Parallel Multilevel Algorithms for Multi-Constraint Graph Partitioning*

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

Technical Report: TR 99-031

September 30, 1999

Abstract

Recently, sequential multi-constraint graph partitioning algorithms have been developed to address the load balancing requirements of emerging multi-phase and multi-physics scientific simulation problems. Effective execution of such 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 quality to 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 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

*This work was supported by DOE contract number LLNL B347881, by Army Research Office contracts DA/DAAG55-98-1-0441 and DA/DAAH04-95-1-0244, by Army High Performance Computing Research Center cooperative agreement number DAAH04-95-2-0003/contract number DAAH04-95-C-0008, the content of which does not necessarily reflect the position or the policy of the government, and no official endorsement should be inferred. Additional support was provided by the IBM Partnership Award, and by the IBM SUR equipment grant. Access to computing facilities was provided by AHPCRC, Minnesota Supercomputer Institute. Related papers are available via WWW at URL: http://www-users.cs.umn.edu/~karypis

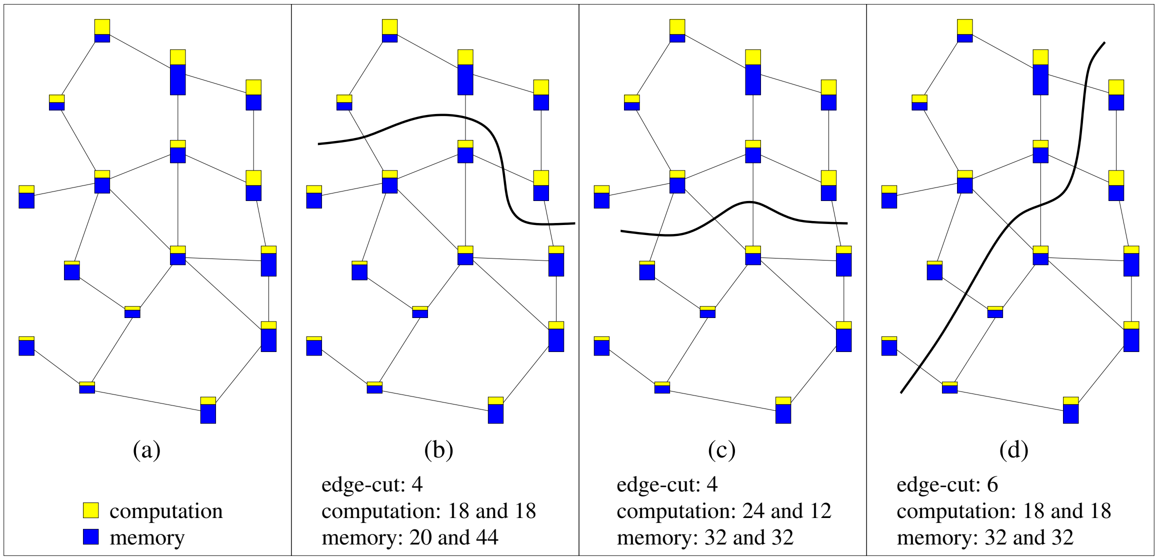

Figure 1: An example of a computation with non-uniform memory requirements. Each vertex in the graph is split into two colors. The size of the yellow portion represents the amount of computation associated with the vertex, while the size of the blue portion represents the amount of memory requirement work associated with the vertex. The partitioning in (b) balances the computation. The partitioning in (c) balances the memory, but only the partitioning in (d) balances both of these.

required to exchange the information between connected mesh elements is minimized. This mapping is commonly found using the traditional graph partitioning problem that computes a partitioning of the graph such that each subdomain has an equal number of vertices and the number of edges cut by the partitioning is minimized. Even though the problem of graph partitioning is NP-complete, multilevel schemes [3, 7] are known to quickly find excellent partitionings of graphs that correspond to the 2D or 3D irregular meshes used for scientific simulations.

In recent years, however, the complexity and fidelity of the models used in scientific simulations have substantially increased to the point that the traditional graph partitioning formulation has become inadequate. For example, in multi-physics computations, a variety of materials and/or processes are simulated together. The result is a class of problems in which the computation as well as the memory requirements are not uniform across the mesh. Existing partitioning schemes can be used to divide the mesh among the processors such that either the amount of computation or the amount of memory required is balanced across the processors. However, they do not allow us to compute a partitioning that simultaneously balances both of these quantities. Our inability to do so can either lead to significant computational imbalances, limiting the overall efficiency, or significant memory imbalances, limiting the size of the problems that can be solved using parallel computers. Figure 1 illustrates this problem. It shows a graph in which the amount of computation and memory associated with a vertex can be different throughout the graph, and gives three possible partitionings of it. The partitioning in Figure 1(b) balances the computation among the subdomains, but creates a serious imbalance for memory requirements. The partitioning in Figure 1(c) balances the memory requirement, while leaving the computation imbalanced. The partitioning in Figure 1(d), that balances both of these, is the desirable solution. In general, multi-physics simulations require the partitioning to satisfy not just one, but a multiple number of balance constraints. (In this case the partitioning must balance two constraints, computation and memory). This requirement is also present in multi-phase computations in which different (possibly overlapping) subsets of nodes participate in different phases of the computation [2].

The common characteristic of these problems is that they all require the computation of partitionings that satisfy more than one balance constraint. Traditional graph partitioning techniques have been designed to balance only a single constraint (i.e., the vertex weight). An extension of the graph partitioning problem that allows us to balance multiple constraints is to assign a weight vector of size m to each vertex. The problem then becomes that of finding a partitioning with a minimal edge-cut, subject to the constraints that each of the m weights is balanced across the subdomains. Such a multi-constraint graph partitioning formulation as well as a serial algorithm for computing multi-constraint partitionings is 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. 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. Thus, an effective parallel multi-constraint graph partitioner is key to the efficient execution of large multi-phase and multi-physics problems.

