Appears in the *8th SIAM Conference on Parallel Processing for Scientific Computing*

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

A Coarse-Grain Parallel Formulation of Multilevel k-way Graph Partitioning Algorithm [*]

George Karypis and Vipin Kumar

University of Minnesota, Department of Computer Science, Minneapolis, MN 55455
Army HPC Research Center, Minneapolis, MN 55415

{karypis, kumar}@cs.umn.edu

Abstract

In this paper we present a parallel formulation of a multilevel k-way graph partitioning algorithm, that is particularly suited for message-passing libraries that have high latency. The multilevel k-way partitioning algorithm reduces the size of the graph by successively collapsing vertices and edges (coarsening phase), finds a k-way partitioning of the smaller graph, and then it constructs a k-way partitioning for the original graph by projecting and refining the partition to successively finer graphs (uncoarsening phase). Our algorithm is able to achieve a high degree of concurrency, while maintaining the high quality partitions produced by the serial algorithm.

1 Introduction

Graph partitioning is an important problem that has extensive applications in many areas, including scientific computing, VLSI design, geographical information systems, operation research, and task scheduling. The problem is to partition the vertices of a graph in p roughly equal partitions, such that the number of edges connecting vertices in different partitions is minimized. The graph partitioning problem is NP-complete. However, many algorithms have been developed that find a reasonably good partition. Recently, a number of researchers have investigated a class of algorithms that are based on multilevel graph partitioning that have moderate computational complexity [4, 5, 11, 12, 14, 7, 28, 17, 16]. In these schemes, the original graph is successively coarsened down until it has only a small number of vertices, a partition of this coarsened graph is computed, and then this initial partition is successively

[*]This work was supported by NSF CCR-9423082, by Army Research Office contract DA/DAAH04-95-1-0538, by Army High Performance Computing Research Center cooperative agreement number DAAH04-95-2-0003/contract number DAAH04-95-C-0008, by the IBM Partnership Award, and by the IBM SUR equipment grant. Access to computing facilities was provided by AHPCRC, Minnesota Supercomputer Institute. Related papers are available via WWW at URL: **http://www.cs.umn.edu/˜karypis**

refined by using a Kernighan-Lin type heuristic as it is being projected back to the original graph. Some of these multilevel schemes [4, 14, 17, 16, 18] provide excellent partitions for a wide variety of graphs. These schemes provide significantly better partitions than those provided by spectral partitioning techniques [29], and are generally at least an order of magnitude faster than even the state-of-the art implementation of spectral techniques [3].

Developing parallel graph partitioning algorithms has received a lot of attention [13, 31, 6, 15, 2, 1, 19] due to its extensive applications in many areas. However, most of this work was concentrated on algorithms based on geometric graph partitioning [13, 6], or algorithms that have very high computational requirements, such as spectral bisection [2, 1, 15]. Geometric graph partitioning algorithms tend to be inherently parallel, but often produce significantly worse partitions compared with the multilevel algorithms. Due to the high computational complexity of the underlying serial algorithm, parallel spectral bisection algorithms running even on 128 or 512 processors tend to be slower than a multilevel graph partitioning algorithm running on a single processor. Development of formulations of multilevel graph partitioning schemes is quite challenging. Coarsening requires that nodes connected via edges be merged together. Since the graph is distributed randomly across the processors, parallel coarsening schemes can require a lot of communication [31, 1, 19]. The Kernighan-Lin refinement heuristic and its variant, that are used during the uncoarsening phase, appear serial in nature [8], and previous attempts to parallelize them have had mixed success [8, 6, 19].

Recently, we developed [20] a parallel formulation for the multilevel k-way partitioning algorithm [18]. Our algorithm is able to achieve high degree of concurrency while it maintains the high quality of the partitions produced by the serial multilevel partitioning algorithm. Our parallel formulation on Cray T3D, using Cray's lightweight SHMEM library for communication, produces high quality 128-way partitions on 128 processors in small amount of time. Graphs with under 250,000 vertices are partitioned in less than a second, while graphs with a million vertices require a little over two seconds. Furthermore, the quality of the produced partitions are comparable (edge-cuts within 5%) to those produced by the serial multilevel k-way algorithm, and are significantly better (edge-cuts up to 75% smaller) than those produced by multilevel spectral bisection algorithm. However, this parallel formulation exploits fine grain parallelism, and is not well suited for message passing libraries that have a high message startup overhead.

In this paper we present a coarse-grain parallel formulation of the multilevel k-way partitioning algorithm. This new formulation performs far fewer communication steps, making it suitable for message-passing libraries (and architectures) that have high message startup overhead. Our MPI-based implementation of the new parallel partitioning algorithm is up to 50% faster than the MPI-based implementation of the algorithm presented in [20], while it produces comparable quality partitionings.

2 Multilevel k-way Graph Partitioning

In [18] we presented a k-way graph partitioning algorithm that is based on the multilevel paradigm, whose complexity is linear on the number of vertices in the graph. The basic structure of a multilevel algorithm is illustrated in Figure 1. The graph $G = (V, E)$ is first coarsened down to a small number of vertices, a k-way partition of this much smaller graph is computed (using multilevel recursive bisection [17]), and then this partition is projected back towards the original graph (finer graph), by periodically refining the partition. Since the finer graph has more degrees of freedom, such refinements improve the quality of the partitions. The experiments presented in [18] show that our algorithm produces partitions that are of comparable or better quality than those produced by the multilevel recursive bisection algorithm [17] and significantly better than those produced by the state-of-the art multilevel spectral bisection algorithm [3]. Furthermore, our k-way partitioning algorithm is up to 5 times faster than the multilevel recursive bisection, and up to 150 times faster than multilevel spectral bisection. The run time of our k-way partitioning algorithm is comparable to the run time of geometric recursive bisection algorithms [13, 30, 26, 25, 27] while it produces partitions that are generally 20% better [17]. Note that geometric methods are applicable only if coordinate information for the graph is available.

