Fast and effective algorithms for graph partitioning and sparse-matrix ordering

by A. Gupta

Graph partitioning is a fundamental problem in several scientific and engineering applications. In this paper, we describe heuristics that improve the state-of-the-art practical algorithms used in graph-partitioning software in terms of both partitioning speed and quality. An important use of graph partitioning is in ordering sparse matrices for obtaining direct solutions to sparse systems of linear equations arising in engineering and optimization applications. The experiments reported in this paper show that the use of these heuristics results in a considerable improvement in the quality of sparse-matrix orderings over conventional ordering methods, especially for sparse matrices arising in linear programming problems. In addition, our graph-partitioning-based ordering algorithm is more parallelizable than minimum-degree-based ordering algorithms, and it renders the ordered matrix more amenable to parallel factorization.

1. Introduction

Graph partitioning is an important problem with extensive application in scientific computing, optimization, VLSI design, and task partitioning for parallel processing. The graph-partitioning problem, in its most general form, requires dividing the set of nodes of a weighted graph into k disjoint subsets or partitions such that the sum of weights of nodes in each subset is nearly the same (within a user-supplied tolerance) and the total weight of all of the edges connecting nodes in different partitions is minimized. In this paper, we describe heuristics that significantly improve the practical state-of-the-art graph-partitioning algorithms in partitioning speed and, for a small number of parts, also in partitioning quality.

An important application of graph partitioning is in computing fill-reducing orderings of sparse matrices for solving large sparse systems of linear equations. Finding an optimal ordering is an NP-complete problem [1], and heuristics must be used to obtain an acceptable non-optimal solution. Improving the run time and quality of ordering heuristics has been a subject of research for

0018-8646/97/$5.00 © 1997 IBM

almost three decades. Two main classes of successful heuristics have evolved over the years: (1) minimum-degree (MD)-based heuristics, and (2) graph-partitioning (GP)-based heuristics. MD-based heuristics are local greedy heuristics that reorder the columns of a symmetric sparse matrix such that the column with the fewest nonzeros at a given stage of factorization is the next one to be eliminated at that stage. GP-based heuristics regard the symmetric sparse matrix as the adjacency matrix of a graph and follow a divide-and-conquer strategy to label the nodes of the graph by partitioning it into smaller subgraphs.

The initial success of MD-based heuristics prompted intense research [2] to improve their run time and quality, and they have been the methods of choice among practitioners. The multiple minimum-degree (MMD) algorithm by George and Liu [2, 3] and the approximate minimum-degree (AMD) algorithm by Davis, Amestoy, and Duff [4] represent the state of the art in MD-based heuristics. Recent work by the author [5, 6], Hendrickson and Rothberg [7], Ashcraft and Liu [8], and Karypis and Kumar [9, 10] suggests that GP-based heuristics are capable of producing better-quality orderings than MD-based heuristics for finite-element problems, while staying within a small constant factor of the run time of MD-based heuristics.

An important area where sparse-matrix orderings are used is that of linear programming. Until now, with the exception of Rothberg and Hendrickson [11], most researchers have focused on ordering sparse matrices arising in finite-element applications, and these applications have guided the development of the ordering heuristics. The use of the interior-point method for solving LP problems is relatively recent. As a result, the LP community has been using these well-established heuristics that were not originally developed for their applications. Our GP-based sparse-matrix-ordering algorithm is capable of generating robust orderings of sparse matrices arising in LP problems, in addition to finite-element and finite-difference matrices. Our experimental results show that the benefits of our ordering over MD-based heuristics are even more pronounced for LP matrices than for FE matrices. Our ordering often turned out to be several hundred times faster than AMD and MMD on our set of LP test matrices, while producing better-quality orderings on average.

We have developed a graph-partitioning and sparse-matrix-ordering package (WGPP) [5][1] based on the heuristics described in this paper. We present experimental results of WGPP for graph partitioning and for ordering sparse matrices arising in two very different types of applications: finite-element (FE) analysis and linear programming (LP). The remainder of the paper is organized as follows. Section 2 briefly discusses the relevance of this paper to parallel solution of large sparse systems of linear equations. Section 3 describes the heuristics used for multilevel graph partitioning. Section 4 describes how graph bisection (partitioning into two parts) is used to obtain fill-reducing orderings of sparse matrices. In Section 5, we compare the quality and run time of WGPP with those of a state-of-the-art graph-partitioning package. Section 5 also presents experimental results comparing sparse-matrix orderings produced by WGPP with those of other softwares for a variety of matrices arising in different applications.

2. Application in parallel solution of linear equations

Both graph partitioning and GP-based sparse-matrix ordering have applications in the parallel solution of large sparse systems of linear equations. Partitioning the graph of a sparse matrix to minimize the edge cut and distributing different partitions to different processors minimizes the communication overhead in parallel sparse-matrix vector multiplication [12]. Sparse-matrix vector multiplication is an integral part of all iterative schemes for solving sparse linear systems.

GP-based ordering methods are more suitable for solving sparse systems using direct methods on distributed-memory parallel computers than MD-based methods, in two respects. First, there is strong theoretical and experimental evidence that the process of graph partitioning and sparse-matrix ordering based on it can be parallelized effectively [13]. On the other hand, the only attempt to perform a minimum-degree ordering in parallel that we are aware of [14] was not successful in reducing the ordering time over a serial implementation. Second, in addition to being parallelizable itself, a GP-based ordering also aids the parallelization of the factorization and triangular solution phases of a direct solver. Gupta, Karypis, and Kumar [15, 16] have proposed a highly scalable parallel formulation of sparse Cholesky factorization. This algorithm derives a significant part of its parallelism from the underlying partitioning of the graph of the sparse matrix. In [17], Gupta and Kumar present efficient parallel algorithms for solving lower- and upper-triangular systems resulting from sparse factorization. In both parallel factorization and triangular solutions, a part of the parallelism would be lost if an MD-based heuristic were used to preorder the sparse matrix.

3. Multilevel graph partitioning

Recent research [10, 18, 19] has shown *multilevel algorithms* to be fast and effective in computing graph partitions. A typical multilevel graph-partitioning algorithm has four components: *coarsening, initial partitioning, uncoarsening*, and *refining*. In the following

[1] The package is available to users in the form of a linkable module.

subsections, we briefly discuss these components of graph partitioning, describing in detail only the new heuristics and improvements over the current techniques.

3.1 Coarsening

The goal of the coarsening phase is to reduce the size of a graph while preserving those of its properties that are essential to finding a good partition. The original graph is regarded as a weighted graph, with a unit weight assigned to each edge and each node. In a coarsening step, a maximal set of edges of the graph is identified such that no two edges have a vertex in common. This set of edges is known as a *matching*. The edges in this set are removed, and the two nodes connected by an edge in the matching are collapsed into a single node whose weight is the sum of the weights of the component nodes. **Figure 1** illustrates the coarsening steps. Note that coarsening also results in some edges being collapsed into one, in which case the collapsed edge is assigned a weight equal to the sum of weights of the component edges.

