Parallel Multilevel Algorithms for Multi-constraint Graph Partitioning*

Kirk Schloegel, George Karypis, and Vipin Kumar

Army HPC Research Center
Department of Computer Science and Engineering
University of Minnesota
Minneapolis, MN 55455
(kirk, karypis, kumar)@cs.umn.edu

Abstract. Sequential multi-constraint graph partitioners have been developed to address the load balancing requirements of multi-phase simulations. The efficient execution of large multi-phase simulations on high performance parallel computers requires that the multi-constraint partitionings are computed in parallel. This paper presents a parallel formulation of a recently developed multi-constraint graph partitioning algorithm. We describe this algorithm and give experimental results conducted on a 128-processor Cray T3E. We show that our parallel algorithm is able to efficiently compute partitionings of similar edge-cuts as serial multi-constraint algorithms, and can scale to very large graphs. Our parallel multi-constraint graph partitioner is able to compute a three-constraint 128-way partitioning of a 7.5 million node graph in about 7 seconds on 128 processors of a Cray T3E.

1 Introduction

Algorithms that find good partitionings of highly unstructured and irregular graphs are critical for the efficient execution of scientific simulations on high performance parallel computers. In these simulations, computation is performed iteratively on each element of a physical (2D or 3D) mesh, and then some information is exchanged between adjacent mesh elements. Efficient execution of these simulations requires a mapping of the computational mesh to the processors such that each processor gets a roughly equal number of elements and the amount of inter-processor communication required to exchange the information

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

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 296–310, 2000.

between adjacent mesh elements is minimized. This mapping is commonly found using a traditional graph partitioning algorithm. Even though the problem of graph partitioning is NP-complete, multilevel schemes [3, 7, 11, 12] have been developed that are able to quickly find excellent partitionings of graphs that correspond to the 2D or 3D irregular meshes used for scientific simulations.

Despite the success that multilevel graph partitioners have enjoyed, for many important classes of scientific simulations, the formulation of the traditional graph partitioning problem is inadequate. For example, in multi-phase simulations such as particle-in-mesh simulations, crash-worthiness testing, and combustion engine simulations, there exists synchronization steps between the different phases of the computation. The existence of these requires that each phase be individually load balanced. That is, it is not sufficient to simply sum up the relative times required for each phase and to compute a decomposition based on this sum. Doing so may lead to some processors having too much work during one phase of the computation (and so, these may still be working after other processors are idle), and not enough work during another. Instead, it is critical that every processor have an equal amount of work from each of the phases of the computation. In general, multi-phase simulations require the partitioning to satisfy not just one, but a number of balance constraints equal to the number of computational phases.

Traditional graph partitioning techniques have been designed to balance a single computational phase only. An extension of the graph partitioning problem that can balance multiple phases is to assign a weight vector of size m to each vertex. The problem then becomes that of finding a partitioning that minimizes the total weight of the edges that are cut by the partitioning (i.e., the *edge-cut*) subject to the constraints that each of the m weights are balanced across the subdomains. Such a *multi-constraint* graph partitioning formulation as well as serial algorithms for computing multi-constraint partitionings are presented in [6].

It is desirable to compute multi-constraint partitionings in parallel for a number of reasons. Computational meshes in parallel scientific simulations are often too large to fit in the memory of one processor. A parallel partitioner can take advantage of the increased memory capacity of parallel machines. Thus, an effective parallel multi-constraint graph partitioner is key to the efficient execution of large multi-phase problems. Furthermore, in adaptive computations, the mesh needs to be partitioned frequently as the simulation progresses. In such computations, downloading the mesh to a single processor for repartitioning can become a major bottleneck.

The multi-constraint partitioning algorithm in [6] can be parallelized using the techniques presented in the parallel formulation of the single-constraint partitioning algorithm [8] as both are based on the multilevel paradigm. This paradigm consists of three phases: coarsening, initial partitioning, and multilevel refinement. In the coarsening phase, the original graph is successively coarsened down until it has only a small number of vertices. In the initial partitioning phase, a partitioning of the coarsest graph is computed. In the multilevel refine-

