CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2002; **14**:219–240 (DOI: 10.1002/cpe.605)

Parallel static and dynamic multi-constraint graph partitioning[‡]

Kirk Schloegel[*,†], George Karypis and Vipin Kumar

Army HPC Research Center, Department of Computer Science and Engineering, University of Minnesota, 4-192 EE/CS Building, 200 Union St., Minneapolis, MN 55455, U.S.A.

SUMMARY

Sequential multi-constraint graph partitioners have been developed to address the static load balancing requirements of multi-phase simulations. These work well when (i) the graph that models the computation fits into the memory of a single processor, and (ii) the simulation does not require dynamic load balancing. The efficient execution of very large or dynamically adapting 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 multi-constraint graph-partitioning algorithm, as well as a new partitioning algorithm for dynamic multi-phase simulations. We describe these algorithms and give experimental results conducted on a 128-processor Cray T3E. These results show that our parallel algorithms are able to efficiently compute partitionings of similar edge-cuts as serial multi-constraint algorithms, and can scale to very large graphs. Our dynamic multi-constraint algorithm is also able to minimize the data redistribution required to balance the load better than a naive scratch-remap approach. We have shown that both of our parallel multi-constraint graph partitioners are as scalable as the widely-used parallel graph partitioner implemented in PARMETIS. Both of our parallel multi-constraint graph partitioners are very fast, as they are able to compute three-constraint 128-way partitionings of a 7.5 million vertex graph in under 7 s on 128 processors of a Cray T3E. Copyright © 2002 John Wiley & Sons, Ltd.

KEY WORDS: multi-constraint graph partitioning; parallel graph partitioning; multilevel graph partitioning; multi-phase scientific simulation

*Correspondence to: Kirk Schloegel, Army HPC Research Center, Department of Computer Science and Engineering, University of Minnesota, 4-192 EE/CS Building, 200 Union St., Minneapolis, MN 55455, U.S.A.
†E-mail: kirk@cs.umn.edu

Contract/grant sponsor: DOE; contract/grant number: LLNL B347881
Contract/grant sponsor: NSF; contract/grant number: CCR-9972519
Contract/grant sponsor: Army Research Office; contract/grant number: DA/DAAG55-98-1-0441
Contract/grant sponsor: Army High Performance Computing Research Center; contract/grant number: DAAH04-95-2-0003/DAAH04-95-C-0008

‡The original version of this article was first published as 'Schloegel K, Karypis G, Kumar V. Parallel static and dynamic multi-constraint graph partitioning. *Euro-Par 2000—Parallel Processing* (*Lecture Notes in Computer Science*, vol. 1900), Bode A, Ludwig T, Karl W, Wismüller R (eds.). Springer, 2000; 296–310', and is reproduced here by kind permission of the publisher.

Received 14 May 2001
Revised 23 May 2001

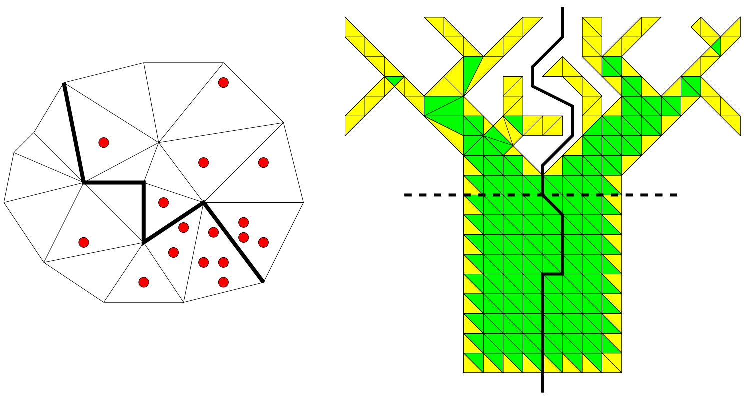
(a) (b)

Figure 1. A computational mesh for a particle-in-cells simulation (a) and a computational mesh for a contact–impact simulation (b). The particle-in-cells mesh is partitioned so that both the number of mesh elements and the number of particles are balanced across the subdomains. Two partitionings are shown for the contact–impact mesh. The dashed partitioning balances only the number of mesh elements. The solid partitioning balances both the number of mesh elements and the number of surface (lightly shaded) elements across the subdomains.

1. INTRODUCTION

Algorithms that find good partitionings of irregular graphs are commonly used to map computational meshes onto the processors of high-performance parallel computers such that each processor gets a roughly equal number of mesh elements (to ensure load balance) and such that the amount of inter-processor communication required to exchange information between adjacent mesh elements is minimized. However, the traditional graph-partitioning-problem formulation is limited in the types of applications that it can effectively model because it specifies that only a single quantity be load balanced. Many important types of multi-phase simulations require that multiple quantities be load-balanced simultaneously. This is because synchronization steps exist between the different phases of the computations, and so each phase must 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 partitioning 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 phase of the computation.

Two examples are particle-in-cells [1] and contact–impact simulations [2]. Figure 1 illustrates the type of partitionings that are needed for these simulations. Figure 1(a) shows a partitioning of the particle-in-cells mesh that balances both the number of mesh elements and the number of particles across the subdomains. Figure 1(b) shows a mesh for a contact–impact simulation. During the contact-detection phase, computation is performed only on the surface (i.e. lightly shaded) elements, while during the impact phase, computation is performed on all of the elements. Therefore, in order to ensure that both phases are load balanced, a partitioning must balance both the total number of mesh elements and the number of surface elements across the subdomains. The solid partitioning in Figure 1(b) does this. The dashed partitioning is similar to that which a traditional graph partitioner might compute. This partitioning balances only the total number of mesh elements. The surface elements are imbalanced by over 50%.

A new formulation of the graph-partitioning problem is presented in [3] that is able to model the problem of balancing multiple computational phases simultaneously, while also minimizing the inter-processor communications. In this formulation, a weight vector of size m is assigned to each vertex of the graph. The *static multi-constraint graph-partitioning problem* formulation then is to partition the vertices into k disjoint subdomains such that the total weight of the edges that are cut by the partitioning (i.e. the *edge-cut*) is minimized, and such that each of the m vertex weights are balanced across the subdomains. A serial static multi-constraint graph-partitioning algorithm is also presented in [3] that is shown to compute high quality partitionings. However, a parallel formulation of this algorithm is desirable. This is because the computational meshes that are used in parallel multi-phase simulations are often too large to fit in the memory of a single processor. A parallel partitioner can take advantage of the increased memory capacity of parallel machines. Thus, a parallel multi-constraint graph partitioner is key to the efficient execution of very large multi-phase simulations.

