Multilevel $k$-way Partitioning Scheme for Irregular Graphs

George Karypis and Vipin Kumar

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

In this paper, we present and study a class of graph partitioning algorithms that reduces the size of the graph by collapsing vertices and edges, we find a $k$-way partitioning of the smaller graph, and then we uncoarsen and refine it to construct a $k$-way partitioning for the original graph. These algorithms compute a $k$-way partitioning of a graph $G = (V, E)$ in $O(|E|)$ time, which is faster by a factor of $O(\log k)$ than previously proposed multilevel recursive bisection algorithms. A key contribution of our work is in finding a high-quality and computationally inexpensive refinement algorithm that can improve upon an initial $k$-way partitioning. We also study the effectiveness of the overall scheme for a variety of coarsening schemes. We present experimental results on a large number of graphs arising in various domains including finite element methods, linear programming, VLSI, and transportation. Our experiments show that this new scheme produces partitions that are of comparable or better quality than those produced by the multilevel bisection algorithm and requires substantially smaller time. Graphs containing up to 450,000 vertices and 3,300,000 edges can be partitioned in 256 domains in less than 40 s on a workstation such as SGI’s Challenge. Compared with the widely used multilevel spectral bisection algorithm, our new algorithm is usually two orders of magnitude faster and produces partitions with substantially smaller edge-cut.

Key Words: graph partitioning; multilevel partitioning methods; Kernighan–Lin heuristic; spectral partitioning methods; parallel sparse matrix algorithms.

1This work was supported by NSF CCR-9423082, by the Army Research Office Contract DA/DAAH04-95-1-0538, by the IBM Partnership Award, and by the 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 DAAH04-95-C-0008, the content of which does not necessarily reflect the position or the policy of the government, and no official endorsement should be inferred. Access to computing facilities was provided by AHPCRC, Minnesota Supercomputer Institute, Cray Research Inc., and the Pittsburgh Supercomputing Center. Related papers are available via WWW at URL: http://www.cs.umn.edu/~karypis.

2E-mail: karypis@cs.umn.edu.

3E-mail: kumar@cs.umn.edu.
1. INTRODUCTION

The graph partitioning problem is to partition the vertices of a graph in \( p \) roughly equal partitions such that the number of edges connecting vertices in different partitions is minimized. This problem finds applications in many areas including parallel scientific computing, task scheduling, and VLSI design. Some examples are domain decomposition for minimum communication mapping in the parallel execution of sparse linear system solvers, mapping of spatially related data items in large geographical information systems on disk to minimize disk I/O requests, and mapping of task graphs to parallel processors. The graph partitioning problem is NP-complete. However, many algorithms have been developed that find reasonably good partitionings [2, 3, 5, 7–9, 12, 13, 16, 18–24].

The \( k \)-way partitioning problem is most frequently solved by recursive bisection. That is, we first obtain a 2-way partitioning of \( V \), and then we recursively obtain a 2-way partitioning of each resulting partition. After \( \log k \) phases, graph \( G \) is partitioned into \( k \) partitions. Thus, the problem of performing a \( k \)-way partitioning is reduced to that of performing a sequence of bisections. Recently [2, 12, 16], a multilevel recursive bisection (MLRB) algorithm has emerged as a highly effective method for computing a \( k \)-way partitioning of a graph. The basic structure of a multilevel bisection 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 partitioning is projected back to the original graph (finer graph) by periodically refining the partitioning. Since the finer graph has more degrees of freedom, such refinements decrease the edge-cut. The experiments presented in [16] show that compared to the state-of-the-art implementation of the well-known spectral bisection [1], MLRB produces partitionings that are significantly better and is an order of magnitude faster. The complexity of the MLRB for producing a \( k \)-way partitioning of a graph \( G = (V, E) \), is \( O(|E| \log k) \) [16].

The multilevel paradigm can also be used to construct a \( k \)-way partitioning of the graph directly as illustrated in Fig. 1. The graph is coarsened successively as before. But the coarsest graph is now directly partitioned into \( k \) parts, and this \( k \)-partitioning is refined successively as the graph is uncoarsened back into the original graph. There are a number of advantages of computing the \( k \)-way partitioning directly (rather than computing it successively via recursive bisection). First, the entire graph now needs to be coarsened only once, reducing the complexity of this phase to \( O(|E|) \) down from \( O(|E| \log k) \). Second, it is well known that recursive bisection can do arbitrarily worse than \( k \)-way partitioning [27]. Thus, a method that obtains a \( k \)-way partitioning directly can potentially produce much better partitionings. Note that the direct computation of a good \( k \)-way partitioning is harder than the computation of a good bisection (although both problems are NP-hard) in general. This is precisely why \( k \)-way partitioning is most commonly computed via recursive bisection. But in the context of multilevel schemes, we only need a rough \( k \)-way partitioning of the coarsest graph, as this can be potentially refined successively as the graph is uncoarsened. For example, a simple method for computing this initial partitioning in the multilevel context is simply to coarsen the graph down to \( k \) vertices. However, in the refinement phase, we need to refine a \( k \)-way partitioning, which is considerably more complicated than refining a bisection. In fact, a direct generalization of the KL refinement algorithm to \( k \)-way partitioning used in [10] is substantially more expensive than performing a KL refinement of a bisection [17].
FIG. 1. The various phases of the multilevel $k$-way partitioning algorithm. During the coarsening phase, the size of the graph is successively decreased; during the initial partitioning phase, a $k$-way partitioning of the smaller graph is computed (a 6-way partitioning in this example); and during the uncoarsening phase, the partitioning is successively refined as it is projected to the larger graphs.

Even for 8-way refinement, the run time is quite high for these schemes [11]. Computing $k$-way refinement for $k > 8$ is prohibitively expensive.

In this paper, we present a $k$-way partitioning algorithm. The run time of this $k$-way multilevel algorithm (ML$k$P) is linear to the number of edges, i.e., $O(|E|)$. A key contribution of our work is a simple and yet powerful scheme for refining a $k$-way partitioning in the multilevel context. This scheme is substantially faster than the direct generalization [11] of the KL bisection refinement algorithm, but it is equally effective in the multilevel context. Furthermore, this new $k$-way refinement algorithm is inherently parallel [15] (unlike the original KL refinement algorithm which is known to be inherently sequential in nature [6]), making it possible to develop high-quality parallel graph partitioning algorithms.

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 this new scheme produces partitionings that are of comparable or better quality than those produced by the state-of-the-art implementation of the ML$R$B algorithm [16], and it requires substantially smaller time. Graphs containing up to 450,000 vertices and 3,300,000 edges, can be partitioned in 256 partitions in less than 40 s on a workstation such as SGI’s Challenge. For many of these graphs, the process of graph partitioning takes even less time than the time to read the graph from the disk into memory. Compared with the widely used multilevel spectral bisection algorithm [12, 22, 23], our new algorithm is usually two orders of magnitude faster and produces partitionings with substantially smaller edge-cut. The run time of our $k$-way partitioning algorithm is comparable to the run time of a small number (2–4) runs of geometric recursive bisection algorithms [9, 18–20, 24]. Note that geometric algorithms are applicable only if coordinates of the
vertices are available and require tens of runs to produce cuts that are of quality similar to those produced by spectral bisection.

The remainder of the paper is organized as follows. Section 2 defines the graph partitioning problem and presents the basic concepts of multilevel $k$-way graph partitioning. Some of the material presented in this section on coarsening strategies is similar to that for multilevel recursive bisection [12, 16], but it is included here to make this paper self-contained. Section 3 presents an experimental evaluation of the various parameters of the multilevel graph partitioning algorithm and compares its performance with that of a multilevel recursive bisection algorithm.

