Multilevel Diffusion Schemes for Repartitioning of Adaptive Meshes *

Kirk Schloegel, George Karypis, Vipin Kumar
University of Minnesota, Department of Computer Science
Technical Report: 97-013

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

June 19, 1997

Abstract

For a large class of irregular grid applications, the structure of the mesh changes from one phase of the computation to the next. Eventually, as the graph evolves, the adapted mesh has to be repartitioned to ensure good load balance. If this new graph is partitioned from scratch, it will lead to an excessive migration of data among processors. In this paper, we present two new schemes for computing repartitionings of adaptively refined meshes. These schemes perform diffusion of vertices in a multilevel framework and 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, is potentially much faster than single-level diffusion in a parallel context, 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 amount of computation associated with a grid point is represented by the weight of its associated vertex. The amount of interaction required between two grid points is represented by the weight of the edge between the associated vertices. Efficient parallel execution of these irregular grid applications requires the partitioning of the associated graph into p parts with the following two constraints: (i) Each partition has an equal amount of total vertex weight; (ii) The total number of 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 work [6, 10].

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 [1], areas of the original graph are selectively coarsened or refined in order to accurately model the dynamic computation. This causes the weights of the vertices and the edges to change. Eventually, as the graph evolves, it becomes necessary to correct the partition in accordance with the structural changes in the computation and to migrate a certain amount of computation between processors. Thus, we need a partitioning or repartitioning algorithm to redistribute the adapted graph. This algorithm should satisfy the following constraints.

*This work was supported by NSF CCR-9423082, by Army Research Office contract DA/DAAH04-95-1-0538, by Army High Performance Computing Research Center cooperative agreement number DAAH04-95-2-0003/contract number DAAH04-95-C-0008, 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

1. **It robustly balances the graph.** Failure to balance the graph will lead to load imbalance, which will result 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 grid computation can begin, data associated with the migrated vertices also needs 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.** The computational cost of repartitioning should be inexpensive since it is done frequently. Also, since the problem studied in this paper is relevant only in the parallel context, the repartitioning algorithm should be parallelizable. Performing the repartitioning on a serial processor can become a very serious bottleneck.

If the adapted graph is partitioned from scratch using a state-of-the-art multilevel graph partitioner such as METIS [11], then it will reasonably optimize criterion 1 and 2. Since a highly parallel formulation of METIS exist [11], 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 [16]) 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, 16]. For these graphs, such a scheme inherits a good (i.e. low edge-cut), but imbalanced starting point in the initial partition. It then attempts to fix the imbalance of this partition while maintaining its good edge-cut. It does this by minimizing the amount of disturbance (i.e. vertex migration) to the initial partition in the balancing phase. 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, 16] can only provide a limited improvement in the edge-cut of the resulting partition.

One promising solution to the problem of edge-cut degradation as the degree of imbalance increases in size and complexity is the use of a multilevel scheme that takes the initial location of the vertices to 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. In addition to the refinement, the movement of graph vertices (to achieve load balance) can also be done at multiple coarsened versions. This multilevel diffusion scheme can move large chunks of vertices at coarser levels, and then achieve better load balance at finer levels. In a multilevel context, a global picture of the graph [8] can be used to guide graph balancing while utilizing a multilevel view to guide refinement.

In this paper, we describe two multilevel diffusion repartitioning algorithms. The first constructs a series of contracted graphs by collapsing pairs of vertices together. Once a sufficiently small graph has been constructed, undirected diffusion is employed to balance the graph. Finally, the graph undergoes multilevel refinement in an attempt to clean up the edge-cut disturbed by the balancing phase. The second multilevel diffusion repartitioning algorithm is similar to the first. Here, however, the partition is balanced by means of directed diffusion [8]. The graph contraction and multilevel refinement phases are otherwise identical to the first algorithm. 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 directed diffusion tends to obtain better results than those obtained by undirected diffusion. Multilevel diffusion can be easily parallelized analogous to multilevel graph partitioning [11] and is potentially much faster than single-level diffusion in a parallel context.

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. Finally, Section 8 states our conclusions.

2 Notations, Definitions, and Issues

In our discussion we include the concepts of both vertex weight and vertex size. Vertex weight is the computational cost of the work represented by the vertex while size is 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 [14]. A method of reducing the vertex migration overhead used in [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 and P be a set of p processors. 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

$$\exists q \mid W(q) > \overline{W} \times k$$

where k is a small constant. If k were to equal 1, 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 k equal to 1.03.

In an imbalanced graph, partitions, whose weights are greater than the average partition weight times k are *overbalanced.* Likewise, those partitions whose weights are less than the average partition weight divided by k are *underbalanced.* 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. 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.

3 Repartitioning Strategies: Review of Previous Work

A repartitioning of a dynamic graph can be computed by simply partitioning the new graph from scratch. For example, a state-of-the-art multilevel partitioner such as METIS [9] can provide a fast, scalable and balanced partition with a low edge-cut. However, intuition tells us that since no concern is given for the existing partition, most vertices are not likely to be assigned to their initial partitions with this method. Thus, vertex migration will be unduly high. The advantage of this strategy is that while vertex migration time is sacrificed, edge-cut is minimized. In fact, this strategy generally resulted in partitions with the lowest edge-cuts of any of the algorithms which we tested. However, because vertex migration overhead is very large with this method, simply partitioning the modified graph from scratch is unacceptable for many applications.

The second strategy is to use the existing partition as input for a repartitioning algorithm and to attempt to minimize the difference between the original partition and the output partition. This strategy has the potential benefit of reducing TotalV by an order of magnitude or more over partitioning 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 repartioning* 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 which 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 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. 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. Edge-cut is 12 here. This is as good as the original partition. However, TotalV is 13. This is because most vertices migrated, not in order to balance the graph, but simply 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 17. Notice that some of the vertices in partition 1 are now disconnected from the rest. This is a result of cut-and-paste repartitioning and explains the edge-cut degradation. 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 again from scratch minimizes edge-cut while resulting in extremely high TotalV results. Diffusion, however, attempts to keep TotalV low by ensuring that the vertices which do not need to be migrated in order to balance the graph are reassigned to their original partitions. It also attempts to keep edge-cut low by maintaining partition connectivity.

