To appear in SIAM Journal on Scientific Computing

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

A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs *

George Karypis and Vipin Kumar

University of Minnesota, Department of Computer Science
AHPCRC
Minneapolis, MN 55455

{karypis, kumar}@cs.umn.edu

Last updated on January 23, 1997 at 10:49am

Abstract

Recently, a number of researchers have investigated 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 [4, 26]. From the early work it was clear that multilevel techniques held great promise; however, it was not known if they can be made to consistently produce high quality partitions for graphs arising in a wide range of application domains. 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 much faster variation of the Kernighan-Lin algorithm for refining during uncoarsening. 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 produces partitions that are consistently better than those produced by spectral partitioning schemes in substantially smaller time. Also, when our scheme is used to compute fill reducing orderings for sparse matrices, it produces orderings that have substantially smaller fill than the widely used multiple minimum degree algorithm.

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

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

1 Introduction

Graph partitioning is an important problem that has extensive applications in many areas, including scientific computing, VLSI design, and task scheduling. 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. A good partition of the graph corresponding to matrix A can significantly reduce the amount of communication in parallel sparse matrix-vector multiplication [32]. 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 [32, 12]. 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 good partitions for a wide class of problems, and they are used quite extensively [47, 46, 24]. However, these methods are very expensive since they require the computation of the eigenvector corresponding to the second smallest eigenvalue (Fiedler vector). Execution time of the spectral methods can be reduced if computation of the Fiedler vector is done by using a multilevel algorithm [2]. This multilevel spectral bisection algorithm (MSB) usually manages to speed up 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 [18].

Another class of graph partitioning techniques uses the geometric information of the graph to find a good partition. Geometric partitioning algorithms [23, 48, 37, 36, 38] tend to be fast but often yield partitions that are worse than those obtained by spectral methods. Among the most prominent of these schemes is the algorithm described in [37, 36]. This algorithm produces partitions that are provably within the bounds that exist for some special classes of graphs (that includes graphs arising in finite element applications). However, due to the randomized nature of these algorithms, multiple trials are often required (5 to 50) to obtain solutions that are comparable in quality to spectral methods. Multiple trials do increase the time [16], but the overall runtime is still substantially lower than the time required by the spectral methods. Geometric graph partitioning algorithms are applicable only if coordinates are available for the vertices of the graph. In many problem areas (*e.g.*, linear programming, VLSI), there is no geometry associated with the graph. Recently, an algorithm has been proposed to compute coordinates for graph vertices [6] by using spectral methods. But these methods are much more expensive and dominate the overall time taken by the graph partitioning algorithm.

Another class of graph partitioning algorithms reduces 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 [4, 7, 19, 20, 26, 10, 43]. Some researchers investigated multilevel schemes primarily to decrease the partitioning time, at the cost of somewhat worse partition quality [43]. Recently, a number of multilevel algorithms have been proposed [4, 26, 7, 20, 10] that further refine the partition during the uncoarsening phase. These schemes tend to give good partitions at a reasonable cost. Bui and Jones [4] use random maximal matching to successively coarsen the graph down to a few hundred vertices; they partition the smallest graph and then uncoarsen the graph level by level, applying Kernighan-Lin to refine the partition. Hendrickson and Leland [26] enhance this approach by using edge and vertex weights to capture the collapsing of the vertex and edges. In particular, this latter work showed that multilevel schemes can provide better partitions than spectral methods at lower cost for a variety of finite element problems.

In this paper we build on the work of Hendrickson and Leland. 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 variation of the Kernighan-Lin algorithm for refining the partition during the uncoarsening phase that is much faster than the Kernighan-Lin refinement used in [26].

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[1]. Compared with the multilevel scheme of [26], our scheme is about two to seven times faster, and is consistently better in terms of cut size. Much of the improvement in run time comes from our faster refinement heuristic, and the improvement in quality is due to the heavy-edge heuristic used during coarsening.

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 [35], which is the most commonly used method for computing fill reducing orderings of a sparse matrix.

Even though multilevel algorithms are quite fast compared to spectral methods, they can still be the bottleneck if the sparse system of equations is being solved in parallel [32, 18]. The coarsening phase of these methods is relatively easy to parallelize [29], but the Kernighan-Lin heuristic used in the refinement phase is very difficult to parallelize [15]. Since both the coarsening phase and the refinement phase with the Kernighan-Lin heuristic take roughly the same amount of time, the overall run-time of the multilevel scheme of [26] cannot be reduced significantly. Our new faster methods for refinement reduce this bottleneck substantially. In fact our parallel implementation [29] 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.

The remainder of the paper is organized as follows. Section 2 defines the graph partitioning problem and describes the basic ideas of multilevel graph partitioning. Sections 3, 4, and 5 describe different algorithms for the coarsening, initial partitioning, and the uncoarsening phase, respectively. Section 6 presents an experimental evaluation of the various parameters of multilevel graph partitioning algorithms and compares their performance with that of multilevel spectral bisection algorithm. Section 7 compares the quality of the orderings produced by multilevel nested dissection to those produced by multiple minimum degree and spectral nested dissection. Section 9 provides a summary of the various results. A short version of this paper appears in [28].

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. The k-way graph partitioning problem can be naturally extended to graphs that have weights associated with the vertices and the edges of the graph. In this case, the goal is to partition the vertices into k disjoint subsets such that the sum of the vertex-weights in each subset is the same, and the sum of the edge-weights 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. Depending on the amount of the computation performed by each task, the vertices are assigned a proportional weight. Similarly, the edges are assigned weights that reflect the amount of data that needs to be exchanged. A k-way partitioning of this computation graph can be used to assign tasks to k processors. Since the partitioning assigns to each processor tasks whose total weight is the same, the work is balanced among k processors. Furthermore, since the algorithm minimizes the edge-cut (subject to the balanced load requirements), the communication overhead is also minimized.

One such example is the sparse-matrix vector multiplication $y = Ax$. Matrix $A_{n\times n}$ and vector x is usually partitioned along rows, with each of the p processors receiving n/p rows of A, and the corresponding n/p elements of x [32]. For matrix A an n-vertex graph G_A, can be constructed such that each row of the matrix corresponds to a vertex,

[1] We used the MSB algorithm in the Chaco [25] graph partitioning package to obtain the timings for MSB.

and if row i has a nonzero entry in column j $(i \neq j)$, then there is an edge between vertex i and vertex j. As discussed in [32], any edges connecting vertices from two different partitions lead to communication for retrieving the value of vector x that is not local but is needed to perform the dot-product. Thus, in order to minimize the communication overhead, we need to obtain a p-way partition of G_A, and then distribute the rows of A according to this partition.

Another important application of recursive bisection is to find a fill reducing ordering for sparse matrix factorization [12, 32, 22]. These 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 partitions are obtained. One way of obtaining a vertex separator is to first obtain a bisection of the graph and then compute 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 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 can be solved by 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 [12, 22].

2.1 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). At each step of the graph uncoarsening, the partition is further refined. Since the finer graph has more degrees of freedom, such refinements usually decrease the edge-cut. This process, is graphically illustrated in Figure 1.

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 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. Some possibilities are shown in Figure 2.

In most coarsening schemes, a set of vertices of G_i is combined 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. Updating the weights of the coarser graph is illustrated in Figure 2.

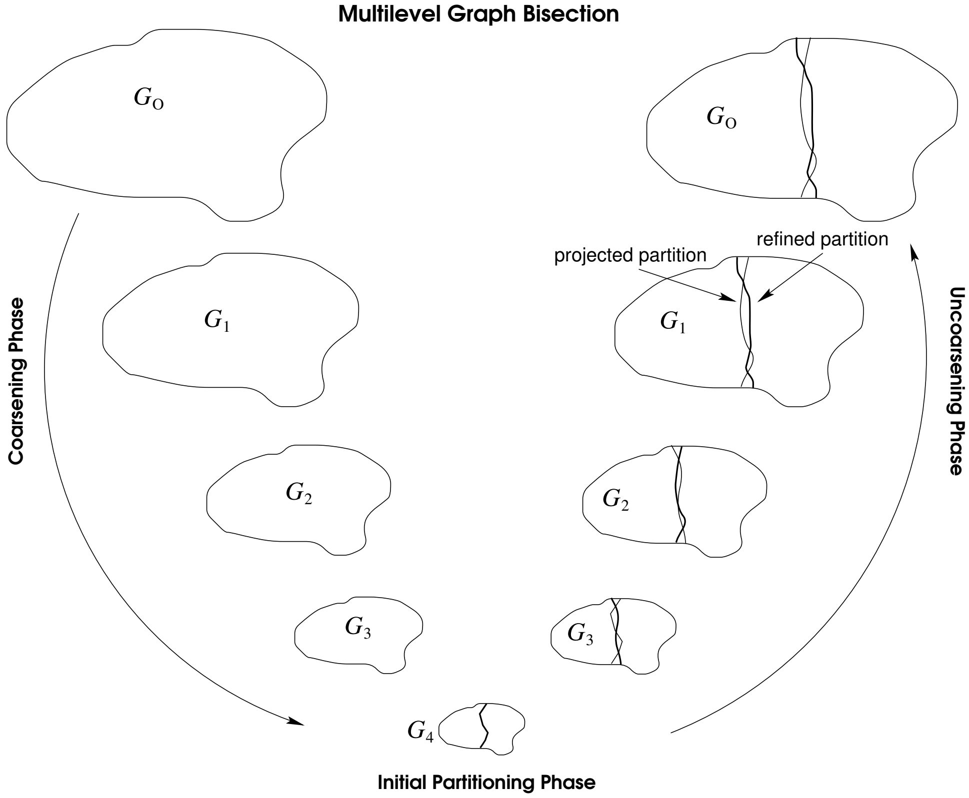

Figure 1: The various phases of the multilevel graph bisection. During the coarsening phase, the size of the graph is successively decreased; during the initial partitioning phase, a bisection of the smaller graph is computed; and during the uncoarsening phase, the bisection is successively refined as 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.

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 [4, 26], while the second approach is based on creating multinodes that are made of groups of vertices that are highly connected [7, 19, 20, 10]. 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 contain a large number of edges. For this reason, ***maximal matchings*** are used to obtain the successively coarse graphs. A matching is maximal if any edge in the graph that is not in the matching has at least one of its endpoints matched. Note that depending on how matchings are computed, the number of edges belonging to the maximal matching may be different. The maximal matching that has the maximum number of edges is called *maximum matching*. However, because the complexity of computing a maximum matching [41] is in general higher than that of computing a maximal matching, the latter are preferred.