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 $i \neq j$, $|V_i| = \lceil n/k \rceil$, and $\bigcup_i V_i = V$, and the number of edges of $E$ whose incident vertices belong to different subsets is minimized. A $k$-way partitioning of $V$ is commonly represented by a partitioning 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 to which vertex $v$ belongs. Given a partitioning $P$, the number of edges whose incident vertices belong to different partitions is called the edge-cut of the partitioning.

The basic structure of a multilevel $k$-way partitioning algorithm is very simple. The graph $G = (V, E)$ is first coarsened down to a small number of vertices, a $k$-way partitioning of this much smaller graph is computed, and then this partitioning is projected back toward the original graph (finer graph) by successively refining the partitioning at each intermediate level. This three-stage process of coarsening, initial partitioning, and refinement is graphically illustrated in Fig. 1.

Next we describe each of these phases in more detail.

2.1. Coarsening Phase

During the coarsening phase, a sequence of smaller graphs $G_i = (V_i, E_i)$, is constructed from the original graph $G_0 = (V_0, E_0)$ such that $|V_i| < |V_{i-1}|$. 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 $V^v_i$ be the set of vertices of $G_i$ combined to form vertex $v$ of $G_{i+1}$. In order for a partitioning 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 $V^v_i$. 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 $V^v_i$. In the case where more than one vertex of $V^v_i$ 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 coarsening method ensures the following properties [12]: (i) the edge-cut of the partitioning in a coarser graph is equal to the edge-cut of the same partition in the finer graph and (ii) a balanced partitioning of the coarser graph leads to a balanced partitioning of the finer graph.

This edge collapsing idea can be formally defined in terms of matchings [2, 12]. 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 is to decrease the size of the graph $G_i$, the matching should be maximal. A matching is called maximal matching if it is not possible to add any other edge to it without making two edges become incident on the same vertex. Note that depending on how matchings are computed, the size of the maximal matching may be different.

The coarsening phase ends when the coarsest graph $G_m$ has a small number of vertices or if the reduction in the size of successively coarser graphs becomes too small. In our experiments, for a $k$-way partition, we stop the coarsening process when the number of vertices becomes less than $ck$, where $c = 15$ in our experiments. The choice of this value of $c$ was to allow the initial partitioning algorithm to create $k$ partitions of roughly the same size. We also end the coarsening phase if the reduction in the size of successive graphs is less than a factor of 0.8.

In the remaining sections we describe three ways that we used to select maximal matchings for coarsening. Two of these matchings, RM [2, 12] and HEM [16], have been previously investigated in the context of MLRB.

**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 [2, 12, 16]. The random maximal matching algorithm works as follows. 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 partitioning 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

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

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 [13] 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 originally introduced in [16]. 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 in practice. The complexity of computing a heavy-edge matching is $O(|E|)$, which is asymptotically similar to that for computing the random matching.

**Modified Heavy Edge Matching (HEM*).** The analysis of the multilevel bisection algorithm in [13] shows that a good edge-cut of a coarser graph is closer to that of a good edge-cut of the original graph if the average degree of the coarser graph is small. The *modified heavy edge matching (HEM*)* is a modification of HEM that tries to decrease the average degree of coarser graphs.

A HEM* is computed using a randomized algorithm similar to that for computing a HEM. The vertices are again visited in random order. Let $v$ be such a vertex, and let $H$ be the set of unmatched adjacent vertices of $v$ that are connected to $v$ by an edge of maximum weight ($H$ can contain more than one vertex if some edges connected to $v$ have identical weights). For each vertex $u \in H$, let $W_{v-u}$ be the sum of the weights of the edges of $u$ that connect $u$ to vertices adjacent to $v$. In the HEM* scheme, $v$ is matched with the vertex $u \in H$, such that $W_{v-u}$ is maximized over all vertices in $H$.

As illustrated in Fig. 2, HEM* leads to fewer edges in the coarser graph and the average weight of edges in coarser graphs tend to be higher. Hence, in subsequent coarsening levels, the weight of the edges included in the matching increases, making HEM* more effective. HEM* is more effective than HEM in producing a good coarsening of the original graph $G_0$ when the edges of $G_0$ have identical weights. In fact, the first level coarser graph $G_1$ produced by HEM is similar to that produced by RM, since there are no heavy edge in $G_0$. In contrast, $G_1$ produced by HEM* will have smaller average degree because the vertices matched by HEM* will be adjacent to many common vertices. The complexity of computing HEM* is $O(|E|)$, which is asymptotically the same as that for computing the random matching and heavy edge matching. But the constant for HEM* is somewhat higher than that for HEM and RM.

**FIG. 2.** Example of the matchings produced by RM and HEM*.
2.2. Initial Partitioning Phase

The second phase of a multilevel k-way partitioning algorithm is to compute a k-way partitioning \( P_m \) of the coarse graph \( G_m = (V_m, E_m) \) such that each partition contains roughly \( |V_0|/k \) vertex weight of the original graph. Since during coarsening, the weights of the vertices and edges of the coarser graph were set to reflect the weights of the vertices and edges of the finer graph, \( G_m \) contains sufficient information to intelligently enforce the balanced partitioning and the minimum edge-cut requirements.

One way to produce the initial k-way partitioning is to keep coarsening the graph until it has only \( k \) vertices left. These coarse \( k \) vertices can serve as the initial k-way partitioning of the original graph. There are two problems with this approach. First, for many graphs, the reduction in the size of the graph in each coarsening step becomes very small after some coarsening steps, making it very expensive to continue with the coarsening process. Second, even if we are able to coarsen the graph down to only \( k \) vertices, the weights of these vertices are likely to be quite different, making the initial partitioning highly unbalanced.

In our algorithm, the k-way partitioning of \( G_m \) is computed using our multilevel bisection algorithm [16]. Our experience has shown that our multilevel recursive bisection algorithm produces good initial partitionings and requires a relatively small amount of time as long as the size of the original graph is sufficiently larger than \( k \).

2.3. Uncoarsening Phase

During the uncoarsening phase, the partitioning \( P_m \) 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 \( v \) of \( G_{i+1} \) contains a distinct subset of vertices \( V^v_i \) of \( G_i \), \( P_i \) is obtained from \( P_{i+1} \) by simply assigning the set of vertices \( V^v_i \) to the partitioning \( P_{i+1}[v] \); i.e., \( P_i[u] = P_{i+1}[v] \), \( \forall u \in V^v_i \).

Note that, even if the partitioning of \( G_j \) is at a local minima, \( ^4 \) the projected partitioning of \( G_{j-1} \) may not be at a local minima. Since \( G_{j-1} \) is finer, it has more degrees of freedom that can be used to further improve the partitioning and thus decrease the edge-cut. Hence, it may still be possible to improve the projected partitioning of \( G_{j-1} \) by local refinement heuristics.

A class of local refinement algorithms that tends to produce very good results are those based on the Kernighan–Lin (KL) partitioning algorithm [17] and their variants [4, 12]. The KL algorithm incrementally swaps vertices among partitions of a bisection to reduce the edge-cut of the partitioning until the partitioning reaches a local minima. One commonly used variation of the KL algorithm for bisection refinement is from Fiduccia–Mattheyses [4]. In particular, for each vertex \( v \), this variation of the KL algorithm computes the gain which is the reduction in the edge-cut achieved by moving \( v \) to the other partition. These vertices are inserted into two priority queues, one for each partition, according to their gains. Initially, all vertices are unlocked; i.e., they are free to move to the other partition. The algorithm iteratively selects an unlocked vertex \( v \) with the largest gain from one of the two priority queues and moves it to the other partition. When a vertex \( v \) is moved, it is locked and the gain of the vertices adjacent to

\(^4\)A partitioning is at a local minima, if movement of any vertex from one part to the other does not improve the edge-cut.
are updated. After each vertex movement, the algorithm also records the size of the cut achieved at this point. Note that the algorithm does not allow locked vertices to be moved since this may result in thrashing (i.e., repeated movement of the same vertex). A single pass of the algorithm ends when there are no more unlocked vertices (i.e., all the vertices have been moved). Then, the recorded cut-sizes are checked, the point where the minimum cut was achieved is selected, and all vertices that were moved after that point are moved back to their original partition. Now, this becomes the initial partitioning for the next pass of the algorithm. In the case of multilevel recursive bisection algorithms [2, 12, 16], KL refinement becomes very powerful, as the initial partitioning available at each successive coarsening level is already a good partition.

