A short version of this paper appears in Supercomputing 1996

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

Parallel Multilevel k-way Partitioning Scheme for Irregular Graphs *

George Karypis and Vipin Kumar

University of Minnesota, Department of Computer Science
Minneapolis, MN 55455, Technical Report: 96-036

{karypis, kumar}@cs.umn.edu

Last updated on October 24, 1996 at 9:02am

Abstract

In this paper we present a parallel formulation of a multilevel k-way graph partitioning algorithm. The multilevel k-way partitioning algorithm reduces the size of the graph by 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). A key innovative feature of our parallel formulation is that it utilizes graph coloring to effectively parallelize both the coarsening and the refinement during the uncoarsening phase. Our algorithm is able to achieve a high degree of concurrency, while maintaining the high quality partitions produced by the serial algorithm. We test our scheme on a large number of graphs from finite element methods, and transportation domains. For graphs with a million vertices, our parallel formulation produces high quality 128-way partitions on 128 processors in a little over two seconds, on Cray T3D. Thus our parallel algorithm makes it feasible to perform frequent dynamic graph partition in adaptive computations without compromising quality.

Keywords: Parallel Graph Partitioning, Multilevel Partitioning Methods, Spectral Partitioning Methods, Kernighan-Lin Heuristic, Parallel Sparse Matrix Algorithms.

*This work was supported by NSF CCR-9423082 and by Army Research Office contract DA/DAAH04-95-1-0538, and by Army High Performance Computing Research Center under the auspices of the Department of the Army, Army Research Laboratory 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. Access to computing facilities was provided by AHPCRC, Minnesota Supercomputer Institute, Cray Research Inc, and by the Pittsburgh Supercomputing Center. Related papers are available via WWW at URL: **http://www.cs.umn.edu/˜karypis**

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. For example, the solution of a sparse system of linear equations $Ax = b$ via iterative methods on a parallel computer gives rise to a graph partitioning problem. A key step in each iteration of these methods is the multiplication of a sparse matrix and a (dense) vector. A good partition of the graph corresponding to matrix A can significantly reduce the amount of communication in parallel sparse matrix-vector multiplication [27].

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, 13, 14, 16, 7, 32, 21, 20]. 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 refined by using a Kernighan-Lin type heuristic as it is being projected back to the original graph. Some of these multilevel schemes [4, 16, 21, 20, 22] provide excellent partitions for a wide variety of graphs. These schemes provide significantly better partitions than those provided by spectral partitioning techniques [33], and are generally at least an order of magnitude faster than even the state-of-the art implementation of spectral techniques [3]. Despite the small run time of multilevel schemes, it is important to develop highly parallel formulations of these schemes for reasons discussed in Section 3.

Developing parallel graph partitioning algorithms has received a lot of attention [15, 35, 6, 19, 2, 1, 23] due to its extensive applications in many areas. However, most of this work was concentrated on algorithms based on geometric graph partitioning [15, 6], or algorithms that have very high computational requirements, such as spectral bisection [2, 1, 19]. 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 [35, 1, 23]. The Kernighan-Lin refinement heuristic and its variant, that are used during the uncoarsening phase, appear serial in nature [9], and previous attempts to parallelize them have had mixed success [9, 6, 23].

In this paper we present a parallel formulation for the multilevel k-way partitioning algorithm [22]. This formulation is also generally applicable to any multilevel graph partitioning algorithm that does coarsening of the graph and refines the partitions during the uncoarsening phase [21, 3]. A key feature of our parallel formulation is that it utilizes graph coloring to successfully parallelize both the coarsening and the refinement phases. 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. This parallel refinement algorithm can also be used in conjunction with any other parallel graph partitioning algorithm that requires k-way refinement (*e.g.*, [6]) to improve its quality. We test our scheme on a large number of graphs from finite element methods, and transportation domains. Our parallel formulation on Cray T3D 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.

The remainder of the paper is organized as follows. Section 2 briefly describes the serial multilevel k-way partitioning algorithm. Section 3 discusses the importance of developing parallel graph partitioning algorithms. Section 4 details our parallel formulation of the multilevel k-way partitioning algorithm. Section 5 provides a theoretical performance and scalability analysis. Section 6 presents an experimental evaluation of the parallel algorithm and compares its performance to that of the serial algorithm.

2 Multilevel k-way Graph Partitioning

In [22] 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 [21]), 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 [22] show that our algorithm produces partitions that are of comparable or better quality than those produced by the multilevel recursive bisection algorithm [21] 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 [15, 34, 30, 29, 31] while it produces partitions that are generally 20% better [21]. Note that geometric methods are applicable only if coordinate information for the graph is available.

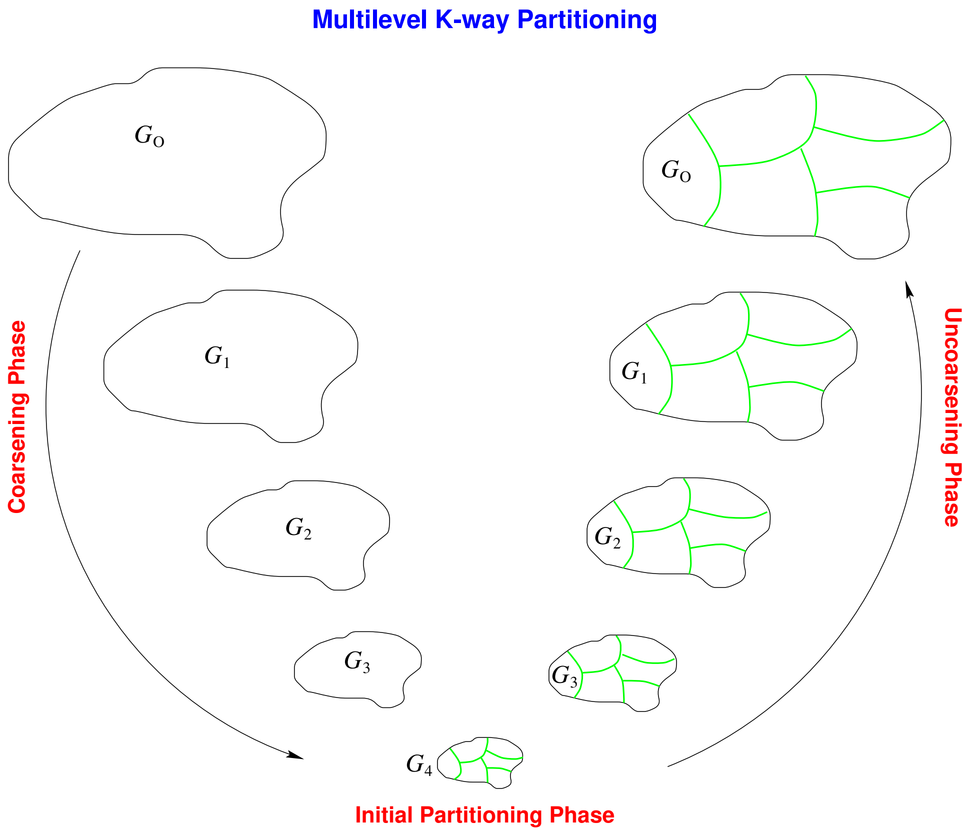


Figure 1: The various phases of the multilevel k-way partitioning algorithm. During the coarsening phase, the size of the graph is successively decreased; during the initial partitioning phase, a k-way partition of the smaller graph is computed; and during the uncoarsening phase, the partitioning is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

The ***k*-way** graph partitioning problem is defined as follows: Given a graph $G = (V, E)$ with $|V| = n$, partition

V into k subsets, $V_1, V_2, \ldots, V_k$ such that $V_i \cap V_j = \emptyset$ for $i \neq j$, $|V_i| = n/k$, and $\bigcup_i V_i = V$, and the number of edges of E whose incident vertices belong to different subsets is minimized. A k-way partitioning of V is commonly represented by a partition vector P of length n, such that for every vertex $v \in V$, $P[v]$ is an integer between 1 and k, indicating the partition at which vertex v belongs. Given a partitioning P, the number of edges whose incident vertices belong to different subsets is called the ***edge-cut*** of the partitioning.