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

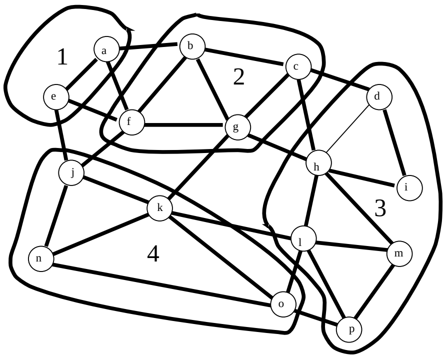

Figure 1(a): Original Graph

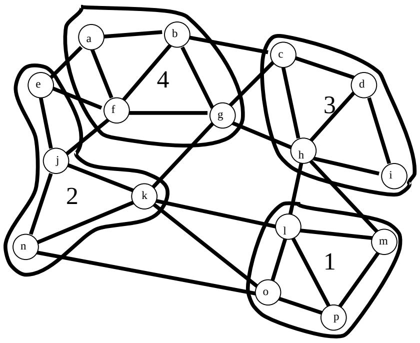

Figure 1(b): Partitioning from Scratch

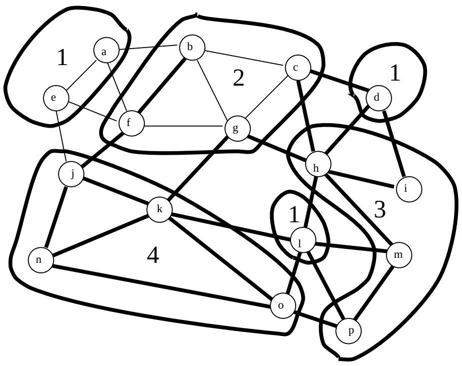

Figure 1(c): Cut-and-Paste Repartitioning

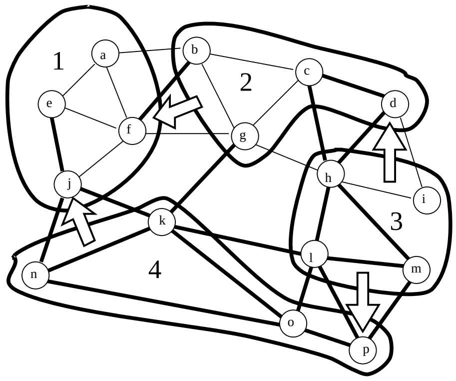

Figure 1(d): Diffusion Repartitioning

Figure 1: Partitioning and Repartitioning Examples

in partition weights between each partition and all of its neighbor partitions. Undirected diffusion has the advantage that it is highly distributed in nature. However, balancing occurs without the guidance of a global view of the graph. This can increase the edge-cut, vertex migration costs, and run time of the algorithm. The general context of load balancing has been studied in [2, 3, 19].

Directed diffusion is diffusion guided by a global view of the graph. It is accomplished by obtaining a *diffusion solution* and applying it to the vertices of the imbalanced graph. The diffusion solution, λ, 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 [7, 8, 17].

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 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 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. Another disadvantage is that, the communication channels are not used efficiently. Many are idle, while a few are overworked. This

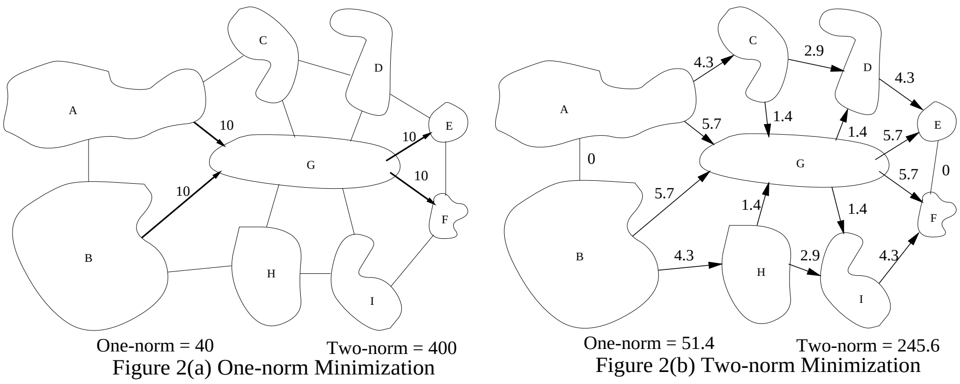

Figure 2(a) One-norm Minimization
Figure 2(b) Two-norm Minimization

Figure 2: One- and Two-Norm Diffusion Examples

can potentially lead to contention when the vertices are actually moved.

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 and `MaxV` is approximately minimized. The trade-off is that `TotalV` will be higher in two-norm minimization.

Ou and Ranka developed a method which optimally minimizes the one-norm of the diffusion solution using linear programming. They used this solution in a repartitioning algorithm, called the Incremental Graph Partitioner (IGP), which calculates the solution vector, moves the necessary vertex weight, and then refines the balanced graph in a order to reduce the edge-cut upset by the shifting vertices. They used this algorithm to repartition a set of graphs and compared the results to the original partitioning which was done by Recursive Spectral Bisection. Their results indicate that the repartitioned edge-cut was comparable to the original edge-cut and that the IGP run time was dramatically lower than the run time of the original partition [15].

Hu and Blake 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 λ is the diffusion solution, b is the vector containing the load of each partition minus the average partition load, and L is a Laplacian matrix, defined as

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

Furthermore, they showed that when using the parallel conjugate gradient algorithm [5] to solve for λ, the algorithm converges in less than p iterations [8].