Coarsening a graph using matchings preserves many properties of the original graph. If G_0 is (maximal) planar, the G_i is also (maximal) planar [34]. This property is used to show that the multilevel algorithm produces partitions that

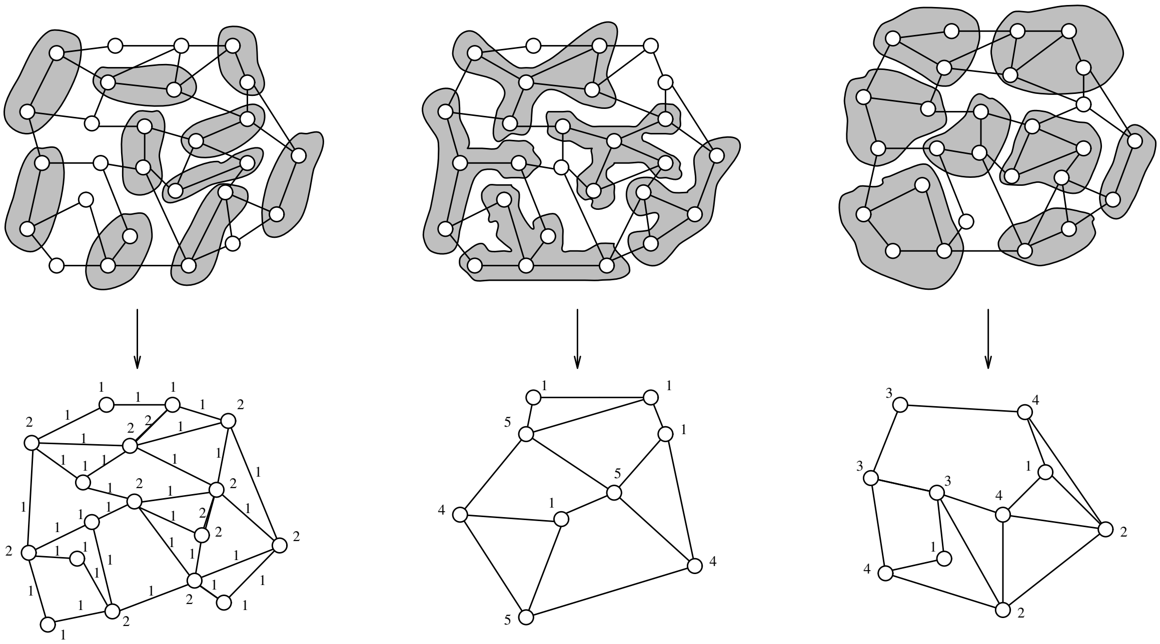

Figure 2: Different ways to coarsen a graph.

are provably good for planar graphs [27].

Since maximal matchings are used to coarsen the graph, the number of vertices in G_{i+1} cannot be less than half the number of vertices in G_i; thus, it will require at least $O(\log(n/n'))$ coarsening phases to coarsen G_0 down to a graph with n' vertices. However, depending on the connectivity of G_i, the size of the maximal matching may be much smaller than $|V_i|/2$. In this case, the ratio of the number of vertices from G_i to G_{i+1} may be much smaller than 2. If the ratio becomes lower than a threshold, then it is better to stop the coarsening phase. However, this type of pathological condition usually arises after many coarsening levels, in which case G_i is already fairly small; thus, aborting the coarsening does not affect the overall performance of the algorithm.

In the remaining sections we describe four ways that we used to select maximal matchings for coarsening.

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 [4, 26]. 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. The complexity of the above algorithm is $O(|E|)$.

Heavy Edge Matching (HEM) Random matching is a simple and efficient method to compute a maximal matching and minimizes the number of coarsening levels in a greedy fashion. However, our overall goal is to find a partition 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

$$W(E_{i+1}) = W(E_i) - W(M_i). \tag{1}$$

Thus, the total edge-weight of the coarser graph is reduced by the weight of the matching. Hence, by selecting a maximal matching M_i whose edges have a large weight, we can decrease the edge-weight of the coarser graph by a greater amount. As the analysis in [27] shows, since the coarser graph has smaller edge-weight, it also has a smaller edge-cut. Finding a maximal matching that contains edges with large weight is the idea behind the ***heavy-edge matching***. A heavy-edge matching is computed using a randomized algorithm similar to that for computing a random matching described earlier. 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 (over all possible matchings), but our experiments have shown that it works very well. The complexity of computing a heavy-edge matching is $O(|E|)$, which is asymptotically similar to that for computing the random matching.

Light Edge Matching (LEM) Instead of minimizing the total edge weight of the coarser graph, one might try to maximize it. From Equation 1, 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 significantly higher than that of G_i. This is important for certain partitioning heuristics such a Kernighan-Lin [4], because they produce good partitions in small amount of time for graphs with high average degree.

To compute a matching with minimal weight we only need to slightly modify the algorithm for computing the maximal-weight matching in Section 3. Instead of selecting an edge (u, v) in the matching such that the weight of (u, v) is the largest, we select an edge (u, v) such that its weight is the smallest. The complexity of computing the minimum-weight matching is also $O(|E|)$.

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 [7, 19, 20, 10].

As in the previous schemes for computing the matching, we compute the heavy clique matching using a randomized algorithm. For the computation of edge density, so far we have only dealt with the case in which the vertices and edges of the original graph $G_0 = (V_0, E_0)$ have unit weight. Consider a coarse graph $G_i = (V_i, E_i)$. For every vertex $u \in V_i$, define $vw(u)$ to be the weight of the vertex. Recall that this is equal to the sum of the weight of the vertices in the original graph that have been collapsed into u. Define $ce(u)$ to be the sum of the weight of the collapsed edges of u. These edges are those collapsed to form the multinode u. Finally, for every edge $e \in E_i$ define $ew(e)$ be the weight of the edge. Again, this is the sum of the weight of the edges that through the coarsening have been collapsed into e. Given these definitions, the edge density between vertices u and v is given by:

$$\frac{2(ce(u) + ce(v) + ew(u, v))}{(vw(u) + vw(v))(vw(u) + vw(v) - 1)}. \tag{2}$$

The randomized algorithm works as follows. The vertices are visited in a random order. An unmatched vertex u, is matched with its unmatched adjacent vertex v such that the edge density of the multinode created by combining u and v is the largest among all possible multinodes involving u and other unmatched adjacent vertices of u. 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.

4 Partitioning Phase

The second phase of a multilevel algorithm computes a high-quality bisection (*i.e.*, small edge-cut) 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. Since during coarsening, the weights of the vertices and edges of the coarser graph were set to reflect the weights of the vertices and edges of the finer graph, G_m contains sufficient information to intelligently enforce the balanced partition and the small edge-cut requirements.

A partition of G_m can be obtained using various algorithms such as (a) spectral bisection [47, 46, 2, 24], (b) geometric bisection [37, 36] (if coordinates are available [2]), and (c) combinatorial methods [31, 3, 11, 12, 17, 5, 33, 21]. Since the size of the coarser graph G_m is small (*i.e.*, $|V_m| < 100$), this step takes a small amount of time.

We implemented four different algorithms for partitioning the coarse graph. The first algorithm uses the spectral bisection. The other three algorithms are combinatorial in nature, and try to produce bisections with small edge-cut using various heuristics. These algorithms are described in the next sections. We choose not to use geometric bisection algorithms, since the coordinate information was not available for most of the test graphs.

4.1 Spectral Bisection (SB)

In the spectral bisection algorithm, the spectral information is used to partition the graph [47, 2, 26]. This algorithm computes the eigenvector y corresponding to the second largest eigenvalue of the Laplacian matrix $Q = D - A$, where

$$a_{i,j} = \begin{cases} ew(v_i, v_j) & \text{if } (v_i, v_j) \in E_m \\ 0 & \text{otherwise} \end{cases}$$

This eigenvector is called the Fiedler vector. The matrix D is diagonal such that $d_{i,i} = \sum ew(v_i, v_j)$ for $(v_i, v_j) \in E_m$. Given y, the vertex set V_m is partitioned into two parts as follows. Let r be the i^{th} element of the y vector. Let $P[j] = 1$ for all vertices such that $y_j \leq r$, and let $P[j] = 2$ for all the other vertices. Since we are interested in bisections of equal size, the value of r is chosen as the weighted median of the values of y_i.

The eigenvector y is computed using the Lanczos algorithm [42]. This algorithm is iterative and the number of iterations required depends on the desired accuracy. In our experiments, we set the accuracy to 10^{-2} and the maximum number of iterations to 100.

4.2 Kernighan-Lin Algorithm (KL)

The Kernighan-Lin algorithm [31] is iterative in nature. It starts with an initial bipartition of the graph. In each iteration it searches for a subset of vertices, from each part of the graph such that swapping them leads to a partition with smaller edge-cut. If such subsets exist, then the swap is performed 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, since the partition is at a local minimum and no further improvement can be made by the KL algorithm. Each iteration of the KL algorithm described in [31] takes $O(|E| \log |E|)$ time. Several improvements to the original KL algorithm have been developed. One such algorithm is by Fiduccia and Mattheyses [9] that reduces the complexity to $O(|E|)$, by using appropriate data structures.

The Kernighan-Lin algorithm finds locally optimal partitions when it starts with a good initial partition and when the average degree of the graph is large [4]. If no good initial partition is known, the KL algorithm is repeated with different randomly selected initial partitions, and the one that yields the smallest edge-cut is selected. Requiring multiple runs can be expensive, especially if the graph is large. However, since we are only partitioning the much smaller coarse graph, performing multiple runs requires very little time. Our experience has shown that the KL algorithm requires only five to ten different runs to find a good partition.

[2] Coordinates for the vertices of the successive coarser graphs can be constructed by taking the midpoint of the coordinates of the combined vertices.

Our implementation of the Kernighan-Lin algorithm is based on the algorithm described by Fiduccia and Mattheyses[3] [9], with certain modifications that significantly reduce the run time. Suppose P is the initial partition of the vertices of $G = (V, E)$. The ***gain*** g_v, of a vertex v is defined as the reduction on the edge-cut if vertex v moves from one partition to the other. This gain is given by:

$$g_v = \sum_{(v,u)\in E \wedge P[v]\neq P[u]} w(v, u) - \sum_{(v,u)\in E \wedge P[v]=P[u]} w(v, u), \tag{3}$$

where $w(v, u)$ is weight of edge (v, u). If g_v is positive, then by moving v to the other partition the edge-cut decreases by g_v; whereas if g_v is negative, the edge-cut increases by the same amount. If a vertex v is moved from one partition to the other, then the gains of the vertices adjacent to v may change. Thus, after moving a vertex, we need to update the gains of its adjacent vertices.