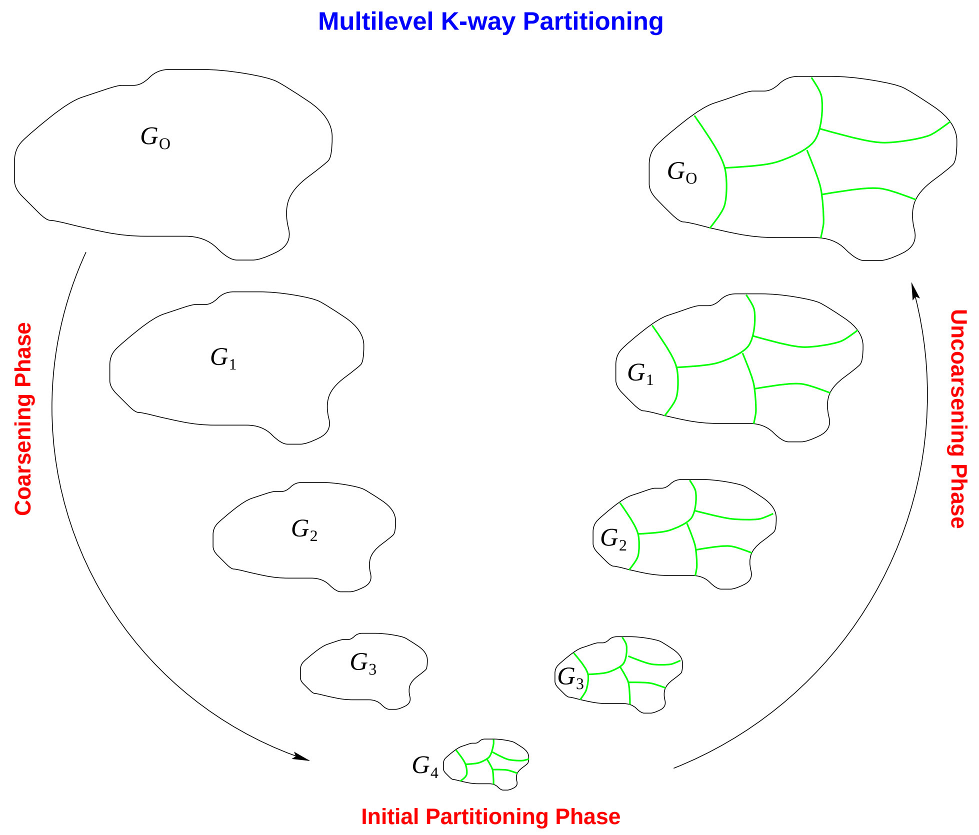


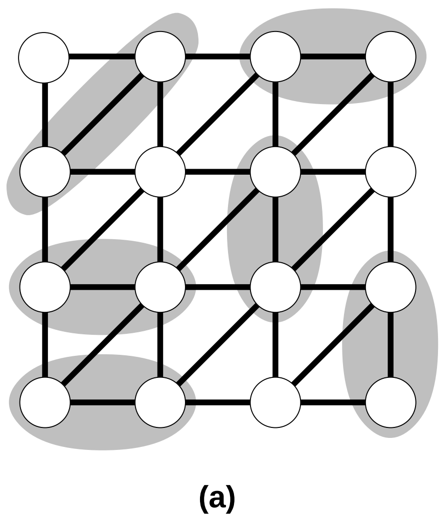
Figure 1: The various phases of the multilevel k-way partitioning algorithm. During the coarsening phase, the size of the graph is successively decreased; during the initial partitioning phase, a k-way partition of the smaller graph is computed; and during the uncoarsening phase, the partitioning is successively refined as it is projected to the larger graphs.

In the rest of this section we briefly describe the various phases of the multilevel algorithm. The reader should refer to [18] for further details.

Coarsening Phase During the coarsening phase, a sequence of smaller graphs $G_i = (V_i, E_i)$, is constructed from the original graph $G_0 = (V_0, E_0)$ such that $|V_i| > |V_{i+1}|$. Graph G_{i+1} is constructed from G_i by finding a maximal matching $M_i \subseteq E_i$ of G_i and collapsing together the vertices that are incident on each edge of the matching. In this process no more than two vertices are collapsed together because a matching of a graph is a set of edges, no two of which are incident on the same vertex. Vertices that are not incident on any edge of the matching are simply copied over to G_{i+1}.

When vertices $v, u \in V_i$ are collapsed to form vertex $w \in V_{i+1}$, the weight of vertex w is set equal to the sum of the weights of vertices v and u, and the edges incident on w is set equal to the union of the edges incident on v and u minus the edge (v, u). For each pair of edges (x, v) and (x, u) (*i.e.*, x is adjacent to both v and u) a single edge (x, w) is created whose weight is set equal to the sum of the weights of these two edges. Thus, during successive coarsening levels, the weight of both vertices and edges increases. The process of coarsening is illustrated in Figure 2. Each vertex and edge in Figure 2(a) has a unit weight. Figure 2(b) shows the coarsened graph that results from the contraction of shaded vertices in Figure 2(a). Numbers on the vertices and edges in Figure 2(b) show their resulting weights.