Walshaw, Cross, and Everett implemented, JOSTLE, a combined partitioner and directed diffusion repartitioner based on an optimization of the Hu and Blake diffusion solver. The JOSTLE algorithm has two distinct phases. The first is a balancing phase in which the diffusion solution guides vertex migration in order to balance the graph. The second is a refinement phase in which a local view of the graph guides vertex migration in order to decrease the edge-cut upset by the balancing phase [17].

Walshaw, Cross, and Everett developed JOSTLE-MD by changing the refinement phase of the original JOSTLE repartitioner to a multilevel refinement phase [16]. A summary of their results is included in Section 7.1. Unlike our algorithm, JOSTLE-MD employs a single-level diffusion scheme for balancing and then performs multilevel refinement. It also does not include the concepts of vertex weight, size, or of `MaxV`. Neither is it able to specifically minimize edge-cut, `TotalV`, or `MaxV`.

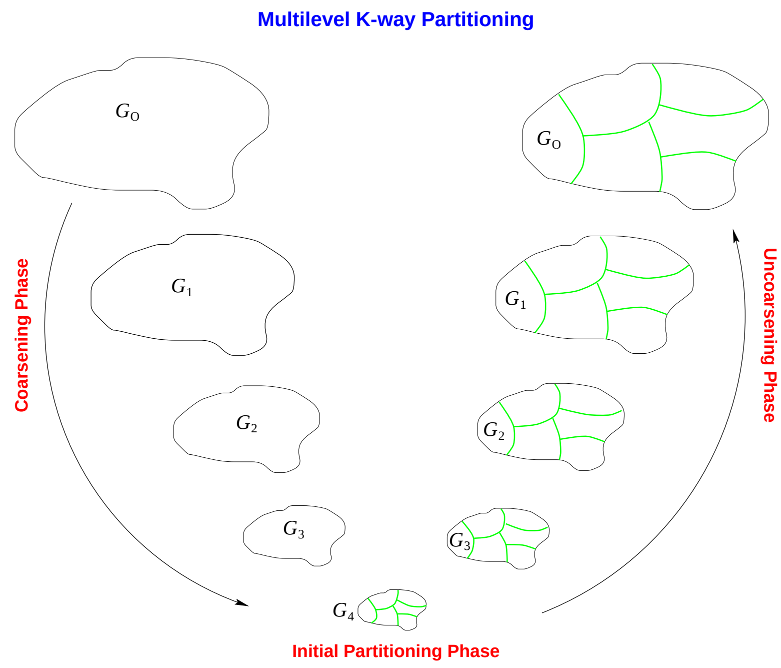

Figure 3: Multilevel K-way Partitioning

4 Multilevel Graph Repartitioning

Multilevel graph repartitioning is essentially a modification of the k-way multilevel partitioning algorithm [11]. Hence, we first review the k-way multilevel scheme for partitioning.

4.1 A Review of Multilevel Schemes for Graph Partitioning

The k-way multilevel graph partitioning algorithm [10] implemented in MeTiS [1] has three phases, a coarsening phase, a partitioning phase, and a refinement (or uncoarsening) phase. 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 using one from among a variety of methods. Finally, the partition of the coarsest graph is projected back to the original graph by refining it at each uncoarsening level. Since now every uncoarsening level consists of a finer graph, each subsequent graph has more degrees of freedom than the previous one had. These degrees of freedom can be used to decrease the edge-cut at each level. Figure 3 illustrates this paradigm. In essence, using this multilevel approach accomplishes three things. First, it speeds up the computation of an initial partition since this is computed on a small graph. Second, this initial partition is quite good if the coarsening is done intelligently. Third, it allows multilevel refinement which improves the quality of the initial partition. Thus, the MeTiS algorithm uses a global view of a graph to quickly find a good initial partition and multilevel views of the graph to further improve this partition.

Refinement is done in MeTiS by a method based on the Kernighan-Lin refinement algorithm [4, 13]. Vertices are visited randomly. Each border vertex visited is checked to see if the migration of the vertex to another partition will

1. decrease the edge-cut while maintaining the graph balance, or

[1]Throughout this paper, we will refer to the k-way multilevel graph partitioning algorithm implemented in MeTiS as simply MeTiS.

2. maintain the edge-cut and improve graph balance.

If so, the vertex is migrated. This process is repeated until it converges [10]. We define these two conditions as the *vertex migration criteria*.

4.2 Multilevel Diffusion Repartitioning Algorithms

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. Since this is done on the coarsest graph, the number of vertices is small. 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 the gain is negative. 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 fine 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 finer vertices. The process described above is then begun on the next coarsest graph. Our experiments have shown that the graph will typically balance on one of the first three coarsest graphs.

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 another partition will

1. maintain the edge-cut, maintain the balance, and the selected partition is the vertex's initial partition from the input graph, or

2. decrease the edge-cut while maintaining the graph balance, or

3. maintain the edge-cut and improve graph balance.

If so, the vertex is migrated. These three conditions make up the *refinement phase vertex migration criteria*. 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`.

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 diffusion solution) guiding vertex migration. That is, the border vertices are visited randomly. If a 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 produce the highest gain. The vertex is migrated even if this gain is negative. 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.

