Multilevel Diffusion Schemes for 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

For a large class of irregular mesh applications, the structure of the mesh changes from one phase of the computation to the next. Eventually, as the mesh evolves, the adapted mesh has to be repartitioned to ensure good load balance. If this new graph is partitioned from scratch, it may lead to an excessive migration of data among processors. In this paper, we present schemes for computing repartitionings of adaptively refined meshes that perform diffusion of

*This work was supported by NSF CCR-9423082, by Army Research Office contracts DA/DAAH04-95-1-0538 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**

vertices in a multilevel framework. These schemes try to minimize vertex movement without significantly compromising the edge-cut. We present heuristics to control the tradeoff between edge-cut and vertex migration costs. We also show that multilevel diffusion produces results with improved edge-cuts over single-level diffusion, and is better able to make use of heuristics to control the trade-off between edge-cut and vertex migration costs than single-level diffusion.

1 Introduction

Mesh partitioning is an important problem which has applications in many areas, including scientific computing. In irregular mesh applications, the computation associated with the mesh can be represented by a graph that has weights associated with its vertices and edges. Weight on the vertices of the graph represents the amount of computation, and the weight of the edges represents the amount of interaction between the computations associated with the vertices. Efficient parallel execution of these irregular grid applications requires the partitioning of the associated graph into p parts with the following two objectives: (i) each partition has an equal amount of total vertex weight; (ii) the total weight of the edges cut by the partitions (thereafter referred to as ***edge-cut***) is minimized. Since the weight of any given edge represents the amount of communication required between nodes, minimizing the number of edges cut by the partition tends to minimize the overall amount of communication required by the computation. This problem has been well defined and discussed in previous works [3, 6, 12, 11].

For a large class of irregular grid applications, the computational structure of the problem changes in an incremental fashion from one phase of the computation to another. For example, in adaptive meshes computations [1], areas of the original mesh are selectively refined or de-refined in order to accurately model the dynamic computation. This changes the amount of work which is required to be performed on each processor in the next computation phase. On a parallel computer, this can result in an uneven distribution of work, making it necessary to repartition and redistribute the adapted mesh across the processors. This repartitioning algorithm should satisfy the following objectives.

1. **It robustly balances the graph.**

   If each partition does not have roughly equal vertex-weight, then the overall parallel run-time will be dominated by the processor containing the highest weight, resulting in higher parallel run time. In order to make the repartitioning algorithm general it must be able to balance graphs from a wide variety of application domains.

2. **It minimizes edge-cut.**

   The redistributed graph should have a small edge-cut to minimize communication overhead in the follow on computation.

3. **It minimizes vertex migration time**.

Once the mesh is repartitioned, and before the computations can restart, data associated with the migrated vertices also need to be moved. In many adaptive computations, the data associated with each vertex is very large. The time for movement of the data can dominate overall run time, especially if the mesh is adapted frequently.

4. **It is fast and scalable.**

   The computational cost of repartitioning should be small since it is done frequently. Also, since the problem studied in this paper is relevant only in the context of parallel computers, the repartitioning algorithm should have an efficient parallel formulation. Performing the repartitioning on a serial processor can become a very serious bottleneck[1].

Objectives 1 and 2 can be optimized reasonably if the adapted graph is partitioned from scratch using a state-of-the-art multilevel graph partitioner such as METIS [9, 12, 11] or Chaco [6]. Since highly parallel formulations of multilevel graph partitioning algorithms are available [10, 13], criterion 4 can also be met to a large extent. Partitioning from scratch will, however, result in high vertex migration, as the partitioning does not take the initial location of the vertices into account. A partitioning method that incrementally constructs a new partition as simply a modification of the input partition (e.g. by diffusion [19, 18]) can potentially move a much smaller number of vertices. Such a method can also be potentially faster than partitioning the graph from scratch.

Repartitioning schemes that incrementally modify an existing partition have been quite successful on graphs that are small perturbations of the original graphs [15, 19]. For only slightly imbalanced graphs, the initial partition does not need to be disturbed very much, and so these algorithms are able to maintain an edge-cut comparable with the initial partition. However, if the initial partition is highly imbalanced, then many vertices need to move in order to balance the graph. Thus, even if the disturbance to the initial partition is minimized, the final partition will necessarily end up quite a bit removed from it. Hence, the balancing phase of such a method will increase the edge-cut considerably. Local refinement [15, 19] can only provide a limited improvement in the edge-cut of the resulting partition.

One promising solution to the problem of edge-cut degradation for highly imbalanced graphs is the use of a multilevel scheme that takes the initial location of the vertices into consideration. The multilevel paradigm allows the local refinement to be performed at multiple coarsened versions of the graph, which has been shown to be quite effective in reducing the edge-cut [3, 6, 12, 11]. In addition to the refinement, the movement of graph vertices (to achieve load balance) can also be done at multiple coarsened versions of the graph. This multilevel diffusion scheme can move large chunks of vertices at coarser levels, and then achieve better load balance at finer levels.

In this paper, we present repartitioning algorithms based upon multilevel diffusion. These algorithms construct a series of contracted graphs by collapsing pairs of vertices together that belong to the same initial partition. Once a suf-

[1] The algorithm should also have limited memory requirements, as any memory used by the partitioner cannot be used by the application.

ficiently small graph has been constructed, the partition is balanced via undirected diffusion (*i.e.*, diffusion in which balancing decisions are made by purely local views of the graph) or via directed diffusion (*i.e.*, diffusion which is directed by a global view of the graph) [8] at different levels. Once the graph is balanced, then multilevel refinement is performed at the remaining levels in order to clean up the edge-cut disturbed by the balancing phase. We further describe two heuristics which are able to control the tradeoff between edge-cut and vertex migration costs when used in a multilevel context. Our results show that multilevel diffusion produces results with improved edge-cuts over single-level diffusion and is better able to make use of heuristics to control the trade-off between edge-cut and vertex migration costs than single-level diffusion. Our results also show that the parallel formulation of multilevel diffusion is similar to that of multilevel k-way graph partitioning [10, 13] in performance and scalability. Walshaw *et al.*, have previously implemented a form of multilevel diffusion in JOSTLE-MD [18], without the heuristics for trading off the vertex migration and edge-cut.

The organization of this paper is as follows. Section 2 describes the notations and definitions which we use throughout this paper. Section 3 reviews previous related work. Section 4 describes our multilevel diffusion repartitioning algorithms in depth. Section 5 gives experimental results of our multilevel repartitioners. Section 6 describes and gives experimental results for our heuristics to control vertex migration costs. Section 7 gives experimental results of repartitioning graphs from two application domains. Section 8 contains results on the parallel execution of our multilevel diffusion scheme. Finally, Section 9 provides some concluding remarks. A preliminary version of this paper appeared in [16].

2 Notations, Definitions, and Issues

In our discussion we include the concepts of both vertex weight and vertex size as introduced in [17, 14]. 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 graph with respect to vertex weight while minimizing vertex migration with respect to vertex size. Depending on the representation and storage policy of the data, size and weight may not necessarily be equal. One example of such a situation arises in [17, 14]. A method for reducing the vertex migration overhead used in [17, 14] is to determine both the coarsening and the refinement of the adaptive mesh prior to repartitioning, but to actually perform only mesh coarsening at this time. This causes the graph to shrink prior to repartitioning. After repartitioning and subsequent data migration, the previously determined refinement of the adaptive mesh can be performed. In this way the cost of migrating the newly created vertices which have been selected to swap processors by the repartitioner need not be paid.

Let $G = (V, E)$ be an undirected graph of V vertices and E edges. Let p be the number of partitions. Let s_i represent the cost of movement of vertex v_i. We will refer to s_i as the ***size*** of vertex i. Let w_i represent the weight (*i.e.*,

computational work) of vertex v_i and $w_e(v_1, v_2)$ equal the amount of communication required between v_1 and v_2. A vertex's ***density*** is equal to its weight divided by its size. We denote $B(q)$ as the set of vertices with partition q. The weight of any partition q can then be defined as

$$W(q) = \sum_{v_i \epsilon B(q)} w_i$$

and so the average partition weight is

$$\overline{W} = \frac{\sum_{i=1}^{p} W(i)}{p}.$$

A graph is ***imbalanced*** if it is partitioned and

$$\exists q \mid W(q) > \overline{W} \times (1 + \epsilon)$$

where ϵ is a small constant. If $\epsilon = 0$, then all partitions would have to be exactly equal in weight in order for the graph to be balanced. However, our results indicated that this is often too strict a definition. For this paper, we set ϵ equal to .03.

In an imbalanced graph, a partition is ***overbalanced*** if its weight is greater than the average partition weight times $1 + \epsilon$. Likewise, a partition is ***underbalanced***, if its weight is less than the average partition weight divided by $1 + \epsilon$. Otherwise, partitions are ***balanced***. The graph is ***balanced*** when no partition is overbalanced. We will use the term ***repartitioning*** when an existing partition is used as an input in an algorithm in order to find a new partition on the same graph and the term ***partitioning*** when no input partition is used.

A vertex is ***clean*** if its current partition is its initial partition on the input graph. Otherwise it is ***dirty***. A vertex is a ***border*** vertex if one of its adjacent vertices is in another partition. If so, then all such partitions are the vertex's ***neighbor*** partitions. If a partition contains at least one vertex which has another partition as a neighbor partition, then those two partitions are neighbor partitions to each other.