Given this definition of gain, the KL algorithm then proceeds by repeatedly selecting from the larger part a vertex v with the largest gain 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 original KL algorithm [9], continues moving vertices between the partitions, until all the vertices have been moved. However, in our implementation, the KL algorithm terminates when the edge-cut does not decrease after x vertex moves. Since the last x vertex moves did not decrease the edge-cut (they may have actually increased it), they are undone. We found that setting $x = 50$ works quite well for our test cases. Note that terminating the KL iteration in this fashion significantly reduces the run time of the KL iteration.

The efficient implementation of the above algorithm depends on the method that is used to compute the gains of the graph and the type of data structure used to store these gains. The implementation of the KL algorithm is described in Appendix A.3.

4.3 Graph Growing Algorithm (GGP)

Another simple way of bisecting the graph is to start from a vertex and grow a region around it in a breath-first fashion, until half of the vertices have been included (or half of the total vertex weight) [12, 17, 39]. The quality of the graph growing algorithm is sensitive to the choice of a vertex from which to start growing the graph, and different starting vertices yield different edge-cuts. To partially solve this problem, we randomly select 10 vertices and we grow 10 different regions. The trial with the smaller edge-cut is selected as the partition. This partition is then further refined by using it as the input to the KL algorithm. Again, because G_m is very small, this step takes a small percentage of the total time.

4.4 Greedy Graph Growing Algorithm (GGGP)

The graph growing algorithm described in the previous section grows a partition in a strict breadth-first fashion. However, as in the KL algorithm, for each vertex v we can define the gain in the edge-cut obtained by inserting v into the growing region. Thus, we can order the vertices of the graph's frontier in non-decreasing order according to their gain. Thus, the vertex with the largest decrease (or smallest increase) in the edge-cut is inserted first. When a vertex is inserted into the growing partition, then the gains of its adjacent vertices already in the frontier are updated, and those not in the frontier are inserted. Note that the data structures required to implement this scheme are essentially those required by the KL algorithm. The only difference is that instead of precomputing all the gains for all the vertices, we do so as these vertices are touched by the frontier.

This greedy algorithm is also sensitive to the choice of the initial vertex, but less so than GGP. In our implementation we randomly select four vertices as the starting point of the algorithm, and we select the partition with the smaller edge-cut. In our experiments, we found that GGGP takes somewhat less time than GGP for partitioning the coarse graph

[3]The algorithm described by Fiduccia and Mattheyses (FM) [9], is slightly different than that originally developed by Kernighan and Lin (KL) [31]. The difference is that in each step, the FM algorithm moves a single vertex from one part to the other whereas the KL algorithm selects a pair of vertices, one from each part, and moves them.

(because it requires fewer runs), and the initial cut found by the scheme is better than that found by GGP.

5 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 set of vertices V_i^v collapsed to $v \in G_{i+1}$ to the partition $P_{i+1}[v]$ (*i.e.*, $P_i[u] = P_{i+1}[v], \ \ \forall u \in V_i^v$).

Even though P_{i+1} is a local minimum partition of G_{i+1}, the projected partition P_i may not be at a local minimum 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 a 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 described in Section 4.2. Recall that the KL algorithm starts with an initial partition and in each iteration it finds subsets A' and B' with the above properties.

In the next sections we describe two different refinement algorithms that are based on similar ideas but differ in the time they require to do the refinement. Details about the efficient implementation of these schemes can be found in Appendix A.3.

5.1 Kernighan-Lin Refinement

The idea of Kernighan-Lin refinement is to use the projected partition of G_{i+1} onto G_i as the initial partition for the Kernighan-Lin algorithm described in Section 4.2. The reason is that this projected partition is already a good partition; thus, KL will converge within a few iterations to a better partition. For our test cases, KL usually converges within three to five iterations.

Since we are starting with a good partition, only a small number of vertex swaps will decrease the edge-cut and any further swaps will increase the size of the cut (vertices with negative gains). Recall from Section 4.2, that in our implementation, a single iteration of the KL algorithm stops as soon as 50 swaps are performed that do not decrease the edge-cut. This feature reduces the runtime when KL is applied as a refinement algorithm, since only a small number of vertices lead to edge-cut reductions. Our experimental results show that for our test cases this is usually achieved 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.

Since we terminate each pass of the KL algorithm when no further improvement can be made in the edge-cut, the complexity of the KL refinement 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 KL(1) refinement algorithm, we take advantage of that by running only a single iteration of the KL algorithm. This usually reduces the total time taken by refinement by a factor of two to four (Section 6.3).

5.2 Boundary Kernighan-Lin Refinement

In both the KL and KL(1) refinement 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 KL or KL(1) algorithms 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 Kernighan-Lin refinement algorithm, we initially insert into the data structures the gains for only the boundary vertices. As in the KL refinement 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 they have positive gain. Notice that the boundary refinement algorithm is quite similar to the KL algorithm, with the added advantage that only vertices are inserted into the data structures as needed and no work is wasted.

As with KL, we have a choice of performing a single pass (boundary KL(1) refinement (BKL(1))) or multiple passes (boundary Kernighan-Lin refinement (BKL)) 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 BKL and the speed of BKL(1) one can combine these schemes into a hybrid scheme that we refer to it as BKL(*,1). The idea behind the BKL(*,1) policy is to use BKL as long as the graph is small, and switch to BKL(1) when the graph is large. The motivation for this scheme is that single vertex swaps in the coarser graphs lead to larger decreases in the edge-cut than in the finer graphs. So by using BKL at these coarser graphs better refinement is achieved, and because these graphs are very small (compared to the size of the original graph), the BKL 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 BKL, otherwise BKL(1) is used. This choice of triggering condition relates the size of the partition boundary, which is proportional to the cost of performing the refinement of a graph, with the original size of the graph to determine when it is inexpensive to perform BKL relative to the size of the graph.

6 Experimental Results—Graph Partitioning

We evaluated the performance of the multilevel graph partitioning algorithm on a wide range of graphs 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 processor. All times reported are in seconds. Since the nature of the multilevel algorithm discussed is randomized, we performed all experiments with a fixed seed. Furthermore, the coarsening process ends when the coarse graph has fewer than 100 vertices.

As discussed in Sections 3, 4, and 5, 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 comparisons of different alternatives for each phase after making a reasonable choice for the other two phases.

6.1 Matching Schemes

We implemented the four matching schemes described in Section 3 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 BKL(*,1) 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).

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 5% of each other for most matrices. Out of these schemes, RM produces the best partition for two matrices, HEM for six matrices, LEM for three, and HCM for one.

The time spent in coarsening does not vary significantly across different schemes. But RM and HEM requires the least amount of time for coarsening, while LEM and HCM require the most (up to 30% more time than RM). This is not surprising since RM looks for the first unmatched neighbor of a vertex (the adjacency lists are randomly permuted).

Graph Name	No. of Vertices	No. of Edges	Description
144	144649	1074393	3D Finite element mesh
4ELT	15606	45878	2D Finite element mesh
598A	110971	741934	3D Finite element mesh
ADD32	4960	9462	32-bit adder
AUTO	448695	3314611	3D Finite element mesh
BCSSTK30	28294	1007284	3D Stiffness matrix
BCSSTK31	35588	572914	3D Stiffness matrix
BCSSTK32	44609	985046	3D Stiffness matrix
BBMAT	38744	993481	2D 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
FINAN512	74752	261120	Linear programming
FLAP	51537	479620	3D Stiffness matrix
INPRO1	46949	1117809	3D Stiffness matrix
KEN-11	14694	33880	Linear programming
LHR10	10672	209093	Chemical engineering
LHR71	70304	1449248	Chemical engineering
M14B	214765	3358036	3D Finite element mesh
MAP1	267241	334931	Highway network
MAP2	78489	98995	Highway network
MEMPLUS	17758	54196	Memory circuit
PDS-20	33798	143161	Linear programming
PWT	36519	144793	3D Finite element mesh
ROTOR	99617	662431	3D Finite element mesh
S38584.1	22143	35608	Sequential circuit
SHELL93	181200	2313765	3D Stiffness matrix
SHYY161	76480	152002	CFD/Navier-Stokes
TORSO	201142	1479989	3D Finite element mesh
TROLL	213453	5885829	3D Stiffness matrix
VENKAT25	62424	827684	2D Coefficient matrix
WAVE	156317	1059331	3D Finite element mesh

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

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 (up to 4% 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 the 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). 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 as described in Appendix A.3. The time spent during uncoarsening for RM is also higher than the time required by the HEM scheme by up to 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 the introduction, this is of particular importance when the parallel formulation of the multilevel algorithm is considered [29].

As the experiments show, HEM is an excellent matching scheme that results in good initial partitions, and requires the smallest overall run time. We selected the HEM as our matching scheme of choice because of its consistently good behavior.

	RM			HEM			LEM			HCM		
	32EC	CTime	UTime	32EC	CTime	UTime	32EC	CTime	UTime	32EC	CTime	UTime
BCSSTK31	44810	5.93	2.46	45991	6.25	1.95	42261	7.65	4.90	44491	7.48	1.92
BCSSTK32	71416	9.21	2.91	69361	10.06	2.34	69616	12.13	6.84	71939	12.06	2.36
BRACK2	20693	6.06	3.41	21152	6.54	3.33	20477	6.90	4.60	19785	7.47	3.42
CANT	323.0K	19.70	8.99	323.0K	20.77	5.74	325.0K	25.14	23.64	323.0K	23.19	5.85
COPTER2	32330	5.77	2.95	30938	6.15	2.68	32309	6.54	5.05	31439	6.95	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	0.77	0.76	1894	0.80	0.78	1992	0.86	0.95	1879	0.92	0.74
INPRO1	78375	9.50	2.90	75203	10.39	2.30	76583	12.46	6.25	78272	12.34	2.30
ROTOR	38723	11.94	5.60	36512	12.11	4.90	37287	13.51	8.30	37816	14.59	5.10
SHELL93	84523	36.18	10.24	81756	37.59	8.94	82063	42.02	16.22	83363	43.29	8.54
TROLL	317.4K	62.22	14.16	307.0K	64.84	10.38	305.0K	81.44	70.20	312.8K	76.14	10.81
WAVE	73364	18.51	8.24	72034	19.47	7.24	70821	21.39	15.90	71100	22.41	7.20

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

	RM	HEM	LEM	HCM
BCSSTK31	144879	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 size of the edge-cut for a 32-way partition when no refinement was performed, for the various matching schemes.

6.2 Initial Partition Algorithms

As described in Section 4, 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).

The result of the partitioning algorithms for some matrices is shown in Table 4. These partitions were produced by using the heavy-edge matching (HEM) during coarsening and the BKL(*,1) refinement policy during uncoarsening. Four quantities are reported for each partitioning algorithm. These are: (a) the edge-cut of the initial partition of the coarsest graph (IEC), (b) the edge-cut of the 2-way partition (2EC), (c) the edge-cut of a 32-way partition (32EC), and (d) the combined time (IRTime) spent in partitioning (ITime) and refinement (RTime) for the 32-way partition (*i.e.*, IRTime = ITime + RTime).