However, refining a k-way partitioning is significantly more complicated because vertices can move from a partition to many other partitions; thus, increasing the optimization space combinatorially. An extension of the KL refinement algorithm in the case of k-way refinement is described in [10]. This algorithm uses k(k − 1) priority queues, one for each type of move. In each step of the algorithm, the moves with the highest gain are found from each of these k(k − 1) queues, and the move with the highest gain that preserves or improves the balance is performed. After the move, all of the k(k − 1) priority queues are updated. The complexity of k-way refinement is significantly higher than that of 2-way refinement, and for a graph with m edges, this complexity is O(k * m). This approach is only practical for small values of k. Due to this high complexity, the multilevel recursive octasection algorithm described in [10], requires the same amount of time as multilevel recursive bisection, even though recursive octasection spends much less time for coarsening.

We have developed simple k-way refinement algorithms that are simplified versions of the k-way Kernighan–Lin refinement algorithm, and their complexity is independent of the number of partitions being refined. As the results in Section 3 show, despite the simplicity of our refinement algorithms, they produce high quality partitionings in a small amount of time. First, we describe some key concepts and definitions that are used in the description of our two k-way partitioning refinement algorithms.

Consider a graph $G_i = (V_i, E_i)$ and its partitioning vector $P_i$. For each vertex $v \in V_i$ we define the neighborhood $N(v)$ of $v$ to be the union of the partitions that the vertices adjacent to $v$ (i.e., $Adj(v)$) belong to; that is, $N(v) = \cup_{u \in Adj(v)} P_i[u]$. Note that if $v$ is an interior vertex of a partition, then $N(v) = \emptyset$. On the other hand, the cardinality of $N(v)$ can be as high as $Adj(v)$ for the case in which each vertex adjacent to $v$ belongs to a different partition. During refinement, $v$ can move to any of the partitions in $N(v)$. For each vertex $v$ we compute the gains of moving $v$ to one of its neighbor partitions. In particular, for every $b \in N(v)$ we compute $ED[v]_b$ as the sum of the weights of the edges $(v, u)$ such that $P_i[u] = b$. Also, we compute $ID[v]$ as the sum of the weights of the edges $(v, u)$ such that $P_i[u] = P_i[v]$. The quantity $ED[v]_b$ is called the external degree of $v$ to partition $b$, while the quantity $ID[v]$ is called the internal degree of $v$. Given these definitions, the gain of moving vertex $v$ to partition $b \in N(v)$ is $g[v]_b = ID[v]$. These definitions are illustrated in Fig. 3. For example, for vertex 5, $N[5] = \{0, 2\}$, $ID[5] = 2$, $ED[5]_0 = 2$, and $ED[5]_2 = 3$.

However, in addition to decreasing the edge-cut, moving a vertex from one partition to another must not create partitions whose size is unbalanced. In particular, our partitioning refinement algorithms move a vertex only if it satisfies the following **Balancing Condition**.
Let $W_i$ be a vector of $k$ elements, such that $W_i[a]$ is the weight of partition $a$ of graph $G_i$, and let $W^{\text{min}} = 0.9|V_0|/k$ and $W^{\text{max}} = C|V_0|/k$. A vertex $v$, whose weight is $w(v)$ can be moved from partition $a$ to partition $b$ only if

$$W_i[b] + w(v) \leq W^{\text{max}},$$

(2)

and

$$W_i[a] - w(v) \geq W^{\text{min}}$$

(3)

The first condition ensures that movement of a node into a partition does not make its weight higher than $W^{\text{max}}$. Note that by adjusting the value of $C$, we can vary the degree of imbalance among partitions. If $C = 1$, then the refinement algorithm tries to make each partition of equal weight. In our experiments we found that letting $C$ be greater than 1.0, tends to improve the quality of the partitionings. However, in order to minimize the load imbalance, we used $C = 1.03$; that puts an upper bound of 3% on load imbalance. Note that the second condition is not critical for load balance, but it ensures that there is no partition with too few vertices.

**Greedy Refinement (GR).** The lookahead in KL algorithm serves a very important purpose. It allows movement of an entire cluster of vertices across a partition boundary. Note that it is quite possible that as the cluster is moved across the partition boundary, the edge-cut increases, but after the entire cluster of vertices moves across the partition, then the overall edge-cut comes down. In the context of multilevel schemes, this lookahead becomes less important. The reason is that these clusters of vertices are coarsened into a single vertex at successive coarsening phases. Hence, movement of a vertex at a coarse level actually corresponds to the movement of a group of vertices in the original graph.

If the lookahead part of KL is eliminated (i.e., if vertices are moved only if they lead to positive gain), then it becomes less useful to maintain a priority queue. In particular, vertices whose move results in a large positive gain will be moved anyway even if they are not moved earlier in the priority order. Hence, a variation of KL that simply visits the boundary vertices in a random order and moves them if they result in a positive gain
is likely to work well in the multilevel context. Our greedy refinement algorithm is based on this observation. It consists of a number of iterations. In each iteration, all the vertices are checked to see if they can be moved so that either the edge-cut of the partitioning can be decreased (while preserving balance), or the balance is improved.

In particular, GR works as follows. Consider a graph $G_i = (V_i, E_i)$, and its partitioning vector $P_i$. The vertices are checked in a random order. Let $v$ be such a vertex, let $P_i[v] = a$ be the partition that $v$ belongs to. If $v$ is a node internal to partition $a$ then $N(v) = \emptyset$, and $v$ is not moved. If $v$ is at the boundary of the partition, then $N(v)$ is nonempty. Let $N'(v)$ be the subset of $N(v)$ that contains all partitions $b$ such that movement of vertex $v$ to partition $b$ does not violate the Balancing Condition. Now vertex $v$ is moved to one of the adjacent partitions $b$, if either one of the following conditions is satisfied:

1. $ED[v]_b > ID[v]$ and $ED[v]_b$ is maximum among all $b \in N'(v)$.
2. $ED[v]_b = ID[v]$ and $W_i[a] - W_i[b] > w(v)$.

That is, the GR algorithm moves $v$ to a partition that leads to the largest reduction in the edge-cut without violating the balance condition. If no reduction in the edge-cut is possible, by moving $v$, then $v$ is moved to the partition (if any) that leads to no increase in the edge-cut but improves the balance. After moving vertex $v$, the algorithm updates the internal and external degrees of the vertices adjacent to $v$ to reflect the change in the partition.

The GR algorithm converges after a small number of iterations. In our experiments, we found that for most graphs, and with the HEM (or HEM*) matching scheme in particular, GR converged within four to eight iterations.

Global Kernighan–Lin Refinement (GKLR). As discussed in the previous section, the GR algorithm lacks any capabilities of climbing out of local minima. Our second refinement heuristic called global Kernighan–Lin, is somewhat more powerful and is closer to the original KL algorithm in spirit. It adds some limited hill-climbing capabilities to the GR algorithm and also uses a priority queue to determine the sequence of vertex moves.

The GKLR algorithm uses a global priority queue that stores the vertices according to their gains. Initially, all the vertices are scanned, and those whose sum of external degrees$^5$ is greater or equal to their internal degrees are inserted into the priority queue. In particular, let $v$ be such a vertex, let $N(v)$ be the neighborhood of $v$, and $b \in N(v)$ such that $ED[v]_b$ is maximum over the external degrees of partitions in $N(v)$. We insert $v$ into the priority queue with a gain equal to $ED[v]_b - ID[v]$.