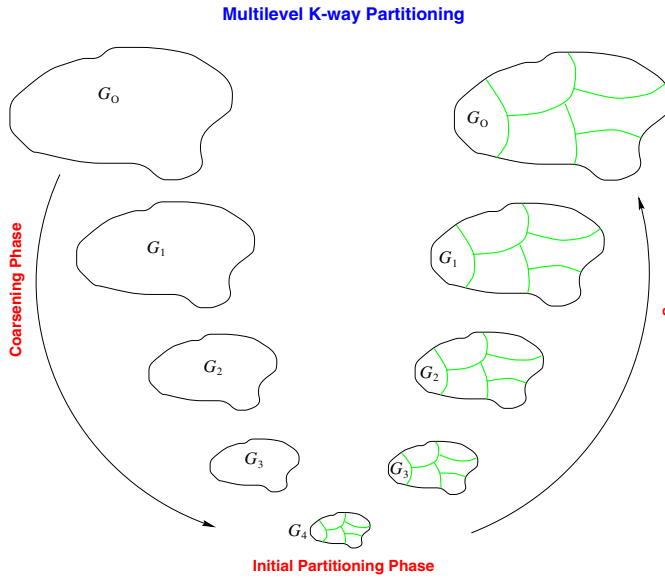

Fig. 1. The three phases of multilevel k-way graph partitioning. During the coarsening phase, the size of the graph is successively decreased. During the initial partitioning phase, a k-way partitioning is computed, During the uncoarsening/refinement phase, the partitioning is successively refined as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

ment phase, the initial partitioning is successively refined using a Kernighan-Lin (KL) type heuristic [10] as it is projected back to the original graph. Figure 1 illustrates the multilevel paradigm. Of these phases, it is straightforward to extend the parallel formulations of coarsening and initial partitioning to the context of multi-constraint partitioning. The key challenge is the parallel formulation of the refinement phase. The refinement phase for single-constraint partitioners is parallelized by relaxing the KL heuristic to the extent that the refinement can be performed in parallel while remaining effective. This relaxation can cause the partition to become unbalanced during the refinement process, but the imbalances are quickly corrected in succeeding iterations. Eventually, a balanced partitioning is obtained at the finest (i. e., the input) graph. Similar relaxation does not work for multi-constraint partitioning because it is non-trivial to correct load imbalances when more than one constraint is involved. A better approach is to avoid situations in which partitionings becomes imbalanced. This can be accomplished by either serializing the refinement algorithm, or else by restricting the amount of refinement that a processor is able to perform. The first will reduce the scalability of the algorithm and the second will result in low quality partitionings. Neither of these is desirable. Hence, the challenge in developing a parallel multi-constraint graph partitioner lies in developing a relaxation of the

refinement algorithm that is concurrent, effective, and maintains load balance for each constraint.

This paper describes a parallel multi-constraint refinement algorithm that is the key component of a parallel multi-constraint graph partitioner. We give experimental results of the full graph partitioning algorithm conducted on a 128-processor Cray T3E. We show that our parallel algorithm is able to compute balanced partitionings that have similar edge-cuts as those produced by the serial multi-constraint algorithm, while also being fast and scalable to very large graphs.

2 Parallel Multi-constraint Refinement

The main challenge in developing a parallel multi-constraint graph partitioner proved to be in developing a parallel multilevel refinement algorithm. This algorithm needs to meet the following criteria.

1. It must maintain the balance of all constraints.
2. It must maximize the possibility of refinement moves.
3. It must be scalable.

We briefly explain why developing an algorithm to meet all three of these criteria is challenging in the context of multiple constraints, and then describe our parallel multilevel refinement algorithm.

In order to guarantee that partition balance is maintained during parallel refinement, it is necessary to update global subdomain weights after every vertex migration. Such a scheme is much too serial in nature to be performed efficiently in parallel. For this reason, parallel single-constraint partitioning algorithms allow a number of vertex moves to occur concurrently before an update step is performed. One of the implications of concurrent refinement moves is that the balance constraint can be violated during refinement iterations. This is because if a subdomain can hold a certain amount of additional vertex weight without violating the balance constraint, then all of the processors assume that they can use all of this extra space for performing refinement moves. Of course, if just two processors move the amount of additional vertices that a subdomain can hold into it, then the subdomain will become overweight.

Parallel single-constraint graph partitioners address this challenge by encouraging subsequent refinement to restore the balance of the partitioning while improving its quality. For example, it is often sufficient to simply disallow further vertex moves into overweight subdomains and to perform another iteration of refinement. In general, the refinement process may not always be able to balance the partitioning while improving its quality in this way (although experience has shown that this usually works quite well). In this case, a few edge-cut increasing moves can be made to move vertices out of the overweight subdomains.

The real challenge is when we consider this phenomenon in the context of multiple balance constraints. This is because once a subdomain become overweight for a given constraint, it can be very difficult to balance the partitioning again.