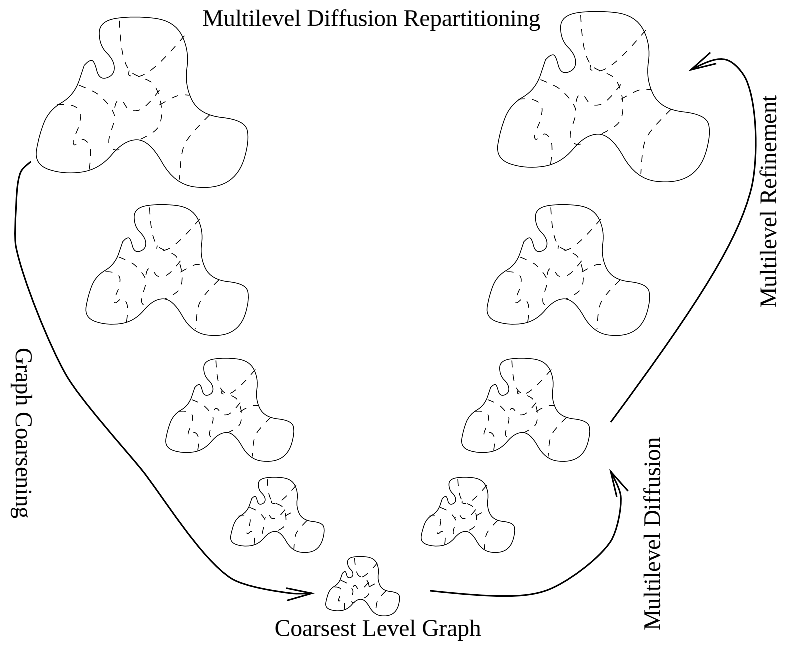

Figure 4: Multilevel Diffusion Repartitioning

In summary, as illustrated in Figure 4, 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 will be used to provide a comparison with our multilevel diffusion schemes. We will refer to a version of our multilevel directed diffusion algorithm in which the graph is not initially contracted as single-level directed diffusion or SLDD.

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 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 in such a way as to produce an average vertex weight of 18 in source partitions, 2 in sink partitions, and 10 in all other partitions.

Table 1: Input Graphs			
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

Figure 5 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 5(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 not unexpected, 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 5(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 5(b), (c), and (d) illustrate this point. Figure 5(b) shows the results of repartitioning on graphs with four source and four sink partitions. Here, the weight of every vertex in each partition was multiplied by a constant. Figure 5(c) shows the results of repartitioning on graphs with four source and four sink partitions. Here, however, the weight of every vertex in each partition was multiplied by a randomly generated number. Figure 5(d) shows the results of repartitioning on graphs with eight source and eight sink partitions and randomly distributed vertex weights. These results show that our multilevel directed diffusion algorithm is effective in keeping the edge-cut degradation and `TotalV` down for arbitrarily 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 very 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. Thus, the multilevel paradigm is so powerful that it can produce good results with even an undirected diffusion algorithm.

6 Heuristics

We showed from the previous results that our multilevel directed diffusion repartitioner can repartition imbalanced graphs resulting in edge-cuts which are lower than those obtained with the single-level diffusion algorithm and with far less data movement than results obtained from partitioning from scratch.

We found that it is also possible to trade edge-cut in order to lower both `TotalV` and `MaxV`. In those applications in which vertex migration time dominates, `TotalV` and/or `MaxV` determines execution time. Here we see that the single-level diffusion algorithm performed comparably or better than the multilevel diffusion repartitioners. In order to improve these results we developed heuristics to lower `TotalV` and `MaxV`,

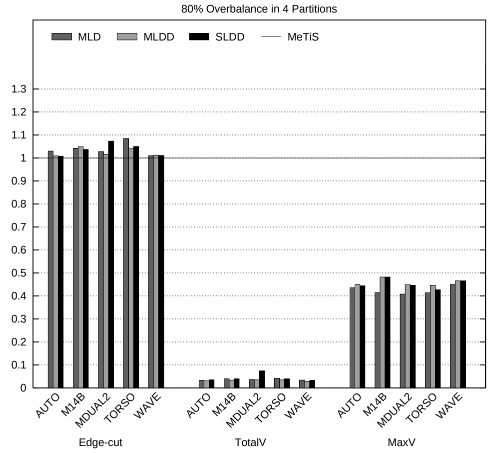

Figure 5(a)

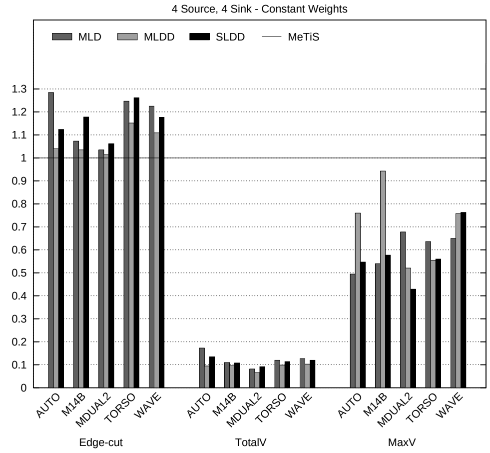

Figure 5(b)

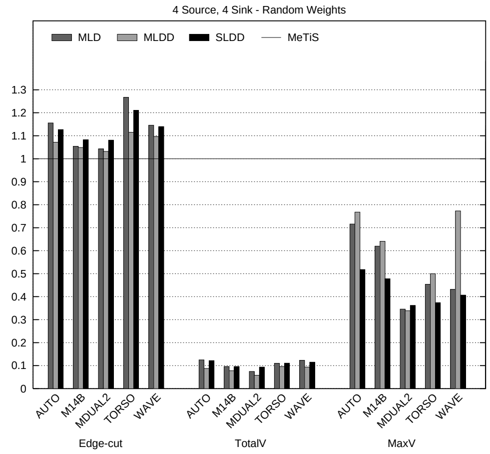

Figure 5(c)

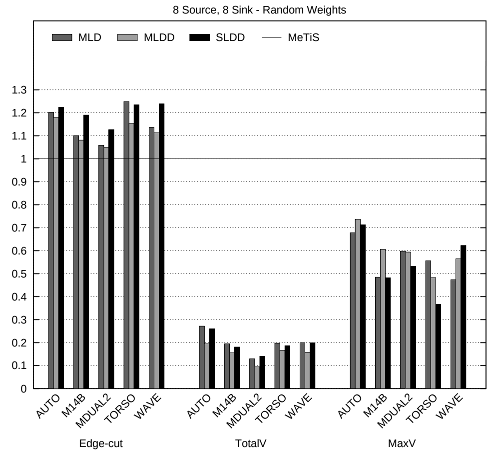

Figure 5(d)

Figure 5: Repartitioning Results

while sacrificing edge-cut only marginally.