TOTALV is defined as the sum of the sizes of vertices which change partitions as the result of partitioning or repartitioning [14]. Thus, it is the sum of the sizes of the dirty vertices. MAXV is defined as the maximum of the sums of the sizes of those vertices which migrate into or out of any one partition as a result of partitioning or repartitioning [14]. As discussed in [14], in many cases MAXV reflects the time for vertex migration more accurately than TOTALV.

3 Repartitioning Strategies: Review of Previous Work

A repartitioning of a dynamic graph can be computed by simply partitioning the new graph from scratch. However, since no concern is given for the existing partition, most vertices are not likely to be assigned to their initial partitions

with this method. Intelligent remapping of the resulting partition can reduce the required movement of vertices [17, 14], but vertex migration can still be quite high.

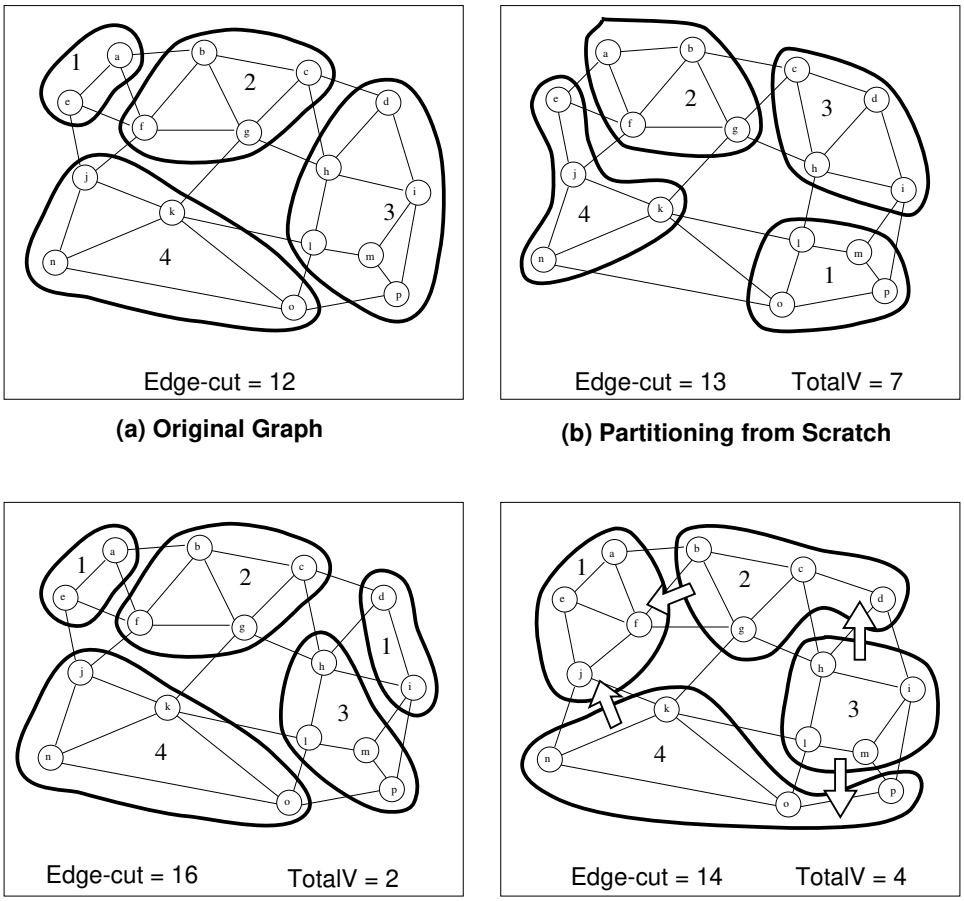
The second strategy is to use the existing partitioning as input for a repartitioning algorithm and to attempt to minimize the difference between the original partition and the output partition. This strategy can result in much smaller vertex migration compared to schemes that partition the modified graph from scratch. TOTALV can be minimized if only a subset of vertices, the sum of whose weight equals the difference between the average partition weight and the actual partition weight, are migrated out of any one partition. This can be trivially accomplished by the following ***cut-and-paste repartitioning*** method: Excess vertices in an overbalanced partition are simply swapped into one or more underbalanced partitions in order to bring these partitions up to balance. However, while this method will optimize TOTALV, it will have an excessively negative effect on the edge-cut compared with more sophisticated approaches.

Another method that reduces edge-cut degradation over cut-and-paste repartitioning, while increasing TOTALV only moderately, is analogous to diffusion from thermal dynamics. The concept is for vertices to move from overbalanced partitions to neighboring underbalanced partitions and to eventually reach balance, just as in the analogous case, uneven temperatures in a space cause the movement of heat towards equilibrium [8].

Figure 1 illustrates these methods for a graph whose vertices and edges have weight of one. In Figure 1(a), the original graph is imbalanced because partition 3 has a partition weight of 6, while the average partition weight is only 4. Edge-cut for the original graph is 12. In Figure 1(b), the original partition was thrown out and the graph was then partitioned from scratch resulting in an edge-cut of 13. This edge-cut is almost as low as that of the original partition. However, TOTALV is 7. This is because many vertices had to be migrated because they were assigned to a new partition which was different from their original partitions. In Figure 1(c), cut-and-paste repartitioning was used. Here, TOTALV is 2, since vertices d and l migrate to partition 1. The edge-cut is now 16. In Figure 1(d), a diffusion-type repartitioning was conducted. Vertex movement increases to 4, but edge-cut drops to 14 in comparison with the cut-and-paste method. Notice that partition 3 migrates vertex d to partition 2 and vertex p to partition 4. This, in turn, causes the recipient partitions to become imbalanced. They then migrate vertices j and f to partition 1. At this point the graph is balanced.

From these examples, we see an illustration of how cut-and-paste repartitioning minimizes TOTALV while completely ignoring edge-cut. Likewise, partitioning the graph from scratch minimizes edge-cut while resulting in high TOTALV. Diffusion, however, attempts to keep TOTALV low by ensuring that the vertices which do not need to be migrated to balance the graph are reassigned to their original partitions. It also attempts to keep edge-cut low by making incremental changes to the current partition.

Undirected diffusion is diffusion which occurs through distributed actions employing only local views of the graph. Thus, vertex migration decisions are made at every partition according to the relative difference in partition weights between each partition and all of its neighbor partitions. Undirected diffusion has the advantage that it is highly dis-



(a) Original Graph **(b) Partitioning from Scratch**

(c) Cut-and-Paste Repartitioning **(d) Diffusion Repartitioning**

Figure 1: Partitioning and Repartitioning Examples

tributed in nature. However, balancing occurs without the guidance of a global view of the graph. This can potentially increase the edge-cut, vertex migration costs, and run time of the algorithm. Diffusion has been studied in the general context of load balancing in [2, 4, 21].

Directed diffusion is diffusion guided by a global view of the graph. It is accomplished by obtaining a ***diffusion solution*** that specifies the number of vertices to be moved [7, 8, 19] for every pair of partitions. Two methods of computing the diffusion solution involve minimization of the one-norm of the diffusion solution and minimization of its two-norm. One-norm minimization is the minimization of the sum of the moduli of the elements of the diffusion solution vector. Two-norm minimization is a minimization of the sum of the squares of the elements of the diffusion solution. Figure 2 shows two different diffusion based solutions for a graph in which partition A and B are overbalanced and partitions E and F are underbalanced. Arrows indicate vertex flow. The numbers next to the arrows indicates the magnitude of this flow. The solution in Figure 2(a) minimizes the one-norm by assigning all of the vertex flow on the shortest route available. Thus, TOTALV is minimized. However, one-norm minimization does not guarantee the minimization of MAXV. This can be seen if we focus on partition G of Figure 2(a). This partition receives all of the

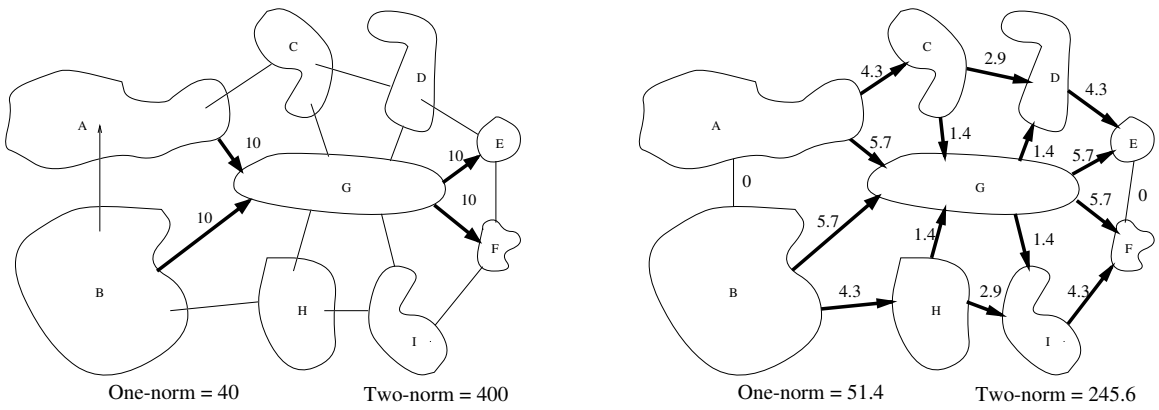


Figure 2: One- and Two-Norm Diffusion Examples

vertex flow from both overbalanced partitions. The total vertex weight both into and out of partition G is 20. The lower bound for MAXV here is 10. Thus, MAXV is twice the minimal necessary to balance the graph. The solution in Figure 2(b) minimizes the two-norm. Here the vertex flow is split among the available channels. Hence, the channel-use is more efficient. The two-norm minimization tends to minimize MAXV, at the expense of higher TOTALV.