The algorithm then proceeds and selects the vertex from the priority queue with the highest gain. Having selected such a vertex $v$, the algorithm selects a part $b \in N(v)$ to move $v$ such that $ED[v]_b$ is maximized while satisfying the balance condition (Eqs. (2) and (3)). Note that these swaps may lead to an increase in the edge-cut, since vertices are moved even if they have a negative gain value. The GKLR algorithm continues moving vertices until it has performed $x$ vertex moves that have not decreased the overall edge-

---

5We used this heuristic to select the vertices that are inserted in the priority queue as a compromise between inserting all the boundary vertices and inserting only the vertices that lead to a reduction in the edge-cut when moved to one of their neighboring partitions. If all the boundary vertices were inserted, then the cost would have been higher. On the other hand, if only the edge-cut reducing vertices were inserted, the hill-climbing capabilities of the algorithm would have been reduced.
In that case, the last \( x \) moves are undone. Once a vertex is moved, it is not considered for movement in the same iteration. This is repeated for a small number of iterations or until convergence.

Note that in each step, the vertices selected for movement by the GKLR algorithm and by the generalized KL of [11] may be quite different. GKLR selects a vertex \( v \) that has a move (among all possible moves to neighboring partitions \( N(v) \)) with the highest gain \( g[v]_{\text{max}} \). However, depending on the weight of the partitions, this move may never take place, and instead \( v \) can be moved to a partition \( a \in N(v) \) that leads to a smaller gain \( g[a]_v \). However, there may be another vertex \( u \) on the priority queue that has a move with the highest gain \( g[u]_{\text{max}} \) that may be permissible. Now if \( g[v]_v < g[u]_{\text{max}} < g[v]_{\text{max}} \), the generalization of the KL algorithm will select to move vertex \( u \) before considering vertex \( v \). Thus, in each step, GKLR does not necessarily select the vertex with the largest realizable gain. Furthermore, since the single priority queue contains only vertices whose sum of the external degrees in greater or equal to the internal degree, GKLR has less powerful hill-climbing capabilities than the generalized KL [11] that uses multiple priority queues and considers all the vertices.

3. EXPERIMENTAL RESULTS

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 graphs are described in Table 1. These graphs are classified into six groups. The first group contains graphs that correspond to finite element meshes, the second group contains graphs that correspond to coefficient matrices (i.e., assembled matrices) with multiple degrees of freedom and linear basis functions, the third group corresponds to assembled matrices with nonlinear basis functions, the fourth group corresponds to graphs that represent highway networks, the fifth group corresponds to graphs arising in linear programming applications, and the sixth group corresponds to graphs that represent VLSI circuits. For each of the first two groups, we have a large number of graphs, but for the last four groups, we have only a few graphs per group. So observed trends for the first two groups are more reliable than those for the last four groups.

All the experiments were performed on an SGI Challenge with 1.2 GBytes of memory and a 200 MHz MIPS R4400 processor. All times reported are in seconds. Since the nature of the multilevel algorithm discussed is randomized, we performed all experiments with fixed seed.

3.1. Matching Schemes

We implemented the three matching schemes described in Section 2.1. These schemes are (a) random matching (RM), (b) heavy edge matching (HEM), and (c) modified heavy edge matching (HEM*). For all the experiments, we used the GR refinement policy during the uncoarsening phase. The results for 32-way and 256-way partitioning are shown in Figs. 4 and 5 for all the graphs in Table 1.

In Fig. 4 we see that both HEM and HEM* consistently produce partitionings whose edge-cut is better than that of the partitionings produced by RM. For some groups of graphs, HEM and HEM* produce partitionings whose edge-cut is better than that of RM by up to 35%. The reason for the poor performance of RM becomes clear in Table 2 that
TABLE 1
Various Graphs Used in Evaluating the Multilevel Graph Partitioning and Sparse Matrix Ordering Algorithm

<table>
<thead>
<tr>
<th>Matrix name</th>
<th>Number of vertices</th>
<th>Number of edges</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>144</td>
<td>144649</td>
<td>1074393</td>
<td>3D Finite element mes (Parafoil)</td>
</tr>
<tr>
<td>598A</td>
<td>110971</td>
<td>741934</td>
<td>3D Finite element mesh (Submarine I)</td>
</tr>
<tr>
<td>AUTO</td>
<td>448695</td>
<td>3314611</td>
<td>3D Finite element mesh (GM Saturn)</td>
</tr>
<tr>
<td>BRACK2</td>
<td>62631</td>
<td>366559</td>
<td>3D Finite element mesh (Bracket)</td>
</tr>
<tr>
<td>COPTER2</td>
<td>55476</td>
<td>352238</td>
<td>3D Finite element mesh (Helicopter blade)</td>
</tr>
<tr>
<td>FLAP</td>
<td>51537</td>
<td>479620</td>
<td>3D Finite element mesh</td>
</tr>
<tr>
<td>M14B</td>
<td>214765</td>
<td>3358036</td>
<td>3D Finite element mesh (Submarine II)</td>
</tr>
<tr>
<td>ROTOR</td>
<td>99617</td>
<td>662431</td>
<td>3D Finite element mesh</td>
</tr>
<tr>
<td>TORSO</td>
<td>201142</td>
<td>1479989</td>
<td>3D Finite element mesh (Human torso)</td>
</tr>
<tr>
<td>WAVE</td>
<td>156317</td>
<td>1059331</td>
<td>3D Finite element mesh</td>
</tr>
<tr>
<td>BCSSTK31</td>
<td>35588</td>
<td>572914</td>
<td>3D Stiffness matrix</td>
</tr>
<tr>
<td>BCSSTK32</td>
<td>44609</td>
<td>985046</td>
<td>3D Stiffness matrix</td>
</tr>
<tr>
<td>CANT</td>
<td>54195</td>
<td>1960797</td>
<td>3D Stiffness matrix</td>
</tr>
<tr>
<td>CYLINDER93</td>
<td>45594</td>
<td>1786726</td>
<td>3D Stiffness matrix</td>
</tr>
<tr>
<td>INPRO1</td>
<td>46949</td>
<td>1117809</td>
<td>3D Stiffness matrix</td>
</tr>
<tr>
<td>SHELL93</td>
<td>181200</td>
<td>2313765</td>
<td>3D Stiffness matrix</td>
</tr>
<tr>
<td>SHYY161</td>
<td>76480</td>
<td>152002</td>
<td>CFD/Navier-Stokes</td>
</tr>
<tr>
<td>TROLL</td>
<td>213453</td>
<td>5885829</td>
<td>3D Stiffness matrix</td>
</tr>
<tr>
<td>VENKAT25</td>
<td>62424</td>
<td>827684</td>
<td>2D Coefficient matrix</td>
</tr>
<tr>
<td>BBMAT</td>
<td>38744</td>
<td>993481</td>
<td>2D Stiffness matrix</td>
</tr>
<tr>
<td>MAP1</td>
<td>267241</td>
<td>334931</td>
<td>Highway network</td>
</tr>
<tr>
<td>MAP2</td>
<td>78489</td>
<td>98995</td>
<td>Highway network</td>
</tr>
<tr>
<td>FINAN512</td>
<td>74752</td>
<td>261120</td>
<td>Linear programming</td>
</tr>
<tr>
<td>KEN-11</td>
<td>14694</td>
<td>33880</td>
<td>Linear programming</td>
</tr>
<tr>
<td>S38584.1</td>
<td>22143</td>
<td>35608</td>
<td>Sequential circuit</td>
</tr>
</tbody>
</table>

contains the size of the edge-cut of the initial \( k \)-way partitioning. For all graphs, the size of the initial edge-cut on the coarsest graph is significantly worse for RM compared with HEM and HEM*. Note that the difference in the size of the initial edge-cut on the coarsest graph is much greater for the three schemes than those shown in Fig. 4. For example, for the first two groups of graphs, the overall quality of RM, HEM, and HEM* is similar, but the edge-cut of the \( k \)-way partitioning in the coarsest graph obtained by HEM and HEM*
FIG. 4. Quality of the partitionings of HEM and HEM* relative to RM matching. For each graph, the ratio of the edge-cut of the HEM and HEM* matching schemes to that of the RM matching scheme is plotted for 32- and 256-way partitionings. Bars under the baseline indicate that the corresponding matching scheme performs better than RM.
FIG. 5. Run time of partitioning using HEM and HEM* relative to RM matching. For each graph, the ratio of the time required by the HEM and HEM* matching schemes to that of the RM matching scheme is plotted for 32- and 256-way partitionings. Bars under the baseline indicate that the corresponding matching scheme is faster than RM.
are 30 to 65% smaller than those obtained by RM (as shown in Table 2). (As a result, for RM, \( k \)-way refinement takes more time compared with HEM and HEM\(^*\).) As discussed in [13], the effectiveness of a coarsening scheme depends on how successful it is in removing a significant amount of edge-weight from the successive coarser graphs. According to this criterion, HEM and HEM\(^*\) are strictly better coarsening schemes than RM because they remove more edge-weight from the graph.

