Appears in the Journal of Parallel and Distributed Computing
A short version of this paper appears in International Parallel Processing Symposium 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

A Parallel Algorithm for Multilevel Graph Partitioning and Sparse Matrix Ordering *

George Karypis and Vipin Kumar

University of Minnesota, Department of Computer Science/ Army HPC Research Center
Minneapolis, MN 55455, Technical Report: 95-036

{karypis, kumar}@cs.umn.edu

Last updated on October 9, 1997 at 5:08pm

Abstract

In this paper we present a parallel formulation of the multilevel graph partitioning and sparse matrix ordering algorithm. A key feature of our parallel formulation (that distinguishes it from other proposed parallel formulations of multilevel algorithms is that it partitions the vertices of the graph into $\sqrt{p}$ parts while distributing the overall adjacency matrix of the graph among all p processors. This mapping results in substantially smaller communication than one-dimensional distribution for graphs with relatively high degree, especially if the graph is randomly distributed among the processors. We also present a parallel algorithm for computing a minimal cover of a bipartite graph which is a key operation for obtaining a small vertex separator that is useful for computing the fill reducing ordering of sparse matrices. Our parallel algorithm achieves a speedup of up to 56 on 128 processors for moderate size problems, further reducing the already moderate serial run time of multilevel schemes. Furthermore, the quality of the produced partitions and orderings are comparable to those produced by the serial multilevel algorithm that has been shown to outperform both spectral partitioning and multiple minimum degree.

*This work was supported by NSF CCR-9423082, by the Army Research Office contract DA/DAAH04-95-1-0538, by the IBM Partnership Award, 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**

Keywords: Parallel Graph Partitioning, Multilevel Partitioning Methods, Fill Reducing Ordering, Numerical Linear Algebra.

1 Introduction

Graph partitioning is an important problem that has extensive applications in many areas, including scientific computing, VLSI design, task scheduling, geographical information systems, and operations research. The problem is to partition the vertices of a graph to p roughly equal parts, such that the number of edges connecting vertices in different parts 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. Partitioning the graph that corresponds to matrix A, is used to significantly reduce the amount of communication [19]. If parallel direct methods are used to solve a sparse system of equations, then a graph partitioning algorithm can be used to compute a fill reducing ordering that lead to high degree of concurrency in the factorization phase [19, 8]. The multiple minimum degree ordering used almost exclusively in serial direct methods is not suitable for parallel direct methods, as it provides limited concurrency in the parallel factorization phase.

The graph partitioning problem is NP-complete. However, many algorithms have been developed that find a reasonably good partition. Recently, a new class of multilevel graph partitioning techniques was introduced by Bui & Jones [4] and Hendrickson & Leland [12], and further studied by Karypis & Kumar [16, 15, 13]. These multilevel schemes provide excellent graph partitionings and have moderate computational complexity. Even though these multilevel algorithms are quite fast compared with spectral methods, parallel formulations of multilevel partitioning algorithms are needed for the following 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. A parallel graph partitioning algorithm can take advantage of the significantly higher amount of memory available in parallel computers. Furthermore, with recent development of highly parallel formulations of sparse Cholesky factorization algorithms [9, 17, 25], numeric factorization on parallel computers can take much less time than the step for computing a fill-reducing ordering on a serial computer. For example, on a 1024-processor Cray T3D, some matrices can be factored in a few seconds using our parallel sparse Cholesky factorization algorithm [17], but serial graph partitioning (required for ordering) takes several minutes for these problems.

In this paper we present a parallel formulation of the multilevel graph partitioning and sparse matrix ordering algorithm. A key feature of our parallel formulation (that distinguishes it from other proposed parallel formulations of multilevel algorithms [2, 1, 24, 14]) is that it partitions the vertices of the graph into $\sqrt{p}$ parts while distributing the overall adjacency matrix of the graph among all p processors. This mapping results in substantially smaller communication than one-dimensional distribution for graphs with relatively high degree, especially if the graph is randomly distributed among the processors. We also present a parallel algorithm for computing a minimal cover of a bipartite graph which is a key operation for obtaining a small vertex separator that is useful for computing the fill reducing ordering of sparse matrices. Our parallel algorithm achieves a speedup of up to 56 on 128 processors for moderate size problems, further reducing the already moderate serial run time of multilevel schemes. Furthermore, the quality of the produced partitions and orderings are comparable to those produced by the serial multilevel algorithm that has been shown to outperform both spectral partitioning and multiple minimum degree [16]. The parallel formulation in this paper is described in the context of the serial multilevel graph partitioning algorithm presented in [16]. However, nearly all of the discussion in this paper is applicable to other multilevel graph partitioning algorithms [4, 12, 7, 22].

The rest of the paper is organized as follows. Section 2 surveys the different types of graph partitioning algorithms that are widely used today. Section 2 briefly describes the serial multilevel algorithm that forms the basis for the parallel algorithm described in Sections 3 and 4 for graph partitioning and sparse matrix ordering respectively. Section 5 analyzes the complexity and scalability of the parallel algorithm. Section 6 presents the experimental evaluation of the parallel multilevel graph partitioning and sparse matrix ordering algorithm. Section 7 provides some concluding remarks.

2 The Graph Partitioning Problem and Multilevel Graph Partitioning

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

The p-way partitioning problem is most frequently solved by recursive bisection. That is, we first obtain a 2-way partition of V, and then we further subdivide each part using 2-way partitions. After $\log p$ phases, graph G is partitioned into p parts. Thus, the problem of performing a p-way partition is reduced to that of performing a sequence of 2-way partitions or bisections. Even though this scheme does not necessarily lead to optimal partition [27, 15], it is used extensively due to its simplicity [8, 10].

The basic structure of the multilevel bisection algorithm is very simple. The graph $G = (V, E)$ is first coarsened down to a few thousand vertices (coarsening phase), a bisection of this much smaller graph is computed (initial partitioning phase), and then this partition is projected back towards the original graph (uncoarsening phase), by periodically refining the partition [4, 12, 16]. Since the finer graph has more degrees of freedom, such refinements improve the quality of the partitions. This process, is graphically illustrated in Figure 1.

During the coarsening phase, a sequence of smaller graphs $G_l = (V_l, E_l)$, is constructed from the original graph $G_0 = (V_0, E_0)$ such that $|V_l| > |V_{l+1}|$. Graph G_{l+1} is constructed from G_l by finding a maximal matching $M_l \subseteq E_l$ of G_l and collapsing together the vertices that are incident on each edge of the matching. Maximal matchings can be computed in different ways [4, 12, 16, 15]. The method used to compute the matching greatly affects both the quality of the partitioning, and the time required during the uncoarsening phase. One simple scheme for computing a matching is the ***random matching*** (RM) scheme [4, 12]. In this scheme vertices are visited in random order, and for each unmatched vertex we randomly match it with one of its unmatched neighbors. An alternative matching scheme that we have found to be quite effective is called ***heavy-edge matching*** (HEM) [16, 13]. The HEM matching scheme computes a matching M_l, such that the weight of the edges in M_l is high. The heavy-edge matching is computed using a randomized algorithm as follows. The vertices are again visited in 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 graph. The coarsening phase ends when the coarsest graph G_m has a small number of vertices.

During the initial partitioning phase a bisection of the coarsest graph is computed. Since the size of the coarser graph G_k is small (often $|V_k|$ is less than 100 vertices), this step takes relatively small amount of time.

During the uncoarsening phase, the partition 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 $u \in V_{l+1}$ contains a distinct subset U of vertices of V_l, the projection of the partition from G_{l+1} to G_l is constructed by simply assigning the vertices in U to the same part in G_l to the same part that vertex u belongs in G_{l+1}. After projecting a partition, a partitioning refinement algorithm is used. The basic purpose of a partitioning refinement algorithm is to select vertices such that when moved from one