Given a weighted graph after any stage of coarsening, there are several choices of matchings for the next coarsening step. A simple matching scheme [19] known as *random matching (RM)* randomly chooses pairs of connected unmatched nodes to include in the matching. In [10], Karypis and Kumar describe a heuristic known as *heavy-edge matching* (HEM) to aid in the selection of a matching that not only reduces the run time of the refinement component of graph partitioning, but also tends to generate partitions with small separators. The strategy is to randomly pick an unmatched node, select the edge with the highest weight among the edges incident on this vertex that connect it to other unmatched vertices, and mark both vertices connected by this edge as matched. Note that the weight of an edge connecting two nodes in a coarsened version of the graph is the number of edges in the original graph that connect the two sets of original nodes collapsed into the two coarse nodes. HEM, by absorbing the heavier edges, generates coarse graphs whose nodes are loosely connected (by the lighter remaining edges), thus ensuring that a partition of the coarse graph corresponds to a good partition of the original graph.

HEM can miss some heavy edges in the graph because the nodes are visited randomly for matching. For example, consider a node i, the heaviest edge incident on which connects it to a node j. If i is visited before j and both i and j are unmatched, the edge (i, j) will be included in the matching. If there exists an edge (j, k), such that i and k are not connected, it will be excluded from the matching even if it is much heavier than (i, j) because j is no longer available for matching. To overcome this problem, after the first few coarsening steps, we switch to what we call the *heaviest-edge matching*. We sort the edges by their weights and visit them in decreasing order of weight to inspect them for possible inclusion in the matching. Ties are broken randomly. The benefit of heaviest-edge matching over heavy-edge matching is more pronounced in the later stages of coarsening, where sorting is not too expensive because the graph has shrunk considerably from its original size.

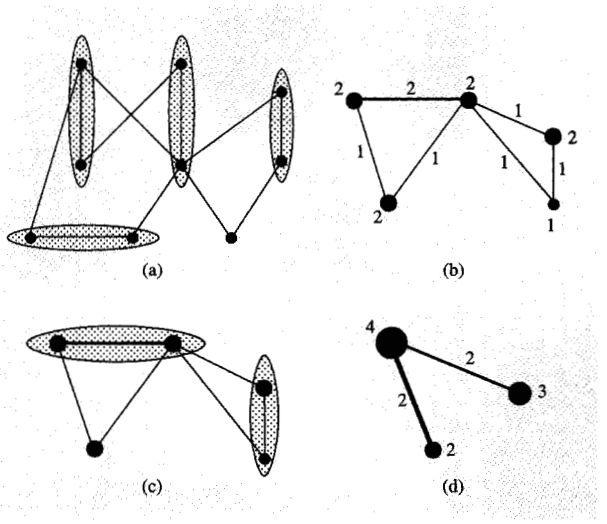

Figure 1

Illustration of graph coarsening: (a) Original graph and a matching; (b) graph after one step of coarsening; (c) matching of the coarse graph; (d) graph after two steps of coarsening.

HEM and its variants reduce the number of nodes in a graph by roughly a factor of 2 at each stage of coarsening. Therefore, the number of coarsening, uncoarsening, and refining steps required to partition an n-node graph into k parts is $\log_2(n/k)$. If r (instead of 2) nodes of the graph are coalesced into one at each coarsening step, the total number of steps can be reduced to about $\log_r(n/k)$. Fewer steps are likely to reduce to overall run time. However, as r is increased, the task of refining after each uncoarsening step (see Section 3.3 for the description of uncoarsening and refinement) becomes harder. This affects both the run time and the quality of refinement. In our experiments, we observed that increasing r from 2 to 3 results in about 20% time savings with only a minor compromise in partitioning quality. In WGPP, we use a combination of heavy-edge matching, heaviest-edge matching, and *heavy-triangle matching (HTM)*. HTM coalesces three nodes at a time by picking an unmatched node at random and matching it with two of its neighbors such that the sum of the weights of the three edges connecting the three nodes is maximized over all pairs of neighbors of the selected node. A nonexistent edge between the two neighbors is

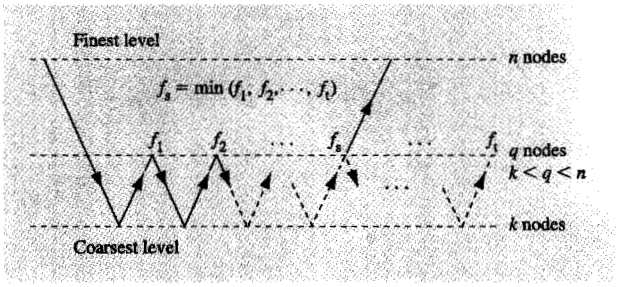

Figure 2

Overview of our uncoarsening and refining scheme for finding a k-way partition; f_i is a weighted average of the edge-separator size and the inverse of weight imbalance between the parts in the ith partition.

regarded as an edge of weight zero. Table 4 (shown later) compares the run time and the quality of 24-way partitions using HEM and HTM.

3.2 *Initial partitioning*

All multilevel graph-partitioning schemes described in the literature stop the coarsening phase when the graph has been reduced to a few hundred to a few thousand nodes and use some heuristic to compute an initial partition of the coarse graph at a substantial run-time cost. In WGPP, we have completely eliminated the initial partitioning phase, thereby simplifying and speeding up the overall partitioning process.

One of the effective heuristics for initial partitioning is *graph growing* [10]. Graph growing computes an initial partition by recursively bisecting the graph into two subgraphs of appropriate weight. For example, a three-way initial partition is produced by bisecting the coarse graph into two parts with a weight ratio of 2:1; the larger subgraph is further bisected into two equal parts. In order to bisect a graph, the graph-growing heuristic would pick up a node at random, tag it, and keep tagging its neighbors and neighbors' neighbors in a breadth-first manner until the ratio between the cumulative weight of the tagged nodes and that of the untagged nodes reaches the desirable value. The sets of tagged and untagged nodes now form the two partitions of the graph.

The basic function of graph growing is to form a cluster of highly connected nodes. Note that coarsening with HEM or HTM also strives to achieve the same goal. In fact, graph growing is the bottom-up equivalent of coarsening. Therefore, for a k-way partition of an n-node graph, WGPP continues to coarsen the graph until it contains exactly k nodes. This coarse k-node graph serves as a good initial partitioning, provided that the coarsening algorithm does not allow a node to participate in matching if its weight substantially exceeds n/k.

3.3 *Uncoarsening and refinement*

The uncoarsening and refining components of graph partitioning work together. Initially, the k nodes of the coarsest graph are assigned different tags indicating that they belong to different initial partitions. They are then split into the nodes that were collapsed to form them during the coarsening phase. This reversal of coarsening is carried out one step at a time. The nodes of the uncoarsened graph inherit their tags from their parent nodes in the coarser graph. At any stage of uncoarsening, the edges connecting a pair of nodes belonging to different partitions constitute an edge-separator of the graph. Removing the edge-separator from the graph breaks it into k disconnected subgraphs. After each step of uncoarsening, the separator is refined. While refining an edge-separator, an attempt is made to minimize its total weight by switching the partitions of some nodes if this switching reduces the separator size and does not make the weights of the parts too imbalanced.

