MULTILEVEL GRAPH PARTITIONING SCHEMES*

George Karypis and Vipin Kumar
Department of Computer Science, University of Minnesota, Minneapolis, MN 55455
{karypis, kumar}@cs.umn.edu

Abstract – In this paper we present experiments with a class of graph partitioning algorithms that reduce the size of the graph by collapsing vertices and edges, partition the smaller graph, and then uncoarsen it to construct a partition for the original graph. We investigate the effectiveness of many different choices for all three phases: coarsening, partition of the coarsest graph, and refinement. In particular, we present a new coarsening heuristic (called heavy-edge heuristic) for which the size of the partition of the coarse graph is within a small factor of the size of the final partition obtained after multilevel refinement. We also present a new scheme for refining during uncoarsening that is much faster than the Kernighan-Lin refinement. We test our scheme on a large number of graphs arising in various domains including finite element methods, linear programming, VLSI, and transportation. Our experiments show that our scheme consistently produces partitions that are better than those produced by spectral partitioning schemes in substantially smaller timer (10 to 35 times faster than multilevel spectral bisection). Also, when our scheme is used to compute fill reducing orderings for sparse matrices, it substantially outperforms the widely used multiple minimum degree algorithm.

1 Introduction

Graph partitioning is an important problem that has extensive applications in many areas, including scientific computing and VLSI design. The problem is to partition the vertices of a graph in 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. The problem of minimizing communication in this step is identical to the problem of partitioning the graph corresponding to the matrix A [26]. 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 [26, 9]. The multiple minimum degree ordering used almost exclusively in serial direct methods is not suitable for parallel direct methods, as it provides very little 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. Spectral partitioning methods are known to produce excellent partitions for a wide class of problems, and they are used quite extensively [33, 20]. However, these methods are very expensive since they require the computation of the eigenvector corresponding to the second smallest eigenvalue (Fiedler vector). Execution of the spectral methods can be speeded up if computation of the Fiedler vector is done by using a multilevel algorithm [2]. This multilevel spectral bisection algorithm (MSB) usually manages to speedup the spectral partitioning methods by an order of magnitude without any loss in the quality of the edge-cut. However, even MSB can take a large amount of time. In particular, in parallel direct solvers, the time for computing ordering using MSB can be several orders of magnitude higher than the time taken by the parallel factorization algorithm, and thus ordering time can dominate the overall time to solve the problem [14]. The execution time of MSB can be further speeded up by computing the Fiedler vector in parallel. The algorithm for computing the Fiedler vector, is iterative and in each iteration it performs a matrix-vector multiplication of a matrix whose graph is identical to the one we are trying to partition. These matrix-vector products can be performed efficiently on a parallel computer only if a good partition of the graph is available—a problem that MSB is trying to solve in the first place. As a result, parallel implementation of spectral methods exhibit poor efficiency since most of the time is spent in performing communication [21, 1].

Another class of graph partitioning techniques uses the geometric information of the graph to find a good partition. Geometric partitioning algorithms [17, 28, 29] tend to be fast but often yield partitions that are worse than those obtained by spectral methods. Among the most prominent of these scheme is the algorithm described in [28]. This algorithm produces partitions that are provably within the bounds that exist for some special classes of graphs. However, due to the randomized nature of these algorithms,

*This work is sponsored by the AHPCRC under the auspices of the DoA, ARL 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 Cray Research Inc. Related papers are available via WWW at URL: http://www.cs.umn.edu/users/kumar/papers.html

multiple trials are often required to obtain solutions that are comparable in quality to spectral methods. Multiple trials do increase the time [13], but the overall runtime is still substantially lower than the time required by the spectral methods. However, geometric graph partitioning algorithms have limited applicability because often the geometric information is not available, and in certain problem areas (*e.g.*, linear programming), there is no geometry associated with the graph. Recently, an algorithm has been proposed to compute geometry information for graphs [4]. However this algorithm is based on computing spectral information, which is expensive and dominates the overall time taken by the graph partitioning algorithm.

Another class of graph partitioning algorithms reduce the size of the graph (*i.e.*, coarsen the graph) by collapsing vertices and edges, partition the smaller graph, and then uncoarsen it to construct a partition for the original graph. These are called multilevel graph partitioning schemes [3, 5, 15, 20, 7, 30]. Some researchers investigated multilevel schemes primarily to decrease the partitioning time, at the cost of somewhat worse partition quality [30]. Recently, a number of multilevel algorithms have been proposed [3, 20, 5, 15, 7] that further refine the partition during the uncoarsening phase. These schemes tend to give good partitions at reasonable cost. In particular, the work of Hendrickson and Leland [20] showed that multilevel schemes can provide better partitions than the spectral methods at lower cost for a variety of finite element problems. Their scheme uses random maximal matching to successively coarsen the graph until it has only a few hundred vertices. Then it partitions this small graph using the spectral methods. Now it uncoarsens the graph level by level, and applies Kernighan-Lin refinement periodically. However, even-though multilevel algorithms have been shown to be good alternatives to both spectral and geometric algorithms, there is no comprehensive study today on their effectiveness on a wide range of problems.

In this paper we experiment with various parameters of multilevel algorithms, and their effect on the quality of partition and ordering. We investigate the effectiveness of many different choices for all three phases: coarsening, partition of the coarsest graph, and refinement. In particular, we present a new coarsening heuristic (called heavy-edge heuristic) for which the size of the partition of the coarse graph is within a small factor of the size of the final partition obtained after multilevel refinement. We also present a new scheme for refining during uncoarsening that is much faster than the Kernighan-Lin refinement used in [20].

We test our scheme on a large number of graphs arising in various domains including finite element methods, linear programming, and VLSI. Our experiments show that our scheme consistently produces partitions that are better than those produced by spectral partitioning schemes in substantially smaller timer (10 to 35 times faster than multilevel spectral bisection). Compared with the scheme of [20], our scheme is about twice as fast, and is consistently better in terms of cut size. Much of the improvement in run time comes from our faster refinement heuristic. We also used our graph partitioning scheme to compute fill reducing orderings for sparse matrices. Surprisingly, our scheme substantially outperforms the multiple minimum degree algorithm [27], which is the most commonly used method for computing fill reducing orderings of a sparse matrix.