Consider a weighted graph $G_0 = (V_0, E_0)$, with weights both on vertices and edges. A multilevel k-way partitioning algorithm works as follows:

Coarsening Phase The graph G_0 is transformed into a sequence of smaller graphs $G_1, G_2, \ldots, G_m$ such that $|V_0| > |V_1| > |V_2| > \cdots > |V_m|$.

Partitioning Phase A k-way partitioning P_m of the graph $G_m = (V_m, E_m)$ is computed that partitions V_m into k partitions, each containing roughly $|V_0|/k$ vertices of G_0.

Uncoarsening Phase The partitioning P_m of G_m is projected back to G_0 by going through intermediate partitioning $P_{m-1}, P_{m-2}, \ldots, P_1, P_0$.

In the rest of this section we briefly describe the various phases of the multilevel algorithm. The reader should refer to [22] for further details.

2.1 Coarsening Phase

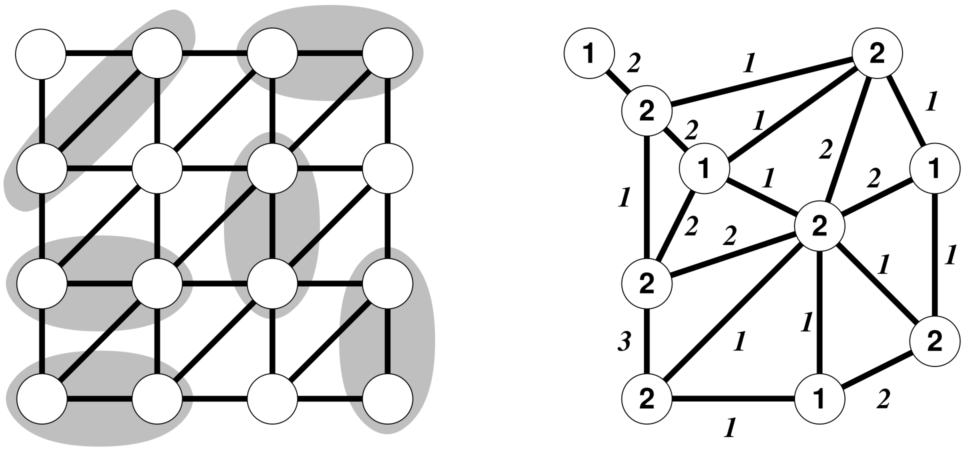
During the coarsening phase, a sequence of smaller graphs $G_i = (V_i, E_i)$, is constructed from the original graph $G_0 = (V_0, E_0)$ such that $|V_i| > |V_{i+1}|$. Graph G_{i+1} is constructed from G_i by finding a maximal matching $M_i \subseteq E_i$ of G_i and collapsing together the vertices that 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 [21, 22]. 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 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.

2.2 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



(a) **(b)**

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.

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 [21]. Our experiments have shown that it produces good initial partitions in relatively small amount of time.

2.3 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 [26] 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.

Key to the GR refinement algorithm is the concept of the decrease in the edge-cut achieved by moving a vertex from one partition to another (***gain***). Consider the graph $G_i = (V_i, E_i)$. For each vertex $v \in V_i$ we define the ***neighborhood*** $N(v)$ of v to be the union of the partitions that the vertices adjacent to v belong to. During refinement, v can move to any of the partitions in $N(v)$. For each vertex v we compute the gains of moving v to one of its neighbor partitions. In particular, for every $b \in N(v)$ we compute the ***external degree*** of v associated with b, $ED[v]_b$ as the sum of the weights of the edges (v, u) such that u belongs to the b part. Also we compute the ***internal degree*** of v, $ID[v]$ as the sum of the weights of the edges (v, u) such that u belongs to the same partition as v. Given these definitions, the gain of moving vertex v to partition $b \in N(v)$ is $ED[v]_b - ID[v]$.

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.*, $N(v)$ is not empty), then v is moved to the partition that leads to the largest reduction in the edge-cut (*i.e.*, the partition with the largest positive gain), 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. After moving vertex v, the algorithm updates the internal and external degrees of the vertices adjacent to v to reflect the change in the partition. 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 Need for Parallel 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 [12, 25, 11, 36], 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 [25], but serial graph partitioning (needed for computing a fill-reducing ordering) takes two orders of magnitude more time.

4 Parallel Formulation

Developing a highly parallel formulation for the multilevel k-way partitioning algorithm is particularly difficult because both the task of computing a maximal matching during the coarsening phase, and the task of refining the partition during the uncoarsening phase appear to be quite serial in nature.

Out of the three phases of the multilevel k-way partitioning algorithm described in Section 2, the coarsening and the uncoarsening phases require the bulk of the computation (over 95%). Hence, it is critical for any efficient parallel formulation of the multilevel k-way partitioning algorithm to successfully parallelize these two phases. Recall that during the coarsening phase (Section 2.1), a matching of the edges is computed, and it is used to contract the graph. One possible way of computing the matching in parallel is to have each processor only compute matchings between the vertices that it stores locally, and use these local matchings to contract the graph. Since each pair of matched vertices resides on the same processor, this approach requires no communication during the contraction step. This approach works well as long as each processor stores relatively well connected portions of the entire graph. In particular, if the graph was distributed among the processors in a partitioned fashioned, then this approach would have worked extremely well. This is not a realistic assumption in most cases, since finding a good partition of the graph is the problem we are trying to solve by the multilevel k-way partitioner. Nevertheless, this approach of *local matchings* can work reasonably well when the number of processors used is small relative to the size of the graph and the average degree of the graph is relatively high. The reason is that even a random partition of a graph among a small number of processors can leave many connected components at each processor. Our earlier work on parallelizing the multilevel recursive bisection algorithm [23] used a two-dimensional distribution of the graph, which required the vertices of the graph to be partitioned only among $\sqrt{p}$ processors. Hence, this graph distribution allowed a moderate amount of coarsening even by using purely local matchings. An alternate approach is to allow vertices belonging to different processors to be matched together. Compared to local matching schemes, this type of matching provides matchings of better quality, and its ability to contract the graph does not depend on the number of processors, or the existence of a good pre-partition. However, this *global matching* significantly complicates the parallel formulation because it requires a distributed matching algorithm. For example, if vertices v and u are located in two different processors P_1 and P_2, then on P_1 vertex v might be matched to u, while on P_2 vertex u may be matched to a different vertex w. Furthermore, another processor P_3 may match its vertex z to vertex u as well. Any correct and usable distributed matching algorithm must resolve both of these conflicts efficiently. Note that since pairs of vertices that are contracted together can reside on different processors, a global communication is required when the contracted graph is constructed.

During the uncoarsening phase, the k-way partition is iteratively refined as it is projected to successively finer graphs. The serial algorithm scans the vertices and moves any vertices that lead to a reduction in the edge-cut. Any parallel formulation of this algorithm will need to move a group of vertices at a time in order to speedup the refinement process. This group of vertices needs to be carefully selected so that every vertex in the group contributes to the reduction in the edge-cut. For example, it is possible that processor P_i decides to move a set of vertices S_i to processor P_j to reduce the edge-cut because the vertices in S_i are connected to a set of vertices T that are located on processor P_j. But, in order for the edge-cut to improve by moving the vertices in S_i, the vertices in T must not move. However, while P_i selects S_i, processor P_j may decide to move some or all the vertices in T to some other processor. Consequently, when both sets of vertices are moved by P_i and P_j, the edge-cut may not improve; and it may even get worse. Clearly, the group selection algorithm must eliminate this type of unnecessary vertex movements. One possible way of performing the k-way refinement is to pairwise refine partitions [6]. That is, assuming that we have four partitions all having common boundaries, we do a 4-way refinement by performing the following 2-way refinements: (1,2), (3,4), (1,3), (2,4), (1,4), (2,3). Since we have a total of four processors, two of these 2-way refinements can go on at the same time. The pairs that can be refined concurrently are determined by a matching of the processor graph [6]. However, this parallel refinement algorithm restricts the type of vertex movements that can be performed in each step. Hence, it lacks the global view that is available in the serial refinement algorithm, in which each vertex is free to move to the partition that leads to the maximum reduction in the edge-cut.