A static partitioning computed *a priori* is sufficient for many types of multi-phase simulations. However, simulations whose computational structure evolves dynamically (e.g. due to adaptive mesh methods or the movement of particles between subdomains) require that dynamic repartitionings be periodically computed to maintain the load balance. These repartitionings should be computed to load balance each phase, to minimize the inter-processor communication, and also to minimize the difference between it and the previous partitioning (i.e. minimize the amount of data redistribution that is required to balance the loads). The *dynamic multi-constraint graph-partitioning problem* formulation then is similar to the static problem, but has an additional objective, that is to minimize the amount of data redistribution required to realize the new partitioning. Furthermore, dynamic repartitionings need to be computed in parallel, as it is not scalable to download the mesh onto a single processor each time that load balancing is required.

This paper describes new parallel static and dynamic multi-constraint graph-partitioning algorithms. We present a scalability analysis of the algorithms as well as experimental results conducted on a 128-processor Cray T3E. Our results show that our parallel algorithms are able to compute balanced partitionings that have similar edge-cuts as those produced by a serial multi-constraint algorithm, while also being fast and scalable to very large graphs. Experimental results also show that our dynamic multi-constraint algorithm requires significantly less data redistribution than a naive scratch-remap (SR) repartitioning approach.

The remainder of the paper is organized as follows. Section 2 describes related work in the area, including serial and parallel multilevel partitioning algorithms and serial multi-constraint graph partitioners. Section 3 presents our new parallel static and dynamic multi-constraint graph-partitioning algorithms and gives a scalability analysis of these. Section 4 gives experimental results for our schemes. Section 5 gives conclusions.

2. BACKGROUND

In this section, we describe related work in parallel multi-constraint partitioning. Then we describe the multilevel graph-partitioning paradigm and its extension to the context of multiple balance constraints.

2.1. Previous work

A lot of work has been done developing parallel static and dynamic single-constraint partitioning algorithms. (See [4] for a survey of these.) However, these problems have received less attention for the more difficult cases of multiple balance constraints [1,5].

Walshaw, Cross, and McManus [5] present a *multi-phase partitioner* that utilizes a slight modification of a traditional graph partitioner as a black box to compute static partitionings for an important class of multi-phase computations. This method splits the vertices into m disjoint subsets depending on the first phase of the m-phase computation in which they are active. Then, the sub-graph that consists of only those vertices that are in the first subset along with the edges that connect them is partitioned normally. The subdomains that are computed for these vertices are locked and cannot later be changed. Next, the vertices that are in the second subset are added to the sub-graph (as well as the edges that connect any of the vertices that are now in the sub-graph), and the sub-graph is partitioned. Note that here, only the vertices from the second subset are free to change subdomains. The result is that these free vertices are likely to be assigned to the same subdomains as their locked neighbors. After the second round of partitioning is completed, the subdomains that were computed for the second subset of vertices are locked, the vertices (and edges) from the third subset are added to the sub-graph, and the sub-graph is again partitioned. This process continues for m iterations, at which time a multi-phase partitioning results.

This algorithm is relatively simple to implement and appears to be equally applicable for dynamic as well as static partitioning. However, the method is not as general as the multi-constraint graph-partitioning formulation as it is not effective for certain classes of multi-phase computation (e.g. those in which all of the mesh elements take part in every computation phase).

Watts, Rieffel, and Taylor [1] present a dynamic multi-phase partitioner that is task-based (as opposed to graph-based). This scheme does not take the interdependencies among the tasks into account when satisfying the balance constraints, and so does not minimize the inter-processor communications.

2.2. Multilevel graph partitioning: a review

The graph-partitioning problem is NP-complete. Therefore, heuristic algorithms are used to compute partitionings on graphs of interesting size. In particular, multilevel schemes [6–10] have been developed that are able to quickly compute excellent partitionings for graphs that correspond to the 2D or 3D irregular meshes used for scientific simulations.

Multilevel graph-partitioning algorithms consists of three phases: graph coarsening, initial partitioning, and uncoarsening/refinement. In the graph-coarsening phase, a series of graphs is constructed by collapsing together selected vertices of the input graph in order to form a related coarser graph. This newly-constructed graph then acts as the input graph for another round of graph coarsening, and so on, until a sufficiently small graph is obtained. A common method for graph coarsening is collapsing together the pairs of vertices that form a matching. A matching of the graph is a set of edges, no two of which are incident on the same vertex. Such matchings can be computed by a number of methods. Possible schemes include random matching, heavy-edge matching [11], maximum weighted matching [12], and approximated maximum weighted matching [9]. Computation of the initial partitioning is performed on the coarsest (and hence smallest) of these graphs, and so is very

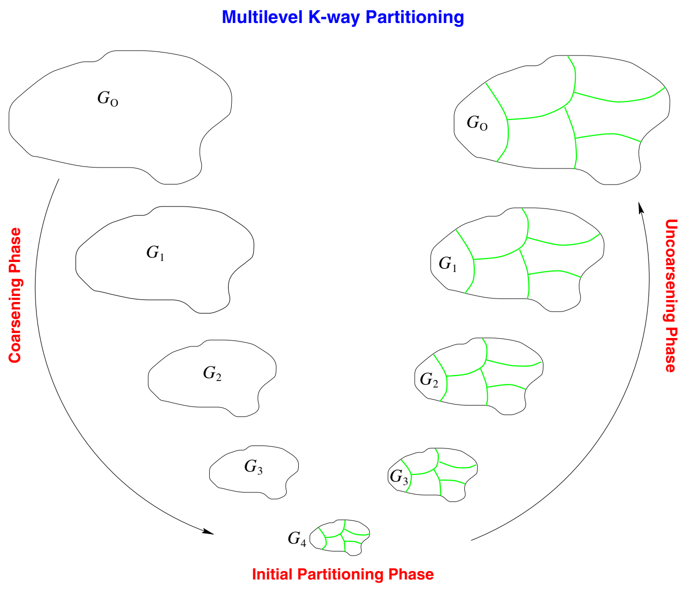

Figure 2. The three phases of multilevel k-way graph partitioning. During the coarsening phase, the size of the graph is successively decreased. During the initial partitioning phase, a k-way partitioning is computed, During the uncoarsening/refinement phase, the partitioning is successively refined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

fast. After the initial partitioning is computed, it is projected back up to the finest (i.e. original) graph. On each subsequent finer graph, iterative refinement is performed using a KL/FM-type algorithm [13,14]. Such algorithms repeatedly run through the vertices that lie on the subdomain boundaries, swapping the subdomains of those that will lead to an improvement in the quality of the partitioning. Figure 2 illustrates the multilevel paradigm.