Figure 2 gives an overview of our uncoarsening and refinement scheme for partitioning into k parts. After coarsening the original n-node graph to k nodes, we uncoarsen it to q nodes, where $k < q < n$, while refining the edge-separator after each step of uncoarsening. At this stage we save the partition, coarsen the q-node graph back to k nodes, and repeat the uncoarsening and refining process. This process is repeated a few times, and of all the partitions of the q-node graph generated, we choose the best for further uncoarsening to obtain a partition of the original n-node graph. The best partition is selected on the basis of a weighted average of the size of the edge-separator and the inverse of the weight imbalance between the parts. The randomization in the coarsening process ensures that the different trials result in sufficiently different partitions. Most of the partition cost is incurred in the first few coarsening steps of the original graph and the last few uncoarsening and refining steps when the size of the graph is large. Therefore, multiple cycles of coarsening and uncoarsening on the smaller, coarser q-node graph have little impact on the overall run time of the entire algorithm.

Depending on the number of partitions k, we either use a variation of the popular Kernighan–Lin heuristic [19–22] or a greedy refinement scheme [10, 23] for refining the edge-separators. The linear-time Fiduccia–Mattheyses variation [21] of the Kernighan–Lin heuristic is used for a small number of partitions, and the greedy algorithm is used if the number of partitions required is large. The Kernighan–Lin and Fiduccia–Mattheyses heuristics were originally developed to refine two partitions, but can be adapted for refining multiple partitions [19, 22] by using multiple-priority queues.

4. Ordering sparse matrices using multiple multilevel recursive bisections

In this section, we describe how graph bisection (partition into two parts) by the process described in Section 3 is used to compute a fill-reducing ordering of a symmetric sparse matrix. The overall ordering algorithm involves several stages, a preprocessing step, and a postprocessing step. In Sections 4.1 and 4.2, we describe the core algorithm and its various stages. Section 4.3 describes some preprocessing steps that have a significant impact on the quality and the run time of ordering. Section 4.4 describes the postprocessing step that may further improve the ordering quality.

Any GP-based ordering algorithm regards the matrix as the adjacency matrix of a graph and assigns labels to the nodes of the graph. These labels specify the sequence in which the matrix columns corresponding to the nodes are eliminated during numerical factorization. The overall approach of our ordering algorithm follows the fundamental technique of generalized nested dissection [24]. The graph is bisected by finding and removing a node-separator, labeling the nodes of the two resulting subgraphs by applying the same technique recursively, and labeling the nodes of the separator after the nodes of the subgraphs have been labeled (i.e., the separator nodes receive a higher label than any of the nodes of the subgraph). The recursion terminates when the subgraphs become too small, at which stage they are labeled using a minimum-degree heuristic.

The reason why this technique works is that, upon elimination, a column corresponding to any of the nodes in the two subgraphs can create a fill only in the columns corresponding to the separator nodes and not any node of the other subgraph. Intuitively, smaller node-separators mean less fill and work during factorization. For finite-element graphs with certain tractable properties, it can be proved that this technique yields orderings within a constant factor of the optimal [3, 24–26]. Such bounds cannot be proved for arbitrary matrices without a well-defined structure, such as the sparse matrices arising in LP computations. However, as we show in [27] and Section 5, with suitable modifications, the strategy works quite well in practice even for matrices with arbitrary sparsity patterns. In addition, graph bisection yields two submatrices of the original matrix that can be factored independently in parallel. This is the basis of many efficient parallel algorithms for sparse-matrix factorization [15, 16, 28].

4.1 Graph bisection

A key step in our ordering algorithm is finding a small node bisector of a graph. This can be accomplished by the heuristics described in Section 3 with some modifications to the coarsening and refinement strategies.

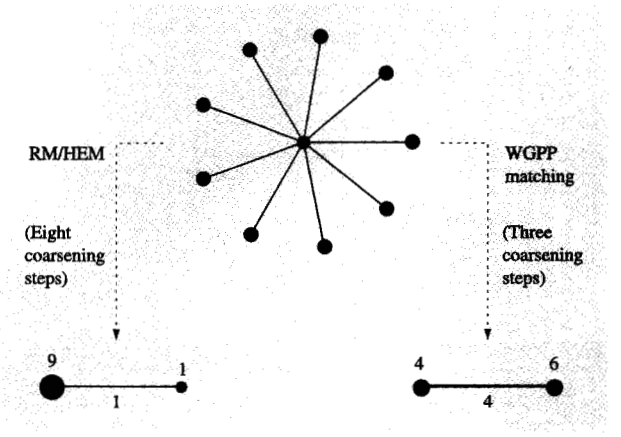

Figure 3

Coarse graphs produced by RM/HEM and the WGPP matching algorithms when a graph with a star-shaped structure is coarsened to a two-node graph.

The heavy- and heaviest-edge coarsening strategies discussed in Section 3.1 may occasionally fail to satisfactorily coarsen the graph of an LP matrix. An example of such a graph is one that has a star-like structure (**Figure 3**). Since RM and HEM disregard the possibility of matching disconnected nodes, they fail to preserve the properties of such a graph upon coarsening and may result in a very unbalanced coarse graph. Moreover, RM and HEM reduce the size of a star-shaped graph by only one node in one coarsening step. In order to overcome these problems, conceptually, WGPP regards every graph as a completely connected graph with weight zero assigned to every edge (i, j) that does not really exist in the physical graph. It then attempts to maximize a modified weight function of any two unmatched nodes i and j when considering them to include in the matching as a pair. The modified edge weight $ew'(i, j)$ between two nodes i and j with node weights $nw(i)$ and $nw(j)$, respectively, and connected by an edge of (possibly zero) weight $ew(i, j)$ is given by

$$ew'(i,j) = \alpha \times ew(i,j) + \frac{\beta}{nw(i) + nw(j)}.$$

Here α and β are constants defined at the beginning of the given level of coarsening. WGPP uses $ew'(i, j)$ instead of the real edge weight $ew(i, j)$ within the HEM framework. The modified edge weight criterion yields a balanced coarsening in fewer steps while preserving the benefits of HEM. To save coarsening time in the actual implementation, WGPP switches to using the modified edge weights only if heavy- and heaviest-edge matchings fail to reduce the size of the graph sufficiently.