We have developed highly parallel formulations for all three phases of the multilevel k-way graph partitioning algorithm. Our formulation utilizes graph coloring to eliminate conflicts in the computation of global matching in the coarsening phase and to eliminate unnecessary vertex movement in our parallel variation of the Kernighan-Lin refinement in the uncoarsening phases. We also exploit the task-level parallelism of the initial graph partitioning algorithm to further reduce the already small run time of this phase.

Let p be the number of processors used to compute a p-way partition of the graph $G = (V, E)$. G is initially distributed among the processors using a one-dimensional distribution, so that each processor receives n/p vertices and their adjacency lists. At the end of the algorithm, a partition number is assigned to each vertex of the graph. In the next sections we describe our parallel formulations for the three phases of the multilevel k-way partitioning algorithm described in Section 2.

4.1 Computing a Coloring of a Graph

A coloring of a graph $G = (V, E)$ assigns colors to the vertices of G so that adjacent vertices have different color. We like to find a coloring such that the number of distinct colors used is small. Our parallel graph coloring algorithm consists of a number of iterations. In each iteration, a maximal independent set of vertices I is selected using a variation of Luby's [28] algorithm. All vertices in this independent set are assigned the same color. Before the next iteration begins, the vertices in I are removed from the graph, and this smaller graph becomes the input graph for the next iteration. A maximal independent set I of a set of vertices S is computed in an incremental fashion using Luby's algorithm as follows. A random number is assigned to each vertex, and if a vertex has a random number that is smaller than all of the random numbers of the adjacent vertices, it is then included in I. Now this process is repeated for the vertices in S that are neither in I nor adjacent to vertices in I, and I is augmented similarly. This incremental augmentation of I ends when no more vertices can be inserted in I. It is shown in [28] that one iteration of Luby's algorithm requires a total of $O(\log |S|)$ such augmentation steps to find a maximal independent set of a set S.

In our implementation of Luby's algorithm, we perform only a single augmentation step to compute the independent set during each iteration. Hence, the independent set computed is not maximal. Even though this leads to an increase in the number of required colors, it significantly reduces the overall run time required to color the graph. Furthermore, this modification does not significantly impact the run times of the coarsening and uncoarsening phases, because the number of colors increases only moderately.

Luby's algorithm can be implemented quite efficiently on a shared memory parallel computer, since for each vertex v, a processor can easily determine if the random value assigned to v is the smaller among all the random values assigned to the adjacent vertices. However, on a distributed memory parallel computer, for each vertex, random values associated with adjacent vertices that are not stored on the same processor needs to be explicitly communicated. In our implementation of Luby's algorithm, prior to performing the coloring in parallel, we perform a *communication setup* phase, in which appropriate data structures are created to facilitate this exchange of random numbers. In particular, we pre-determine which vertices are located on a processor boundary (*i.e.*, a vertex connected with vertices residing on different processors), and which are internal vertices (*i.e.*, vertices that are connected only to vertices on the same processors). These data structures are used in all the phases of our parallel multilevel graph partitioning algorithm.

4.2 Coarsening Phase

Recall from Section 2.1 that during the coarsening phase a sequence $G_1, G_2, \ldots, G_m$ of successively smaller graphs is constructed. Graph G_{i+1} is derived from G_i by finding a maximal matching M_i of G_i and then collapsing the vertices incident on the edges of M_i. Since the matching M_i is a maximal independent set of edges, we can use Luby's parallel algorithm on the dual graph of G_i to compute a global matching in parallel. However, computing a matching using this algorithm can be quite expensive since the dual graph usually has significantly more vertices than G_i, and it is somewhat denser. For this reason, we use a matching algorithm that is based on the coloring of the graph. This coloring algorithm also happens to be essential for parallelizing the partitioning refinement phase.

Our parallel matching algorithm is based on an extension of the serial algorithm and utilizes graph coloring to structure the sequence of computations. Consider the graph $G_i = (V_i, E_i)$ that has been colored using our parallel formulation of Luby's algorithm, and let *Match* be a variable associated with each vertex of the graph, that is initially set to -1. At the end of the computation, the variable *Match* for each vertex v stores the vertex that v is matched to. If v is not matched, then *Match* = v. To simplify the presentation, we first describe the algorithm assuming that the target parallel computer has a shared memory architecture, and later show how this algorithm is implemented on a distributed memory machine.

The matching M_i is constructed in an iterative fashion. During the c^{th} iteration, vertices of color c that have not been matched yet (*i.e.*, *Match* = *-1*) select one of their unmatched neighbors using the heavy-edge heuristic, and modify the *Match* variable of the selected vertex by setting it to their vertex number. Let v be a vertex of color c and (v, u) be the edge that is selected by v. Since the color of u is not c, this vertex will not be selecting a partner vertex at this iteration. However, there is a possibility that another vertex w of color c may select (w, u). Since both vertices v and w perform their selections at the same time, there is no way of preventing that. This is handled as follows. After all vertices of color c select an unmatched neighbor, they synchronize. The vertices of color c that have just selected a neighbor, read the *Match* variable of their selected vertex. If the value read is equal to their vertex number, then their matching was successful, and they set their *Match* variable equal to the selected vertex; otherwise the matching fails, and the vertex remains unmatched. Note that if more than one vertex (*e.g.*, v and w) want to match with the same vertex (*e.g.*, u), only one of the writes in the *Match* variable of the selected vertex will succeed; and this determines which matching survives. However, by using coloring, we restrict which vertices select partner vertices during each iteration; thus, the number of such conflicts is significantly reduced. Also note that even though a vertex of color c may fail to have its matching accepted due to conflicts, this vertex can still be matched during a subsequent iteration corresponding to a different color.

The above algorithm is implemented quite easily on a distributed memory parallel computer as follows. The *writes* into the *Match* variables are gathered all together and are sent to the corresponding processors in a single message. If a processor receives multiple *write* requests for the same vertex, the one that corresponds to the heavier edge is selected. Any ties are broken arbitrarily. Similarly, the *reads* from the *Match* variables are gathered by the processors that store the corresponding variables and they are sent in a single message to the requesting processors. Furthermore, during this *read* operation, the processors who own the *Match* variables also determine if they will be the ones storing the collapsed vertex in G_{i+1}. This is done by using a uniformly distributed random variable. The vertex is kept or given away with the same probability. Our experiments has shown that this simple heuristic leads to a very good load balance.

After a matching M_i is computed, each processor knows how many vertices (and the associated adjacency lists) it needs to send and how many it needs to receive. Each processor then sends and receives these subgraphs, and it forms the next level coarser graph by merging the adjacency lists of the matched vertices. The coarsening process ends when the graph has $O(p)$ vertices.

4.3 Partitioning Phase

During the partitioning phase, a p-way partition of the graph is computed using a recursive bisection algorithm. Since the coarsest graph has only $O(p)$ vertices, this step can be performed serially in $O(p \log p)$ time without significantly affecting the performance of the entire algorithm. Nevertheless, in our algorithm we also 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 [27]. At this point the processors perform recursive bisection using an algorithm that is based on nested dissection [8] and greedy partition refinement. However, as illustrated in Figure 3, 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. Note that after the initial all-to-all broadcast operation, the algorithm proceeds without any further communication.

4.4 Uncoarsening Phase

In the uncoarsening phase, the partition is projected from the coarse graph to the next level finer graph, and it is refined using the greedy refinement algorithm (Section 2.3). Recall that during a single phase of the refinement algorithm the vertices are randomly traversed, and the vertices that lead to a decrease in the edge cut switch partitions. After each such vertex movement, the external degrees of the adjacent vertices are updated to reflect the new partition.