Comparing HEM against HEM\(^*\), we see that for most graphs, their performance is comparable. The only notable exception is BBMAT for which HEM\(^*\) does up to 10% better than HEM. BBMAT is the type of graph in which applying RM at the finest graph \((G_0)\) significantly increases the average degree of the first level coarser graph \((G_1)\). Note that HEM and RM compute the same first level coarse graph \(G_1\), since the weights of all edges in \(G_0\) is the same. Hence, for BBMAT the average degree of \(G_1\) obtained by HEM is much higher than that obtained using HEM\(^*\). For other types of graphs, particularly those that correspond to finite element meshes, RM increases the average degree only slightly in going from \(G_0\) to \(G_1\), which in turn allows HEM to perform good coarsening.

As a result, for BBMAT, the initial partitioning found by HEM is much worse than that found by HEM\(^*\). This can be seen in Table 2. Note that the initial edge-cuts for HEM and HEM\(^*\) are similar for all problems except BBMAT.

In Fig. 5 we see that for 32-way partition, HEM is up to 20% faster than RM, while HEM\(^*\) is up to 41% slower than RM. HEM is faster than RM because it requires much less refinement, and the coarsening step of HEM is only slightly slower than the coarsening step in RM. HEM\(^*\) is slower than RM because coarsening using HEM\(^*\) is much slower than coarsening using RM. For a 256-way partition, HEM is again faster than RM (quite consistently), but now for 7 graphs HEM\(^*\) is faster than RM. This is because RM requires substantially more refinement time and because the coarsest graph \(G_m\) produced by RM has many more edges than that produced by HEM\(^*\), increasing the initial partitioning time.

### Table 2

<table>
<thead>
<tr>
<th>Graph</th>
<th>64EC</th>
<th>256EC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RM</td>
<td>HEM</td>
</tr>
<tr>
<td></td>
<td>RM</td>
<td>HEM</td>
</tr>
<tr>
<td>144</td>
<td>200855</td>
<td>142464</td>
</tr>
<tr>
<td>AUTO</td>
<td>525526</td>
<td>343154</td>
</tr>
<tr>
<td>FLAP</td>
<td>58034</td>
<td>42810</td>
</tr>
<tr>
<td>BCSSTK32</td>
<td>221234</td>
<td>155286</td>
</tr>
<tr>
<td>INPRO1</td>
<td>244035</td>
<td>159632</td>
</tr>
<tr>
<td>BBMAT</td>
<td>324794</td>
<td>154878</td>
</tr>
<tr>
<td>MAP2</td>
<td>1064</td>
<td>911</td>
</tr>
<tr>
<td>KEN-11</td>
<td>16273</td>
<td>15677</td>
</tr>
</tbody>
</table>

*Note.* 64EC and 256EC are the edge-cuts of 64- and 256-way partitionings.
As the experiments show, for most of the graphs, HEM is an excellent matching scheme that produces good partitionings, and requires the smallest overall run time. However, for a certain class of graphs, HEM* does better than HEM.

3.2. Refinement Policies

As described in Section 2.3, there are different ways that a partitioning can be refined during the uncoarsening phase. We evaluated the performance of two refinement policies, both in terms of the quality of the partitionings they produce and also how much time they require. The refinement policies that we evaluate are greedy refinement (GR) and global Kernighan–Lin refinement (GKLR).

The results of these refinement policies for computing a 32- and 256-way partition for the graphs in Table 1 are shown in Figs. 6–9. Figures 6 and 7 show the edge-cut of the partitionings produced by GKLR relative to those produced by GR for the three different coarsening schemes, while Figs. 8 and 9 show the amount of time required by GKLR relative to GR for computing these partitionings.

A number of observations can be made from Figs. 6 and 7. GKLR is significantly better than GR only for BBMAT. For other problems, the difference is minor. If RM coarsening is used, then GKLR does better than GR more consistently. If HEM or HEM* coarsening is used, then GKLR performs quite similar to GR for all problems. Even for BBMAT, the gap between the performance of GKLR and GR is narrower for HEM and HEM* compared with RM. If we combine the 32- and 256-way partitionings as a set of 150 different runs, GKLR produces better partitionings for 31 out of these 150 runs. Out of these 31 runs, 14 were obtained using RM, 7 using HEM, and 10 using HEM*.

Another interesting observation is that for most graphs the difference in the quality of the partitionings produced by GR and GKLR is very small. The difference in the quality is less than 2% for 139 out of the 150 different runs. The only notable exceptions are KEN-11 for which GR does up to 7% better than GKLR and BBMAT for which GKLR does up to 21% better than GR. From these experimental results, it is clear that a simple refinement scheme such as GR is quite adequate, particularly if the initial partitioning for the coarsest graph is quite good. The additional power of GKLR is useful only when it is used in conjunction with the RM matching scheme which leads to poor initial partitionings.

From Figs. 8 and 9 we see that the amount of time required for a 32- and 256-way partitioning using GKLR is significantly higher than the time required using GR. GKLR requires more time for each of the 150 different runs. In some cases, GKLR requires more than twice the time required by GR. Comparing the different matching schemes, we see that the relative increase in the run time is higher for RM than for HEM and HEM*. This is not surprising since RM requires more refinement and also RM benefits the most from GKLR.

In summary, GR and GKLR tend to produce partitionings that have similar edge-cuts, but with GKLR requiring significantly more time than GR.

3.3. Comparison with Other Partitioning Schemes

Figure 10 shows the relative quality of our multilevel $k$-way partitioning algorithm (MLkP) compared to the multilevel recursive bisection algorithm (MLRB) described in [16] (implemented in METIS [14]). METIS is a set of programs for partitioning unstructured
FIG. 6. Quality of GKLR refinement scheme for 32-way partitioning for RM, HEM, and HEM* coarsening schemes relative to GR refinement scheme. For each graph, the ratio of the edge-cut of the GKLR refinement algorithm to that of the GR algorithm scheme is plotted for RM, HEM, and HEM* matching schemes. Bars under the baseline indicate that GKLR performs better than GR for the corresponding matching scheme.
FIG. 7. Quality of GKLR refinement scheme for 256-way partitioning for RM, HEM, and HEM* coarsening schemes relative to GR refinement scheme. For each graph, the ratio of the edge-cut of the GKLR refinement algorithm to that of the GR algorithm scheme is plotted for RM, HEM, and HEM* matching schemes. Bars under the baseline indicate that GKLR performs better than GR for the corresponding matching scheme.
FIG. 8. Run time for the 32-way partitionings produced by the GR and GKLR refinement algorithms for RM, HEM, and HEM* coarsening schemes. For each graph, the ratio of the time required for partitioning using the GKLR refinement algorithm to that of the GR algorithm scheme is plotted for RM, HEM, and HEM* matching schemes. Bars under the baseline indicate that GKLR is faster than GR for the corresponding matching scheme.
FIG. 9. Run time for the 256-way partitionings produced by the GR and GKLR refinement algorithms for RM, HEM, and HEM* coarsening schemes. For each graph, the ratio of the time required for partitioning using the GKLR refinement algorithm to that of the GR algorithm scheme is plotted for RM, HEM, and HEM* matching schemes. Bars under the baseline indicate that GKLR is faster than GR for the corresponding matching scheme.
FIG. 10. Quality of the partitionings produced by MLkP relative to MLRB. The multilevel $k$-way partitioning algorithm uses HEM during coarsening and GR during refinement. For each graph, the ratio of the edge-cut of the $k$-way partitioning algorithm to that of the recursive bisection algorithm is plotted for 32-, 64-, and 256-way partitionings. Bars under the baseline indicate that $k$-way partitioning performs better than recursive bisection.
graphs and for ordering sparse matrices that implements various algorithms described in [16]. For each graph, we plot the ratio of the edge-cut of the MLkP algorithm to the edge-cut of the MLRB algorithm (the actual edge-cuts are shown in Table 3). Ratios that are less than one indicate that MLkP produces better partitionings than MLRB. For