Ou and Ranka [15] developed a method which optimally minimizes the one-norm of the diffusion solution using linear programming. Hu and Blake [8] described a method which computes the diffusion solution while optimally minimizing the two-norm. They proved that this solution can be found by solving the linear equation

$$L\lambda = b$$

where b is the vector containing the load of each partition minus the average partition load, and L is a Laplacian matrix, defined as

$$(L)_{qr} = \begin{cases} -1, & \text{if } q \neq r, \, q \text{ and } r \text{ are neighbors,} \\ deg(q), & \text{if } q = r, \\ 0, & \text{otherwise.} \end{cases}$$

and λ, the diffusion solution, is a vector with p elements. An amount of vertex weight equal to $\lambda_q - \lambda_r$ needs to be moved from partition q to partition r for every partition r which is adjacent to partition q in order for the graph to balance. A negative value indicates vertex flow in the opposite direction. Hu and Blake [8] showed that when using the parallel conjugate gradient algorithm [5] to solve for λ, the algorithm converges in fewer than p iterations.

Walshaw, Cross, and Everett implemented, JOSTLE, a combined partitioner and directed diffusion repartitioner based on an optimization of the Hu and Blake [8] diffusion solver [19].

4 Multilevel Graph Repartitioning

Multilevel graph partitioning has been studied in [3, 6, 12, 11]. Our multilevel graph repartitioning algorithm is essentially a modification of the k-way multilevel partitioning algorithm [11]. Hence, we first review the k-way multilevel scheme for partitioning. Throughout this paper, we will refer to the k-way multilevel graph partitioning algorithm implemented in METIS as simply METIS.

The k-way multilevel graph partitioning algorithm [11] implemented in METIS has three phases, a coarsening phase, a partitioning phase, and a refinement (or uncoarsening) phase[2]. During the coarsening phase, a sequence of smaller graphs are constructed from an input graph by collapsing vertices together. When enough vertices have been collapsed together so that the coarsest graph is sufficiently small, a k-way partition is found. Finally, the partition of the coarsest graph is projected back to the original graph by refining it at each uncoarsening level using a k-way partitioning refinement algorithm.

The k-way refinement algorithm in METIS uses a simple randomized algorithm that moves vertices among the partitions to reduce the edge-cut and to improve the balance. Consider a graph $G = (V, E)$. For each vertex $v \in V$ we define the ***neighborhood*** $N(v)$ of v to be the union of the partitions to which the vertices adjacent to v belong (excluding v's partition). Note that if v is a border vertex, then $N(v) \neq \emptyset$, and if v is an interior vertex, then $N(v) = \emptyset$. In the k-way refinement algorithm, each vertex of the graph is visited randomly. A vertex v is moved to one of the neighboring partitions in $N(v)$ if any of the following ***vertex migration criteria*** is satisfied.

1. The edge-cut is reduced while maintaining the balance.

2. The balance improves while maintaining the edge-cut.

If the first criterion is satisfied, then the algorithm tries to move v into a partition that will lead to a reduction in the edge-cut, subject to balance constraints. If multiple partitions satisfy this criterion, the vertex is moved to the one that lead to the highest reduction in the edge-cut. If the second criterion is satisfied (but the first is not), then the algorithm tries to move v to a partition that will improve the balance without changing the edge-cut. If multiple partitions satisfy this criterion, then the vertex is moved to the one that leads to the highest improvement in balance. Note that since each vertex is moved to its neighboring partitions, the k-way refinement algorithm moves only border vertices. This process is repeated a small number of times, or until we can obtain no further reduction in the edge-cut [11].

[2]Note that the meanings of the terms *coarsen* and *refine* here in the multilevel partitioning context is different from their meanings in the adaptive mesh context.

4.1 Repartitioning Algorithms Based on Multilevel Diffusion

A multilevel undirected diffusion repartitioning algorithm (MLD) as a modification of the multilevel k-way partitioning algorithm implemented in METIS can be derived as follows. In the coarsening phase, only pairs of nodes that belong to the same partition are considered for merging. Hence, the initial partition of the coarsest level graph is identical to the input partition of the graph that is being repartitioned, and thus does not need to be computed. This makes the coarsening phase completely parallelizable, as coarsening is local to each processor.

The uncoarsening phase of MLD contains two subphases: multilevel diffusion and multilevel refinement. In the multilevel diffusion phase, balance is sought on the coarsest graph in a process similar to multilevel refinement. This is accomplished by forcing the migration of vertices out of overbalanced partitions. The vertices are visited in a random order. Each border vertex is examined. If a vertex is in an overbalanced partition and is neighbors with a non-overbalanced partition, then that vertex will migrate to the non-overbalanced partition. If the vertex is neighbors with several non-overbalanced partitions, then it will migrate to the partition that produces the greatest improvement in edge-cut. The vertex is migrated even if this results in an increased edge-cut. After each border vertex is visited exactly once, the process repeats until either balance is obtained or no balancing progress is made.

Given this scheme, it may not be possible to balance the graph at the coarsest graph level. That is, there may not be sufficiently many vertices of small weight (*i.e.*, vertices composed of few sub-vertices) on the coarsest graph to allow for total balancing. If this is the case, the graph needs to be uncoarsened one level in order to increase the number of smaller weight vertices. The process described above is then begun on the next coarsest graph. Our experiments have shown that the graph will typically balance within a few levels.

After the graph is balanced, multilevel diffusion ends and multilevel refinement begins on the current graph. Here, the emphasis is on improving the edge-cut. The vertices are visited randomly. Each border vertex visited is checked to see if the migration of the vertex to a neighboring partition will satisfy one of the following ***refinement phase vertex migration criteria***.

1. The selected partition is the vertex's initial partition from the input graph, and the move does not increase the edge-cut, and the balance is maintained.

2. The edge-cut is reduced while maintaining the balance.

3. The balance improves while maintaining the edge-cut.

Criterion 1 allows vertices to migrate to their initial partitions (as long as the migration does not increase the edge-cut and worsen the load balance), and therefore, to lower TOTALV and possibly MAXV. If more than one of these criteria is satisfied, then priority is given to moving the vertex back to its initial partition, and then to reducing the edge-cut.

Our multilevel directed diffusion repartitioning algorithm (MLDD) is as follows. Coarsening is accomplished as described for MLD above. However, balance is sought by means of a global picture of the graph (*i.e.*, the 2-norm minimizing diffusion solution) guiding vertex migration. That is, the border vertices are visited randomly. If the visited vertex is neighbors with a partition which has a positive flow value according to the diffusion solution with respect to the vertex's current partition and this flow value is greater than 90% of the vertex's weight, then that vertex is migrated to the neighbor partition. If a vertex is neighbors with more than one such partition, it is migrated to that partition which will result in the lowest edge-cut. The vertex is migrated even if this results in an increased edge-cut. When a vertex is migrated, the flow value obtained by the diffusion solution for the two partitions is updated by decreasing it by the migrating vertex's weight. After each border vertex is visited exactly once, the process repeats until either balance is obtained or no balancing progress is made. Once balance is obtained, multilevel refinement is begun as described in the MLD algorithm above.

In summary, as illustrated in Figure 3, our multilevel diffusion repartitioning algorithms are made up of three phases, graph coarsening, multilevel diffusion, and multilevel refinement. The coarsening phase results in a series of contracted graphs. The multilevel diffusion phase balances the graph using the very coarsest graphs. The multilevel refinement phase seeks to improve the edge-cut disturbed by the balancing process. Optionally, the multilevel diffusion can be guided by a diffusion solution. We will refer to our multilevel undirected diffusion repartitioning algorithm as MLD and to our multilevel directed diffusion repartitioning algorithm as MLDD. Single-level directed diffusion (SLDD) will be used to provide a comparison with our multilevel diffusion schemes. In SLDD, diffusion and refinement are performed only on the original input graph and thus, no graph contraction is performed.

Walshaw, Cross, and Everett [18] have previously implemented a related scheme in JOSTLE-MD. The JOSTLE-MD algorithm performs both diffusion and refinement starting on the coarsest graph and continuing on every subsequently finer graph. Our scheme performs multilevel diffusion only on the coarsest graphs until balance is reached and then multilevel refinement only on the remaining finer graphs.