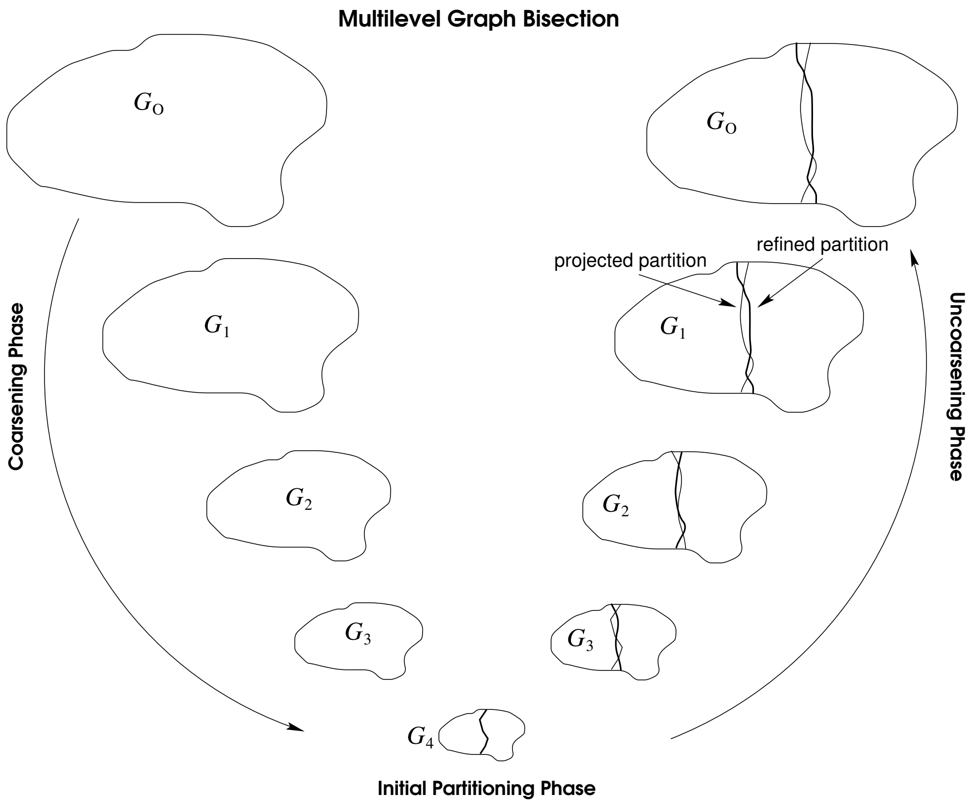


Figure 1: The various phases of the multilevel graph bisection. During the coarsening phase, the size of the graph is successively decreased; during the initial partitioning phase, a bisection of the smaller graph is computed; and during the uncoarsening phase, the bisection is successively refined as it is projected to the larger graphs. During the uncoarsening phase the light lines indicate projected partitions, and dark lines indicate partitions that were produced after refinement.

partition to another the resulting partitioning has smaller edge-cut and remains balanced (i.e., each part has the same weight). A class of local refinement algorithms that tend to produce very good results are those that are based on the Kernighan-Lin (KL) partitioning algorithm [18] and their variants (FM) [6, 12, 16].

3 Parallel Multilevel Graph Partitioning Algorithm

There are two types of parallelism that can be exploited in the p-way graph partitioning algorithm based on the multilevel bisection described in Section 2. The first type of parallelism is due to the recursive nature of the algorithm. Initially a single processor finds a bisection of the original graph. Then, two processors find bisections of the two subgraphs just created and so on. However, this scheme by itself can use only up to $\log p$ processors, and reduces the overall run time of the algorithm only by a factor of $O(\log p)$. We will refer to this type of parallelism as the parallelism associated with the ***recursive step***.

The second type of parallelism that can be exploited is during the ***bisection step***. In this case, instead of performing the bisection of the graph on a single processor, we perform it in parallel. We will refer to this type of parallelism as the parallelism associated with the bisection step. Note that if the bisection step is parallelized, then the speedup obtained by the parallel graph partitioning algorithm can be significantly higher than $O(\log p)$.

The parallel graph partitioning algorithm we describe in this section exploits both of these types of parallelism. Initially all the processors cooperate to bisect the original graph G, into G^0 and G^1. Then, half of the processors bisect G^0, while the other half of the processors bisect G^1. This step creates four subgraphs G^{00}, G^{01}, G^{10}, and G^{11}. Then each quarter of the processors bisect one of these subgraphs and so on. After $\log p$ steps, the graph G has been partitioned into p parts.

In the next three sections we describe how we have parallelized the three phases of the multilevel bisection algorithm.

3.1 Coarsening Phase

As described in Section 2, during the coarsening phase, a sequence of coarser graphs is constructed. A coarser graph $G_{l+1} = (V_{l+1}, E_{l+1})$ is constructed from the finer graph $G_l = (V_l, E_l)$ by finding a maximal matching M_l and contracting the vertices and edges of G_l to form G_{l+1}. This is the most time consuming phase of the three phases; hence, it needs be parallelized effectively. Furthermore, the amount of communication required during the contraction of G_l to form G_{l+1} depends on how the matching is computed.

The randomized algorithms described in Section 2 for computing a maximal matching on a serial computer are simple and efficient. However, computing a maximal matching in parallel is hard, particularly on a distributed memory parallel computer. A direct parallelization of the serial randomized algorithms or algorithms based on depth first graph traversals require significant amount of communication. For instance, consider the following parallel implementation of the randomized algorithms. Each processor contains a (random) subset of the graph. For each local vertex v, processors select an edge (v, u) to be in the matching. Now, the decision of whether or not an edge (v, u) can be included in the matching may result in communication between the processors that locally store v and u, to determine if vertex u has been matched or not. In addition to that, care must be taken to avoid race conditions, since vertex u may be checked due to another edge (w, u), and only one of the (v, u) and (w, u) edges must be included in the matching. Similar problems arise when trying to parallelize algorithms based on depth-first traversal of the graph. Another possibility is to adapt some of the algorithms that have been developed for the PRAM model. In particular the algorithm of Luby [21] for computing the maximal independent set can be used to find a matching. However, parallel formulations of this type of algorithms also have high communication overhead because each processor p_i needs to communicate with all other processors that contain neighbors of the nodes local at p_i. Furthermore, having computed M_l using any one of the above algorithms, the construction of the next level coarser graph, G_{l+1} requires significant amount of communication. This is because each edge of M_l may connect vertices whose adjacent lists are stored on different processors, and during the contraction at least one of these adjacency lists needs to be moved from one processor to another. Communication overhead in any of the above algorithms can become small if the graph is initially partitioned among processors in such a way so that the number of edges going across processor boundaries are small. But this requires solving the p-way graph partitioning problem that we are trying to solve using these algorithms.

Another way of computing a maximal matching is to divide the n vertices among p processors and then compute matchings between the vertices locally assigned within each processor. The advantages of this approach is that no communication is required to compute the matching, and since each pair of vertices that gets matched belongs to the same processor, no communication is required to move adjacency lists between processors. However, this approach causes problems because each processor has very few nodes to match from. Also, even though there is no need to exchange adjacency lists among processors, each processor needs to know matching information about all the vertices that its local vertices are connected to in order to properly form the contracted graph. As a result significant amount of communication is required. In fact this computation is very similar in nature to the multiplication of a randomly sparse matrix (corresponding to the graph) with a vector (corresponding to the matching vector).

In our parallel coarsening algorithm, we retain the advantages of the local matching scheme, but minimize its drawbacks by computing the matchings between groups of $n/\sqrt{p}$ vertices. This increases the size of the computed matchings, and also reduces the communication overhead for constructing the coarse graph. Specifically, our parallel coarsening algorithm treats the p processors as a two-dimensional array of $\sqrt{p} \times \sqrt{p}$ processors (assume that $p = 2^{2r}$). The vertices of the graph $G_0 = (V_0, E_0)$ are distributed among this processor grid using a cyclic mapping [19]. The vertices V_0 are partitioned into $\sqrt{p}$ subsets, $V_0^0, V_0^1, \ldots, V_0^{\sqrt{p}-1}$. Processor $P_{i,j}$ stores the edges of E_0 between the subsets of vertices V_0^i and V_0^j. Having distributed the data in this fashion, the algorithm then proceeds to find a matching. This matching is computed by the processors along the diagonal of the processor-grid. In particular, each processor $P_{i,i}$ finds a heavy-edge matching M_0^i using the set of edges it stores locally. The union of these $\sqrt{p}$ matchings is taken as the overall matching M_0. Since the vertices are split into $\sqrt{p}$ parts, this scheme finds larger matchings than the one that partitions vertices into p parts.

In order for the next level coarser graph G_1 to be created, processor $P_{i,j}$ needs to know the parts of the matching that were found by processors $P_{i,i}$ and $P_{j,j}$ (*i.e.*, M_0^i and M_0^j, respectively). Once it has this information, then it can proceed to create the edges of G_1 that are to be stored locally without any further communication. The appropriate parts of the matching can be made available from the diagonal processors to the other processors that require them by two single node broadcast operations [19]—one along the row and one along the columns of the processor grid. These steps are illustrated in Figure 2. At this point, the next level coarser graph $G_1 = (V_1, E_1)$ has been created such that the vertices V_1 are again partitioned in $\sqrt{p}$ subsets, and processor $P_{i,j}$ stores the edges of E_1 between subsets of vertices V_1^i and V_1^j. The next level coarser graphs are created in a similar fashion.