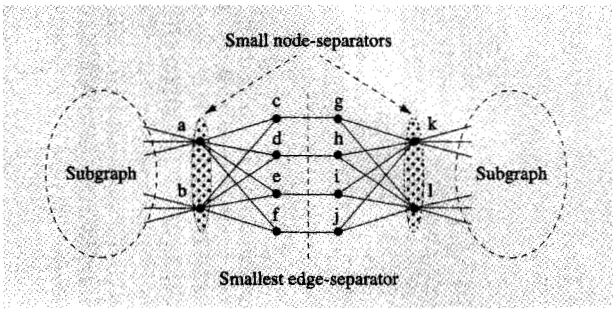

Figure 4

Drawbacks of choosing a node-separator from the minimum edge-separator.

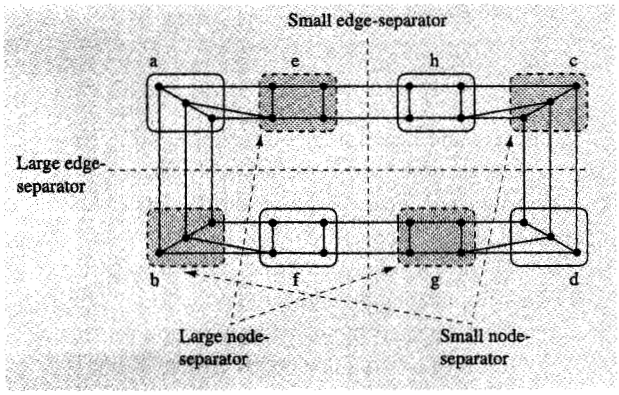

Figure 5

Drawbacks of choosing a small node-separator for refining in a coarse graph.

The second modification required to adapt the graph-partitioning algorithm to bisection for ordering is in the refinement phase. A typical graph-partitioning application requires the total weight of the edge-separator to be minimized. However, while computing a fill-reducing ordering of a sparse matrix, it is the size of the node-separator that must be minimized. Current graph-partitioning-based ordering algorithms follow two different approaches to finding a small node-separator. Karypis and Kumar [9] refine the edge-separator between the two subgraphs after each step of uncoarsening so that few edges connect nodes of different subgraphs in the final partitioning of the original graph. Then they use an algorithm for finding a minimum cover [29, 30] to compute a node-separator from the edge-separator. This approach relies heavily on the assumption that the size of a node-separator is proportional to the size of the edge-separator containing it. This assumption is often incorrect, especially for the highly unstructured LP matrices. For example, consider the graph in **Figure 4**. Minimizing the edge-separator will partition the graph between nodes c, d, e, f and nodes g, h, i, j. The smallest possible node-separator that can be extracted from the edge-separator $(c, g), (d, h), (e, i), (f, j)$ contains at least four nodes. On the other hand, node-separators of size two (i.e., a, b or k, l) are available with almost the same degree of imbalance between the two partitions.

Ashcraft and Liu [8] find a node-separator within the coarse graph and refine it into a node-separator of the original graph in order to overcome the drawback of the edge-separator approach. In **Figure 5**, we show that a heavier node-separator in the coarse graph can result in a smaller node-separator in the original graph after uncoarsening (and vice versa). The weight of a node in the coarse graph is the number of nodes of the original graph that are collapsed during the coarsening steps to form the coarse node. Not all of the component nodes of a coarse separator node may have edges crossing partition boundaries. As the graph is uncoarsened, such nodes are eliminated from the node-separator by the refining process. How many such nodes are eliminated from a node-separator depends on the connectivity of these nodes to the nodes outside the separator. For example, in Figure 5, the smaller coarse node-separator consisting of nodes b and c has a total weight of 6. This coarse separator results in a final node-separator of size 6. This is because all six nodes have interpartition connections, and none of them can be removed from the node-separator by refining. On the other hand, the coarse separator of weight 8 consisting of nodes e and g results in a final separator containing only four nodes. This is because two of the four component nodes of both coarse nodes c and g have no interpartition edges and are eliminated from the separator by refinement. In this example, choosing the smaller of the two edge-separators in the coarse graph yields the smaller node-separator in the original graph. In contrast to the original graph, the size of the edge-separator is not completely irrelevant in coarse versions of the graph. In fact, the coarser the graph, the more relevant the edge-separator size is in predicting the size of the node-separator in the original graph.

With these motivations, we slightly modify the graph-partitioning strategy illustrated in Figure 2 to compute bisections for sparse-matrix ordering. We coarsen the original n-node graph to two nodes. We then repeatedly uncoarsen it to q nodes, where $2 < q < n$, and recoarsen it to two nodes to select an initial partition. The uncoarsening from 2 to q is accompanied by refining the edge-separator. We select the best bisection of the q-node graph on the basis of a weighted average of the size of the

edge-separator, the size of the node-separator, and the inverse of the imbalance between the two parts. The selected partition of the q-node graph is uncoarsened to the original n-node graph one step at a time. After each uncoarsening step, we refine the node-separator; i.e., in the final steps of uncoarsening, the subject of refinement is changed from the edge-separator to a node-separator. For refining the node-separator, we use Ashcraft and Liu's modification [31] of the Fiduccia–Mattheyses algorithm [21].

4.2 *Recursive bisection and ordering*

Once a node-separator of the original graph is found, it is removed, and the entire procedure is repeated recursively on the two disconnected subgraphs. The recursion terminates if a subgraph is too small, in which case it is not partitioned any further but ordered using a minimum-degree heuristic. In our implementation, the size of the terminal subgraphs ranges from a hundred nodes to a few hundred nodes depending on the size of the original graph. We use the AMD variation (due to Davis, Amestoy, and Duff [4]) of the minimum-degree heuristic for ordering these small subgraphs.

After the nodes of the two subgraphs at any level of recursion have been labeled, the nodes of the separator at that level are labeled and the algorithm returns to the previous level, if any. This procedure ensures that the labels assigned to the nodes of any separator are higher than any label in the two disconnected subgraphs separated by the separator. Therefore, during numerical factorization, the columns corresponding to the separator nodes are eliminated after all columns corresponding to the nodes of the two subgraphs have been eliminated.

4.3 *Preprocessing*

Sections 4.1 and 4.2 describe the process of labeling the nodes of the graph corresponding to a sparse matrix. In this section, we describe two simple preprocessing steps that, if performed before subjecting the graph to the labeling process described previously, may significantly improve the run time and the quality of the ordering heuristic. The appropriate preprocessing step is applied in WGPP to a matrix depending on its origin, which is specified by the user. In the default case, both optimizations are attempted.