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. (See Figure 2.) 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 refinement phase, the initial partitioning is successively refined using a Kernighan-Lin (KL) type heuristic [10] as it is being projected back to the original graph. 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 in single constraint partitioning algorithms 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 level 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. In fact, the challenge of balancing multi-constraint partitionings is so difficult that a better solution is to avoid situations in which the partitioning 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 presents a parallel multi-constraint graph partitioning algorithm based on the serial scheme presented in [6]. We describe this algorithm and give experimental results conducted on a 128-processor Cray T3E. We show that our parallel algorithm is able to compute balanced partitionings that are of similar quality to those produced by the serial multi-constraint algorithm, while also being scalable to very large graphs.

The remainder of this paper is organized as follows. Section 2 gives background, including de-

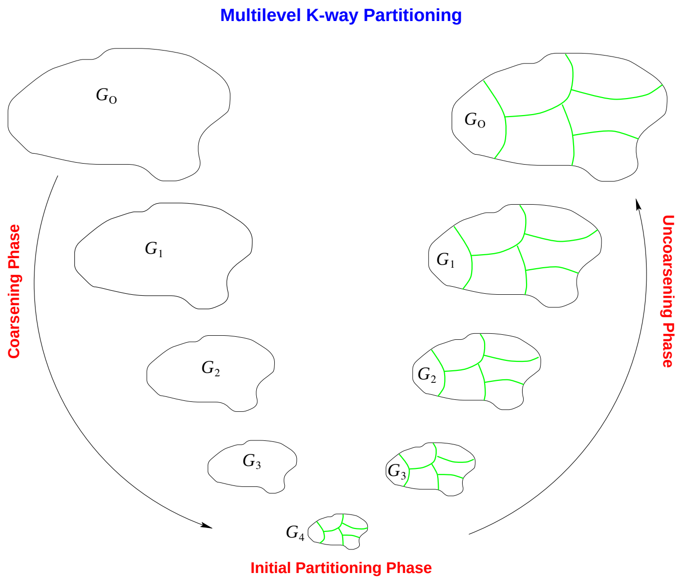

Figure 2: The three phases of multilevel k-way graph partitioning. During the coarsening phase, the size of the graph is successively decreased. During the initial partitioning phase, a k-way partitioning is computed, During the multilevel refinement phase, the partitioning is successively refined as 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.

scriptions of previous work in single and multi-constraint graph partitioning. Section 3 describes our parallel multi-constraint graph partitioning algorithm. Section 4 presents experimental results performed on 128 processors of a Cray T3E. Section 5 gives conclusions.

2 Background

Here we provide on overview of serial multi-constraint graph partitioning schemes and parallel single constraint graph partitioning schemes.

2.1 Multi-constraint Graph Partitioning

The multi-constraint graph partitioning problem can be formulated as follows. Consider a graph $G = (V, E)$, such that each vertex $v \in V$ has a weight vector w^v of size m associated with it, and each edge $e \in E$ has a scalar weight attached to it. There are no restrictions on the weights of the edges, but we will assume, without loss of generality, that the weight vectors of the vertices satisfy the property that $\sum_{\forall v \in V} w_i^v = 1.0$ for $i = 1, 2, \ldots, m$. If the vertex weights do not satisfy the above property, we can divide each w_i^v by $\sum_{\forall v \in V} w_i^v$ to ensure that the property is satisfied. Note that this normalization does not in any way limit our modeling ability. Let P be the partitioning vector of size $|V|$, such that for each vertex v, $P[v]$ stores the partition number to which v belongs. For any such k-way partitioning vector, the load imbalance l_i with respect to the ith weight of the k-way partitioning is defined as follows:

$$l_i = k \max_j \left(\sum_{\forall v : P[v] = j} w_i^v \right) \tag{1}$$

If the ith weight is perfectly balanced in the k-way partitioning, then $\sum_{\forall v:P[v]=j} w_i^v$ for all j is $1/k$, and $l_i = 1$. A load imbalance of $l_i = x$ indicates that a computation of size W performed on k processors during the ith phase takes xW/k time instead of W/p time needed in the case of perfect load balance, under the assumption of zero communication overhead. A load imbalance of $1 + \alpha$ indicates that the partitioning is load imbalanced by α%. The goal is to find a k-way partitioning P of G such that the sum of the weights of the edges that are cut by the partitioning is minimized subject to the constraint $\forall i,\ l_i \leq c_i$. Where c is a vector of size m such that $\forall i,\ c_i \geq 1.0$. The vector c is specified by the user and reflects the amount of load imbalance the user is willing to accept for each of the constraints.

A serial multi-constraint graph partitioning algorithm is presented in [6]. The coarsening phase of the algorithm is similar to the one used for the single constraint algorithm in [7]. After a sufficiently coarse graph is constructed, initial partitioning is performed on this graph by recursive bisection. The problem of computing a balanced multi-constraint bisection has been studied in [6]. The authors present a lemma that proves that a set of two-weight objects can be partitioned into two disjoint subsets such that the difference between either of the weights of the two sets is bounded by twice the maximum weight of any object. They further show that this bound can be generalized to m weights. However, in this case maintaining the weight bound depends on the presence of sufficiently many objects with certain weight characteristics. The lemma leads to an algorithm for computing such a bisection. This algorithm is the basis for the initial partitioning phase. In the multilevel refinement phase, a greedy k-way refinement algorithm operates as follows. The vertices along the boundary of the partitioning are visited in a random order. They are examined and moved to one of their adjacent subdomains if this move:

(a) improves the quality of the partitioning without violating the specified balance requirements for each constraint.

(b) improves the balance of the multiple constraints without worsening the partition quality.

A small number of such passes through the vertices is performed at each successively finer graph.