In the parallel formulation of greedy refinement, we retain the spirit of the serial algorithm, but we change the order in which the vertices are traversed to determine if they can be moved to different partitions. 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 (or improve the balance without increasing 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

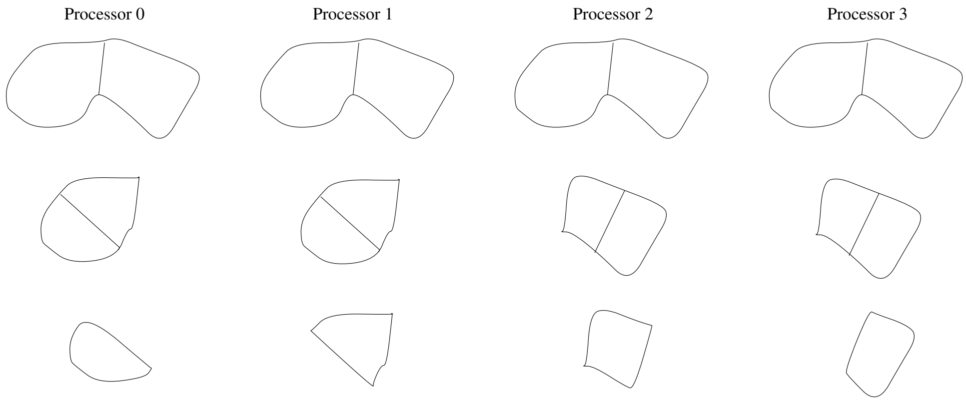


Figure 3: Performing the initial k-way partitioning in parallel. Each processor explores only a single path from the root to the leaves in the recursive bisection tree.

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 external degrees of the vertices adjacent to this group are updated, and the next color is considered.

During the parallel refinement, we can physically move the vertices as they change partitions. That is, each processor initially stores all the vertices of a single part, and as vertices move between partitions during refinement, they can also move between the corresponding processors. However, in the context of multilevel graph partitioning such an approach requires significant communication. This is because for each vertex v in the coarse graph G_i that we move we need to send not only the adjacency list of v but also the adjacency lists of all the vertices collapsed in v for the higher level finer graphs $G_{i-1}, G_{i-2}, \ldots, G_0$. In our parallel refinement algorithm we solve this problem as follows. Vertices do not move from processor to processor, but only the partition number associated with each vertex changes. Since the vertices are initially distributed in a random order, each processor stores vertices that belong to almost all p partitions. This ensures that during refinement each processor will have some boundary vertices that needs to be moved, leading to a generally load balanced computation. Furthermore, this also leads to a simpler implementation of the parallel refinement algorithm, since vertices (and their adjacency lists) do not have to be moved around. Of course, all the vertices are moved to their proper location at the end of the partitioning algorithm, using a single all-to-all personalized communication [27].

The balance conditions are maintained as follows. Initially, each processor knows the weights of all p partitions. During each refinement sub-phase, each processor enforces balance constraints based on these partition weights. For every vertex it decides to moves it locally updates these weights. At the end of each sub-phase, the global partition weights are recomputed, so that each processor knows the exact weights. Even though, this scheme is less exact than the serial balance constraints, our experiments have shown that the hybrid of local and global partition weight constraints is able to produce well balanced partitions.

The above parallel refinement algorithm is highly concurrent, since the number of colors is very small (less than 20 for 3D finite element meshes) while the number of vertices is very large. Furthermore, since both the serial and parallel refinement algorithms are similar in spirit, both exhibit similar partition refinement capabilities. Furthermore, this coloring-based parallel refinement algorithm can be used in any other algorithm that uses Kernighan-Lin-type local refinement.

5 Performance and Scalability Analysis

The parallel formulation of the multilevel k-way partitioning algorithm described in Section 4 is made of five different parallel algorithms, namely coloring, matching, contraction, initial partitioning, and refinement. Out of these five algorithms, four of them (coloring, matching, contraction, and refinement) have similar requirements in terms of computation and communication. For this analysis, we assume that the number of colors needed is a fixed constant irrespective of the size of the graph. This is a reasonable assumption for all graphs that arise in finite element applications.

The amount of computation performed by each one of these four algorithms is proportional to the size of the graph stored locally on each processor. Furthermore, since the size of successively coarser graphs roughly decreases by a factor of two, the computation performed for the original graph dominates the computation performed for the subsequent $O(\log n)$ coarser graphs. Thus, the amount of computation performed is

$$T_{calc} = O\left(\frac{n}{p}\right). \tag{1}$$

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 external degrees. 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 [27], whose complexity is $O(n/p) + O(p)$. The communication complexity over all $O(\log n)$ coarsening levels, is

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

since the size of the graph is successively halved.

Note that for Equation 2 to be valid, the data to be communicated among processor must be roughly equally distributed. Since the graph is randomly distributed, this is a reasonable assumption (otherwise, a somewhat more expensive generalized all-to-all personalized communication [38] is needed). Furthermore, for the successive coarser graphs, the graphs also remains randomly distributed because during matching, decisions regarding where the contracted vertex will reside is done randomly.

In addition to the communication of the interface vertices, both matching and refinement perform additional communication. During matching, a prefix-sum is performed to determine the numbering of the vertices in the coarser graph. The complexity of this operation over all $O(\log n)$ coarsening levels is $O(\log p \log n)$ [27]. During refinement, a reduction of a p-vector is performed to compute the weights of the partitions. Since the size of the vector is equal to p, each such reduction can be done in $O(p)$ time [27]. Thus, the complexity over the all $O(\log n)$ coarsening levels is $O(p \log n)$. Finally, all four algorithms require global synchronizations, whose complexity is $O(\log p \log n)$. However, the complexity of the above communication overheads are subsumed by the complexity of sending information about interface vertices (Equation 2).

During the initial partitioning phase, a graph of size $O(p)$ is partitioned into p partitions using recursive bisection. As described in Section 4.3, the graph is gathered on each processor using an all-to-all broadcast operation [27], whose complexity is $O(p)$. After that each processor performs recursive bisection, but only keeps one of the two bisections. Thus, the computational complexity of the initial partition is $O(p)$.

Thus, from Equations 1 and 2 we have that the parallel run time of the multilevel partitioning algorithm is

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

The run time of any parallel graph partitioning algorithm is bounded from below by the amount of time required to permute a graph given a partition vector. Assuming that the graph is randomly distributed among the processors, this permutation is equivalent to an all-to-all personalized communication of the original graph. The run time of this communication operation is $O(n/p) + O(p)$ [27]. Thus, the run time of our parallel graph partitioning algorithm is only slightly higher (by a factor of $O(\log n)$ in the second term of Equation 3) than this absolute lower bound. Finally, since the sequential complexity of the serial algorithm is $O(n)$, the isoefficiency function [27] of our algorithm is $O(p^2 \log p)$.

6 Experimental Results

We evaluated the performance of our parallel multilevel k-way graph partitioning algorithm on a wide range of graphs arising in different application domains. The characteristics of these graphs are described in Table 1.

We implemented our parallel multilevel algorithm on a 128-processor Cray T3D parallel computer. 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 SHMEM message passing library for communication. In our experimental setup, we obtained a peak bandwidth of 90MBytes and an effective startup time of 4 microseconds.

Since, each processor on the T3D has only 64MBytes of memory, some of the larger graphs could not be partitioned on a single processor. For this reason, we compare the parallel run time on the T3D with the run time of the serial multilevel k-way algorithm running on a SGI Challenge with 0.5GBytes of memory and 150MHz Mips R4400. Even though the R4400 has a peak integer performance that is 10% lower than the Alpha, due to the significantly higher amount of secondary cache available on the SGI machine (1 Mbyte on SGI versus 0 Mbytes on T3D processors), the code running on a single processor T3D is about 20% slower than that running on the SGI. Since the nature of the multilevel algorithm discussed is randomized, we performed all experiments with fixed seed.

Graph Name	No. of Vertices	No. of Edges	Description
144	144649	1074393	3D Finite element mesh
598A	110971	741934	3D Finite element mesh
AUTO	448695	3314611	3D Finite element mesh
BRACK2	62631	366559	3D Finite element mesh
COPTER2	55476	352238	3D Finite element mesh
M14B	214765	1679018	3D Finite element mesh
MAP1	267241	334931	Highway network
MDUAL	258569	513132	Dual of a 3D Finite element mesh
MDUAL2	988605	1947069	Dual of a 3D Finite element mesh
OCEAN	143437	409593	3D Finite element mesh
ROTOR	99617	662431	3D Finite element mesh
WAVE	156317	1059331	3D Finite element mesh

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

Note that the algorithm used for computing the initial partition of the graph in the parallel multilevel algorithm (see Section 4.3) is different than the multilevel recursive bisection used in the serial algorithm. The multilevel algorithm produces significantly better initial partitions than nested dissection but it requires more time. Consequently, the initial partitioning step may become a bottleneck for very large number of processors, particularly for smaller graphs. However, due to the k-way refinement performed in the uncoarsening phase, the final partitions are only slightly worse than those produced by the serial k-way algorithm (that uses the multilevel recursive bisection algorithm for computing initial partitions).