Maximal matchings can be computed in different ways [17, 18]. The method used to compute the matching greatly affects both the quality of the partition, and the time required during the uncoarsening phase. The matching scheme that we use is called ***heavy-edge matching*** (HEM), and computes a matching M_i, such that the weight of the edges in



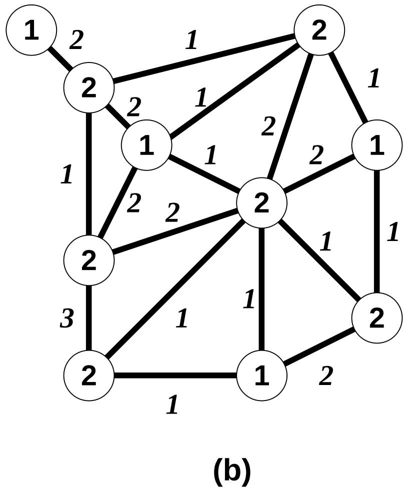


Figure 2: The process of finding a maximal matching and contracting the graph to obtain the next level coarser graph. Note that the vertices and edges of the coarse graph have weights to reflect the number of vertices and edges that are collapsed together.

M_i is high. The heavy-edge matching is computed using a randomized algorithm as follows. The vertices are visited in a random order. However, instead of randomly matching a vertex with one of its adjacent unmatched vertices, HEM matches it with the unmatched vertex that is connected with the heavier edge. As a result, the HEM scheme quickly reduces the sum of the weights of the edges in the coarser. The coarsening phase ends when the coarsest graph G_m has a small number of vertices.

Partitioning Phase The second phase of a multilevel k-way partition algorithm is to compute a k-way partition of the coarse graph $G_m = (V_m, E_m)$ such that each partition contains roughly $|V_0|/k$ vertex weight of the original graph. Since during coarsening, the weights of the vertices and edges of the coarser graph were set to reflect the weights of the vertices and edges of the finer graph, G_m contains sufficient information to intelligently enforce the balanced partition and the minimum edge-cut requirements. In our partitioning algorithm, the k-way partition of G_m is computed using our multilevel recursive bisection algorithm [17]. Our experiments have shown that it produces good initial partitions in relatively small amount of time.

Uncoarsening Phase During the uncoarsening phase, the partitioning of the coarser graph G_m is projected back to the original graph by going through the graphs $G_{m-1}, G_{m-2}, \ldots, G_1$. Since each vertex $u \in V_{i+1}$ contains a distinct subset U of vertices of V_i, the projection of the partition from G_{i+1} to G_i is constructed by simply assigning the vertices in U to the same partition in G_i that vertex u belongs in G_{i+1}.

Even though the partition of G_{i+1} is at a local minima, the projected partition of G_i may not. Since G_i is finer, it has more degrees of freedom that can be used to improve the partition and thus decrease the edge-cut. Hence, it may still be possible to improve the projected partition by local refinement heuristics. For this reason, after projecting a partition, a partition refinement algorithm is used. The basic purpose of a partition refinement algorithm is to select vertices such that when moved from one partition to another the resulting partition has smaller edge-cut and remains balanced (*i.e.*, each partition has the same weight).

The multilevel k-way partitioning algorithm is based on a simplified version of the Kernighan-Lin [22] algorithm, extended to provide k-way partition refinement. This algorithm, is called ***greedy refinement*** (GR). Its complexity is largely independent of the number of partitions being refined. The GR algorithm consists of a number of iterations, and in each iteration all the vertices are checked in a random order to see if they can be moved. Let v be such a vertex. If v is a boundary vertex (*i.e.*, it is connected with a vertex that belongs to an other partition), then v is moved to the partition that leads to the largest reduction in the edge-cut, subject to partition weight constraints. These weight constraints ensure that all partitions have roughly the same weight. If the movement of v cannot achieve any reduction in the edge-cut, it is then moved to the partition (if any) that improves the partition-weight balance but leads to no increase in the edge-cut. The GR algorithm converges after a small number of iterations (within four to eight iterations). If the GR algorithm is not able to enforce the partition balance constraints, an explicit balancing phase is used that moves vertices between partitions even if this movement leads to an increase in the edge-cut.

3 Parallel Multilevel k-way Graph Partitioning

Even though the multilevel partitioning algorithms produce high quality partitions in a very small amount of time, the ability to perform partitioning in parallel is important for many reasons. The amount of memory on serial computers is not enough to allow the partitioning of graphs corresponding to large problems that can now be solved on massively parallel computers and workstation clusters. By performing graph partitioning in parallel, the algorithm can take advantage of the significantly higher amount of memory available in parallel computers. In the context of large-scale finite element simulations, adaptive grid computations dynamically adjust the discretization of the physical domain. Such dynamic adjustments to the grid lead to load imbalances, and thus require repartitioning of the graph for efficient parallel computation. Being able to compute good partitions fast (in parallel) is essential for reducing the overall run time of this type of applications. In some problems, the computational effort in each grid cell changes over time [6]. For example, in many codes that advect particles through a grid, large temporal and spatial variations in particle density can introduce substantial load imbalance. Dynamic repartition of the corresponding vertex-weighted graph is crucial to balance the computation among processors. Furthermore, with recent development of highly parallel formulations of sparse Cholesky factorization algorithms [10, 21, 9, 32], numeric factorization on parallel computers can take much less time than the step for computing a fill-reducing ordering on a serial computer, making that the new bottleneck. For example, on a 1024-processor Cray T3D, some matrices can be factored in less that two seconds using our parallel sparse Cholesky factorization algorithm [21], but serial graph partitioning (needed for computing a fill-reducing ordering) takes two orders of magnitude more time.

