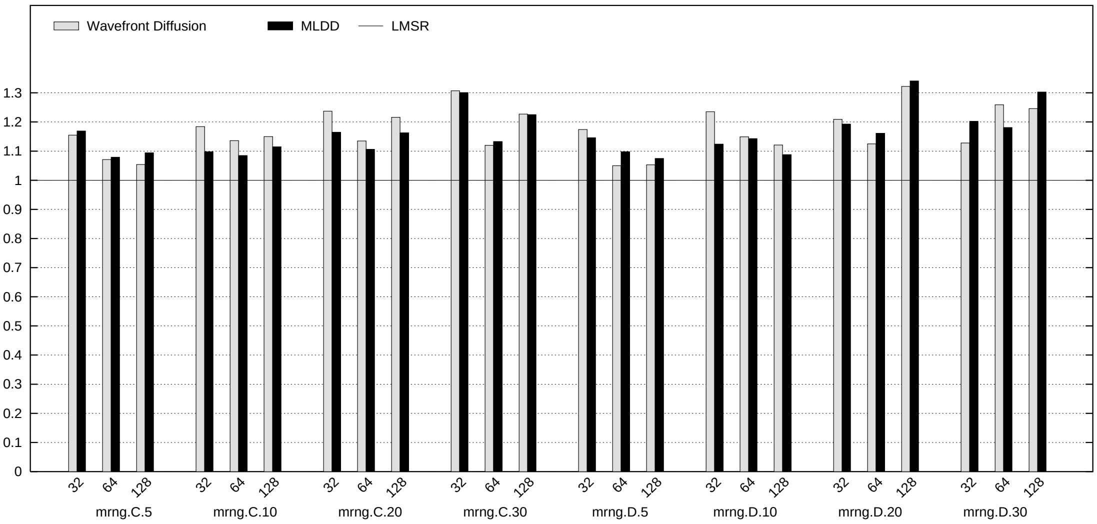

Figure 16: A comparison of edge-cut results obtained from the WD, MLDD and LMSR algorithms.

node graph or over three seconds for the eight million node graph on 128 processors. All of the schemes obtained run times for a given experiment within 30% of each other.

6.2 Experimental Results on Helicopter Blade Graphs

Experimental results given in Section 6.1 were for synthetically generated adaptive meshes. In this section, we present results from our schemes on an application domain. Figure 18 shows the results from a series of application meshes with a high degree of localized adaptation at each stage. These graphs are 3-dimensional mesh models of a rotating helicopter blade. As the blade spins, the mesh is adapted by refining it in the area where the rotor has entered and coarsening it in the area of the mesh where the rotor has passed through. These meshes are examples of applications in which high levels of adaptation occurs in localized areas of the the mesh. They were provided by the authors of [13].

Here, the first of a series of six graphs, $G_1, G_2, \ldots G_6$ was originally partitioned into eight domains with the multilevel graph partitioner implemented in ParMETIS [12]. 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*, and so on, through experiment *Fifth*.

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 application meshes. 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.

Figure 18 gives a comparison of the edge-cut, TotalV, and MaxV results of the five experiments, (followed by the sum of these) for WD, MLDD, LMSR, and SR. The results obtained by WD, MLDD, and LMSR 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.