2.2 Parallel Single Constraint Graph Partitioning

Parallelizing a multilevel graph partitioner requires parallel algorithms for each of the three phases of the multilevel paradigm (i.e., graph coarsening, initial partitioning, and multilevel refinement). In this section, we briefly describe the key features of each of these algorithms.

Parallel Graph Coarsening. In the coarsening phase, all the processors collaborate to compute a matching of the vertices in parallel. In these schemes, each processor computes a matching for the vertices that it owns[1]. The challenge in performing this efficiently is resolving contentions for vertices that belong to different processors. For example, a vertex on one processor may want to match with a vertex that is owned by a different processor. However, the processor that owns the second vertex may want to match it with a different vertex. In the general case, resolving

[1]Parallel partitioners assume that a graph is originally distributed across the processors such that each processor has a roughly equal number of local vertices together with their associated adjacency information.

these conflicts requires an all-to-all communication between the processors. Existing schemes either subdivide the computation in such a way that conflicts do not occur [8] or else use efficient communication protocols to reduce the overhead [4, 11].

Parallel Initial Partitioning. In the initial partitioning phase, a partitioning of the coarsest graph is quickly computed. This can be performed by a task decomposition scheme [8]. Here, the coarsest graph is assembled on a single processor and then broadcast to all of the processors. Each processor then computes the same bisection of this graph concurrently. If k equals two, the initial partitioning is complete. Otherwise, the bisection is used to construct two subgraphs, where each subgraph contains the vertices from only one subdomain of the bisection. In this way, the coarsest graph is split evenly in two. Next, half of the processors compute a bisection of the first subgraph and the other half compute a bisection of the second subgraph. If k is greater than four, these bisections are used to split the two subgraphs in half and again half of the processors in each group compute a bisection of each of the newly constructed subgraphs. This recursive splitting continues until a k-way partitioning is computed.

Parallel Multilevel Refinement. Multilevel refinement can be performed in parallel by having each processor apply a local refinement algorithm on each level graph, starting at the coarsest graph and continuing through all of the finer graphs up to and including the input graph. The local refinement algorithm consists of a number of refinement iterations. Each iteration consists of a single pass through the vertices followed by an update and synchronization step. During a pass, all of the processors simultaneously visit their local vertices and determine if moving a vertex to an adjacent subdomain will increase the partition quality while maintaining the balance constraint or improve the balance while maintaining the quality. During the update step, the relevant processors are informed of vertex moves, and a global reduction operation is performed in order to compute (and broadcast) the new weights of the subdomains.

One of the challenges in performing refinement in this way is in ensuring that concurrent vertex moves do actually lead to an improvement in the partition quality. Figure 4 illustrates a case where this is not true. In the figure on the left, the movement of vertex q from subdomain A to subdomain B will result in a decrease in the edge-cut by three. Likewise, the movement of vertex r from subdomain B to subdomain A will result in a decrease in the edge-cut by two. However, if both moves are performed concurrently, the result is an increase in the edge-cut by five. The parallel refinement algorithms described in [4, 8, 11] address this problem by breaking up each refinement pass through the vertices into sub-phases. During each sub-phase, only certain subsets of vertices are allowed to move. These subsets are selected in such a way as to either eliminate [8, 11] or substantially limit [4] this phenomenon.

One possible method to implement parallel partition refinement is to require that each processor own all of the vertices from only a single subdomain [11]. Here, each processor represents a distinct subdomain. However, this approach is not efficient, as it makes it necessary to transfer all of the data associated with a vertex between processors when a vertex switches subdomains. In the multilevel context, this includes not only the data associated with the current graph, but also the data associated with all of the finer graphs up to the original graph. This data migration is not necessary when processors can own vertices from arbitrary subdomains. Instead, when a vertex swaps subdomains, an array entry can simply be updated. Since this fast update technique can translate into a significant performance gain [11], in this paper we focus on schemes do allow

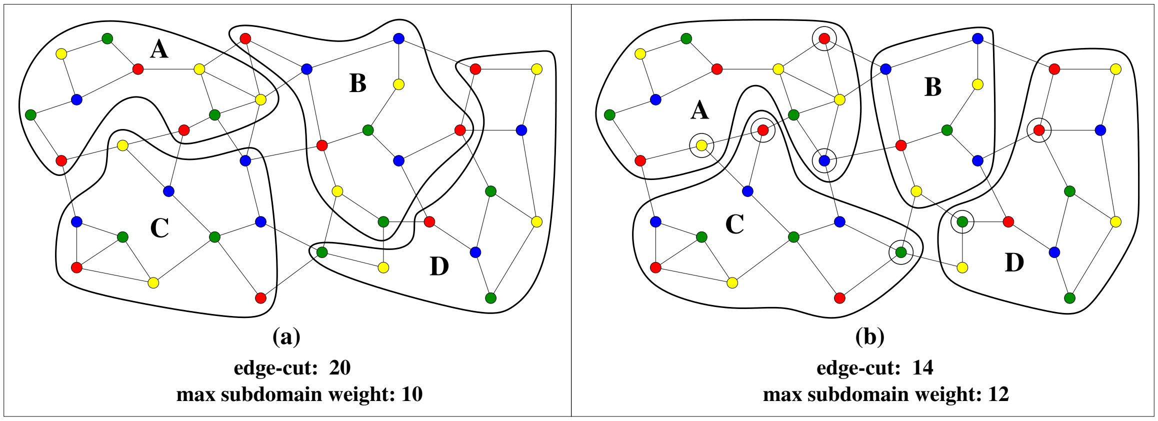

Figure 3: This figure shows the state of a partitioning prior to parallel refinement (a) and after a single refinement iteration (b). The circled vertices in (b) have changed subdomains.

