The algorithms described in this paper are implemented by the
'PARMETIS: Parallel Graph Partitioning and Sparse Matrix Ordering Library'.
PARMETIS is available on WWW at URL: http://www.cs.umn.edu/~metis

A Performance Study of Diffusive vs. Remapped Load-Balancing Schemes *

Kirk Schloegel†, George Karypis‡, Vipin Kumar§,
Rupak Biswas¶, and Leonid Oliker‖

Abstract

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. Eventually, as the graph evolves, it becomes necessary to correct the partition in accordance with the structural changes in the computation. Partitioning the graph from scratch and then intelligently remapping the resulting partition will accomplish this task. Two different types of schemes to accomplish this task have been developed recently. In one scheme, the graph is partitioned from scratch and then the resulting partition is remapped intelligently to the original partition. The second type of scheme use a multilevel diffusion repartitioner. In this paper, we conduct a comparison study on repartitioning via intelligent remapping versus repartitioning by diffusion. We show that multilevel diffusion algorithms generally produce significantly lower data migration overhead for adaptive graphs in which low magnitude localized or non-localized imbalances occur and for graphs in which high magnitude imbalances occurs globally throughout the domains than partitioning from scratch and remapping the resulting partition. We show that for the class of problems in which high magnitude imbalances occur in localized areas of the graph, partitioning from scratch and remapping the resulting partition will result in very low edge-cuts and data migration overheads which are similar to those obtained by diffusive schemes. Finally, we show that the run times of the various schemes are all similar.

*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**

†University of Minnesota, Department of Computer Science & Engineering / Army HPC Center, Minneapolis, MN
‡University of Minnesota, Department of Computer Science & Engineering / Army HPC Center, Minneapolis, MN
§University of Minnesota, Department of Computer Science & Engineering / Army HPC Center, Minneapolis, MN
¶NASA Ames Research Center, Moffett Field, CA
‖NASA Ames Research Center, Moffett Field, CA

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 can be represented by the weight of its associated vertex. Similarly, the amount of runtime interaction required between two grid points can be 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 objective that the total number of edges cut by the partitions (hereafter referred to as *edge-cut*) is minimized, and subject to the constraint that each partition has an equal amount of total vertex weight. 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, while requiring that each partition has the same amount of vertex weight ensures load balance. This problem has been well defined and discussed in previous work [3, 7].

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 [2], areas of the original graph are selectively coarsened or refined in order to accurately model the dynamic nature of the problem. The adapted meshes can be represented by appropriately modifying the weights of the vertices and edges of the original graph. The advantage of this strategy to repeatedly use the initial graph during the course of an adaptive computation over constructing a new graph with vertex and edge weights equal to one is that the connectivity and the partitioning complexity remain unchanged. 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.

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 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.

Any repartitioning algorithm also needs to satisfy the following objectives.

1. **It minimizes edge-cut.** The redistributed graph should have a small edge-cut to minimize communication overhead in the actual computation phase.

2. **It minimizes vertex migration time**. Once the mesh is repartitioned, and before the 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 therefore dominate the overall run time, especially if the mesh is adapted frequently.

Partitioning the graph from scratch and then intelligently remapping the resulting partition is one way to meet the above criteria. Various algorithms which do so are described in [8]. Here, the imbalanced graph is partitioned from scratch using one of the multilevel graph partitioning algorithms described in [6, 7]. The resulting partition is then intelligently mapped to the processors in order to reduce the amount of vertex migration required. In [1], a simple greedy remapping algorithm was described and shown to obtain near-optimal results on application graphs.

Another way to meet the above criteria is through diffusive repartitioning. Multilevel diffusion schemes have been developed that incrementally construct a new partition of the graph [10, 9, 11]. These schemes generally have three phases: a graph coarsening phase, a diffusion phase, and a multilevel refinement phase. However, the diffusion and refinement phases may be interleaved.

In the graph coarsening phase, these algorithms attempt to coarsen the input graph recursively by collapsing matched vertices into a single vertex on the next coarser graph. Vertices are matched together by computing a maximal set of independent edges. Most multilevel diffusion schemes use some variation of heavy-edge matching [5] to compute this set. In the diffusion phase, the coarsest (and hence smallest) graph is balanced by incrementally modifying the input partition. By beginning this process on the coarsest graph, these algorithms are able to move large chunks of well-connected vertices in a single step. Thus, the bulk of the work required to balance the graph is done

quickly. Eventually, due to the coarseness of the graph, an incremental diffusion process may not be able to improve the load balance. At this point, either refinement is begun on the current graph (in the case of interleaved diffusion and refinement phases) or the partition is projected to the next finer graph and another round of diffusion begins. In these algorithms, diffusion may be directed globally or locally. In multilevel refinement, border vertices (or pairs of border vertices) from the coarsest graph are selected in some order. Each vertex is examined to determine if migrating it to an adjacent domain will accomplish the primary objective (usually to reduce the edge-cut) while maintaining the balance constraint. If this is so, the switch is made. Once a local minima in this search space is reached for the current graph, the partition is projected to the next finer level graph and this refinement process begins again.

2 Notations, Definitions, and Issues

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 moving vertex v_i. We will refer to s_i as the *size* of vertex i [10]. 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. We denote $B(q)$ as the set of vertices with partition q. The weight of any partition q can then be defined as:

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

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. In this paper, we refer to a k-way partition as being composed of k disjointed *domains*.

As discussed in the introduction, the effectiveness of repartitioning algorithms quite often is determined by how successful they are in load balancing the computations while minimizing the edge-cut as well as the cost associated in redistributing the load in order to realize the new partitioning. Two metrics that are widely used [1, 10, 11] for measuring this redistribution cost are TOTALV which measures the total volume of data moved among all processors, and MAXV which measures maximum flow of data to or from any single processor. Specifically, TOTALV is defined as the sum of the sizes of the 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 and out of any one partition as a result of partitioning or repartitioning. Note that by minimizing TOTALV, we try to reduce the redistribution time by reducing the overall network contention and the total number of elements that is moved, whereas by minimizing MAXV, we try to reduce the redistribution time by reducing the maximum amount of data any processor needs to send and receive.