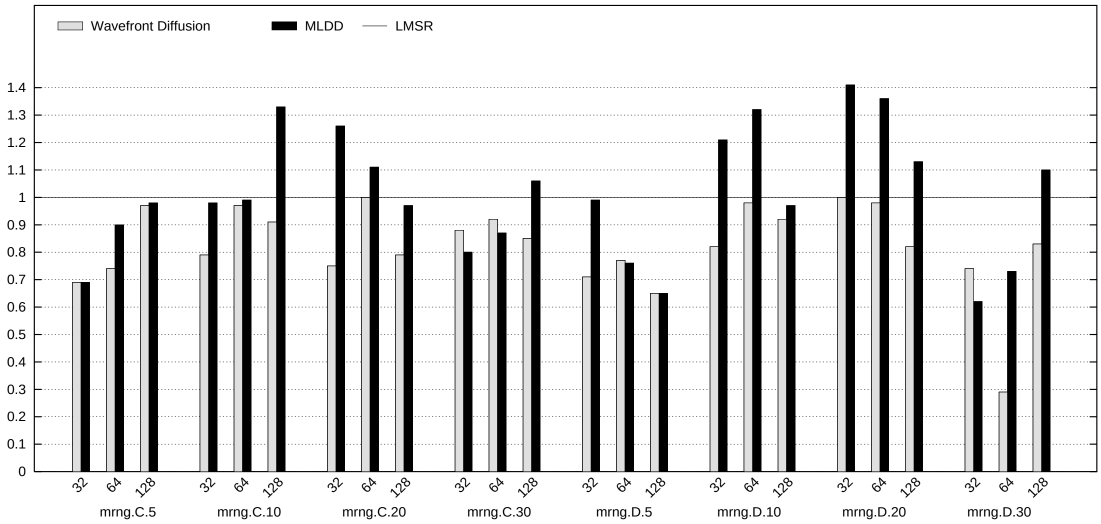

Figure 17: A comparison of MAXV results obtained from the WD, MLDD and LMSR algorithms.

Figure 18 shows that the WD scheme obtained better edge-cut results than MLDD. Specifically, Wavefront Diffusion obtained edge-cut results that are on average 20% less than MLDD. At the same time, the WD algorithm obtained lower TOTALV results on average compared to MLDD. Specifically, the Wavefront Diffusion scheme obtained TOTALV results that are on average lower than those obtained by MLDD by 60%. Wavefront Diffusion also obtained generally lower MAXV results. On average, these are 40% lower than MLDD MAXV results. These results confirm that WD is able to obtain lower edge-cut, TOTALV, and MAXV results than MLDD for graphs that are highly imbalanced in localized areas.

The two scratch-remap schemes obtained similar edge-cut and MAXV results. However, the LMSR scheme obtained significantly better TOTALV results compared to SR. The LMSR algorithm obtained TOTALV results that are less than 80% of those obtained by SR on average.

Comparing the scratch-remap schemes with the diffusion-based schemes, we again see a tradeoff between vertex migration costs and edge-cut.

7 Conclusions

In this paper, we have presented two new repartitioning algorithms. We have shown that Wavefront Diffusion obtains lower edge-cut and vertex migration results than current state-of-the-art diffusion-based repartitioners, especially for graphs that are highly imbalanced in localized areas. We have also shown that LMSR obtains lower vertex migration results than current state-of-the-art scratch-remap repartitioners, and that the difference between the schemes tends to increase for less imbalanced graphs.

Compared against each other, these two schemes present a clear tradeoff between edge-cut and TOTALV. That is, the Wavefront Diffusion algorithm minimizes the amount of vertex migration required to realize the balanced repartitioning, while the LMSR algorithm minimizes the edge-cut of the repartitioning.

It is important to note that the Wavefront Diffusion algorithm has a serial phase (multilevel diffusion) that increases in complexity linearly to the number of processors. While its run time results are similar to those

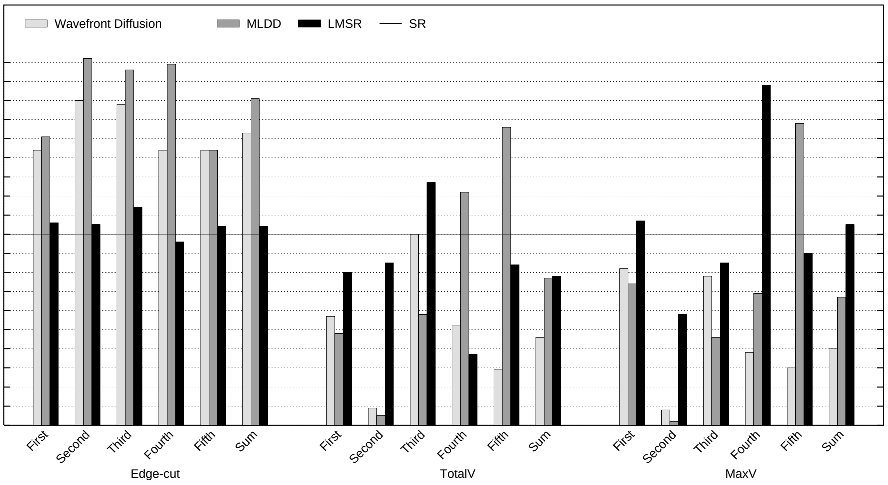

Figure 18: A comparison of edge-cut, TotalV, and MaxV results obtained by the WD, MLDD, LMSR, and SR algorithms on a series of application meshes of a helicopter blade rotating through space.