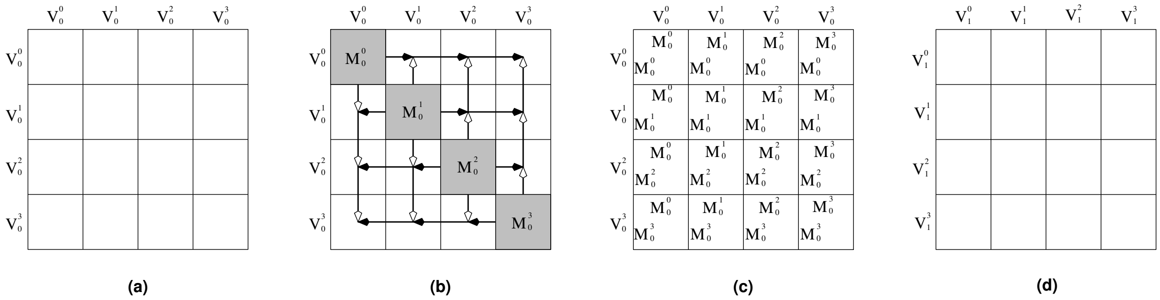


Figure 2: The various phases of a coarsening level. (a) The distribution of the vertices of the graph. (b) Diagonal processors compute the matchings and broadcast them along the rows and the columns. (c) Each processor locally computes the next level coarser graph assigned to it. (d) The distribution of the vertices for the next coarsening level.

The coarsening algorithm continues until the number of vertices between successive coarser graphs does not substantially decrease. Assume that this happens after k coarsening levels. At this point, graph $G_k = (V_k, E_k)$ is folded into the lower quadrant of the processor subgrid as shown in Figure 3. The coarsening algorithm then continues by creating coarser graphs. Since the subgraph of the diagonal processors of this smaller processor grid contains more vertices and edges, larger matchings can be found and thus the size of the graph is reduced further. This process of coarsening followed by folding continues until the entire coarse graph has been folded down to a single processor, at which point the sequential coarsening algorithm is employed to coarsen the graph.

Since, between successive coarsening levels, the size of the graph decreases, the coarsening scheme just described utilizes more processors during the coarsening levels in which the graphs are large and fewer processors for the smaller graphs. As our analysis in Section 5 shows, decreasing the size of the processor grid does not affect the overall performance of the algorithm as long as the graph size shrinks by a large enough factor between successive graph foldings.

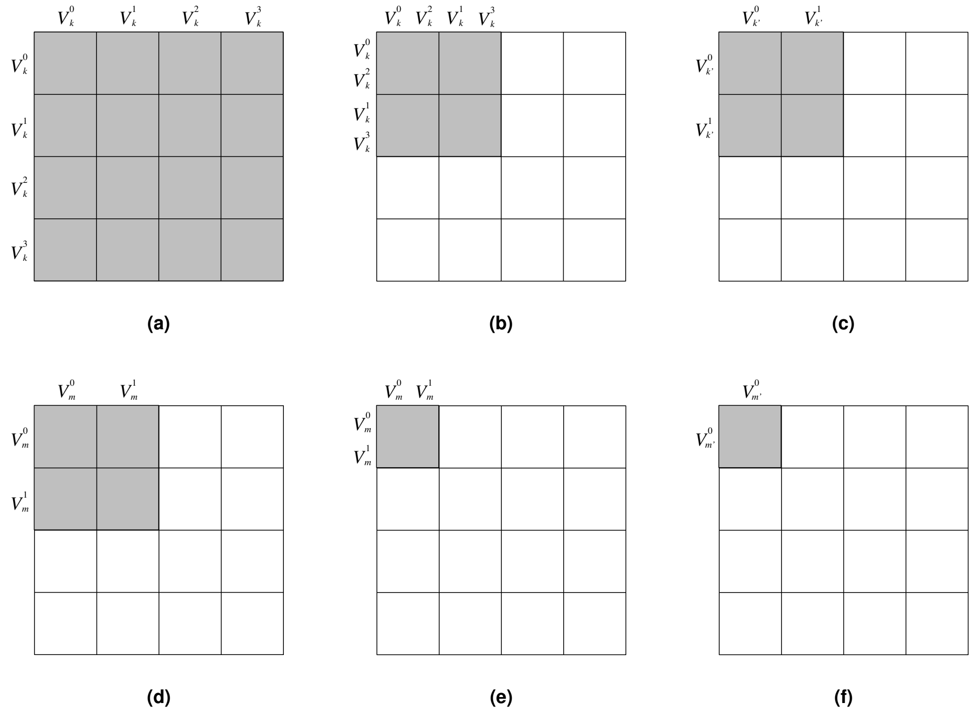


Figure 3: The process of folding the graph. (a) The distribution of the vertices of the graph prior to folding. (b, c) Processors send their graphs to the processors at the lower quadrant. The same processes is repeated after $m - k$ coarsening levels, in which case the graph is folded to a single processor (d, e, f).

3.2 Initial Partitioning Phase

At the end of the coarsening phase, the coarsest graph resides on a single processor. We use the GGGP algorithm described in [16] to partition the coarsest graph. We perform a small number of GGGP runs starting from different random vertices and the one with the smaller edge-cut is selected as the partition. Instead of having a single processor performing these different runs, the coarsest graph can be replicated to all (or a subset of) processors, and each of these processors can perform its own GGGP partition. We did not implement it, since the run time of the initial partition phase is only a very small fraction of the run time of the overall algorithm.

3.3 Uncoarsening Phase

During the uncoarsening phase, the partition of the coarsest graph G_m is projected back to the original graph by going through the intermediate graphs $G_{m-1}, G_{m-2}, \cdots, G_1$. After each step of projection, the resulting partition is further refined by using vertex swap heuristics that decrease the edge-cut as described in Section 2. Further, recall that during the coarsening phase, the graphs are successively folded to smaller processor grids just before certain coarsening levels. This process is reversed during the parallel uncoarsening phase for the corresponding uncoarsening levels; *i.e.*, the partition (besides being projected to the next level finer graph) is unfolded to larger processor grids. The step of projection and unfolding to larger processor grids are parallelized in a way similar to their counterparts in the coarsening phase. Here we describe our parallel implementation of the refinement step.

For refining the coarser graphs that reside on a single processor, we use the boundary Kernighan-Lin refinement algorithm (BKLR) described in [16]. However, the BKLR algorithm is sequential in nature and it cannot be used in its current form to efficiently refine a partition when the graph is distributed among a grid of processors. In this case we use a different algorithm that tries to approximate the BKLR algorithm but is more amenable to parallel computations. The key idea behind our parallel refinement algorithm is to select a group of vertices to swap from one part to the other instead of selecting a single vertex. Refinement schemes that use similar ideas are described in [26, 5];. However, our algorithm differs in two important ways from the other schemes: (i) it uses a different method for selecting vertices; (ii) it uses a two-dimensional partition to minimize communication.

Consider a $\sqrt{p} \times \sqrt{p}$ processor grid on which graph $G = (V, E)$ is distributed. Furthermore each processor $P_{i,j}$ computes the gain in the edge-cut obtained from moving vertex $v \in V^j$, to the other part by considering only the part of G (*i.e.*, vertices and edges of G) stored locally at $P_{i,j}$. This locally computed gain is called lg_v. The gain g_v, of moving vertex v is computed by a sum-reduction of the lg_v, along the columns of the processor grid. Let the processors along the diagonal of the grid store the g_v values for the subset of V assigned locally.

The parallel refinement algorithm consists of a number of steps. During each step, at each diagonal processor a group of vertices is selected from one of the two parts and is moved to the other part. The group of vertices selected by each diagonal processor corresponds to the vertices that have positive g_v values (*i.e.*, lead to a decrease in the edge-cut). Each diagonal processor $P_{i,i}$ then broadcasts the group of vertices U_i it selected along the rows and the columns of the processor grid. Now, each processor $P_{i,j}$ knows the group of vertices U_i and U_j from V^i and V^j respectively that have been moved to the other part and updates the lg_v values of the vertices in U_j and of the vertices that are adjacent to vertices in U_i. The updated gain values g_v are computed by a reduction along the columns of the modified lg_v values. This process continues by alternating the part from where vertices are moved, until either no further improvement in the overall edge-cut can be made, or a maximum number of iterations has been reached. In our experiments, the maximum number of iterations was set to six. Balance between partitions is maintained by (a) always starting the sequence of vertex swaps from the heavier part of the partition, and (b) by employing an explicit balancing iteration at the end of each refinement phase if there is more than 2% load imbalance between the parts of the partition.

Our parallel refinement algorithm has a number of interesting properties that positively affect its performance and its ability to refine the partition. First, the task of selecting the group of vertices to be moved from one part to the other is distributed among the diagonal processors instead of being done serially. Secondly, the task of updating the internal and external degrees of the affected vertices is distributed among all the p processors. Furthermore, we restrict the moves in each step to be unidirectional (*i.e.*, they go only from one partition to other) instead of being bidirectional (*i.e.*, allow both types of moves in each phase). This guarantees that each vertex in the group of vertices $U = \bigcup_i U_i$ being moved reduces the edge-cut. In particular, let $g_U = \sum_{v \in U} g_v$, be the sum of the gains of the vertices in U. Then the reduction in the edge-cut obtained by moving the vertices of U to the other part is at least g_U. To see that, consider a vertex $v \in U$ that has a positive gain (*i.e.* $g_v > 0$); the gain will decrease if and only if some of the adjacent vertices of v that belong to the other part move. However, since in each phase we do not allow vertices from the other part to move, the gain of moving v is at least g_v irrespective of whatever other vertices on the same side as v have been moved. It follows that the gain achieved by moving the vertices of U can be higher than g_U.