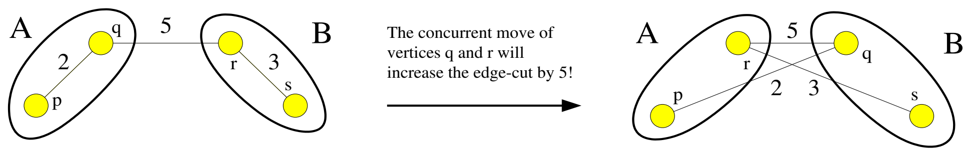

Figure 4: This figure shows that the concurrent movement of vertices q and r will result in an increase in the edge-cut by five, even though both would have decreased the edge-cut if moved alone.

processors to own vertices from arbitrary subdomains.

Figure 3 illustrates a single pass of the refinement algorithm. Figure 3(a) shows the state of a four-way partitioning of a graph immediately before parallel refinement. Here, the color of a vertex (red, yellow, green, or blue) indicates the processor to which it is local, while the curves surrounding the vertices indicate the subdomain to which it belongs (A, B, C, or D). There are 40 vertices in the graph. Each has a weight of one. Therefore, in order to perfectly balance the partitioning, each subdomain should be of weight 10. (Note, the weight of a subdomain is equal to the sum of the weights of its vertices.) In this example, let us assume that the user has specified 20% as acceptable imbalance. Therefore, subdomains of weight 12 are allowed. In Figure 3(a), the partitioning is perfectly balanced and 20 edges are cut by the partitioning. Figure 3(b) shows the state of the partitioning after each processor simultaneously performs one pass of refinement through its local vertices. Here, the circled vertices indicate those that were migrated. The edge-cut has been reduced to 14, and the maximum subdomain weight is now 12.

3 Parallel Multi-constraint Graph Partitioning

In this section, we present a parallel formulation of the multilevel algorithm for multi-constraint graph partitioning described in [6].

The parallel formulation of the graph coarsening phase computes a matching of the vertices in parallel utilizing the same heuristics as the serial multi-constraint algorithm (i.e., heavy-edge matching with a balance-edge tie-breaking scheme), and is implemented in the same way as the coarse-grain matching scheme described in [4]. The parallel formulation of the initial partitioning phase is a

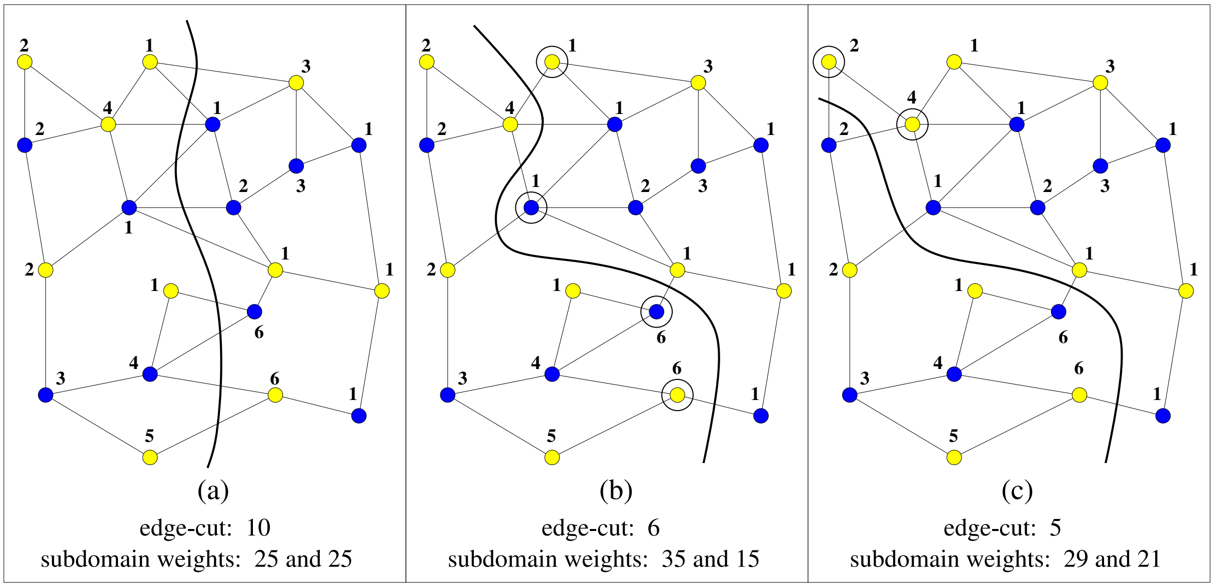

Figure 5: This figure shows a single constraint partitioning (a). During parallel refinement, the blue and yellow processors each move two vertices (b). This results in an imbalance in the partitioning. If we simply disallow vertices to move into the overweight domain, balance can be obtained easily (c).

modification of the task decomposition scheme described in [8] with each processor calling the multi-constraint bisection algorithm [6] implemented in METIS. In order to improve the quality and balance of the initial partitioning, we split the processors into four groups and have each group compute a different partitioning by the task decomposition method. We then select the best of these four partitionings to broadcast to all of processors.

Parallel Multi-constraint Multilevel 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

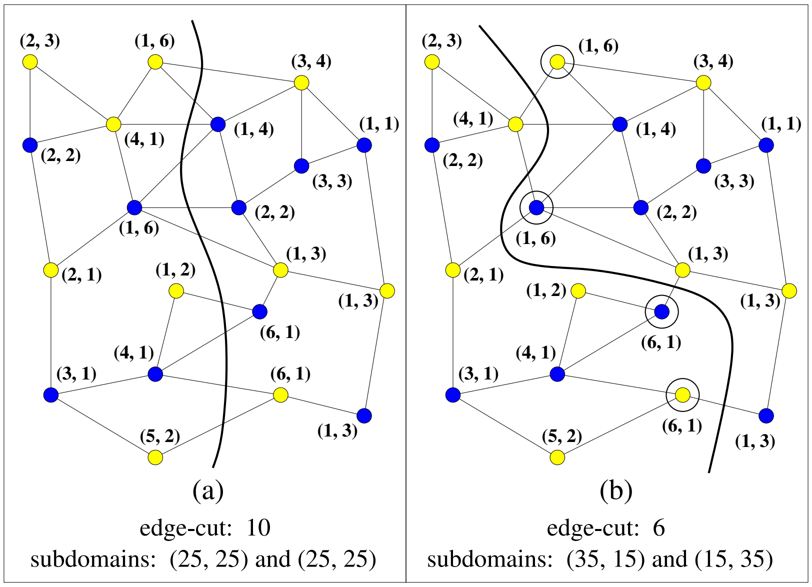

Figure 6: This figure shows a two-constraint partitioning (a). During parallel refinement, the blue and yellow processors each move two vertices (b). This results in an imbalance that is not easily corrected. Moving the same two vertices from Figure 5(c) will result in the second constraint becoming even more imbalanced.