In [20] we presented a parallel formulation of the multilevel k-way graph partitioning algorithm described in Section 2. This formulation relies extensively on computing a coloring of the vertices at each successive coarse graph $G_i = (V_i, E_i)$. This coloring is computed using Luby's [24] algorithm, adapted for distributed memory parallel computers.

Consider the graph $G_i = (V_i, E_i)$. Our parallel algorithm constructs the matching in an iterative fashion. During the c^{th} iteration, each processor scans the locally unmatched vertices, and for each vertex of color c, it matches it with one of its unmatched adjacent vertices (if any of them exist) using the heavy-edge heuristic. If this adjacent vertex is stored locally, then the matching is granted right away; otherwise, a match request is sent to the processor that stores this vertex. Next, all processors proceed to service the match requests that they received. They grant the requests that involve vertices that are not matched by local vertices, and notify the appropriate processors. If there are any conflicts, these are broken randomly; however, the use of coloring reduces the number of such conflicts.

During the greedy refinement performed in the uncoarsening phase, the coloring of V_i is used to select the groups of vertices to be moved among partitions in order to improve the quality of the partitioning. In particular, the single phase of the refinement algorithm is broken up into c sub-phases, where c is the number of colors of the graph to be refined. During the i^{th} phase, all the vertices of color i are considered for movement, and the subset of these vertices that lead to a reduction in the edge-cut are moved. Since, the vertices with the same color form an independent set, the total reduction in the edge-cut achieved by moving all vertices at the same time is equal to the sum of the edge-cut reductions achieved by moving these vertices one after the other. After performing this *group* movement, the partitioning information of the vertices adjacent to this group are updated, and the next color is considered.

During the initial partitioning phase, a p-way partition of the graph is computed using a multilevel recursive bisection algorithm. In our algorithm we parallelize this phase by using a parallel algorithm that parallelizes the recursive nature of the algorithm. This is done as follows: The various pieces of the coarse graph are gathered to all the processors using an all-to-all broadcast operation [23]. At this point the processors perform multilevel recursive bisection. However, each processor explores only a single path of the recursive bisection tree. At the end each processor stores the vertices that correspond to its partition of the p-way partition.

In [20] we presented results of the implementation of our parallel partitioning algorithm on Cray T3D. This implementation was done using the one-sided communication primitives available on Cray T3D as part of the SHMEM communication library. Our algorithm is able to achieve high degree of concurrency while it maintains the high quality of the partitions produced by the serial multilevel partitioning algorithm. On 128 processors, graphs with under 250,000 vertices are partitioned in 128 parts in less than a second, while graphs with a million vertices require a little over two seconds. Furthermore, the quality of the produced partitions are comparable (edge-cuts within 5%) to those produced by the serial multilevel k-way algorithm, and are significantly better (edge-cuts up to 75% smaller) than those produced by multilevel spectral bisection algorithm. However, this parallel formulation exploits fine grain parallelism, and is not well suited for message passing libraries that have a high message startup overhead.

3.1 Communication Pattern of the Algorithm

The parallel formulation of the multilevel k-way partitioning algorithm described in Section 3 is made of five different parallel algorithms, namely coloring, matching, contraction, initial partitioning, and refinement. Out of these five algorithms, three of them (coloring, matching, and refinement) have similar communication requirements. The amount of communication performed by each one of these four algorithms depends on the number of interface vertices. For example, during coloring, each processor needs to know the random numbers of the vertices adjacent to the locally stored vertices. Similarly, during refinement, every time a vertex is moved, the adjacent vertices need to be notified to update their partitioning information. Initially, each processor stores n/p vertices and nd/p edges, where d is the average degree of the graph. Thus, the number of interface vertices is at most $O(n/p)$. Since the vertices are initially distributed randomly, these interface vertices are equally distributed among the p processors. Hence, each processor needs to exchange data with $O(n/p^2)$ vertices of each processor. Alternatively, each processor needs to send information for about $O(n/p^2)$ locally stored vertices to each other processor. This can be accomplished by using the all-to-all personalized communication operation [23]. As the size of the coarser graphs successively decreases, the amount of data that needs to be exchanged also decreases. However, each processor still needs to send and receive data from almost all other processors. If t_s is the message startup overhead, then each of these all-to-all personalized communication operations requires pt_s time just due to startup overhead.

The number of all-to-all personalized operations performed for each coarse graph G_i depends on the number of colors c of G_i. In particular, we perform c operations during coloring (one for each color that we compute), $2c$ during matching (two for each color), and $4c$ during refinement (we perform two passes of the refinement algorithm, each consisting of c sub-phases and we need to communicate twice during each sub-phase). Thus, for each graph G_i we perform $7c$ all-to-all personalized operations.

Consider now a graph with a million vertices, that is partitioned on 128 processors, and that the coarsest graph consists of about one thousand vertices. This level of contraction can be achieved by going through about ten coarsening levels. Also, assume that the average number of colors of each coarse graph is around ten (quite reasonable for graphs that correspond to the dual of 3D finite element meshes with tetrahedron elements). Given these parameters, the parallel multilevel k-way partitioning algorithm will perform a total of 700 all-to-all personalized communication operations. On 128 processors, these operations will incur a total of $89600t_s$ overhead due to message startup time.