A number of interesting observations can be made from Table 4. The edge-cut of the initial partition (IEC) for the GGGP scheme is consistently smaller than the other two schemes (4ELT is the only exception as SBP does slightly better). SBP takes more time than GGP or GGGP to partition the coarse graph. But ITime for all these schemes are fairly small (less than 20% of IRTime) in our experiments. Hence, much of the difference in the run time of the three different initial partition schemes is due to refinement time associated with each. Furthermore, SBP produces partitions that are significantly worse than those produced by GGP and GGGP (as it is shown in the IEC column of Table 4). This happens because either the iterative algorithm used to compute the eigenvector does not converge within the allowable number of iterations[4], or the initial partition found by the spectral algorithm is far from a local minimum.

When the edge cut of the 2-way and 32-way partition is considered, the SBP scheme still does worse than GGP and

[4] In our experiments we set the maximum number of iterations to 100.

	SBP				GGP				GGGP			
	IEC	2EC	32EC	IRTime	IEC	2EC	32EC	IRTime	IEC	2EC	32EC	IRTime
BCSSTK31	28305	3563	45063	2.74	7594	3563	43900	1.43	7325	3563	43991	1.40
BCSSTK32	17166	6006	74776	2.25	13506	6541	72745	1.70	11023	4856	68223	1.58
BRACK2	1771	846	22284	3.45	1508	774	21697	2.42	1335	765	20631	2.38
CANT	50211	18951	325394	4.18	41500	18941	326164	4.32	36542	18958	322709	3.46
COPTER2	14177	2883	31639	3.41	10301	2318	31947	1.91	7148	2191	30584	1.88
CYLINDER93	41934	21581	204752	2.71	32374	20621	201827	2.08	28956	20621	202702	2.01
4ELT	258	152	1788	1.0	274	153	1791	0.72	259	140	1755	0.70
INPRO1	18539	8146	79016	2.35	14575	7313	76190	1.59	13444	7455	74933	1.61
ROTOR	9869	2123	37006	4.75	6998	2123	39880	4.30	6479	2123	36379	3.36
SHELL93	49	0	91846	6.01	0	0	84197	5.02	0	0	82720	4.89
TROLL	138494	51845	318832	7.82	102518	48090	303842	6.37	95615	41817	312581	5.92
WAVE	56920	9987	74754	7.18	27020	9200	71774	4.84	24212	9086	71864	4.57

Table 4: Performance of various algorithms for performing the initial partition of the coarse graph.

GGGP, although the relative difference in values of 2EC (and also 32EC) is smaller than it is for IEC. For the 2-way partition SBP performs better for only one matrix, and for the 32-way partition for none. Comparing GGGP with GGP we see that, GGGP performs better than GGP for 9 matrices in the 2-way partition and for 9 matrices in the 32-way partition. On the average for 32EC, SBP does 4.3% worse than GGGP and requires 47% more time, and GGP does 2.4% worse than GGGP requiring 7.5% more time. Looking at the combined time required by partitioning and refinement we see that GGGP, in all but one case, requires the least amount of time. This is because the initial partition for GGGP is better than that for GGP; this good initial partition leads to less time spent in refinement during the uncoarsening phase. In particular, for each matrix the performance for GGGP is better or very close to the best scheme both in terms of edge-cut and runtime.

We also implemented the Kernighan-Lin partitioning algorithm (Section 4.2). Its performance was consistently worse than that of GGGP in terms of IEC, and it also required more overall run time. Hence, we omitted these results here.

In summary, the results in Table 4 show that GGGP consistently finds smaller edge-cuts than the other schemes, and even requires slightly smaller run time. Furthermore, there is no advantage in choosing spectral bisection for partitioning the coarse graph.

6.3 Refinement Policies

As described in Section 5, there are different ways that a partition can be refined during the uncoarsening phase. We evaluated the performance of five refinement policies, in terms of partition quality as well as execution time. The refinement policies that we evaluate are (a) single pass of Kernighan-Lin (KL(1)), (b) Kernighan-Lin refinement (KL), (c) single pass of boundary Kernighan-Lin refinement (BKL(1)), (d) boundary Kernighan-Lin refinement (BKL), and (e) the combination of BKL and BKL(1) (BKL(*,1)).

The result of these refinement policies for computing a 32-way partition of graphs corresponding to some of the matrices in Table 1 is shown in Table 5. These partitions were produced by using the heavy-edge matching (HEM) during coarsening and the greedy graph growing algorithm for initially partitioning the coarser graph.

A number of interesting conclusions can be drawn from Table 5. First, for each of the matrices and refinement policies, the size of the edge-cut does not vary significantly for different refinement policies; all are 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. KL requires the most time while BKL(1) requires the least.

Comparing KL(1) with KL, we see that KL performs better than KL(1) 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 KL is significantly higher than that of KL(1). Usually, KL requires two to three times more time than KL(1).

Comparing the KL(1) and KL refinement schemes against their boundary variants, we see that the times required by

	KL(1)		KL		BKL(1)		BKL		BKL(*,1)	
	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 5: 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.

the boundary policies are significantly less than those required by their non-boundary counterparts. The time of BKL(1) ranges from 29% to 75% of the time of KL(1), while the time of BKL ranges from 19% to 80% of the time of KL. This seems quite reasonable, given that BKL(1) and BKL are more efficient implementations of KL(1) and KL, respectively, that take advantage of the fact that the projected partition requires little refinement. But surprisingly, BKL(1) and BKL lead to better edge-cut (than KL(1) and KL, respectively) in many cases. On the average, BKL(1) performs similarly with KL(1), while BKL does better than KL by 2%. BKL(1) does better than KL(1) in 6 out of the 12 matrices, and BKL does better than KL in 10 out the 12 matrices. Thus, overall the quality of the boundary refinement policies is at least as good as that of their non-boundary counterparts.

The difference in quality between KL and BKL is because each algorithm inserts vertices into the KL data-structures in a different order. At any given time, we may have more than one vertex with the same largest gain. Thus, a different insertion order may lead to a different ordering of the vertices with the largest gain. Consequently, the KL and BKL algorithms may move different subsets of vertices from one part to the other.

Comparing BKL(1) with BKL we see that the edge-cut is better for BKL for nearly all matrices, and the improvement is relatively small (less than 4% on the average). However, the time required by BKL is always higher than that of BKL(1) (in some cases up to four times higher). Thus, marginal improvement in the partition quality comes at a significant increase in the refinement time. Comparing BKL(*,1) against BKL we see that its edge-cut is on the average within 2% of that of BKL, while its runtime is significantly smaller than that of BKL and only somewhat higher than that of BKL(1).

In summary, both the BKL and the BKL(*,1) refinement policies require substantially less time than KL, and produce smaller edge-cuts when coupled with the heavy-edge matching scheme. We believe that the BKL(*,1) refinement policy strikes a good balance between small edge-cut and fast execution.

6.4 Comparison of Our Multilevel Scheme 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 SYMMLQ [40]. By using this multilevel approach, the MSB algorithm is able to compute the Fiedler vector of the graph in much less time than that taken by the original spectral bisection algorithm. 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 [25] graph partitioning package to produce partitions for some of the matrices in Table 1 and compared the results against the partitions produced by our multilevel algorithm that uses HEM during coarsening phase, GGGP during

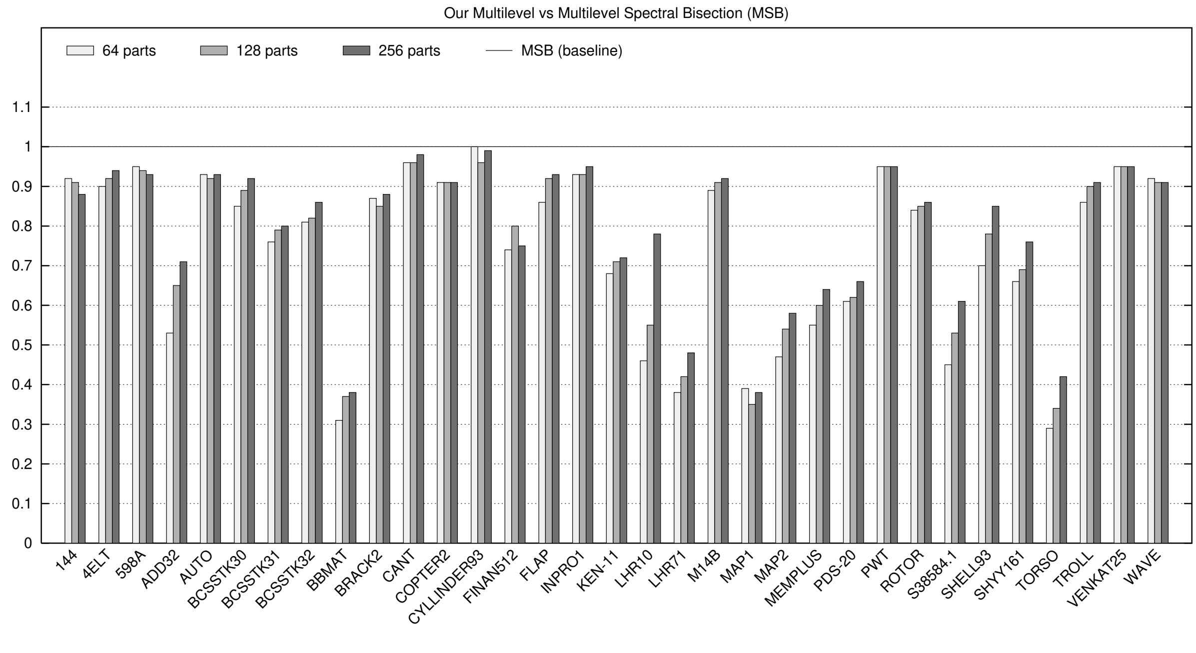

Figure 3: Quality of our multilevel algorithm compared to the multilevel spectral bisection algorithm. For each matrix, the ratio of the cut-size of our multilevel algorithm to that of the MSB algorithm is plotted for 64-, 128- and 256-way partitions. Bars under the baseline indicate that our multilevel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