of the other schemes compared on up to 128 processors in our experiments, if scalability to a larger number of processors is of key concern, other repartitioning algorithms may be appropriate. Utilizing graph-based, undirected diffusion algorithms [15], for example, may allow a tradeoff between edge-cut, vertex migration results, and algorithm scalability. We suggest improvements and scalability performance studies of existing undirected diffusion algorithms as a topic of future research.

The parallel repartitioning algorithms described in this paper are available in the ParMETIS graph partitioning library that is publicly available on the WWW at http://www.cs.umn.edu/~metis.

References

[1] R. Biswas and L. Oliker. Experiments with repartitioning and load balancing adaptive meshes. Technical Report NAS-97-021, NASA Ames Research Center, Moffett Field, CA, October 1997.

[2] J. E. Boillat. Load balancing and poisson equation in a graph. *Concurrency: Practice and Experience*, 2:289–313, 1990.

[3] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Journal of Parallel and Distributed Computing*, 7(2):279–301, 1989.

[4] R. Diekmann, A. Frommer, and B. Monien. Nearest neighbor load balancing on graphs. Technical report, University of Paderborn, Department of Mathematics and Computer Science, 1998.

[5] Bruce Hendrickson and Robert Leland. A multilevel algorithm for partitioning graphs. Technical Report SAND93-1301, Sandia National Laboratories, 1993.

[6] G. Horton. A multi-level diffusion method for dynamic load balancing. *Parallel Computing*, 9:209–218, 1993.

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

[8] G. Karypis and V. Kumar. A coarse-grain parallel multilevel k-way partitioning algorithm. In *Proceedings of the 8th SIAM conference on Parallel Processing for Scientific Computing*, 1997.

[9] G. Karypis and V. Kumar. METIS 3.0: Unstructured graph partitioning and sparse matrix ordering system. Technical Report 97-061, Department of Computer Science, University of Minnesota, 1997. Available on the WWW at URL *http://www.cs.umn.edu/~metis*.

[10] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1), 1998.

[11] G. Karypis and V. Kumar. A fast and highly quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 1998, to appear. A short version appears in Intl. Conf. on Parallel Processing 1995.

[12] G. Karypis, K. Schloegel, and V. Kumar. PARMETIS: Parallel graph partitioning and sparse matrix ordering library. Technical report, University of Minnesota, Department of Computer Science and Engineering, 1997.

[13] L. Oliker and R. Biswas. Plum: Parallel load balancing for adaptive unstructured meshes. *Journal of Parallel and Distributed Computing*, 52(2):150–177, 1998.

[14] S. Plimpton, S. Attaway, B. Hendrickson, J. Swegle, C. Vaughan, and D. Gardner. Transient dynamics simulations: Parallel algorithms for contact detection and smoothed particle hydrodynamics. *Journal of Parallel and Distributed Computing*, 1998. To appear.

[15] K. Schloegel, G. Karypis, and V. Kumar. Multilevel diffusion schemes for repartitioning of adaptive meshes. *Journal of Parallel and Distributed Computing*, 47(2):109–124, 1997.

[16] 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 Conference on Parallel and Distributed Computing Systems*, pages 59–66, September 1998. http://www.cs.umn.edu/˜karypis.

[17] A. Sohn, R. Biswas, and H. Simon. Impact of load balancing on unstructured adaptive grid computations for distributed-memory multiprocessors. *Proc. 8th IEEE Symposium on Parallel and Distributed Processing (SPDP)*, pages 26–33, 1996.

[18] C. Walshaw, M. Cross, and M. G. Everett. Parallel dynamic graph partitioning for adaptive unstructured meshes. *Journal of Parallel and Distributed Computing*, 47(2):102–108, 1997.

[19] C. Z. Xu and F. C. M. Lau. The generalized dimension exchange method for load balancing in k-ary ncubes and variants. *Journal of Parallel and Distributed Computing*, 24:72–85, 1995.