Even though multilevel algorithms are quite fast compared with spectral methods, they can still be the bottleneck if the sparse system of equations is being solved in parallel [26, 14]. The coarsening phase of these methods is easy to parallelize [23], but the Kernighan-Lin heuristic used in the refinement phase is very difficult to speedup in parallel computers [12]. Since both the coarsening phase and the refinement phase with Kernighan-Lin heuristic take roughly the same amount of time, the overall scheme cannot be speeded up significantly. Our new faster methods for refinement reduce this bottleneck substantially. In fact our parallel implementation [23] of this multilevel partitioning is able to get a speedup of as much as 56 on a 128-processor Cray T3D for moderate size problems.

2 Graph Partitioning

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 partition 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 partition P, the number of edges whose incident vertices belong to different subsets is called the ***edge-cut*** of the partition.

The efficient implementation of many parallel algorithms usually requires the solution to a graph partitioning problem, where vertices represent computational tasks, and edges represent data exchanges. A k-way partition of the computation graph can be used to assign tasks to k processors. Because the partition assigns equal number of computational tasks to each processor the work is balanced among k processors, and because it minimizes the edge-cut, the communication overhead is also minimized.

Another important application of recursive bisection is to find a fill reducing ordering for sparse matrix factorization [9, 26, 16]. This type of algorithms are generally referred to as nested dissection ordering algorithms. Nested dissection recursively splits a graph into almost equal halves by selecting a vertex separator until the desired number of par-

titions are obtained. The vertex separator is determined by first bisecting the graph and then computing a vertex separator from the edge separator. The vertices of the graph are numbered such that at each level of recursion, the separator vertices are numbered after the vertices in the partitions. The effectiveness and the complexity of a nested dissection scheme depends on the separator computing algorithm. In general, small separators result in low fill-in.

The k-way partition 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 $\log k$ phases, graph G is partitioned into k parts. Thus, the problem of performing a k-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, it is used extensively due to its simplicity [9, 16].

3 Multilevel Graph Bisection

The graph G can be bisected using a multilevel algorithm. The basic structure of a multilevel algorithm is very simple. The graph G is first coarsened down to a few hundred vertices, a bisection of this much smaller graph is computed, 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 usually decrease the edge-cut.

Formally, a multilevel graph bisection algorithm works as follows: Consider a weighted graph $G_0 = (V_0, E_0)$, with weights both on vertices and edges. A multilevel graph bisection algorithm consists of the following three phases.

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 2-way partition P_m of the graph $G_m = (V_m, E_m)$ is computed that partitions V_m into two parts, each containing half the vertices of G_0.

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

3.1 Coarsening Phase

During the coarsening phase, a sequence of smaller graphs, each with fewer vertices, is constructed. Graph coarsening can be achieved in various ways. In most coarsening schemes, a set of vertices of G_i is combined together to form a single vertex of the next level coarser graph G_{i+1}. Let V_i^v be the set of vertices of G_i combined to form vertex v of G_{i+1}. We will refer to vertex v as a ***multinode***. In order for a bisection of a coarser graph to be good with respect to the original graph, the weight of vertex v is set equal to the sum of the weights of the vertices in V_i^v. Also, in order to preserve the connectivity information in the coarser graph, the edges of v are the union of the edges of the vertices in V_i^v. In the case where more than one vertex of V_i^v, contain edges to the same vertex u, the weight of the edge of v is equal to the sum of the weights of these edges. This is useful when we evaluate the quality of a partition at a coarser graph. The edge-cut of the partition in a coarser graph will be equal to the edge-cut of the same partition in the finer graph.

Two main approaches have been proposed for obtaining coarser graphs. The first approach is based on finding a random matching and collapsing the matched vertices into a multinode [3, 20, 2], while the second approach is based on creating multinodes that are made of groups of vertices that are highly connected [5, 15, 7]. The later approach is suited for graphs arising in VLSI applications, since these graphs have highly connected components. However, for graphs arising in finite element applications, most vertices have similar connectivity patterns (*i.e.*, the degree of each vertex is fairly close to the average degree of the graph). In the rest of this section we describe the basic ideas behind coarsening using matchings.

Given a graph $G_i = (V_i, E_i)$, a coarser graph can be obtained by collapsing adjacent vertices. Thus, the edge between two vertices is collapsed and a multinode consisting of these two vertices is created. This edge collapsing idea can be formally defined in terms of matchings. A ***matching*** of a graph, is a set of edges, no two of which are incident on the same vertex. Thus, the next level coarser graph G_{i+1} is constructed from G_i by finding a matching of G_i and collapsing the vertices being matched into multinodes. The unmatched vertices are simply copied over to G_{i+1}. Since the goal of collapsing vertices using matchings is to decrease the size of the graph G_i, the matching should be of maximal size. That is, it should contain all possible the edges, no two of which are incident on the same vertex. The matching of maximal size is called ***maximal matching***. Note that depending on how matchings are computed, the size of the maximal matching may be different.

In the remaining sections we describe four ways that we used to select maximal matchings for coarsening. The complexity of all these schemes is $O(|E|)$.

Random Matching (RM) A maximal matching can be generated efficiently using a randomized algorithm. In our experiments we used a randomized algorithm similar to that described in [3, 20]. The random maximal matching algorithm is the following. The vertices are visited in random order. If a vertex u has not been matched yet, then we randomly select one of its unmatched adjacent vertices. If such a vertex v exists, we include the edge (u, v) in the matching and mark vertices u and v as being matched. If there

is no unmatched adjacent vertex v, then vertex u remains unmatched in the random matching.

Heavy Edge Matching (HEM) While performing the coarsening using random matchings, we try to minimize the number of coarsening levels in a greedy fashion. However, our overall goal is to find a bisection that minimizes the edge-cut. Consider a graph $G_i = (V_i, E_i)$, a matching M_i that is used to coarsen G_i, and its coarser graph $G_{i+1} = (V_{i+1}, E_{i+1})$ induced by M_i. If A is a set of edges, define $W(A)$ to be the sum of the weights of the edges in A. It can be shown that $W(E_{i+1}) = W(E_i) - W(M_i)$. Thus, the total edge weight of the coarser graph is reduced by the weight of the matching. Hence, by selecting a matching M_i that has a maximal weight, we can maximize the decrease in the edge weight of the coarser graph. Now, since the coarser graph has smaller edge weight, it is more likely to have a smaller edge-cut.

Finding a matching with maximal weight is the idea behind the ***heavy-edge matching***. A maximal weight matching is computed using a randomized algorithm similar to that for computing a random matching described in Section 3.1. The vertices are again visited in random order. However, instead of randomly matching a vertex u with one of its adjacent unmatched vertices, we match u with the vertex v such that the weight of the edge (u, v) is maximum over all valid incident edges (heavier edge). Note that this algorithm does not guarantee that the matching obtained has maximum weight, but our experiments has shown that it works very well.