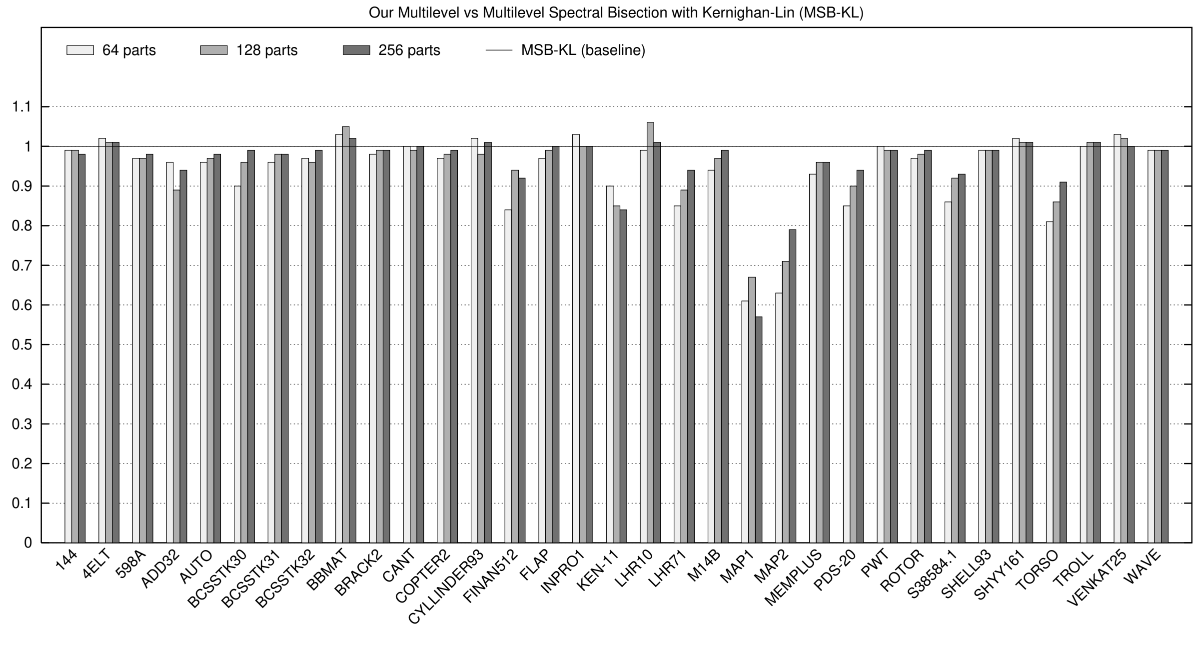

Figure 4: Quality of our multilevel algorithm compared to the multilevel spectral bisection algorithm with Kernighan-Lin refinement. For each matrix, the ratio of the cut-size of our multilevel algorithm to that of the MSB-KL algorithm is plotted for 64-, 128- and 256-way partitions. Bars under the baseline indicate that our multilevel algorithm performs better.

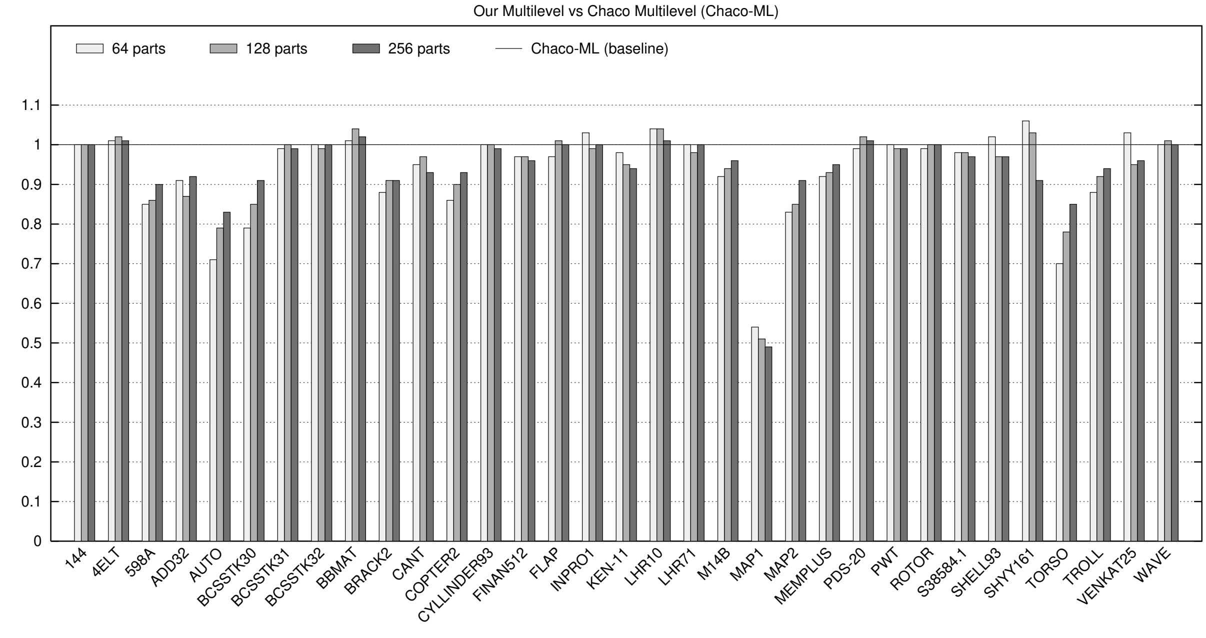

Figure 5: Quality of our multilevel algorithm compared to the multilevel Chaco-ML algorithm. For each matrix, the ratio of the cut-size of our multilevel algorithm to that of the Chaco-ML algorithm is plotted for 64-, 128- and 256-way partitions. Bars under the baseline indicate that our multilevel algorithm performs better.

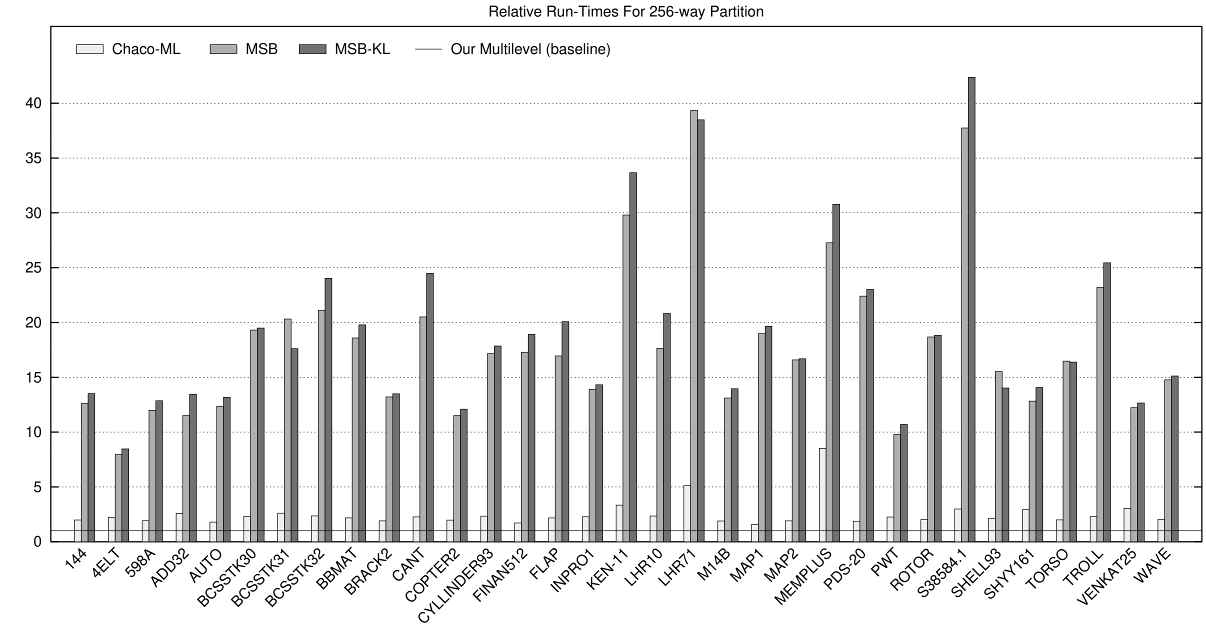

Figure 6: The time required to find a 256-way partition for Chaco-ML, MSB, and MSB-KL relative to the time required by our multilevel algorithm.

partitioning phase, and BKL(*,1) during the uncoarsening phase.

Figure 3 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 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 70%. Furthermore, the time required by our multilevel algorithm is significantly smaller than that required by MSB. Figure 6 shows the time required by different algorithms relative to that required by our multilevel algorithm. From Figure 6, we see that compared with MSB, our algorithm is usually 10 times faster for small problems, and 15 to 35 times faster for larger problems.

One way of improving the quality of MSB algorithm is to use the Kernighan-Lin algorithm to refine the partitions (MSB-KL). Figure 4 shows the relative performance of our multilevel algorithm compared against the MSB-KL algorithm. Comparing Figures 3 and 4 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 6, making the difference in the run time of MSB-KL and our multilevel algorithm even greater.

	MSB			MSB-KL			Chaco-ML			Our Multilevel		
Matrix	64EC	128EC	256EC	64EC	128EC	256EC	64EC	128EC	256EC	64EC	128EC	256EC
144	96538	132761	184200	89272	122307	164305	89068	120688	161798	88806	120611	161563
4ELT	3303	5012	7350	2909	4567	6838	2928	4514	6869	2965	4600	6929
598A	68107	95220	128619	66228	91590	121564	75490	103514	133455	64443	89298	119699
ADD32	1267	1934	2728	705	1401	2046	738	1446	2104	675	1252	1929
AUTO	208729	291638	390056	203534	279254	370163	274696	343468	439090	194436	269638	362858
BCSSTK30	224115	305228	417054	211338	284077	387914	241202	318075	423627	190115	271503	384474
BCSSTK31	86244	123450	176074	67632	99892	143166	65764	98131	141860	65249	97819	140818
BCSSTK32	130984	185977	259902	109355	158090	225041	106449	153956	223181	106440	152081	222789
BBMAT	179282	250535	348124	54095	88133	129331	55028	89491	130428	55753	92750	132387
BRACK2	34464	49917	69243	30678	43249	61363	34172	46835	66944	29983	42625	60608
CANT	459412	598870	798866	444033	579907	780978	463653	592730	835811	442398	574853	778928
COPTER2	47862	64601	84934	45178	59996	78247	51005	65675	82961	43721	58809	77155
CYLINDER93	290194	431551	594859	285013	425474	586453	289837	417837	595055	289639	416190	590065
FINAN512	15360	27575	53387	13552	23564	43760	11753	22857	41862	11388	22136	40201
FLAP	35540	54407	80392	31710	50111	74937	31553	49390	74416	30741	49806	74628
INPRO1	125285	185838	264049	113651	172125	249970	113852	172875	249964	116748	171974	250207
KEN-11	20931	23308	25159	15809	19527	21540	14537	17417	19178	14257	16515	18101
LHR10	127778	148917	178160	59648	77694	137775	56667	79464	137602	58784	82336	139182
LHR71	540334	623960	722101	239254	292964	373948	204654	267197	350045	203730	260574	350181
M14B	124749	172780	232949	118186	161105	216869	120390	166442	222546	111104	156417	214203
MAP1	3546	6314	8933	2264	3314	5933	2564	4314	6933	1388	2221	3389
MAP2	1759	2454	3708	1308	1860	2714	1002	1570	2365	828	1328	2157
MEMPLUS	32454	33412	36760	19244	20927	24388	19375	21423	24796	17894	20014	23492
PDS-20	39165	48532	58839	28119	33787	41032	24083	29650	38104	23936	30270	38564
PWT	9563	13297	19003	9172	12700	18249	9166	12737	18268	9130	12632	18108
ROTOR	63251	88048	120989	54806	76212	105019	53804	75140	104038	53228	75010	103895
S38584.1	5381	7595	9609	2813	4364	6367	2468	4077	6076	2428	3996	5906
SHELL93	178266	238098	318535	126702	187508	271334	122501	191787	276979	124836	185323	269539
SHYY161	6641	9151	11969	4296	6242	9030	4133	6124	9984	4365	6317	9092
TORSO	413501	473397	522717	145149	186761	241020	168385	205393	257604	117997	160788	218155
TROLL	529158	706605	947564	455392	630625	851848	516561	691062	916439	453812	638074	864287
VENKAT25	50184	77810	116211	46019	72249	110331	45918	77889	114553	47514	73735	110312
WAVE	106858	142060	187192	98720	131484	172957	97558	128792	170763	97978	129785	171101