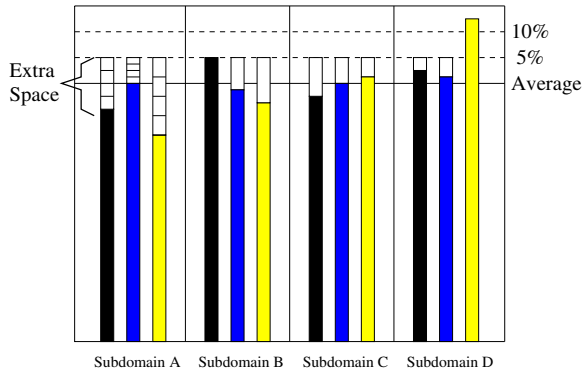

Fig. 2. This figure shows the subdomain weights for a 4-way partitioning of a 3-constraint graph. The white bars represent the extra space in a subdomain for each weight given a 5% user specified load imbalance tolerance.

Furthermore, the problem becomes more difficult as the number of constraints increases, as the multiple constraints are increasingly likely to interfere with each other during balancing. Given the difficulty of balancing multi-constraint partitionings, a better approach is to avoid situations in which the partitioning becomes imbalanced. Therefore, we would like to develop a multi-constraint refinement algorithm that can help to ensure that balance is maintained during parallel refinement.

One scheme that ensures that the balance is maintained during parallel refinement is to divide the amount of extra vertex weight that a subdomain can hold without becoming imbalanced by the number of processors. This then becomes the maximum vertex weight that any one processor is allowed to move into a particular subdomain in a single pass through the vertices. Consider the example illustrated in Figure 2. This shows the subdomain weights for a 4-way, 3-constraint partitioning. Lets assume that the user specified tolerance is 5%. The shaded bars represent the subdomain weights for each of the three constraints. The white bars represent the amount of weight that if added to the subdomain, would bring the bring the total weight to 5% above the average. In other words, the white bars show the amount of *extra space* each subdomain has for a particular weight given a 5% load imbalance tolerance. Figure 2 shows how the extra space in subdomain A can be split up for the four processors. If each processor is limited to moving the indicated amounts of weight into subdomain A, it is not possible for the 5% imbalance tolerance to be exceeded.

While this method guarantees that no subdomain (that is not overweight to start with) will become overweight beyond the imbalance tolerance, it is overly restrictive. This is because in general not all processors will need to use up their allocated space, while others may want to move more vertex weight into a subdomain than allowed by their slice. Furthermore, as the numbers of either processors or constraints increases, this effect increases. The reason is that as

the number of processors increases, the slices allocated to each processor get thinner. As the number of constraints increases, each additional constraint will also be sliced. This means that every vertex proposed for a move will be required to fit the slices of all of the constraints. For example, consider a three-constraint, ten-way partitioning computed on ten processors. If subdomain A can hold 20 units of the first weight, 30 units of the second weight, and 10 units of the third weight, then every processor must ensure that the sum of the weight vectors of all of the vertices that it moves into subdomain A is less than (2, 3, 1). It could very easily be the case then that this restriction is too severe to allow any one processor to perform their desired refinement.

It is possible to allocate the extra space of the subdomains more intelligently than simply giving each processor an equal share. We have investigated schemes that make the allocations based on a number of factors, such as the potential edge-cut improvements of the border vertices from a specific processor to a specific subdomain, the weights of these border vertices, and the total number of border vertices on each processor. While these schemes allow a greater number of refinement moves to be made than the straightforward scheme, they still restrict more edge-cut reducing moves than the serial algorithm. Our experiments have shown that these schemes produce partitionings that are up to 50% worse in quality than the serial multi-constraint algorithm. (Note, these results are not presented in this paper.)

Our Parallel Multi-constraint Refinement Algorithm. We have developed a parallel multi-constraint refinement algorithm that is no more restrictive than the serial algorithm with respect to the number of refinement moves that it allows, while also helping to ensure that none of the constraints become overly imbalanced. In the multilevel context, this algorithm is just as effective in improving the edge-cuts of partitionings as the serial algorithm.

This algorithm (essentially a reservation scheme) performs an additional pass through the vertices on every refinement iteration. In the first pass, refinement moves are made concurrently (as normal), however, only temporary data structures are updated. Next, a global reduction operation is performed to determine whether or not the balance constraints will be violated if these moves commit. If none of the balance constraints are violated, the moves are committed as normal. Otherwise, each processor is required to disallow a portion[1] of its proposed vertex moves into those subdomains that would be overweight if all of the moves are allowed to commit. The specific moves to be disallowed are selected randomly by each processor. While selecting moves randomly can negatively impact the edge-cut, this is usually not a problem because further refinement can easily correct the effects of any poor selections that happen to be made. Except for these modifications, our multi-constraint refinement algorithm is similar to the coarse-grain refinement algorithm described in [4].

[1] This portion is equal to one minus the amount of extra space in the subdomain divided by the total weight of all of the proposed moves into the subdomain.