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. In order to reduce this sum we developed a heuristic called the *cleanness factor*. 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 cleanness factor. During multilevel refinement, only those clean vertices whose gain resulting from migrating partitions is greater than their size times the cleanness factor are considered for migration. Thus, we limit the movement of larger clean vertices who result in only small edge-cut gains. If the cleanness factor is infinity, the result is that only dirty vertices are considered for migration during multilevel refinement. If it is zero, then all vertices, clean and dirty, are considered and may be migrated even if they do not reduce the edge-cut.

Figure 6 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. *Clean = .0001* indicates results from the multilevel diffusion repartitioner with a cleanness factor of .0001. Thus, a clean vertex is migrated only if it produces a positive gain. In other words, for Clean = .0001, item 3 of the refinement phase vertex migration criteria is not applicable. Likewise *Clean = .0* and *Clean = 999999* indicate results from multilevel diffusion with cleanness factors of zero and 999999, respectively. Note that *Clean = 999999* does not allow clean vertices to migrate. The results are normalized against those obtained with the cleanness factor of zero.

In each of the results, `TotalV` decreases as the cleanness factor 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, we have shown 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 the cleanness factor increases. This would be the result when `MaxV` is dominated by the sum of the sizes of the incoming vertices 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. The max of the sum of the sizes of vertices which migrate **out** of any one partition is lower bounded by the most overbalanced partition. That is, a certain weight of vertices must migrate out of this partition in order to obtain balance. It is, of course, upper bounded by the partition with the highest sum of vertex sizes. Our experiments have shown that the outgoing component of `MaxV` is not usually a concern. Intuitively, this is because vertices tend to migrate out of an overbalanced partition only until the partition is balanced. Furthermore, overbalanced partitions generally have an ample supply of average to highly dense vertices. That is, they tend to have a good supply of vertices whose weight divided by their size is relatively high. Choosing highly dense vertices whenever possible balances the graph while keeping the cost of vertex migration down. Simply by selecting vertices 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 problematical. Overbalanced partitions tend to be full of average to highly dense vertices, and so it is relatively easy to select a good (i.e. dense) vertex for migration. However, 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 7 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 will initially be full of relatively low-density vertices since it is also an underbalanced partition. Since these vertices are of low density, it will take a much greater number of them to balance partition C than it would have taken average or highly dense vertices. If this happens, partition

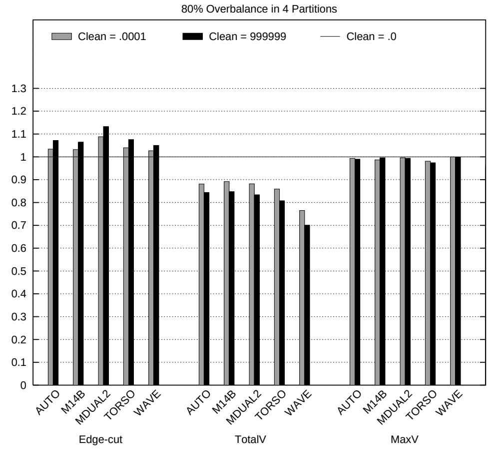

Figure 6(a)

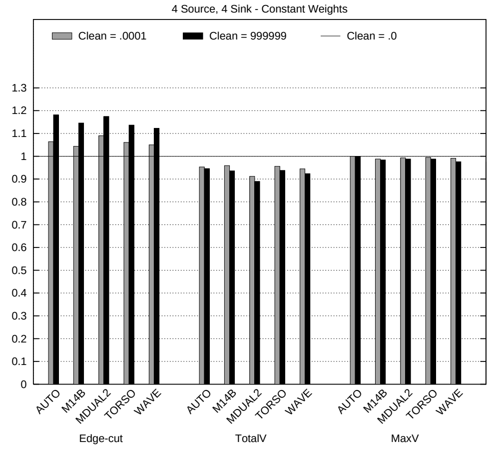

Figure 6(b)

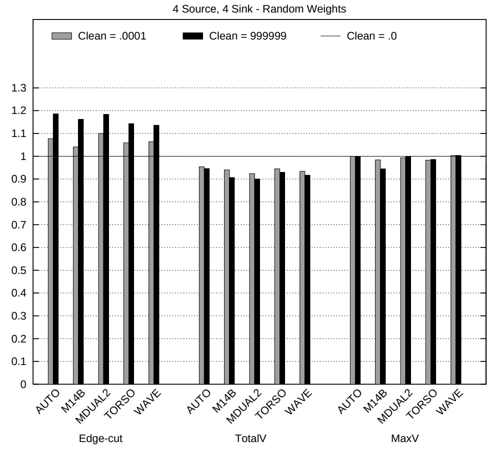

Figure 6(c)

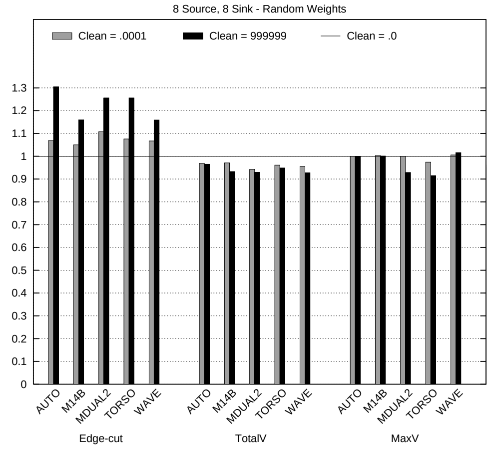

Figure 6(d)

Figure 6: Repartitioning with Cleanness

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 developed a heuristic called the *suppression factor*.