5 Experimental Results

The experiments in Sections 5 and 6 were performed using five different graphs arising in finite element applications. They are enumerated and described in Table 1. METIS was originally used on the input graphs to obtain a 128-way partition. Then the weights of some randomly selected vertices were increased so as to overbalance and underbalance certain partitions. Specifically, for one series of experiments four of the 128 partitions were overbalanced by 80%. This was accomplished by doubling the vertex weights of 80% of the vertices in each of the four selected partitions. In the next series of experiments, four partitions were overbalanced and four partitions were underbalanced in order to create source and sink partitions. Here, partition weights were modified by multiplying the vertex weight of each vertex in

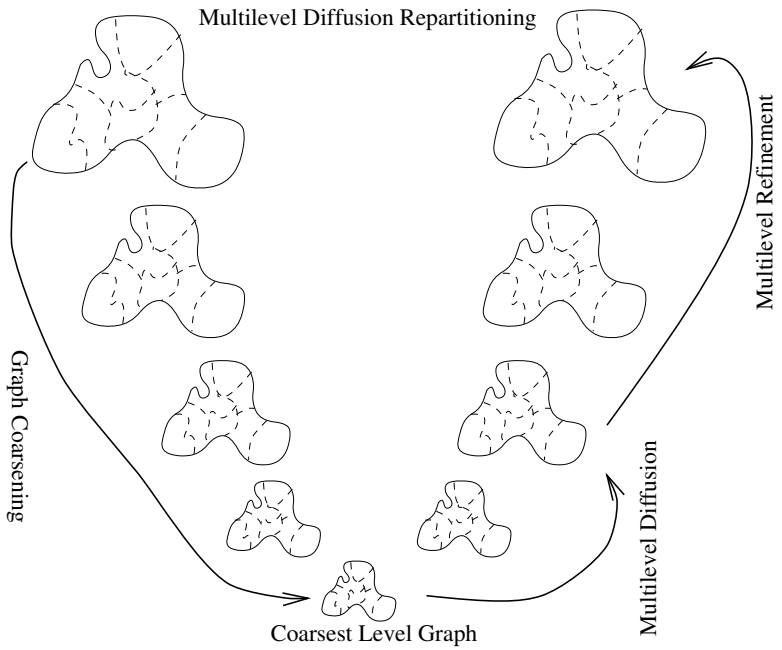


Figure 3: Multilevel Diffusion Repartitioning

Graph	Num of Verts	Num of Edges	Description
AUTO	448,695	3,314,611	3D mesh of GM's Saturn
m14b	214,765	1,679,018	3D mesh of submarine
MDUAL2	988,605	1,947,069	dual of a 3D mesh
TORSO	201,142	1,479,989	3D mesh of a human thorax
WAVE	156,317	1,059,331	3D mesh of a submarine

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

a given partition by a constant. All source partition vertex weights were multiplied by 19. All sink partition vertex weights were multiplied by 1. All others were multiplied by 10. Finally, in two series of experiments, source and sink partitions were created by multiplying the vertex weights of each vertex in a partition by a random number. These random numbers were distributed to produce an average vertex weight of 18 in source partitions, 2 in sink partitions, and 10 in all other partitions.

Figure 4 compares the results from single-level directed diffusion with two multilevel diffusion schemes. All of the results are normalized against the results obtained by partitioning the imbalanced graph from scratch using METIS.

Figure 4(a) shows the results of repartitioning using these three schemes on graphs which were overbalanced by 80% in four partitions. First we see that TOTALV and MAXV for all three of these schemes are much better compared with partitioning from scratch. This is expected, since METIS does not make use of the information provided by an input partition. Thus, it is highly unlikely that vertices are reassigned to their initial partitions. Figure 4(a) also shows that for this simple balancing problem, there is not much difference between the results from MLD, MLDD, and SLDD. These results confirm our hypothesis that for relatively simple balancing problems, SLDD is able to maintain a good

edge-cut. It is only for more complex imbalance problems that the SLDD algorithm begins to break down.

Figures 4(b), (c), and (d) illustrate this point. Figures 4(b) and (c) show the results of repartitioning on graphs with four source and four sink partitions. In Figure 4(b), every vertex in a partition has the same vertex weight as every other vertex in the partition. In Figure 4(c) however, the weight of every vertex in each partition was multiplied by a randomly generated number. Figure 4(d) shows the results of repartitioning on graphs with eight source and eight sink partitions and randomly distributed vertex weights. These results show that the multilevel directed diffusion algorithm is effective in keeping the edge-cut degradation and TOTALV down for complex balancing problems. MLDD consistently results in lower edge-cuts and TOTALV than SLDD and MLD in every experiment. The edge-cuts of SLDD and MLD are generally similar. With respect to MAXV, the MLDD scheme did not fair as well. In seven of the 20 results, the MLDD scheme resulted in MAXV results which were 10% to 50% higher than the other repartitioners. However, these were still lower than the MAXV results from partitioning from scratch.

The results indicate that the multilevel diffusion paradigm is quite powerful. Both multilevel diffusion algorithms (MLD and MLDD) are able to repartition each of the imbalanced graphs effectively. We see that multilevel directed diffusion is more effective at keeping edge-cut and TOTALV results down than multilevel undirected diffusion. However, this difference is not as great as that obtained when we compared the results (not shown in this paper) from single-level undirected diffusion to those of single-level directed diffusion. Here, edge-cut, TOTALV, MAXV and the repartitioning algorithm run time were all higher virtually across the board for single-level undirected diffusion compared to single-level directed diffusion.

6 Heuristics

In this section we present heuristics to lower TOTALV and MAXV while marginally sacrificing edge-cut. This can be very useful in those applications in which the time required to migrate vertices dominates the total execution time.

6.1 A Heuristic to Reduce TOTALV

As defined in Section 2, a vertex is dirty if it is currently in a partition different from its initial partition on the input graph. TOTALV then is the sum of the sizes of the dirty vertices. During the multilevel diffusion phase, a certain amount of vertices become dirty. This is unavoidable, as the graph must be balanced. These vertices can be migrated further, however, without increasing TOTALV. Hence, in the multilevel refinement phase, if only dirty vertices are migrated, TOTALV cannot increase further, and it may even decrease if dirty vertices find their way back to their original partitions. However, it appears overly restrictive to completely eliminate the migration of clean vertices, as it may result in higher edge-cuts. Nevertheless, it appears reasonable to restrict the migration of clean vertices. This is done by means of the

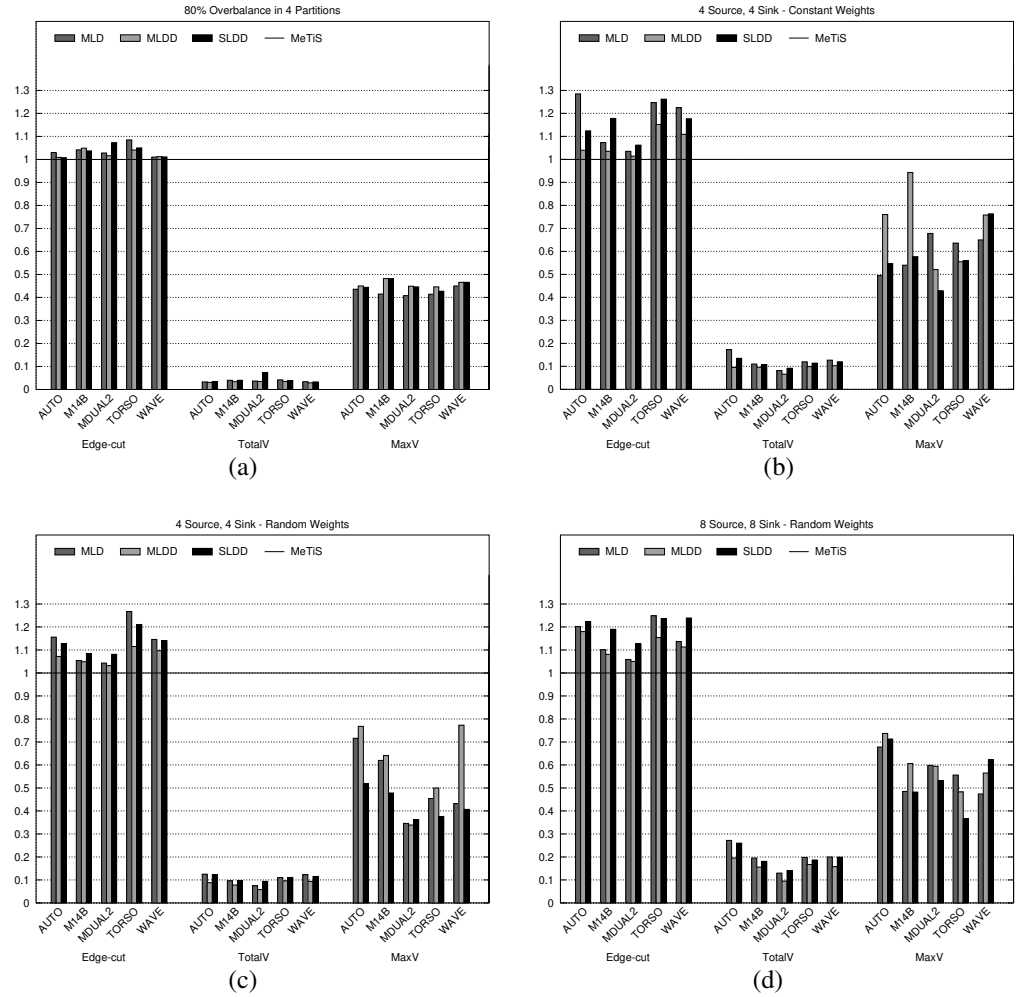


Figure 4: Repartitioning Results

cleanness factor (CF). During multilevel refinement, a clean vertex is moved only if

$$\frac{\text{decrease in edge-cut achieved by moving the vertex}}{\text{size of vertex}} > \text{CF}.$$

Thus, we limit the movement of clean vertices who result in only small edge-cut decreases relative to their size. If CF is set to infinity, then only dirty vertices are considered for migration during multilevel refinement. If CF is zero, then all vertices, clean and dirty, are considered and may be migrated even if they do not reduce the edge-cut. An interesting and important case is when CF $= \epsilon$, where ϵ is a small number such as 0.0001. In this case, a clean vertex is moved only if it reduces the edge-cut. Note that the refinement criteria of METIS allows the move of a vertex that maintains the edge-cut, but improves the balance. When CF $= \epsilon$, the second criterion is not applied for clean vertices.