2.2.1. *Parallel formulations of single-constraint multilevel graph-partitioning algorithms*

Parallelizing a multilevel graph partitioner requires parallel formulations for the three phases of the multilevel paradigm. We briefly describe parallelization techniques that have been developed for each of these.

The key component of the parallel formulation of the graph-coarsening phase is a distributed matching scheme. Since the input graph is distributed across the processors, a naive scheme can require high degrees of fine-grain inter-processor communication. Distributed matching is also subject to contentions among matched vertices that reside on different processors. To address these issues, fast and scalable distributed matching schemes have been developed that either subdivide the matching computation into independent phases so that conflicts are avoided [10,15] or else implement distributed contention-resolution protocols [16].

The parallel formulation of the initial partitioning phase is more straightforward. Since initial partitioning is performed only one time on an arbitrarily small graph, it can be done serially. A recursive task-decomposition approach can also be employed to reduce the run time of this phase.

KL/FM-type refinement that is used during the uncoarsening phase is serial in nature, as it selects vertices for movement from a priority queue that is updated after every move. A simplified (and generalized) version of this algorithm, presented in [15], does not use a priority queue, but selects vertices for movement in a random order. This step is easier to parallelize because it allows processors to simultaneously move their local vertices. However, it still requires some method to ensure that concurrent moves of adjacent vertices do not interfere with one another, resulting in worse edge-cuts than expected. Therefore, parallel refinement algorithms typically subdivide refinement into independent phases (similarly to distributed matching schemes).

2.3. Partitioning for multiple balance constraints

Just as a parallel multilevel algorithm requires parallel formulations for each of the three phases of the multilevel paradigm, so too does a multi-constraint graph-partitioning algorithm require that each phase be extended to support multiple balance constraints. In the remainder of this section, we describe such extensions.

The specific matching scheme used during graph coarsening can have a significant impact on the quality of the final partitioning. Single-constraint partitioners typically use variations of the heavy-edge matching heuristic, as this heuristic effectively reduces exposed edge weight in the coarsest graph, and so, makes initial partitioning easier [4] compared to other schemes. A new *balanced-edge* heuristic is presented in [3] for multi-constraint partitioning. This heuristic tends to reduce the heterogeneity of vertex weights on the coarsest graph, and so, makes the problem of finding a balanced initial partitioning easier compared to other schemes. A combination of these two heuristics (i.e. heavy-edge matching using the balanced-edge heuristic as a tie breaker) is shown to result in high-quality, balanced partitionings for a number of diverse problems [3].

The initial partitioning must be computed such that multiple balance constraints are satisfied. The authors of [3] 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 (an assumption that usually holds for medium- to large-size graphs). The lemma leads to an algorithm for computing such a bisection, which can be applied recursively to compute a k-way partitioning.

The uncoarsening/refinement phase can be extended to the context of multiple balance constraints straightforwardly. In particular, prior to each refinement move, the vertex weight vector must be checked to ensure that moving the vertex does not upset any of the balance constraints. The result is that as the number of constraints increase, it becomes increasingly difficult to find vertices that are able to be moved to improve the edge-cut. Therefore, as the number of constraints increases, the quality of the partitioning tends to decrease.

3. PARALLEL MULTI-CONSTRAINT GRAPH PARTITIONING

In this section, we describe our static and dynamic multi-constraint graph-partitioning algorithms and give a scalability analysis of them.

3.1. A parallel static multi-constraint graph-partitioning algorithm

A parallel formulation of the multi-constraint partitioning algorithm presented in [3] can use a number of the techniques described in Section 2.2 for parallel single-constraint partitioning. In particular, it is straightforward to extend the parallel formulations of the coarsening and initial partitioning phases to the context of multi-constraint partitioning. The key challenge is the parallel formulation of the refinement algorithm used during the uncoarsening phase. We briefly explain why parallel refinement in the context of multiple balance constraints is challenging and then describe our parallel multilevel refinement algorithm.

3.1.1. Issues

In order to guarantee that balance is maintained during parallel refinement, it is necessary to update global subdomain weights after every vertex move. 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 a balanced partitioning may become imbalanced during refinement. For example, if two processors each send enough vertex weight into a single subdomain to fill its capacity, then the combined effect will cause the subdomain to become overweight.

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

A similar strategy does not work for multi-constraint partitioning because correcting load imbalances in the context of multiple constraints is a much more difficult problem than the single-constraint case. This is because a two-way transfer of vertices between adjacent subdomains is typically required to equalize their weights (instead of only a one-way transfer from the heavier to the lighter subdomain). The problem becomes increasingly difficult as the number of constraints increases, as the different weights are increasingly likely to interfere with each other during balancing. Given the difficulty of balancing multiple vertex weights, a better approach is to avoid situations in which partitionings become imbalanced in the first place. Thus, 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 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 3. This shows the subdomain weights for a four-way, three-constraint partitioning. Lets assume

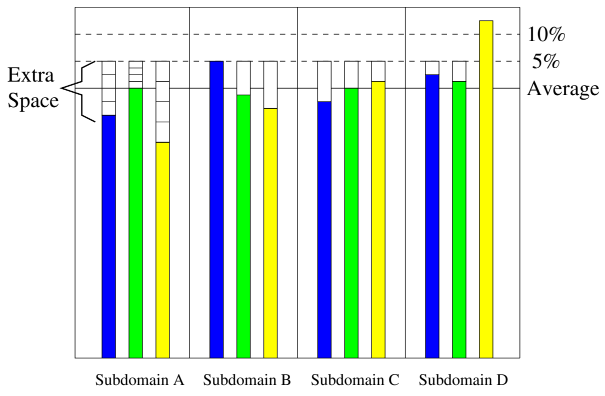

Figure 3. This figure shows the subdomain weights for a four-way partitioning of a three-constraint graph. The white bars represent the extra capacity in a subdomain for each weight given a 5% user specified load-imbalance tolerance.