In the serial implementation of BKLR, it is possible to make vertex moves that initially lead to worse partition, but eventually (when more vertices are moved) better partition is obtained. Thus, the serial implementation has the ability to climb out of local minima. However, the parallel refinement algorithm lacks this capability, as it never moves vertices if they increase the edge-cut. Also, the parallel refinement algorithm, is not as precise as the serial algorithm as it swaps groups of vertices rather than one vertex at a time. However, our experimental results show that it produces results that are not much worse than those obtained by the serial algorithm. The reason is that the graph coarsening process provides enough global view and the refinement phase only needs to provide minor local improvements.

4 Parallel Multilevel Sparse Matrix Ordering Algorithm

The parallel multilevel graph bisection algorithm can be used to generate a fill reducing ordering using nested dissection. To obtain a fill reducing ordering we need an algorithm that constructs a vertex-separator from the bisection produced by the parallel graph bisection algorithm.

Let A and B be the sets of vertices along the boundary of the bisection, each belonging to one of the two different parts. A boundary induced separator can be easily constructed by simply choosing the smaller of A and B. However, a different separator can be constructed using a minimum cover algorithm for bipartite graphs [23] that contains subsets of vertices from both A and B. In many cases, this new separator S may have 20% to 40% fewer vertices than either A or B. Since, the size of the vertex separator directly affects the fill and thus, the time required to factor the matrix, small separators are extremely important.

The worst-case complexity of the minimum cover algorithm is $O(|A \cup B|^2)$ [23]. However, $A \cup B$ can be fairly large ($O(|V|^{2/3})$) for three-dimensional finite element graphs; hence, this step needs to be performed in parallel. The minimum cover algorithm is based on bipartite matching which uses depth-first traversal of the bipartite graph, making it hard to obtain an efficient parallel formulation. For this reason our parallel formulation implements a relaxed version of the minimum cover algorithm.

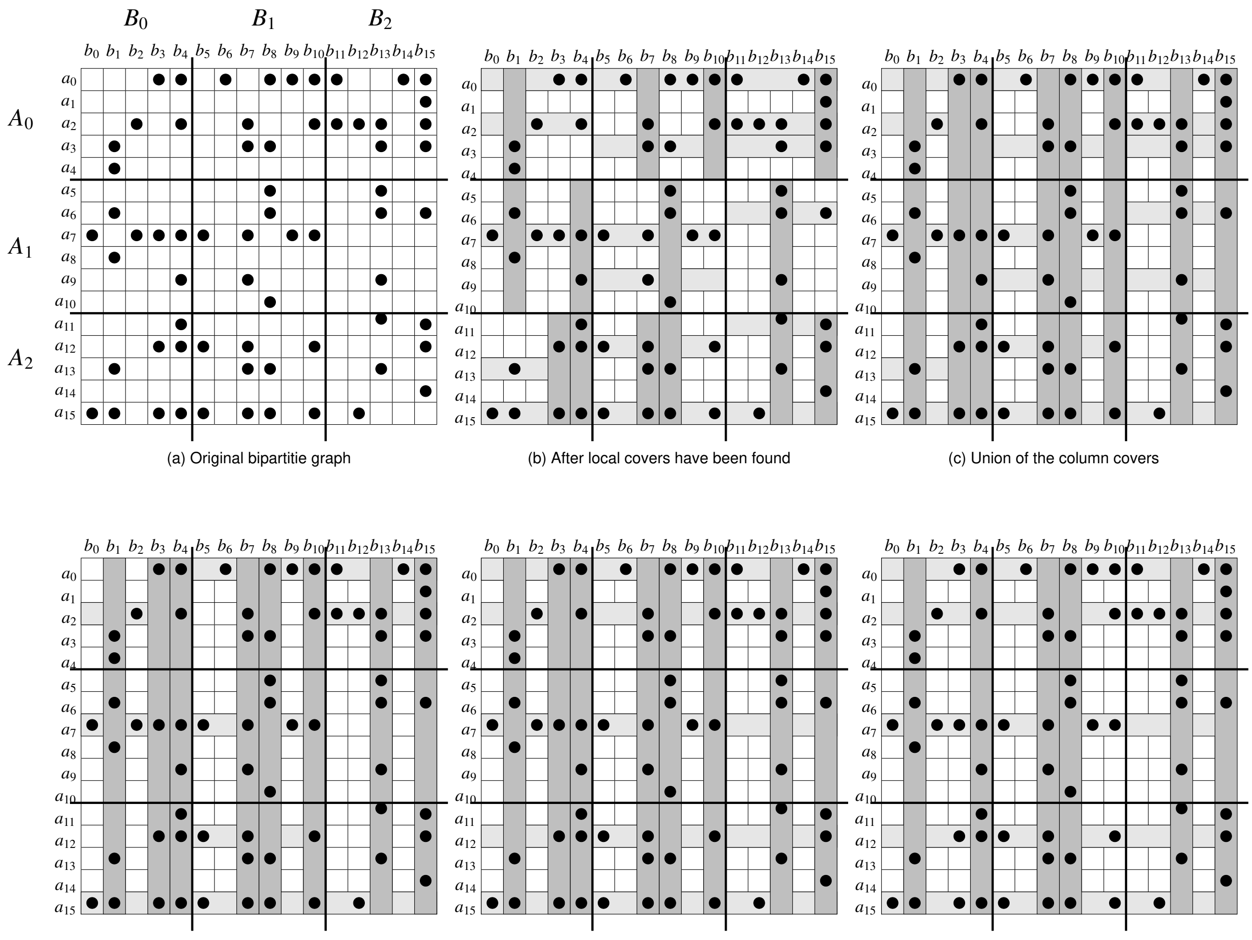
Parallel Vertex Cover Algorithm Recall from Section 3.1 that the vertices of a graph $G = (V, E)$ are distributed on a two-dimensional processor grid so that each processor $P_{i,j}$ contains the edges between vertices V^i and V^j. Let $A^i = A \cap V^i$ and $B^j = B \cap V^j$ that is, A^i and B^j are the subsets of boundary vertices stored locally at processor $P_{i,j}$. Figure 4(a) illustrates this notation by an example. In this figure we represent the connections between sets A and B by a matrix whose rows correspond to the vertices in set A and the columns correspond to the vertices in set B. This matrix is distributed among the processor grid, which in the example is of size 3×3. Using this representation of the bipartite graph, a vertex cover of A and B corresponds to a selection of rows and columns that includes all of the non-zeros in the matrix.

Each processor $P_{i,j}$ finds locally the minimum cover of edges between A^i and B^j. Let $A_c^{i,j} \subseteq A^i$ and $B_c^{i,j} \subseteq B^j$ such that $A_c^{i,j} \cup B_c^{i,j}$ is this minimum cover. Figure 4(b) shows the minimum covers computed by each processor. For example, the minimum cover computed by processor $P_{0,0}$ contains the vertices $\{a_0, a_1, b_1\}$, that is $A_c^{0,0} = \{a_0, a_1\}$ and $B_c^{0,0} = \{b_1\}$. Note that the union of $A_c^{i,j}$ and $B_c^{i,j}$ over all the processors is a cover for all the edges between A and B. This cover can be potentially smaller than either A or B. However, this cover is necessarily minimal, as it may contain vertices from A (or B) such that the edges covered by them are also covered by other vertices in the cover. This can happen because the minimum cover for the edge-set at each processor $P_{i,j}$ is computed independently of the other processors. For example, the union of the local covers computed by all the processors in Figure 4(b) is

$$\{a_0, a_2, a_3, a_6, a_7, a_9, a_{11}, a_{12}, a_{13}, a_{15}, b_1, b_3, b_4, b_7, b_8, b_{10}, b_{14}, b_{15}\}$$

which is not minimal, since for example we can remove a_3 and still have a vertex cover. The size of this cover can be potentially reduced as follows.

Let $B_c^j = \cup_i B_c^{i,j}$. That is B_c^j is the union of $B_c^{i,j}$ along the columns of the processor grid. This union is computed via a reduction along the columns and is broadcasted to all the processors along the corresponding grid column. For the set of edges at each processor $P_{i,j}$, the set $A_c^{i,j} \cup B_c^j$ is also a cover, which may not be minimum since $B_c^j \supseteq B_c^{i,j}$. Now each processor removes vertices from $A_c^{i,j}$ such that every edge covered by these vertices is also covered by some vertex in B_c^j. More precisely, a vertex $v \in A_c^{i,j}$ is removed if for all edges (v, u) at processor $P_{i,j}$, $u \in B_c^j$. Figure 4(c)



(a) Original bipartitie graph

(b) After local covers have been found

(c) Union of the column covers

(d) After locally removing some of the row cover

(e) Union of the row covers

(f) After removing column covers

Figure 4: An example of the computations performed while computing a minimal vertex cover in parallel.

shows vertex covers at each processor after the union of $B_c^{i,j}$ has been computed along the columns. Figure 4(d) shows the covers after some vertices of set A have been removed. For example, processor $P_{0,0}$ can remove vertex a_0 from the cover of the locally stored edges, because it now has vertices b_3 and b_4 in the vertex cover.