### TABLE 3

The Edge-Cuts Produced by the Multilevel Recursive Bisection, Multilevel Recursive Bisection, and Multilevel k-way Partitioning

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Multilevel spectral bisection</th>
<th>Multilevel recursive bisection</th>
<th>Multilevel k-way partition</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>64EC 128EC 256EC</td>
<td>64EC 128EC 256EC</td>
<td>64EC 128EC 256EC</td>
</tr>
<tr>
<td>144</td>
<td>96538 132761 184200</td>
<td>88086 120611 161563</td>
<td>87750 118112 156145</td>
</tr>
<tr>
<td>598A</td>
<td>68107 95220 128619</td>
<td>64443 89298 119699</td>
<td>63262 86909 114846</td>
</tr>
<tr>
<td>AUTO</td>
<td>208729 291638 390056</td>
<td>194436 269638 362858</td>
<td>193092 263228 349137</td>
</tr>
<tr>
<td>BRACK2</td>
<td>34464 49917 69243</td>
<td>29983 42625 60608</td>
<td>29742 42170 59847</td>
</tr>
<tr>
<td>COPTER2</td>
<td>47862 64601 84934</td>
<td>43721 58809 77155</td>
<td>42411 56100 73946</td>
</tr>
<tr>
<td>FLAP</td>
<td>35540 54407 80392</td>
<td>30741 49806 74626</td>
<td>30461 49203 73641</td>
</tr>
<tr>
<td>M14B</td>
<td>124749 172780 232949</td>
<td>111104 156417 214203</td>
<td>109013 150331 206129</td>
</tr>
<tr>
<td>ROTOR</td>
<td>63251 88048 120989</td>
<td>53228 75010 103895</td>
<td>52069 73841 101732</td>
</tr>
<tr>
<td>TORSO</td>
<td>413501 473397 522717</td>
<td>117997 160788 218155</td>
<td>112797 155087 209895</td>
</tr>
<tr>
<td>WAVE</td>
<td>106858 142060 187192</td>
<td>97978 129785 171101</td>
<td>94251 124377 164187</td>
</tr>
<tr>
<td>BCSSTK31</td>
<td>86244 123450 176074</td>
<td>65249 97819 140818</td>
<td>66039 100713 143749</td>
</tr>
<tr>
<td>BCSSTK32</td>
<td>130984 185977 259902</td>
<td>106440 152081 222789</td>
<td>106661 160651 223545</td>
</tr>
<tr>
<td>CANT</td>
<td>459412 598870 798866</td>
<td>442398 574853 778928</td>
<td>428754 567478 756061</td>
</tr>
<tr>
<td>CYLINDER93</td>
<td>290194 431551 594859</td>
<td>289639 416190 590065</td>
<td>284012 409445 582015</td>
</tr>
<tr>
<td>INPRO1</td>
<td>125285 185838 264049</td>
<td>116748 171974 250207</td>
<td>118176 172592 251628</td>
</tr>
<tr>
<td>SHELL93</td>
<td>178266 238098 318535</td>
<td>124836 185323 269539</td>
<td>123437 181203 261296</td>
</tr>
<tr>
<td>SHYY161</td>
<td>6641 9151 11969</td>
<td>4365 6317 9092</td>
<td>4607 6591 9251</td>
</tr>
<tr>
<td>TROLL</td>
<td>529158 706605 947564</td>
<td>453812 638074 864287</td>
<td>445215 630918 846822</td>
</tr>
<tr>
<td>VENKAT25</td>
<td>50184 77810 116211</td>
<td>47514 73735 110312</td>
<td>49137 74470 111249</td>
</tr>
<tr>
<td>BBMAT</td>
<td>179282 250535 348124</td>
<td>55753 92750 132387</td>
<td>62018 109495 158990</td>
</tr>
<tr>
<td>MAP1</td>
<td>3546 6314 8933</td>
<td>1388 2221 3389</td>
<td>1122 1892 3108</td>
</tr>
<tr>
<td>MAP2</td>
<td>1759 2454 3708</td>
<td>828 1328 2157</td>
<td>726 1213 1984</td>
</tr>
<tr>
<td>FINAN512</td>
<td>15360 27575 53387</td>
<td>11388 22136 40201</td>
<td>11853 23365 42589</td>
</tr>
<tr>
<td>KEN-11</td>
<td>20931 23308 25159</td>
<td>14257 16515 18101</td>
<td>12360 13563 15836</td>
</tr>
<tr>
<td>S38584.1</td>
<td>5381 7595 9609</td>
<td>2428 3996 5906</td>
<td>2362 3869 5715</td>
</tr>
</tbody>
</table>

*Note.* 64EC, 128EC, and 256EC are the edge-cuts of 64-, 128-, and 256-way partitionings, respectively.
FIG. 11. Run time of MLkP relative to MLRB for 256-way partitioning. The multilevel k-way partitioning algorithm uses HEM during coarsening and GR during refinement. For each graph, the ratio of the run time of recursive bisection algorithm to that of the k-way partitioning algorithm is plotted for 256-way partitionings. Bars above the baseline indicate that k-way partitioning is faster than recursive bisection.
MULTILEVEL $k$-WAY PARTITIONING SCHEME

From this comparison and for the rest of the comparisons in this section, the ML$k$P algorithm uses HEM during coarsening and GR during refinement.

From this figure, we see that for almost all problems, ML$k$P and MLRB produce partitionings of similar quality. In particular, for the two highway networks (MAP1 and MAP2), ML$k$P produces up to 19% smaller edge-cuts than MLRB. For the graphs that correspond to finite element meshes (144, 598A, AUTO, BRACK2, COPTER2, M14B, ROTOR, TORSO, and WAVE), ML$k$P does slightly (up to 5%) and consistently better than MLRB. For the graphs that correspond to coefficient matrices of finite element applications with multiple degrees of freedom (BCSSTK31, BCSSTK32, CANT, CYLINDER93, FLAP, INPRO1, SHELL93, SHY161, TROLL, and VENKAT25), ML$k$P and MLRB perform quite similarly (within 6% of each other). The only problem for which ML$k$P performs significantly worse than MLRB is BBMAT, for which ML$k$P performs up to 20% worse than MLRB. As discussed in Section 2.1, these graphs correspond to assembled matrices with nonlinear basis functions, and the HEM coarsening scheme does not lead to good coarsenings. However, for this graph, both HEM* coarsening and GKLR refinement perform substantially better than HEM and GR, respectively. In particular, if we use HEM* for coarsening and GKLR for refinement, then the edge-cut for 128-way partitioning produced by ML$k$P is better by 2% than that of MLRB. In summary, for a large class of graphs, ML$k$P produces partitionings that are equally good or even better than those produced by the MLRB algorithm. Furthermore, the combination of HEM and GR seems quite adequate for most problems. However, for some problems HEM* and GKLR may be better choices for coarsening and refinement, respectively.