Table 6: The edge-cuts produced by the multilevel spectral bisection (MSB), multilevel spectral bisection followed by Kernighan-Lin (MSB-KL), the multilevel algorithm implemented in Chaco (Chaco-ML), and our multilevel algorithm.

The graph partitioning package Chaco implements its own multilevel graph partitioning algorithm that is modeled after the algorithm by Hendrickson and Leland [26, 25]. 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 5 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 45%. For the cases where Chaco-ML does better, it is only marginally better (less than 2%). Our algorithm is usually two to seven times faster than Chaco-ML. Most of the savings come from the choice of refinement policy (we use BKL(*,1)) which is usually four to six times faster than the Kernighan-Lin refinement implemented by Chaco-ML. Note that we are able to use BKL(*,1) 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. This makes a difference in those cases in which the graph coarsening phase aborts before the number of vertices becomes very small. Also, for some problems, the Lanczos algorithm does not converge, which explains the poor performance of Chaco-ML for graphs such as MAP1.

Table 6 shows the edge-cuts for 64-way, 128-way, and 256-way partition for different algorithms. Table 7 shows the run time of different algorithms for finding a 256-way partition.

Matrix	MSB	MSB-KL	Chaco-ML	Our Multilevel
144	607.27	650.76	95.59	48.14
4ELT	24.95	26.56	7.01	3.13
598A	420.12	450.93	67.27	35.05
ADD32	18.72	21.88	4.23	1.63
AUTO	2214.24	2361.03	322.31	179.15
BCSSTK30	426.45	430.43	51.41	22.08
BCSSTK31	309.06	268.09	39.68	15.21
BCSSTK32	474.64	540.60	53.10	22.50
BBMAT	474.23	504.68	55.51	25.51
BRACK2	218.36	222.92	31.61	16.52
CANT	978.48	1167.87	108.38	47.70
COPTER2	185.39	194.71	31.92	16.11
CYLINDER93	671.33	697.85	91.41	39.10
FINAN512	311.01	340.01	31.00	17.98
FLAP	279.67	331.37	35.96	16.50
INPRO1	341.88	352.11	56.05	24.60
KEN-11	121.94	137.73	13.69	4.09
LHR10	142.58	168.26	18.95	8.08
LHR71	2286.36	2236.19	297.02	58.12
M14B	970.58	1033.82	140.34	74.04
MAP1	850.16	880.16	71.17	44.80
MAP2	195.09	196.34	22.41	11.76
MEMPLUS	117.89	133.05	36.87	4.32
PDS-20	249.93	256.90	20.85	11.16
PWT	70.09	76.67	16.22	7.16
ROTOR	550.35	555.12	59.46	29.46
S38584.1	178.11	199.96	14.11	4.72
SHELL93	1111.96	1004.01	153.86	71.59
SHYY161	129.99	142.56	29.82	10.13
TORSO	1053.37	1046.89	127.76	63.93
TROLL	3063.28	3360.00	302.15	132.08
VENKAT25	254.52	263.34	63.49	20.81
WAVE	658.13	673.45	90.53	44.55

Table 7: The time required to find a 256-way partition by the multilevel spectral bisection (MSB), multilevel spectral bisection followed by Kernighan-Lin (MSB-KL), the multilevel algorithm implemented in Chaco (Chaco-ML), and our multilevel algorithm. All times are in seconds.

7 Experimental Results—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. In the nested dissection ordering algorithms, a vertex separator is computed from the edge separator of a 2-way partition. 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 [41, 44]. The minimum vertex cover has been found to produce very small vertex separators.

The overall quality of a fill reducing ordering depends on whether or not the matrix is factored on a serial or parallel

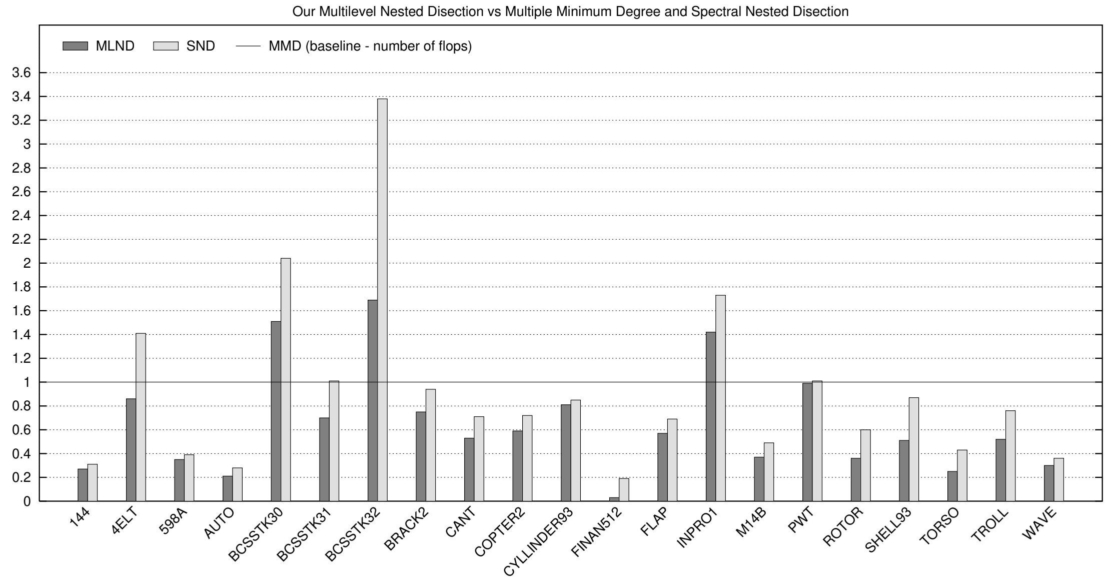

Figure 7: Quality of our multilevel nested dissection relative to the multiple minimum degree, and the spectral nested dissection algorithm. Bars under the baseline indicates that MLND performs better than MMD.

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 non-zeros in the Cholesky factors. The fewer non-zeros usually lead to fewer operations. However, similar fills may have different operation counts; hence, 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 [13, 35].

The minimum degree [13] 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 [35] 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 (MLND) compared to that of MMD is shown in Table 8 and Figure 7. For our multilevel algorithm, we used the HEM scheme during coarsening, the GGGP scheme for partitioning the coarse graph and the BKL(*,1) refinement policy during the uncoarsening phase. Looking at this figure we see that our algorithm produces better orderings for 18 out of the 21 test problems. For the other three problems MMD does better. Also, from Figure 7 we see that MLND does consistently better as the size of the matrices increases. In particular, for large finite element meshes, such as AUTO, MLND requires half the amount of memory required by MMD, and 4.7 times fewer operations. When all 21 test matrices are considered, MMD produces orderings that require a total of 4.81 teraflops, whereas the orderings produced by MLND require only 1.23 teraflops. Thus, the ensemble of 21 matrices can be factored roughly 3.9 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 [32, 12, 18]. On the other hand, orderings based on nested dissection produce orderings that have both more concurrency and better balance [30, 22]. 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 is usually two to three times faster than MLND for ordering the matrices in Table 1. However, efforts to parallelize the MMD algorithm have had no success [14]. In fact, the MMD algorithm appears to be inherently

Matrix	MMD	SND	MLND
144	2.4417e+11	7.6580e+10	**6.4756e+10**
4ELT	1.8720e+07	2.6381e+07	**1.6089e+07**
598A	6.4065e+10	2.5067e+10	**2.2659e+10**
AUTO	2.8393e+12	7.8352e+11	**6.0211e+11**
BCSSTK30	**9.1665e+08**	1.8659e+09	1.3822e+09
BCSSTK31	2.5785e+09	2.6090e+09	**1.8021e+09**
BCSSTK32	**1.1673e+09**	3.9429e+09	1.9685e+09
BRACK2	3.3423e+09	3.1463e+09	**2.4973e+09**
CANT	4.1719e+10	2.9719e+10	**2.2032e+10**
COPTER2	1.2004e+10	8.6755e+09	**7.0724e+09**
CYLINDER93	6.3504e+09	5.4035e+09	**5.1318e+09**
FINAN512	5.9340e+09	1.1329e+09	**1.7301e+08**
FLAP	1.4246e+09	9.8081e+08	**8.0528e+08**
INPRO1	**1.2653e+09**	2.1875e+09	1.7999e+09
M14B	2.0437e+11	9.3665e+10	**7.6535e+10**
PWT	1.3819e+08	1.3919e+08	**1.3633e+08**
ROTOR	3.1091e+10	1.8711e+10	**1.1311e+10**
SHELL93	1.5844e+10	1.3844e+10	**8.0177e+09**
TORSO	7.4538e+11	3.1842e+11	**1.8538e+11**
TROLL	1.6844e+11	1.2844e+11	**8.6914e+10**
WAVE	4.2290e+11	1.5351e+11	**1.2602e+11**

Table 8: The number of operations required to factor various matrices when ordered with multiple minimum degree (MMD), spectral nested dissection (SND), and our multilevel nested dissection (MLND).

serial in nature. On the other hand, the MLND algorithm is amenable to parallelization. In [29] we present a parallel formulation of our MLND algorithm that achieves a speedup of as much as 57 on 128-processor Cray T3D (over the serial algorithm running on a single T3D processor) for some graphs.