It is important to note that the above scheme does not guarantee that the balance constraints will be maintained. This is because when we disallow a number of vertex moves, the weights of the subdomains from which these vertices were to have moved become higher than the weights that had been computed with the global reduction operation. It is therefore possible for some of these subdomains to become overweight. To correct this situation, a second global reduction operation can be performed followed by another round in which a number of the (remaining) proposed vertex moves are disallowed. These corrections might then lead to other imbalances, whose corrections might lead to others, and so on. We can easily allow this process to iterate until it converges (or until all of the proposed moves have been disallowed). Instead, we have chosen to simply ignores this problem. This is because the number of disallowed moves is a very small fraction of the total number of vertex moves, and so, any imbalance that is brought about by them is quite modest. Our experimental results show that the amount of imbalance introduced in this way is small enough that further refinement is able to correct it. In fact, as long as the amount of imbalance introduced is correctable, such a scheme can potentially result in higher quality partitionings compared to schemes that explore the feasible solution space only. (See the discussion in Section 3.)

Scalability Analysis. The scalability analysis of a parallel multilevel (single-constraint) graph partitioner is presented in [8]. This analysis assumes that (i) each vertex in the graph has a small bounded degree, (ii) this property is also satisfied by the successive coarser graphs, and (iii) the number of nodes in successive coarser graphs decreases by a factor of $1 + \epsilon$, where $0 < \epsilon \leq 1$. (Note, these assumptions hold true for all graphs that correspond to well-shaped finite element meshes.) Under these assumptions, the parallel run time of the single-constraint algorithm is

$$T_{par} = O\left(\frac{n}{p}\right) + O(p \log n) \tag{1}$$

and the isoefficiency function is $O(p^2 \log p)$, where n is the number of vertices and p is the number of processors. The parallel run time of our multi-constraint graph partitioner is similar (given the two assumptions). However, during both graph coarsening and multilevel refinement, all m weights must be considered. Therefore, the parallel run time of the multi-constraint algorithm is m times longer, or

$$T_{par} = O\left(\frac{nm}{p}\right) + O(pm \log n). \tag{2}$$

Since the sequential complexity of the serial multi-constraint algorithm is $O(nm)$, the isoefficiency function of the multi-constraint partitioner is also $O(p^2 \log p)$.

3 Experimental Results

In this section, we present experimental results of our parallel multi-constraint k-way graph partitioning algorithm on 32, 64, and 128 processors of a Cray T3E.

We constructed two sets of test problems to evaluate the effectiveness of our parallel partitioning algorithm in computing high-quality, balanced partitionings quickly. Both sets of problems were generated synthetically from the four graphs described in Table 1.

The purpose of the first set of problems is to test the ability of the multi-constraint partitioner to compute a balanced k-way partitioning for some relatively hard problems. From each of the four input graphs we generated graphs with two, three, four, and five weights, respectively. For each graph, the weights of the vertices were generated as follows. First, we computed a 16-way partitioning of the graph and then we assigned the same weight vector to all of the vertices in each one of these 16 subdomains. The weight vector for each subdomain was generated randomly, such that each vector contains m (for $m = 2, 3, 4, 5$) random numbers ranging from 0 to 19. Note that if we do not compute a 16-way partitioning, but instead simply assign randomly generated weights to each of the vertices, then the problem reduces to that of a single-constraint partitioning problem. The reason is that due to the random distribution of vertex weights, if we select any l vertices, the sum of their weight vectors will be around (lr, lr, ..., lr) where r is the expected average value of the random distribution. So the weight vector sums of any two sets of l vertices will tend to be similar regardless of the number of constraints. Thus, all we need to do to balance m constraints is to ensure that the subdomains contain a roughly equal number of vertices. This is the formulation for the single-constraint partitioning problem. Requiring that all of the vertices within a subdomain have the same weight vector avoids this effect. It also better models many applications. For example, in multi-phase problems, different regions of the mesh are active during different phases of the computation. However, those mesh elements that are active in the same phase typically form groups of contiguous regions and are not distributed randomly throughout the mesh. Therefore, each of the 16 subdomains in the first problem set models a contiguous region of mesh elements.

The purpose of the second set of problems is to test the performance of the multi-constraint partitioner in the context of multi-phase computations in which different (possibly overlapping) subsets of nodes participate in different phases. For each of the four graphs, we again generated graphs with two, three, four, and five weights corresponding to a two-, three-, four-, and five-phase computation, respectively. In the case of the five-phase graph, the portion of the graph that is active (i.e., performing computations) is 100%, 75%, 50%, 50%, and 25% of the subdomains. In the four-phase case, this is 100%, 75%, 50%, and 50%. In the three- and two-phase cases, it is 100%, 75%, and 50% and 100% and 75%, respectively. The portions of the graph that are active was determined as follows. First, we computed a 32-way partitioning of the graph and then we randomly selected a subset of these subdomains according to the overall active percentage. For instance, to determine the portion of the graph that is active during the second phase, we randomly selected 24 out of these 32 subdomains (i.e., 75%). The weight vectors associated with each vertex depends on the phases in which it is active. For instance, in the case of the five-phase computation, if a vertex