3 Repartitioning via Intelligent Remapping

Repartitioning algorithms can take advantage of a key fact: multilevel graph partitioning algorithms exist which are able to consistently compute high quality partitions [3, 6, 7, 12]. Thus, the edge-cut results obtained by these schemes are extremely difficult to beat.

Once a new partition is computed, it is useful to map the new sub-domains to the processors such that the redistribution cost is minimized. In theory, the number of new sub-domains can be an integer multiple f of the number of processors. Each processor is then assigned f unique sub-domains. The rationale for allowing multiple sub-domains per processor is that data mapping at a finer granularity reduces the volume of data movement at the cost of a slightly larger partitioning time. The first step toward processor reassignment is to compute a similarity measure that indicates how the vertex sizes of the new sub-domains are distributed over the p processors. It is straightforward to represent this measure as a matrix S, where entry S_{ij} is the sum of the sizes of all the graph vertices in new partition j that already reside on processor i. A similarity matrix for $p = 4$ and $f = 2$ is shown in Figure 1. Only the non-zero entries are shown.

The goal of the processor reassignment phase is to find a mapping between the new partitions and the processors such that the data redistribution cost is minimized. Both the optimal and a heuristic greedy algorithm have been implemented for solving the processor reassignment problem using TOTALV [8]. The optimal solution can be obtained from the corresponding maximally-weighted bipartite graph. Applying the heuristic procedure to the similarity matrix

		New Partitions							
		0	1	2	3	4	5	6	7
Old Processors	0		1020		120				
	1			500		443	372		
	2	129	130		229			43	446
	3	13	410	281				198	
		3	0	1	2	1	0	3	2
		New Processors							

Figure 1: A similarity matrix after processor reassignment using the heuristic algorithm and the TOTALV metric.

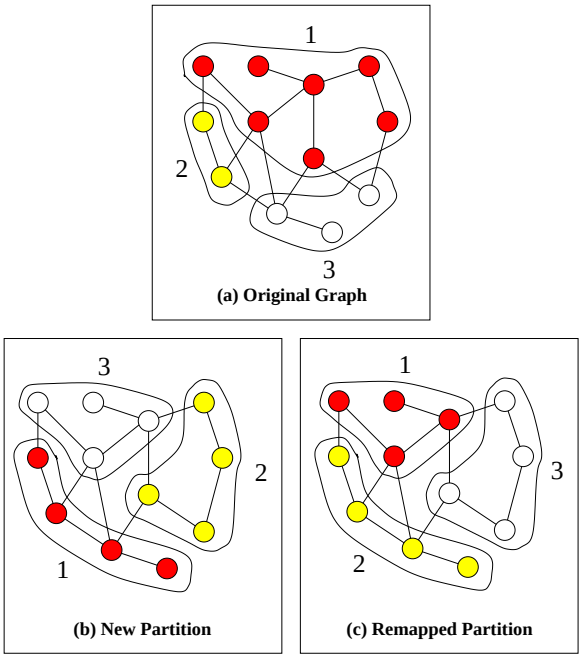

Figure 2: Repartitioning via Intelligent Remapping

in Fig. 1 generates the processor assignment shown in the bottom row. It was proved in [8] that a processor assignment obtained using the heuristic algorithm can never result in a data movement cost that is twice that of the optimal assignment. MAXV, on the other hand, considers data redistribution in terms of solving a load imbalance problem, where it is more important to minimize the workload of the most heavily-weighted processor than to minimize the sum of all the loads. The optimal algorithm for solving the assignment problem using MAXV has also been implemented [8]. The optimal solution for MAXV can be obtained by considering processor reassignment as a bottleneck maximum cardinality matching problem.

Fig. 2 shows a partition remapping example. If we assume the weight of each vertex to be one, the graph in Fig. 2(a) is imbalanced because domain 1 has seven vertices while domain 2 has two and domain 3 has three. In Fig. 2(b), the graph has been partitioned again from scratch. Since, the new partition was computed without regard for the original partition, a large amount of vertex migration is required here. In this case, every vertex must swap domains. However, by remapping the newly computed partition with respect to the original partition-to-processor assignment, the amount of vertex migration can be substantially reduced. In Fig. 2(c), this has been done. The number of vertices which are required to swap domains has now dropped from 12 (all of them) to five. Notice that since remapping only changes the labels of the domains and not the partition itself, the edge-cut is not affected. Thus, intelligent remapping can reduce the amount of data required to balance the graph while maintaining the edge-cut of the newly computed partition.

3.1 Parallel Repartitioning via Intelligent Remapping

Efficient parallel graph partitioners exist [7]. Furthermore, the remapping phase can be performed serially on a single processor, since the complexity of remapping is dominated by a sorting routine on the number of processors. Therefore, parallelizing intelligent remapping schemes is straightforward. Each processor can simultaneously compute one row of the matrix S, based on the mapping between its current sub-domain and the new partitioning. This information is then gathered by a single processor that assembles the complete similarity matrix, computes the new partition-to-processor mapping, and scatters the solution back to the processors. These gather and scatter operations require a negligible amount of time since only one row of S needs to be communicated to the host processor.