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 is required to bring the total weight up to 5% above the average. In other words, the white bars show the amount of *extra capacity* each subdomain has for a particular weight given a 5% load imbalance tolerance. Figure 3 shows how the extra capacity that is associated with each vertex weight can be split up for the four processors in subdomain A. If each processor is limited to moving the indicated amounts of weight into subdomain A, then it is guaranteed that the 5% imbalance tolerance will not be exceeded.

While this method is guaranteed to maintain multiple balance constraints, it becomes increasingly restrictive in terms of allowing vertex moves as the number of processors increases. This is because whenever a processor does not use up all of its allocated capacity in a subdomain, the remainder of the capacity is not usable by any other processor. Hence, some of the capacity is wasted. As the number of processors increases, processors are allocated thinner slices of each subdomain's capacity, and so the potential for wasted capacity increases. Furthermore, as the number of constraints increases, this problem also becomes worse, as each constraint imposes its own restriction. Since the serial algorithm is able to use up all of the capacity in every subdomain effectively, it allows more edge-cut-reducing refinement moves to be made in general than this scheme, and so, results in higher-quality partitionings.

It is possible to allocate the extra capacity 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 naive 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 that these results are not presented in this paper.)

3.1.2. Our parallel multi-constraint refinement algorithm

The above discussion suggests that the key requirements of a parallel multi-constraint refinement algorithm are (i) that it has a high degree of concurrency, (ii) that it helps to maintain multiple balance constraints, and (iii) that it is highly effective. We have developed a parallel multi-constraint refinement algorithm that meets all three of these requirements. We next describe the algorithm.

The algorithm (essentially a reservation scheme) performs two passes through the vertices during 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 all of these moves commit. If none of the balance constraints are violated, the moves are committed. 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. This portion is equal to the total vertex weight that exceeded the capacity in the subdomain divided by the total weight of all of the 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 serious problem because the number of disallowed moves is usually small and further refinement can easily correct the effects of any poor selections that happen to be made here. Except for these modifications, our multi-constraint refinement algorithm is similar to the coarse-grain refinement algorithm described in [16].

It is important to note that the above scheme still does not guarantee to maintain the balance constraints. This is because whenever some number of vertex moves are disallowed; the weights of the subdomains from which these vertices were to have moved are actually higher than the weights that were computed by the global reduction operation. Some of these subdomains might then become overweight if the remainder of the vertex moves are allowed to commit. It is possible to correct this situation by performing a second global reduction operation followed by a second round in which moves are disallowed. These corrections might then lead to other imbalances, whose corrections might lead to others, and so on. This process can iterate until either it converges or until all of the proposed moves have been disallowed.

Instead, we have chosen to ignore 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 these is quite modest. Our experimental results show that the amount of imbalance that is introduced in this way is small enough that further refinement is usually 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 4.)

This parallel refinement algorithm, combined with the straightforward extensions to existing parallel multilevel partitioning algorithms, constitute the components of our parallel static multi-constraint graph-partitioning algorithm.

3.2. A dynamic multi-constraint graph partitioner for adaptive simulations

The dynamic graph-partitioning problem is similar to the static problem. However, there is an additional objective here. That is, the amount of data redistribution that is required to realize the new partitioning needs to be minimized. Two general approaches to computing dynamic repartitionings have been developed. The first is to compute a new partitioning from scratch and the second is to use a diffusion-based method to perturb the original partitioning enough so as to balance it. Preliminary results on diffusion-based repartitioning for multi-constraint problems indicate that this approach is effective only for two or three constraints [17]. This is due to the difficulty of balancing multiple vertex weights (as discussed in Section 3.1). Therefore, we focus this paper on the more general approach of computing a multi-constraint repartitioning by partitioning the graph from scratch.

3.2.1. Multi-constraint SR

Using an existing static partitioner to compute dynamic repartitionings can be fairly effective, as the inter-processor communications will be minimized and all of the balance constraints satisfied. However, the data redistribution costs will not be minimized. A simple method, referred to as SR repartitioning, can reduce data redistribution costs by intelligently remapping the subdomain labels of the new partitionings to those of the original partitionings [18,19]. A fast and effective greedy scheme for partition remapping is described in [18]. By applying this scheme, our parallel static multi-constraint partitioner can effectively be used as a dynamic multi-constraint partitioning algorithm. We refer to this algorithm as Mc-SR.

3.2.2. Multi-constraint LMSR

Experimental results presented in [20] show that data redistribution costs can be reduced compared to a naive SR approach by (i) utilizing the information that is implicit in the original partitioning during the computation of the new partitioning, and by (ii) modifying the refinement algorithm to minimize data redistribution costs as a secondary objective. A scheme called, Locally-Matched Scratch-Remap (LMSR), is presented in [20] that achieves both of these by modifying the graph-partitioning algorithm as follows: (i) graph coarsening is purely local, that is, vertices may be coarsened together only if they belong to the same subdomain on the original partitioning; (ii) partition remapping is performed immediately after the computation of the initial partitioning and before refinement begins; (iii) the refinement scheme is modified to minimize the total amount of data redistribution as a secondary objective during the uncoarsening phase. These modifications can be made to our parallel multi-constraint partitioner without difficulty. We refer to the algorithm that incorporates all three of these as Mc-LMSR.

3.3. Scalability analysis

The scalability analysis of a parallel multilevel single-constraint graph partitioner is presented in [15]. 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 vertices in successive coarser graphs decreases by a factor of $1 + \epsilon$, where $0 < \epsilon \leq 1$. (Note that these assumptions typically hold true for graphs that correspond to well-shaped finite-element meshes.) Under these assumptions, the parallel run time of the single-constraint algorithm is

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

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 static and dynamic multi-constraint graph partitioners are similar (given the above assumptions). However, during both graph coarsening and refinement, all m vertex weights must be considered. Therefore, the parallel run times of the multi-constraint algorithms are m times longer, or

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

Since the sequential complexity of the serial multi-constraint algorithm is $O(nm)$, the isoefficiency functions of the multi-constraint static and dynamic partitioners are also $O(p^2 \log p)$.