Figure 5 shows the results of repartitioning using three different values for the cleanness factor. These experiments are performed on the same imbalance problems as described in Section 5. All of the results are from the multilevel

directed diffusion repartitioner with vertex cleanness and suppression (MLDD-CS). Each experiment is conducted with an input suppression factor heuristic of .5. The suppression factor heuristic is described in Section 6.2. The results are normalized against those obtained with the cleanness factor of zero.

In each of the results, TOTALV decreases as CF increases. This is as expected, as raising the cleanness factor decreases the number of vertices allowed to migrate during multilevel refinement. We also see a corresponding rise in the edge-cut as the cleanness factor increases. Thus, the results show that it is possible to lower TOTALV by trading edge-cut.

This decrease in TOTALV is able to affect MAXV in certain cases. Since, there is less total vertex migration, it stands to reason that the maximum vertex migration into or out of any one partition might also drop. However, this is not necessarily the case. In fact, MAXV could increase as CF increases. This would be the result when MAXV is dominated by the sum of the sizes of the vertices migrating into one partition. Since dirty vertices are free to migrate regardless of the cleanness factor, there is nothing stopping them (apart from balance constraints) from migrating into this partition. Doing so would, of course, increase the sum of the sizes of the vertices migrated into the partition. This would, in turn, increase MAXV as MAXV was equal to the prior sum.

6.2 A Heuristic to Reduce MAXV

MAXV is the max of the sum of the sizes of vertices into or out of any one partition. Our experiments have shown that the outgoing component of MAXV is usually not a concern. Intuitively, this is because vertices tend to migrate out of an overbalanced partition only until the partition is balanced. Furthermore, overbalanced partitions may have an ample supply of average to highly dense vertices. That is, they may have a good supply of vertices whose weight divided by their size is relatively high. Choosing high density vertices for movement whenever possible balances the graph while keeping the cost of vertex migration down. Even if vertices are selected randomly for migration in overbalanced partitions, there is a good chance that mostly relatively dense vertices will be migrated. Thus, the sum of the sizes of the outgoing vertices will be in the vicinity to the lower bound.

On the other hand, the max of the sum of the sizes of vertices which migrate ***into*** any one partition is potentially problematic. Underbalanced partitions must depend on neighbor partitions to migrate vertices into them. There is no guarantee that an underbalanced partition's neighbors will have a large supply of dense vertices to migrate. The worst case scenario is when two underbalanced partitions are neighbors and only one of these partitions is neighbors with an overbalanced partition. Figure 6 illustrates this point. Here partition A is overbalanced, partitions B and C are underbalanced, and partitions D and E are balanced. The diffusion solution will call for vertex migration as indicated by the arrows. Notice that partition B is supposed to migrate vertices into partition C. However, partition B may initially be full of relatively low-density vertices since it is also an underbalanced partition. It will take a much greater

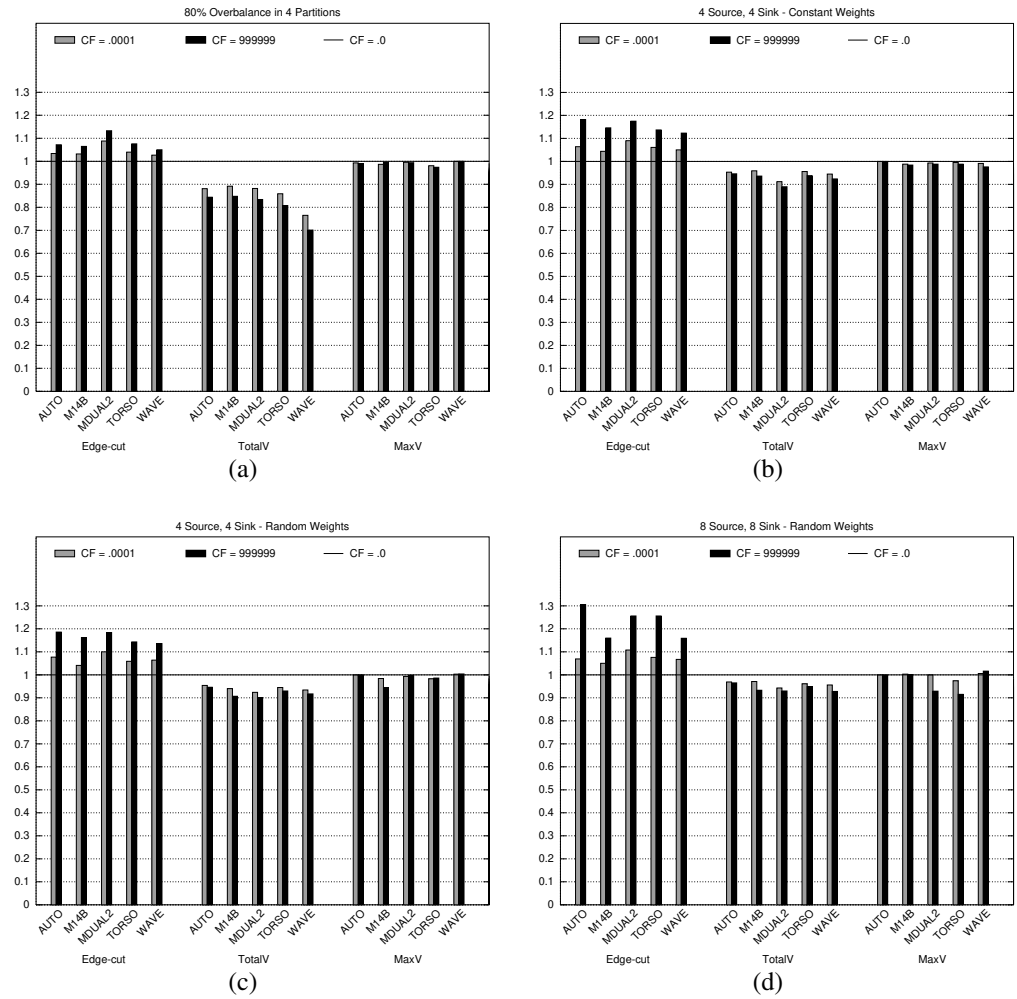


Figure 5: Repartitioning with Cleanness

number of low-density vertices to balance partition C than it would have taken average or highly dense vertices. If this happens, partition C will get an overabundance of inflowing, low-density vertices. These vertices will dominate MAXV. However, if the migration of these low-density vertices could be suppressed, the result would be that only average density vertices from partitions D and E would be able to migrate into partition C. Thus, MAXV would be reduced.

The underlying problem lies in the migration of low-density vertices. Partitions are balanced according to vertex weights. However, vertex migration costs are paid in terms of vertex size. Therefore, migrating vertices with relatively low weight-to-size ratios will tend to increase the vertex migration cost necessary to balance the graph. In order to avoid this situation, we suppress the movement of low density vertices by using a parameter called ***suppression factor*** (SF).

During multilevel diffusion a vertex v is considered for move only if

$$\frac{\text{Density of } v}{\text{Average Vertex Density}} > \text{SF}.$$

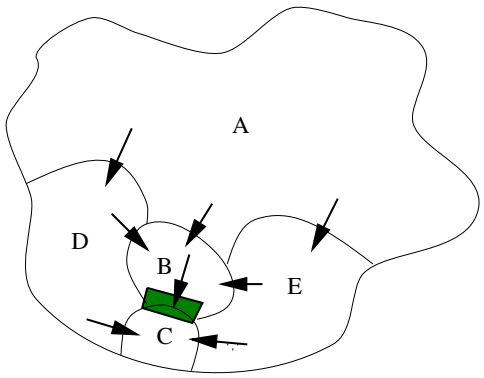


Figure 6: Blocking a Sink to Sink Transfer

Therefore, if SF is zero, no vertex migration is suppressed. If SF is set to infinity, all vertex migration is suppressed during multilevel diffusion. In this case it is, of course, impossible for the graph to balance, as no vertices are allowed to move. If SF is set to one, only vertices that are above the average density are allowed to migrate during balancing. The tradeoff here is that the larger the value of SF, the less free vertices are to migrate, and so the more difficult it is to balance the graph. With larger values of SF, the graph will tend to balance at higher and higher uncoarsening levels. This can also potentially degrade the edge-cut, as multilevel refinement is the key to keeping edge-cut low. (Multilevel refinement begins only after multilevel diffusion completes.) Thus, the more uncoarsening levels it takes to balance the graph, the fewer levels are available for refinement to reduce the edge-cut.

Figure 7 shows the results of repartitioning the imbalance problems (described above) using a range of values for SF. We have set the cleanness factor (CF) at a constant .0001 for each of these experiments. Here, the results are normalized against the results obtained from using SF equal to zero. We see that even small values for the suppression factor reduces MAXV by up to 55% for all but the first set of graphs. Meanwhile, across the board, edge-cut is increased by only a few percent. As SF increases, MAXV tends to decrease, while edge-cut increases. Thus, the results show that by employing vertex suppression in a multilevel context, it is possible to decrease MAXV by trading edge-cut.

An interesting side-effect occurs with respect to TOTALV. Since suppression keeps low-density vertices from migrating during multilevel diffusion, load balancing is accomplished through the migration of higher density vertices. Thus, TOTALV also tends to drop.