Spectral nested dissection (SND) [45] can be used for ordering matrices for parallel factorization. The SND algorithm is based on the spectral graph partitioning algorithm described in Section 4.1. We have implemented the SND algorithm described in [45]. 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 7. From this figure we can see that MLND produces orderings that are better than SND for all 21 test matrices. The total number of operations required to factor the matrices ordered using SND is 1.68 teraflops which is 37% more than the of MLND. However, as discussed in Section 6.4, the runtime of SND is substantially higher than that of MLND. Also, SND cannot be parallelized any better than MLND [29, 1]; therefore, it will always be slower than MLND.

8 Characterization of Different Graph Partitioning Schemes

Due to the importance of the problem, a large number of graph partitioning schemes have been developed. These schemes differ widely in the edge-cut quality produced, run time, degree of parallelism, and applicability to certain kind of graphs. Often, it is not clear as to which scheme is better under what scenarios. In this section, we categorize these properties of some graph partitioning algorithms that are commonly used in finite element applications. This task is quite difficult, as it is not possible to precisely model the properties of the graph partitioning algorithms. Furthermore, we don't have enough data on the edge-cut quality and run time for a common pool of benchmark graphs. This paper presents extensive comparisons of multilevel scheme with MSB and MSB-KL. Limited comparison with other schemes can be made by looking at the edge-cut quality and run time for graphs that are used in this paper as well as in the evaluation of other schemes elsewhere. We try to make reasonable assumptions whenever enough data is not available. For the sake of simplicity, we have chosen to represent each property in terms of a small discrete scale. In absence of extensive data, we could not have done any better anyway.

In Table 9 we show three different variations of spectral partitioning [47, 46, 26, 2], the multilevel partitioning described in this paper, the levelized nested dissection [11], the Kernighan-Lin partition [31], the coordinate nested dissection (CND) [23], two variations of the inertial partition [38, 25], and two variants of geometric partitioning [37, 36, 16].

	Number of Trials	Needs Coordinates	Quality	Local View	Global View	Run Time	Degree of Parallelism
Spectral Bisection	1	no	●●●●	○	●●●●	■■■■	▲▲
Multilevel Spectral Bisection	1	no	●●●●	○	●●●●	■■■	▲▲
Mulitlevel Spectral Bisection-KL	1	no	●●●●●●	●●	●●●●	■■■	▲▲
Multilevel Partitioning	1	no	●●●●●●	●●	●●●●	■■	▲▲
Levelized Nested Dissection	1	no	●●	○	●●	■■	▲▲
Kernighan-Lin	1	no	●●	●●	○	■■	▲
	10	no	●●●◐	●●	●◐	■■■	▲▲
	50	no	●●●●	●●	●●	■■■◧	▲▲
Coordinate Nested Dissection	1	yes	●	○	●	■	▲▲▲
Inertial	1	yes	●●	○	●●	■	▲▲▲
Inertial-KL	1	yes	●●●●	●●	●●	■■	▲
Geometric Partitioning	1	yes	●●	○	●●	■	▲▲▲
	10	yes	●●●◐	○	●●●◐	■■	▲▲▲
	50	yes	●●●●	○	●●●●	■■◧	▲▲▲
Geometric Partitioning-KL	1	yes	●●●●	●●	●●	■■	▲
	10	yes	●●●●●◐	●●	●●●◐	■■■	▲▲
	50	yes	●●●●●●	●●	●●●●	■■■◧	▲▲

Table 9: Characteristics of various graph partitioning algorithms.

For each graph partitioning algorithm, Table 9 shows a number of characteristics. The first column shows the number of trials that are often performed for each partitioning algorithm. For example, for Kernighan-Lin, different trials can be performed each starting with a random partition of the graph. Each trial is a different run of the partitioning algorithm, and the overall partition is determined as the best of these multiple trials. As we can see from this table, some algorithms require only a single trial either because, multiple trials will give the same partition (*i.e.*, the algorithm is deterministic), or the single trial gives very good results (as in the case of multilevel graph partitioning). However, for some schemes like Kernighan-Lin and geometric partitioning, different trials yield significantly different edge-cuts because these schemes are highly sensitive to the initial partition. Hence, these schemes usually require multiple trials in order to produce good quality partitions. For multiple trials, we only show the case of 10 and 50 trials, as often the quality saturates beyond 50 trials, or the run time becomes too large. The second column shows whether the partitioning algorithm requires coordinates for the vertices of the graph. Some algorithms such as CND and Inertial can work only if coordinate information is available. Others only require the set of vertices and edges connecting them.

The third column of Table 9 shows the relative quality of the partitions produced by the various schemes. Each additional circle corresponds to roughly 10% improvement in the edge-cut. The edge-cut quality for CND serves as the base, and it is shown with one circle. Schemes with two circles for quality should find partitions that are roughly 10% better than CND. This column shows that the quality of the partitions produced by our multilevel graph partitioning algorithm and the multilevel spectral bisection with Kernighan-Lin is very good. The quality of geometric partitioning with Kernighan-Lin refinement is also equally good, when around 50 or more trials are performed[5]. The quality of

[5]This conclusion is an extrapolation of the results presented in [16] where it was shown that the geometric partitioning with 30 trials (*default*

the other schemes is worse than the above three by various degrees. Note that for both Kernighan-Lin partitioning and geometric partitioning the quality improves as the number of trials increases.

The reason for the differences in the quality of the various schemes can be understood if we consider the degree of quality as a sum of two quantities that we refer to as ***local view*** and ***global view***. A graph partitioning algorithm has a local view of the graph if it is able to do localized refinement. According to this definition, all the graph partitioning algorithms that use at various stages of their execution variations of the Kernighan-Lin partitioning algorithm possess this local view, whereas the other graph partitioning algorithms do not. Global view refers to the extent that the graph partitioning algorithm takes into account the structure of the graph. For instance, spectral bisection algorithms take into account only global information of the graph by minimizing the edge-cut in the continuous approximation of the discrete problem. On the other hand, schemes such as a single trial of Kernighan-Lin, utilize no graph structural information, since it starts from a random bisection. Schemes that require multiple trials, improve the amount of global graph structure they exploit as the number of trials increases. Note that the sum of circles for global and local view columns is equal to the number of circles for quality for various algorithms. The global view of multilevel graph partitioning is among the highest of that of the other schemes. This is because the multilevel graph partitioning captures global graph structure at two different levels. First, it captures global structure through the process of coarsening [27], and second, it captures global structure during the initial graph partitioning by performing multiple trials.

The sixth column of Table 9 shows the relative time required by different graph partitioning schemes. CND, inertial, and geometric partitioning with one trial require relatively small amount of time. We show the run time of these schemes by one square. Each additional square corresponds to roughly a factor of 10 increase in the run time. As we can see, spectral graph partition schemes require several orders of magnitude more time than the faster schemes. However, the quality of the partitions produced by the faster schemes is relatively poor. The quality of the geometric partitioning scheme can be improved by increasing the number of trials and/or by using the Kernighan-Lin algorithm, both of which significantly increase the run time of this scheme. On the other hand, multilevel graph partitioning requires moderate amount of time, and produces partitions of very high quality.

The degree of parallelizability of different schemes differs significantly and is depicted by a number of triangles in the seventh column of Table 9. One triangle means that the scheme is largely sequential, two triangles means that the scheme can exploit a moderate amount of parallelism, and three triangles means that the scheme can be parallelized quite effectively. Schemes that require multiple trials are inherently parallel, as different trials can be done on different processors. In contrast, a single trial of Kernighan-Lin is very difficult to parallelize [15], and appears inherently serial in nature. Multilevel schemes that do not rely upon Kernighan-Lin [29] and the spectral bisection scheme are moderately parallel in nature. As discussed in [29], the asymptotic speedup for these schemes is bounded by $O(\sqrt{p})$. $O(p)$ speedup can be obtained in these schemes only if the graph is nearly well partitioned among processors. This can happen if the graph arises from an adaptively refined mesh. Schemes that rely on coordinate information do not seem to have this limitation, and in principle it appears that these schemes can be parallelized quite effectively. However, all available parallel formulation of these schemes [23, 8] obtained no better speedup than obtained for the multilevel scheme in [29].

9 Conclusion and Direction for Future Research

Our experiments with multilevel schemes have shown that they work quite well for many different types of coarsening, initial partition, and refinement schemes. In particular, all the coarsening schemes we experimented with, provide a good global view of the graph, and the Kernighan-Lin algorithm or its variants used for refinement during the uncoarsening phase provide a good local view. Due to the combined global and local view provided by the coarsening and refinement schemes, the choice of the algorithm used to partition the coarse graph seems to have relatively small effect on the overall quality of the partition. In particular, there seems to be no advantage gained by using spectral bisection for partitioning the coarsest graph. The multilevel algorithm when used to find a fill reducing ordering is consistently better than spectral nested dissection, and substantially better than multiple minimum degree for large graphs. The rea-

geometric) produces partitions comparable to that of multilevel spectral bisection without Kernighan-Lin refinement.

son is that unlike the multilevel algorithm, the multiple minimum degree algorithm does not have a global view of the graph, which is critical for good performance on large graphs. The multilevel algorithm that uses HEM for coarsening and BKL(1) or BKL(*,1) for refinement can be parallelized effectively. The reason is that this combination requires very little time for refinement, which is the most serial part of the algorithm. The coarsening phase is relatively much easier to parallelize [29].

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] Earl R. Barnes. An algorithm for partitioning the nodes of a graph. *SIAM J. Algebraic Discrete Methods*, 3(4):541–550, December 1984.

[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] T. N. Bui, S. Chaudhuri, F. T. Leighton, and M. Sipser. Graph bisection algorithms with good average case behavior. *Combinatorica*, 7(2):171–191, 1987.

[6] Tony F. Chan, John R. Gilbert, and Shang-Hua Teng. Geometric spectral partitioning. Technical report, Xerox PARC Tech. Report., 1994. Available at ftp://parcftp.xerox.com/pub/gilbert/index.html.

[7] 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.

[8] 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.

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

[10] 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.

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

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

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

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

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

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

[17] T. Goehring and Y. Saad. Heuristic algorithms for automatic graph partitioning. Technical report, Department of Computer Science, University of Minnesota, Minneapolis, 1994.

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

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

[20] 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.

[21] S.W. Hammond. *Mapping Unstructured Grid Problems to Massively Parallel Computers*. PhD thesis, Rensselaer Polytechnic Institute, Troy, New York, 1992.

[22] 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.

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

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

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

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

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

[28] G. Karypis and V. Kumar. Multilevel graph partition and sparse matrix ordering. In *Intl. Conf. on Parallel Processing*, 1995. Available on WWW at URL http://www.cs.umn.edu/˜karypis/papers/mlevel_serial.ps.