4 Multilevel Diffusion Algorithms

Several serial multilevel diffusion algorithms are described in [10, 11]. These algorithms have three phases: a coarsening phase, a multilevel diffusion phase, and a multilevel refinement phase. 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. Different algorithms implement these steps somewhat differently. In this paper, we will use two of the multilevel diffusion schemes described in [10]. In the multilevel diffusion phase, balance is sought by means of either ***undirected*** or ***directed*** diffusion. In the case of undirected diffusion, the border vertices are visited in a random order. If a vertex belongs to an overweight domain (*i.e.*, a domain whose weight is higher than the average weight), then it is moved to an adjacent domain with lower weight. If there are more than one adjacent domain satisfying this condition, the one that leads to the smaller edge-cut is selected. If a vertex belongs to a domain with average weight, then it is moved to a domain that leads to a reduction in the edge-cut as long as it does not make the destination domain overweight. Again, if there are more than one domains satisfying this condition, the one that leads to the smaller edge-cut is selected. If a vertex belongs to an under-weight domain, it is not moved. Note that a vertex belonging to an overweight domain can move even if this migration leads to an increase in the edge-cut. This process is repeated for a small number of steps or until either balance is obtained or no progress is made in balancing. Note that in this scheme, diffusion is performed using only local information.

In the case of directed diffusion, a global picture is used to guide the vertex migration. This global picture is computed by the 2-norm minimization solution described in [4]. The result of this computation is a transfer matrix that indicates how much weight needs to be transfered between neighboring domains. Using this transfer matrix, the directed diffusion scheme works as follows. Again, the border vertices are visited in a random order. If a vertex has a neighbor domain which according to the transfer matrix needs work, then the vertex can be migrated to the neighbor domain. If a vertex is a neighbor of more than one such domain, it is migrated to the domain that will produce the highest gain in edge-cut. The vertex is migrated even if the gain in edge-cut is negative. After a vertex is migrated, the transfer matrix is updated to reflect the vertex migration (*i.e.*, the weight of the vertex that was moved is subtracted from the appropriate entry of the transfer matrix). After each border vertex is visited exactly once, the process repeats until either balance is obtained or no progress is made in balancing.

In either of these schemes that are used during the multilevel diffusion phase, 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.

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 border vertices are again visited randomly and are checked to see if they can be migrated to another domain so that

1. the edge-cut and graph balance are maintained, and the selected domain is the vertex's initial domain from the input graph, or
2. the edge-cut is decreased while the graph balance is maintained, or
3. the edge-cut is maintained and the graph balance is improved.

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 domains (as long as the migration does not increase the edge-cut and worsen the load balance), and therefore, to lower TOTALV and possibly MAXV [10].

4.1 Parallel Multilevel Diffusion Algorithms

Parallel versions of multilevel diffusion algorithms have been described in [9, 11]. Here, vertices are initially assumed to be distributed across p processors. This division of vertices corresponds to the original partition of a static partitioner and is assumed to be of good quality (*i.e.*, low edge-cut). However, the sums of the vertex weights of the vertices resident on each processor are assumed to be variant. Thus, the original partition is not balanced and so there is a need for repartitioning.

Parallel multilevel repartitioning algorithms typically begin with a coarsening phase in which a sequence $G_i = (V_i, E_i)$ for $i = 0, 1, ..., m$, of successively coarser graphs is constructed. Graph G_{i+1} is constructed from G_i by first computing a matching of vertices of G_i and then collapsing together the matched vertices. The matchings computed are restricted to vertices residing on the same processors. By adhering to this restriction, coarsening is almost embarrassingly parallel.

The parallel formulation of the multilevel diffusion phase described in [10] depends on whether directed or undirected diffusion is being utilized. In the directed case, diffusion is performed on the coarsest graph serially. Since the coarse graph is very small (its size is proportional to the number of processors), this serial computation does not significantly affect the overall performance and scalability of the parallel multilevel directed diffusion algorithm. Furthermore, the additional processors are utilized to obtain a higher-quality partition as follows. The graph is broadcast to all processors. Each processor then simultaneously balances the coarsest graph using the directed diffusion algorithm described in Section 4. Since the diffusion scheme is inherently random, each processor computes a potentially unique partition. The best partition is selected. The selection criteria are either; (i) lowest edge-cut, (ii) lowest TOTALV, (iii) lowest MAXV, or (iv) best balance. In [10], the partition that has the lowest edge-cut is selected. In some cases, the coarsest graph may be too coarse to allow for complete balancing. For this reason, the parallel undirected diffusion algorithm (described in the next paragraph) is then utilized to correct any minor imbalances.

The parallel formulation of the undirected diffusion algorithm described in [9] is modeled after coarse-grained parallel multilevel refinement algorithm [7]. Each iteration of the parallel multilevel refinement algorithm consists of two sub-phases. During the first sub-phase, vertices are migrated only from lower- to higher-numbered domains. During the second sub-phase, vertices are migrated from higher- to lower-numbered domains. In this way, unexpected edge-cut increases caused by the simultaneous migration of neighboring vertices is avoided. Furthermore, these schemes avoid any bias towards the lower- or higher-numbered domains, by using a random partition ordering at each step. In each sub-phase, vertices are visited and selected for migration according to the criteria for undirected diffusion described earlier.

The parallel formulation of multilevel refinement algorithms are similar to that for undirected diffusion with the exception that vertices are moved according to the *refinement phase vertex migration criteria* described in [9].

5 Experimental Results

We tested our parallel repartitioning algorithms on a Cray T3E-1200 with 128 processors. Each processor on the T3E-1200 is a 600 Mhz Dec Alpha (EV5). The processors are interconnected via a three dimensional torus network that has a peak unidirectional bandwidth of 450 bytes per second, and a small latency. We used Cray's MPI library for communication. Cray's MPI achieves a peak bandwidth of 200 MBytes and an effective startup time of 30 microseconds.

We evaluated the performance of the parallel repartitioning algorithms described above on synthetically generated adaptive meshes, derived from three medium to large size 3D finite element meshes, on 32, 64, and 128 processors. The characteristics of the corresponding dual graphs of these meshes are described in Table 1. Our synthetically generated problems simulate the process of mesh adaptation by changing the weight of the vertices and the edges of the graph. The weight of some of the vertices was changed from one (prior to adaptation) to a number greater than one (indicating the degree of adaptation). The weight of edges between *adapted* vertices was also changed to reflect the higher degree of connectivity in the adapted graph. For every edge (v_i, v_j), its weight was set to $(\min(w_i, w_j))^{2/3}$ (where w_i is the weight of vertex v_i). Note that this edge-weighting model tries to capture the face connectivity of 3D finite element meshes. For each of these synthetically generated graphs, the parallel repartitioning algorithms are used to compute a new partitioning such that the sum of the weight of the vertices assigned to each processor is the same.

