RC 20467 (90480) May 21, 1996
Computer Science/Mathematics

IBM Research Report

Graph Partitioning Based Sparse Matrix Orderings for Interior-Point Algorithms

Anshul Gupta

IBM Research Division
T.J. Watson Research Center
P. O. Box 218
Yorktown Heights, New York 10598

anshul@watson.ibm.com

IBM Research Division
Almaden · Austin · China · Haifa · T.J. Watson · Tokyo · Zurich

Graph Partitioning Based Sparse Matrix Orderings for Interior-Point Algorithms

Anshul Gupta

IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598

anshul@watson.ibm.com

IBM Research Report RC 20467

May 21, 1996

Abstract

The key computational step in interior-point (IP) methods for solving linear programming (LP) problems is the solution of a sparse symmetric system of linear equations $(AD^2A^T)\Delta y = b$. The time and the memory requirements of this step depend on the initial ordering of rows and columns in the symmetric matrix AD^2A^T. Computing an optimal ordering for sparse matrix factorization is an NP-complete problem and developing fast and effective ordering heuristics has been a subject of research for almost three decades. Currently, a heuristic known as *multiple minimum degree (MMD)* (or one of its variants) is almost universally employed by the LP community while using IP methods. In this paper, we show that a completely different approach to sparse matrix ordering that is based on graph partitioning is significantly more suitable for the matrices arising in an IP computation than MMD or its variants. Experiments with our ordering algorithm show a cumulative speedup by a factor of 2.2 and an average speedup of 1.45 over a minimum-degree based ordering for solving a comprehensive suite of real optimization problems. In addition, our graph-partitioning based ordering algorithm is more parallelizable than minimum-degree based orderings algorithm and it renders the ordered matrix more amenable to parallel factorization.

Keywords: Linear Programming, Interior-Point Methods, Sparse Matrices, Ordering, Graph Partitioning, Sparse Linear Systems.

1 Introduction

Over the past decade, interior-point (IP) or barrier methods [20, 13, 31, 29, 32, 30] have become increasingly popular for solving medium- to large-size linear programming (LP) problems. IP algorithms are iterative in nature and the most time-consuming computational step in each iteration is solving a (typically) sparse symmetric positive definite system of linear equations of the form $(AD^2A^T)\Delta y = b$. Solving this system using direct methods involves four steps [9], namely, ordering, symbolic factorization, numerical factorization, and triangular solutions. Sparse matrix factorization derives its efficiency from avoiding arithmetic operations with zeros, which constitute the bulk of the matrix. During the process of factorization, some of the original zero entries in the matrix are overwritten by a nonzero value. This phenomenon is know as *fill-in*. The time and memory requirements of numerical factorization and triangular solutions depend on the amount of fill-in, which is a function of the initial ordering of row and columns of the sparse symmetric matrix AD^2A^T. The structure of the coefficient matrix AD^2A^T remains the same in each iteration of an IP algorithm. Therefore, a single ordering and symbolic factorization suffices for numerical factorization in each iteration. For a majority of sparse matrices, numerical factorization is the most time-consuming of the four steps. However, for some LP problems, the ordering time using a conventional ordering method can be comparable to or exceed the total factorization time of all iterations. Therefore, depending on the problem at hand, the overall run time of an IP algorithm is dominated by either ordering or numerical factorization or both. In any case, the choice of a good ordering technique is crucial to the efficiency of the entire solution process. In this paper, we describe ordering heuristics that offer significant improvement over traditional ordering heuristics for LP problems, both in terms of ordering time and quality.

In addition to IP methods, computing a fill-reducing ordering or permutation of the rows and columns of a sparse matrix is an important problem in any application that uses a direct method for solving large sparse systems of linear equations. Finding an optimal ordering is an NP-complete problem [36] 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 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 [10] 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 [9, 10] and the Approximate Minimum Degree (AMD) algorithm by Davis, Amestoy, and Duff [5] represent the state of the art in MD based heuristics. Until now, with the exception of Rothberg and Hendrickson [35], most researchers have focused on ordering sparse matrices arising in finite-element applications and these applications have guided the development of the ordering heuristics. Extensive use of the interior-point methods for solving LP problems is relatively recent. As a result, the linear programming community has been using these well established heuristics that were not originally developed for their applications. This paper shows that MD based heuristics can be quite unsuitable for many linear programming problems.

Recent work by the author [14], Hendrickson and Rothberg [19], Ashcraft and Liu [1], and Karypis and Kumar [23, 22] 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. In this paper, we present a suite of algorithms that enhance the GP based ordering approach to generate robust orderings of sparse matrices arising in LP problems. We show that, with our modifications, the benefits of GP based ordering over MD based ordering are even more pronounced for LP problems than for finite-element problems. In [35], Rothberg and Hendrickson independently presented evidence of the superiority of GP based ordering methods for LP problems using quite different graph partitioning methods. However, they use conventional graph coarsening methods for partitioning that occasionally fail for LP problems [34]. The algorithms presented in this paper are robust and generate good quality orderings for both LP and finite-element problems. These algorithms are included in the graph partitioning and sparse matrix ordering package (WGPP) [15][1] developed by the author. In [14], we present experimental results of using WGPP for graph partitioning and for ordering sparse matrices arising in different applications. In this paper, we show that WGPP offers significant advantages over MD based heuristics for the matrices arising in IP computations. For such matrices, WGGP produces better quality orderings (on an average) much faster than AMD or MMD (unlike finite element matrices, where the run time of WGPP is higher than that of its MD based counterparts). Section 3 shows that replacing an MD based ordering by the WGPP ordering reduced the total solution time of a suite of 52 medium to large LP problems by a factor of 2.2.

Both MMD and AMD algorithms rely heavily on the concepts of mass elimination and indistinguishable nodes [10, 5] in order to achieve their run time efficiency. These concepts work very well for finite-element graphs, but may not be very effective for many relatively unstructured LP problems. We have observed that for sparse matrices of similar size (in terms of number of nonzero entries), the run time of MMD and AMD is a decreasing function of the average supernode size in the factor (larger supernodes mean more successful mass elimination). A supernode is a set of consecutive columns in a triangular factor of a sparse matrix with identical off-diagonal nonzero structure. LP matrices typically have much smaller supernodes than finite-element matrices. The run time of WGPP is independent of the size of the supernodes in the factor and depends solely on the number of nonzeros in the original sparse matrix. As a result, WGPP has been observed to be faster than AMD by a factor of several hundred on some of our test problems with very small or no supernodes (Table 1). WGPP achieves this improvement in ordering time without compromising the quality of ordering. In fact, our experiments show that WGPP produces orderings that require less work in factorization on an average on a large suite of randomly selected LP matrices (Tables 1 and 2).

In addition to generating better quality orderings faster, GP based methods are more suitable for distributed-memory parallel computers than MD based methods. The only attempt to perform a minimum degree ordering in parallel that we are aware of [12] was not successful in reducing the ordering time over a serial implementation. On the other hand, there is a strong theoretical and experimental evidence of efficient parallel algorithms for graph partitioning and sparse matrix orderings based on graph partitioning [24]. GP based ordering also aids the parallelization of the factorization and triangular solution phases of a direct solver. Gupta, Karypis, and Kumar [16, 21] have proposed a highly scalable parallel formulation of sparse Cholesky factorization that 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 is used to preorder the sparse matrix.

The remainder of the paper contains two major sections. In Section 2, we describe the overall ordering process and the heuristics used in its various stages. In Section 3, we present experimental results comparing the WGPP orderings with the orderings produced by the state-of-the-art MD based heuristics. In Appendix A, we graphically show the effect of our ordering on the nonzero pattern of some LP matrices.

2 The Multiple Multilevel Recursive Bisections Heuristic

The complete ordering algorithm involves several stages, a preprocessing step, and a postprocessing step. In Sections 2.1 and 2.2, we describe the core algorithm and its various stages. Section 2.3 describes the preprocessing step that has a significant impact on the quality and the run time of ordering. Section 2.4 describes a postprocessing step that may aid certain factorization implementations. Figure 7 illustrates the overall ordering procedure for LP matrices by means of a flowchart.

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

2.1 Graph Bisection

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 [8]. The graph is bisected by finding and removing a node-separator, the nodes of the two resulting disconnected subgraphs are labeled by applying the same technique recursively, and the nodes of the separator are labeled after the nodes of the subgraphs have been labeled (i.e. the separator nodes get higher labels than any of the nodes of the two subgraphs). The recursion terminates when the subgraphs become too small, at which stage they are labeled using a minimum degree heuristic.