Partition Quality Table 2 shows the quality of the partitions produced by the parallel k-way algorithm as well as the amount of time it took to produce these partitions on a Cray T3D for the problems of Table 1. Partitions in 16, 32, 64, and 128 partitions are shown, each produced on 16, 32, 64, and 128 processors, respectively. Table 3 shows

	16-way		32-way		64-way		128-way	
Graph Name	EdgeCut	Time	EdgeCut	Time	EdgeCut	Time	EdgeCut	Time
144	44742	3.021	65845	1.956	87573	1.270	120365	0.977
598A	31211	2.436	47037	1.557	61225	1.039	90724	0.805
AUTO	91540	8.384	139760	5.071	197166	3.255	264662	2.215
BRACK2	13454	0.858	20675	0.589	30161	0.426	43937	0.389
COPTER2	20677	0.938	30714	0.638	42047	0.475	58470	0.424
M14B	50554	4.455	77944	2.834	108294	1.834	159825	1.383
MAP1	343	0.942	701	0.583	1174	0.398	1956	0.344
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
OCEAN	10392	1.240	16529	0.799	25311	0.551	35846	0.446
ROTOR	25146	1.684	38134	1.128	53547	0.819	78163	0.670
WAVE	49502	1.914	72969	1.245	98572	0.894	131896	0.743

Table 2: The performance of the parallel multilevel k-way partitioning algorithm on Cray T3D. For each graph, the performance is shown for 16-, 32-, 64-, and 128-way partitions on 16, 32, 64, and 128 processors, respectively. The times are in seconds.

	16-way		32-way		64-way		128-way	
Graph Name	EdgeCut	Time	EdgeCut	Time	EdgeCut	Time	EdgeCut	Time
144	42987	12.140	63425	12.900	85967	13.620	116870	15.380
598A	30081	9.230	44604	9.320	62520	10.190	86891	11.050
AUTO	88125	48.490	135629	49.880	190508	51.640	259948	54.610
BRACK2	13539	3.680	20133	4.000	29515	4.520	42775	5.740
COPTER2	20852	3.970	30273	4.510	41672	5.160	56619	5.990
M14B	49029	18.830	75316	19.440	108874	20.560	153048	22.070
MAP1	323	9.580	674	10.020	1104	10.140	1881	11.210
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
OCEAN	10033	7.160	16183	7.650	24483	8.370	34015	9.640
ROTOR	24723	7.140	36396	7.680	52463	8.380	73881	9.530
WAVE	47939	10.850	69370	11.490	95747	12.240	125925	14.100

Table 3: The performance of the serial multilevel k-way partitioning algorithm. For each graph, the performance is shown for 16, 32, 64, and 128-way partitions. The times are in seconds on an SGI Challenge.

the quality of the partitions and the amount of time required by the serial algorithm running on the SGI for the same problems.

The quality of the partitions produced by the parallel algorithm relative to those produced by the serial k-way partitioning algorithm is graphically shown in Figure 4. From this figure we see that the edge-cut produced by the parallel algorithm is quite close to that produced by the serial algorithm. For most graphs, the edge-cut of the parallel algorithm is worse than that of the serial algorithm by at most a factor of 5%, while for some graphs, the parallel algorithm is somewhat better (by 1% to 3%). Since both the coarsening and uncoarsening phases of the parallel algorithm are similar (Sections 4.2 and 4.4), the reason for the small deviation compared to the serial algorithm can be traced back to the use of nested-dissection in the initial partition phase. However, the quality differences can be eliminated if the multilevel bisection is used during the initial partitioning phase.

The quality of the parallel partitioning algorithm relative to the widely used, multilevel spectral bisection (MSB) [3], is shown in Figure 5. The MSB partitions were produced using the state-of-the art MSB algorithm as implemented in the Chaco 2 [17] graph partitioning package. From this figure we see that the quality of our parallel multilevel k-way partitioning algorithm is usually 10% to 20% better than that of MSB, and for some graphs, it is up to 75% better.

Parallel Runtime From Table 2 we can see that the run time of the parallel algorithm is very small. For 9 out of the 12 graphs, the parallel algorithm requires less than one second to produce an 128-way partition on 128 processors. Even for the larger graphs (AUTO with half a million vertices, and MDUAL2 with one million vertices) it requires only 2.2 seconds.

Table 4, analytically shows the amount of time required by the different phases of the parallel graph partitioning

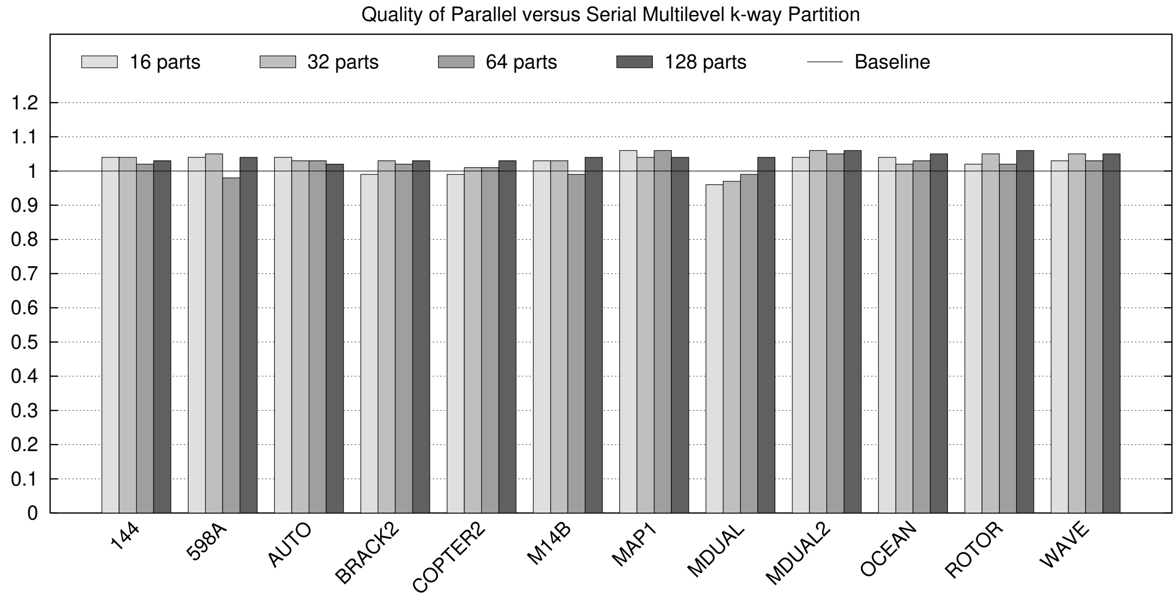


Figure 4: Quality of the partitions produced by the parallel relative to the serial multilevel k-way partitioning algorithm. For each graph, the ratio of the edge-cut of the parallel to that of the serial algorithm is plotted for 16-, 32-, 64-, and 128-way partitions. Bars under the baseline indicate that the parallel algorithm produces partitions with smaller edge-cut than the serial algorithm.