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. This is illustrated by the example in Figure 5. Figure 5(a) shows a single constraint two-way partitioning of a graph performed on two processors. Each of the vertices has a single weight associated with it that is shown in the figure. The subdomain weights are 25 and 25. Therefore, the partitioning is perfectly balanced. Let the imbalance tolerance be 20%. Hence, subdomains can be of weight 30 or less. Figure 5(b) shows the state of the partitioning after parallel refinement is performed. Here, the yellow processor moved one vertex (circled) of weight one from the left to the right subdomain and one vertex (circled) of weight six from the right to the left subdomain. The yellow processor assumes that these moves will result in subdomain weights of 30 (left) and 20 (right), and so will maintain the balance constraint. Simultaneously, the blue processor moves one vertex (circled) of weight one from the left to the right subdomain and one vertex (circled) of weight six from the right to the left subdomain. The blue processor also believes that the balance constraint will be maintained. Together, however, these moves result in the left subdomain becoming overweight with a weight of 35 and the right subdomain becoming underweight with a weight of 15.

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. Doing so in this case results in the partitioning becoming balanced again (as illustrated in Figure 5(c)). 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. Figure 6 illustrates this point. It shows the graph from Figure 5, but now each vertex has two weights. In Figure 6(a), the subdomain weights are (25, 25) and (25, 25), and so the partitioning is perfectly balanced. Again, lets assume that the user specified imbalance tolerance is 20% for both constraints. Figure 6(b) shows the state of the partitioning after the same refinement moves are made as are made in Figure 5(b). This time we cannot simply disallow the movement of vertices into the overweight subdomains and otherwise continue refinement as normal. This is because both subdomains are overweight here, the left subdomain is overweight with respect to the first weight, and the right subdomain is overweight with respect to the second weight. Here, for example, moving the two vertices that moved in Figure 5(c) will not work. Even though this will decrease the imbalance with respect to the first constraint, it will increase the imbalance with respect to the second constraint. In fact, what is required is a more complex approach in which vertices that are higher in the first weight than the second are moved from the left to the right subdomain, and vertices that are higher in the second weight than the first are moved from the right to the left subdomain. While balancing a two-constraint partitioning in this way is challenging, the problem of balancing a multi-constraint partitioning becomes even more difficult as the number of constraints increases, as the multiple constraints are increasingly likely to interfere with each other. Given the difficulty of balancing multi-constraint partitionings, a better solution is to avoid a situation 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 way to ensure 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 7. 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 7 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. In the context of single level refinement schemes (i.e., those schemes in which refinement is performed only on the input graph and not on different coarsened versions of the graph), this can be achieved by allowing an additional number of passes through the vertices. In this case, a local minima of the edge-cut will eventually be reached. However, in the context of multilevel refinement, a number of refinement moves on the coarse graphs may never come about (and so a local minima may not be reached on these graphs). This is because the granularity of the vertices here may prohibit a number of potential moves (if the weights of vertices exceed the slices alloted to the processors). In this case, even additional refinement iterations will not allow these moves to be made. As the partitioning is projected to

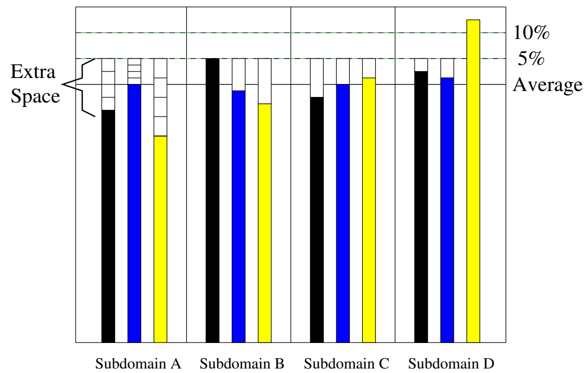

Figure 7: This figure shows the subdomain weights for a 4-way partitioning of a 3-constraint graph. The white bars represent the extra space in a subdomain for each weight given a 5% user specified load imbalance tolerance.

subsequent finer graphs, the decreasing granularity of the vertices will allow greater freedom of movement, and so this effect will be reduced. However, the result is that the number of edge-cut reducing refinement moves performed on the coarse graphs is reduced compared to serial multi-constraint multilevel refinement algorithms (which do not require a scheme that slices up the extra space of a subdomain), and so, this parallel scheme results in lower quality partitionings than serial schemes. 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 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 on the coarse graphs than the straightforward scheme, they still restrict more 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 on every level graph, 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 quality 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, we update only temporary data structures. Next, we perform a global reduction operation