4. EXPERIMENTAL RESULTS

In this section, we present experimental results of our parallel multi-constraint graph-partitioning algorithms on up to 128 processors of a Cray T3E. We constructed two sets of test problems, each set consisting of two-, three-, four-, and five-constraint problems, in order to evaluate the effectiveness of our parallel partitioning algorithms in computing high-quality, balanced partitionings quickly. Both sets of problems were generated synthetically from the four graphs described in Table I.

The purpose of the first set of problems was to test our multi-constraint partitioners on some relatively hard problems. For each graph in Table I, a 16-way partitioning of the graph was first computed by the ParMETIS_PartKway routine in PARMETIS [21] using unit vertex weights. Then, every vertex belonging to a given subdomain was assigned the same weight vector. The weight vectors for the subdomains were 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 models a class of multi-phase applications. In these, 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. In this first test set, each of the 16 subdomains represents such a contiguous region of mesh elements.

The purpose of the second set of problems was to test the performance of the multi-constraint partitioners for multi-phase computations in which different (and possibly overlapping) subsets of elements participate in different phases. Test problems with two, three, four, and five weights were constructed for each of the four graphs. These represent two-, three-, four-, and five-phase computations. In the case of five phases, the portions of the graph that are active (i.e. performing computations) are 100, 75, 50, 50, and 25% of the subdomains. In the four-phase case, these are 100, 75, 50, and 50%. In the three- and two-phase cases, they are 100, 75, and 50%, and 100 and 75%,

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

Graph	No. of verts	No. 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

respectively. The portions of the graph that are active was determined as follows. First, a 32-way partitioning of the graph was computed and then a subset of these subdomains was randomly selected 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. This is an accurate model of the overall information exchange between vertices because a vertex exchanges data with its neighbor only if both are active in a given phase.

4.1. Comparison of the serial and parallel static multi-constraint algorithms

Figures 4, 5, and 6 compare the edge-cuts of the partitionings produced by our parallel static multi-constraint graph-partitioning algorithm with those produced by the serial multi-constraint algorithm implemented in the METIS software library [22], and give the maximum load imbalance of the partitionings produced by our algorithm for 32, 64, and 128 processors of a Cray T3E. Each figure shows four sets of results, one for each of the four graphs described in Table I. 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. 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 shown in Figures 4, 5, and 6 give the arithmetic means of three runs by our algorithm utilizing different random seeds each time. We give the average of three runs to verify the robustness of the parallel algorithm. Note that in every case the results of each individual run are within a few per cent of the averaged results shown.

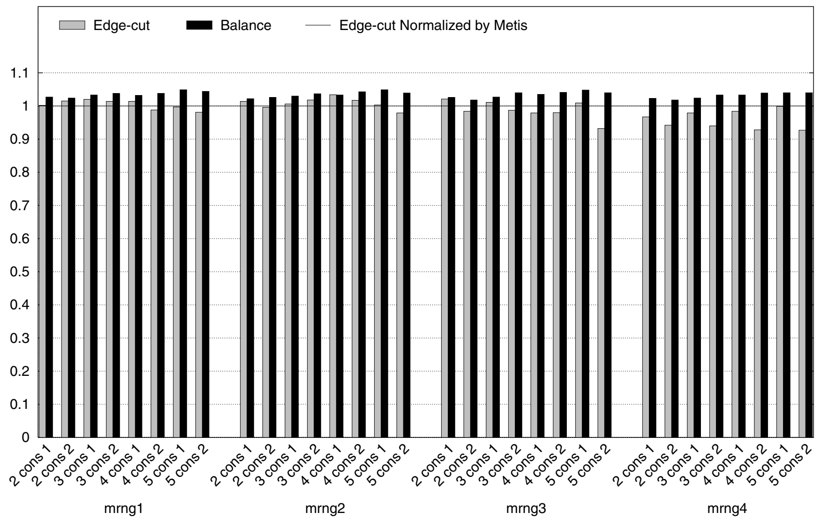

Figure 4. This figure shows the edge-cut and balance results from the parallel multi-constraint algorithm on 32 processors. The edge-cut results are normalized by the results obtained from the serial multi-constraint algorithm implemented in METIS.

Figures 4, 5, and 6 show that our parallel static 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 slightly 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 [3]) 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 by different processors [16].) Therefore, a smaller number of vertices match together with the parallel algorithm than with the serial algorithm. The result is that a graph coarsened by the parallel algorithm tends to be larger than one coarsened by serial algorithm. Naturally, this means that the parallel algorithm takes more coarsening levels to obtain a sufficiently small graph. While this slows down the coarsening phase of the parallel algorithm, it also tends to result in coarsest-level graphs that have less exposed edge weight (i.e. the total weight of the edges on the graph) than those computed by the serial algorithm. This characteristic improves the effectiveness of the initial partitioning algorithm, and so, results in higher-quality partitionings. As mentioned above, 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 (and the parallel efficiency is decreased). Note that this phenomenon of slow coarsening affecting the partition quality was also observed in the context of hypergraphs in [23].

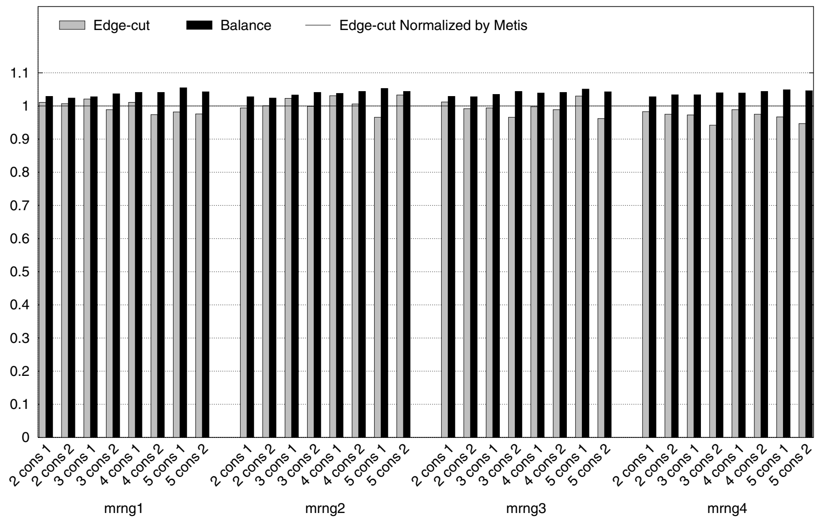

Figure 5. This figure shows the edge-cut and balance results from the parallel multi-constraint algorithm on 64 processors. The edge-cut results are normalized by the results obtained from the serial multi-constraint algorithm implemented in METIS.