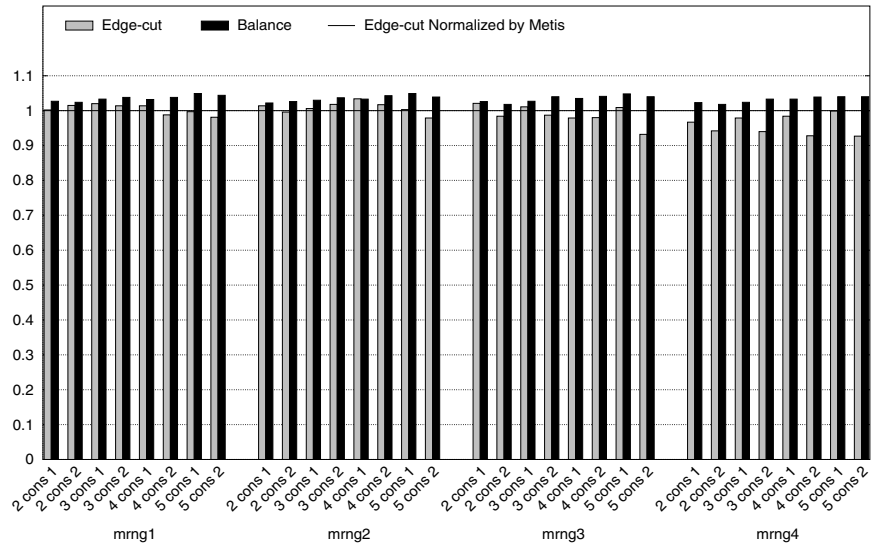

Fig. 3. This figure shows the edge-cut and balance results from the parallel multi-constraint algorithm on 32 processors. The edge-cut results are normalized by the results obtained from the serial multi-constraint algorithm implemented in METIS.

is active only during the first, second, and fifth phase, its weight vector will be (1, 1, 0, 0, 1). In generating these test problems we also assigned weight to the edges to better reflect the overall communication volume of the underlying multi-phase computation. In particular, the weight of an edge (v, u) was set to the number of phases that both vertices v and u are active at the same time. This is an accurate model of the overall information exchange between vertices since during each phase, vertices access each other's data only if both are active.

Graph	Num of Verts	Num of Edges
*mrng*1	257,000	1,010,096
*mrng*2	1,017,253	4,031,428
*mrng*3	4,039,160	16,033,696
*mrng*4	7,533,224	29,982,560

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

Comparison of Serial and Parallel Multi-constraint Algorithms. Figures 3, 4, and 5 compare the edge-cuts of the partitionings produced by our parallel multi-constraint graph partitioning algorithm with those produced by the serial multi-

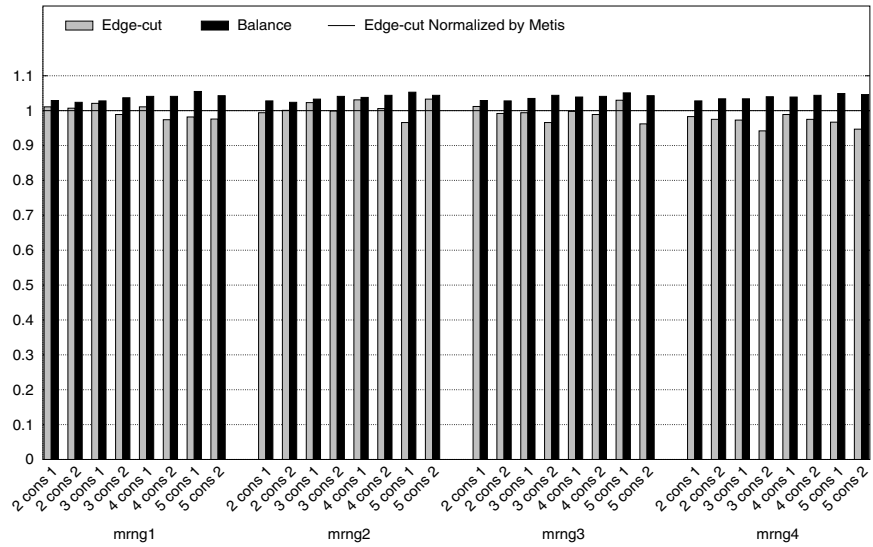

Fig. 4. This figure shows the edge-cut and balance results from the parallel multi-constraint algorithm on 64 processors. The edge-cut results are normalized by the results obtained from the serial multi-constraint algorithm implemented in METIS.