Notice in Figure 7(a) that suppression has had no effect. This is because for this problem, vertex density is highly homogeneous across partitions. The densities range from one to two here. The average vertex density for the graph is 1.03. Thus, in order to suppress the lowest density vertices (those of density one), the suppression factor will have to be greater than .97. Since 97.5% of the vertices in these imbalance problems are of density one, this suppression factor is much too large to allow the graph to be balanced. In fact, we conducted experiments with a suppression factor as large as one and none of the graphs consistently balanced.

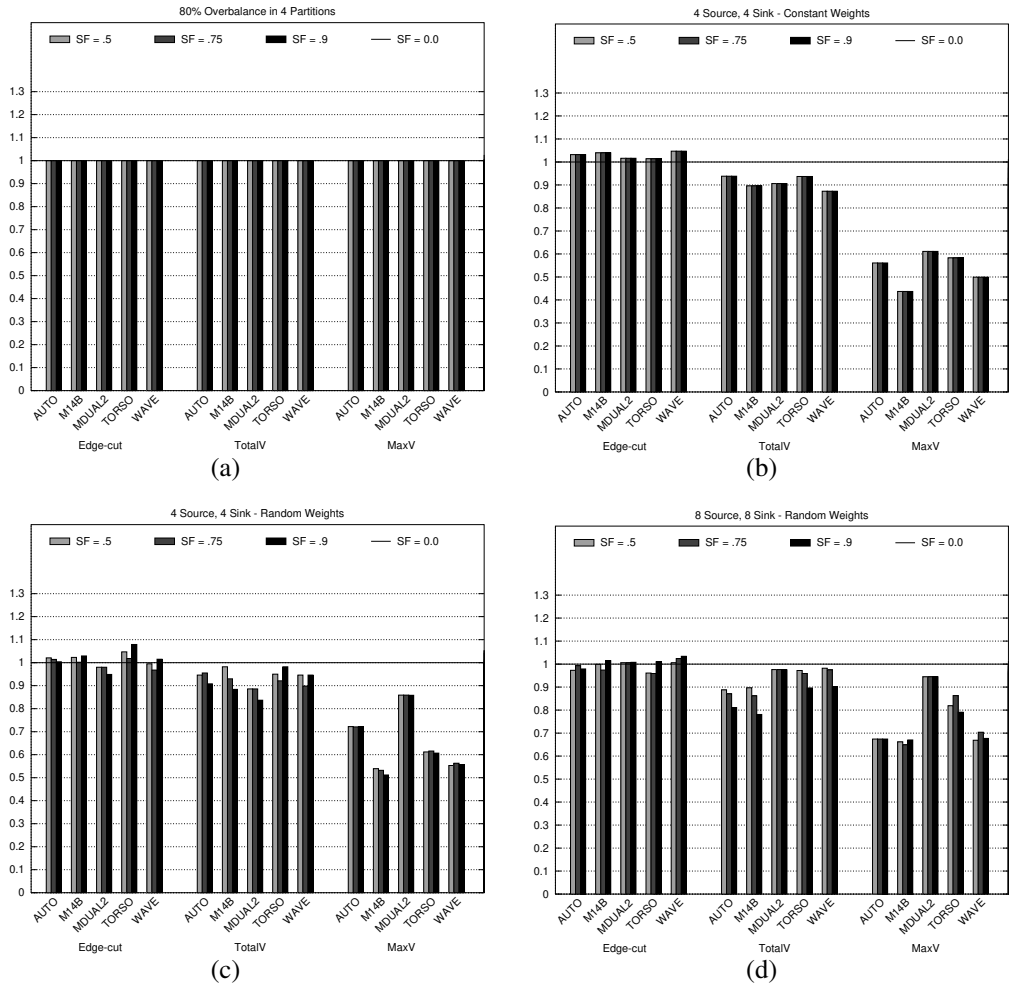


Figure 7: Repartitioning with Suppression

Figures 7(b) through (d) show that as the homogeneity of vertex density decreases, vertex suppression becomes more effective. However, by reexamining the results from Figure 4, we also see that as homogeneity decreases, MAXV becomes more problematic for the multilevel schemes. Thus, while vertex suppression is less effective on homogeneous graphs, it tends to be less necessary here, as well. That is, as the homogeneity decreases, MAXV becomes more problematic. At the same time however, vertex suppression becomes more effective. These results show that vertex suppression is a powerful heuristic for controlling MAXV.

6.3 Dynamic Suppression

As the previous results have shown, increasing the suppression factor tends to decrease MAXV. If the suppression factor is set too low, no vertices will be suppressed, and so vertex suppression will be ineffective. However, if this suppression factor is too high, the majority of vertices will be suppressed and the graph will not be balanced. If the characteristics of mesh adaptation are known in advance, then the suppression factor can be set at an appropriate level. However, if this

is not the case, then it may be difficult to set the suppression factor at an appropriate level. To handle these situations, we have implemented a scheme that dynamically adjusts SF as follows. At the beginning of each uncoarsening level, SF is initialized to some quantity (*e.g.*, 1). During multilevel diffusion, after every vertex has been visited, the dynamic suppression algorithm checks to see if at least 80% of vertices were suppressed. If this is the case, then the suppression factor is divided by 1.3 prior to the start of the next iteration. The next Section shows the results of using MLDD with dynamic suppression on two application domains.

7 Repartitioning of Application Graphs

We have conducted experiments on repartitioning application graphs from two domains. The first set is taken from the DIME software package [20]. The application solves Laplace's equation with Dirichelet boundary conditions on a square, 2-dimensional mesh with a stylized 'S' hole. The problem is solved by Jacobi iteration, refined, and load-balanced [18]. The result is a domain with a small degree of change at each successive stage in the mesh adaptation.

The second set shows a series of application meshes with a high degree of adaptation at each stage. These graphs are 3-dimensional mesh models of a rotating helicopter blade. As the blade spins, the mesh must be adapted by refining the mesh 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 were provided by the authors of [14].

For each of these application domains, the first of a series of x graphs, G_1, G_2, ... G_x was originally partitioned with METIS. The partition of graph G_1 acted as the input partition for graph G_2. Repartitioning this now imbalanced graph, G_2, resulted in the experiment named *First* and the input partition for graph G_3. Similarly, the repartition of graph G_3 resulted in experiment *Second*, and so on. For the first application domain below x is 10. Therefore, there are $x - 1$, or nine, repartitioning experiments. For the second domain x is 7, so there are six repartitioning experiments.

7.1 Laplace's Equation Solver

Table 2 shows the performance of our single and multilevel directed diffusion repartitioning algorithm on the first application domain. Here, the edge-cuts and run times are averaged over the nine experiments. Also, the TOTALV results are first divided by the total number of vertices in each graph and then averaged together to obtain TOTALV. SLDD indicates results obtained from our single-level directed diffusion repartitioning algorithm with CF and SF set at zero. MLDD indicates results obtained from the MLDD algorithm. MLDD-CS indicates results obtained from our multilevel directed diffusion repartitioning algorithm with CF set to 0.0001 and SF set to 0.25. MLDD-CdS indicates results from the MLDD-CS algorithm with dynamic suppression and CF set to 0.0001. Note, we have chosen CF to be 0.0001 because the graph imbalance is very slight here. Thus, the initial low edge-cut partition is not perturbed significantly. This allows us to concentrate the power of the multilevel paradigm on minimizing TOTALV instead of decreasing the

Algorithm	Edge-Cut	TOTALV %
SLDD	2,677	5.10
MLDD	2,463	3.16
MLDD-CS	2,553	2.38
MLDD-CdS	2,673	1.39
METIS	2,485	97.3

Table 2: Performance of different diffusion schemes.

edge-cut. METIS indicates results from partitioning from scratch with METIS. We use nine graphs with sizes from 31,624 vertices and 46,986 edges to 281,706 vertices and 421,172 edges. All of the results are obtained using a 64-way partition.

The results show that the multilevel scheme provides better edge-cut and TOTALV compared with the single-level scheme. Also, the dynamic suppression is quite effective, as it marginally increases the edge-cut, while substantially decreasing TOTALV.

7.2 Helicopter Blade Application

Figure 8 shows the results obtained from repartitioning a series of rapidly changing adapted meshes described above on a 64-way partition. The results are normalized against those in which METIS was used to partition the imbalanced graph from scratch. MLDD-dS indicates the results from the multilevel undirected diffusion repartitioning algorithm using dynamic suppression. SLDD-dS indicates the results from the single-level directed diffusion repartitioner with dynamic suppression.

We used the multilevel undirected diffusion algorithm because the multilevel directed diffusion algorithm was unable to balance the graph. This was because the weights of the vertices were highly heterogeneous. That is, they differed from each other by up to a factor of 1,000. We found that the vertices were often too coarse to be able to be guided by the diffusion solution. For example, quite often during the execution of the algorithm, the diffusion solution called for vertices totaling a weight of 500 to be migrated, but all of the boundary vertices had much larger weights (*i.e.*, greater than 700), as all of the lower weight vertices had already been transferred. For some cases, even on the finest graphs, it was not possible to achieve balance using the diffusion solution. Undirected diffusion allows for more degrees of freedom. That is, extremely high weight vertices are more free to migrate about, eventually reaching underbalanced partitions via circuitous routes.

The results confirm the robustness of multilevel diffusion. We see that the multilevel undirected diffusion repartitioner substantially outperforms the single-level scheme. It obtains lower edge-cut and TOTALV results across the board than the single-level scheme. It also obtains MAXV results lower than SLDD-dS in four out of six cases.