Let $A_{c'}^{i,j}$ be the reduced version of $A_c^{i,j}$. Note that the union of $A_{c'}^{i,j}$ and $B_c^{i,j}$ over all the processors is also a cover of all the edges between A and B. This cover is potentially smaller than the cover obtained as the union of $A_c^{i,j}$ and $B_c^{i,j}$. However, we can still may able to further reduce the size of this cover as follows. Let $A_{c'}^{i} = \cup_j A_{c'}^{i,j}$. That is $A_{c'}^{i}$ is the union of $A_{c'}^{i,j}$ along the rows of the processor grid. See Figure 4(e) for an illustration. This union is computed via a reduction along the rows and is broadcasted to all the processors along the corresponding rows. For each processor $P_{i,j}$ the set $A_{c'}^{i} \cup B_c^{j}$ is also a cover. However, since $A_{c'}^{i} \supseteq A_{c'}^{i,j}$, some of the vertices of B_c^{j} may not be needed in the cover that keeps $A_{c'}^{i}$ fixed, and thus they can be removed. The removal of the vertices in B_c^{j} is shown in Figures 4(e) and 4(f), respectively. Note that vertices b_3 and b_{10} can now be removed from the vertex cover.

Let $B_{c'}^{i,j}$ be the reduced version of B_c^{j}. Let $B_{c'}^{j}$ be the union of the reduced $B_{c'}^{i,j}$'s along the columns. The separator S is then constructed as

$$S = \left(\bigcup_{i=0}^{\sqrt{p}-1} A_{c'}^{i} \right) \bigcup \left(\bigcup_{j=0}^{\sqrt{p}-1} B_{c'}^{j} \right).$$

Note that $A_{c'}^{i,j}$ and $B_{c'}^{i,j}$ cannot be further reduced, even if the entire S is made available at each processor. The reason is that at this time, each $b_i \in B_{c'}^{i,j}$ covers at least one edge that is not covered by any $a_i \in A_{c'}^{i,j}$, and each $a_i \in A_{c'}^{i,j}$ covers at least one edge that is not covered by $b_i \in B_{c'}^{i,j}$. Hence, the set S is at a local minimum.

In Figure 5 we plotted the reduction in the size of the top level vertex separator obtained by using our parallel minimum cover algorithm over the boundary induced vertex separator for 16, 32, and 64 processors for some matrices. For most matrices, the approximate minimum cover algorithm reduces the size of the vertex separator by at least 10%, and for some other it decreases by as much as 25%. Furthermore, our experiments (not reported here) show that their sizes are usually close to those obtained using the minimum cover algorithm running on a serial computer.

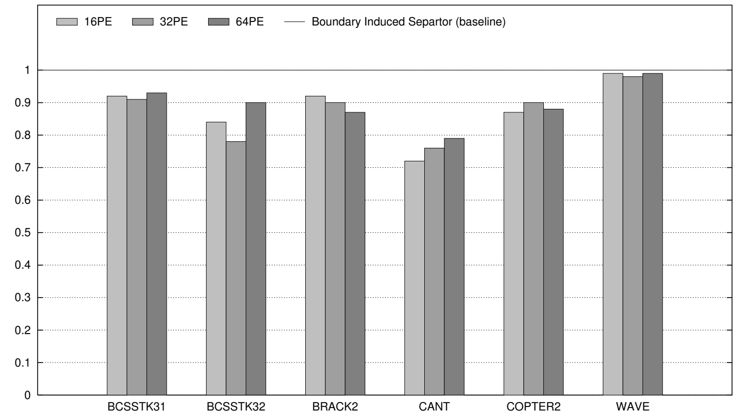


Figure 5: The size of the approximate minimum cover vertex separator relative to the boundary induced vertex separator.

5 Performance and Scalability Analysis

A complete analysis of our parallel multilevel algorithm has to account for the communication overhead in each coarsening step and the idling overhead that results from folding the graph onto smaller processor grids. The analysis presented in this section is for hypercube-connected parallel computers, but it is applicable to a much broader class of architectures for which the bisection bandwidth is $O(p)$ (*e.g.*, fat trees, crossbar, and multistage networks).

Consider a hypercube-connected parallel computer with $p = 2^{2r}$ processors arranged as a $\sqrt{p} \times \sqrt{p}$ grid. Let $G_0 = (V_0, E_0)$ be the graph that is partitioned into p parts, and let $n = |V_0|$ and $m = |E_0|$. During the first coarsening level, the diagonal processors determine the matching, they broadcast it to the processors along the rows and columns of the processor grid, and all the processors construct the local part of the next level coarser graph. The time required to find the matching, and to create the next level coarser graph is of the order of the number of edges stored in each processor *i.e.*, $O(m/p)$. Each diagonal processor finds a matching of the $O(n/\sqrt{p})$ vertices it stores locally, and broadcasts it along the rows and columns of the processor grid. Since the size of these vectors are much larger than $\sqrt{p}$, this broadcast can be performed in time linear to the message size, by performing a one-to-all personalized broadcast followed by an all-to-all broadcast (Problem 3.24 in [19]). Thus, the time required by the broadcast is $O(n/\sqrt{p})$.

If we assume (see the discussion in Section 6.3) that in each successive coarsening level the number of vertices decreases by a factor greater than one, and that the size of the graphs between successive foldings decreases by a factor greater than four, then the amount of time required to compute a bisection is dominated by the time required to create the first level coarse graph. Thus, the time required to compute a bisection of graph G_0 is:

$$T^{bissection} = O\left(\frac{m}{p}\right) + O\left(\frac{n}{\sqrt{p}}\right). \qquad (1)$$

After finding a bisection, the graph is split and the task of finding a bisection for each of these subgraphs is assigned to a different half of the processors. The amount of communication required during this graph splitting is proportional to the number of edges stored in each processor; thus, this time is $O(m/p)$, which is of the same order as the communication time required during the bisection step. This processes of bisection and graph splitting continues for a total of $\log p$ times. At this time a subgraph is stored locally on a single processor and the p-way partition of the graph has been found. The time required to compute the bisection of a subgraph at level i is

$$T_i^{bissection} = O\left(\frac{m/2^i}{p/2^i}\right) + O\left(\frac{n/2^i}{\sqrt{p/2^i}}\right) = O\left(\frac{m}{p}\right) + O\left(\frac{n}{\sqrt{p}}\right),$$

the same for all levels. Thus, the overall run time of the parallel p-way partitioning algorithm is

$$T^{partition} = \left(O\left(\frac{m}{p}\right) + O\left(\frac{n}{\sqrt{p}}\right)\right)\log p = O\left(\frac{n\log p}{\sqrt{p}}\right) \tag{2}$$

Equation 2 shows that asymptotically, only a speedup of $O(\sqrt{p})$ can be achieved in the algorithm. However, as our experiments in Section 6 show, higher speedup can be obtained. This is because the constant hidden in front of $O(m/p)$ is often much higher than that hidden in front of $O(n/\sqrt{p})$ particularly for 3D finite element graphs. Nevertheless, from Equation 2 we have that the partitioning algorithm is asymptotically unscalable. That is, it is not possible to obtain constant efficiency on increasingly large number of processors even if the problem size ($O(n)$) is increased arbitrarily.

However, a linear system solver that uses this parallel multilevel partitioning algorithm to obtain a fill reducing ordering prior to Cholesky factorization is not unscalable. This is because, the time spent in ordering is considerably smaller than the time spent in Cholesky factorization. The sequential complexity of Cholesky factorization of matrices arising in 2D and 3D finite elements applications is $O(n^{1.5})$ and $O(n^2)$, respectively. The communication overhead of parallel ordering over all p processors is $O(n\sqrt{p}\log p)$, which can be subsumed by the serial complexity of Cholesky factorization provided n is large enough relative to p. In particular, the isoefficiency [19] for 2D finite element graphs is $O(p^{1.5}\log^3 p)$, and for 3D finite element graphs is $O(p\log^2 p)$. We have recently developed a highly parallel sparse direct factorization algorithm [17, 9]. the isoefficiency of this algorithm is $O(p^{1.5})$ for both 2D and 3D finite element graphs. Thus, for 3D problems, the parallel ordering does not affect the overall scalability of the ordering-factorization algorithm.