### TABLE 4

The Time Required to Find a 256–Way Partitioning by the Multilevel Spectral Bisection, Multilevel Recursive Bisection, and Multilevel $k$-way Partition (All Times Are in Seconds)

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Multilevel spectral bisection</th>
<th>Multilevel recursive bisection</th>
<th>Multilevel $k$-way partition</th>
<th>Matrix</th>
<th>Multilevel spectral bisection</th>
<th>Multilevel recursive bisection</th>
<th>Multilevel $k$-way partition</th>
</tr>
</thead>
<tbody>
<tr>
<td>144</td>
<td>607.27</td>
<td>48.14</td>
<td>13.40</td>
<td>CYLINDER93</td>
<td>671.33</td>
<td>39.10</td>
<td>13.07</td>
</tr>
<tr>
<td>598A</td>
<td>420.12</td>
<td>35.05</td>
<td>9.92</td>
<td>INPRO1</td>
<td>341.88</td>
<td>24.60</td>
<td>7.88</td>
</tr>
<tr>
<td>AUTO</td>
<td>2214.24</td>
<td>179.15</td>
<td>39.67</td>
<td>SHELL93</td>
<td>1111.96</td>
<td>71.59</td>
<td>17.40</td>
</tr>
<tr>
<td>BRACK2</td>
<td>218.36</td>
<td>16.52</td>
<td>5.65</td>
<td>SHY161</td>
<td>129.99</td>
<td>10.13</td>
<td>3.42</td>
</tr>
<tr>
<td>COPTER2</td>
<td>185.39</td>
<td>16.11</td>
<td>5.71</td>
<td>TROLL</td>
<td>3063.28</td>
<td>132.08</td>
<td>29.08</td>
</tr>
<tr>
<td>FLAP</td>
<td>279.67</td>
<td>16.50</td>
<td>5.21</td>
<td>VENKAT25</td>
<td>254.52</td>
<td>20.81</td>
<td>5.54</td>
</tr>
<tr>
<td>M14B</td>
<td>970.58</td>
<td>74.04</td>
<td>18.30</td>
<td>BBMAT</td>
<td>474.23</td>
<td>22.51</td>
<td>10.37</td>
</tr>
<tr>
<td>ROTOR</td>
<td>550.35</td>
<td>29.46</td>
<td>8.71</td>
<td>MAP1</td>
<td>850.16</td>
<td>44.80</td>
<td>8.12</td>
</tr>
<tr>
<td>TORSO</td>
<td>1053.37</td>
<td>63.93</td>
<td>17.13</td>
<td>MAP2</td>
<td>195.09</td>
<td>11.76</td>
<td>3.07</td>
</tr>
<tr>
<td>WAVE</td>
<td>658.13</td>
<td>44.55</td>
<td>12.94</td>
<td>FINAN512</td>
<td>311.01</td>
<td>17.98</td>
<td>6.49</td>
</tr>
<tr>
<td>BCSSTK31</td>
<td>309.06</td>
<td>15.21</td>
<td>5.53</td>
<td>KEN-11</td>
<td>121.94</td>
<td>4.09</td>
<td>3.13</td>
</tr>
<tr>
<td>BCSSTK32</td>
<td>474.64</td>
<td>22.50</td>
<td>7.39</td>
<td>S38584.1</td>
<td>178.11</td>
<td>4.72</td>
<td>2.55</td>
</tr>
<tr>
<td>CANT</td>
<td>978.48</td>
<td>47.70</td>
<td>17.44</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
FIG. 12. Quality of MLkP relative to multilevel spectral bisection. For each graph, the ratio of the edge-cut of the $k$-way partitioning algorithm to that of the recursive bisection algorithm is plotted for 32-, 64-, 128-, and 256-way partitionings. Bars under the baseline indicate that $k$-way partitioning performs better than multilevel spectral bisection.
FIG. 13. Run time of MLkp relative to spectral bisection for 256-way partitioning. For each graph, the ratio of the run time of multilevel spectral bisection algorithm to that of the $k$-way partitioning algorithm is plotted for 256-way partitionings. Bars above the baseline indicate that $k$-way partitioning is faster than multilevel spectral bisection.
FIG. 14. Quality of the partitionings produced by MLkP relative to Chaco’s multilevel recursive octasection algorithm. The MLkP algorithm uses HEM during coarsening and GR during refinement. For each graph, the ratio of the edge-cut of the MLkP algorithm to that of Chaco’s recursive octasection algorithm is plotted for 8- and 64-way partitionings. Bars under the baseline indicate that MLkP performs better than Chaco’s recursive octasection.
Figure 11 shows the amount of time required by the MLRB algorithm relative to the
time required by the MLkP algorithm for 256-way partitionings. From this graph we see
that MLkP is usually two to four times faster than MLRB. In particular, for moderate
size problems, MLkP is over three times faster, while for the larger problems, MLkP
is over four times faster. The actual run times for a 256-way partitioning is shown in
Table 4. From this table we see that even the larger problem (448,000 vertex mesh of
GM’s Saturn car) is partitioned in under 40 s.

Figures 12 and 13 present the relative quality and run time, respectively, of MLkP with
respect to multilevel spectral bisection (MSB) [1]. From these figures we see that for all
the graphs, MLkP produces better partitionings than MSB. In some cases MLkP produces
partitionings that cut over 70% fewer edges than those cut by the MSB. Furthermore,
from Fig. 13 we see that MLkP is up to two orders of magnitude faster than the MSB.

The graph partitioning package Chaco 2.0 [11, 12] also implements multilevel
quadrisection and octasection partitioning algorithms. Chaco uses random matching
during coarsening and spectral quadrisection and octasection methods to directly divide
the coarsest graph into four and eight parts, respectively6 [10]. The key difference
between our scheme and the one implemented in Chaco’s recursive octasection is that
their Kernighan–Lin refinement algorithm is direct generalization of the 2-way refinement
algorithm to handle both 4- and 8-way refinement. For example, in the case of 8-way
refinement, their algorithm uses $8 \times 7$ priority queues for all the different types of moves.
This algorithm is significantly slower than either the greedy or global Kernighan–Lin
refinement algorithms used by our multilevel $k$-way partition. In fact, Chaco’s recursive
octasection is not any faster than its recursive bisection. Furthermore, Chaco’s recursive
octasection is even more expensive to generalize beyond 8-way refinement.

Figure 14 shows the relative performance of our MLkP algorithm compared to Chaco’s
multilevel recursive octasection for 8- and 64-way partitionings. Note that for 8-way
partition, no recursive partitioning is performed by Chaco, while for a 64-way partition,
only one level of recursion is performed. From this figure we can see that for both 8- and
64-way partitioning, MLkP produces partitionings that are in general better than those
produced by Chaco’s recursive octasection. For some graphs, MLkP cuts up to 70% fewer
edges than Chaco does. The difference in quality is due to the following two reasons. First,
Chaco’s recursive octasection algorithm uses RM matching during coarsening, which
leads to successive coarser graphs with higher edge-weight. Second, the initial partitioning
obtained by spectral octasection is worse (cuts more edges) than the initial partitioning
obtained by MLRB. Thus, even though Chaco’s recursive octasection algorithm uses the
generalized KL refinement algorithm, it does not seem to be able to gain the losses due
to coarsening and initial partitioning. Figure 15 shows the relative run time of Chaco’s
multilevel recursive octasection compared to our multilevel $k$-way partitioning algorithm.
From this figure we see that our algorithm is considerably faster. MLkP computes an
8-way partitioning about 2 to 6 times faster than Chaco, and a 64-way partitioning about
4 to 14 times faster. In summary, for most graphs, MLkP produces better or comparable
partitionings than Chaco’s multilevel recursive octasection in significantly less time. This
indicates that for most graphs, greedy refinement coupled with the HEM coarsening and

---

6Chaco also has recursive bisection scheme that is similar to MLRB.
FIG. 15. Run time of MLkP relative to Chaco’s multilevel recursive octasection algorithm. The MLkP algorithm uses HEM during coarsening and GR during refinement. For each graph, the ratio of the run time of Chaco’s recursive octasection to that of the MLkP algorithm is plotted for 256-way partitionings. Bars above the baseline indicate that MLkP is faster than Chaco’s recursive octasection.
a good initial $k$-way partition is a much better choice than the computationally expensive 8-way Kernighan–Lin refinement.