With respect to partitioning from scratch, the multilevel scheme again reduces both TOTALV and MAXV while in-

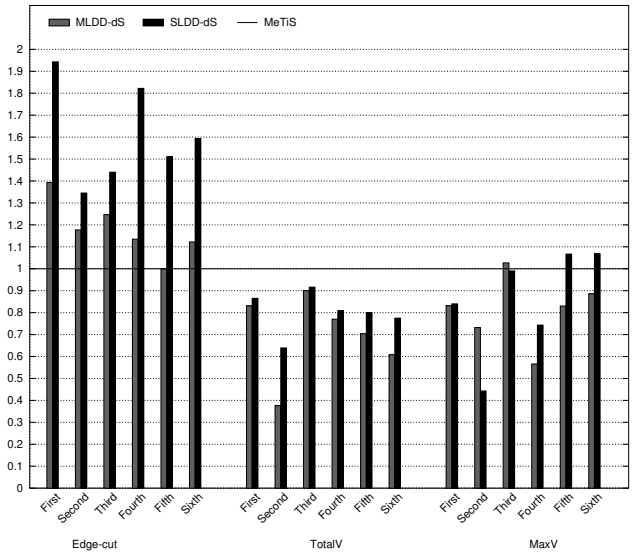


Figure 8: Repartitioning of Helicopter Blade Application Graphs

creasing edge-cut. Here, however, the TOTALV and MAXV of MLD are only slightly improved over that of partitioning from scratch. In fact, for some graphs MAXV of partitioning from scratch actually beats that of repartitioning. This is due to the complexity of the imbalance problem, which necessitates migration of a large number of vertices in order to balance the graph. For some of these graphs, so much vertex migration is necessary that repartitioning brings little benefit here over partitioning from scratch.

8 Parallel Formulation of Multilevel Diffusion

We have implemented the multilevel diffusion algorithms in PARMETIS, which contains a parallel formulation of our multilevel k-way graph partitioning algorithm. Vertices are initially assumed to be distributed across p processors. This division of vertices corresponds to the original partitioning. The coarsening phase of parallel multilevel diffusion algorithms requires no communication, as the computed matchings are restricted to vertices residing on the same processors. Otherwise, this phase is identical as described in [13].

The parallel formulation of the multilevel diffusion phase depends on whether we are using directed or undirected diffusion. The parallel formulation of the undirected diffusion algorithm is modeled after our coarse-grained parallel multilevel refinement algorithm [13]. The only difference is that vertices are visited and selected for migration according to the criteria for undirected diffusion described in Section 4.1. In the case of directed diffusion, our current implementation performs directed diffusion only for the coarsest graph. Since the coarsest graph is very small (its size is proportional to the number of processors), this is done serially, as the computation does not significantly affect the overall performance and scalability of our parallel multilevel directed diffusion algorithm. Furthermore, we use the additional processors to obtain a better directed diffusion as follows. The graph is broadcast to all processors. Each pro-

cessor then simultaneously balances the coarsest graph using the directed diffusion algorithm described in Section 4.1 with dynamic suppression (Section 6.3). Since the diffusion scheme is inherently random, each processor computes a potentially unique partition. The best partition can then be selected by focusing on the edge-cut, the TOTALV, the MAXV, or the balance. In our experiments, we select the partition that has the lowest edge-cut. If the coarsest graph is too coarse to allow for complete balancing, then the parallel undirected diffusion algorithm is used to balance any remaining imbalances.

The parallel formulation of the multilevel refinement algorithm is similar to that for undirected diffusion with the exception that vertices are moved according to the *refinement phase vertex migration criteria* described in Section 4.1. Furthermore, the concepts of vertex cleanness and suppressions are also employed to reduce TOTALV and MAXV. Otherwise, our multilevel refinement algorithm is identical to the coarse-grained parallel multilevel refinement algorithm described in [13].

8.1 Experimental Results

We tested our parallel multilevel repartitioning algorithms on a Cray T3D with 256 processors. Each processor on the T3D is a 150Mhz Dec Alpha (EV4). The processors are interconnected via a three dimensional torus network that has a peak unidirectional bandwidth of 150Bytes per second, and a small latency. We used Cray's MPI library for communication. Cray's MPI achieves a peak bandwidth of 45MBytes and an effective startup time of 57 microseconds.

We evaluated the performance of our parallel adaptive multilevel diffusion algorithms on four medium to large size graphs arising in finite element computations. These graphs are duals of 3D finite element meshes with tetrahedra elements. We have selected these larger graphs to demonstrate that our schemes can perform adaptive repartitioning of very large graphs quickly. The characteristics of these graphs are described in Table 3.

For each graph we synthetically generated adaptive graphs by randomly changing the weights of the vertices. For example, on 64 processors, we first computed a 64-way partition and moved the graph so that processor P_i stores the vertices that belong to partition i. Next, each processor P_i selects a random number r_i between zero and the number of vertices that it stores, and randomly selects r_i of its vertices and changes their weight from one to three. We found that this scheme leads to graphs that are about 50% load imbalanced, *i.e.*, there are partitions whose weight are 50% higher than the average weight. These synthetically generated graphs are then used as the input to our parallel multilevel diffusion and partitioning algorithms. In all of our experiments, a graph that is 5% load imbalance is assumed to be well balanced, and in all experiments this load imbalance level is easily achieved.

Table 4 shows the results obtained by our parallel multilevel directed and undirected diffusion algorithms for the test problems on 64, 128, and 256 processors. For each problem, this table shows the edge-cut of the resulting partitioning, the total number of vertices that needs to be moved (TOTALV), the maximum number of vertices that needs to be moved

Graph Name	Number of Vertices	Number of Edges
MRNGA	257000	505048
MRNGB	1017253	2015714
MRNGC	4039160	8016848
MRNGD	7833224	15291280

Table 3: The various graphs used in the experiments.

in and out of any particular processor (MAXV), as well as the amount of time (in seconds) required to compute the repartitioning. The rows labeled 'Scratch' show similar performance metrics for the scheme in which the adaptive graph is being repartitioned from scratch using the parallel multilevel k-way partitioning algorithm described in [13]. Finally, the rows labeled 'Imbalance' indicate the load imbalance generated by our synthetic adaptation scheme for each one of the graph-processor combinations. As the reader can see, the load imbalance ranges between 1.42 (42%) and 1.52 (52%). Note that MRNGD could not run on 64 processors due to the limited amount of memory on each processor.

Graph	Scheme	64 Processors				128 Processors				256 Processors			
		Cut	TotalV	MaxV	Time	Cut	TotalV	MaxV	Time	Cut	TotalV	MaxV	Time
MRNGA	Imbalance	1.4244				1.4391				1.4971			
	Undirected	26457	24819	2427	0.573	32898	24089	1391	0.424	42536	25423	730	0.436
	Directed	25895	25858	2363	0.666	34635	26670	1430	0.547	42502	25620	755	1.066
	Scratch	25099	253856	9815	0.902	32855	252054	4991	1.019	43130	256020	2660	2.247
MRNGB	Imbalance	1.4876				1.4955				1.4854			
	Undirected	69428	92806	10625	1.377	87687	94704	5342	1.017	112522	94328	2821	0.817
	Directed	67032	94397	10447	1.534	86836	90785	5376	1.174	112199	95074	3028	1.161
	Scratch	65395	996825	39444	2.039	85812	1016832	20365	1.799	110459	1016166	10507	3.294
MRNGC	Imbalance	1.4715				1.4920				1.4995			
	Undirected	169336	342078	46182	3.994	217165	358453	26919	2.435	278607	368216	11513	1.689
	Directed	169559	386720	44182	4.385	223031	382812	27300	2.685	279832	351109	11299	2.092
	Scratch	167617	3943621	166742	5.056	222368	3998118	83983	3.558	281195	3984525	42061	4.352
MRNGD	Imbalance					1.4470				1.5152			
	Undirected					345122	640172	40995	4.145	452386	743892	22439	2.747
	Directed					350334	700727	46231	4.511	445263	768296	22902	3.203
	Scratch					343354	7487361	156524	5.511	438704	7507463	83205	4.692

Table 4: The performance of the parallel adaptive repartitioning schemes on the four test graphs for 64, 128, and 256 processors on Cray T3D. All run-times are in seconds.

To better compare the three schemes, we graphically depicted the results shown in Table 4 in the sequence of graphs shown in Figure 9. In particular, Figure 9(a) compares the quality in terms of edge-cut produced by the two multilevel diffusion algorithms relative to partitioning from scratch. For each experiment, we computed the ratio of the edge-cut produced by the diffusion algorithms to that of partitioning from scratch and plotted it using a bar chart. For example, the bars for MRNGB-128 corresponds to partitioning the adaptively refined graph MRNGB on 128 processors. Looking at this figure we can see that the edge-cuts of the partitionings produced by the two multilevel diffusion algorithms are in general within 5% of the edge-cut produced by partitioning the adaptive graphs from scratch. Furthermore, for some of these graphs (*e.g.*, MRNGA-256 and MRNGC-128) the multilevel diffusion algorithms produced partitionings whose edge-cut was slightly better than that of partitioning from scratch. These results show that our adaptive

multilevel diffusion algorithms are able to produce partitionings whose quality is comparable to the quality obtained by the parallel multilevel partitioning algorithm. Comparing the two multilevel diffusion algorithms together, we see that in general they lead to partitionings that have comparable edge-cuts.