constraint algorithm [6], and give the maximum load imbalance of the partitionings produced by our algorithm. Each figure shows four sets of results, one for each of the four graphs described in Table 1. Each set is composed of two-, three-, four-, and five-constraint Type 1 and 2 problems. These are labeled "m cons t" where m indicates the number of constraints and t indicates the type of problem (i.e., Type 1 or 2). So the results labeled "2 cons 1" indicates the edge-cut and balance results for a two-constraint Type 1 problem. The edge-cut results shown are those obtained by our parallel algorithm normalized by those obtained by the serial algorithm. Therefore, a bar below the 1.0 index line indicates that our parallel algorithm produced partitionings with better edge-cuts than the serial algorithm. The balance results indicate the maximum imbalance of all of the constraints. (Here, imbalance is defined as the maximum subdomain weight divided by the average subdomain weight for a given constraint.) These results are not normalized. Note that we set an imbalance tolerance of 5% for all of the constraints. The results given in Figures 3, 4, and 5 give the arithmetic means of three runs by our algorithm utilizing different random seeds each time. Note that in every case, the results of each individual run were within a few percent of the averaged results shown. For each figure, the number of subdomains computed is equal to the number of processors.

Figures 3, 4, and 5 show that our parallel multi-constraint graph partitioning algorithm is able to compute partitionings with similar or better edge-cuts

compared to the serial multi-constraint graph partitioner, while ensuring that multiple constraints are balanced.

Notice that the parallel algorithm is sometimes able to produce partitionings with better edge-cuts than the serial algorithm. There are two reasons for this. First, the parallel formulation of the matching scheme used (heavy-edge matching using the balanced-edge heuristic as a tie-breaker [6]) is not as effective in finding a maximal matching as the serial formulation. (This is due to the protocol that is used to arbitrate between conflicting matching requests made in parallel [4].) Therefore, a smaller number of vertices match together with the parallel algorithm than with the serial algorithm. The result is that the newly computed coarsened graph tends to be larger for the parallel algorithm than for the serial algorithm, and so, the parallel algorithm takes more coarsening levels to obtain a sufficiently small graph. The effect of this is that the matching algorithm usually has one or more additional coarsening levels in which to remove exposed edge weight (i. e., the total weight of the edges on the graph). By the time the parallel algorithm computes the coarsest graph, it can have significantly less exposed edge weight than the coarsest graph computed by the serial algorithm. This makes it easier for the initial partitioning algorithm to compute higher-quality partitionings. During multilevel refinement, some of this advantage is maintained, and so, the final partitioning can be better than those computed by the serial algorithm. The disadvantage of slow coarsening is that the additional coarsening and refinement levels take time, and so, the execution time of the algorithm is increased. This phenomenon of slow coarsening was also observed in the context of hypergraphs in [1].

The second reason is that in the serial algorithm, once the partitioning becomes balanced it will never explore the infeasible solution space in order to improve the edge-cut. Since the parallel refinement algorithm does not guarantee to maintain partition balance, the parallel graph partitioner may do so. This usually happens on the coarse graphs. Here, the granularity of the vertices makes it more likely that the parallel multi-constraint refinement algorithm will result in slightly imbalanced partitionings. Essentially, the parallel multi-constraint refinement algorithm it too aggressive in reducing the edge-cut here, and so, makes too many refinement moves. This is a poor strategy if the partitioning becomes so imbalanced that subsequent refinement is not able to restore the balance. However, since our parallel refinement algorithm helps to ensure that the amount of imbalance introduced is small, subsequent refinement is able to restore the partition balance while further improving its edge-cut.

Run Time Results. Table 2 compares the run times of the parallel multi-constraint graph partitioning algorithm with the serial multi-constraint algorithm implemented in the MÉTIS library [5] for *mrng*1. These results show only modest speedups for the parallel partitioner. The reason is that the graph *mrng*1 is quite small, and so, the communication and parallel overheads are significant. However, we use *mrng*1 because it is the only one of the test graphs that is small enough to run serially on a single processor of the Cray T3E.