Light Edge Matching (LEM) Instead of minimizing the total edge weight of the coarser graph, one might try to maximize it. This is achieved by finding a matching M_i that has the smallest weight, leading to a small reduction in the edge weight of G_{i+1}. This is the idea behind the ***light-edge matching***. It may seem that the light-edge matching does not perform any useful transformation during coarsening. However, the average degree of G_{i+1} produced by LEM is significant higher than that of G_i. Graphs with high average degree are easier to partition using certain heuristics such as Kernighan-Lin [3].

Heavy Clique Matching (HCM) A ***clique*** of an unweighted graph $G = (V, E)$ is a fully connected subgraph of G. Consider a set of vertices U of V ($U \subset V$). The subgraph of G induced by U is defined as $G_U = (U, E_U)$, such that E_U consists of all edges $(v_1, v_2) \in E$ such that both v_1 and v_2 belong in U. Looking at the cardinality of U and E_U we can determined how close U is to a clique. In particular, the ratio $2|E_U|/(|U|(|U|-1))$ goes to one if U is a clique, and is small if U is far from being a clique. We refer to this ratio as ***edge density***.

The ***heavy clique matching*** scheme computes a matching by collapsing vertices that have high edge density. Thus, this scheme computes a matching whose edge density is maximal. The motivation behind this scheme is that subgraphs of G_0 that are cliques or almost cliques will most likely not be cut by the bisection. So, by creating multinodes that contain these subgraphs, we make it easier for the partitioning algorithm to find a good bisection. Note that this scheme tries to approximate the graph coarsening schemes that are based on finding highly connected components [5, 15, 7].

As in the previous schemes for computing the matching, we compute the heavy clique matching using a randomized algorithm. Note that HCM is very similar to the HEM scheme. The only difference is that HEM matches vertices that are only connected with a heavy edge irrespective of the contracted edge-weight of the vertices, whereas HCM matches a pair of vertices if they are both connected using a heavy edge and if each of these two vertices have high contracted edge-weight.

3.2 Partitioning Phase

The second phase of a multilevel algorithm is to compute a minimum edge-cut bisection P_m of the coarse graph $G_m = (V_m, E_m)$ such that each part contains roughly half of the vertex weight of the original graph.

A partition of G_m can be obtained using various algorithms such as (a) spectral bisection [33, 2, 18], (b) geometric bisection [28] (if coordinates are available), and (c) combinatorial methods [25, 8, 9]. Since the size of the coarser graph G_m is small (*i.e.*, $|V_m| < 100$), this step takes a small amount.

We implemented three different algorithms for partitioning the coarse graph. The first algorithm uses the spectral bisection [33], and the other two use graph growing heuristics. The first graph-growing heuristic (GGP) randomly selects a vertex v and grows a region around it in a breadth-first fashion until half of the vertex-weight has been included. The second graph-growing heuristic (GGGP) also starts from a randomly selected vertex v but it includes vertices that lead to the smaller increase in the edge-cut. Since the quality of the partitions obtained by GGP and GGGP depends on the choice of v, a number of different partitions are computed starting from different randomly selected vertices and the best is used as the initial partition. In the experiments in Section 4.1 we selected 10 vertices for GGP and 5 for GGGP. We found all of these partitioning schemes to produce similar partitions with GGGP consistently performing better.

3.3 Uncoarsening Phase

During the uncoarsening phase, the partition P_m 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 of G_{i+1} contains a distinct subset of vertices of G_i, obtaining P_i from P_{i+1} is done by simply assigning the vertices collapsed to $v \in G_i$ to the partition $P_{i+1}[v]$.

Even though P_{i+1} is a local minima partition of G_{i+1}, the projected partition P_i may not be at a local minima with respect to G_i. Since G_i is finer, it has more degrees of freedom that can be used to improve P_i, and decrease the edge-cut. Hence, it may still be possible to improve the projected partition of G_{i-1} 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 two subsets of vertices, one from each part such that when swapped the resulting partition has smaller edge-cut. Specifically, if A and B are the two parts of the bisection, a refinement algorithm selects $A' \subset A$ and $B' \subset B$ such that $A \backslash A' \cup B'$ and $B \backslash B' \cup A'$ is a bisection with a smaller edge-cut.

A class of algorithms that tend to produce very good results are those that are based on the Kernighan-Lin (KL) partition algorithm [25, 6, 20]. The KL algorithm is iterative in nature. It starts with an initial partition and in each iteration it finds subsets A' and B' with the above properties. If such subsets exist, then it moves them to the other part and this becomes the partition for the next iteration. The algorithm continues by repeating the entire process. If it cannot find two such subsets, then the algorithm terminates.

The KL algorithm we implemented is similar to that described in [6] with certain modifications that significantly reduce the run time. The KL algorithm, computes for each vertex v a quantity called ***gain*** which is the decrease (or increase) in the edge-cut if v is moved to the other part. The algorithm then proceeds by repeatedly selecting a vertex v with the largest gain from the larger part and moves it to the other part. After moving v, v is marked so it will not be considered again in the same iteration, and the gains of the vertices adjacent to v are updated to reflect the change in the partition. The algorithm terminates when the edge-cut does not decrease after x number of vertex moves. Since, the last x vertex moves did not decrease the edge-cut they are undone. The choice of $x = 50$ works quite well for all our graphs.

The efficient implementation of the above algorithm relies on the method used to compute the gains of successive finer graphs and the use of appropriate data structure to store these gains. Our algorithm computes the gains of the vertices during the projection of the partition. In doing so, it utilizes the computed gains for the vertices of the coarser graph and it only needs to compute the gains of the vertices that are along the boundary of the partition. The data structure used to store the gains is a hash table that allow insertions, updates, and extraction of the vertex with maximum gain in constant time. Details about the implementation of the KL algorithm can be found in [22].

In the next section we describe three different refinement algorithms that are based on the KL algorithm but differ in the time they require to do the refinement.

Kernighan-Lin Refinement The idea of Kernighan-Lin refinement (KLR) is to use the projected partition of G_{i+1} onto G_i as the initial partition for the Kernighan-Lin algorithm. The KL algorithm has been found to be effective in finding locally optimal partitions when it starts with a fairly good initial partition [3]. Since the projected partition is already a good partition, KL substantially decreases the edge-cut within a small number of iterations. Furthermore, since a single iteration of the KL algorithm stops as soon as x swaps are performed that do not decrease the edge-cut, the number of vertices swapped in each iteration is very small. Our experimental results show that a single iteration of KL terminates after only a small percentage of the vertices have been swapped (less than 5%), which results in significant savings in the total execution time of this refinement algorithm.