For each of the three meshes and for each processor configuration, we generated two types of adaptive meshes, TYPEA and TYPEB. Meshes of TYPEA attempt to simulate the situation in which the mesh is adapted at various regions throughout the mesh, whereas meshes of TYPEB attempt to simulate the situation in which the adaptation

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

Table 1: The various graphs used in the experiments.

occurs in a small region of the mesh. For both TYPEA and TYPEB meshes on p processors, we first computed a p-way partitioning of the graph (using the parallel k-way graph partitioner implemented in PARMETIS), and then redistributed the graph according to this partitioning. This became the initial partitioning that we used to adjust the weight of the vertices to emulate the effect of adaptation.

For meshes of TYPEA, the weights of the vertices were changed as follows. Each processor generated a random number r between zero and 100, then it randomly selected $r\%$ of its vertices, and set their weight to α. The weight of the remaining vertices was set to one. For meshes of TYPEB, the weights of the vertices were changed as follows. Each processor generated a random number between zero and $p-1$, and then for the processors in which r was less than $0.05p$, the weight of all the vertices in these processors was set to α. The weight of the remaining vertices was set to one. Note that the expected number of processors that will increase the weight of its vertices is 1.6, 3.2, and 6.4 for 32, 64, and 128 processors, respectively. For both TYPEA and TYPEB problems, we let α take the values of 2, 5, 10, 20, 30, and 40. These synthetically generated graphs are then used as the input to the parallel repartitioning algorithms described in Sections 3 and 4. In all of our experiments, a graph that has less than 5% load imbalance is assumed to be well balanced. Also, in all of our experiments, the cost to migrate a vertex between processors is assumed to be proportional to its weight.

The parallel implementations which we used for the experiments are all from PARMETIS. That is, PARMETIS implements the undirected diffusion algorithm described in Section 4.1 in the subroutine PARUAMETIS, the directed diffusion algorithm in PARDAMETIS, and the intelligent greedy remapping algorithm in PARPAMETIS.

In order to compare the three schemes we graphically depicted the results in the sequence of graphs shown in Figs. 3– 8. In particular, Fig. 3 compares the quality in terms of edge-cut and TOTALV produced by the two multilevel diffusion schemes (PARUAMETIS and PARDAMETIS) relative to partitioning from scratch and then performing a greedy remapping (PARPAMETIS) for TYPEA experiments in which α was set to 2, 5, and 10. For each experiment, we computed the ratio of the edge-cut and TOTALV produced by the diffusion algorithms to that of PARPAMETIS and plotted it using a bar chart.

Performance on Slightly to Moderately Adapted Graphs The experiments shown in Fig. 3 simulate a low to moderate degree of adaption taking place globally throughout the adapted graph. Fig. 3 shows that the edge-cuts obtained by the multilevel diffusion algorithms were, in general, 0% to 10% higher than those obtained by PARPAMETIS. Furthermore, the degradation in edge-cut for the diffusion schemes tended to increase with α. Thus, as the level of imbalance increased, PARPAMETIS did an increasingly better job than either PARUAMETIS or PARDAMETIS in minimizing the edge-cut. The TOTALV produced by both of the diffusion schemes is considerably less than that obtained by PARPAMETIS. In general, diffusion resulted in TOTALV which is 15% to 30% that produced by the intelligent remapping scheme. As the level of imbalance increased, the difference in TOTALV results tended to decrease. Comparing PARUAMETIS with PARDAMETIS, the edge-cut and TOTALV results are generally very similar (within 5%). However, PARUAMETIS tended to obtain slightly lower TOTALV results than PARDAMETIS.

The experiments shown in Fig. 4 simulate a low degree of adaption taking place in localized areas of the adapted graph. Fig. 4 shows that the edge-cuts obtained by the multilevel diffusion algorithms were, in general, 0% to 20% higher than those obtained by PARPAMETIS. Again, the degradation in edge-cut for the diffusion schemes tended to increase with α. The TOTALV produced by both of the diffusion schemes is again considerably less than that obtained by PARPAMETIS. Here, diffusion resulted in TOTALV which is 10% to 70% of that produced by PARPAMETIS. Furthermore, as the level of imbalance increased, the difference in TOTALV results tended to decrease dramatically. This indicates that as the level of localized imbalance increases, PARPAMETIS's performance increasingly approaches that of the diffusion algorithm with respect to TOTALV. Comparing PARUAMETIS with PARDAMETIS, the edge-cut and TOTALV results are again similar (usually within 4-6%).

In summary, for this class of problems, the results indicate that multilevel diffusion can significantly reduce the amount of total vertex migration required to realize the new partition while maintaining quality in terms of edge-cut