The first preprocessing step is applicable only to matrices arising from LP problems. Most matrices of the form AD^2A^T that have to be factored while solving an LP problem using a barrier method have the property that a few (typically about 2%) of the columns are much denser than the remaining columns. The number of nonzeros in these few columns is typically several times greater than the average number of nonzeros per column of the sparse matrix. The preprocessing step, illustrated in **Figure 6**, simply removes the nodes with excessively high degrees

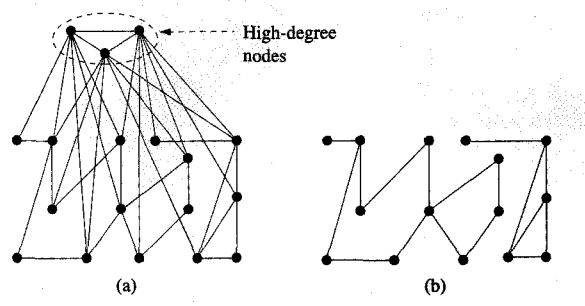

Figure 6

Illustration of the preprocessing step performed in WGPP before ordering an LP matrix: (a) Original graph; (b) graph after removal of the three high-degree nodes.

from the graph. The remainder of the graph is labeled using the heuristics described in Sections 4.1 and 4.2, and then the previously removed high-degree nodes are assigned the highest labels. Typically, the removal of these nodes either breaks the graph up into several disconnected components, or renders it much more loosely connected than the original graph. This aids the partitioning algorithm in quickly finding node-separators of a small (or zero) size.

The second preprocessing technique applies to a class of finite-element matrices. Certain finite-element meshes have many points with more than one degree of freedom. The sparse matrices resulting from such finite-element problems have many small groups of nodes that share the same adjacency structure. In [32], Ashcraft describes a technique to compress such graphs into smaller graphs by coalescing the nodes with identical adjacency structures. As a result of compression, the ordering algorithm must process a much smaller graph; this, depending on the degree of compression achieved, can reduce the ordering time. WGPP ordering has the option of compressing the graph as a preprocessing step and then working with the compressed graph.

4.4 *Postprocessing*

As the factorization of a sparse symmetric matrix proceeds from top left to bottom right, the columns that are eliminated create nonzero entries, or fill-in, in the remainder of the matrix. Toward the end of the factorization process, the lower right corner of the matrix often becomes fairly dense. Many implementations of sparse factorization switch to dense-matrix factorization at this stage because the cost of additional arithmetic operations is more than compensated for by the

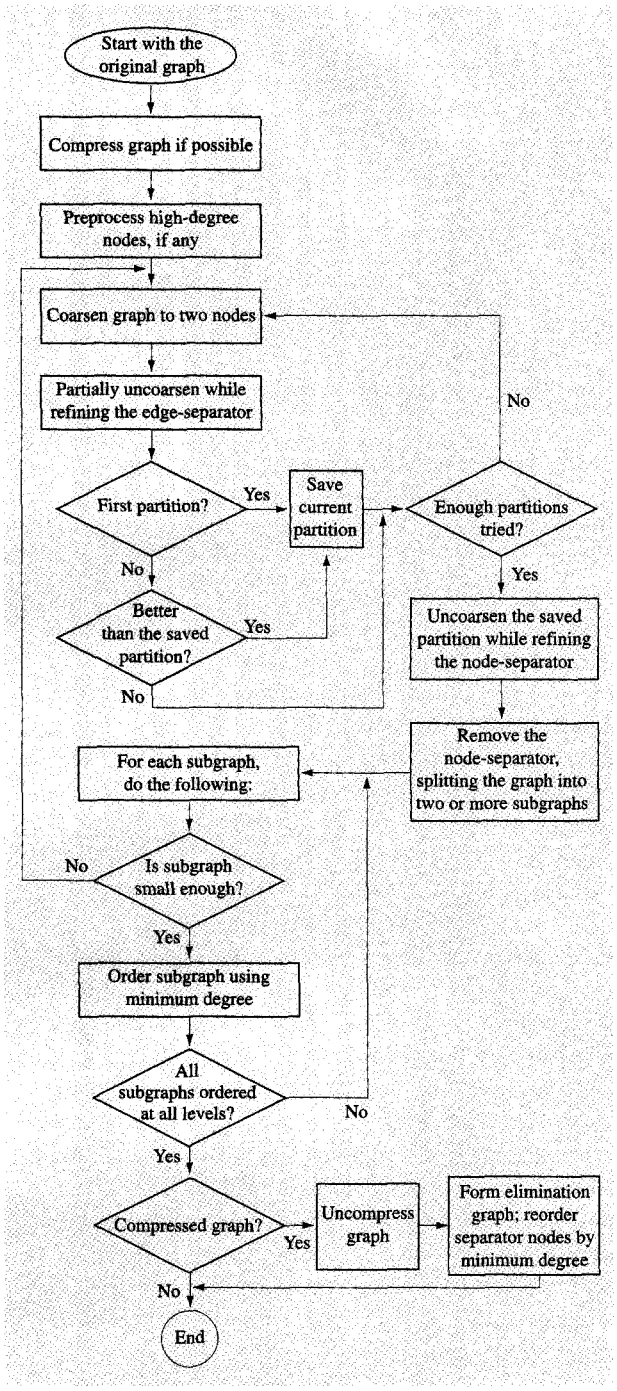

Figure 7

Overview of the ordering algorithm used in WGPP.

advantages of dense factorization, such as the absence of indirect addressing and better utilization of memory hierarchy.

With a GP-based ordering, the matrix columns corresponding to the nodes of a separator usually tend to become denser during factorization than the columns corresponding to the nodes of the subgraphs that the separator separates. This is because the separator columns receive fill-in from the columns of both of the subgraphs that they separate. On the other hand, the columns of each subgraph receive fill-in from the nodes of only that subgraph. In addition, separators typically have fewer nodes than the separated subgraphs. A large number of columns contributing fill-in to a small number of separator columns results in the separator columns becoming relatively dense during factorization.

An implementation of sparse-matrix factorization that switches to dense factorization would benefit from a larger and denser bottom right leftover portion of the matrix during factorization. In order to aid such implementations, we collect all separators larger than a certain size and relabel their nodes so that they receive the highest labels. In other words, we regard the top m levels of recursive bisection as a single partition into 2^m parts and all of the $2^m - 1$ bisectors (a separator that bisects) as a single composite separator whose removal results in 2^m disconnected subgraphs. The nodes of these 2^m subgraphs are labeled before the nodes of the composite separator. While labeling the nodes of the composite separator, we make sure that the nodes are labeled according to the depth of the level of recursion in which they were included in the separator. For example, the nodes of the first bisector of the original graph receive higher labels than the nodes of the two bisectors of the two resulting subgraphs.

This relabeling of separator nodes does not affect the total amount of fill-in during factorization or its operation count. It affects only the location of fill-in and attempts to direct most of the fill-in toward the high-index columns of the matrix. However, for most finite-element and finite-difference matrices, it is possible to relabel these separator nodes to reduce the fill-in and the operation count. This technique, which has been used in [7, 8], involves creating an *elimination graph* consisting of the separator nodes and using a minimum-degree heuristic to order this graph. The elimination graph is the graph of the partially factored sparse matrix in which all columns except those corresponding to the separator nodes have been eliminated.

Figure 7 illustrates the overall ordering procedure by means of a flowchart.

5. Experimental results

In this section, we present experimental results on the run time and quality of graph partitions and sparse-matrix orderings generated by WGPP. To the best of our knowledge, the METIS [9] software package represents

Table 1 Description of the sparse matrices (and graphs) used in the experiments.