6 Experimental Results

We evaluated the performance of the parallel multilevel graph partitioning and sparse matrix ordering algorithm on a wide range of matrices arising in finite element applications. The characteristics of these matrices 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 chip. 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 matrices 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 algorithm running on a SGI Challenge with 1.2GBytes of memory and 150MHz Mips R4400. Even though

Matrix Name	No. of Vertices	No. of Edges	Description
4ELT	15606	45878	2D Finite element mesh
BCSSTK31	35588	572914	3D Stiffness matrix
BCSSTK32	44609	985046	3D Stiffness matrix
BRACK2	62631	366559	3D Finite element mesh
CANT	54195	1960797	3D Stiffness matrix
COPTER2	55476	352238	3D Finite element mesh
CYLINDER93	45594	1786726	3D Stiffness matrix
ROTOR	99617	662431	3D Finite element mesh
SHELL93	181200	2313765	3D Stiffness matrix
WAVE	156317	1059331	3D Finite element mesh

Table 1: Various matrices used in evaluating the multilevel graph partitioning and sparse matrix ordering algorithm.

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 15% slower than that running on the SGI. The computed speedups in the rest of this section are scaled to take this into account[1]. All times reported are in seconds. Since our multilevel algorithm uses randomization in the coarsening step, we performed all experiments with a fixed seed.

6.1 Graph Partitioning

The performance of the parallel multilevel algorithm for the matrices in Table 1 is shown in Table 2 for a p-way partition on p processors, where p is 16, 32, 64, and 128. The performance of the serial multilevel algorithm for the same set of matrices running on an SGI is shown in Table 3. For both the parallel and the serial multilevel algorithm, the edge-cut and the run time are shown in the corresponding tables. In the rest of this section we will first compare the quality of the partitions produced by the parallel multilevel algorithm, and then the speedup obtained by the parallel algorithm.

	$p = 16$			$p = 32$			$p = 64$			$p = 128$		
Matrix	T_p	EC_{16}	S	T_p	EC_{32}	S	T_p	EC_{64}	S	T_p	EC_{128}	S
4ELT	0.48	1443	6.0	0.48	1995	7.0	0.48	3210	8.5	0.48	4734	11.1
BCSSTK31	1.28	27215	10.7	1.02	43832	17.0	0.87	67134	23.6	0.78	98675	31.6
BCSSTK32	1.69	43987	12.0	1.33	71378	19.2	1.05	104532	28.4	0.92	155321	37.9
BRACK2	2.14	14987	8.6	1.83	21545	12.2	1.56	32134	16.8	1.35	45345	21.9
CANT	3.20	199567	13.4	2.29	322498	23.7	1.71	441459	38.0	1.47	575231	49.7
COPTER2	2.05	22498	7.4	1.78	32765	11.1	1.59	45230	14.0	1.42	60543	18.2
CYLINDER93	2.35	131534	14.3	1.71	198675	24.5	1.34	288340	39.2	1.05	415632	56.3
ROTOR	3.16	26532	11.0	2.89	39785	14.4	2.40	57540	20.0	2.10	77450	26.4
SHELL93	5.80	54765	13.9	4.40	86320	22.5	3.25	130856	35.3	2.67	200057	49.9
WAVE	5.10	57543	10.3	4.70	76785	13.3	3.73	101210	19.9	3.09	138245	26.8

Table 2: The performance of the parallel multilevel graph partitioning algorithm. For each matrix, the performance is shown for 16, 32, 64, and 128 processors. T_p is the parallel run time for a p-way partition on p processors, EC_p is the edge-cut of the p-way partition, and S is the speedup over the serial multilevel algorithm.

Figure 6 shows the size of the edge-cut of the parallel multilevel algorithm compared to the serial multilevel algorithm. Any bars above the baseline indicate that the parallel algorithm produces partitions with higher edge-cut than the serial algorithm. From this graph we can see that for most matrices, the edge-cut of the parallel algorithm is worse than that of the serial algorithm. This is due to the fact that the coarsening and refinement performed by the parallel

[1]The speedup is computed as $1.15 * T_{SGI}/T_{T3D}$, where T_{SGI} and T_{T3D} are the run times on SGI and T3D, respectively.

Matrix	T_{16}	EC_{16}	T_{32}	EC_{32}	T_{64}	EC_{64}	T_{128}	EC_{128}
4ELT	2.49	1141	2.91	1836	3.55	2965	4.62	4600
BCSSTK31	11.96	25831	15.08	42305	17.82	65249	21.40	97819
BCSSTK32	17.62	43740	22.21	70454	25.92	106440	30.29	152081
BRACK2	16.02	14679	19.48	21065	22.78	29983	25.72	42625
CANT	37.32	199395	47.22	319186	56.53	442398	63.50	574853
COPTER2	13.22	21992	17.14	31364	19.30	43721	22.50	58809
CYLINDER93	29.21	126232	36.48	195532	45.68	289639	51.39	416190
ROTOR	30.13	24515	36.09	37100	41.83	53228	48.13	75010
SHELL93	69.97	51687	86.23	81384	99.65	124836	115.86	185323
WAVE	45.75	51300	54.37	71339	64.44	97978	71.98	129785

Table 3: The performance of the serial multilevel graph partitioning algorithm on an SGI, for 16-, 32-, 64-, and 128-way partition. T_p is the run time for a p-way partition, and EC_p is the edge-cut of the p-way partition.

algorithm are less powerful. But in most cases, the difference in edge-cut is quite small. For nine out of the ten matrices, the edge-cut of the parallel algorithm is within 10% of that of the serial algorithm. Furthermore, the difference in quality decreases as the number of partitions increases. The only exception is 4ELT, for which the edge-cut of the parallel 16-way partition is about 27% worse than the serial one. However, even for this problem, when larger partitions are considered, the relative difference in the edge-cut decreases; and for the of 128-way partition, parallel multilevel does slightly better than the serial multilevel.

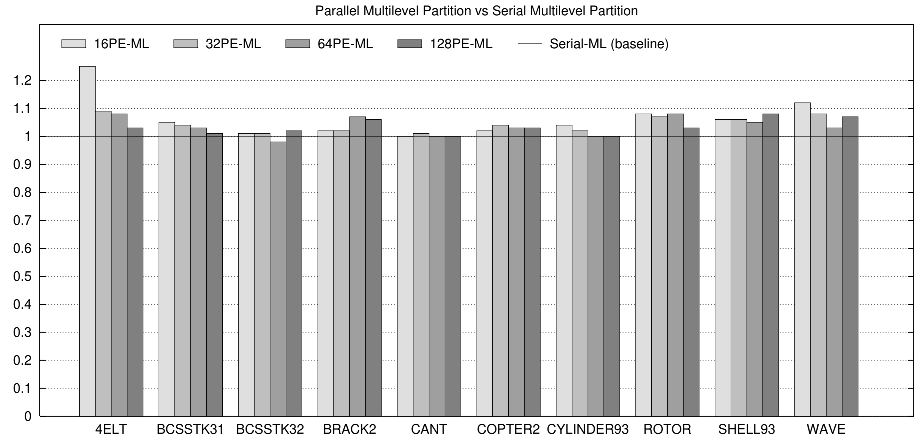


Figure 6: Quality (size of the edge-cut) of our parallel multilevel algorithm relative to the serial multilevel algorithm.

Figure 7 shows the size of the edge-cut of the parallel algorithm compared to the Multilevel Spectral Bisection algorithm (MSB) [3]. The MSB algorithm is a widely used algorithm that has been found to generate high quality partitions with small edge-cuts. We used the Chaco [11] graph partitioning package to produce the MSB partitions. As before, any bars above the baseline indicate that the parallel algorithm generates partitions with higher edge-cuts. From this figure we see that the quality of the parallel algorithm is almost never worse than that of the MSB algorithm. For eight out of the ten matrices, the parallel algorithm generated partitions with fewer edge-cuts, up to 50% better in some cases. On the other hand, for the matrices that the parallel algorithm performed worse, it is only by a small factor (less than 6%). This figure (along with Figure 6) also indicates that our serial multilevel algorithm outperforms the MSB algorithm. An extensive comparison between our serial multilevel algorithm and MSB, can be found in [16].

Tables 2 and 3 also show the run time of the parallel algorithm and the serial algorithm, respectively. A number of conclusions can be drawn from these results. First, as p increases, the time required for the p-way partition on p-

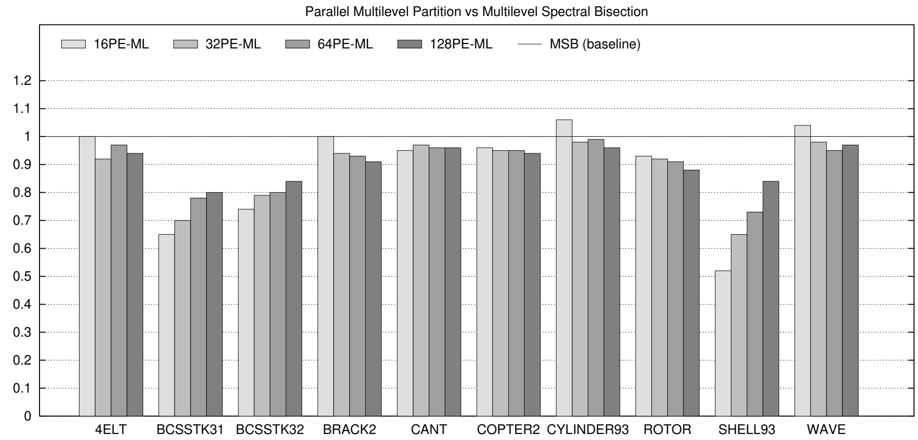


Figure 7: Quality (size of the edge-cut) of our parallel multilevel algorithm relative to the multilevel spectral bisection algorithm.

processors decreases. Depending on the size and characteristics of the matrix this decrease is quite substantial. The decrease in the parallel run time is not linear to the increase in p but somewhat smaller for the following reasons: (a) As p increases, the time required to perform the p-way partition also increases; (there are more partitions to perform). (b) The parallel multilevel algorithm incurs communication and idling overhead that limits the asymptotic speedup to $O(\sqrt{p})$ unless a good partition of the graph is available even before the partitioning process starts (Section 5).