Greedy Refinement Since we terminate each pass of the KL algorithm as soon as no further improvement can be made in the edge-cut, the complexity of the KLR scheme described in the previous section is dominated by the time required to insert the vertices into the appropriate data structures. Thus, even though we significantly reduced the number of vertices that are swapped, the overall complexity does not change in asymptotic terms. Furthermore, our experience shows that the largest decrease in the edge-cut is obtained during the first pass. In the greedy refinement algorithm (GR), we take advantage of that by running only a single iteration of the KL algorithm [3]. This usually reduces the total time taken by refinement by a factor of two to four (Section 4.1).

Boundary Refinement In both the KLR and GR algorithms, we have to insert the gains of all the vertices in the data structures. However, since we terminate both algorithms as soon as we cannot further reduce the edge-cut, most of this computation is wasted. Furthermore, due to the nature of the refinement algorithms, most of the nodes swapped by either the KLR and the GR algorithm are along the boundary of the cut, which is defined to be the vertices that have edges that are cut by the partition.

In the boundary refinement algorithm, we initially insert into the data structures the gains for only the boundary vertices. As in the KLR algorithm, after we swap a vertex v, we update the gains of the adjacent vertices of v not yet being swapped. If any of these adjacent vertices become a boundary vertex due to the swap of v, we insert it into the data structures if the have positive gain. Notice that the boundary refinement algorithm is quite similar to the KLR algorithm, with the added advantage that only vertices are

inserted into the data structures as needed and no work is wasted.

As with KLR, we have a choice of performing a single pass (boundary greedy refinement (BGR)) or multiple passes (boundary Kernighan-Lin refinement (BKLR)) until the refinement algorithm converges. As opposed to the non-boundary refinement algorithms, the cost of performing multiple passes of the boundary algorithms is small, since only the boundary vertices are examined.

To further reduce the execution time of the boundary refinement while maintaining the refinement capabilities of BKLR and the speed of BGR one can combine these schemes into a hybrid scheme that we refer to it as BKLGR. The idea behind the BKLGR policy is to use BKLR as long as the graph is small, and switch to BGR when the graph is large. The motivation for this scheme is that single vertex swaps in the coarser graphs lead to larger decrease in the edge-cut than in the finer graphs. So by using BKLR at these coarser graphs better refinement is achieved, and because these graphs are very small (compared to the size of the original graph), the BKLR algorithm does not require a lot of time. For all the experiments presented in this paper, if the number of vertices in the boundary of the coarse graph is less than 2% of the number of vertices in the original graph, refinement is performed using BKLR, otherwise BGR is used.

4 Experimental Results

We evaluated the performance of the multilevel graph partitioning algorithm on a wide range of matrices arising in different application domains. The characteristics of these matrices are described in Table 1. All the experiments were performed on an SGI Challenge, with 1.2GBytes of memory and 200MHz Mips R4400. All times reported are in seconds. Since the nature of the multilevel algorithm discussed is randomized, we performed all experiments with fixed seed.

4.1 Graph Partitioning

As discussed in Sections 3.1, 3.2, and 3.3, there are many alternatives for each of the three different phases of a multilevel algorithm. It is not possible to provide an exhaustive comparison of all these possible combinations without making this paper unduly large. Instead, we provide comparison of different alternatives for each phase after making a reasonable choice for the other two phases.

Matching Schemes We implemented the four matching schemes described in Section 3.1 and the results for a 32-way partition for some matrices is shown in Table 2. These schemes are (a) random matching (RM), (b) heavy edge matching (HEM), (c) light edge matching (LEM), and (d) heavy clique matching (HCM). For all the experiments, we used the GGGP algorithm for the initial partition phase and the BKLGR as the refinement policy during the uncoarsening phase. For each matching scheme, Table 2 shows the edge-cut, the time required by the coarsening phase (CTime), and the time required by the uncoarsening phase (UTime). UTime is the sum of the time spent in partitioning the coarse graph (ITime), the time spent in refinement (RTime), and the time spent in projecting the partition of a coarse graph to the next level finer graph (PTime).

Matrix Name	Order	Nonzeros	Description
BCSSTK28 (BC28)	4410	107307	Solid element model
BCSSTK29 (BC29)	13992	302748	3D Stiffness matrix
BCSSTK30 (BC30)	28294	1007284	3D Stiffness matrix
BCSSTK31 (BC31)	35588	572914	3D Stiffness matrix
BCSSTK32 (BC32)	44609	985046	3D Stiffness matrix
BCSSTK33 (BC33)	8738	291583	3D Stiffness matrix
BCSPWR10 (BSP10)	5300	8271	Eastern US power network
BRACK2 (BRCK)	62631	366559	3D Finite element mesh
CANT (CANT)	54195	1960797	3D Stiffness matrix
COPTER2 (COPT)	55476	352238	3D Finite element mesh
CYLINDER93 (CY93)	45594	1786726	3D Stiffness matrix
FINAN512 (FINC)	74752	335872	Linear programming
4ELT (4ELT)	15606	45878	2D Finite element mesh
INPRO1 (INPR)	46949	1117809	3D Stiffness matrix
LHR71 (LHR)	70304	1528092	3D Coefficient matrix
LSHP3466 (LS34)	3466	10215	Graded L-shape pattern
MAP (MAP)	267241	937103	Highway network
MEMPLUS (MEM)	17758	126150	Memory circuit
ROTOR (ROTR)	99617	662431	3D Finite element mesh
S38584.1 (S33)	22143	93359	Sequential circuit
SHELL93 (SHEL)	181200	2313765	3D Stiffness matrix
SHYY161 (SHYY)	76480	329762	CFD/Navier-Stokes
TROLL (TROL)	213453	5885829	3D Stiffness matrix
WAVE (WAVE)	156317	1059331	3D Finite element mesh

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

	RM	HEM	LEM	HCM
BCSSTK31	14489	84024	412361	115471
BCSSTK32	184236	148637	680637	153945
BRACK2	75832	53115	187688	69370
CANT	817500	487543	1633878	521417
COPTER2	69184	59135	208318	59631
CYLINDER93	522619	286901	1473731	354154
4ELT	3874	3036	4410	4025
INPRO1	205525	187482	821233	141398
ROTOR	147971	110988	424359	98530
SHELL93	373028	237212	1443868	258689
TROLL	1095607	806810	4941507	883002
WAVE	239090	212742	745495	192729

Table 3: The edge-cut for a 32-way partition when no refinement was performed, for the various matching schemes.