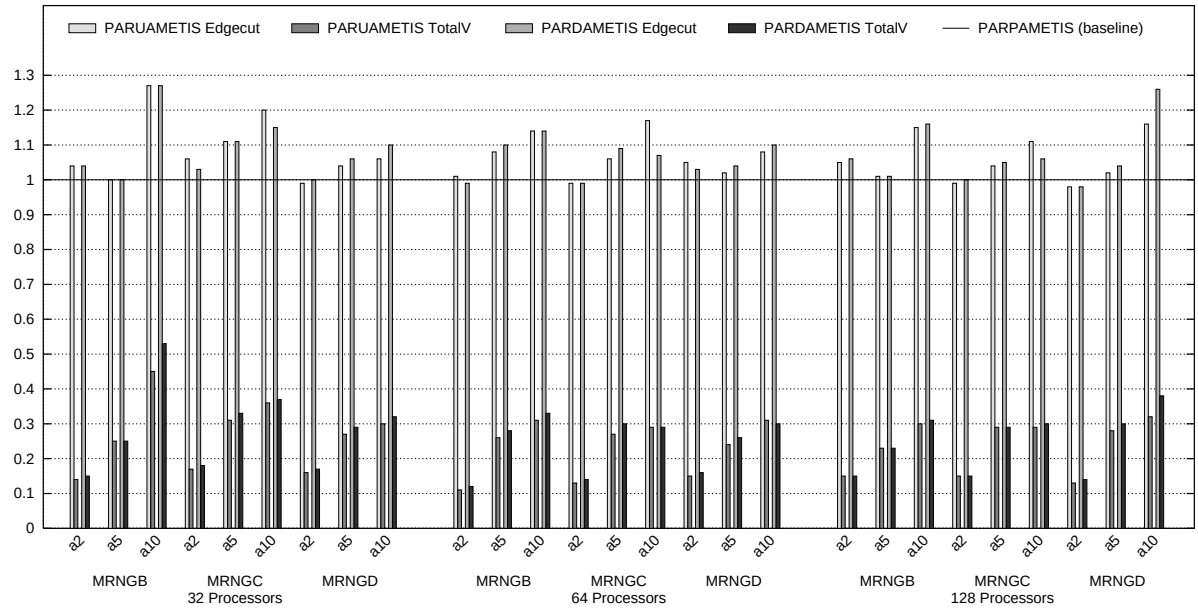

Figure 3: Quality in terms of edge-cut and TOTALV of the partitionings produced by the multilevel directed and undirected diffusion algorithms relative to partitioning from scratch followed by intelligent remapping for low imbalance TYPEA problems. For each graph, the ratio of the edge-cut and TOTALV of the multilevel diffusion algorithms to that of multilevel k-way partitioning and subsequent greedy remapping is plotted for 32-, 64-, and 128-way partitions on 32, 64, and 128 processors, respectively. Bars under the baseline indicate that the multilevel diffusion algorithms perform better than the remapping algorithm.

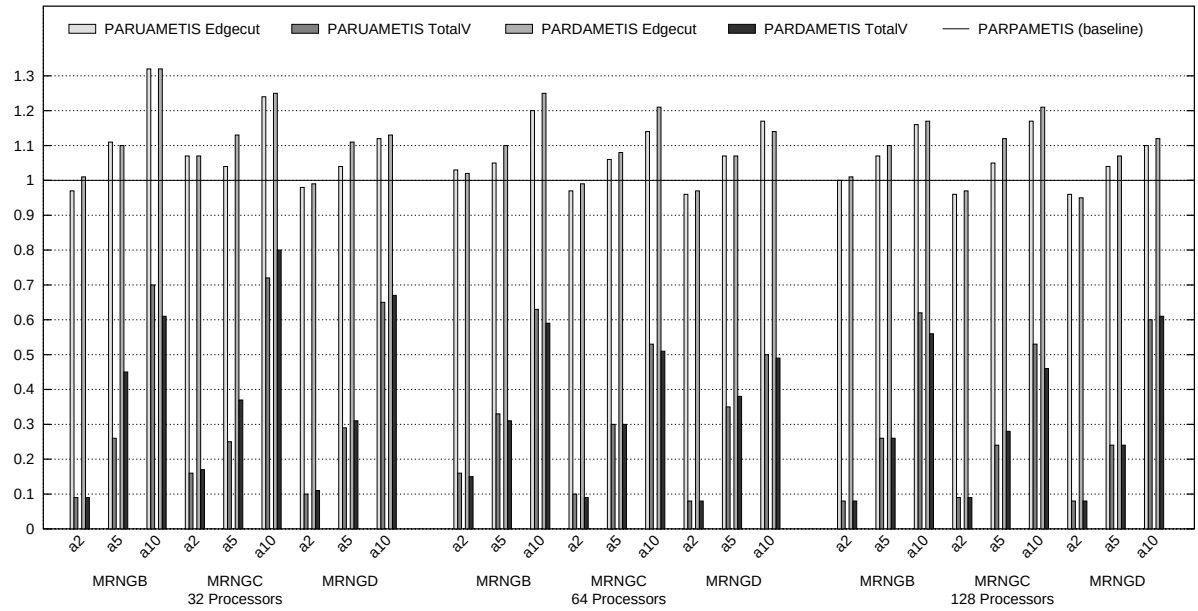

Figure 4: Quality in terms of edge-cut and TOTALV of the partitionings produced by the multilevel directed and undirected diffusion algorithms relative to partitioning from scratch followed by intelligent remapping for low imbalance TYPEB problems. For each graph, the ratio of the edge-cut and TOTALV of the multilevel diffusion algorithms to that of multilevel k-way partitioning and subsequent greedy remapping is plotted for 32-, 64-, and 128-way partitions on 32, 64, and 128 processors, respectively. Bars under the baseline indicate that the multilevel diffusion algorithms perform better than the remapping algorithm.

which is comparable to that of intelligent remapping algorithms. These results indicate that the way to do repartitioning for applications with low levels of adaption is to perform multilevel diffusion. However, after a number of diffusion iterations, the quality of the partition may begin to deteriorate. Therefore, a new partition should be computed (and remapped) in order to restore a high-quality starting point for further diffusion iterations. Thus, the edge-cut and the amount of data migration overhead are both kept low.

Performance on Highly Adapted Graphs The experiments shown in Fig. 5 simulate a high degree of adaption taking place globally throughout the adapted graph. Fig. 5 shows that the edge-cuts obtained by the multilevel diffusion algorithms were, in general, 20% to 30% greater than those obtained by PARPAMETIS. Again, the degradation in edge-cut for the diffusion schemes tended to increase with α. The TOTALV produced by both of the diffusion schemes is considerably less than that obtained by PARPAMETIS. Diffusion resulted in TOTALV which is 30% to 40% that produced by the intelligent remapping scheme. Here however, the relative TOTALV results remain constant (independent of α) between the diffusion algorithms and PARPAMETIS. Comparing PARUAMETIS with PARDAMETIS, we again see only a slight difference among the schemes.