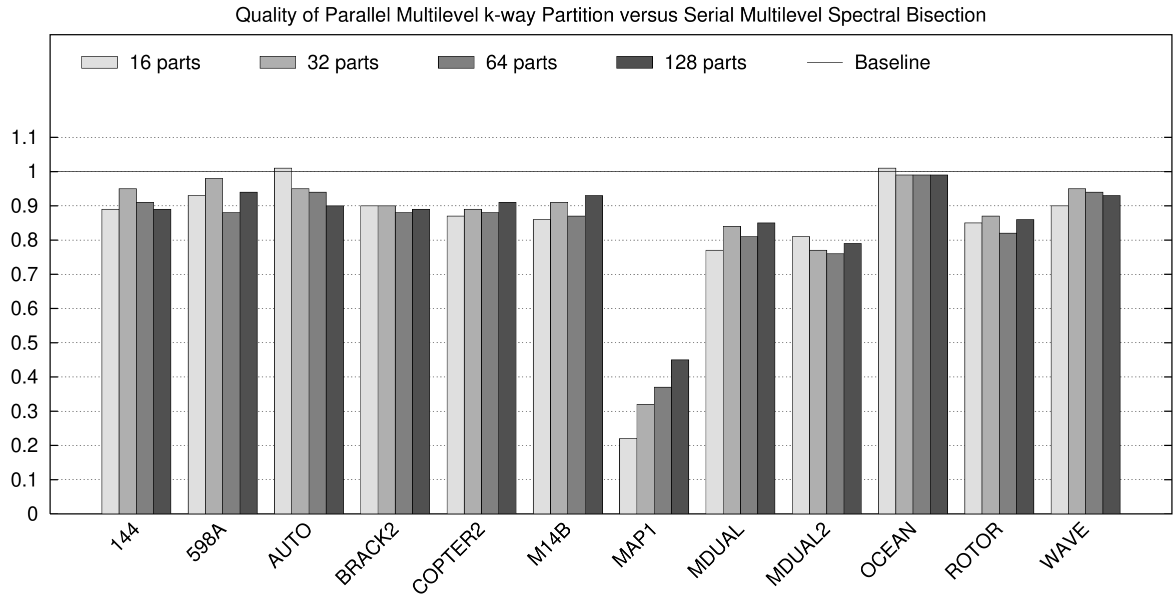


Figure 5: Quality of the partitions produced by the parallel multilevel k-way partitioning algorithm relative to the multilevel spectral bisection (MSB). For each graph, the ratio of the edge-cut of the parallel to that of the serial algorithm is plotted for 16-, 32-, 64-, and 128-way partitions. Bars under the baseline indicate that the parallel algorithm produces partitions with smaller edge-cut than the spectral bisection algorithm.

	AUTO		MDUAL		MDUAL2	
Phase Name	16PEs	128PEs	16PEs	128PEs	16PEs	128PEs
Communication Setup	0.978	0.279	0.290	0.114	1.730	0.386
Graph Coloring	2.480	0.477	0.581	0.102	2.239	0.351
Computing Matching	1.271	0.353	0.458	0.111	1.752	0.385
Graph Contraction	2.115	0.421	0.676	0.122	2.674	0.436
Initial Partition	0.006	0.051	0.009	0.079	0.004	0.098
k-way Refinement	1.534	0.634	0.623	0.267	1.842	0.594
Total Runtime	8.384	2.215	2.637	0.795	10.241	2.250

Table 4: The amount of time (in seconds) required by the different phases of the parallel partitioning algorithm for some graphs, on 16 and 128 processors.

algorithm for some of the graphs of our experimental testbed. Note that during the *communication setup* phase, the processors determine how many interface vertices they need to send and receive, and setup the appropriate data structures for this communication. From this table we see that as the number of processors increase the amount of time required by each phase decreases. The only exception is the initial partitioning phase, for which the time actually increases. This is because, both the size of the coarsest graph and the number of partitions increases with the number of processors. However, the amount of time required by this phase is very small compared to the run time of the entire partitioning algorithm.

The speedup achieved by the parallel algorithm on Cray T3D over the serial algorithm running on SGI is shown in Figure 6. For the smaller graphs, the parallel algorithm achieves a speedup in the range of 14 to 17 on 128 processors, and as the size of the graphs increases the speedup improves to the 20 to 35 range. As discussed earlier, due to architectural differences between Cray T3D and SGI Challenge, the run time of the multilevel partitioning code running on a single processor of the SGI is somewhat smaller than that running on a single processor of the Cray T3D. Thus, the actual speedups (*i.e.*, with respect to the serial algorithm running on a single processor of the Cray T3D) are higher by a factor of about 20%. Furthermore, as discussed in Section 4, the parallel algorithm incurs the additional computational overhead of computing graph coloring during the coarsening phase, an overhead that it is not present in the serial algorithm. In addition to the coloring overhead, the parallel algorithm also requires a communication setup phase that is used to exchange information about the interface vertices. Again, on the serial algorithm, this overhead is not present. For instance, for AUTO, from Table 4 we see that out of the run time of 2.2 seconds on 128 processors, the above two overheads take 0.8 seconds, which is 36% of the total run time. Also note that for MAP1, MDUAL, and OCEAN, for which the above two overheads are smaller (since these graphs have smaller average degrees), they achieve better speedup than other graphs with similar number of vertices.

Experimental Scalability From Table 2, we see that for each graph, the run time of the parallel algorithm decreases as the number of processors and partitions increases. From Table 3, we see that the run times of the serial multilevel k-way partitioning algorithm increases as k increases. Since the asymptotic complexity of the serial algorithm is $O(n)$ [22], this increase in run time is due to an increase in the number of interface vertices that exist as the number of partitions increases. Refining these interface vertices also leads to more work, but does not increase the asymptotic complexity of the algorithm. Evidence of the increased computational requirements during the k-way refinement can also be seen in Table 4. From this table we see that as the number of processors increases, the amount of time required by refinement, decreases at a slower rate than the time required for by either coloring, matching, or contraction. For instance, for MDUAL going from 64 to 128 processors, the run time of matching decreased by 43%, while the run time for refinement decreased only by 19%.

This modest increase of the computational requirements, makes it hard to draw any conclusions about the experimental scalability of the parallel algorithm from the raw parallel run times in Table 2. However, from the serial run times, we know by how much the computational requirements increase as k increases. For this reason, we use the increase in the serial run time to compute the *scaled relative efficiencies* shown in Figure 7 for some graphs. These

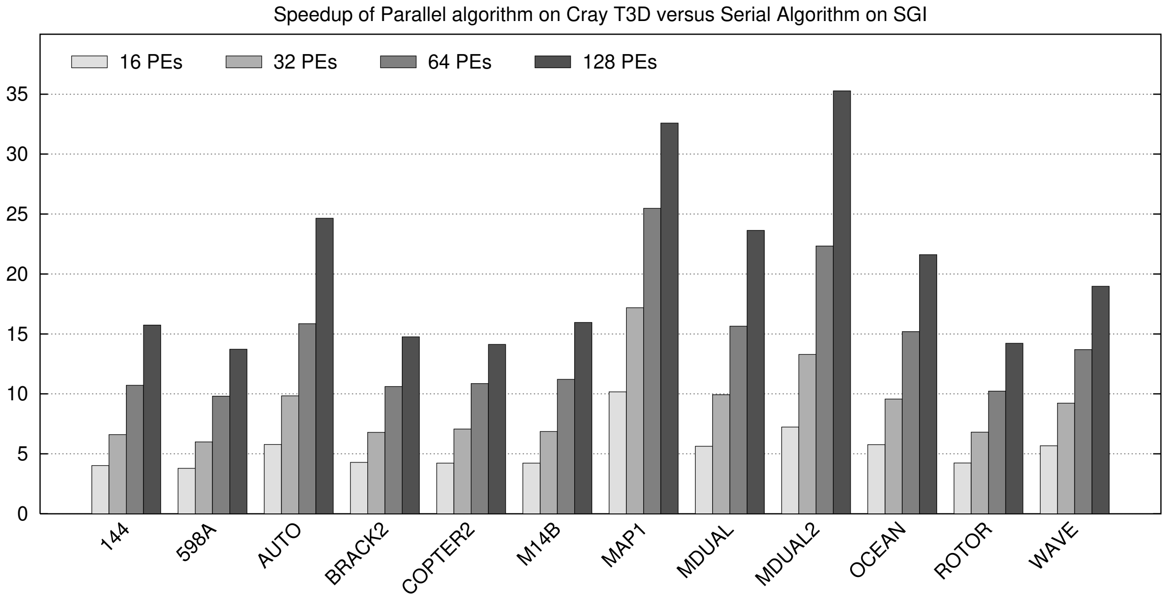


Figure 6: The speedup achieved by the parallel partitioning algorithm running on Cray T3D over the serial algorithm running on SGI. For each graph, the speedup on 16, 32, 64, and 128 processors is shown.

efficiencies are relative to the 16-processor run times, and they are scaled to reflect the increase in the computation. For example, we know from Table 3 that for AUTO, going from 16-way to 128-way partition, the run time increases from 48.49 seconds to 54.61 seconds, a 12.6% increase. To compute the relative speedup of the parallel algorithm on 128-processors relative to that on 16-processors, we multiply the run time on 16-processors (8.38 seconds) by 1.126 (*i.e.*, 12.6% increase in computational requirements), and divide it by the run time on 128-processors (2.21 seconds). This relative speedup is 4.27; thus, yielding a relative efficiency of 0.53 (since $128 = 8 * 16$). From Figure 7 we see that for any graph, as the number of processors increases the efficiency decreases. This is expected for any non-trivial parallel algorithm, since the communication overhead increases, as the processors increases. Similarly, as the size of the graphs increases, the achieved efficiency improves because the communication overhead increases slower than the amount of computation performed.