4. CONCLUSIONS AND DIRECTIONS FOR FUTURE RESEARCH

Our experiments have shown that the multilevel $k$-way partitioning algorithm is significantly faster than recursive bisection based $k$-way partitioning scheme. The complexity of the coarsening and refinement phases of our $k$-way partition algorithm is $O(|E|)$, assuming that in each coarsening step the number of vertices is reduced by a factor larger than $1 + \epsilon$, where $\epsilon$ is a constant greater than zero. The complexity of obtaining the initial $k$-way partitioning of the coarsest graph using MLRB is $O(k \log k)$. Since $O(k \log k)$ is often smaller than $O(|E|)$, the overall complexity of the algorithm is $O(|E|)$. For instance, for TORSO the run time for a 2-way partitioning is 10.42 s while the run time for a 256-way partitioning is only 1.64 times higher (i.e., 17.13 s). As the problem size increases, this factor decreases. For example, for AUTO the runtime for a 2-way partitioning is 31.03 s while the run time for a 256-way partitioning is only 1.29 times higher (i.e., 40 s).

The quality of the partitionings produced by the $k$-way partitioning algorithm is comparable or better than that produced by the multilevel recursive bisection algorithm for a wide range of graphs. The scheme works well for a number of reasons. For coarsening heuristics such as HEM and HEM*, the edge-cut of the $k$-way partitioning produced by MLRB on the coarsest graph is usually within a factor of 1.3 of the final edge-cut. This occurs because the coarsening process creates an excellent smaller replica of the original graph, and MLRB finds a very good $k$-way partitioning on this small graph. A simple $k$-way refinement scheme such as GR is able to further improve the initial $k$-way edge-cut because the refinements needed are fairly local in nature. Hence, the extra power of generalized KL schemes (in terms of its capability of look-ahead) is often unnecessary because the refinement needed are fairly local in nature. (In our experiments, the look-ahead capability of GKLR refinement was found useful only for one type of graph.) Furthermore, even a simple refinement scheme such as GR is quite capable of moving large portions of graphs across the initial $k$-way partitioning because the refinement is done in a multilevel context. For coarse graphs, even a movement of a single vertex at the partition boundary is equivalent to moving a large number of vertices in the original graph. In fact, as discussed in [16], even for MLRB, many simpler variations of the KL refinement algorithm result in equally effective refinement scheme due to the same reason.

Absence of a priority queue in our GR refinement algorithm makes it naturally suited for parallel implementations. In contrast, the original KL refinement algorithm (and its generalization in the $k$-way partitioning context) are inherently sequential [6]. In [15] we have developed a highly parallel formulation of our multilevel $k$-way partitioning algorithm that uses the vertex-coloring of the successively coarser graph to effectively parallelize both the coarsening as well as the $k$-way refinement algorithms. Our experiments on the Cray T3D show that graphs with over a million vertices can be partitioned in 128 partitions in about 2 s on 128 processors.

An additional advantage of the MLkP algorithm over MLRB is that MLkP is much more suited in the context of parallel execution of adaptive computations [25, 26]. For example, in adaptive finite element computations, the mesh that models the physical
domain changes dynamically as the simulation progresses. In particular, some parts of
the mesh become finer and other parts get coarser. Such dynamic adjustments to the mesh
require partitioning of the mesh to improve load balance. This repartitioning also results in
movement of data structures associated with graph vertices. Hence, a good repartitioning
algorithm should minimize the movement of vertices (in addition to balancing the load
and minimizing the cut of the resulting new partition). If started with the multilevel
representation of the current partitioning of the graph, our \( k \)-way partitioning refinement
algorithm makes only minor adjustments to the previous partitioning and reduces the
overall movement of vertices and associated data structures.

In all of our experiments, we tried to minimize the edge-cut. However, for many
applications, minimizing other quantities, such as the number of vertices at the boundary
of the partitions, the number of adjacent partitions, or the shape of the partitions, may
be desirable. This can be accomplished by modifying the refinement algorithm to take
shape of the partitions, may be desirable. This can be accomplished by modifying the
refinement algorithm to take into account a different objective function. Even though
recursive bisection algorithms can also be modified to use objective functions other than
minimization of edge-cut, the multilevel \( k \)-way partitioning algorithm provides a much
better framework for this task. This is because multilevel \( k \)-way makes it possible to
incorporate “global” objective functions that cannot be achieved by recursive bisection
schemes. For example, the overall communication overhead of a processor in parallel
sparse matrix–vector multiplication is not proportional to the number of edges that
connect nonlocal vertices. Actually, it is proportional to the number of vertex values
it must communicate to neighboring processors. If a vertex on processor \( P_i \) is connected
to many vertices on processor \( P_j \), then the vertex value must be sent to processor \( P_j \) only
once (rather than once for each edge). Hence, the overall communication volume for a
processor is equal to \( \sum_v N_v \), where \( v \) are the boundary vertices in a processor, and \( N_v \)
is the number of other processors to which vertex \( v \) is connected. Note that this metric
can easily be used as the objective function in the \( k \)-way partitioning algorithm. But this
cannot be used in recursive bisection-based schemes, because \( \sum_v N_v \), for each processor
can be computed only in the context of a \( k \)-way partition.

The \( k \)-way partitioning algorithms described in this paper are available in the Metis 3.0
graph partitioning package that is publicly available on WWW at http://www.cs.umn.edu/~metis.
GEORGE KARYPIS received his Ph.D. in computer science at the University of Minnesota, and he is currently an assistant professor at the Department of Computer Science and Engineering at the University of Minnesota. His research interests span the areas of parallel algorithm design, applications of parallel processing in scientific computing and optimization, sparse matrix computations, parallel programming languages and libraries, and data mining. His recent work has been in the areas of parallel sparse direct solvers, serial and parallel graph partitioning algorithms, parallel matrix ordering algorithms, and scalable parallel preconditioners. His research has resulted in the development of software libraries for unstructured mesh partitioning (METIS and ParMETIS) and for parallel Cholesky factorization (PSPASES). He is the author of over 20 research articles and a coauthor of the widely used text book “Introduction to Parallel Computing.”

VIPIN KUMAR received his Ph.D. in computer science at the University of Maryland, and he is currently a professor at the Department of Computer Science and Engineering at the University of Minnesota. His current research interests include parallel computing, parallel algorithms for scientific computing problems, and data mining. His research has resulted in the development of highly efficient parallel algorithms and software for sparse matrix factorization (PSPASES), graph partitioning (METIS and ParMETIS), and dense hierarchical solvers. Kumar’s research in performance analysis resulted in the development of the isoefficiency metric for analyzing the scalability of parallel algorithms. He is the author of over 100 research articles and a coauthor of the widely used text book “Introduction to Parallel Computing.” Kumar has given over 50 invited talks at various conferences, workshops, and national labs and has served as chair/co-chair for many conferences/workshops in the area of parallel computing. Kumar serves on the editorial boards of IEEE Parallel and Distributed Technology, IEEE Transactions of Data and Knowledge Engineering, Parallel Computing, and the Journal of Parallel and Distributed Computing. He is a senior member of IEEE and a member of SIAM and ACM.

Received November 1, 1996; revised October 15, 1997; accepted October 20, 1997