In terms of the size of the edge-cut, there is no clear cut winner among the various matching schemes. The value of 32EC for all schemes are within 10% of each other. Out of these schemes, RM does better for 2 matrices, HEM does better for six matrices, LEM for three, and HCM for one.

The time spent in coarsening does not vary significantly across different schemes. But RM requires the least amount of time for coarsening, while LEM and HCM require the most (upto 38% more time than RM). This is not surprising since RM looks for the first unmatched neighbor of a vertex

	RM			HEM			LEM			HCM		
	32EC	CTime	UTime	32EC	CTime	UTime	32EC	CTime	UTime	32EC	CTime	UTime
BCSSTK31	44810	5.93	2.46	45991	6.55	1.95	42261	7.65	4.90	44491	7.48	1.92
BCSSTK32	71416	9.21	2.91	69361	10.26	2.34	69616	12.13	6.84	71939	12.06	2.36
BRACK2	20693	6.86	3.41	21152	7.54	3.33	20477	7.90	4.40	19785	8.07	3.42
CANT	323.0K	20.34	8.99	323.0K	22.39	5.74	325.0K	27.14	23.64	323.0K	26.19	5.85
COPTER2	32330	5.18	2.95	30938	6.39	2.68	32309	6.94	5.05	31439	7.25	2.73
CYLINDER93	198.0K	16.49	5.25	198.0K	18.65	3.22	199.0K	21.72	14.83	204.0K	21.61	3.24
4ELT	1826	0.82	0.76	1894	0.91	0.78	1992	0.92	0.95	1879	1.08	0.74
INPRO1	78375	10.40	2.90	75203	11.56	2.30	76583	13.46	6.25	78272	13.34	2.30
ROTOR	38723	12.94	5.60	36512	14.31	4.90	37287	15.51	8.30	37816	16.59	5.10
SHELL93	84523	36.18	10.24	81756	40.59	8.94	82063	46.02	16.22	83363	48.29	8.54
TROLL	317.4K	67.75	14.16	307.0K	74.21	10.38	305.0K	93.44	70.20	312.8K	89.14	10.81
WAVE	73364	20.87	8.24	72034	22.96	7.24	70821	25.60	15.90	71100	26.98	7.20

Table 2: Performance of various matching algorithms during the coarsening phase. *32EC* is the edge-cut of a 32-way partition, *CTime* is the time spent in coarsening, and *RTime* is the time spent in refinement.

(the adjacency lists are randomly permuted). On the other hand, HCM needs to find the edge with the maximum edge density, and LEM produces coarser graphs that have vertices with higher degree than the other three schemes; hence, LEM requires more time to both find a matching and also to create the next level coarser graph. The coarsening time required by HEM is only slightly higher (upto 10% more) than the time required by RM.

Comparing the time spent during uncoarsening, we see that both HEM and HCM require the least amount of time, while LEM requires the most. In some cases, LEM requires as much as 7 times more time than either HEM or HCM. This can be explained by results shown in Table 3. This table shows the edge-cut of 32-way partition when no refinement is performed (*i.e.*, the final edge-cut is exactly the same as that found in the initial partition of the coarsest graph). Table 3 shows that the edge-cut of LEM on the coarser graphs is significantly higher than that for either HEM or HCM. Because of this, all three components of UTime increase for LEM relative to those of the other schemes. The ITime is higher because the coarser graph has more edges, RTime increases because a large number of vertices need to be swapped to reduce the edge-cut, and PTime increases because more vertices are along the boundary; which requires more computation [22]. The time spent during uncoarsening for RM is also higher than the time required by the HEM scheme by upto 50% for some matrices for somewhat similar reasons.

From the discussion in the previous paragraphs we see that UTime is much smaller than CTime for HEM and HCM, while UTime is comparable to CTime for RM and LEM. Furthermore, for HEM and HCM, as the problem size increases UTime becomes an even smaller fraction of CTime. As discussed in introduction, this is of particular importance when the parallel formulation of the multilevel algorithm is considered.

As the experiments show, HEM is a good matching scheme that results in good initial partitions, and requires little refinement. Even though it requires slightly more time than RM, it produces consistently smaller edge-cut. We selected the HEM as our matching scheme of choice because of its consistent good behavior.

Initial Partition Algorithms As described in Section 3.2, a number of algorithms can be used to partition the coarse graph. We have implemented the following algorithms: (a) spectral bisection (SBP), (b) graph growing (GGP), and (c) greedy graph growing (GGGP). Due to space limitations we do not report the results here but they can be found in [22]. In summary, the results in [22] show that GGGP consistently finds smaller edge-cuts than the other schemes at slightly better run time. Furthermore, there is no advantage in choosing spectral bisection for partitioning the coarse graph.

Refinement Policies As described in Section 3.3, there are different ways that a partition can be refined during the uncoarsening phase. We evaluated the performance of five refinement policies, both in terms of how good partitions they produce and also how much time they require. The refinement policies that we evaluate are (a) Greedy refinement (GR), (b) Kernighan-Lin refinement (KLR), (c) boundary Greedy refinement (BGR), (d) boundary Kernighan-Lin refinement (BKLR), and (e) the combination of BKLR and BGR (BKLGR).

The result of these refinement policies for partitioning graphs corresponding to some of the matrices in Table 1 in 32 parts is shown in Table 4. These partitions were produced by using the heavy-edge matching (HEM) during coarsening and the GGGP algorithm for initially partitioning the coarser graph.

A number of interesting conclusions can be drawn out of Table 4. First, for each of the matrices and refinement policies, the size of the edge-cut does not vary significantly for different refinement policies. For each matrix the edge cut of every refinement policy is within 15% of the best refinement policy for that particular matrix. On the other hand, the time required by some refinement policies does vary significantly. Some policies require up to 20 times more time than others. KLR requires the most time while BGR requires the least.