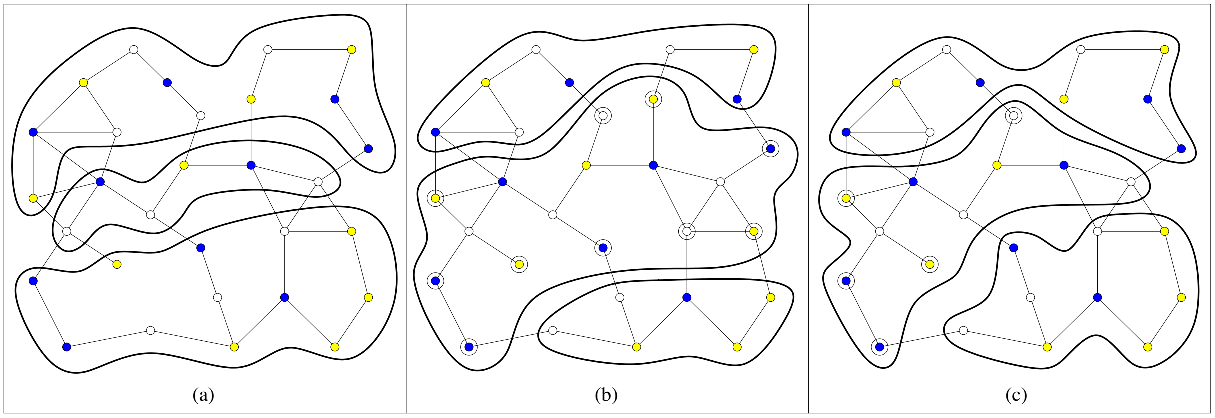

Figure 8: This figure gives an example of a our parallel refinement algorithm for a three-way single constraint partitioning. Here the imbalance tolerance is set to 10%. In figure (a) the top and bottom subdomains are overweight. In figure (b) each processor concurrently proposes a number of refinement moves. If all of these moves are committed, the middle subdomain will become overweight. In figure (c) 50% of the proposed moves on each processor are disallowed. The result is that the partitioning is balanced to within the tolerance.

to determine whether or not the balance constraints will be violated if these moves commit. If none of the balance constraints are violated, we commit these moves as normal. Otherwise, each processor is required to disallow a portion of its proposed vertex moves into those subdomains that would be overweight if all of the moves are allowed to commit. We determine how many moves into potentially overweight subdomains should be disallowed as follows. The percentage of excess vertex weight moved into these subdomains is computed, and every processor is required to disallow this percentage of the weight of its proposed moves into the subdomain. The specific moves to be disallowed are selected randomly by each processor. While selecting moves randomly can negatively impact the edge-cut, this is 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 single constraint refinement algorithm described in [4].

Figure 8 illustrates this process in the context of a single constraint. This is an example of a three-way partitioning on three processors. Each vertex has a (single) weight of one. There are 30 vertices in this graph, so each subdomain should ideally have a weight of ten. However, we set a 10% imbalance tolerance, and so subdomains of weights of 11 are acceptable. Figure 8(a) gives the partitioning prior to the start of our refinement algorithm. Here, the subdomain weights are (from top to bottom) 12, 6, and 12. Therefore, the top and bottom subdomains are overweight by 20%, while the middle subdomain is underweight. Figure 8(b) shows the partitioning after the first pass of our refinement algorithm. The circled vertices represent proposed moves into the middle subdomain. If all of these moves are committed, the subdomain weights will be 8, 16, and 6. Since this would imbalance the middle subdomain over the 10% tolerance, we must disallow a number of these moves. Specifically, the middle subdomain is originally weighted at 6 and can hold up to 11 while still falling within the imbalance tolerance. Therefore, we can move five vertices into it. However, there are ten proposed vertex moves into this subdomain. We divide these two numbers and find that each processor must disallow half of its moves into the middle subdomain. So the blue and yellow processors must each disallow two of their four proposed moves. The white processor must disallow one of its two proposed moves. Figure 8(c) shows the partitioning after this has been done. Now the moves of the circled vertices can be safely committed.

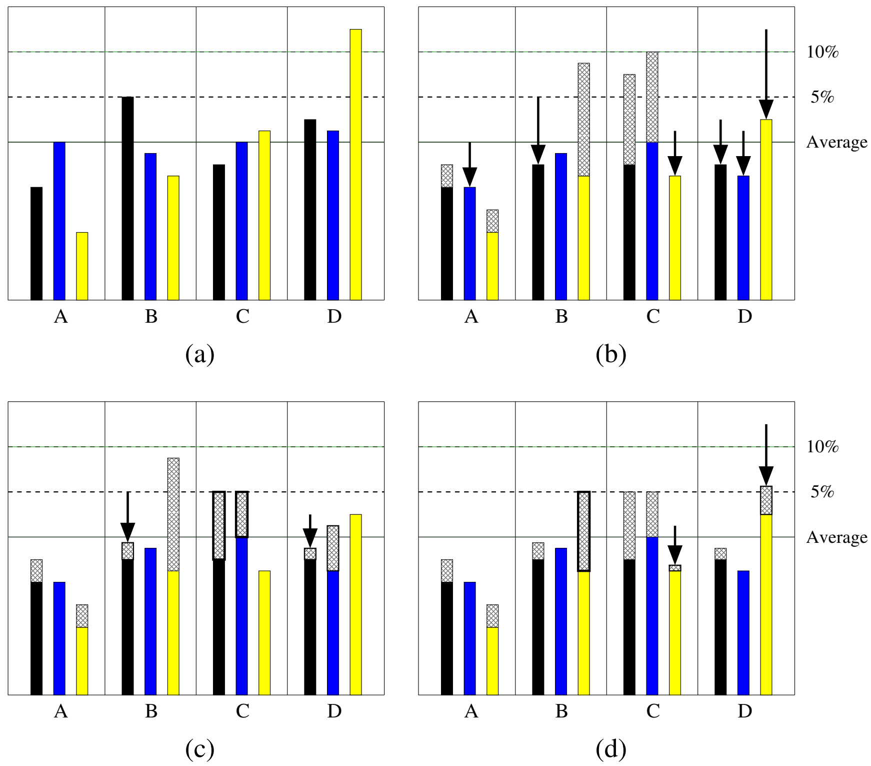

Figure 9: This figure shows the state of a partitioning prior to refinement (a), the proposed state of the partitioning after the first pass of our algorithm (b), the proposed state after moves are disallowed in order to maintain the balance of the first constraint (c), and the proposed state after moves are disallowed in order to maintain the balance of the third constraint (d).