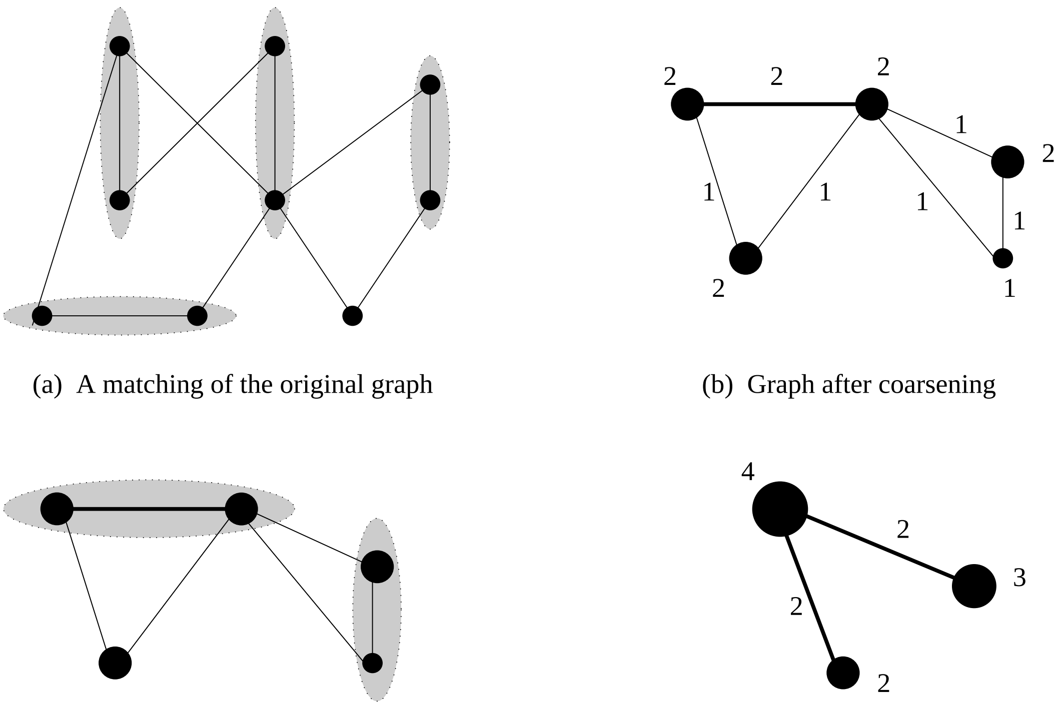
The reason why this technique works is that, upon elimination, a column corresponding to any of the nodes in the two subgraphs can create a fill only in the columns corresponding to the separator nodes and not any node of the other subgraph. Intuitively, smaller node-separators mean less fill and less work during factorization. For finite-element graphs with certain nice properties, it can be proved that this technique yields orderings within a constant factor of the optimal [28, 27, 8, 9]. Such bounds cannot be proved for the unstructured LP matrices. However, as we show in Section 3, with suitable modifications, the strategy works quite well in practice even for matrices with arbitrary sparsity patterns. Appendix A shows the variety of sparsity patterns of matrices for which WGPP produced orderings significantly better than the best MD based heuristics. 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 [16, 21, 11].

A key step in our ordering algorithm is finding a small node bisector of a graph. Recent research [3, 18, 22, 14] has shown *multilevel algorithms* to be fast and effective in computing graph partitions. A typical multilevel graph partitioning algorithm has four components, namely *coarsening*, *initial partitioning*, *uncoarsening*, and *refining*. In [14], we describe in detail the general heuristic to partition a graph into k parts. In the following subsections, we briefly discuss the components of graph partitioning in the context of bisection (partioning into two parts) with a node-separator.

2.1.1 Coarsening

The goal of the coarsening phase is to reduce the size of a graph while preserving its properties that are essential to finding good a partition. Using the appropriate coarsening algorithm is very important for robust and good quality graph partitioning. For coarsening, 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 in this set have a node in common. This set of edges is known as a *matching*. The edges in this set are then removed. The two nodes that an edge in the matching connects are collapsed into a single node. This composite node is assigned a weight that is the sum of the weights of the two component nodes. A coarsening step usually reduces the number of nodes in the graph by nearly half. 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 [18] known as *random matching (RM)* randomly chooses pairs of connected unmatched nodes to include in the matching. In [22], Karypis and Kumar describe a heuristic known as *heavy-edge matching (HEM)*. HEM not only helps reduce the run time of the refinement component of graph partitioning, but also tends to generate partitions with small separators. The HEM strategy randomly picks an unmatched node, selects the edge with the highest weight among the edges incident on this node that connect it to other unmatched nodes, and marks both nodes 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,



(a) A matching of the original graph (b) Graph after coarsening

(c) A matching of the coarse graph (d) After 2 steps of coarsening

Figure 1: Illustration of graph coarsening.

by absorbing the heavier edges, generates coarse graphs whose nodes are loosely connected (by the lighter remaining edges). This ensures 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 in random order 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. HEM was originally devised for finite-element graphs, for which it works reasonably well because the range of edge weights is usually small. However, in the graphs of LP matrices, there is a large variation among the edge weights (especially in the later stages of coarsening) and heavy edges left out from the matching adversely affect the quality of coarsening. 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 the decreasing order of their weights to inspect them for their possible inclusion in the matching. Ties are broken randomly. The benefit of heaviest-edge matching over Karypis and Kumar's HEM are more pronounced in the later stages of coarsening and that is where sorting is not too expensive because the graph has shrunk considerably from its original size.

Even the modified HEM 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 2). 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 a 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

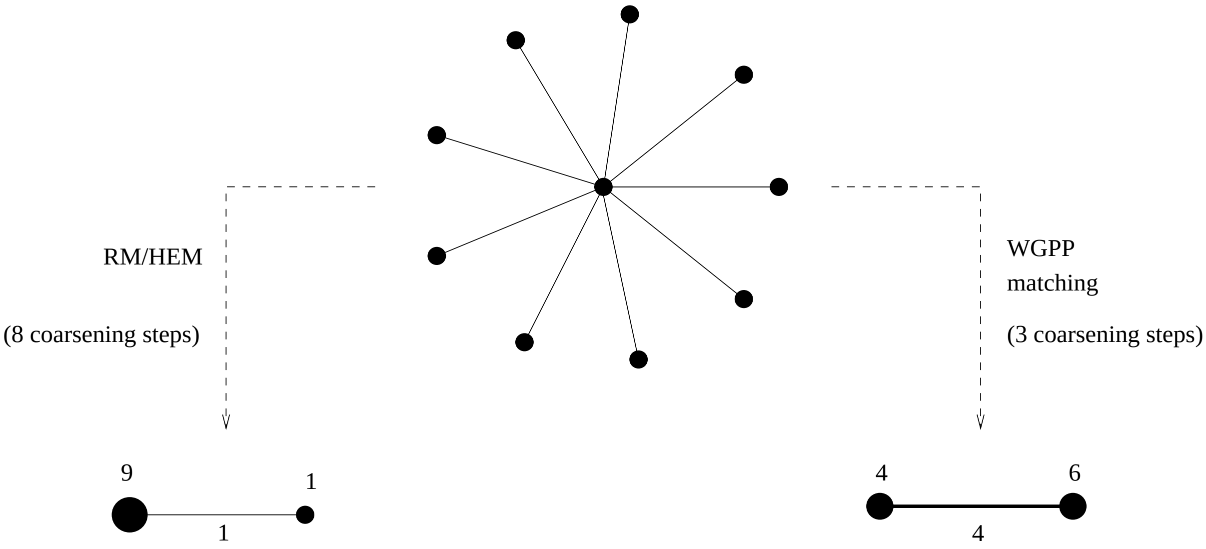


Figure 2: Difference between the 2-node coarse graph produced by RM or HEM and that generated by the WGPP matching algorithms when the original graph has a "star" structure.

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. It is clear from the above equation that 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.

2.1.2 Initial partitioning

In [14], we show that the initial partitioning phase can be completely eliminated, thereby simplifying and speeding up the overall partitioning algorithm. 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. One of the effective heuristics for initial partitioning is *graph growing* [22]. 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 cumulative weight of the tagged nodes is roughly equal to the cumulative weight of the untagged nodes. The set of tagged and untagged nodes now form the two partitions of the graph. In [14], we discuss that graph growing is the bottom-up equivalent of coarsening. We continue to coarsen the graph until it contains only two nodes, which constitute a natural initial partition.

2.1.3 Uncoarsening and refinement

The uncoarsening and refining components of graph partitioning work together. The uncoarsening and refining phase of WGPP is quite different from most other multilevel graph partitioning algorithms. Initially, the two 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

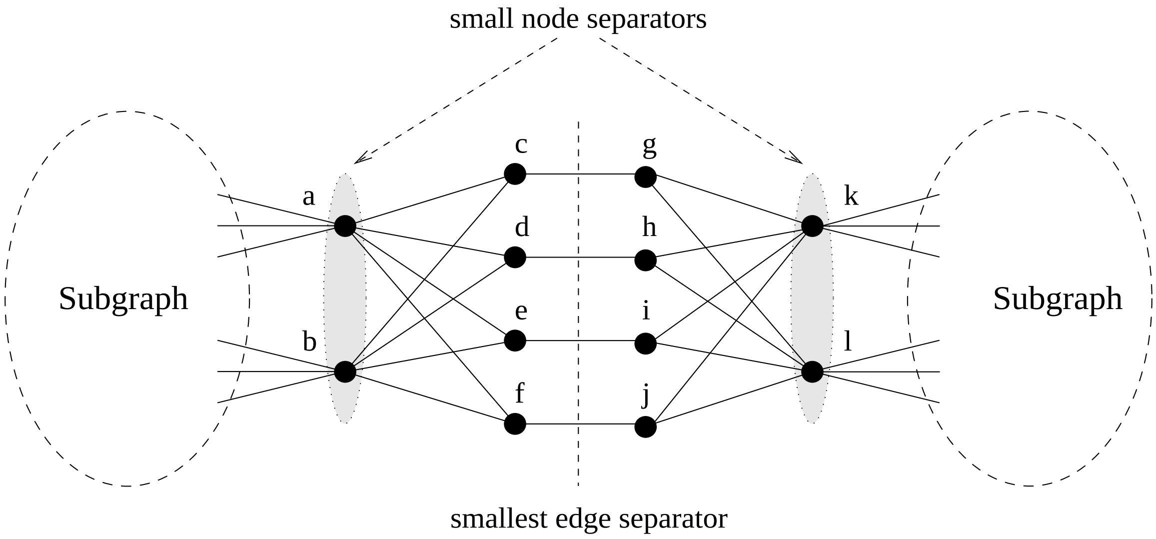


Figure 3: Drawbacks of choosing a node-separator from the minimum edge-separator.