	GR		KLR		BGR		BKLR		BKLGR	
	32EC	RTime	32EC	RTime	32EC	RTime	32EC	RTime	32EC	RTime
BCSSTK31	45267	1.05	46852	2.33	46281	0.76	45047	1.91	45991	1.27
BCSSTK32	66336	1.39	71091	2.89	72048	0.96	68342	2.27	69361	1.47
BRACK2	22451	2.04	20720	4.92	20786	1.16	19785	3.21	21152	2.36
CANT	323.4K	3.30	320.5K	6.82	325.0K	2.43	319.5K	5.49	323.0K	3.16
COPTER2	31338	2.24	31215	5.42	32064	1.12	30517	3.11	30938	1.83
CYLINDER93	201.0K	1.95	200.0K	4.32	199.0K	1.40	199.0K	2.98	198.0K	1.88
4ELT	1834	0.44	1833	0.96	2028	0.29	1894	0.66	1894	0.66
INPRO1	75676	1.28	75911	3.41	76315	0.96	74314	2.17	75203	1.48
ROTOR	38214	4.98	38312	13.09	36834	1.93	36498	5.71	36512	3.20
SHELL93	91723	9.27	79523	52.40	84123	2.72	80842	10.05	81756	6.01
TROLL	317.5K	9.55	309.7K	27.4	314.2K	4.14	300.8K	13.12	307.0K	5.84
WAVE	74486	8.72	72343	19.36	71941	3.08	71648	10.90	72034	4.50

Table 4: Performance of five different refinement policies. All matrices have been partitioned in 32 parts. *32EC* is the number of edges crossing partitions, and *RTime* is the time required to perform the refinement.

Comparing GR with KLR, we see that KLR performs better than GR for 8 out of the 12 matrices. For these 8 matrices, the improvement is less than 5% on the average; however, the time required by KLR is significantly higher than that of GR. Usually, KLR requires two to three times more time than GR.

Comparing the GR and KLR refinement schemes against their boundary variants, we see that the time required by the boundary policies is significantly less than that required by their non-boundary counterparts. The time of BGR ranges from 29% to 75% of the time of GR, while the time of BKLR ranges from 19% to 80% of the time of KLR. This seems quite reasonable, given that BGR and BKLR are simpler versions of GR and KLR, respectively. But surprisingly, BGR and BKLR lead to better edge-cut (than GR and KLR, respectively) in many cases. BGR does better than GR in 6 out of the 12 matrices, and BKLR does better than KLR in 10 out the 12 matrices. Thus, the quality of the boundary refinement policies is similar if not better than their non-boundary counterparts.

Even though BKLR appears to be just a simplified version of KLR, in fact they are two distinct schemes. In each scheme, a set of vertices from the two parts of the partition is swapped in each iteration. In BKLR, the set of vertices to be swapped from either part is restricted to be only along the boundary, whereas in the KLR it can potentially be any subset. BKLR performs better in conjunction with the HEM coarsening scheme, because for HEM the first partition of the coarsest graph is quite good (consistently better than the partition that can be obtained for other coarsening schemes such as RM and LEM), and it does not change significantly with each uncoarsening phase. Note that by restricting each iteration of KL on the boundary vertices, more iterations are needed for the algorithm to converge to a local minima. However, these iterations take very little time. Thus, BKLR provides the very precise refinement that is needed by HEM.

For the other matching schemes, and for LEM in particular, the partition of the coarse graph is far from being close to a local minima when it is projected in the next level finer graph, and there is room for significant improvement not just along the boundary. This is the reason why LEM requires the largest refinement time among all the matching schemes, irrespective of the refinement policy. Since boundary refinement schemes consider only boundary vertices, they may miss sequences of vertex swaps that involve non boundary vertices and lead to a better partition. To compare the performance of the boundary refinement policies against their non-boundary counterparts, for both RM and LEM, we performed another set of experiments similar to those shown in Table 4. For the RM coarsening scheme, BGR outperformed GR in 5 matrices, and BKLR outperformed KLR only in 5 matrices. For the LEM coarsening scheme, BGR outperformed GR only in 4 matrices and BKLR outperformed KLR only in 3 matrices.

Comparing BGR with BKLR we see that the edge-cut is better for BKLR for 11 matrices, and they perform similarly for the remaining matrix. Note that the improvement performed by BKLR over BGR is relatively small (less than 4% on the average). However, the time required by BKLR is always higher than that of BGR (in some cases upto four times higher). Again we see here that marginal improvements in the partition quality come at a significant increase in the refinement time. Comparing BKLGR against BKLR we see that its edge-cut is on the average within 2% of that of BKLR, while its runtime is significantly smaller than that of BKLR and somewhat higher than that of BGR.

In summary, when it comes to refinement policies, a relatively small decrease in the edge-cut usually comes at a significant increase in the time required to perform the refinement. Both the BGR and the BKLGR refinement policies require little amount of time and produce edge-cuts that are fairly good when coupled with the heavy-edge matching scheme. We believe that the BKLGR refinement policy strikes a good balance between small edge-cut and fast execution.

4.2 Comparison with Other Partitioning Schemes

The multilevel spectral bisection (MSB) [2] has been shown to be an effective method for partitioning unstructured problems in a variety of applications. The MSB algorithm coarsens the graph down to a few hundred vertices using random matching. It partitions the coarse graph using spectral bisection and obtains the Fiedler vector of the coarser graph. During uncoarsening, it obtains an approximate Fiedler vector of the next level fine graph by interpolating the Fiedler vector of the coarser graph, and computes a more accurate Fiedler vector using the SYMMLQ. The MSB algorithm computes the Fiedler vector of the graph using this multilevel approach. This method is much faster than computing the Fiedler vector of the original graph directly. Note that MSB is a significantly different scheme than the multilevel scheme that uses spectral bisection to partition the graph at the coarsest level. We used the MSB algorithm in the Chaco [19] graph partitioning package to produce partitions for some of the matrices in Table 1 and compared them against the partitions produced by our multilevel algorithm that uses HEM during coarsening phase, GGGP during partitioning phase, and BKLGR during the uncoarsening phase.

Figure 1 shows the relative performance of our multilevel algorithm compared to MSB. For each matrix we plot the ratio of the edge-cut of our multilevel algorithm to the edge-cut of the MSB algorithm. Ratios that are less than one indicate that our multilevel algorithm produces better partitions than MSB. From this figure we can see that for almost all the problems, our algorithm produces partitions that have smaller edge-cuts than those produced by MSB. In some cases, the improvement is as high as 60%. For the cases where MSB does better, the difference is very small (less than 1%). However the time required by our multilevel algorithm is significantly smaller than that required by MSB. Figure 4 shows the time required by the MSB algorithm relative to that required by our multilevel algorithm. Our algorithm is usually 10 times faster for small problems, and 15 to 35 times faster for larger problems. The relative difference in edge-cut between MSB and our multilevel algorithm decreases as the number of partitions increases. This is a general trend, since as the number of partitions increase both schemes cut more edges, to the limiting case in which $|V|$ partitions are used in which case all $|E|$ edges are cut.