To compare the decrease in the parallel run time against various ideal situations, we constructed Figure 8. In this figure we plotted the decrease in the run time for p equal to 32, 64, and 128, relative to the run time of $p = 16$ for some representative matrices from Table 1. On the same graph, we also plotted the decrease in the run time if the speedup had been $O(p)$, $O(\sqrt{p})$, and $O(\log p)$, respectively. From Section 5, we know that the asymptotic speedup obtained by the parallel multilevel algorithm is bounded by $O(\sqrt{p})$. Also, from Section 3, we know that if only the parallelism due to the recursive step is exploited, then the speedup of the parallel algorithm is bounded by $O(\log p)$. From this figure, we see that for the larger problems, the decrease in the run time is much larger than that predicted by the $O(\sqrt{p})$ speedup curve. This is because (as discussed in Section 5) the constants in Equation 2, for the $O(m/p)$ term are much smaller than those of the $O(n/\sqrt{p})$ term for graphs with relatively high average degree.

Thus, for problems with relatively high degree, (such as coefficient matrices for 3D finite element problems), our parallel algorithm performs fairly well. This can also be seen by looking at the speedup achieved by the parallel algorithm shown in Table 2. We see that for most such problems, speedup in the range of 14 to 25 was achieved on 32 processors, and in the range of 22 to 56 on 128 processors. Since, the serial multilevel algorithm is quite fast (much faster than MSB), these speedups lead to a significant reduction in the time required to perform the partition. For most problems in our test set, it takes no more than two seconds to obtain an 128-partition on 128 processors. However, for the problems with small average degrees, the decrease is very close to $O(\sqrt{p})$ as predicted by our analysis. The only exception is 4ELT, for which the speedup is close to $O(\log p)$. We suspect it is because the problem is too small, as even on a serial computer 128-way partition takes less than 5 seconds.

6.2 Sparse Matrix Ordering

We used the parallel multilevel graph partitioning algorithm to find a fill reducing ordering via nested dissection. The performance of the parallel multilevel nested dissection algorithm (MLND) for various matrices is shown in Table 4. For each matrix, the table shows the parallel run time and the number of nonzeros in the Cholesky factor L of the resulting matrix for 16, 32, and 64 processors. On p processors, the ordering is computed by using nested dissection

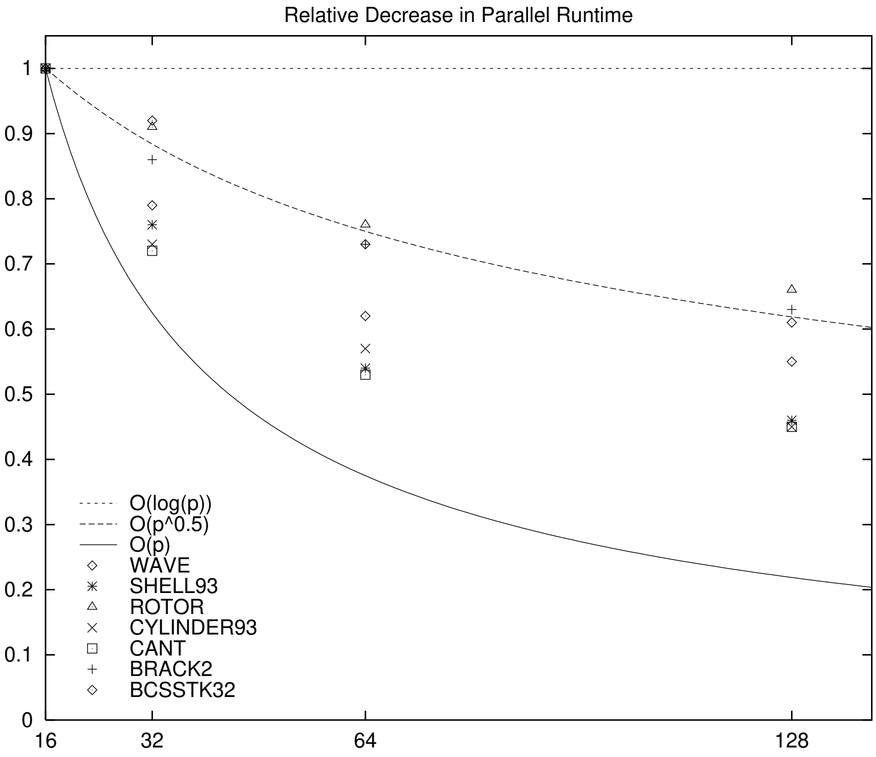


Figure 8: The decrease in the parallel run time for 32, 64, and 128 processors relative to the run time on 16 processors.

for the first log p levels, and then multiple minimum degree [20] (MMD) is used to order the submatrices stored locally on each processor.

Matrix	T_{16}	$\|L\|$	T_{32}	$\|L\|$	T_{64}	$\|L\|$
BCSSTK31	1.7	5588914	1.3	5788587	1.0	6229749
BCSSTK32	2.2	7007711	1.7	7269703	1.3	7430756
BRACK2	2.9	7788096	2.5	7690143	1.8	7687988
CANT	4.4	29818759	2.8	28854330	2.2	28358362
COPTER2	2.6	12905725	2.1	12835682	1.6	12694031
CYLINDER93	3.5	15581849	2.2	15662010	1.7	15656651
ROTOR	6.1	23193761	4.0	24196647	3.0	24624924
SHELL93	8.5	40968330	5.7	40089031	4.5	35174130
WAVE	8.7	87657783	6.3	85317972	4.8	87243325

Table 4: The performance of the parallel MLND algorithm on 16, 32, and 64 processors for computing a fill reducing ordering of a sparse matrix. T_p is the run time in seconds and $|L|$ is the number of nonzeros in the Cholesky factor of the matrix.

Figure 9 shows the relative quality of both serial and parallel MLND versus the MMD algorithm. These graphs were obtained by dividing the number of operations required to factor the matrix using MLND by that required by MMD. Any bars above the baseline indicate that the MLND algorithm requires more operations than the MMD algorithm. From this graph, we see that in most cases, the serial MLND algorithm produces orderings that require fewer operations than MMD. The only exception is BCSSTK32, for which the serial MLND requires twice as many operations.

Comparing the parallel MLND algorithm against the serial MLND, we see that the orderings produced by the parallel algorithm requires more operations (see Figure 9). This is mainly due to the following three reasons:

a. The bisections produced by the parallel multilevel algorithm are somewhat worse than those produced by the serial algorithm.

b. The parallel algorithm uses an approximate minimum cover algorithm (Section 4). Even though, this approximate algorithm finds a small separator, its size is somewhat larger than that obtained by the minimum cover algorithm used in serial MLND. From some matrices, the true minimum cover separator may be up to 15% smaller than the approximate one. As a result, the orderings produced by the parallel MLND require more operations than the serial MLND.

c. The parallel algorithm performs multilevel nested dissection ordering only for the first log p levels. After that it switches over to MMD. The serial MLND algorithm performs $O(\log n)$ levels of nested dissection and only switches to MMD when the submatrix is very small (fewer than 100 vertices). On the other hand, depending on p, the parallel MLND algorithm switches to MMD much earlier. Since, MLND tends to perform better than MMD for larger matrices arising in 3D finite element problems (Figure 9), the overall quality of the ordering produced by parallel MLND can be slightly worse. This effect becomes less pronounced as p increases, because MMD is used on smaller and smaller submatrices. Indeed on some problems (such as CANT and SHELL93), parallel MLND performs better as the number of processors increases.

However, as seen in Figure 9, the overall quality of the parallel MLND algorithm is usually within 20% of the serial MLND algorithm. The only exception in Figure 9 is SHELL93. Also, the relative quality changes slightly as the number of processors used to find the ordering increases.