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. A subset of the nodes which the separator edges are incident on is a node-separator if (and only if) this subset covers all the edges of the separator. The removal of an edge- or a node-separator from the graph breaks it into two 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 two parts too imbalanced. While refining a node-separator, nodes are moved between the separator and the two subgraphs partitioned by it in an attempt to minimize the total weight of the separator nodes.

The ultimate goal of the graph bisection phase of ordering is to find a small node-separator. Current GP based ordering algorithms follow two different approaches to finding a small node-separator. A conventional approach [9, 23] refines 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 an algorithm for finding a minimum cover [6, 33] is used 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 not too unreasonable for the relatively structured finite-element meshes, but usually does not work well for LP matrices. For example, consider the graph in Figure 3. 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.

Another approach [2, 1, 19] is to find a node-separator within the coarse graph and refine it into a node-separator of the original graph in order to overcome the drawbacks of working with an edge-separator. However, even this approach is also not completely free of pitfalls. In Figure 4, we show that the heavier node-separator in the coarse graph may result in the 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 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 4, the smaller

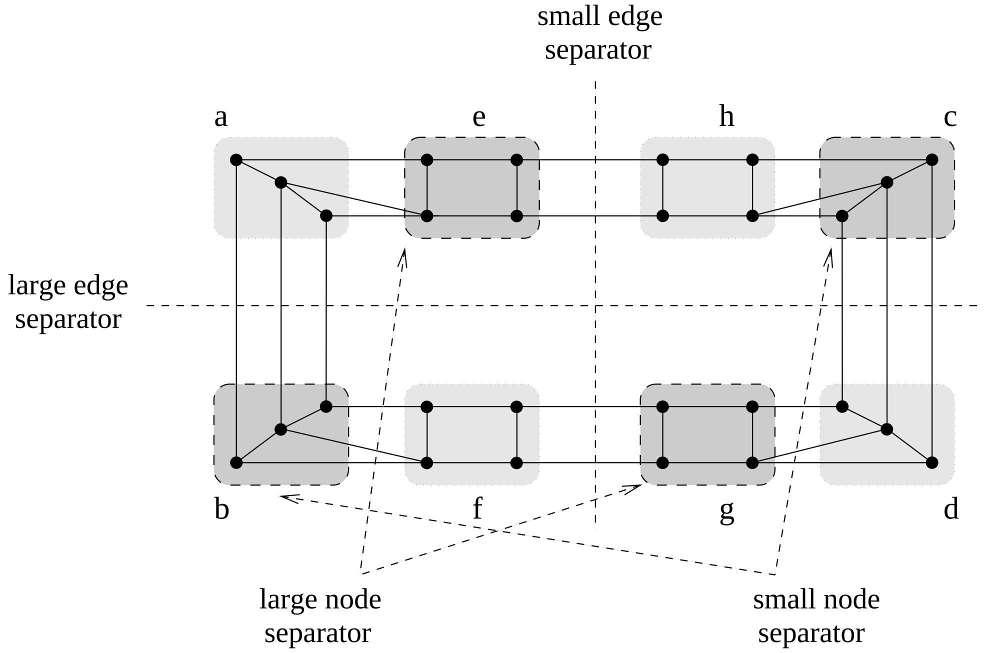


Figure 4: Drawbacks of choosing a small node-separator for refining in a coarse graph.

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 inter-partition 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 4 nodes. This is because two of the four component nodes of both coarse nodes c and g have no inter-partition 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. Unlike in the original graph, the size of the edge-separator is not completely irrelevant in coarse versions of the graph.

With these motivations, we now present our uncoarsening and refining scheme, which is illustrated in Figure 5. We first perform as many steps of coarsening on the original n-node graph as required to coarsen it down to two nodes. We then uncoarsen the graph to q nodes, where $2 < q < n$, while refining the edge-separator after each step of uncoarsening. At this stage we save the partition, coarsen the q-node graph back to two 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 chose 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 edge-separator, the size of the node-separator, and the inverse of the imbalance between the two parts. Unlike finite-element graphs, the graphs from the unstructured LP problems usually show large variations in the quality of different partitions of the same graph. The randomization in the coarsening process ensures that the different trials result in sufficiently different partitions. Selecting the best amongst many partitions goes a long way in finding a good separator for the original graph. Most of the partitioning 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.

Finally, 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 edge-separators, we use the popular Fiduccia-Mattheyses [7] heuristic, which is a linear time version of the Kernighan-Lin [25]

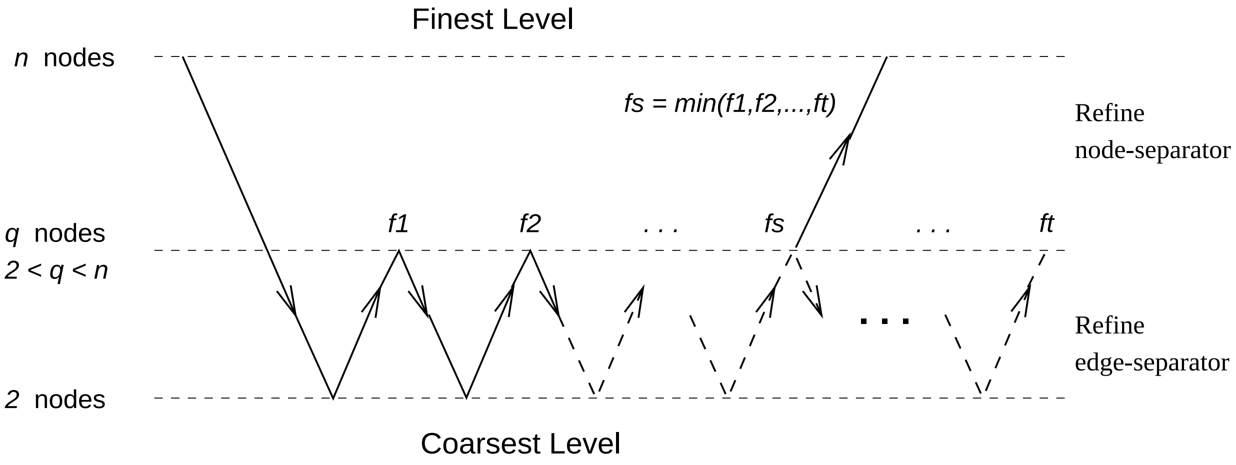


Figure 5: An overview of our uncoarsening and refining scheme for finding a node-separator from the best amongst multiple coarse bisections. f_i is a weighted average of the node-separator size, edge-separator size, and the inverse of imbalance between the two parts in the i-th bisection.

heuristic. In [2], Ashcraft and Liu extended the Fiduccia-Mattheyses framework to refine node-separators as well. We use a similar approach when node-separators are refined in the final steps of uncoarsening.

2.2 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 few dozen 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 [5]) 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.

2.3 Preprocessing

Sections 2.1 and 2.2 describe the process of labeling the nodes of the graph corresponding to a sparse matrix. In this section, we describe a simple preprocessing step that, if performed before subjecting the graph to the labeling process described previously, improves both the run time and the quality of the ordering heuristic. Most matrices of the form AD^2A^T that need to be factored while solving an LP problem using a barrier method have a few (typically about 2%) the columns that are much denser than the remaining columns. The number of nonzeros in these few columns is typically several times more 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 from the graph. The remainder of the graph is labeled using the heuristics described in Sections 2.1 and 2.2 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 loosely connected than the original graph. This aids the partitioning algorithm in quickly