One way of improving the quality of MSB algorithm is to use the Kernighan-Lin algorithm to refine the partitions (MSB-KL). Figure 2 shows the relative performance of our multilevel algorithm compared against the MSB-KL algorithm. Comparing Figures 1 and 2 we see that the Kernighan-Lin algorithm does improve the quality of the MSB algorithm. Nevertheless, our multilevel algorithm still produces better partitions than MSB-KL for many problems. However, KL refinement further increases the run time of the overall scheme as shown in Figure 4; thus, increases the gap in the run time of MSB-KL and our multilevel algorithm.

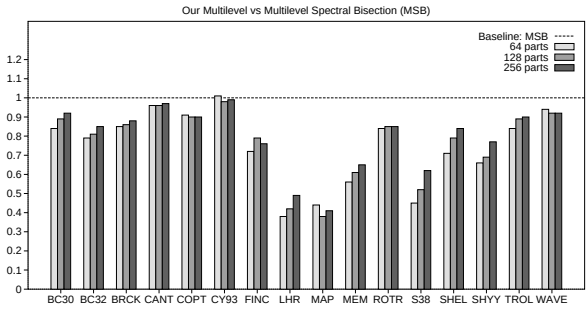

Figure 1: Quality of our multilevel algorithm compared to the multilevel spectral bisection algorithm. For each matrix, the ratio of the cut-size of our multilevel algorithm to that of the MSB algorithm is plotted for 64-, 128- and 256-way partitions. Bars under the baseline indicate that the multilevel algorithm performs better.

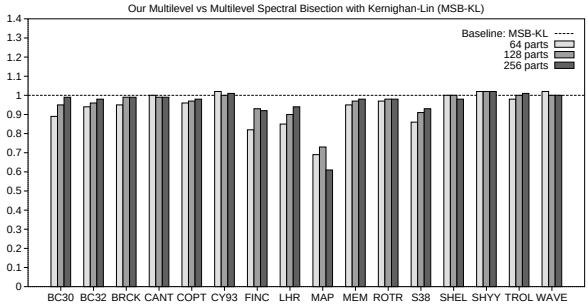

Figure 2: Quality of our multilevel algorithm compared to the multilevel spectral bisection algorithm with Kernighan-Lin refinement. For each matrix, the ratio of the cut-size of our multilevel algorithm to that of the MSB-KL algorithm is plotted for 64-, 128- and 256-way partitions. Bars under the baseline indicate that our multilevel algorithm performs better.

The graph partitioning package Chaco implements its own multilevel graph partitioning algorithm that is modeled after the algorithm by Hendrickson and Leland [20, 19]. This algorithm, which we refer to as Chaco-ML, uses random matching during coarsening, spectral bisection for partitioning the coarse graph, and Kernighan-Lin refinement every other coarsening level during the uncoarsening phase. Figure 3 shows the relative performance of our multilevel algorithms compared to Chaco-ML. From this figure we can see that our multilevel algorithm usually produces partitions with smaller edge-cut than that of Chaco-ML. For some problems, the improvement of our algorithm is between 10% to 50%. Again for the cases where Chaco-ML does better, it is only marginally better (less than 2%). Our algorithm is usually two to six times faster than Chaco-ML (Figure 4). Most of the savings come from the choice of refinement policy (we use BKLGR) which is usually four to six times faster than the Kernighan-Lin refinement

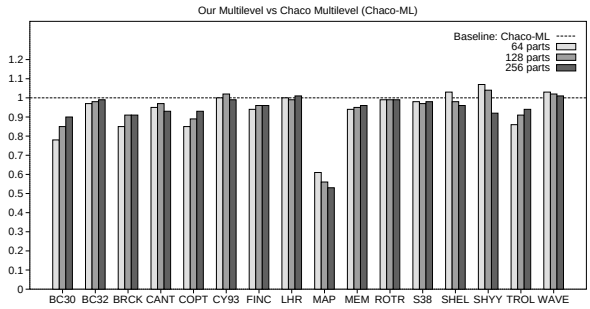

Figure 3: Quality of our multilevel algorithm compared to the multilevel Chaco-ML algorithm. For each matrix, the ratio of the cut-size of our multilevel algorithm to that of the Chaco-ML algorithm is plotted for 64-, 128- and 256-way partitions. Bars under the baseline indicate that our multilevel algorithm performs better.

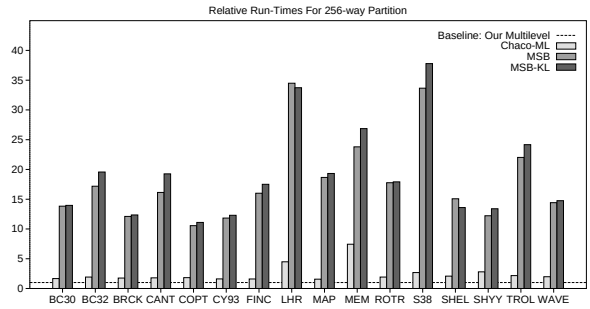

Figure 4: The time required to find a 256-way partition for Chaco-ML, MSB, and MSB-KL relative to the time required by our multilevel algorithm.

implemented by Chaco-ML. Note that we are able to use BKLGR without much quality penalty only because we use the HEM coarsening scheme. In addition, the GGGP used in our method for partitioning the coarser graph requires much less time than the spectral bisection which is used in Chaco-ML.

4.3 Sparse Matrix Ordering

The multilevel graph partitioning algorithm can be used to find a fill reducing ordering for a symmetric sparse matrix via recursive nested dissection. Let S be the vertex separator and let A and B be the two parts of the vertex set of G that are separated by S. In the nested dissection ordering, A is ordered first, B second, while the vertices in S are numbered last. Both A and B are ordered by recursively applying nested dissection ordering. In our multilevel nested dissection algorithm (MLND) a vertex separator is computed from an edge separator by finding the minimum vertex cover [31]. The minimum vertex cover has been found to produce very small vertex separators.

Overall quality of a fill reducing ordering depends on whether or not the matrix is factored on a serial or parallel computer. On a serial computer, a good ordering is the one that requires the smaller number of operations during factorization. The number of operations required is usually related to the number of nonzeros in the Cholesky factors. The fewer nonzeros usually lead to fewer operations. However, since the number of operations is the square of the number of nonzeros, similar fills may have different operation counts. For this reason, all comparisons in this section are only in terms of the number of operations. On a parallel computer, a fill reducing ordering, besides minimizing the operation count, should also increase the degree of concurrency that can be exploited during factorization. In general, nested dissection based orderings exhibit more concurrency during factorization than minimum degree orderings [10, 27] that have been found to be very effective for serial factorization.