During multilevel diffusion only those vertices whose densities are greater than the average density of all of the vertices on the graph multiplied by the suppression factor are considered for migration. Therefore, if the suppression factor is zero, no vertex migration is *suppressed*. If the suppression factor is 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 the suppression factor is one, only vertices which are above the average density are allowed to migrate during balancing. The cost here is that the higher the amount of vertex suppression, the less free vertices are to migrate, and so the more difficult it is to balance the graph. With higher suppression factors, the graph will tend to balance at higher and higher uncoarsening

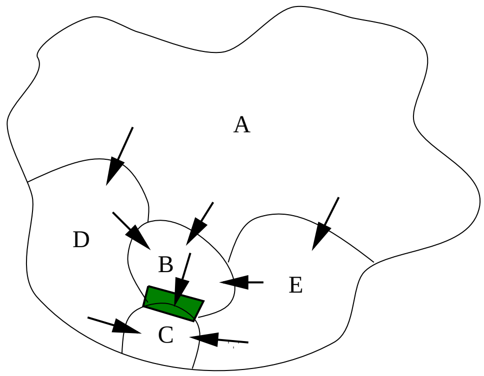

Figure 7: Blocking a Sink to Sink Transfer

levels. This is harmful to 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 less levels are free for refinement of the edge-cut.

Figure 8 shows the results of repartitioning the imbalance problems (described above) using a range of values for the suppression factor. We have set the cleanness factor at a constant .0001 for each of these experiments. Here, the results are normalized against the results obtained from using a suppression factor of zero (i.e. no suppression of vertices during balancing). We see that as the balancing problem becomes more complex, using even small positive values for the suppression factor reduces `MaxV` by up to 55%. Meanwhile, across the board, edge-cut is increased by only a few percent. As the suppression factor increases, `MaxV` tends to decrease, while edge-cut decreases. 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` tends to drop.

Notice in Figure 8(a) that suppression has had no effect. This is because in this imbalance problem vertex density is highly homogeneous. 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.

Figures 8(b) through (d) show that as the homogeneity of vertex density decreases, vertex suppression becomes more effective. However, by reexamining the results from Figure 5, we also see that as homogeneity decreases, `MaxV` becomes more problematical 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 problematical, 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 The MLDD-CS Algorithm Results

Figure 9 shows the results of repartitioning with a cleanness factor of zero and a suppression factor of .5. These were chosen as likely parameters for our multilevel diffusion algorithm since they create a nice compromise between edge-cut, `TotalV`, and `MaxV`. Each of these results are normalized against the results obtained from partitioning the imbalanced graphs from scratch using METIS. These are the same results as those shown under METIS in Figure 5. MLDD-CS indicates results obtained from our multilevel directed diffusion algorithm with cleanness factor of zero and a suppression factor of .5. SLDD indicates results

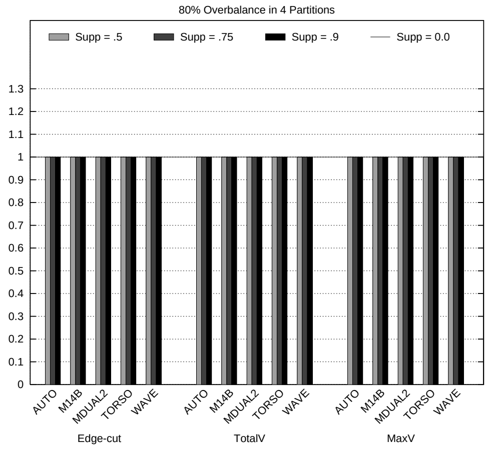

Figure 8(a)

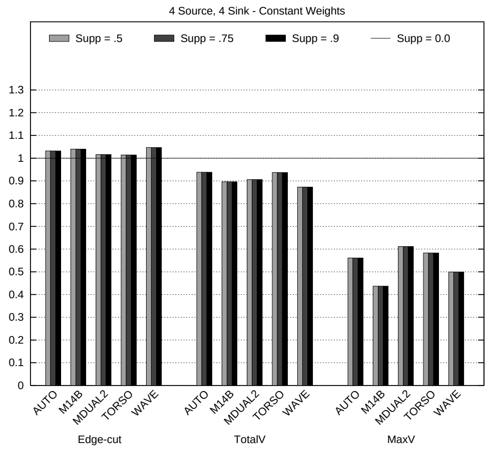

Figure 8(b)

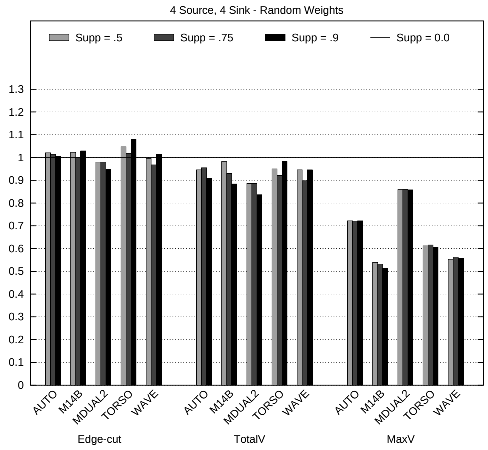

Figure 8(c)