Our implementation of the multilevel partitioning algorithm on Cray T3D, used the SHMEM communication library. This library has a very small message startup overhead of about 2 microseconds. Thus, for our million node graph on 128 processors, the message startup overhead is a total of about 0.1792 seconds, which is not very significant. In fact, our parallel partitioning algorithm is able to partition a million node graph on 128 processors in around two seconds. However, if instead of using SHMEM, we use Cray's MPI library, then the startup overhead is about 57 microseconds, which brings the amount of time due to message startup overhead to about 5.11 seconds, which is quite significant. In the next section we present our modifications to the algorithm presented in Section 3 such that it reduces the number of all-to-all personalized communication operations; thus, leading to a fast partitioning algorithm even when the message startup time is quite high.

4 A Coarse-Grain Parallel Multilevel k-way Graph Partitioning

Because of the high message startup overhead of current MPI implementations[1] we need to modify the parallel multilevel k-way partitioning algorithm presented in Section 3 so that it performs fewer all-to-all personalized communication operations. From the discussion in Section 3.1, it is clear that one way of doing this is to develop matching and refinement algorithms that do not depend on the number of colors of the graph.

We can easily modify the matching algorithm presented in Section 3 so that it does not require coloring. In fact, the motivation in using a coloring-based matching algorithm was to minimize the number of conflicts; thus, we can use a non-coloring based algorithm at the expense of a higher number of matching conflicts. The new matching algorithms consists of a number of phases. During phase i, each processor scans its local unmatched vertices. For each such vertex v, it matches it with another unmatched vertex u (if such a vertex exists) using the heavy-edge heuristic. If u is stored locally, then the matching is granted right away, otherwise a matching request is issued to the processor that stores u, depending on the ordering of v and u. In particular, if i is odd, a match request is issued only if $v < u$, whereas if i is even, a match request is issued only if $v > u$. This ordering is done to ensure that conflicts can be resolved with a single communication step. Next, every processor processes the matching requests that it received, grants some of these requests by breaking conflicts arbitrarily, and notifies the corresponding processors. The matching algorithm terminates when a large fraction of the vertices has been matched. Our experiments show that for most graphs, very large matchings can be obtained with only four phases. Thus, the number of all-to-all personalized communications are reduced from two times the number of colors to only eight.

However, performing refinement without using coloring is somewhat more difficult. Recall from Section 3 (and from [20]) that by moving a group of vertices of a single color at a time, we were able to ensure that no thrashing occurs during refinement. For example, consider the situation illustrated in Figure 3(a), in which two vertices v and u are connected via an edge and belong to partitions i and j, respectively. Note that if we move vertex v to partition j we reduce the edge-cut by two, and if we move vertex u to partition i we reduce the edge-cut by three. However, as illustrated in Figure 3(b), if we move both vertex v to partition j and vertex u to partition i, then the edge-cut actually increases by five.

The coloring-based refinement algorithm is able to prevent such moves since it only allows concurrent movement of vertices that are not connected (*i.e.*, independent). However, by looking closer at this example we see that an alternate way of preventing such type of movements is to devise a refinement algorithm that does not concurrently move vertices between the same partitions. That is, during each refinement step, for any pair of partitions i and j, it should only move vertices in one direction, *i.e.*, it should move vertices only from partition i (j) to partition j (i). In particular, each phase of the new refinement algorithm consists of only two sub-phases. In the first sub-phase, the group of vertices to be moved is selected so that vertices move from lower- to higher-numbered partitions, and during the second sub-phase, vertices move in the opposite direction. Thus, the new refinement algorithm reduces the number of all-to-all personalized communication operations that are required in each refinement phase from two times the

[1] As MPI becomes more wide-spread the message startup overhead of future MPI implementations will decrease and becomes closer to that of the underlying hardware.

number of colors to four.