Matrix/Graph	*Number of columns or nodes*	*Number of nonzeros*	*Density* (nonzeros per column)	*Source*
1. ALLGRADE	21601	2829003	131.0	Linear programming
2. BCSSTK-13	2003	83883	41.9	Fluid flow
3. BCSSTK-15	3948	117816	29.8	Structural engineering
4. BCSSTK-25	15439	252241	16.3	Structural engineering
5. BCSSTK-29	13992	619488	44.3	Structural engineering
6. BCSSTK-30	28924	2043492	70.7	Structural engineering
7. BCSSTK-31	35588	1181416	33.2	Structural engineering
8. BCSSTK-32	44609	2014701	45.2	Structural engineering
9. COMP-1	16783	9323427	555.5	Linear programming
10. COPTER-2	55476	759952	13.7	3D finite element mesh
11. CUBE-35	42875	292775	6.8	3D finite difference grid
12. CRONE	21901	152073	6.9	2D finite element mesh
13. DWT0512	512	3502	6.8	Submarine model
14. FLEET-12	20922	379234	18.1	Linear programming
15. GISMONDI	17447	475835	27.3	Linear programming
16. GRID-127	16129	143641	8.9	2D finite difference grid
17. HSCT16K-A	16152	769017	47.6	Structural analysis
18. HSCT16K-B	16146	1015156	62.9	Structural analysis
19. HSCT22K	21954	2119062	96.5	Structural analysis
20. HSCT44K	44396	3990735	90.0	Structural analysis
21. HSCT88K	88404	3544842	40.1	Structural analysis
22. K-8	21059	425533	20.2	Linear programming
23. KEN-18	39919	385235	9.7	Linear programming
24. KK-6	62064	4248286	68.5	Linear programming
25. PDS-20	28062	285966	10.2	Linear programming
26. PILOT	1441	120465	83.6	Linear programming
27. USAIR	13564	181634	13.4	Linear programming

the state of the art in graph partitioning—both in terms of partitioning time and quality. In Section 5.1, we compare WGPP with METIS. Currently the best minimum-degree-based code available for computing fill-reducing ordering for sparse matrices is that of approximate minimum degree (AMD) [4]. METIS is one of the well-known graph-partitioning-based sparse-matrix-ordering software packages. Recent work by Ashcraft and Liu [8] and by Hendrickson and Rothberg [7] reports graph-partitioning-based ordering heuristics that are better than METIS, but the corresponding software is not available for direct comparison with WGPP. Therefore, in Section 5.2, we compare WGPP's sparse-matrix orderings with those of AMD and METIS. Results for the traditional multiple minimum-degree (MMD) algorithm are also included for reference. In **Table 1**, we introduce the graphs and sparse matrices for which the experimental data are presented in Sections 5.1 and 5.2. All of the codes being compared were compiled with the -O3 option using XLF 3.2 and run on an IBM RS/6000™ Model 590 workstation.

5.1 Graph partitioning

The quality of a partition of an unweighted graph is measured in terms of the edge-cut, or the total number of edges between nodes belonging to different partitions, and the balance in the number of nodes assigned to each part. In this section, we compare the edge-cuts of the partitions produced by WGPP and METIS. In METIS, the upper bound on the imbalance between the weights of the partitions is 3%; i.e., the number of nodes in any partition does not exceed $1.03 \times n/k$, where n is the total number of nodes in the graph and k is the number of parts. In WGPP, the user has the option of specifying the maximum tolerable imbalance. The experiments in this section were conducted with this option set at 3%. Although well below 3%, the actual imbalance was observed to be somewhat higher for WGPP than for METIS. The comparisons in this section are made for 2, 24, and 160 parts, and the respective comparisons are representative of partitioning graphs into small, medium, and large numbers of parts. Both METIS and WGPP offer a few different choices of coarsening and refinement algorithms. For 2 and 160 parts, the best choices of METIS are compared with the default choices of WGPP. For the 24-way partition, since there is no obvious best choice, the three most appropriate combinations of coarsening and refinement methods are compared for both METIS and WGPP.

Table 2 Run time and edge-cut comparison between WGPP and METIS for graph bisection.

Graph	*PMETIS (HEM + BKL)*		*WGPP (HEM + BKL)*		$\frac{T_{METIS}}{T_{WGPP}}$	$\frac{\|EC\|_{METIS}}{\|EC\|_{WGPP}}$
	T (s)	$\|EC\|$	T (s)	$\|EC\|$		
1. BCSSTK-15	0.452	1558	0.139	1528	3.25	1.02
2. COPTER-2	2.394	2252	1.012	2068	2.37	1.09
3. CRONE	0.487	183	0.304	164	1.60	1.12
4. CUBE-35	1.267	1387	0.766	1295	1.65	1.07
5. GRID-127	0.467	395	0.285	379	1.64	1.04
6. HSCT16K-B	2.763	6508	0.817	6488	3.38	1.00
7. HSCT88K	9.621	10933	2.659	8327	3.62	1.31
8. PILOT	0.464	3408	0.130	3379	3.57	1.01
Total	17.92	26624	6.20	23628	2.93	1.13
Average					2.64	1.08

Table 3 Run times and edge-cuts for partitioning graphs into 24 parts using three different variations of METIS.

Graph	*PMETIS (HEM + BKL)*		*KMETIS (HEM + BKL)*		*KMETIS (HEM + BGR)*	
	T (s)	$\|EC\|$	T (s)	$\|EC\|$	T (s)	$\|EC\|$
1. BCSSTK-15	1.286	12684	1.146	12904	0.977	12886
2. COPTER-2	7.034	26539	3.948	25257	3.434	25801
3. CRONE	1.558	1831	0.780	1819	0.747	1838
4. CUBE-35	4.050	8576	2.215	9161	1.806	9527
5. GRID-127	1.456	3153	0.824	3112	0.758	3167
6. HSCT16K-B	7.135	53814	4.115	53719	3.762	53541
7. HSCT88K	23.264	91056	11.333	88802	10.948	90182
8. PILOT	1.225	39901	1.224	39534	1.222	39534
Total	47.00	237554	25.59	234308	23.65	236476

Table 4 Run times and edge-cuts for partitioning graphs into 24 parts using three different variations of WGPP.