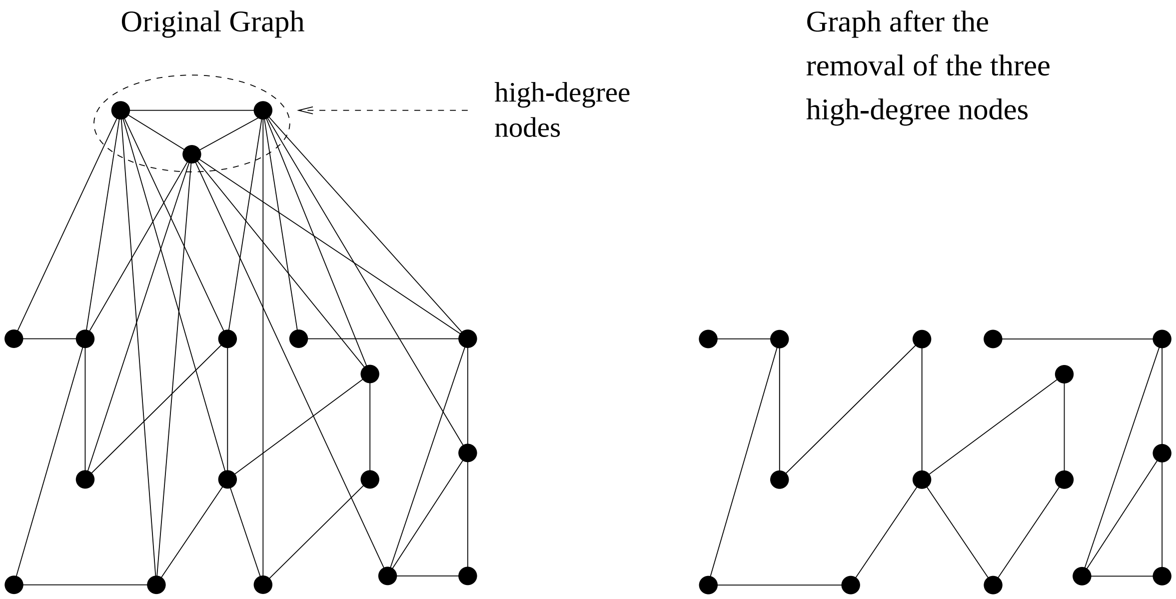


Figure 6: Illustration of the preprocessing step performed in WGPP before ordering an LP matrix.

finding node-separators of a small (or zero) size.

2.4 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. Towards 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 advantages of dense factorization, such as, 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 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, typically separators 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 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 receives higher labels than the nodes of the two bisectors of the two resulting subgraphs.

This relabeling of separator nodes does not effect the total amount of fill-in during factorization or its operation count. It only effects the location of fill-in and attempts to direct most of the fill-in towards the

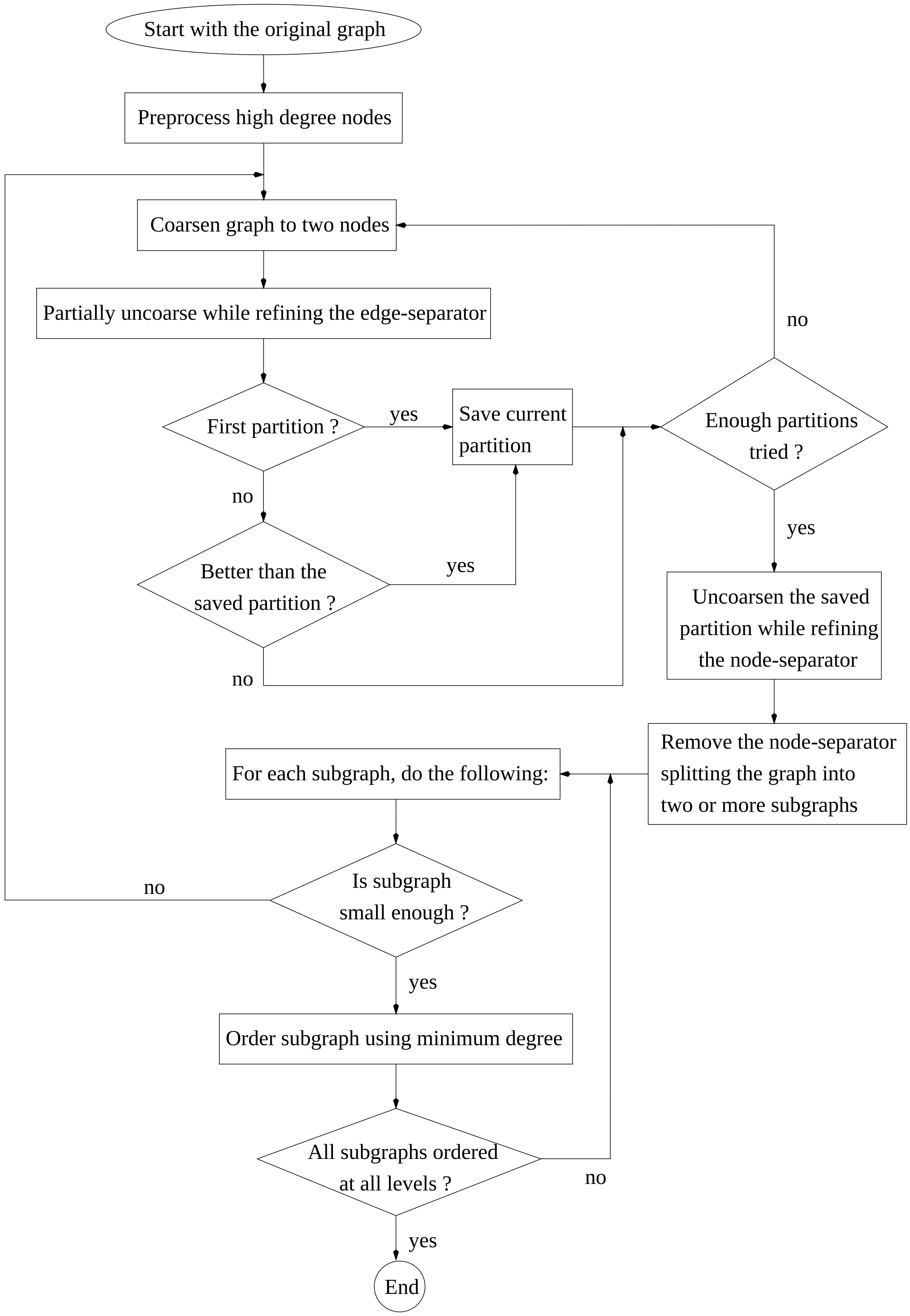


Figure 7: An overview of the ordering algorithm for LP matrices in WGPP.

high-index columns of the matrix. The figures in Appendix A depict the effect of WGPP ordering on some LP matrices.

3 Experimental Results

In this section, we present detailed experimental comparison of the WGPP ordering based on multiple multi-level recursive bisection with the best minimum-degree based ordering algorithms currently in use for solving LP problems using barrier methods. The other two graph-partitioning based ordering softwares that we are aware of, are Metis [23] and DDSEP [1]. Both of these were designed primarily for finite-element problems. Of these, only Metis is available at the time of writing this paper, and it does not work for a majority of LP problems.

3.1 Comparison with MMD and AMD

In Table 1, we compare three codes, namely MMD by George and Liu [9], AMD by Davis, Amestoy, and Duff [5], and WGPP [15] described in this paper on a dozen large LP problems. The table reports the run times of the three ordering codes on an IBM RS6000/590 workstation. All three were compiled with the highest level of optimization (-O3 option) of XLF 3.2 Fortran compiler. The table also reports the quality of ordering in terms of the operation count in Cholesky factorization and the average supernode size. From Table 1, it can be observed that, on an average, MMD produces slightly better ordering than AMD, but takes significantly more time. The ratios of the AMD operation count to WGPP operation count in the last row of the table show that WGPP produces much better ordering than its MD based counterparts for these problems. The run time of WGPP relative to AMD depends on the average supernode size. For matrices with larger supernodes (e.g., GRADE, GSMND, PILOT), AMD is more than twice as fast as WGPP. However, for matrices with very small or no supernodes (e.g., KK3, KK6, OSA60), WGPP can be several hundred times faster than AMD.

Although it so happens that WGPP ordering reduces the factorization operation count over AMD for all the matrices in Table 1, more extensive experiments (Table 2) show that there are some case in which WGPP results in a higher fill-in and operation count that an MD based ordering. The dependence of the run time of MD based heuristics on the average supernode size can be exploited to improve the ordering in many such cases. From Table 1, one can observe that the average supernode size is a property of the matrix and it shows only marginal variations for different orderings of the same matrix. Therefore, after performing the GP based ordering and computing the resulting average supernode size, it is possible to predict whether or not an MD based heuristic will be fast for that matrix. If the average supernode size is large, then an MD based ordering can be attempted and the better of the two orderings can be used for factorization. This would result in only a slight increase in the ordering time because the second ordering (minimum degree) is attempted only when its run time is known to be small. WGPP has a parameter by which the user can switch the option of attempting the second ordering on or off. When this option is on, WGPP attempts the second ordering if the average supernode size is large enough. Thus, for the class of problems for which minimum degree is both fast and better than the graph partitioning based ordering, WGPP returns the permutation corresponding to the former.