The performance in terms on the number of vertices that need to be moved (TOTALV and MAXV) is shown in Figures 9(b) and 9(c), respectively. Looking at these two figures we can clearly see the advantage of the multilevel diffusion algorithms over partitioning from scratch in significantly reducing the number of vertices that needs to be moved in order to achieve load balance. The multilevel diffusion schemes move fewer than 10% percent of the total number of vertices that are required to be moved by partitioning from scratch (Figure 9(b)), and the maximum number of vertices that is sent and received by any processor is fewer than 30% of that required by partitioning from scratch (Figure 9(c)). As was the case with the edge-cut, the relative performance of directed and undirected diffusion is quite similar, with both schemes performing within a couple percentage points of each other.

Notice that our serial algorithms' results from Section 5 indicate that directed diffusion is generally more effective than undirected diffusion in keeping edge-cut and vertex movement low. However, our parallel formulations of directed and undirected diffusion have similar performance. This is because our current parallel implementation of multilevel directed diffusion is a simplified version of the serial algorithm. Specifically, as described in Section 8, directed diffusion occurs only on the coarsest graph in the parallel formulation. At this point if the graph is not balanced, undirected diffusion takes over. Hence, the two schemes obtain results that are quite similar.

Finally, Figure 9(d) shows the amount of time required by the two multilevel diffusion algorithms relative to partitioning from scratch. From this figure we can see that the multilevel diffusion algorithms are considerably faster than partitioning the graph from scratch. In particular, for MRNGD on 256 processors, undirected diffusion is about 40% faster than partitioning from scratch, requiring only 2.75 seconds (partitioning from scratch requires 4.69 seconds). This run-time difference is mostly due to the fact that the multilevel diffusion algorithms do not have to compute an initial k-way partitioning of the coarsest graph, as they inherit the original partitioning. Comparing the multilevel undirected diffusion with the directed diffusion algorithm we see that the former is faster, and the performance gap increases with the number of processors. This is because the directed diffusion algorithm needs to diffuse the coarsest graph serially, whereas the undirected diffusion algorithm does not have this serial component. Looking at the run-times shown in Table 4 we can see that both multilevel diffusion algorithms scale very well with the number of processors, and they are able to partition graphs with around eight million vertices in around three seconds.

9 Conclusions

Our results on a variety of synthetic and application meshes show that multilevel diffusion is a robust scheme for repartitioning a wide variety of adaptive meshes. The resulting edge-cuts are generally close to those resulting from partition-

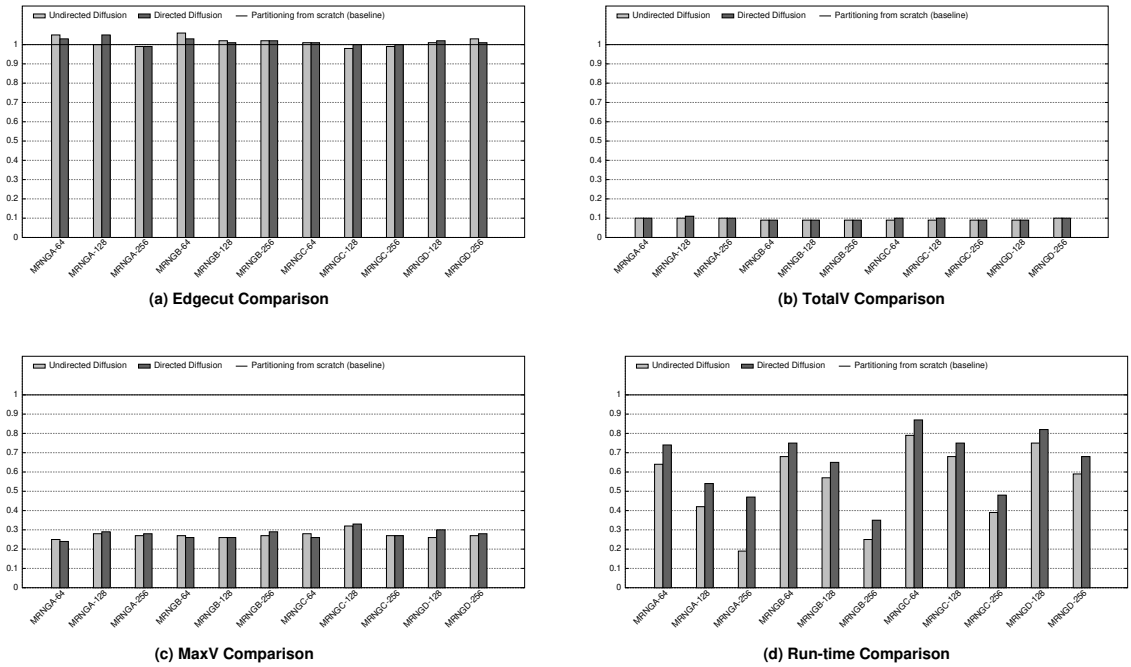


Figure 9: Comparison of the parallel formulations of directed and undirected relative to partitioning from scratch. Diagram (a) compares the quality in terms of edge-cuts. Diagram (b) compares the quality in terms of the total number of vertices that need to be moved (TOTALV). Diagram (c) compares the quality in terms of the maximum number of vertices that need to be moved in and out of a partition (MAXV). Diagram (d) compares the run time of the various algorithms. In each diagram, the results are normalized with respect to partitioning from scratch. Bars under the baseline indicate that the multilevel diffusion algorithms perform better than partitioning from scratch.

ing from scratch, while vertex movement is quite reduced. Furthermore, parameterized heuristics allow for edge-cut, TOTALV, or MAXV to be specifically optimized depending on application requirements. Multilevel diffusion also produces significantly better edge-cuts compared with single-level directed diffusion. Our experiments show that directed diffusion tends to obtain results improved over those obtained by undirected diffusion.

Throughout this paper, we have reported results from various values of parameters CF and SF. Our experiments have shown that the use of either one will reduce vertex movement, while increasing edge-cut slightly. The higher these parameters are set at, the greater these trade-offs become. When they are used in conjunction with one another this effect is increased. Thus, for a nice compromise of vertex movement and edge-cut, employing only one parameter or the other will usually be sufficient. In general, the appropriate values of these parameters will be different for different applications. We provide the following general guidelines for specific scenarios. If edge-cut of the resulting partition is most important, then set both CF and SF to zero. This will allow for the maximum movement of vertices to minimize edge-cut. If vertex densities are homogeneous, set SF to zero, as suppression is ineffective. If vertex densities are highly heterogeneous, then prefer high values of SF (over CF) to strike a trade-off with edge-cut. Finally, if vertex migration time is most important, then use a high value for both CF and SF in order to minimize vertex movement.

Our results also show that for highly complex balancing problems, the benefits obtained from repartitioning over

partitioning from scratch are reduced. The helicopter blade experiments illustrated that MAXV results obtained from multilevel diffusion might degrade to the point in which they approach or even surpass those of partitioning from scratch for highly imbalanced graphs. In a bandwidth-rich system, MAXV will tend to determine vertex movement time. Therefore, for certain application domains, it may well be beneficial to partition from scratch in order to maintain low edge-cut while not giving up much in terms of MAXV. Thus, for such applications, repartitioning may seldom be preferable to partitioning from scratch.

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

References

[1] R. Biswas and R. C. Strawn. A new procedure for dynamic adaption of three-dimensional unstructured grids. *Applied Numerical Mathematics*, 13:437–452, 1994.

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

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

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

[5] Gene H. Golub and Charles Van Loan. *Matrix Computations: Second Edition*. The Johns Hopkins University Press, Baltimore, MD, 1989.

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

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

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

[9] G. Karypis and V. Kumar. METIS: Unstructured graph partitioning and sparse matrix ordering system. Technical report, Department of Computer Science, University of Minnesota, 1995. Available on the WWW at URL *http://www.cs.umn.edu/~karypis/metis*.

[10] G. Karypis and V. Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. Technical Report TR 96-036, Department of Computer Science, University of Minnesota, 1996. Also available on WWW at URL http://www.cs.umn.edu/~karypis. A short version appears in Supercomputing 96.

[11] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, Accepted for publication, 1997. Also available on WWW at URL http://www.cs.umn.edu/~karypis.

[12] G. Karypis and V. Kumar. A fast and highly quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, to appear. Also available on WWW at URL http://www.cs.umn.edu/~karypis. A short version appears in Intl. Conf. on Parallel Processing 1995.

[13] George Karypis and Vipin Kumar. A coarse-grain parallel multilevel k-way partitioning algorithm. In *Proceedings of the eighth SIAM conference on Parallel Processing for Scientific Computing*, 1997.

[14] Leonid Oliker and Rupak Biswas. Efficient load balancing and data remapping for adaptive grid calculations. Technical report, NASA Ames Research Center, Moffett Field, CA, 1997.

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

[16] Kirk Schloegel, George Karypis, and Vipin Kumar. Repartitioning of adaptive meshes: Experiments with multilevel diffusion. *Third International Euro-Par Conference Proceedings*, pages 945–949, August 1997.

[17] Sohn, Biswas, and 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. Dynamic load-balancing for parallel adaptive unstructured meshes. *Parallel Processing for Scientific Computing*, 1997.

[19] C. Walshaw, M. Cross, and M. G. Everett. Dynamic mesh partitioning: A unified optimisation and load-balancing algorithm. Technical Report 95/IM/06, Centre for Numerical Modelling and Process Analysis, University of Greenwich, London, UK, December 1995.

[20] R. D. Williams. Dime: Distributed irregular mesh environment. Technical Report C3P 861, Caltech Concurrent Computation Report, 1990.

[21] 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.