The experiments shown in Fig. 6 simulate a high degree of adaption taking place in localized areas of the adapted graph. Fig. 6 shows that the edge-cuts obtained by the multilevel diffusion algorithms were, in general, 20% to 30% greater than those obtained by PARPAMETIS. Notice that for three of the experiments, the edge-cuts obtained by the diffusion algorithms were 58% to 98% higher than those obtained by PARPAMETIS. This is because in these experiments, the diffusion algorithms failed to balance the graphs to within 5%. Because of this, multilevel refinement never began, and so the edge-cut quality suffered. Fig. 6 also shows that the TOTALV produced by PARPAMETIS is generally similar or lower than that produced by the diffusion algorithms. This is especially true of the highly imbalanced problems. Note, these results indicate that for this class of problems, PARPAMETIS is more effective in reducing both the edge-cut and the TOTALV than multilevel diffusion algorithms.

In summary, for these classes of problems, the results indicate that multilevel diffusion can reduce the amount of total vertex migration required to realize the new partition while maintaining quality in terms of edge-cut only for graphs in which imbalances occur globally throughout the graph. For localized adaptions of the mesh with high levels of imbalance, partitioning from scratch and remapping the newly computed partition produces both edge-cut and TOTALV results which are similar or lower than those obtained by diffusion. For this class of problems, the adapted graph should be partitioned from scratch after each iteration as diffusion will not help in lowering data migration.

MAXV Results Fig. 7 gives the relative MAXV performance of the three schemes for TYPEA experiments. Results show that, in general, the diffusion schemes produced much lower MAXV results than PARPAMETIS. Specifically, the MAXV results obtained by the multilevel diffusion algorithms were about 20% to 70% of those obtained by PARPAMETIS.

Fig. 8 gives the relative MAXV performance of the three schemes for TYPEB experiments and shows that the diffusion schemes again produced lower MAXV results, in general, than PARPAMETIS. However, here there was a convergence point as α increased in which all three algorithms produced generally similar MAXV results. This is because the MAXV of these problems is dominated by the amount of vertex weight that needs to move out of the few overbalanced domains. This migration is required in order to balance the graph, and so becomes the lower bound for MAXV.

Run Time Results Table 2 gives some sample data points with respect to the execution times of the various algorithms. All of these results are from the largest graph, MRNGD. Table 2 shows that the run times of all of the schemes are quite fast. Also, the run times are very similar among the schemes. Notice, however, that the diffusion algorithms tend to take longer for the TYPEB problems than they do for TYPEA problems, while PARPAMETIS times are more consistent. This is because localized imbalances cause diffusion to propagate further on average than global imbalances. Thus, diffusion algorithms will require more iterations of diffusion in order to become balanced for TYPEB problems than for TYPEA problems. Hence, the run times are higher. Computing a new partition from scratch avoids this problems. Thus, the PARPAMETIS times are not affected by the localized imbalances.

6 Conclusion and Related Work

In this paper, we have shown that multilevel diffusion algorithms are generally able to produce significantly lower TOTALV and MAXV results for adaptive graphs in which either localized or non-localized imbalances are low in

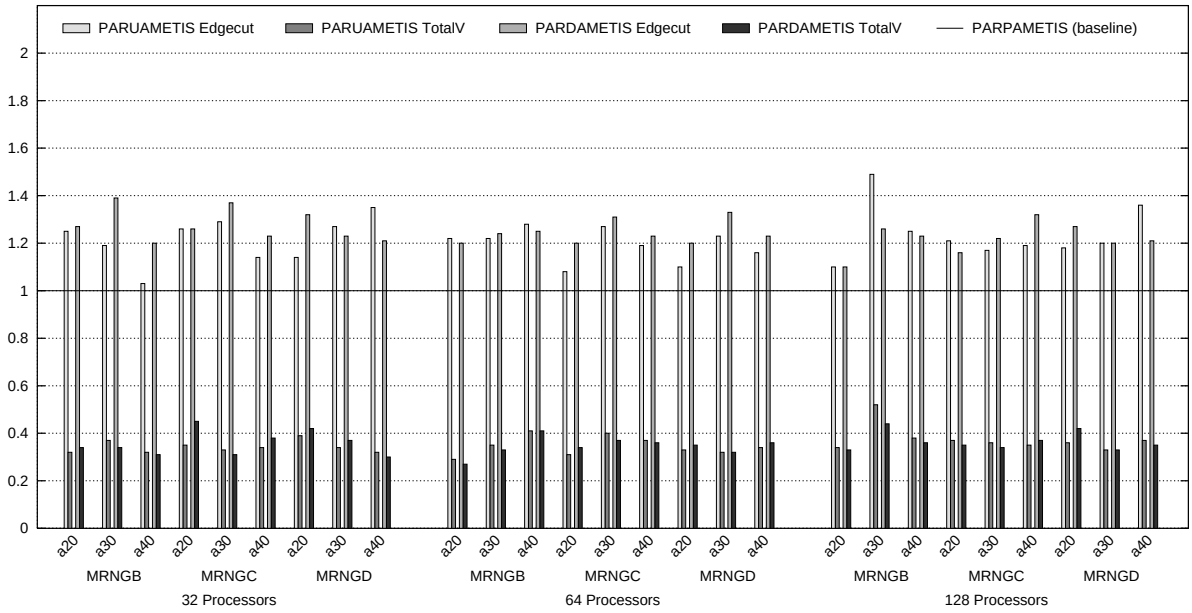

Figure 5: Quality in terms of edge-cut and TOTALV of the partitionings produced by the multilevel directed and undirected diffusion algorithms relative to partitioning from scratch followed by intelligent remapping for high imbalance TYPEA problems. For each graph, the ratio of the edge-cut and TOTALV of the multilevel diffusion algorithms to that of multilevel k-way partitioning and subsequent greedy remapping is plotted for 32-, 64-, and 128-way partitions on 32, 64, and 128 processors, respectively. Bars under the baseline indicate that the multilevel diffusion algorithms perform better than the remapping algorithm.