3.2 Impact on IBM Optimization Subroutine Library

The IBM Optimization Subroutine Library (OSL) [26] is a collection of high-performance mathematical subroutines for use by application programs that solve optimization problems. In addition to the traditional Simplex method [4], OSL includes subroutines that implement all three popular algorithmic approaches for the interior-point (barrier) methods, namely, the primal barrier method [13], the primal-dual method [31, 29], and the predictor-corrector method [32, 30]. WGPP contains subroutines that replace OSL's internal minimum-

Matrix →		GRADE	COMP1	F12	GSMND	K8	KEN18	KK3	KK6	OSA60	PDS20	PILOT	USAIR
Dimension (N)		21601	16783	20922	17447	21059	39919	31802	62064	10209	28062	1441	13564
Nonzeros ($NNZ_A \div 10^3$)		2829.0	9323.4	379.2	475.8	425.5	385.2	2164.2	4248.3	53.8	286.0	120.5	181.6
Density (NNZ_A/N)		131.0	555.5	18.1	27.3	20.2	9.7	68.1	68.5	5.3	10.2	83.6	13.4
M	Runtime (sec.)	20.7	662.9	90.0	11.5	14.7	38.6	1323.4	6601.2	105.9	393.4	0.9	24.3
M	Avg. supnode size	3.212	1.200	1.366	2.415	1.263	1.209	1.027	1.030	1.000	1.250	2.203	1.461
D	Opcount ($\div 10^6$)	1438	2676	6294	303770	2153	186	270	1872	0.1103	7069	47	15486
A	Runtime (sec.)	8.1	337.5	36.8	2.7	15.4	14.1	880.3	2371.5	27.6	5.9	0.2	2.7
M	Avg. supnode size	3.230	1.215	1.370	2.394	1.278	1.179	1.026	1.030	1.000	1.246	2.231	1.465
D	Opcount ($\div 10^6$)	1334	2753	4926	304937	2460	204	318	2054	0.1104	7907	50	15225
W	Runtime (sec.)	16.3	16.6	3.0	5.6	3.1	6.2	2.1	6.4	0.1	7.1	0.7	3.7
G	Avg. supnode size	3.383	1.096	1.425	2.645	1.248	1.187	1.023	1.024	1.000	1.341	1.990	1.498
P	Opcount ($\div 10^6$)	1226	2181	3307	128431	492	204	286	1881	0.1104	2130	39	6091
P	$\frac{\text{Opcount}_{AMD}}{\text{Opcount}_{WGPP}}$	1.09	1.26	1.49	2.37	5.00	1.00	1.11	1.09	1.00	3.71	1.28	2.50

Table 1: Ordering time and quality comparisons between MMD, AMD, and WGPP.

Matrix		OSL Ordering			WGPP Ordering			
		NNZ_{OSL} ($\div 10^3$)	nf_{OSL}	T_{OSL} (seconds)	NNZ_{WGPP} ($\div 10^3$)	nf_{WGPP}	T_{WGPP} (seconds)	T_{OSL}/T_{WGPP}
1.	25FV47	31.0	27	5.9	*27.3*	27	*5.5*	1.07
2.	80BAU3B	*39.4*	64	39.0	44.1	52	*34.7*	1.12
3.	AAMAT	303.3	101	574.2	*263.6*	101	*468.7*	1.23
4.	ALLGRADE	4931.0	96	8638.4	*4676.4*	95	*6626.2*	1.30
5.	AMOCO2	318.3	69	245.9	*256.0*	67	*222.9*	1.10
6.	BNL2	81.3	36	22.6	*77.7*	33	*18.1*	1.25
7.	CAPLOAD	*843.4*	62	*838.7*	865.7	63	859.3	0.98
8.	CHEVD4	*42.3*	69	*52.4*	50.7	92	80.9	0.65
9.	COMP1	5648.4	16	6327.1	*5325.9*	16	*3256.3*	1.94
10.	CRED	*222.4*	44	180.0	227.3	44	*156.9*	1.15
11.	D2Q06C	114.2	37	36.6	*83.0*	36	*26.3*	1.39
12.	DBTMIN	3456.7	52	8407.1	*2814.0*	52	**3608.1**	2.33
* 13.	DFL001	1609.4	42	1740.7	*1577.6*	43	*1692.5*	1.03
14.	EUROPCAR	102.3	37	35.1	*67.2*	39	*26.3*	1.33
15.	F12	5071.1	33	9332.6	*4206.2*	33	**3873.1**	2.41
16.	GISMONDI	50233.2	-	-	*39934.7*	-	-	2.10
* 17.	GRADE2	*160.3*	34	*53.3*	190.4	36	58.6	0.91
* 18.	GRADE4	*267.8*	58	*154.5*	319.7	57	166.4	0.93
* 19.	GREENBEA	*42.8*	67	*28.2*	49.4	63	30.3	0.93
* 20.	GREENBEB	*41.5*	38	*15.5*	47.0	46	20.3	0.77
21.	HMV1	*582.8*	28	*294.2*	619.3	29	321.6	0.91
22.	HMV2	*1204.5*	36	*902.3*	1206.3	38	972.0	0.93
23.	K8	3412.0	64	5921.5	*1829.5*	63	**2249.3**	2.63
24.	KEN18	1825.7	101	2660.2	*1796.3*	101	*1961.2*	1.36
25.	KK3	*2009.7*	85	15020.6	2027.5	81	**3136.1**	4.79
26.	KK6	*6093.6*	107	73648.6	6331.9	101	**16152.6**	4.56
27.	L60A13L1D	1202.3	94	1266.8	*912.3*	93	*844.6*	1.50
28.	L99A13L1D	3799.4	76	3886.9	*2960.7*	90	*2999.4*	1.30
29.	M42	*176.9*	54	*304.6*	177.9	54	310.2	0.98
30.	M44	*191.5*	52	*318.2*	194.1	53	332.0	0.96
31.	MOD31	*835.2*	48	1987.9	1023.0	48	**904.9**	2.20
32.	MOD32	*941.0*	101	3483.9	1190.2	101	*2125.2*	1.64
33.	MICHELIN	*531.1*	49	893.0	599.3	50	*454.3*	1.97
34.	MPS_SLOW	*2718.5*	101	*7057.9*	4254.3	101	9223.4	0.77
35.	OSA60	21.8	32	313.1	21.8	32	*278.0*	1.13
36.	PDS20	7192.0	55	20490.5	*4102.5*	55	**6154.4**	3.33
37.	PEUGEOT	590.4	13	220.6	*587.4*	13	*166.4*	1.33
* 38.	PILOT87	*450.4*	48	*348.6*	458.4	53	350.9	0.99
39.	PILOTS	227.7	33	87.3	*184.8*	35	*75.6*	1.15
40.	RCH0110	*3660.9*	72	*9304.8*	5065.2	72	11093.4	0.84
41.	RTPPS2	3222.0	100	9254.8	*2951.9*	100	*6430.3*	1.44
42.	SHELL10B	58.3	41	20.9	*49.3*	37	*17.6*	1.19
* 43.	SHIP121	*8.7*	20	*4.5*	9.7	20	4.5	1.00
44.	SOR90	92.7	32	23.6	*71.7*	32	*20.5*	1.13
* 45.	STOCFOR3	*253.5*	34	*72.6*	347.5	34	98.8	0.73
46.	TMPROB	*254.5*	63	330.8	305.0	61	*293.9*	1.13
47.	UNILEVER	*178.3*	38	*145.9*	378.7	39	285.4	0.51
48.	UNOCAL	1534.5	105	3834.2	*1495.2*	104	*3300.4*	1.16
49.	USDA2	2619.9	60	8201.9	*2413.5*	58	*7129.6*	1.15
50.	USAIRORIG	11460.7	39	33194.8	*5492.5*	39	**9073.8**	3.66
51.	W01M	*579.0*	101	1835.6	637.1	101	*1729.5*	1.06
* 52.	WORK	*83.0*	19	*25.0*	89.9	22	29.2	0.86
53.	ZS1	0.30	75	*17.8*	*0.25*	86	21.4	0.83

Table 2: Experimental comparison between the minimum-degree based OSL ordering and graph-partitioning based WGPP ordering. For the problems with a $*$, the actual orderings generated by WGPP are based on minimum degree and are better than the ones shown in the table.

	T_{OSL} (seconds)	T_{WGPP} (seconds)	T_{OSL}/T_{WGPP} (ratio)
Total of all problems excluding GISMONDI	242101.7	109771.9	2.21
Total of largest 10 problems	195494.1	73277.6	2.67
Total of smallest 10 problems	235.7	220.1	1.07
Total of OSL's best 10 (lowest ratios)	17941.7	22205.1	0.81
Total of WGPP's best 10 (highest ratios)	168896.6	45606.6	3.70
Average of the ratio T_{OSL}/T_{WGPP}			1.45
Median of the ratio T_{OSL}/T_{WGPP}			1.15

Table 3: Some statistics on total run time with OSL and WGPP orderings on the problems in Table 2 (excluding GISMONDI).

degree based ordering subroutines (see [15] for details) if an application program is linked with both WGPP and OSL libraries.[2]

In Table 2, we compare OSL's run times for solving a large randomly chosen suite of LP problems using IP methods with and without linking WGPP. The problems range in size from a a few seconds to several hours of execution time. The experiments shown in this table were conducted on an IBM RS6000/250 workstation. To facilitate a comparison between MD and GP based orderings, the results in Table 2 were generated with the option of attempting the second ordering turned off. The problems that have a $*$ before their serial numbers actually ran faster and with less fill-in when this option was on. The table shows a set of three columns of numbers for OSL's internal ordering and WGPP ordering. The first column (*NNZ*) for each ordering is the number of nonzeros in the lower triangular factor and is indicative of the quality of ordering. The second column (*nf*) shows the number of iterations required by the barrier method used to solve the problem. This is the number of numerical factorizations that were performed on a symmetric matrix whose row and column permutation was computed once. The third column (T) gives the total run time. Depending on the problem, the total run time for OSL's ordering is dominated either by the ordering time, or numerical factorization time, or both. The total run time for WGPP ordering, which is uniformly fast, is almost always dominated by the numerical factorization time. Symbolic factorization, triangular solution, and computing the product AD^2A^T consume only a small fraction of the total run time for both orderings (unless A is a very narrow rectangular matrix, in which case, computing the product AD^2A^T can be relatively costly). Note that, besides ordering time and ordering quality, the number of iterations also affects the overall solution time. The number of iterations (nf) often varies slightly between the two orderings, but this variation is completely random and it is safe to assume that the effects of this variation cancel out over a large set of test problems. The last column of Table 2 is the ratio of the run times with OSL's internal MD based ordering (T_{OSL}) and WGPP ordering (T_{WGPP}). An entry greater than 1.0 in the last columns indicates that that the problem was solved faster with the WGPP ordering.