From the analysis in Section 5, we have shown that the isoefficiency function of our parallel algorithm is $O(p^2 \log p)$. That is, in order to maintain a fixed efficiency, the graph size should increase as $O(p^2 \log p)$ [27]. For example, if we double the number of processors, then we need to increase the size of the graph by a factor little over 4 in order to achieve the same efficiency. In order to experimentally evaluate the scalability of the algorithm, we use the speedup obtained by the parallel algorithm over the serial algorithm running on the SGI. Because of the differences between the serial and parallel algorithm discussed earlier (additional use of coloring by the parallel algorithm), it is important to compare the efficiencies achieved on graphs that have similar structure, so they will lead to similar coloring overheads. Among the graphs in our experimental testbed (Table 1), the following pairs of graphs 144 with AUTO, and MDUAL with MDUAL2 require the same number of colors and they have appropriate relative sizes. AUTO is about 3.1 times larger than 144, while MDUAL2 is about 3.83 times larger than MDUAL. From Figure 6 we see that the speedup achieved by AUTO (and MDUAL2) on 32, 64, and 128 processors are comparable to the speedup achieved by 144 (and MDUAL) on 16, 32, and 64 processors, respectively. Thus, the experiments confirm that the isoefficiency function of our parallel graph partitioning algorithm is $O(p^2 \log p)$.

Effects of Initial Graph Distribution The experiments shown in Table 2 were performed by initially distributing the graphs to the processors in a block distribution. That is as the graphs were read from the file, consecutive n/p ver-

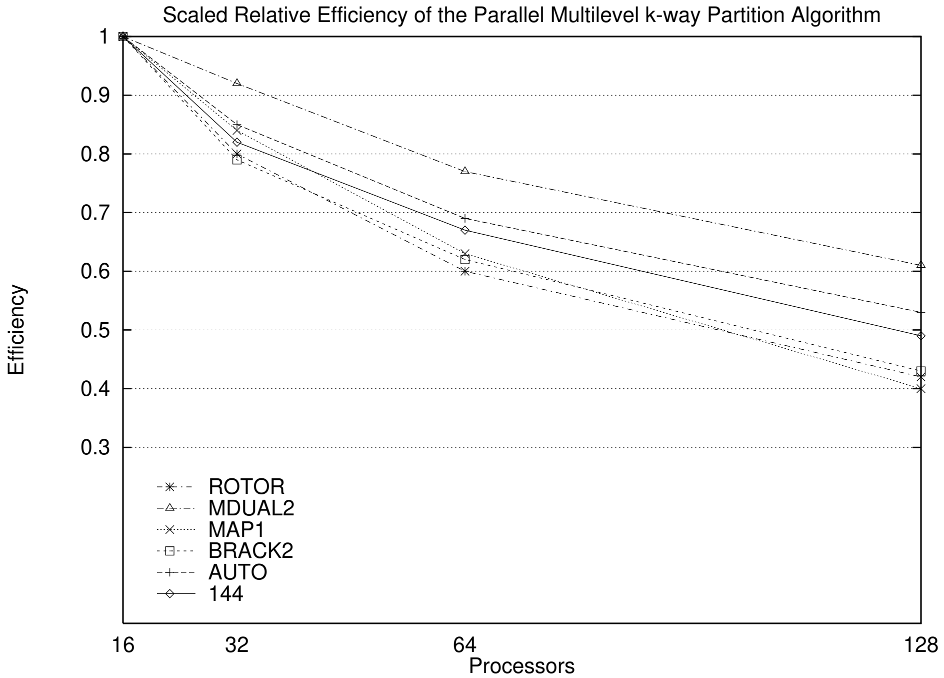


Figure 7: The scaled efficiencies achieved by the parallel algorithm for some graphs. The efficiencies are relative to the 16-processor runs, and the run times are scaled to reflect the increase in the amount of work performed as the number of partitions increases.

tices were assigned to each processor. We refer to this as the *as-is distribution*. This ordering is somewhat different than the random distribution that was assumed in the description and analysis of the parallel algorithm (Sections 4 and 5), and was chosen for its simplicity. To study the performance of our parallel algorithm under different initial graph distribution schemes we performed experiments using both random and pre-partitioned distributions. In both cases, a permutation was applied to the graph before distributing onto the processors. In the case of random distribution, this permutation was computed randomly, whereas in the case of the pre-partitioned distribution, this permutation was computed from a serial p-way partition of the graph.

	AUTO				**MDUAL2**			
	16PEs		128PEs		16PEs		128PEs	
Phase Name	Random	Pre-Partitioned	Random	Pre-Partitioned	Random	Pre-Partitioned	Random	Pre-Partitioned
Communication Setup	1.002	0.391	0.288	0.180	1.732	0.447	0.403	0.311
Graph Coloring	2.503	1.840	0.493	0.231	2.257	1.384	0.354	0.191
Computing Matching	1.265	0.726	0.362	0.129	1.772	0.812	0.386	0.139
Graph Contraction	2.122	1.192	0.429	0.163	2.692	1.296	0.438	0.181
Initial Partition	0.007	0.005	0.060	0.054	0.004	0.010	0.088	0.075
k-way Refinement	1.541	1.216	0.663	0.550	1.853	1.264	0.597	0.400
Total Runtime	8.430	5.370	2.295	1.310	10.310	5.213	2.266	1.297

Table 5: The amount of time (in seconds) required by the different phases of the parallel partitioning algorithm for different initial vertex distributions, on 16 and 128 processors.

Table 5 shows run time of these two different distribution schemes for two of the larger graphs in our experimental testbed. Comparing these run times with those shown in Table 4 we see that there is little difference between the random and the as-is distributions. The run time of the random distribution is only higher by less than 1%, which was expected since both distributions result in initial partitions that cut more than 90% of the edges. However, the run time is significantly reduced when the pre-partitioned distribution is used. For example, in the case of MDUAL2, on 16 processors, the run time of the pre-partitioned distribution is almost half of that achieved by either the random or the as-is distributions. This reduction in run time is due to the following two reasons: (a) reduced communication requirements, and (b) better cache utilization.

For the pre-partitioned graph distribution, the number of edges that get cut as a result of the initial distribution is significantly reduced to only 7.5% for AUTO, and 3.6% for MDUAL2. Consequently, the distributed graph has significantly fewer interface vertices. In each of the graph coloring, matching, contraction, and partition refinement algorithms, communication is a significant fraction of the overall run time. So reduction in the run time is due to reduced communication required for the pre-partitioned graph. The reduced communication requirements can be clearly seen in the amount of time required by the *communication setup* phase (especially for 16 processors), whose complexity highly depends on the number of interface vertices. Besides reducing communication overheads, the much better data locality that is produced by the pre-partitioned distribution, also significantly improves cache utilization. This is particularly important on a machine like the Cray T3D, since it has only a small amount of primary cache (8Kbytes) and no secondary cache. This improved cache reuse is the primary reason for the almost 50% improvement achieved by the coloring, matching, and contraction algorithms. The primary significance of the cache can also be seen when looking at the time required by the k-way refinement. In this case, the improvements are not as dramatic (somewhere between 27% and 40% on 16 processors). This is because, during k-way refinement only a few vertices get moved; hence, there is limited cache reuse.