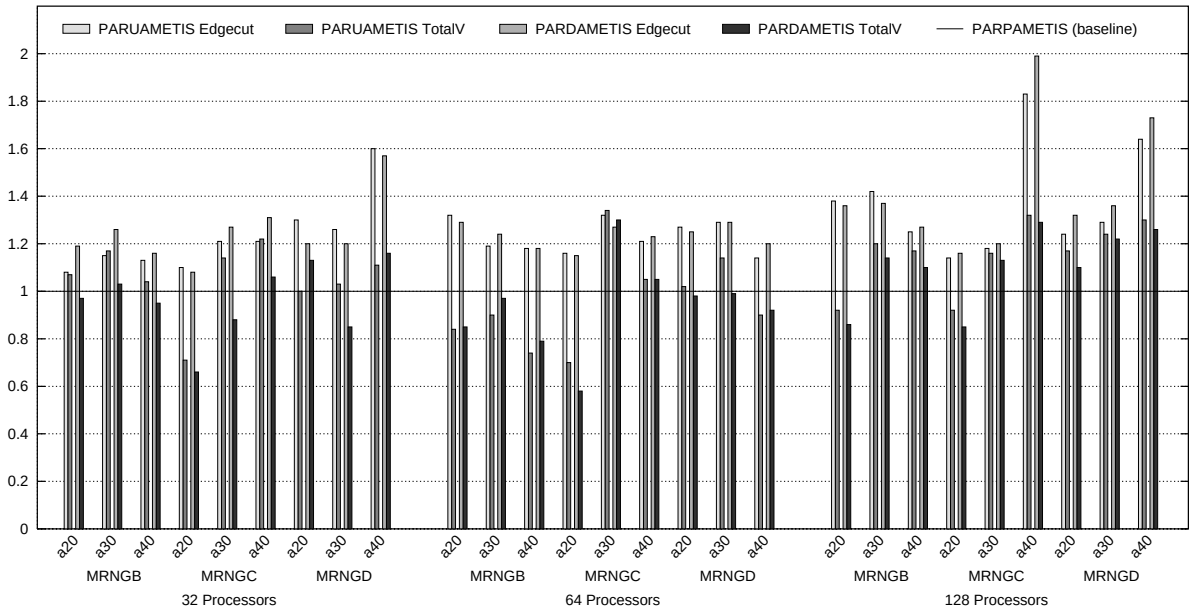

Figure 6: Quality in terms of edge-cut and TOTALV of the partitionings produced by the multilevel directed and undirected diffusion algorithms relative to partitioning from scratch followed by intelligent remapping for high imbalance TYPEB problems. For each graph, the ratio of the edge-cut and TOTALV of the multilevel diffusion algorithms to that of multilevel k-way partitioning and subsequent greedy remapping is plotted for 32-, 64-, and 128-way partitions on 32, 64, and 128 processors, respectively. Bars under the baseline indicate that the multilevel diffusion algorithms perform better than the remapping algorithm.

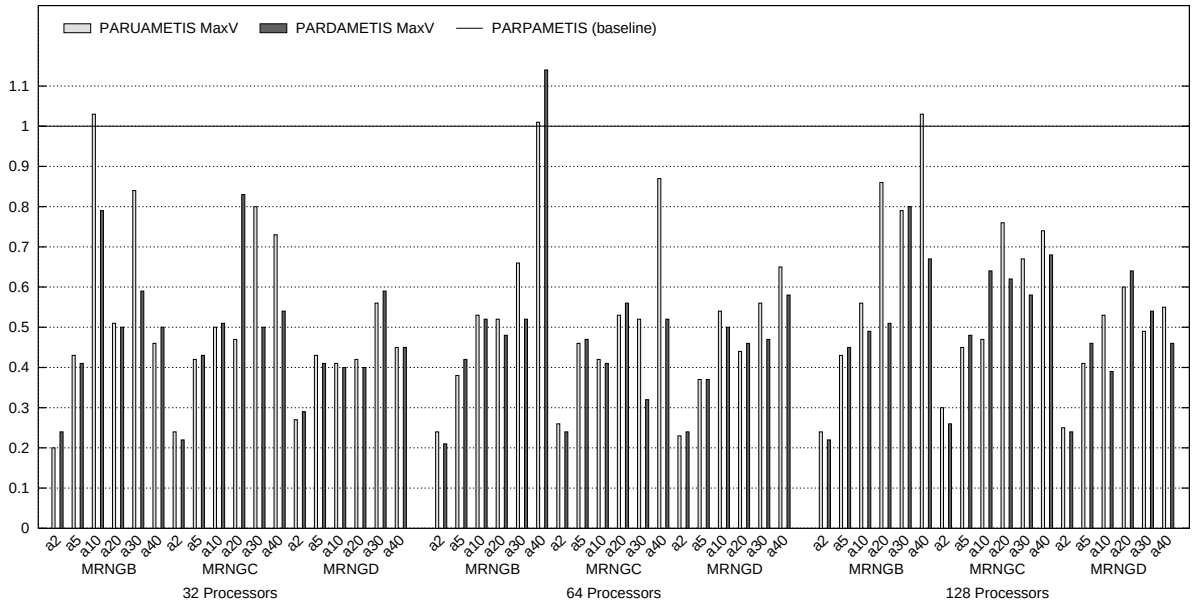

Figure 7: Quality in terms of MAXV of the partitionings produced by the multilevel directed and undirected diffusion algorithms relative to partitioning from scratch followed by intelligent remapping for TYPEA problems. For each graph, the ratio of the MAXV produced by the multilevel diffusion algorithms to that of multilevel k-way partitioning and subsequent greedy remapping is plotted for 32-, 64-, and 128-way partitions on 32, 64, and 128 processors, respectively. Bars under the baseline indicate that the multilevel diffusion algorithms perform better than the remapping algorithm.