Table 3 abstracts some statistics from the run times of the LP problems listed in Table 2 for the two orderings. The entire suite of problems (excluding GISMONDI[3]), was solved about 2.2 times faster with WGPP ordering than with OSL's internal ordering. The advantage of WGPP was more pronounced for the larger problems; its use reduced the total run time for the ten most time-consuming problems by a factor of almost 2.7. At the other end of the spectrum, WGPP resulted in only a marginal speedup by a factor of 1.07 for the ten smallest problems. Since all ordering methods are heuristics, no single one can be guaranteed to be

[2] Current OSL users can use WGPP in this manner and may get enhancement performance from OSL for interior-point methods.

[3] The largest problem of the suite, GISMONDI, could not proceed beyond symbolic factorization (which computes the number of nonzeros in the factor before actual numerical factorization) because there wasn't enough memory to store the factor.

	NNZ_{OSL} $(\div 10^6)$	NNZ_{WGPP} $(\div 10^6)$	NNZ_{OSL}/NNZ_{WGPP} (ratio)
Total of all 53 problems	131.57	110.89	1.19
Total of largest 10 problems	101.55	80.91	1.26
Total of smallest 10 problems	0.45	0.47	0.96
Total of OSL's best 10 (smallest ratios)	9.36	13.17	0.71
Total of WGPP's best 10 (highest ratios)	77.93	55.71	1.40
Average of the ratio NNZ_{OSL}/NNZ_{WGPP}			1.06
Median of the ratio NNZ_{OSL}/NNZ_{WGPP}			1.00

Table 4: Some statistics on the number of nonzeros in the triangular factor with OSL and WGPP orderings on the problems in Table 2.

better than others for all problems. There were several problems for which WGPP resulted in a slower solution. However, an average problem was speeded up by more than 30% and half of the problems experienced close to 15% speedup with the use of WGPP. WGPP increased the cumulative run time of the ten worst cases by 19%, but reduced the cumulative run time by 270% in the ten best cases. Out of 53, there were eight problems for which WGPP cut the solution time in less than half, while there were none for which the solution time increased by more than a factor of two. The run time reduced for 35 problems and increased for 18.

The results in Table 3 capture the effect of both ordering time and ordering quality. In Table 4, we present similar statistics on the fill-in or the number of nonzeros in the triangular factor resulting from the two orderings. This is solely an indication of ordering quality. The reader must note that typically a small reduction (increase) in fill-in results in a large reduction (increase) in the operation count, and hence the run time, of numerical factorization [9]. Table 4 shows that WGPP reduced the fill-in in almost as many problems as the number of problems in which it increased the fill-in. However, most increases were much smaller than most decreases. This is indicated by the fact that the original OSL ordering produced 6% more average and 19% more cumulative fill-in than WGPP over the entire set of test problems. It is also important that the improvements were more significant for larger problems than for smaller ones. For the largest ten problems, the original ordering was 26% worse in terms of fill-in, whereas for the smallest ten problems, WGPP was off by 4%.

The averages shown in Tables 3 and 4 improved further when the option to attempt a second ordering

	T_{OSL} (seconds)	T_{WGPP} (seconds)	T_{OSL}/T_{WGPP} (ratio)
Total of all problems excluding GISMONDI	242101.7	109609.1	2.21
Total of largest 10 problems	195494.1	73277.6	2.67
Total of smallest 10 problems	235.7	216.3	1.09
Total of OSL's best 10 (lowest ratios)	17941.7	22157.7	0.81
Total of WGPP's best 10 (highest ratios)	168896.6	45606.6	3.70
Average of the ratio T_{OSL}/T_{WGPP}			1.47
Median of the ratio T_{OSL}/T_{WGPP}			1.15

Table 5: The statistics on total run time with OSL and WGPP orderings on the problems in Table 2 (excluding GISMONDI) with the WGPP option of attempting a second ordering based on minimum degree turned on.

	NNZ_{OSL} ($\div 10^6$)	NNZ_{WGPP} ($\div 10^6$)	NNZ_{OSL}/NNZ_{WGPP} (ratio)
Total of all 53 problems	131.57	110.89	1.19
Total of largest 10 problems	101.55	80.91	1.26
Total of smallest 10 problems	0.45	0.45	1.00
Total of OSL's best 10 (smallest ratios)	10.36	14.17	0.73
Total of WGPP's best 10 (highest ratios)	77.93	55.71	1.40
Average of the ratio NNZ_{OSL}/NNZ_{WGPP}			1.09
Median of the ratio NNZ_{OSL}/NNZ_{WGPP}			1.03

Table 6: The statistics on the number of nonzeros in the triangular factor with OSL and WGPP orderings for the problems in Table 2 with the WGPP option of attempting a second ordering based on minimum degree turned on.

was turned on. This option reduced fill-in and the overall run time for the 9 problems whose serial numbers are prefixed by a $*$ in Table 2. There was no noticeable increase in the overall run time of the problems for which this option attempted a second ordering (minimum degree) but did not succeed in reducing the fill-in and used the first (the original WGPP) ordering. The overall run time of such problems remained practically unchanged. Tables 5 and 6 show the statistics on the total solution time and fill-in with this option turned on.

4 Concluding Remarks

In this paper, we present strong empirical evidence that graph-partitioning (GP) based sparse matrix ordering is a suitable alternative to the traditional minimum-degree (MD) based ordering techniques for interior-point methods. We describe a GP based ordering heuristic for LP problems that is a part of the more comprehensive WGPP library [15] developed at IBM Research for graph partitioning and ordering different types of sparse matrices. By means of experiments over a large suite of real-life LP problems, we demonstrate the superiority of WGPP ordering over other state-of-the-art ordering softwares in use, in terms of both ordering time and quality. Our experiments show that WGPP ordering is particularly useful for solving large problems. Therefore, we hope that GP based orering will not only help reduce the solution time of large LP problems, but will also render larger problems solvable on today's workstation class computers by reducing the run time and memory requirements of these problems. GP based ordering also aids in effective parallelization of some of the key computations in an interior-point algorithm [16, 17, 21].

5 Acknowledgements

I am grateful to John Forrest for providing the test problems, for providing access to his workstation to run the experiments, and for his help in creating the experimental set up by exposing OSL's internal ordering routines. I am also thankful to George Karypis and Vipin Kumar for useful discussions during the earlier part of this work and to Edward Rothberg for his comments on an earlier draft of this paper. The sparse matrix ordering subroutine of WGPP uses the AMD [5] code written by Tim Davis, Patrick Amestoy, and Iain Duff to label small subgraphs with a few hundred nodes.

References

[1] 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.

[2] Cleve Ashcraft and Joseph W.-H. Liu. Generalized nested dissection: Some recent progress. In *Proceedings of Fifth SIAM Conference on Applied Linear Algebra*, Snowbird, Utah, June 1994.

[3] T. Bui and C. Jones. A heuristic for reducing fill in sparse matrix factorization. In *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, pages 445–452, 1993.

[4] G. B. Dantzig. *Linear Programming and Extensions.* Princeton University Press, Princeton, NJ, 1963.

[5] 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.

[6] Iain S. Duff and Torbjorn Wiberg. Remarks on implementations of $O(n^{1/2}\tau)$ assignment algorithms. *ACM Transactions on Mathematical Software*, 14:267–287, 1988.

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

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

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

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

[11] Alan George, Joseph W.-H. Liu, and Esmond G.-Y. Ng. Communication reduction in parallel sparse Cholesky factorization on a hypercube. In M. T. Heath, editor, *Hypercube Multiprocessors 1987*, pages 576–586. SIAM, Philadelphia, PA, 1987.

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

[13] P. E. Gill, W. Murray, M. A. Saunders, J. A. Tomlin, and M. H. Wright. On projected Newton barrier methods for linear programming and an equivalence to Karmarkar's projective method. *Mathematical Programming*, 36:183–209, 1986.

[14] Anshul Gupta. Fast and effective algorithms for graph partitioning and sparse matrix reordering. Technical Report (Number to be assigned), IBM T. J. Watson Research Center, Yorktown Heights, NY, 1996.