The second reason that the parallel algorithm is often able to compute higher-quality partitionings than the serial algorithm is because it is more free than the serial algorithm to explore the infeasible solution space in order to improve the quality of the partitioning. In the serial algorithm, once the partitioning becomes balanced it will never again explore the infeasible solution space. 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 is 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.

4.2. Comparison of the dynamic multi-constraint algorithms

We present experimental results for the Mc-SR and Mc-LMSR algorithms on 128 processors of a Cray T3E. We evaluated our Mc-LMSR algorithm on a series of problems that use the same test sets that were constructed to evaluate our static multi-constraint algorithm. For each test problem, we first computed a p-way, single-constraint partitioning with the ParMETIS_PartKway routine in

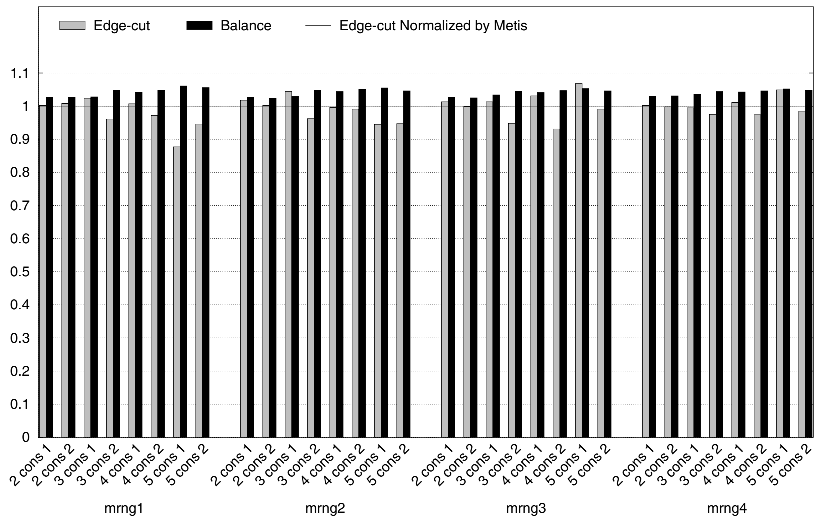

Figure 6. This figure shows the edge-cut and balance results from the parallel multi-constraint algorithm on 128 processors. The edge-cut results are normalized by the results obtained from the serial multi-constraint algorithm implemented in METIS.

PARMETIS, where p is equal to the number of processors. The computation of this partitioning was based only on the first vertex weight. Next, we redistributed the graph across the processors with respect to this partitioning. The result is a multi-constraint repartitioning problem in which one of the balance constraints is satisfied, and $m - 1$ of them are imbalanced. These are particularly difficult problems as some of the imbalance levels for the ignored vertex weights are over 1000%.

Figure 7 compares the edge-cuts, the total amounts of data redistribution required, and the partition balance results obtained from the Mc-SR algorithm with those obtained from the Mc-LMSR algorithm. The edge-cut and data-redistribution bars indicate results obtained by the Mc-LMSR algorithm normalized by those obtained by the Mc-SR algorithm. Therefore, a bar below the 1.0 index line indicates that the Mc-LMSR algorithm produced better results than the Mc-SR algorithm. The balance bars indicate the maximum imbalance of all of the constraints obtained by the Mc-LMSR algorithm. These results are not normalized. Finally, the results shown are the arithmetic means of three runs by our algorithms utilizing different random seeds.

Figure 7 shows that the Mc-LMSR algorithm is able to compute balanced partitionings of similar quality to the Mc-SR scheme, while obtaining data redistribution results that are 50 to 70% of those obtained by the Mc-SR scheme. The edge-cut results of the Mc-LMSR scheme are generally a few per cent higher than the edge-cut results of the Mc-SR algorithm. This is consistent with the (single-constraint) SR and LMSR results reported in [20]. These results show that the Mc-LMSR algorithm is able to compute high-quality repartitionings that minimize the data redistribution costs better than a naive scratch-remap algorithm.

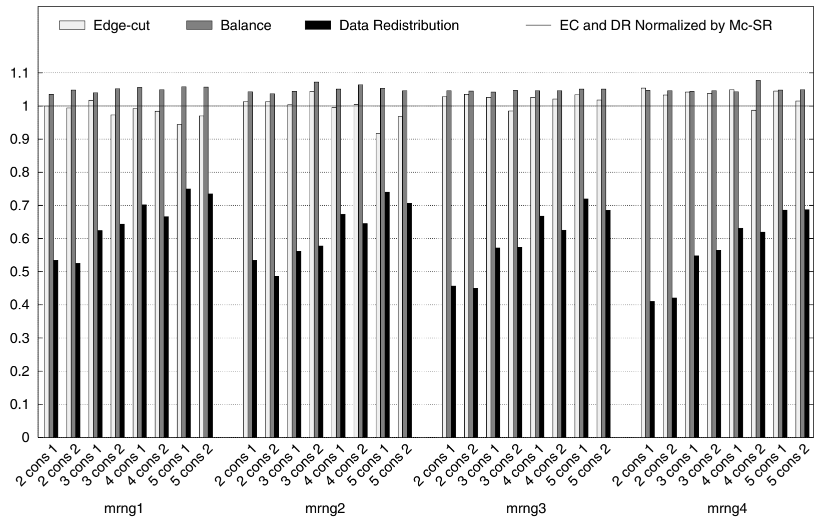

Figure 7. The edge-cut, data redistribution, and balance results for the Mc-SR and Mc-LMSR algorithms on a 128-processor Cray T3E. The edge-cut and data redistribution results obtained from Mc-LMSR are normalized by those obtained by Mc-SR.