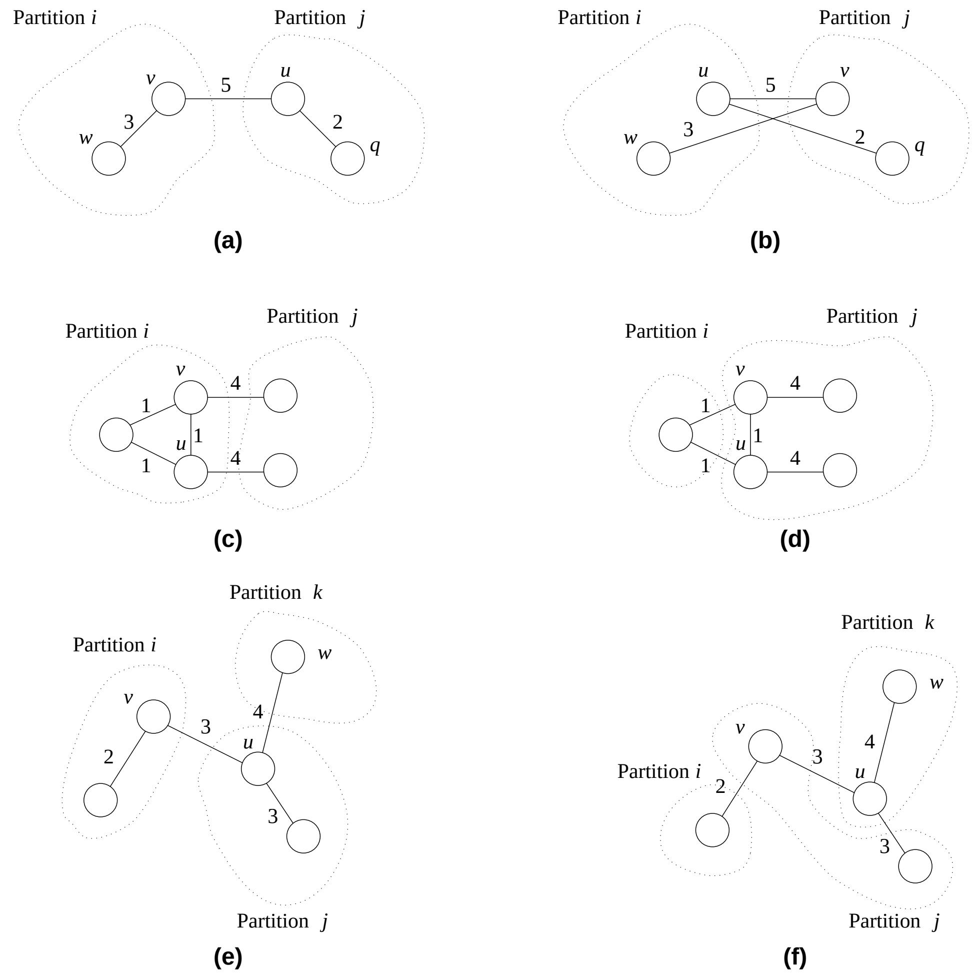


Figure 3: Moves that can be performed concurrently and their properties.

Note that this new refinement scheme allows vertices that are connected and belong to the same partition to be moved concurrently. For example, consider the example illustrated in Figure 3(c), in which vertices v and u, both of them belonging to partition i, are moved concurrently to partition j, since each such move individually leads to a reduction in the edge-cut. However, this type of moves will never lead to thrashing. In fact, the reduction in the edge-cut obtained by concurrently moving connected vertices from the same partition, is at least as high as the sum of the edge-cut reductions of each individual move. This is illustrated by the example in Figure 3(d) [19]. Thus, the coloring-based refinement algorithm was in essence too restrictive while selecting vertices for movement. However, by using the new refinement scheme, there are certain type of moves that may potentially lead to thrashing. Consider the example shown in Figure 3(e), in which vertex u is connected to vertices v and w each belonging to a different partition. If vertex v is moved to partition j the edge-cut reduces by one, and if vertex u moves to partition k the edge-cut reduces by one. However, as illustrated in Figure 3(f), if both moves take place concurrently, then the edge-cut actually increases by one. Fortunately, there are not many vertices that can lead to this type of movement. This is because, this type of moves can only happen among a sequence of vertices that are connected via path and they are interface vertices to multiple domains.

The number of all-to-all personalized communication operations required for each graph G_i by the new matching and refinement algorithms is now only 16 (eight for matching and eight for refinement, assuming we perform two passes of the refinement algorithm). Thus, the total message startup overhead for our one million node graph (described in Section 3.1) on 128 processors, is now only $20480 t_s$ which translates to 1.17 seconds, better than one fourth of the overhead incurred by the coloring-based algorithm.

The coarse-grained parallel multilevel k-way partitioning algorithm has a number of enhancements over the parallel algorithm presented in [20] that both improve its performance as well as extend its functionality.

As the the size of the successively coarser graphs decreases, the amount of time required to generate the next level coarser graphs is dominated by the communication overheads. This is because, the graphs become too small and the message startup overheads dominate the communication time. At this point, the overall amount of time required to generate the remaining coarse graphs as well as the amount of time spent in refining them, will decrease if the work associated with that is assigned to fewer processors. The coarse-grain parallel algorithm performs such type of graph foldings. In particular, as the coarsening progresses, the size of the coarse graph is monitored, and if it falls bellow a certain threshold, it is then folded to only half the processors. Now these processors perform any subsequent coarsening (and refinement during the uncoarsening phase). This folding of the graph to fewer processors is repeated again if necessary. Our experiments have shown that this successive folding of the graphs to fewer processors improves the overall run-time of the partitioning algorithm. The size of the graph after which folding is triggered depends on the characteristics of the underlying interconnection network. If the message startup overhead is very small, then smaller graphs will trigger a folding, whereas, if the message startup time is high, a larger graph will be required.

Our implementation of this coarse-grain algorithm is memory efficient. In particular, each processor requires memory proportional to the size of the locally stored portion of the graph, *i.e.*, $O(n/p)$, where n is the number of vertices in the graph. This is achieved by performing a relabeling of the locally stored graph.

Partition Refinement The coarse-grain parallel graph partitioner can also be used to refine existing partitions or to repartition graphs that have been adaptively refined. In either case, since the initial graph distribution cuts relatively few edges, a local matching algorithm is used during the coarsening phase. This matching algorithm, only matches vertices that already belong to the same processor. Thus, successively coarser graphs are generated with very little communication (communication is still required to label the interface vertices). Also, during the uncoarsening phase, the refinement algorithm incurs very small communication overheads, since there are few interface vertices.

5 Experimental Results

We have tested our coarse-grain parallel multilevel k-way partitioning algorithm on a Cray T3D with 128 processors. Each processor on the T3D is a 150Mhz Dec Alpha (EV4). The processors are interconnected via a three dimensional torus network that has a peak unidirectional bandwidth of 150Bytes per second, and a small latency. We used Cray's MPI library for communication. Cray's MPI achieves a peak bandwidth of 45MBytes and an effective startup time of 57 microseconds.

We evaluated the performance of our parallel multilevel k-way graph partitioning algorithm on three small to medium size graphs arising in finite element computations. The characteristics of these graphs are described in Table 1.

Graph Name	No. of Vertices	No. of Edges	Description
AUTO	448695	3314611	3D Finite element mesh
MDUAL	258569	513132	Dual of a 3D Finite element mesh
MDUAL2	988605	1947069	Dual of a 3D Finite element mesh

Table 1: Various graphs used in evaluating the parallel multilevel k-way graph partitioning algorithm.