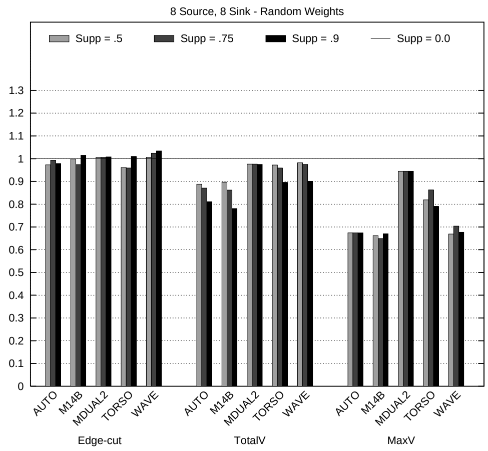

Figure 8(d)

Figure 8: Repartitioning with Suppression

obtained from the single-level directed diffusion algorithm. SLDD-CS indicates results obtained from the single-level directed diffusion algorithm with cleanness factor of zero and a suppression factor of .5. In comparing these results, we see that the multilevel scheme is more effective in employing our heuristics than the single-level diffusion algorithm. By comparing these results with those from Figure 5 in which cleanness and suppression were not used, we can see that `TotalV` and `MaxV` are down considerably for multilevel directed diffusion. The `MaxV` results for SLDD is down as well, but this improvement is smaller, while edge-cut degradation is greater compared with MLDD-CS.

The results show that MLDD-CS employing vertex suppression is able to lower `MaxV` considerably while increasing edge-cut only marginally. This is due to the fact that the effects of using such heuristics are increased by the multilevel view of the graph which the multilevel paradigm provides.

6.4 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

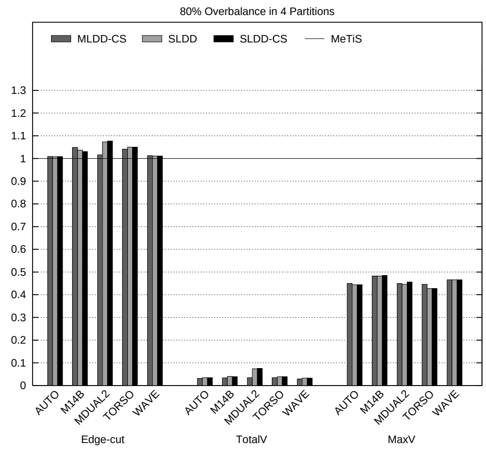

Figure 9(a)

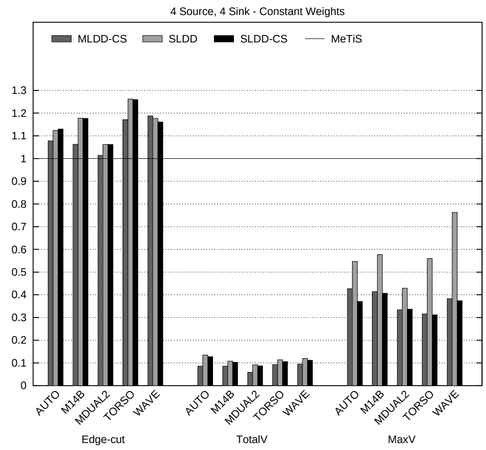

Figure 9(b)

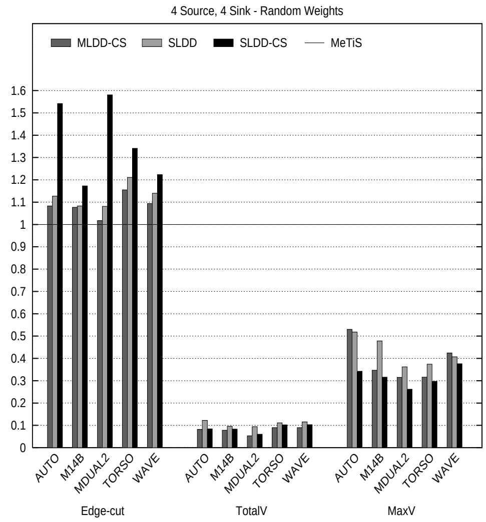

Figure 9(c)

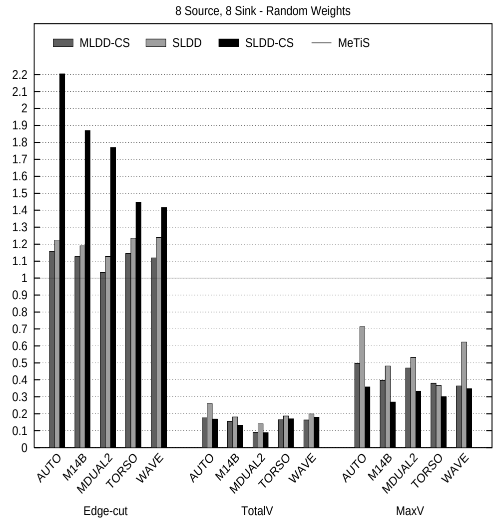

Figure 9(d)

Figure 9: Comparison of Repartitioning Schemes

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. Thus, we have implemented *dynamic suppression*. 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 suppression factor is then reset at each uncoarsening level. The next Section shows the results of using MLDD with dynamic suppression on two application domains.

7 Repartioning of Application Graphs

We have conducted experiments on repartitioning application graphs from two domains. The first set is taken from the DIME software package [18]. 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 [16]. 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 coarsening the mesh in the area where the rotor has entered and refining it in the area of the mesh where the rotor has passed through. These meshes were provided by Rupak Biswas of MRJ Technology Solutions, NASA Ames Research Center, Moffett Field, CA.