The minimum degree [10] ordering heuristic is the most widely used fill reducing algorithm that is used to order sparse matrices for factorization on serial computers. The minimum degree algorithm has been found to produce very good orderings. The multiple minimum degree algorithm [27] is the most widely used variant of minimum degree due to its very fast runtime.

The quality of the orderings produced by our multilevel nested dissection algorithm compared to that of MMD is shown in Figure 5. For our multilevel algorithm, we used the HEM scheme during coarsening, the GGGP scheme for partitioning the coarse graph and the BKLGR refinement policy during the uncoarsening phase. Looking at this figure we see that our algorithm produces better orderings for 11 out of the 18 test problems. For the other seven problems MMD does better. However, for many of these 7 matrices, MMD does only slightly better than MLND. The only exception is BCSPRW10 for which all nested dissection schemes perform poorly.

However, for the matrices arising in finite element domains, MLND does consistently better than MMD, and is some cases by a large factor (two to three times better for CANT, ROTR, SHEL, and WAVE). Also, from Figure 5 we see that MLND does consistently better as the size of the matrices increases and as the matrices become more unstructured. When all 18 test matrices are considered, MMD produces orderings that require a total of 702 billion operations, whereas the orderings produced by MLND require only 293 billion operations. Thus, the ensemble of 18 matrices can be factored roughly 2.4 times faster if ordered with MLND.

However, another, even more important, advantage of MLND over MMD, is that it produces orderings that exhibit significantly more concurrency than MMD. The elimination trees produced by MMD (a) exhibit little concurrency (long and slender), and (b) are unbalanced so that subtree-to-subcube mappings lead to significant load imbalances [26, 9, 14]. One the other hand, orderings based on nested dissection produce orderings that have both more

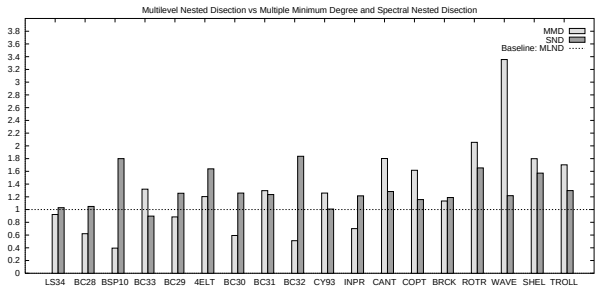

Figure 5: Quality of our multilevel nested dissection relative to the multiple minimum degree, and the spectral nested dissection algorithm. The matrices are displayed in increasing number of equations. **Bars above the baseline indicate that the MLND algorithm performs better.**

concurrency and better balance [24, 16]. Therefore, when the factorization is performed in parallel, the better utilization of the processors can cause the ratio of the run time of parallel factorization algorithms running ordered using MMD and that using MLND to be substantially higher than the ratio of their respective operation counts.

The MMD algorithm usually takes two to three times less time to order the matrices in Table 1 than the time required by MLND. However, efforts to parallelize the MMD algorithm have had no success [11]. In fact, the MMD algorithm appears to be inherently serial in nature. On the other hand, the MLND algorithm is amenable to parallelization. In [23] we present a parallel formulation of our MLND algorithm that achieves a speedup of 57 on 128-processor Cray T3D.

Spectral nested dissection (SND) [32] is a widely used ordering algorithm for ordering matrices for parallel factorization. As in the case of MLND, the minimum vertex cover algorithm was used to compute a vertex separator from the edge separator. The quality of the orderings produced by our multilevel nested dissection algorithm compared to that of the spectral nested dissection algorithm is also shown in Figure 5. From this figure we can see that MLND produces orderings that are better than SND for 17 out of the 18 test matrices. The total number of operations required to factor the matrices ordered using SND is 378 billion which is 30% more than the of MLND. Furthermore, as discussed in Section 4.2, the runtime of SND is substantially higher than that of MLND. Also, SND cannot be parallelized any better than MLND; therefore, it will always be slower than MLND.

References

[1] 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.

[2] 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.

[3] 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.

[4] Tony F. Chan, John R. Gilbert, and Shang-Hua Teng. Geometric spectral partitioning (draft). Technical Report In Preparation, 1994.

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

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

[10] A. George and J. W.-H. Liu. The evolution of the minimum degree ordering algorithm. *SIAM Review*, 31(1):1–19, March 1989.

[11] Madhurima Ghose and Edward Rothberg. A parallel implementtaion of the multiple minimum degree ordering heuristic. Technical report, Old Dominion University, Norfolk, VA, 1994.

[12] 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.

[13] John R. Gilbert, Gary L. Miller, and Shang-Hua Teng. Geometric mesh partitioning: Implementation and experiments. In *Proceedings of International Parallel Processing Symposium*, 1995.

[14] 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. Submitted for publication in *IEEE Transactions on Parallel and Distributed Computing*. Available on WWW at URL ftp://ftp.cs.umn.edu/users/kumar/sparse-cholesky.ps.

[15] 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.

[16] 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.

[17] M. T. Heath and P. Raghavan. A Cartesian nested dissection algorithm. Technical Report UIUCDCS-R-92-1772, Department of Computer Science, University of Illinois, Urbana, IL 61801, 1992. To appear in *SIAM Journal on Matrix Analysis and Applications*, 1994.

[18] Bruce Hendrickson and Rober Leland. An improved spectral graph partitioning algorithm for mapping parallel computations. Technical Report SAND92-1460, Sandia National Laboratories, 1992.

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

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

[21] 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.

[22] G. Karypis and V. Kumar. Multilevel graph partitioning schemes. Technical report, Department of Computer Science, University of Minnesota, 1995. Available on WWW at URL ftp://ftp.cs.umn.edu/users/kumar/mlevel_serial.ps.

[23] G. Karypis and V. Kumar. Parallel multilevel graph partitioning. Technical report, Department of Computer Science, University of Minnesota, 1995. Available on WWW at URL ftp://ftp.cs.umn.edu/users/kumar/mlevel_parallel.ps.

[24] George Karypis, Anshul Gupta, and Vipin Kumar. A parallel formulation of interior point algorithms. In *Supercomputing 94*, 1994. Available on WWW at URL ftp://ftp.cs.umn.edu/users/kumar/interior-point.ps.

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

[26] 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.

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

[28] 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.

[29] 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.

[30] 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.

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

[32] Alex Pothen, H. D. Simon, and Lie Wang. Spectral nested dissection. Technical Report 92-01, Computer Science Department, Pennsylvania State University, University Park, PA, 1992.

[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.