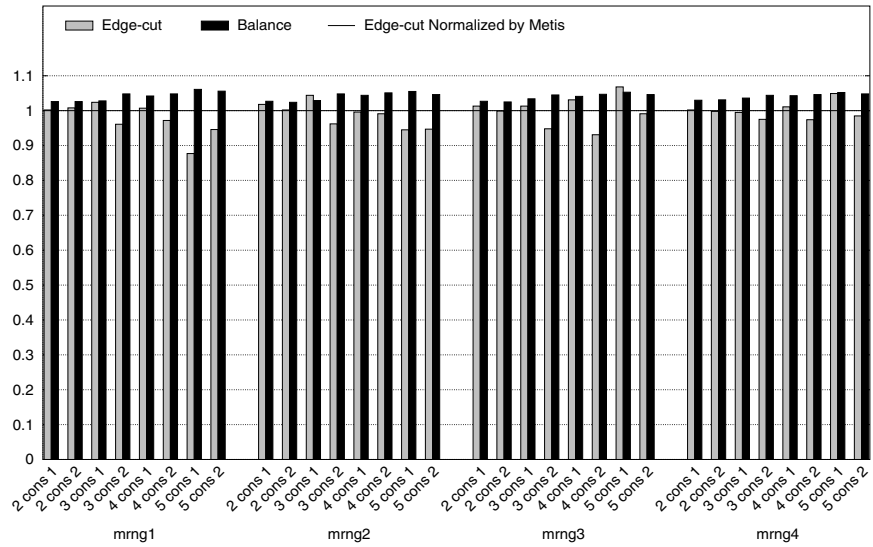

Fig. 5. This figure shows the edge-cut and balance results from the parallel multi-constraint algorithm on 128 processors. The edge-cut results are normalized by the results obtained from the serial multi-constraint algorithm implemented in METIS.

Table 3 gives selected run time results and efficiencies of our parallel multi-constraint graph partitioning algorithm on up to 128 processors. Table 3 shows that our algorithm is very fast, as it is able to compute a three-constraint 128-way partitioning of a 7.5 million node graph in about 7 seconds on 128 processors of a Cray T3E. It also shows that our parallel algorithm obtains similar run times as you double (or quadruple) both the size of problem and the number of processors. For example, the time required to partition $mrng2$ (with 1 million vertices) on eight processors is similar to that of partitioning $mrng3$ (4 million vertices) on 32 processors and $mrng4$ (7.5 million vertices) on 64 processors.

k	serial time	parallel time
2	7.3	6.4
4	7.5	4.4
8	8.0	2.5
16	8.3	1.7

Table 2. Serial and parallel run times of the multi-constraint graph partitioner for a three-constraint problem on $mrng1$.

Graph	8-processors		16-processors		32-processors		64-processors		128-processors	
	time	efficiency	time	efficiency	time	efficiency	time	efficiency	time	efficiency
*mrng*2	9.8	100%	5.3	92%	3.5	70%	2.5	49%	3.1	20%
*mrng*3	31.8	100%	16.9	94%	9.3	85%	5.7	70%	4.4	45%
*mrng*4	out of mem.		30.7	100%	16.7	92%	9.2	83%	6.4	60%

Table 3. Parallel run times and efficiencies of our multi-constraint graph partitioner on three-constraint type 1 problems.

Graph	8-processors	16-processors	32-processors	64-processors	128-processors
*mrng*2	5.4	3.1	2.1	1.5	1.7
*mrng*3	15.8	8.8	4.8	3.0	2.7
*mrng*4	38.6	16.2	8.8	5.0	3.6

Table 4. Parallel run times of the single-constraint graph partitioner implemented in ParMETIS.

Table 4 gives the run times of the k-way single-constraint parallel graph partitioning algorithm implemented in the ParMETIS library [9] on the same graphs used for our experiments. Comparing Tables 3 and 4 shows that computing a three-constraint partitioning takes about twice as long as computing a single-constraint partitioning. For example, it takes 9.3 seconds to compute a three-constraint partitioning and 4.8 seconds to compute a single-constraint partitioning for *mrng*3 on 32 processors. Also, comparing the speedups indicates that the multi-constraint algorithm is slightly more scalable than the single-constraint algorithm. For example, the speedup from 16 to 128 processors for *mrng*3 is 3.84 for the multi-constraint algorithm and 3.26 for the single-constraint algorithm. The reason is that the multi-constraint algorithm is more computationally intensive than the single-constraint algorithm, as multiple (not single) weights must be computed regularly.

Parallel Efficiency. Table 3 gives selected parallel efficiencies of our parallel multi-constraint graph partitioning algorithm on up to 128 processors. The efficiencies are computed with respect to the smallest number of processors shown. Therefore, for *mrng*2 and *mrng*3, we set the efficiency of eight processors to 100%, while we set the efficiency of 16 processors to 100% for *mrng*4. The parallel multi-constraint graph partitioner obtained efficiencies between 20% and 94%. The efficiencies were good (between 90% - 70%) when the graph was sufficiently large with respect to the number of processors. However, these dropped off for the smaller graphs on large number of processors. The isoefficiency of the parallel multi-constraint graph partitioner is $O(p^2 \log p)$. Therefore, in order to