This example demonstrates the parallel refinement algorithm for a single constraint only. Additional issues result in the context of multiple constraints. This is because (i) the vertices are not of unit weight, and (ii) we must commit or disallow vertex moves based on multiple constraints. If vertices are not of unit weight, we must not simply disallow a specific number of vertex moves, but instead must disallow some number of moves such that the sum weight of these meets the required percent (of the total weight of the proposed moves) specified by the global reduction operation. We address the second issue by taking care of each constraint in turn. Figure 9 illustrates this process. Figure 9(a) is a blow up of the example in Figure 7. Figure 9(b) gives the state after the first pass of our refinement algorithm. The down-pointing arrows indicate the amount of vertex weight to be moved out of the subdomain, while the checkered boxes indicate the amount of vertex weight is to be moved into the subdomain. In this example we assume the user specified the imbalance tolerance to be 5%. If the proposed moves commit, subdomain B will be overweight with respect to the third constraint and subdomain C will be overweight with respect to the first and second constraints. Therefore, it is necessary it disallow some of these proposed moves. We start with the first constraint. We must disallow a number of moves into subdomain C such that the total weight of these moves brings the imbalance (of the first constraint) to down to 5%. In Figure 9(c), this has been done. Here, subdomains B and D disallow a number of moves into subdomain C. Notice that in correcting the imbalance of the first constraint, enough moves were disallowed from subdomain D to subdomain C so that the second constraint was also corrected. This was inadvertent. However, it is beneficial in that we can now move on directly to the third constraint. (Note, that if the excess imbalance of the second constraint had not been entirely eliminated, it would have been necessary to disallow the movement of more vertices into subdomain C.) In order to ensure the balance of the third constraint, it is necessary to disallow the transfer of weight into subdomain B. In Figure 9(d), this has been done. The movement of vertices from subdomains C and D are disallowed. In this case, disallowing the movement of vertices from subdomain D to subdomain B results in subdomain D being overweight by a modest amount with respect to the third constraint. Now that we have taken care of all of the constraints, the remaining moves are able to committed.

It is important to note that the above scheme still 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, which require this process to iterate until it converges (or until all of the proposed moves have been disallowed). We have chosen in our implementation 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, as discussed in Section 4.

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

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

and the isoefficiency function is $O(p^2 \log p)$. 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

$$T_{par} = O\left(\frac{nm}{p}\right) + O(pm \log n). \tag{3}$$

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

4 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 $(lr, lr, \ldots, lr)$ 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-physics problems, different regions of the mesh can represent different materials. This can bring about different computation and memory requirements as discussed in Section 1. However, mesh elements that represent the same material typically form 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 similar 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 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
$mrng1$	257,000	1,010,096
$mrng2$	1,017,253	4,031,428
$mrng3$	4,039,160	16,033,696
$mrng4$	7,533,224	29,982,560

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

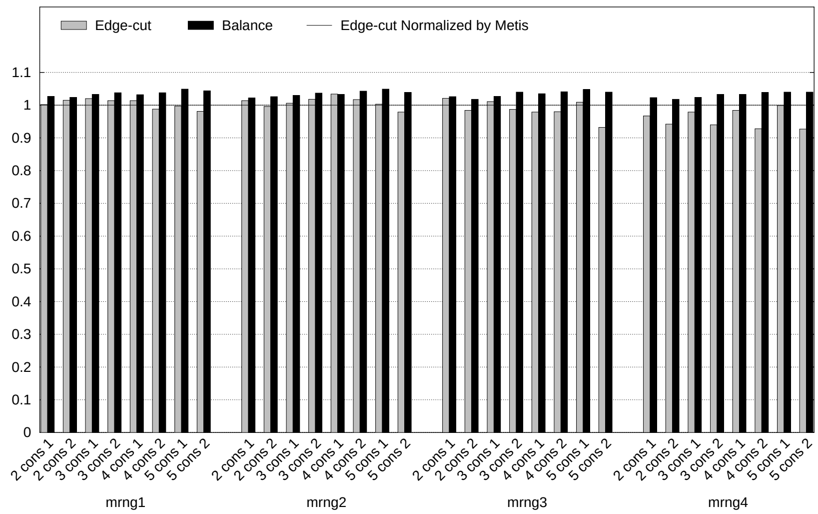

Figure 10: This figure shows the edge-cut and balance results from the parallel multi-constraint algorithm on 32 processors. The edge-cut results are normalized by the results obtained from the serial multi-constraint graph partitioning algorithm implemented in METIS.

Comparison of Serial and Parallel Multi-Constraint Algorithms. Figures 10, 11, and 12 compare the quality of the partitionings produced by our parallel multi-constraint graph partitioning algorithm with the quality of those produced by the serial multi-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 higher-quality partitionings that the serial algorithm. The balance results indicate the maximum imbalance of all of the constraints. These results are not normalized. (Note that we set an imbalance tolerance of 5% for all of the constraints.) Finally, the results given in Figures 10, 11, and 12 are the average of three runs by our algorithm utilizing different random seeds each time. We give the average results of three runs in order to demonstrate the robustness of our parallel algorithm.