Table 2 shows the performance of various implementations of the multilevel k-way partitioning algorithm. The first two subtables show the performance of the coarse-grain and SHMEM-based parallel partitioning algorithms, respec-

Coarse-Grain Parallel Partitioner								
	16-way		32-way		64-way		128-way	
Graph Name	EdgeCut	Time	EdgeCut	Time	EdgeCut	Time	EdgeCut	Time
AUTO	91640	12.203	138983	7.929	196997	5.422	265573	4.545
MDUAL	13584	4.045	19624	2.897	25780	2.578	35130	3.055
MDUAL2	24449	14.497	36220	8.572	50816	5.725	70128	4.604
SHMEM-Based Parallel Partitioner								
	16-way		32-way		64-way		128-way	
Graph Name	EdgeCut	Time	EdgeCut	Time	EdgeCut	Time	EdgeCut	Time
AUTO	91540	8.384	139760	5.071	197166	3.255	264662	2.215
MDUAL	13144	2.637	20004	1.600	25575	1.058	35457	0.795
MDUAL2	24800	10.241	36227	5.778	50114	3.442	71355	2.250
Coarse-Grain Parallel Multilevel Refinement								
	16-way		32-way		64-way		128-way	
Graph Name	EdgeCut	Time	EdgeCut	Time	EdgeCut	Time	EdgeCut	Time
AUTO	90073	2.843	134524	1.770	189638	1.045	247458	0.750
MDUAL	13271	0.931	18322	0.630	24184	0.487	33374	0.417
MDUAL2	22816	2.819	33074	1.682	46058	1.023	64941	0.753
Serial run-times on an SGI workstation, R4400 @150MHZ								
	16-way		32-way		64-way		128-way	
Graph Name	EdgeCut	Time	EdgeCut	Time	EdgeCut	Time	EdgeCut	Time
AUTO	88125	48.490	135629	49.880	190508	51.640	259948	54.610
MDUAL	13688	14.840	20715	15.890	25946	16.560	34235	18.790
MDUAL2	23891	74.050	34144	76.800	47628	76.910	67364	79.380

Table 2: The performance of different implementations of multilevel k-way partitioning algorithm. This table shows the performance of the MPI- and SHMEM-based parallel algorithm, of the coarse-grain parallel multilevel refinement algorithm, and of the serial algorithm on an SGI workstation. In the case of the results of the parallel algorithms, for each graph, the performance is shown for 16-, 32-, 64-, and 128-way partitions on 16, 32, 64, and 128 processors, respectively. All times are in seconds.

tively. Comparing the two subtables we see that both implementations are able to produce partitions of comparable quality. However, as expected, the coarse-grain implementation is somewhat slower. In particular, on 16 processors, the coarse-grain algorithm requires 40% to 50% more time than the SHMEM-based algorithm. This relative difference slowly increases with the number of processors, and on 128 processors the coarse-grain algorithm is about two times slower than the SHMEM-based algorithm for the two larger graphs. Note that for MDUAL the coarse-grain algorithm is about four times slower. This is because the graph is too small for 128 processors, and the overall runtime is dominated by communication overhead, particularly due to message startup overheads (as discussed in Section 3.1). However, considering the significantly higher message startup overhead and the smaller peak bandwidth of the MPI implementation, the performance achieved by the coarse-grain algorithm is quite reasonable. Also, because the coarse-grain implementation is memory efficient, this increases the amount of time spent in the algorithm to set-up the appropriate data structures.

The third subtable in Table 2 shows the performance achieved by the coarse-grain parallel multilevel refinement algorithm. These results were obtained by using as the initial graph distribution, the partitioning obtained by the parallel multilevel k-way partitioning algorithm. We then performed local coarsening and refinement as discussed in Section 4. Note that in this case, we did not perform initial partitioning, as we simply inherited the original partitioning of the graph. Comparing the first and the third subtables we see that the solution quality improves by about 5% to 10%; indicating that this parallel multilevel refinement algorithm can be used effectively to further refine existing partitions. For each case, the run-times in the third subtable are much smaller than those in the first subtable. This shows that repartitioning of adaptive grids can be done quite quickly. Also note that the relative speedups (with respect to 16 processors) are consistently better for the coarse-grain parallel multilevel refinement algorithm than for the SHMEM-based parallel partitioner. This shows that the coarse-grain parallel multilevel refinement algorithm is highly scalable especially in the context of adaptive grid computations.

References

[1] Stephen T. Barnard. Pmrsb: Parallel multilevel recursive spectral bisection. In *Supercomputing 1995*, 1995.

[2] Stephen T. Barnard and Horst Simon. A parallel implementation of multilevel recursive spectral bisection for application to adaptive unstructured meshes. In *Proceedings of the seventh SIAM conference on Parallel Processing for Scientific Computing*, pages 627–632, 1995.

[3] Stephen T. Barnard and Horst D. Simon. A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. In *Proceedings of the sixth SIAM conference on Parallel Processing for Scientific Computing*, pages 711–718, 1993.

[4] T. Bui and C. Jones. A heuristic for reducing fill in sparse matrix factorization. In *6th SIAM Conf. Parallel Processing for Scientific Computing*, pages 445–452, 1993.

[5] Chung-Kuan Cheng and Yen-Chuen A. Wei. An improved two-way partitioning algorithm with stable performance. *IEEE Transactions on Computer Aided Design*, 10(12):1502–1511, December 1991.