For each of these application domains, the first of a series of x graphs, G_1, G_2, G_3, ... , 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 a comparison between our single and multilevel directed diffusion repartitioning algorithm and JOSTLE-D and JOSTLE-MD 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`%. JOSTLE-D is a single-level diffusion algorithm described in [17]. JOSTLE-MD is a single-level diffusion algorithm with multilevel refinement described in [16]. SLDD indicates results obtained from our single-level directed diffusion repartitioning algorithm with cleanness and suppression factors set at zero. MLDD indicates results obtained from the MLDD algorithm. MLDD-CS indicates results obtained from our multilevel directed diffusion repartitioning algorithm with cleanness factor of .0001 and suppression factors of .25. MLDD-CdS indicates results from the MLDD-CS algorithm using a cleanness factor of .0001 and a *dynamic* suppression factors of 1. METIS indicates results from partitioning from scratch with METIS. We have taken the results for JOSTLE-D and JOSTLE-MD directly out of [16]. Note that although experiments in [16] were also done on the graphs obtained from the DIME software package, the graphs used in our experiments are not identical to those used in [16]. However, we attempted to reconstruct the graphs used in [16]. Our graphs are very similar in size and nature to those used in [16]. We use nine graphs with sizes from 31,624 vertices and 46,986 edges to 281,706 vertices and 421,172 edges, while the graphs in [16] range from 23,787 vertices and 35,281 edges to 224,843 vertices and 336,024 edges. All of the results are obtained using a 64-way partition.

The results show that even though our graphs were about 25-30% larger than those repartitioned by JOSTLE-D and JOSTLE-MD, our multilevel diffusion schemes produced edge-cuts within 3, 6, and 11% of JOSTLE-MD while obtaining `TotalV` results 84, 63, and 37% those obtained by JOSTLE-D. All of the multilevel schemes have comparable run times.

Table 2: MLDD Compared to JOSTLE

Algorithm	Edge-Cut	`TotalV`%
JOSTLE-D	2,598	3.76
JOSTLE-MD	2,410	8.82
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

7.2 Helicopter Blade Application

Figure 10 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

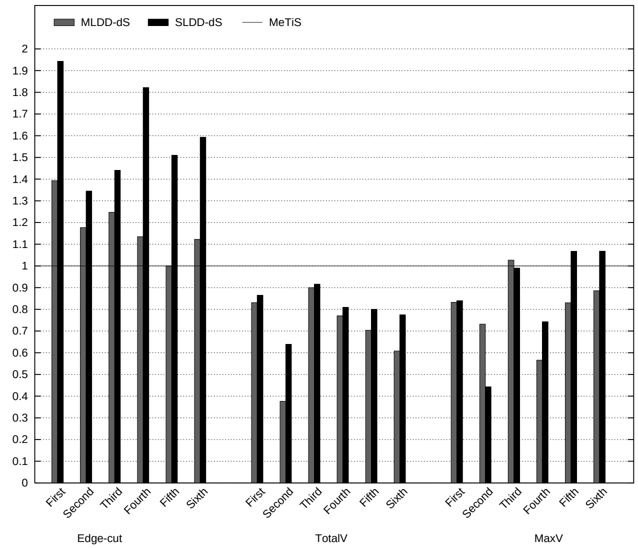

Figure 10: Repartitioning of Helicopter Blade Application Graphs

partition the imbalanced graph from scratch. MLDD-dS indicates the results from the multilevel undirected diffusion repartitioning algorithm using a dynamic suppression factor of .75. SLDD-dS indicates the results from the single-level directed diffusion repartitioner with a dynamic suppression factor of .75.

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. Thus, vertices were often too coarse to be able to be guided my the diffusion solution.

The results confirm that multilevel diffusion is very powerful. We see that the multilevel undirected diffusion repartitioner 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 increasing 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 Conclusions

Our results on a variety of synthetic and application meshes show that multilevel diffusion is a highly robust scheme for repartitioning adaptive meshes. The resulting edge-cuts are generally close to those resulting from partitioning from scratch with a state-of-the-art graph partitioner, 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.

The multi-level diffusion scheme and its variants discussed in this paper are at least as easy to parallelize as the multilevel graph partitioner discussed in [11]. Recently, scalable parallel formulations of METIS's multilevel k-way graph partitioning have been developed [11, 12] that are able to significantly reduce the amount of time required to partition large graphs. The multilevel diffusion algorithms described in this paper can be effectively parallelized using similar techniques. In particular, the local coarsening phase of our multilevel diffusion algorithms are highly parallel since each processor can independently compute successive coarse graphs. The only communication required during this phase are a prefix sum followed by an exchange of labels for the interface vertices in order to maintain consistent numbering. Note that since the graph is already nicely partitioned among the processors, the number of interface vertices is small; hence, the communication overhead of each coarsening step is very small. During the refinement phases groups of vertices are moved among partitions in order to improve the balance as well as reduce the edge-cut. A parallel formulation of this step can select these vertices using either independent sets discussed in [11], or the odd-even scheme discussed in [12]. In either case, the communication overhead of each refinement step is very small as it is proportional to the number of interface vertices of the original partitioning. Note also that in the case of the directed diffusion schemes, the diffusion solution can be easily computed in parallel using a Conjugate-Gradient iterative solver. However, it can also be done serially since the size of the system is only p and the overall amount of time is very small. Our preliminary experiments with such parallel formulation indicates that meshes with over eight million vertices can be repartitioned on 256-processor Cray T3D in well under two seconds.

However, the 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 used in favor of partitioning from scratch.

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

[4] C. M. Fiduccia and R. M. Mattheyses. A linear time heuristic for improving network partitions. In *In Proc. 19th IEEE Design Automation Conference*, pages 175–181, 1982.

[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/metis.html*.

[10] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. Technical Report TR 95-064, Department of Computer Science, University of Minnesota, 1995. Also available on WWW at URL http://www.cs.umn.edu/˜karypis/papers/mlevel_kway.ps.

[11] 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/papers/mlevel_kparallel.ps. A short version appears in Supercomputing 96.

[12] 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.

[13] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 1970.

[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] C. Walshaw, M. Cross, and M. G. Everett. Dynamic load-balancing for parallel adaptive unstructured meshes. *Parallel Processing for Scientific Computing*, 1997.

[17] 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.

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

[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.