Graph	*WGPP (HTM + BKL)*		*WGPP (HEM + BKL)*		*WGPP (HEM + BGR)*	
	T (s)	$\|EC\|$	T (s)	$\|EC\|$	T (s)	$\|EC\|$
1. BCSSTK-15	0.650	12563	0.960	12244	0.152	12759
2. COPTER-2	3.047	25566	3.218	32303	1.361	27269
3. CRONE	0.894	1811	0.980	1894	0.427	2053
4. CUBE-35	2.960	8917	3.300	9032	1.060	9923
5. GRID-127	1.050	2922	1.043	3039	0.363	3158
6. HSCT16K-B	1.299	68356	3.068	56441	0.873	57207
7. HSCT88K	5.558	88482	6.801	88993	3.141	96065
8. PILOT	0.384	42027	0.580	41362	0.286	42185
Total	15.46	250644	19.95	245308	7.66	250619

Tables 2 through 5 give the various results. In these tables, HEM refers to heavy-edge matching in the context of METIS and heaviest-edge matching in the context of WGPP, HTM refers to heavy-triangle matching, BKL refers to boundary Kernighan–Lin refinement, and BGR refers to boundary greedy refinement. PMETIS is a variation of METIS that uses recursive bisection for partitioning, and KMETIS is a variation that uses a single coarsening and refining cycle to generate the entire partition.

Table 5 Run time and edge-cut comparison between WGPP and METIS for partitioning graphs into 160 parts.

Graph	*KMETIS (HEM + BGR)*		*WGPP (HEM + BGR)*		$\frac{T_{METIS}}{T_{WGPP}}$	$\frac{\|EC\|_{METIS}}{\|EC\|_{WGPP}}$
	T (s)	|*EC*|	*T* (s)	|*EC*|		
1. BCSSTK-15	1.624	32197	0.348	31797	4.67	1.01
2. COPTER-2	5.271	61802	2.118	62302	2.49	0.99
3. CRONE	1.459	6120	0.543	6894	2.69	0.89
4. CUBE-35	3.241	20213	1.590	20962	2.04	0.96
5. GRID-127	1.421	8885	0.487	9126	2.92	0.97
6. HSCT16K-B	6.231	178985	1.688	180410	3.69	0.99
7. HSCT88K	13.560	250598	3.993	271210	3.40	0.92
8. PILOT	2.075	54872	0.321	54777	6.39	1.00
Total	34.88	613672	11.09	637478	3.15	0.96
Average					3.54	0.97

Table 6 Run time and factorization operation count (in millions) comparison among MMD, AMD, METIS, and WGPP. The best time and operation count appear in parentheses for each matrix. A dash implies that the ordering failed to complete.

Matrix	*MMD*		*AMD*		*METIS*		*WGPP*	
	T (s)	*OPC*	*T* (s)	*OPC*	*T* (s)	*OPC*	*T* (s)	*OPC*
1. ALLGRADE	20.7	1438	(8.1)	1334	24.9	2042	16.3	(1226)
2. BCSSTK-13	0.2	57	(0.1)	54	0.6	64	0.6	(53)
3. BCSSTK-15	0.3	173	(0.1)	169	1.1	101	1.1	(89)
4. BCSSTK-25	1.0	325	(0.5)	354	3.5	458	3.3	(284)
5. BCSSTK-29	0.9	516	(0.3)	457	5.7	492	4.7	(435)
6. BCSSTK-30	1.4	1060	(0.7)	(945)	19.4	1502	4.2	1132
7. BCSSTK-31	2.1	2651	(0.9)	2579	13.8	1615	5.5	(1410)
8. BCSSTK-32	2.1	1262	(0.8)	(953)	22.2	1995	4.5	1630
9. COMP-1	662.9	2676	337.5	2753	—	—	(16.6)	(2181)
10. COPTER-2	3.6	11741	(2.7)	12225	13.5	6812	12.5	(5584)
11. CUBE-35	2.5	14251	(1.7)	14198	7.4	9692	6.8	(8788)
12. CRONE	(0.4)	26	0.5	24	2.8	27	2.6	(23)
13. DWT0512	0.01	0.05	(0.01)	(0.05)	0.03	0.06	0.03	0.06
14. FLEET-12	90.0	6294	36.8	4926	—	—	(3.0)	(3307)
15. GISMONDI	11.5	303770	(2.7)	304937	4.3	133882	5.6	(128431)
16. GRID-127	(0.2)	48	0.3	(46)	2.5	49	2.1	52
17. HSCT16K-A	1.0	772	(0.3)	(720)	7.3	830	4.0	792
18. HSCT16K-B	1.5	1093	(0.4)	1131	9.2	952	7.1	(609)
19. HSCT22K	4.2	1358	(0.9)	(1244)	19.4	2456	7.2	1872
20. HSCT44K	5.5	(2257)	(1.1)	2276	39.8	5046	4.6	3576
21. HSCT88K	8.8	12975	(1.4)	10872	41.8	12637	6.7	(9871)
22. K-8	14.7	2153	15.4	2460	8.8	1335	(3.1)	(492)
23. KEN-18	38.6	(186)	14.1	204	—	—	(6.2)	204
24. KK-6	6601.2	(1872)	2371.5	2054	—	—	(6.4)	1881
25. PDS-20	393.4	7069	(5.9)	7907	6.9	5683	7.1	(2130)
26. PILOT	0.9	47	(0.2)	50	0.8	50	0.7	(39)
27. USAIR	24.3	15486	(2.7)	15225	3.2	17073	3.7	(6091)

Table 2 compares two-way partitions using METIS and WGPP. WGPP produced a smaller edge-cut for each graph and was much faster than METIS—about 2.64 times on average. **Table 3** gives the run times and edge-cuts for three different variations of METIS, and **Table 4** gives the run times and edge-cuts for three different variations of WGPP. The (HTM + BKL) variation of WGPP compares quite favorably with all three variations of METIS. It is much faster and generates partitions with a smaller edge-cut for most graphs. However, it is

significantly worse on one graph, HSCT16K-B; as a result, the total edge-cut of all eight graphs (250644) is about 7% worse than that of the best total of METIS (234308). **Table 5** compares METIS and WGPP for 160-way partitions. WGPP turns out to be more than three times as fast as METIS on average for 160-way partitions, while producing edge-cuts that are a few percent higher.

5.2 Sparse-matrix ordering

Table 6 compares MMD, AMD, METIS, and WGPP for the ordering time and the number of floating-point operations (in millions) on 27 sparse matrices from various sources. The best time and ordering for each matrix are enclosed in parentheses. AMD is the fastest of all orderings for a majority of the problems, but its run time is very inconsistent for the linear programming matrices. In the worst case, it is 370 times slower than WGPP. In this suite of test problems, WGPP produces the best ordering for two thirds of the matrices. It produces less than half the operation count of its nearest rival (AMD) in 5 of the 27 problems, but generates only 1.7 times the operation count of AMD in the worst case. On the whole, WGPP is quite competitive with the other three ordering codes, markedly so for the linear programming problems.

6. Concluding remarks

This paper presents heuristics that improve the run time and the quality of the state-of-the-art practical methods for graph-partitioning and sparse-matrix ordering. We have developed a graph-partitioning and sparse-matrix-ordering package (WGPP) [5] based on the heuristics described in this paper. For graph partitioning, WGPP is considerably faster than the well-known package METIS, while generating partitions of almost comparable quality. A comparison of WGPP with three other widely used sparse-matrix-ordering codes shows it to be the best in quality and consistent in run time on a suite of 27 randomly selected sparse matrices.