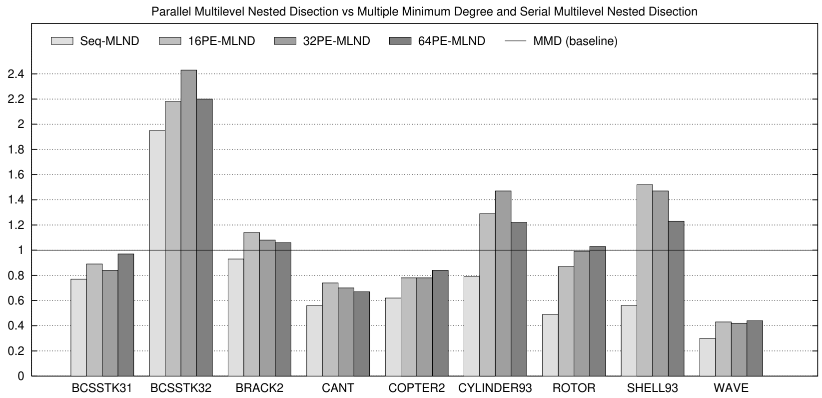


Figure 9: Quality of our parallel MLND algorithm relative to the multiple minimum degree algorithm and the serial MLND algorithm.

Comparing the run time of the parallel MLND algorithm (Table 4) with that of the parallel multilevel partitioning algorithm (Table 2) we see that the time required by ordering is somewhat higher than the corresponding partitioning time. This is due to the extra time taken by the approximate minimum cover algorithm and the MMD algorithm used during ordering. But the relative speedup between 16 and 64 processors for both cases are quite similar.

6.3 How Good Is the Diagonal Coarsening

The analysis presented in Section 5 assumed that the size of the graph (*i.e.*, the number of edges) decreases by a factor greater than four before successive foldings. The amount of coarsening that can take place depends on the number of edges stored locally on each diagonal processor. If this number is very small, then maximal independent subsets found by each diagonal processor will be very small. Furthermore, the next level coarser graph will have even a smaller number of edges, since (a) edges in the matching are removed, and (b) some of the edges of the matched vertices are

Graph	p	4×4		2×2		1×1	
		V	E	V	E	V	E
CANT	16			74.34	241.95	1.72	2.26
$d = 72$	64	16.71	39.09	2.63	3.51	1.78	2.20
BCSSTK31	16			49.23	118.57	1.90	2.50
$d = 44$	64	8.67	29.06	2.67	2.99	2.60	3.20
BCSSTK32	16			16.04	32.55	2.67	3.73
$d = 32$	64	4.99	7.43	2.94	3.40	1.98	2.33
BRACK2	16			7.43	7.06	2.48	3.14
$d = 12$	64	6.84	6.41	1.83	1.93	1.82	2.06
ROTOR	16			12.31	12.60	1.90	2.06
$d = 12$	64	5.07	4.77	2.47	2.70	1.85	1.99
COPTER2	16			7.61	7.64	1.89	2.04
$d = 12$	64	2.77	2.59	1.96	1.92	2.62	3.02
4ELT	16			2.01	2.04	4.48	5.76
$d = 6$	64	1.30	1.31	1.72	1.76	4.19	5.50

Table 5: The reduction in the number of vertices and edges between successive graph foldings. For each graph, the columns labeled with V (E) gives the factor by which the number of vertices (edges) is reduced between successive foldings. These results are shown for three processor grids 4×4, 2×2 and 1×1 that corresponds to the quadrant of the grid to which the graph was folded. For example, for BCSSTK32, when 64 processors are used, the number of vertices was reduced by a factor of 4.99 before being folded to 16 processors (4×4 grid). For the same graph, the number of vertices was reduced further by a factor of 2.94 before being folded to 4 processors, and by another 1.98 factor before being folded down to a single processor. Thus, the graph that a single processor receives has $4.99 \times 2.94 \times 1.98 = 16.65$ times fewer vertices than the original graph. Under each graph, the average degree d of the graph is shown.

common and thus are collapsed together. On the other hand, if the average degree of a vertex is fairly high, then significant coarsening can be performed before folding. To illustrate the relation between the average degree of a graph and the amount of coarsening that can be performed for the first bisection, we performed a number of experiments on 16 and 64 processors. In Table 5 we show the reduction in the number of vertices and edges between foldings.

A number of interesting conclusions can be drawn out of this table. For graphs with relatively high average degree (*e.g.*, CANT, BCSSTK31, BCSSTK32), most of the coarsening is performed on the entire processor grid. For instance, on 64 processors, for CANT, the average degree of a vertex on the diagonal processors is $72/8 = 9$. As a result significant coarsening can be performed before the edges of the diagonal processors get depleted. By the time the parallel multilevel algorithm is forced to perform a folding, both the number of vertices and the number of edges have decreased by a large factor. In many cases, this factor is substantially higher than that required by the analysis. For most of these graphs, over 90% of the overall computation of coarsening is performed using all the processors, and only a small fraction is performed by smaller processor grids. This type of graphs correspond to the coefficient matrices of 3D finite element problems with multiple degrees of freedom that are widely used in scientific and engineering applications. From this we can see that the diagonal coarsening can easily be scaled up to 256 or even 1024 processors for graphs with average degree higher than 25 or 40 respectively.

For low degree graphs (*e.g.* BRACK2, COPTER2, and ROTOR) with average degree of 12 or less, the number of vertices decreases by a smaller factor than the high degree graphs. For BRACK2, for each vertex, each diagonal processor has on the average $12/8 = 1.5$ vertices; thus, only limited amount of coarsening can be performed. Note that for 4ELT, the number of vertices and the number of edges decrease only by a small factor, which explains the poor speedup obtained for this problem.

7 Conclusion

In this paper we presented a parallel formulation of multilevel recursive bisection algorithm for partitioning a graph and for producing a fill reducing order via nested dissection. Our experiments show that our parallel algorithms are able to produce good partitions and orderings for a wide range of problems. Furthermore, our algorithms achieve a speedup of up to 56 on 128-processor Cray T3D.

Due to the two-dimensional mapping scheme used in the parallel formulation, its asymptotic speedup is limited to $O(\sqrt{p})$ because the matching operation is performed only on the diagonal processors. In contrast, for one-dimensional mapping scheme used in [24, 1, 14], the asymptotic speedup can be $O(p)$ for large enough graphs. However, this two-dimensional mapping has the following advantages. First, the actual speedup on graphs with large average degrees is quite good as shown in Figure 8. The reason is that for these graphs, the formation of the next level coarser graph (which is completely parallel with two-dimensional mapping) dominates the computation of the matching. Second, the two-dimensional mapping requires fewer communication operations (only broadcast and reduction operations along the rows and columns of the processor grid) in each coarsening step compared with one-dimensional mapping. Hence on machines with slow communication network (high message startup-time and/or small communication bandwidth), the two-dimensional mapping can provide better performance even for graphs with small degree. Third, the two-dimensional mapping is central to the parallelization of the minimal vertex cover computation presented in Section 4. It is unclear if the algorithm for computing minimal vertex cover of a bipartite graph can be efficiently parallelized with one-dimensional mapping.

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] 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.

[6] C. M. Fiduccia and R. M. Mattheyses. A linear time heuristic for improving network partitions. In *In Proc. 19th IEEE Design Automation Conference*, pages 175–181, 1982.

[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 and J. W.-H. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[9] Anshul Gupta, George Karypis, and Vipin Kumar. Highly scalable parallel algorithms for sparse matrix factorization. *IEEE Transactions on Parallel and Distributed Systems*, 8(5):502–520, May 1997. Available on WWW at URL http://www.cs.umn.edu/˜karypis.

[10] M. T. Heath, E. G.-Y. Ng, and Barry W. Peyton. Parallel algorithms for sparse linear systems. *SIAM Review*, 33:420–460, 1991. Also appears in K. A. Gallivan et al. *Parallel Algorithms for Matrix Computations*. SIAM, Philadelphia, PA, 1990.

[11] Bruce Hendrickson and Robert Leland. The chaco user's guide, version 1.0. Technical Report SAND93-2339, Sandia National Laboratories, 1993.

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

[13] 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. A short version appears in Supercomputing 95.

[14] 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. A short version appears in Supercomputing 96.

[15] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, Accepted for publication, 1997. Also available on WWW at URL http://www.cs.umn.edu/˜karypis.

[16] G. Karypis and V. Kumar. A fast and highly quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, to appear. Also available on WWW at URL http://www.cs.umn.edu/˜karypis. A short version appears in Intl. Conf. on Parallel Processing 1995.

[17] 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.

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

[19] 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.

[20] J. W.-H. Liu. Modification of the minimum degree algorithm by multiple elimination. *ACM Transactions on Mathematical Software*, 11:141–153, 1985.

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

[22] 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.

[23] A. Pothen and C-J. Fan. Computing the block triangular form of a sparse matrix. *ACM Transactions on Mathematical Software*, 1990.

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

[25] 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.

[26] J. E. Savage and M. G. Wloka. Parallelism in graph partitioning. *J. Par. Dist. Computing*, 13:257–272, 1991.

[27] Horst D. Simon and Shang-Hua Teng. How good is recursive bisection? Technical Report RNR-93-012, NAS Systems Division, Moffet Field, CA, 1993.