[15] Anshul Gupta. WGPP: Watson graph partitioning (and sparse matrix ordering) package: Users manual. Technical Report RC 20453 (90427), IBM T. J. Watson Research Center, Yorktown Heights, NY, May 6, 1996. Available on the WWW at the *IBM Research CyberJournal* site at *http://watson.ibm.com:8080.*

[16] Anshul Gupta, George Karypis, and Vipin Kumar. Highly scalable parallel algorithms for sparse matrix factorization. Technical Report 94-63, Department of Computer Science, University of Minnesota, Minneapolis, MN, 1994. To appear in *IEEE Transactions on Parallel and Distributed Systems*, 1997. Postscript file available via anonymous FTP from the site *ftp://ftp.cs.umn.edu/users/kumar.*

[17] Anshul Gupta and Vipin Kumar. Parallel algorithms for forward and back substitution in direct solution of sparse linear systems. In *Supercomputing '95 Proceedings*, December 1995.

[18] Bruce Hendrickson and Robert Leland. A multilevel algorithm for partitioning graphs. In *Supercomputing '95 Proceedings*, 1995. Also available a Technical Report SAND93-1301, Sandia National Laboratories, Albuquerque, NM.

[19] Bruce Hendrickson and Edward Rothberg. Improving the runtime and quality of nested dissection ordering. Technical Report SGI manuscript, Silicon Graphics, Inc., Mountain View, CA, 1996.

[20] Narendra Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(8):373–395, 1984.

[21] 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, MN, April 1994. A short version appears in *Supercomputing '94 Proceedings*.

[22] 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, 1995.

[23] George Karypis and Vipin Kumar. METIS: Unstructured graph partitioning and sparse matrix ordering system. Technical report, Department of Computer Science, University of Minnesota, 1995.

[24] George Karypis and Vipin Kumar. Parallel multilevel graph partitioning. Technical Report TR 95-036, Department of Computer Science, University of Minnesota, 1995.

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

[26] Optimization Subroutine Library. *Guide and Reference*. IBM Corporation, Release 2.1, Fifth Edition, February 1995. Publication number SC23-0519-04.

[27] R. J. Lipton, D. J. Rose, and R. E. Tarjan. Generalized nested dissection. *SIAM Journal on Numerical Analysis*, 16:346–358, 1979.

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

[29] Irvin J. Lustig, Roy E. Marsten, and David F. Shanno. Computational experience with a primal-dual interior-point method for linear programming. *Linear Algebra and Its Applications*, 152:191–222, 1991.

[30] Irvin J. Lustig, Roy E. Marsten, and David F. Shanno. On implementing Mehrotra's predictor-corrector interior-point method for linear programming. *SIAM Journal on Optimization*, 2(3):435–449, August 1992.

[31] N. Megiddo. Pathways to the optimal set in linear programming. In N. Megiddo, editor, *Progress in Mathematical Programming*, pages 131–158. Springer-Verlag, New York, 1989.

[32] Sanjay Mehrotra. On the implementation of a primal-dual interior point method. *SIAM Journal on Optimization*, 2(4):575–601, November 1992.

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

[34] Edward Rothberg. Personal Communication. Silicon Graphics, Inc., Mountain View, CA, 1996.

[35] Edward Rothberg and Bruce Hendrickson. Sparse matrix ordering methods for interior point linear programming. Technical Report SGI manuscript, Silicon Graphics, Inc., Mountain View, CA, 1996.

[36] M. Yannakakis. Computing the minimum fill-in is NP-complete. *SIAM Journal on Algebraic Discrete Methods*, 2:77–79, 1981.

Appendix A: The Effect of WGPP Ordering on Nonzero Patterns

In Figures 8 through 12, we graphically show the effect of our ordering on the nonzero pattern of some LP matrices. Each figure shows the nonzero pattern of the original unordered matrix and the nonzero pattern after WGPP ordering. Note that the nonzero patterns of the matrices after the ordering are strikingly similar, irrespective of the original nonzero patterns. Each pattern consists of several small blocks around the diagonal and a collection of dense columns towards the end. Each block around the diagonal corresponds to a small subgraph whose nodes have edges only withing itself and to the separator nodes bounding the subgraph. The dense rows and columns towards the bottom and right side of the ordered matrices correspond to the nodes of large separators that are pushed towards the end during the postprocessing phase (Section 2.4). The relative size of the bottom right dense posrtion depends on the sizes of the separtors that are found during various levels of graph bisection.

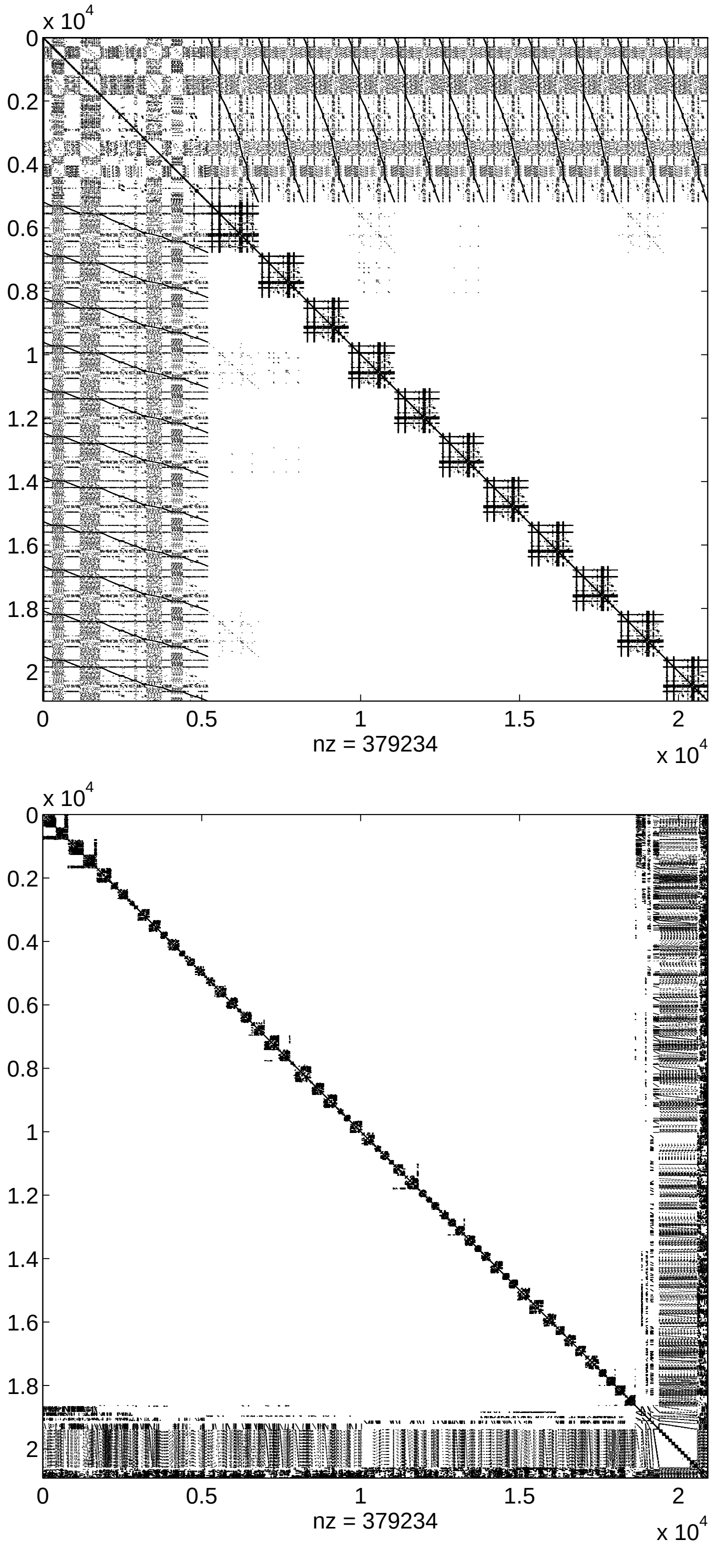


Figure 8: F12, before and after reordering.