Figures 10, 11, and 12 show that our parallel multi-constraint graph partitioning algorithm is able to compute partitionings of similar or better quality than the serial multi-constraint graph partitioner,

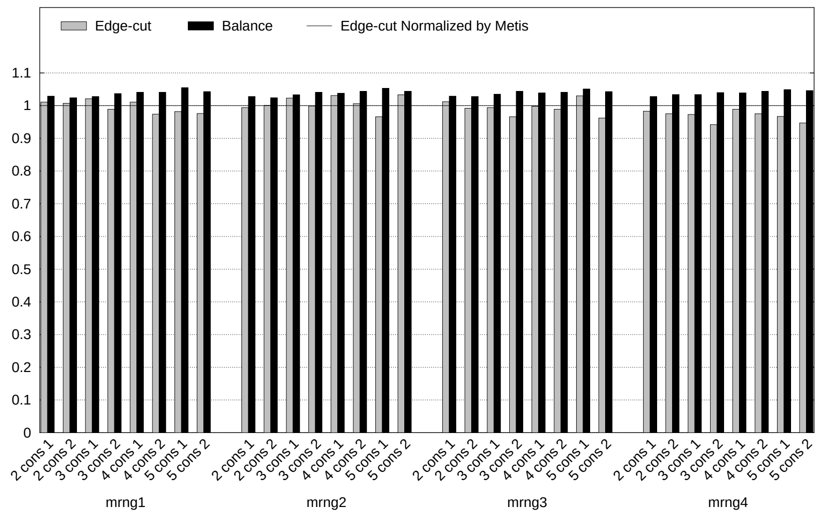

Figure 11: This figure shows the edge-cut and balance results from the parallel multi-constraint algorithm on 64 processors. The edge-cut results are normalized by the results obtained from the serial multi-constraint graph partitioning algorithm implemented in METIS.

while ensuring that multiple constraints are balanced.

Notice that the parallel algorithm is sometimes able to produce higher-quality partitionings than the serial algorithm. There are two reasons for this. First, our parallel matching scheme is not as effective in finding a maximal matching as the serial algorithm. 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. This causes the parallel algorithm to take more coarsening levels to obtain a sufficiently small graph. The result is that the matching algorithm usually has one or more additional coarsening levels in which to remove the exposed edge weight (i.e., the total weight of the edges on the graph). By the time our parallel algorithm computes the coarsest graph, it can have less exposed edge weight than the coarsest graphs computed by the serial algorithm. This allows 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 in the serial algorithm, once the partitioning becomes balanced it will never explore the infeasible solution space in order to improve the partition quality. 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 quality.

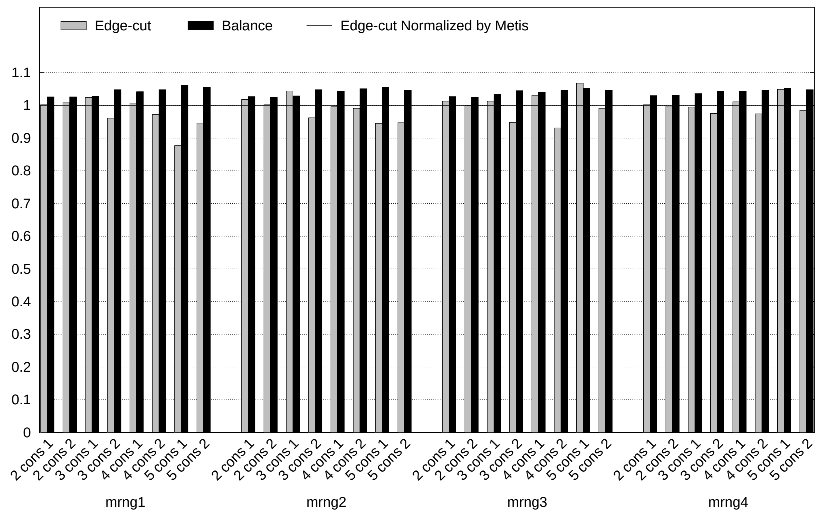

Figure 12: This figure shows the edge-cut and balance results from the parallel multi-constraint algorithm on 128 processors. The edge-cut results are normalized by the results obtained from the serial multi-constraint graph partitioning algorithm implemented in the METIS library.

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 $mrng1$.

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 METIS library [5] for $mrng1$. These results show only modest speedups for the parallel partitioner. The reason is that the graph $mrng1$ is quite small, and so the communication and parallel overheads are significant. However, we use $mrng1$ 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.

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.

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

Graph	8-processors		16-processors		32-processors		64-processors		128-processors	
	time	efficiency	time	efficiency	time	efficiency	time	efficiency	time	efficiency
$mrng2$	9.8	100%	5.3	92%	3.5	70%	2.5	49%	3.1	20%
$mrng3$	31.8	100%	16.9	94%	9.3	85%	5.7	70%	4.4	45%
$mrng4$	out of memory		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
$mrng2$	5.4	3.1	2.1	1.5	1.7
$mrng3$	15.8	8.8	4.8	3.0	2.7
$mrng4$	38.6	16.2	8.8	5.0	3.6

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

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 $mrng3$ 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 $mrng3$ 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 $mrng2$ and $mrng3$, we set the efficiency of eight processors to 100%, while we set the efficiency of 16 processors to 100% for $mrng4$. 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 $mrng3$ is approximately four times as large as $mrng2$ we can test the isoefficiency function experimentally. The efficiency of the multi-constraint partitioner with 32 processors for $mrng2$ is 70%. Doubling the number of processors to 64 and increasing the data size by four times (64-processors on $mrng3$) 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 $mrng2$ and 128 processors on $mrng3$ 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 $mrng2$ and 32 processors on $mrng3$ 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.

5 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 of similar quality to 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, partition quality, and balance results presented in Section 4.

While the experimental results presented in Section 4 (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 still maintaining all of the balance constraints decreases. Therefore, the quality of the produced partitionings decreases more significantly.

References

[1] C. J. Alpert, J. H. Huang, and A. B. 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. METIS: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices, version 4.0. 1998.

[6] G. Karypis and V. Kumar. Multilevel algorithms for multi-constraint graph partitioning. Technical Report TR 98-019, Dept. of Computer Science, Univ. of Minnesota, 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. ParMETIS: Parallel graph partitioning and sparse matrix ordering library. Technical report, University of Minnesota, Department of Computer Science and Engineering, 1997.

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

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