4.3. Experimental results from a dynamically evolving application domain

We tested the Mc-SR and Mc-LMSR algorithms on a problem set originating in a scientific simulation of a diesel internal combustion engine. This is a two-phase computation. The first phase involves the mesh-based computation of the combustion engine. The second phase involves the tracking of fuel particles throughout the engine. At first, no fuel particles are present in the combustion chamber. As the computation progresses, fuel particles are injected into the chamber at a single point and begin to spread out. In order to balance both phases, it is necessary to ensure that each processor has the same number of mesh elements and the same number of fuel particles. This cannot be done by a static partitioning because the fuel particles are not present in the initial mesh and they move around between subdomains during the course of the computation.

Table II gives the edge-cut, data redistribution, and balance results obtained from the Mc-SR and Mc-LMSR algorithms, compared to those obtained by a static (and single-constraint) partitioning of the mesh. These results are the average from four repartitionings of a 175-thousand element mesh. The balance results have two numbers. The first is the load balance for the mesh-based computation. The second is the load balance for the particle-tracking computation. Repartitioning was performed every 150 time-steps of the simulation. Results are given for 8-, 16-, 32-, and 64-way partitionings.

The results in Table II show that the Mc-LMSR algorithm was able to compute high-quality repartitionings that minimize the number of elements moved, while balancing both phases of the computation. The Mc-SR algorithm was able to compute good quality, balanced repartitionings,

Table II. Edge-cut, data redistribution, and balance results obtained from Mc-SR, Mc-LMSR, and from the static (single-constraint) k-way partitioner implemented in METIS for a sequence of problems derived from a particle-in-cells simulation of the combustion chamber of a diesel engine.

Scheme	Edge-cut	Data redist.	Load	Balance
8 processors				
Mc-SR	9412	23 171	1.01	1.05
Mc-LMSR	8188	897	1.01	1.05
Static	8028	0	1.02	3.47
16 processors				
Mc-SR	17 398	60 364	1.01	1.06
Mc-LMSR	15 073	5772	1.01	1.07
Static	15 068	0	1.03	12.14
32 processors				
Mc-SR	25 243	75 534	1.04	1.15
Mc-LMSR	22 635	3200	1.03	1.11
Static	22 825	0	1.03	9.29
64 processors				
Mc-SR	33 665	91 364	1.23	1.49
Mc-LMSR	31 168	24 431	1.37	1.59
Static	31 127	0	1.03	33.45

but resulted in many more elements being moved (typically by an order of magnitude). The static partitioning did not move any elements. However, it resulted in particle tracking computations that are highly imbalanced. For example, the imbalance result of 33.45 for the particle-tracking component of the 64-way partitioning means that the best possible speed-up that could be obtained for this phase of the computation on 64 processors is $64/33.45 = 1.9$.

Note that in these experiments, Mc-LMSR is able to compute partitionings with better edge-cuts than Mc-SR. This is because so little data redistribution is required to balance the load for this problem set (only 14% of the vertices are moved by Mc-LMSR for the 64-way repartitioning), that the Mc-LMSR algorithm is able to compute partitionings that are very similar to the original partitioning. The Mc-LMSR scheme has the advantage here because, by using purely local coarsening, it is able to maintain much of the structure of the good original partitioning. The Mc-SR algorithm knows nothing about the structure of the previous partitioning until the remapping phase, and so, it is required to compute the repartitioning without the benefit of this information.

Also note that the balance results obtained by the Mc-SR and Mc-LMSR algorithms are not always within 5% of the ideal. This is because it was not always possible to balance the particle-tracking computation as the particles were initially injected into the mesh at a single point. Only as particles spread out was a balanced partitioning for the second phase obtainable. This effect is quite significant on the 64-processor results. Here, in fact, the Mc-SR and Mc-LMSR algorithms failed to balance

Table III. Serial and parallel run times of static multi-constraint graph partitioners for a three-constraint problem on *mrng*1. The entry in the column labeled k indicates the number of subdomains that were computed by both the serial and the parallel algorithms, as well as the number of processors that was used to compute each partitioning in the case of the parallel run times. Note that a single processor was used to find the serial times regardless of the value of k.

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

either phase of the computation. This is because these schemes try to minimize the average of the load imbalances. In this case, they traded off good load balance in the first phase of the computation in order to improve the load balance of the second phase.

4.4. Run time results

Table III compares the run times of the parallel static multi-constraint graph-partitioning algorithm with the serial multi-constraint algorithm implemented in the METIS library [22] for *mrng*1. The entry in the column labeled k indicates the number of subdomains that were computed by both the serial and the parallel algorithms, as well as the number of processors that was used to compute each partitioning in the case of the parallel run times. (Note that a single processor was used to find the serial times regardless of the value of k.) 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.

Table IV gives selected run time results and efficiencies of our parallel static multi-constraint graph-partitioning algorithm on up to 128 processors. Table IV shows that our algorithm is very fast, as it is able to compute a three-constraint 128-way partitioning of a 7.5 million vertex graph in about 7 s on 128 processors of a Cray T3E. It also shows that our parallel algorithm requires 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 *mrng*2 (with 1 million vertices) on eight processors is similar to that of partitioning *mrng*3 (4 million vertices) on 32 processors and *mrng*4 (7.5 million vertices) on 64 processors.

Note that the run times for the Mc-LMSR algorithm are close to those required by the static algorithm, and are usually a bit faster. Purely local coarsening speeds up the Mc-LMSR algorithm compared to the static partitioner during the coarsening phase. However, partition remapping and the additional complexity of the refinement algorithm may increase its run time slightly during the uncoarsening phase. The big advantage of the dynamic partitioner is that the initial distribution of the graph across the processors is extremely good, as this distribution was computed either by a static

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

	8 processors		16 processors		32 processors		64 processors		128 processors	
Graph	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 V. Parallel run times of the single-constraint graph partitioner implemented in PARMETIS.

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

partitioner or else by a previous iteration of a dynamic repartitioner. This distribution increases the locality of data accesses and minimizes the inter-processor communications.