[6] Pedro Diniz, Steve Plimpton, Bruce Hendrickson, and Robert Leland. Parallel algorithms for dynamically partitioning unstructured grids. In *Proceedings of the seventh SIAM conference on Parallel Processing for Scientific Computing*, pages 615–620, 1995.

[7] J. Garbers, H. J. Promel, and A. Steger. Finding clusters in VLSI circuits. In *Proceedings of IEEE International Conference on Computer Aided Design*, pages 520–523, 1990.

[8] J. R. Gilbert and E. Zmijewski. A parallel graph partitioning algorithm for a message-passing multiprocessor. *International Journal of Parallel Programming*, (16):498–513, 1987.

[9] Anshul Gupta, George Karypis, and Vipin Kumar. Highly scalable parallel algorithms for sparse matrix factorization. Technical Report 94-63, Department of Computer Science, University of Minnesota, Minneapolis, MN, 1994. To appear in *IEEE Transactions on Parallel and Distributed Computing*. Available on WWW at URL http://www.cs.umn.edu/~karypis/papers/sparse-cholesky.ps.

[10] Anshul Gupta and Vipin Kumar. A scalable parallel algorithm for sparse matrix factorization. Technical Report 94-19, Department of Computer Science, University of Minnesota, Minneapolis, MN, 1994. A shorter version appears in Supercomputing '94. TR available in *users/kumar/sparse-cholesky.ps* at anonymous FTP site *ftp.cs.umn.edu*.

[11] Lars Hagen and Andrew Kahng. Fast spectral methods for ratio cut partitioning and clustering. In *Proceedings of IEEE International Conference on Computer Aided Design*, pages 10–13, 1991.

[12] Lars Hagen and Andrew Kahng. A new approach to effective circuit clustering. In *Proceedings of IEEE International Conference on Computer Aided Design*, pages 422–427, 1992.

[13] M. T. Heath and Padma Raghavan. A Cartesian parallel nested dissection algorithm. *SIAM Journal of Matrix Analysis and Applications*, 16(1):235–253, 1995.

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

[15] Zdenek Johan, Kapil K. Mathur, S. Lennart Johnsson, and Thomas J. R. Hughes. Finite element methods on the connection machine cm-5 system. Technical report, Thinking Machines Corporation, 1993.

[16] G. Karypis and V. Kumar. Analysis of multilevel graph partitioning. Technical Report TR 95-037, Department of Computer Science, University of Minnesota, 1995. Also available on WWW at URL http://www.cs.umn.edu/~karypis/papers/mlevel_analysis.ps. A short version appears in Supercomputing 95.

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

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

[19] G. Karypis and V. Kumar. A parallel algorithms for multilevel graph partitioning and sparse matrix ordering. Technical Report TR 95-036, Department of Computer Science, University of Minnesota, 1995. Also available on WWW at URL http://www.cs.umn.edu/~karypis/papers/mlevel_parallel.ps. A short version appears in Intl. Parallel Processing Symposium 1996.

[20] G. Karypis and V. Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. Technical Report TR 96-036, Department of Computer Science, University of Minnesota, 1996. Also available on WWW at URL http://www.cs.umn.edu/~karypis/papers/mlevel_kparallel.ps. A short version appears in Supercomputing 96.

[21] George Karypis and Vipin Kumar. Fast sparse Cholesky factorization on scalable parallel computers. Technical report, Department of Computer Science, University of Minnesota, Minneapolis, MN, 1994. A short version appears in the *Eighth Symposium on the Frontiers of Massively Parallel Computation*, 1995. Available on WWW at URL http://www.cs.umn.edu/~karypis/papers/frontiers95.ps.

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

[23] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin/Cummings Publishing Company, Redwood City, CA, 1994.

[24] Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM Journal on Computing*, 15(4):1036–1053, 1986.

[25] Gary L. Miller, Shang-Hua Teng, W. Thurston, and Stephen A. Vavasis. Automatic mesh partitioning. In A. George, John R. Gilbert, and J. W.-H. Liu, editors, *Sparse Matrix Computations: Graph Theory Issues and Algorithms*. (An IMA Workshop Volume). Springer-Verlag, New York, NY, 1993.

[26] Gary L. Miller, Shang-Hua Teng, and Stephen A. Vavasis. A unified geometric approach to graph separators. In *Proceedings of 31st Annual Symposium on Foundations of Computer Science*, pages 538–547, 1991.

[27] B. Nour-Omid, A. Raefsky, and G. Lyzenga. Solving finite element equations on concurrent computers. In A. K. Noor, editor, *American Soc. Mech. Eng*, pages 291–307, 1986.

[28] R. Ponnusamy, N. Mansour, A. Choudhary, and G. C. Fox. Graph contraction and physical optimization methods: a quality-cost tradeoff for mapping data on parallel computers. In *International Conference of Supercomputing*, 1993.

[29] Alex Pothen, Horst D. Simon, and Kang-Pu Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM Journal of Matrix Analysis and Applications*, 11(3):430–452, 1990.

[30] P. Raghavan. Line and plane separators. Technical Report UIUCDCS-R-93-1794, Department of Computer Science, University of Illinois, Urbana, IL 61801, February 1993.

[31] Padma Raghavan. Parallel ordering using edge contraction. Technical Report CS-95-293, Department of Computer Science, University of Tennessee, 1995.

[32] Edward Rothberg. Performance of panel and block approaches to sparse Cholesky factorization on the iPSC/860 and Paragon multicomputers. In *Proceedings of the 1994 Scalable High Performance Computing Conference*, May 1994.