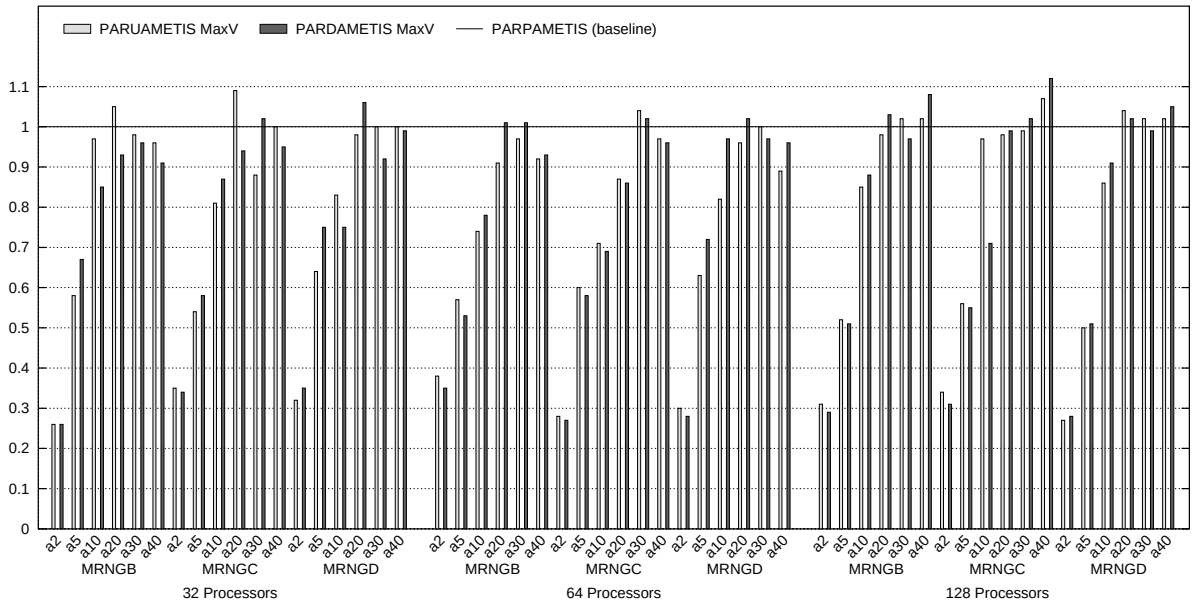

Figure 8: Quality in terms of MAXV of the partitionings produced by the multilevel directed and undirected diffusion algorithms relative to partitioning from scratch followed by intelligent remapping for TYPEB problems. For each graph, the ratio of the MAXV produced by the multilevel diffusion algorithms to that of multilevel k-way partitioning and subsequent greedy remapping is plotted for 32-, 64-, and 128-way partitions on 32, 64, and 128 processors, respectively. Bars under the baseline indicate that the multilevel diffusion algorithms perform better than the remapping algorithm.

Problem	Partition	α	PARUAMETIS	PARDAMETIS	PARPAMETIS
TYPEA	32-way	5	5.469	5.529	5.134
TYPEA	32-way	30	5.555	5.551	5.198
TYPEA	64-way	5	2.834	2.859	2.725
TYPEA	64-way	30	2.816	2.870	2.799
TYPEA	128-way	5	1.546	1.602	1.791
TYPEA	128-way	30	1.581	1.661	1.846
TYPEB	32-way	5	5.568	5.488	5.079
TYPEB	32-way	30	5.470	5.488	5.047
TYPEB	64-way	5	2.860	2.880	2.743
TYPEB	64-way	30	2.930	3.029	2.758
TYPEB	128-way	5	1.563	1.641	1.764
TYPEB	128-way	30	1.895	2.059	1.817

Table 2: The run times of selected experiments.

magnitude as well as for graphs in which high magnitude imbalances occur globally throughout the grid. This is because the optimal solution for these problems is relatively near to the initial partition in the search space. Hence, diffusion, which attempts to minimize the difference between the original partition and the output partition, is a good strategy here. For these classes of problems, diffusive repartitioning should be applied a number of times, in order to keep data migration overhead low, followed by a single iteration of partitioning from scratch and remapping the resulting partition. This will allow edge-cut results to remain low, while minimizing the data migration overhead.

For the class of problems in which a large amount of imbalance occurs in localized areas of the graph, partitioning from scratch and remapping the resulting partition will result in very low edge-cuts and TOTALV and MAXV results which are similar to those obtained by multilevel diffusive schemes. This is because the optimal solution for these problems is substantially removed from the initial partition in the search space. Hence, partitioning from scratch and intelligent remapping should be followed in favor of multilevel diffusion for such cases.

References

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

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

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

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

[5] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. Technical Report TR 95-035, Department of Computer Science, University of Minnesota, 1995. Also available on WWW at URL http://www.cs.umn.edu/~karypis. A short version appears in Intl. Conf. on Parallel Processing 1995.

[6] 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.

[7] 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.

[8] L. Oliker and R. Biswas. Plum: Parallel load balancing for adaptive unstructured meshes. Technical Report NAS-97-020, NASA Ames Research Center, Moffett Field, CA, 1997.

[9] Kirk Schloegel, George Karypis, and Vipin Kumar. Parallel multilevel diffusion algorithms for repartitioning of adaptive meshes. Technical Report TR 97-014, University of Minnesota, Department of Computer Science, 1997. http://www.cs.umn.edu/~karypis.

[10] Kirk Schloegel, George Karypis, and Vipin Kumar. Multilevel diffusion schemes for repartitioning of adaptive meshes. *Journal of Parallel and Distributed Computing*, 47(2):109–124, Dec 15, 1997. http://www.cs.umn.edu/~karypis.

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

[12] C. Walshaw, M. Cross, S. Johnson, and M. G. Everett. Jostle: Partitioning of unstructured meshes for massively parallel machines. *Proc. Parallel CFD'94, Kyoto*, 1994.