Table V gives run time results for the parallel single-constraint graph-partitioning algorithm implemented as ParMETIS_PartKway in the PARMETIS library [21] on some of the same graphs used for our experiments. Comparing Tables IV and V shows that computing a three-constraint partitioning takes about twice as long as computing a single-constraint partitioning. For example, it takes 9.3 s to compute a three-constraint partitioning and 4.8 s 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. This is because the multi-constraint algorithm is more computationally intensive than the single-constraint algorithm, as multiple (not single) weights must be computed regularly.

4.5. Parallel efficiency

Table IV gives relative parallel efficiencies of our static 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. Table IV shows that the multi-constraint 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 numbers of processors.

The isoefficiency function for our parallel multi-constraint graph partitioners is $O(p^2 \log p)$. Therefore, in order to maintain a constant efficiency when doubling the number of processors, the size of the data should be increased by a little more than four times. Since *mrng*3 is approximately four times as large as *mrng*2, it is possible to 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 to increase the size of the data by more than four times here to maintain 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 49–45%. 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 (92–85%). 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 minimum 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 different depending on the particulars of the refinement process.

5. CONCLUSIONS

This paper has presented parallel static and dynamic multi-constraint graph-partitioning algorithms for mapping multi-phase computations onto the processors of a high-performance parallel machine. These algorithms are as scalable as the widely used parallel graph-partitioning algorithm implemented in ParMeTiS. Experimental results conducted on a 128-processor Cray T3E show that our parallel algorithms are able to compute balanced partitionings with similar edge-cuts as the serial algorithm very quickly. For example, our static multi-constraint graph partitioner is able to compute a three-constraint 128-way partitioning of a 7.5 million vertex graph in about 7 s on 128 processors of a Cray T3E.

Although most of the experiments presented in this paper are conducted on synthetic graphs, our parallel multi-constraint partitioning algorithms have also been tested on real application graphs. Basermann *et al.* [24] have used these for load balancing multi-phase car crash simulations of an Audi and a BMW in frontal impacts with a wall. The reported results are consistent with the run time, edge-cut, and balance results presented in this paper.

While the experimental results presented in Section 4 (and [24]) are quite good, it is important to note that the effectiveness of our algorithms 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 uncoarsening phase. Our experiments (not presented in this paper) have shown that an initial partitioning that is more than 20% imbalanced for one or more vertex weights is unlikely to be either balanced or improved during 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 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 selection of vertices that can be moved while maintaining all of the balance constraints drops off. Therefore, the quality of the produced partitionings can likewise be poor.

ACKNOWLEDGEMENTS

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.

REFERENCES

1. Watts J, Rieffel M, Taylor S. A load balancing technique for multi-phase computations. *Proceedings of High Performance Computing '97*, 1997; 15–20.
2. Fingberg J, Basermann A, Lonsdale G, Clinckemaillie J, Gratien J, Ducloux R. Dynamic load-balancing for parallel structural mechanics simulations with DRAMA. *Proceedings of ECT2000*, 2000.
3. Karypis G, Kumar V. Multilevel algorithms for multi-constraint graph partitioning. *Proceedings of Supercomputing '98*, 1998.
4. Schloegel K, Karypis G, Kumar V. Graph partitioning for high performance scientific simulations. *CRPC Parallel Computing Handbook*. Morgan Kaufmann, 2001.
5. Walshaw C, Cross M, McManus K. Multiphase mesh partitioning. *Applied Mathematical Modelling* 2000; **25**(2):123–140.
6. Gupta A. Fast and effective algorithms for graph partitioning and sparse matrix reordering. *IBM Journal of Research and Development* 1996; **41**(1/2):171–183.
7. Hendrickson B, Leland R. A multilevel algorithm for partitioning graphs. *Proceedings of Supercomputing '95*, 1995.
8. Karypis G, Kumar V. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing* 1998; **48**(1).
9. Monien B, Preis R, Diekmann R. Quality matching and local improvement for multilevel graph-partitioning. *Technical Report*, University of Paderborn, 1999.
10. Walshaw C, Cross M. Parallel optimisation algorithms for multilevel mesh partitioning. *Parallel Computing* 2000; **26**(12):1635–1660.
11. Karypis G, Kumar V. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing* 1998; **20**(1):359–392.
12. Gabow H. Data structures for weighted matching and nearest common ancestors with linking. *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, 1990; 434–443.
13. Fiduccia C, Mattheyses R. A linear time heuristic for improving network partitions. *Proceedings 19th IEEE Design Automation Conference*, 1982; 175–181.
14. Kernighan B, Lin S. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal* 1970; **49**(2):291–307.
15. Karypis G, Kumar V. Parallel multilevel k-way partitioning scheme for irregular graphs. *SIAM Review* 1999; **41**(2):278–300.
16. Karypis G, Kumar V. A coarse-grain parallel multilevel k-way partitioning algorithm. *Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing*, 1997.
17. Schloegel K. Graph partitioning for emerging scientific simulations. *PhD Thesis*, University of Minnesota, 1999.

18. Oliker L, Biswas R. PLUM: Parallel load balancing for adaptive unstructured meshes. *Journal of Parallel and Distributed Computing* 1998; **52**(2):150–177.
19. Sohn A, Simon H. JOVE: A dynamic load balancing framework for adaptive computations on an SP-2 distributed-memory multiprocessor. *Technical Report 94-60*, Department of Computer and Information Science, NJIT, 1994.
20. Schloegel K, Karypis G, Kumar V. Wavefront diffusion and LMSR: Algorithms for dynamic repartitioning of adaptive meshes. *IEEE Transactions on Parallel and Distributed Systems* 2000, to appear.
21. Karypis G, Schloegel K, Kumar V. ParMeTiS: Parallel graph partitioning and sparse matrix ordering library. *Technical Report*, University of Minnesota, Department of Computer Science and Engineering, 1997.
22. Karypis G, Kumar V. MeTiS: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices, version 4.0. *Technical Report*, University of Minnesota, Department of Computer Science and Engineering, 1999.
23. Alpert C, Huang J, Kahng A. Multilevel circuit partitioning. *Proceedings 34th ACM/IEEE Design Automation Conference*, 1997.
24. Basermann A, Fingberg J, Lonsdale G, Maerten B, Walshaw C. Dynamic multi-partitioning for parallel finite element applications. *ParCo '99*. Submitted.