References

1. M. Yannakakis, "Computing the Minimum Fill-In Is NP-Complete," *SIAM J. Alg. Discrete Meth.* **2**, 77–79 (1981).
2. Alan George and Joseph W.-H. Liu, "The Evolution of the Minimum Degree Ordering Algorithm," *SIAM Rev.* **31**, 1–19 (March 1989).
3. Alan George and Joseph W.-H. Liu, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1981.
4. Timothy A. Davis, Patrick Amestoy, and Iain S. Duff, "An Approximate Minimum Degree Ordering Algorithm," *Technical Report TR-94-039*, Computer and Information Sciences Department, University of Florida, Gainesville, FL, 1994.
5. Anshul Gupta, "WGPP: Watson Graph Partitioning (and Sparse Matrix Ordering) Package: Users' Manual," *Technical Report RC-20453 (90427)*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, May 6, 1996.
6. Anshul Gupta, "Graph Partitioning Based Sparse Matrix Ordering Algorithms for Finite-Element and Optimization Problems," *Proceedings of the Second SIAM Conference on Sparse Matrices*, October 1996.
7. Bruce Hendrickson and Edward Rothberg, "Improving the Runtime and Quality of Nested Dissection Ordering," *Technical Report SAND96-0868J*, Sandia National Laboratories, Albuquerque, NM, 1996.
8. Cleve Ashcraft and Joseph W.-H. Liu, "Robust Ordering of Sparse Matrices Using Multisection," *Technical Report CS 96-01*, Department of Computer Science, York University, Ontario, Canada, 1996.
9. George Karypis and Vipin Kumar, "METIS: Unstructured Graph Partitioning and Sparse Matrix Ordering System," Technical report, Department of Computer Science, University of Minnesota, Minneapolis, 1995.
10. George Karypis and Vipin Kumar, "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs," *Technical Report TR 95-035*, Department of Computer Science, University of Minnesota, Minneapolis, 1995.
11. Edward Rothberg and Bruce Hendrickson, "Sparse Matrix Ordering Methods for Interior Point Linear Programming," *Technical Report SAND96-0475J*, Sandia National Laboratories, Albuquerque, NM, 1996.
12. Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis, *Introduction to Parallel Computing: Design and Analysis of Algorithms*, Benjamin Cummings/Addison Wesley, Redwood City, CA, 1994.
13. George Karypis and Vipin Kumar, "Parallel Multilevel Graph Partitioning," *Technical Report TR 95-036*, Department of Computer Science, University of Minnesota, Minneapolis, 1995.
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. 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, 1994. To appear in *IEEE Transactions on Parallel and Distributed Systems*, 1997. Postscript file available via anonymous FTP from the site *ftp://ftp.cs.umn.edu/users/kumar*.
16. George Karypis, Anshul Gupta, and Vipin Kumar, "Parallel Formulation of Interior Point Algorithms," *Technical Report 94-20*, Department of Computer Science, University of Minnesota, Minneapolis, April 1994. Shorter versions appear in *Supercomputing '94 Proceedings* and *Proceedings of the 1995 DIMACS Workshop on Parallel Processing of Discrete Optimization Problems*.
17. Anshul Gupta and Vipin Kumar, "Parallel Algorithms for Forward and Back Substitution in Direct Solution of Sparse Linear Systems," *Supercomputing '95 Proceedings*, December 1995.
18. T. Bui and C. Jones, "A Heuristic for Reducing Fill in Sparse Matrix Factorization," *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, 1993, pp. 445–452.
19. Bruce Hendrickson and Robert Leland, "A Multilevel Algorithm for Partitioning Graphs," *Supercomputing '95 Proceedings*, 1995. Also available as *Technical Report SAND93-1301*, Sandia National Laboratories, Albuquerque, NM, 1993.
20. B. W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *Bell Syst. Tech. J.*, 1970.

21. C. M. Fiduccia and R. M. Mattheyses, "A Linear Time Heuristic for Improving Network Partitions," *Proceedings of the 19th IEEE Design Automation Conference*, 1982, pp. 175–181.
22. W. J. Camp, S. J. Plimpton, B. A. Hendrickson, and R. W. Leland, "Massively Parallel Methods for Engineering and Science Problems, *Commun. ACM* **37**, 31–41 (April 1994).
23. George Karypis and Vipin Kumar, "Multilevel k-way Partitioning Scheme for Irregular Graphs," *Technical Report TR 95-064*, Department of Computer Science, University of Minnesota, Minneapolis, 1995.
24. Alan George, "Nested Dissection of a Regular Finite-Element Mesh," *SIAM J. Numer. Anal.* **10**, 345–363 (1973).
25. R. J. Lipton and R. E. Tarjan, "A Separator Theorem for Planar Graphs," *SIAM J. Appl. Math.* **36**, 177–189 (1979).
26. R. J. Lipton, D. J. Rose, and R. E. Tarjan, "Generalized Nested Dissection," *SIAM J. Numer. Anal.* **16**, 346–358 (1979).
27. Anshul Gupta, "Graph Partitioning Based Sparse Matrix Ordering Algorithms for Interior-Point Methods," *Technical Report RC-20467 (90480)*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, May 21, 1996. Available on the WWW at the *IBM Research CyberJournal* site at *http://www.research.ibm.com:8080/*.
28. Alan George, Joseph W.-H. Liu, and Esmond G.-Y. Ng, "Communication Reduction in Parallel Sparse Cholesky Factorization on a Hypercube," *Hypercube Multiprocessors 1987*, M. T. Heath, Ed., SIAM, Philadelphia, 1987, pp. 576–586.
29. Iain S. Duff and Torbjorn Wiberg, "Remarks on Implementations of $O(n^{1/2}\tau)$ Assignment Algorithms," *ACM Trans. Math. Software* **14**, 267–287 (1988).
30. Alex Pothen and C.-J. Fan, "Computing the Block Triangular Form of a Sparse Matrix," *ACM Trans. Math. Software*, 1990.
31. Cleve Ashcraft and Joseph W.-H. Liu, "Generalized Nested Dissection: Some Recent Progress," *Proceedings of the Fifth SIAM Conference on Applied Linear Algebra*, Snowbird, UT, June 1994.
32. Cleve Ashcraft, "Compressed Graphs and the Minimum Degree Algorithm," *SIAM J. Sci. Computing* **16**, 1404–1411 (1995).

Received July 9, 1996; accepted for publication October 8, 1996

Anshul Gupta *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (anshul@watson.ibm.com).* Dr. Gupta received the B.Tech. degree from the Indian Institute of Technology, New Delhi, in 1988 and the Ph.D. in 1995 from the University of Minnesota, both in computer science. His research interests include parallel algorithms, scalability and performance analysis of parallel systems, sparse-matrix computations, and applications of parallel processing in scientific computing and optimization. He has coauthored several journal articles and conference papers on these topics and a book entitled *Introduction to Parallel Computing* (Benjamin Cummings/Addison Wesley, 1994).