maintain a constant efficiency when doubling the number of processors, we need to increase the size of the data by a little more than four times. Since *mrng*3 is approximately four times as large as *mrng*2 we can test the isoefficiency function experimentally. The efficiency of the multi-constraint partitioner with 32 processors for *mrng*2 is 70%. Doubling the number of processors to 64 and increasing the data size by four times (64-processors on *mrng*3) yields a similar efficiency. This is better than expected, as the isoefficiency function predicts that we need in increase the size of the data set by more than four times to obtain the same efficiency. If we examine the results of 64 processors on *mrng*2 and 128 processors on *mrng*3 we see a slightly decreasing efficiency of 49% to 45%. This is what we would expect based on the isoefficiency function. If we examine the results of 16 processors on *mrng*2 and 32 processors on *mrng*3 we see that the drop in efficiency is larger (92% to 85%). So here we get a slightly worse efficiency than expected. These experimental results are quite consistent with the isoefficiency function of the algorithm. The slight deviations can be attributed to the fact that the number of refinement iterations on each graph is upper bounded. However, if a local minima is reached prior to this upper bound, then no further iterations will be performed on this graph. Therefore, while the upper bound on the amount of work done by the algorithm is the same for all of the experiments, the actual amount of work done can be slightly different depending on the refinement process.

4 Conclusions

This paper has presented a parallel formulation of the multi-constraint graph partitioning algorithm for partitioning 2D and 3D irregular and unstructured meshes used in scientific simulations. This algorithm is essentially as scalable as the widely used parallel formulation of the single-constraint graph partitioning algorithm [8]. Experimental results conducted on a 128-processor Cray T3E show that our parallel algorithm is able to compute balanced partitionings with similar edge-cuts as the serial algorithm. We have shown that the run time of our algorithm is very fast. Our parallel multi-constraint graph partitioner is able to compute a three-constraint 128-way partitioning of a 7.5 million node graph in about 7 seconds on 128 processors of a Cray T3E.

Although the experiments presented in this paper are all conducted on synthetic graphs, our parallel multi-constraint partitioning algorithm has also been tested on real application graphs. Basermann et al. [2] have used the parallel multi-constraint graph partitioner described in this paper for load balancing multi-phase car crash simulations of an Audi and a BMW in frontal impacts with a wall. These results are consistent with the run time, edge-cut, and balance results presented in Section 3.

While the experimental results presented in Section 3 (and [2]) are quite good, it is important to note that the effectiveness of the algorithm depends on two things. First, it is critical that a relatively balanced partitioning be computed during the initial partitioning phase. This is because if the partitioning starts out

imbalanced, there is no guarantee that it will ever become balanced during the course of multilevel refinement. Our experiments (not presented in this paper) have shown that an initial partitioning that is more than 20% imbalanced for one or more constraints is unlikely to be improved during multilevel refinement. Second, as is the case for the serial multi-constraint algorithm, the quality of the final partitioning is largely dependent on the availability of vertices that can be swapped across subdomains in order to reduce the edge-cut, while maintaining all of the balance constraints. Experimentation has shown that for a small number of constraints (i.e., two to four) there is a good availability of such vertices, and so, the quality of the computed partitionings is good. However, as the number of constraints increases further, the number of vertices that can be moved while maintaining all of the balance constraints decreases. Therefore, the quality of the produced partitionings can drop off dramatically.

References

[1] C. Alpert, J. Huang, and A. Kahng. Multilevel circuit partitioning. In *Proc. of the 34th ACM/IEEE Design Automation Conference*, 1997.

[2] A. Basermann, J. Fingberg, G. Lonsdale, B. Maerten, and C. Walshaw. Dynamic multi-partitioning for parallel finite element applications. *Submitted to ParCo '99*, 1999.

[3] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. *Proceedings Supercomputing '95*, 1995.

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

[5] G. Karypis and V. Kumar. MéTiS: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices, version 4.0. Technical report, Univ. of MN, Dept. of Computer Sci. and Engr., 1998.

[6] G. Karypis and V. Kumar. Multilevel algorithms for multi-constraint graph partitioning. In *Proceedings of Supercomputing '98*, 1998.

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

[8] G. Karypis and V. Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. *Siam Review*, 41(2):278–300, 1999.

[9] G. Karypis, K. Schloegel, and V. Kumar. ParMéTiS: Parallel graph partitioning and sparse matrix ordering library. Technical report, Univ. of MN, Dept. of Computer Sci. and Engr., 1997.

[10] B. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49(2):291–307, 1970.

[11] B. Monien, R. Preis, and R. Diekmann. Quality matching and local improvement for multilevel graph-partitioning. Technical report, University of Paderborn, 1999.

[12] C. Walshaw and M. Cross. Parallel optimisation algorithms for multilevel mesh partitioning. Technical Report 99/IM/44, University of Greenwich, UK, 1999.