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

[30] George Karypis, Anshul Gupta, and Vipin Kumar. A parallel formulation of interior point algorithms. In *Supercomputing 94*, 1994. Available on WWW at URL http://www.cs.umn.edu/˜karypis/papers/interior-point.ps.

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

[32] 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.

[33] Tom Leighton and Satish Rao. An approximate max-flow min-cut theorem for uniform multicommodity flow problems with applications to approximation algorithms. In *29th Annual Symposium on Foundations of Computer Science*, pages 422–431, 1988.

[34] R. J. Lipton and R. E. Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36:177–189, 1979.

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

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

[37] 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.

[38] 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.

[39] Jr. P. Ciarlet and F. Lamour. On the validity of a front-oriented approach to partitioning large sparse graphs with a connectivity constraint. Technical Report 94-37, Computer Science Department, UCLA, Los Angeles, CA, 1994.

[40] C. C. Paige and M. A. Saunders. Solution to sparse indefinite systems of linear equations. *SIAM Journal on Numerical Ananlysis*, 12:617–629, 1974.

[41] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization*. Prentice Hall, Englewood Cliffs, NJ, 1982.

[42] B. N. Parlett. *The Symmetric Eigenvalue Problem*. Prentice Hall, Englewood Cliffs, NJ, 1980.

[43] 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.

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

[45] 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.

[46] Alex Pothen, H. D. Simon, Lie Wang, and Stephen T. Bernard. Towards a fast implementation of spectral nested dissection. In *Supercomputing '92 Proceedings*, pages 42–51, 1992.

[47] 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.

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

A Implementation Framework

The multilevel algorithm described in Section 2.1 consists of a number of different algorithms. Efficient implementation of these algorithms and good methods of passing information between the various phases is essential to the overall fast execution of the algorithm. In the next sections we briefly describe the algorithms and data structures we used for the implementation of the algorithm.

A.1 Graph Data Structure

The data structure used to store graph $G = (V, E)$ consists of two arrays. The first array called *Vtxs* stores information about the vertices while the second array called *Adjncy* stores the adjacency lists of the vertices. For each vertex $v \in V$, *Vtxs[v]* contains the following five quantities:

vwgt the weight of v.

nedges the size of the adjacency list of v.

iedges the index into *Adjncy* that is the beginning of the adjacency list of v.

cewgt the weight of the edges that have been contracted to create v (if v is a multinode).

adjwgt the sum of the weight of the edges adjacent to v.

These quantities are used during different phases of the multilevel algorithm, and greatly improve the performance of the multilevel algorithm. Also, as the next section shows, they are computed incrementally during coarsening, hence they do not increase the overall run time of the algorithm.

A.2 Coarsening Phase

Each coarsening phase consists of two different stages. During the first stage (matching stage), a matching is computed using either the RM, HEM, LEM, or HCM schemes, while during the second stage (contraction stage), a coarser graph is created by contracting the vertices as dictated by the matching.

The output of the matching stage, is two vectors *Match* and *Map*, such that for each vertex v, *Match*$[v]$ stores the vertex with which v has been matched, and *Map*$[v]$ stores the label of v in the coarser graph. If during the matching stage, vertex v remains unmatched, then *Match*$[v] = v$. Note that for every pair of matched vertices, (v, u), *Map*$[v]$ =*Map*$[u]$. Initially, the vector *Match* is initialized to indicate that all vertices are unmatched. Since all the matching schemes use randomized algorithms, a random permutation vector is created to guide the order at which vertices are visited. As the vertices are visited in random order, they are assigned consecutive labels.

The RM matching scheme requires that a vertex is randomly matched with one if its unmatched adjacent vertices. To avoid having to traverse the entire adjacency list to find all the unmatched vertices and then randomly select one of them, we initially permute the adjacency lists of all the vertices of G_0 randomly. By doing that we only have to look for the first unmatched vertex, since the randomization of G_0 coupled with the random visitation order ensures good randomization.

During the contraction step, the *Match*, and *Map* vectors are used to contract the graph. Let v_1, v_2 be two vertices that have been matched. The label of the contracted vertex is u_1 =*Map*[v_1]. The vertices adjacent to u_1 are given by

$$Adj(u_1) = \Big(\{Map[x] | x \in Adj(v_1)\} \bigcup \{Map[x] | x \in Adj(v_2)\}\Big) - \{u_1\},$$

and the weight of an edge (u_1, u_2) is given by

$$w(u_1, u_2) = \sum_x \{w(v_1, x) | Map[x] = u_2\} + \sum_x \{w(v_2, x) | Map[x] = u_2\}.$$

To efficiently implement the above operation for all matched vertices, we use a table to keep track of the vertices seen so far. These data structures allow us to implement graph contraction by visiting each edge only once; thus, graph contraction takes time proportional to the number of edges.

Also, while computing the adjacency list for each vertex u_1 in the coarser graph, we also compute the remaining three quantities associated with each vertex (*i.e.*, *vwgt*, *cewgt*, and *adjwgt*). The *vwgt* of u_1 is computed as the sum of the *vwgt*s of v_1 and v_2. The *cewgt* of u_1 is computed as the sum of the *cewgt*s of v_1 and v_2, plus the weight of the edge connecting these vertices. Finally the *adjwgt* of u_1 is computed sum of the *adjwgt*s of v_1 and v_2, minus the weight of the edge connecting them.

A.3 Uncoarsening Phase

The uncoarsening phase consists of two separate stages. In the fist stage, the partition P_{i+1} of the graph G_{i+1} is projected back to G_i (projection stage), and during the second stage, P_i is refined using one of the refinement schemes described in Section 5 (refinement stage).

All the various partition refinement schemes described in Section 5 are based on swapping vertices between partitions based on the reduction in the edge-cut using variations of the Kernighan-Lin algorithm. As described in Section 4.2, the selections made by the Kernighan-Lin algorithm are driven by the gain value of a vertex (Equation 3). The gain values are often computed using two arrays ID and ED where for each vertex v,

$$ID[v] = \sum_{(v,u) \in E \wedge P[v] = P[u]} w(v, u), \quad \text{and} \quad ED[v] = \sum_{(v,u) \in E \wedge P[v] \neq P[u]} w(v, u). \tag{4}$$

The value $ID[v]$ is called the *internal degree* of v, and is the sum of the edge-weights of the adjacent vertices of v that are in the same partition as v, and the value of $ED[v]$ is called the *external degree* of v and is the sum of the edge-weights of the adjacent vertices of v that are at a different partition. Given these arrays, the gain of a vertex v is given by $g_v = ED[v] - ID[v]$. Note that the edge-cut of a partition is given by $0.5 \sum_v ED[v]$, and vertex v belongs at the boundary if and only if $ED[v] > 0$.

In our implementation, after partitioning the coarse graph G_m using one of the algorithms described in Section 4, the internal and external degrees of the vertices of G_m are computed using Equation 4. This is the only time that these quantities are computed explicitly. The internal and external degrees of all other graphs G_i with $i < m$, are computed incrementally during the projection stage. This is done as follows. Consider vertex $v \in V_i$ and let v_1 and v_2 be the vertices of V_{i-1} that where combined into v. Depending on the values of $ID[v]$, and $ED[v]$ we have three different cases.

$ED[v] = 0$
: In this case, $ED[v_1] = 0$ and $ID[v_1]$ is equal to the sum of the edge-weights of v_1. Similarly $ED[v_2] = 0$ and $ID[v_2]$ is equal to the sum of the edge-weights of v_2.

$ID[v] = 0$
: In this case $ID[v_1]$ and $ID[v_2]$ is equal to the weight of the edge (v_1, v_2) that can be computed from the difference

of the contracted edge weights of v_1, v_2, and v. The value for $ED[v_1]$ ($ED[v_2]$) is equal to the sum of the edge-weights of v_1 (v_2) minus $ID[v_1]$ ($ID[v_2]$).

$ED[v] > 0$ **and** $ID[v] > 0$
In this case the value of $ID[v_1]$ and $ED[v_1]$ are computed explicitly, and the values of $ID[v_2]$ and $ED[v_2]$ are computed as a difference of those for v_1 and v. Specifically $ED[v_2] = ED[v] - ED[v_1]$, and $ID[v_2] = ID[v] - ID[v_1] - w(v_1, v_2)$.

Thus, only when vertex v is at the partition boundary we need to explicitly compute its internal and/or external degrees. Since boundary vertices are a small percentage of the total number of vertices, computing the internal and external degrees during projection results in dramatic speed improvements. During the refinement stage, the internal and external degrees are kept consistent with respect to the partition. This is done by updating the degrees of the vertices adjacent to the one just being moved from one partition to the other (a computation that is required by the Kernighan-Lin algorithm), and by rolling back any speculative computation at the end of the refinement algorithm.

Given the above framework, the boundary refinement algorithms described in Section 5.2 require inserting into the data structures only the vertices whose external degree is positive.

A.4 Data Structures for Kernighan-Lin

As discussed in Section 4.2, the efficiency of the KL algorithm depends on the data structure used to store the gains of the vertices that have not been swapped yet. In our algorithm, for each partition, depending on the level of the coarse graph, we use either a doubly-linked list of gain buckets, or a table of gain-buckets.

The doubly-linked list is maintained in a decreasing gain order. Vertices with the same gain are inserted in the same gain-bucket. The table of gain-buckets, contains an entry for each possible value of the gain, and is effective when the range of values is small. This is usually the case for G_i when i is small. When i is large (coarser graphs), the range of values that the gain can get is high, making this implementation more expensive that the one that uses linked-lists.

Each gain-bucket is implemented as a doubly-linked list and contains the vertices that have a particular gain. An auxiliary table is used that stores for each vertex v a pointer to the node of the gain-bucket that stores v. This table allow us to locate the gain-bucket's node for a vertex in constant time.

When the gains are stored using a doubly-linked list of gain-buckets, extracting the maximum gain vertex takes constant time, however, inserting a vertex takes time linear to the size of the doubly-linked list. However, when using an array of gain-buckets, inserting a vertex takes constant time, but extracting the vertex with the maximum gain may sometimes take more than constant time.

In the implementation of the boundary KL algorithm, the method that is used to store the boundary vertices is also important. One possibility is to not store the boundary vertices anywhere, and simply determine them during each iteration of the boundary KL algorithm. Determining if a vertex is on the boundary is simple since, its external degree is greater than zero. However, in doing so, we make the complexity of the boundary KL algorithm to be in the order of the number of vertices, even when the boundary is very small. In our implementation, we use a hash-table to store the boundary vertices. A vertex is in the boundary if it is stored in the hash-table. The size of the hash-table is set to be twice the size of the boundary of the next level finer graph. During BKL, any vertices that move away from the boundary are removed from the hash-table, and any vertices that move to the boundary are inserted in the hash-table.