7 Related Work

Raghavan [35] presents a parallel formulation of a nested dissection ordering algorithm that is based on multilevel graph partitioning. Raghavan's parallel algorithm uses one-dimensional partitioning of the graphs and construct successive coarser graphs by computing matchings between different pairs of processors at each coarsening level. Although this matching pairs vertices located on different processors, it does not make use of a global maximal matchings across all processors in each coarsening step. Another difference with our scheme is that this algorithm projects the bisection of the coarser graph directly onto the original graph without any refinement. This algorithm obtains speedup in the range of 25 to 40 on 128-processor CM5. Due to limited global matching and the absence of partition refinement, the orderings produced by this algorithm are worse than those produced by state-of-the-art multilevel ordering algorithms [21, 18, 10]. The speedup obtained by this algorithm are similar to those obtained by our algorithm, as the underlying communication in the coarsening phase is similar to that in the different phases of our algorithm.

Barnard [1] has developed a parallel formulation of multilevel spectral algorithm. This algorithm uses a one-dimensional mapping of the graph to the processors and uses a parallel formulation of Luby's [28] algorithm to compute a maximal independent set of vertices to construct the next level coarser graph. Note that the coarsening scheme used in Barnard's algorithm can not be used in any multilevel graph partitioning algorithm [4, 16, 21, 22, 35]. The reason is that the coarsened graph of the multilevel spectral algorithms does not have enough information to enforce balance constraints and partition quality. This coarsening scheme uses a maximal independent set of vertices (instead of a maximal independent set of edges used in multilevel graph partitioning algorithms); hence, it can use Luby's algorithm directly on the original graph. Table 6 shows the parallel performance of the of multilevel spectral bisection and our multilevel k-way partition for two of the problems in our experimental testbed[1]. From this table we see that our parallel algorithm produces partitions whose quality is significantly better than those produced by the parallel multilevel spectral bisection algorithm. In particular for MDUAL on 16 processors, our algorithm cuts 2.8 times fewer edges than the spectral algorithm. Furthermore, our algorithm is 25 to 30 times faster than the spectral algorithm, which is consistent with the serial computational requirements of the two algorithms. This difference in the edge-cuts of the two parallel formulations is similar to that for their serial counterparts [22]. Since both parallel formulations of multilevel spectral bisection and

[1]The parallel multilevel spectral bisection of Barnard was made available to us by Cray Research.

the multilevel k-way partitioning algorithms have similar communication overheads (*i.e.*, proportional to the number of interface vertices), their relative run time requirements do not change with the use of parallel computers, as both scale similarly.

	598a				MDUAL			
	16PEs		128PEs		16PEs		128PEs	
Parallel Algorithm	Edge-cut	Time	Edge-cut	Time	Edge-cut	Time	Edge-cut	Time
Recursive Spectral Bisection	37583	67.450	103928	22.478	37235	65.490	54692	18.971
Multilevel k-way Partitioning	31211	2.436	90724	0.805	13144	2.637	35457	0.795

Table 6: The performance of the parallel multilevel spectral bisection and our parallel k-way partitioning algorithms on 16- and 128-processor Cray T3D.

Diniz *et al.*, [6] present a parallel formulation of the inertial algorithm [31] for partitioning. This algorithm computes a k-way partition using inertial recursive bisection (which is naturally parallel), and then does pairwise partition refinement using the Kernighan-Lin heuristic as described in Section 4. Their experiments show that the quality of the partitions produced by the parallel inertial algorithm, are 10% to 30% worse compared to the serial implementation of the inertial algorithm that uses sequential KL refinement. This decrease in partition quality is due to fact that pairwise partition refinement is not as effective as the coloring-based global refinement scheme used by our algorithm.

Karypis and Kumar [23] present a parallel formulation of the serial multilevel recursive bisection algorithm [21] for graph partitioning and sparse matrix ordering. That algorithm uses a two-dimensional distribution of the graph to the processors and computes a local heavy-edge matching on the diagonal processors as discussed in Section 4. When the size of the matchings produced in successive coarsening levels becomes small, the graph is successively folded to smaller halves of the processor grid. This formulation minimizes communication overhead during the coarsening phase. This local matching produces sufficient coarsening as long as the average degree of the coarse graphs is sufficiently large (proportional to the square root of the number of processors). However, if the degree of the graphs is small (as it is the case for finite element meshes and their duals), then this local matching cannot sufficiently reduce the size of the graph before folding is required, resulting in poor speedup. Hence, this scheme is more appropriate for graphs with relatively high degree (over 30), especially if the target parallel computer has slow communication network (*e.g.*, a cluster of workstations). For a variety of problems, our parallel formulation of multilevel k-way partitioning presented in this paper is 3 to 4 times faster on a 128 processor Cray T3D, while the quality is better by about 5% to 10% compared with the formulation in [23]. This difference in run times for these parallel formulations is primarily because the algorithm in [23] does recursive bisection (requiring $\log p$ rounds of coarsening), and the algorithm in this paper does k-way partitioning (requiring only one coarsening phase). The edge-cuts produced by the parallel scheme in [23] are also somewhat worse because matching is limited to local vertices only.

8 Conclusion

In this paper we presented a scalable and highly parallel formulation of the multilevel k-way partitioning algorithm that is able to produce very good partitions of very large unstructured graphs in very small amount of time. The theoretical analysis presented in Section 5 shows that both the run time and scalability of our algorithm is within a factor of $O(\log p)$ from the theoretical lower bound for any parallel graph partitioning algorithm. In the context of repartitioning adaptively refined graphs, the run time of our parallel multilevel k-way partitioning algorithm can be further reduced. In this context, both the coloring and the matching phases can be modified to utilize much faster serial algorithms on the vertices that are internal to the domains assigned to each processor. By only requiring to perform distributed coloring and distributed matching for the interface nodes of the various domains, the overall run time of these phases can be reduced even further.

To our knowledge this is the first algorithm that provides a highly parallel and effective formulation of the k-way partitioning refinement algorithm. Even though our partition refinement is based on a relatively simple variant of the Kernighan-Lin type of algorithms, the concurrency that is exposed by using coloring can also be used to implement more sophisticated algorithms. For example, refinement algorithms that are able to climb out of local minima by performing some moves that do not decrease the edge-cut [16, 21] can be easily implemented using the techniques described in this paper.

The performance achieved by our algorithm allows for the development of efficient and scalable parallel formulations for many diverse problems that utilize and operate on unstructured graphs. The domain decomposition techniques used extensively in scientific computing [37] can be completely parallelized, removing the computational bottleneck created by serial domain decomposition prior to parallel computation. This also allows for the creation of highly parallel preconditioners for iterative methods based on domain decomposition as well as on incomplete factorizations [24]. Furthermore, adaptive finite element methods can now be effectively parallelized, since the mesh can be repartitioned on the fly very fast.

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] A. George. Nested dissection of a regular finite-element mesh. *SIAM Journal on Numerical Ananlysis*, 10:345–363, 1973.

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

[10] Anshul Gupta. Fast and effective algorithms for graph partitioning and sparse matrix reordering. Technical Report RC 20496 (90799), IBM T. J. Watson Research Center, Yorktown Heights, NY, July 10, 1996.

[11] 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.

[12] 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*.

[13] 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.

[14] 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.

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

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

[17] Bruce Hendrickson and Robert Leland. The chaco user's guide, version 2.0. Technical Report SAND94-2692, Sandia National Laboratories, 1994.

[18] Bruce Hendrickson and Edward Rothberg. Improving the runtime and quality of nested dissection ordering. Technical Report CS-96-000, Sandia National Laboratories, 1996.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] George Karypis. *Graph Partitioning and Its Applications to Scientific Computing*. PhD thesis, University of Minnesota, Minneapolis, MN, 1996.

[25] 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.

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

[27] 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.

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

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

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

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

[36] 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.

[37] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. PWS Publishing, Boston, MA, 1996.

[38] Ravi V. Shankar and Sanjay Ranka. Random data accesses on coarse-grained parallel machine. *Journal of Parallel and Distributed Computing*, 1995. To appear.