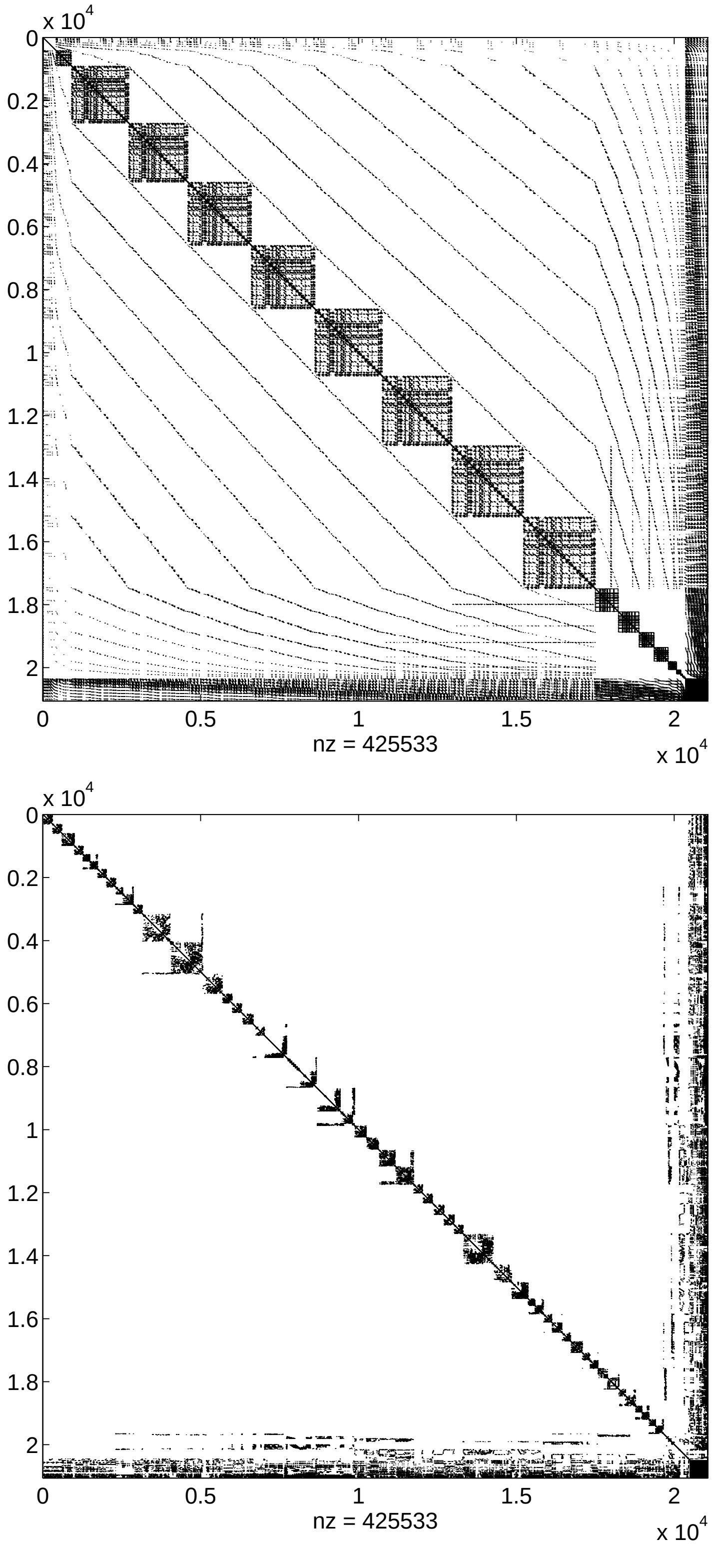


Figure 9: K8, before and after reordering.

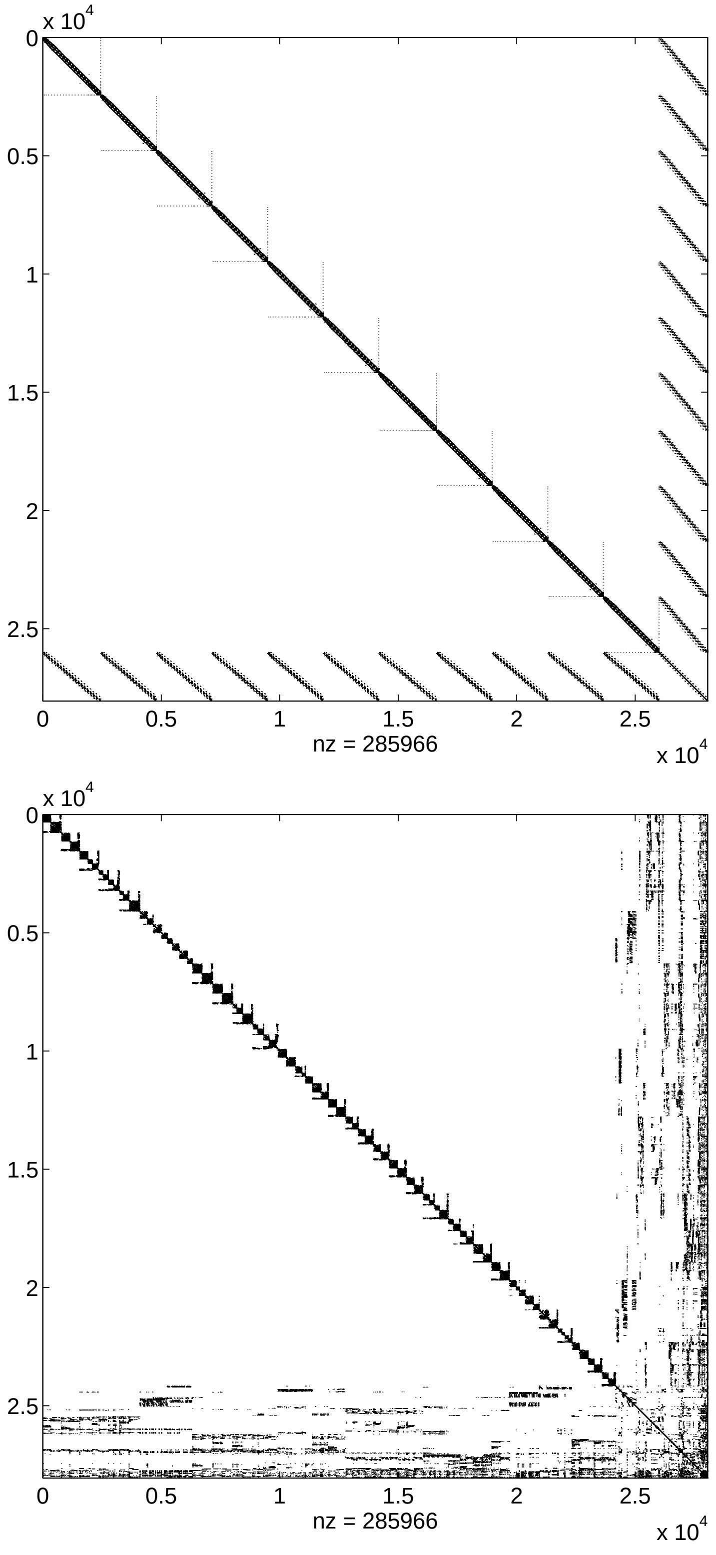


Figure 10: PDS20, before and after reordering.

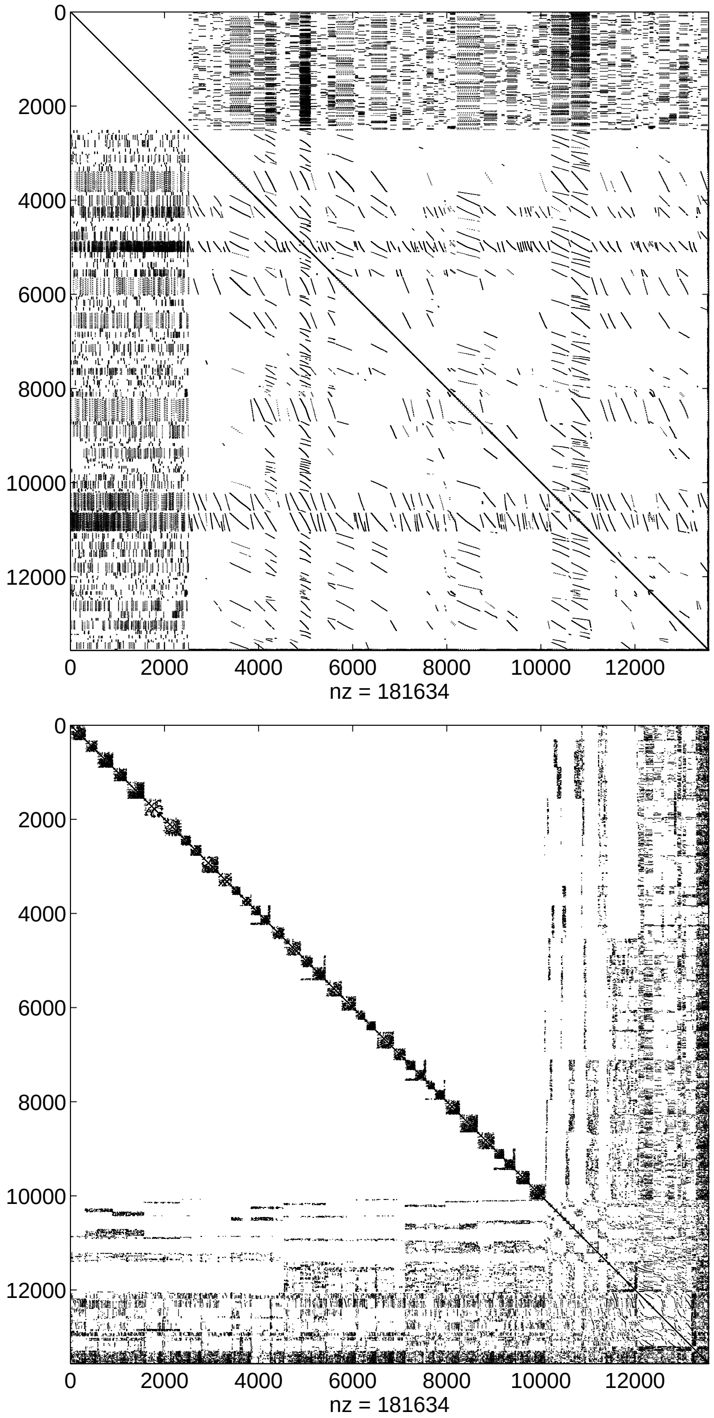


Figure 11: USAIR, before and after reordering.

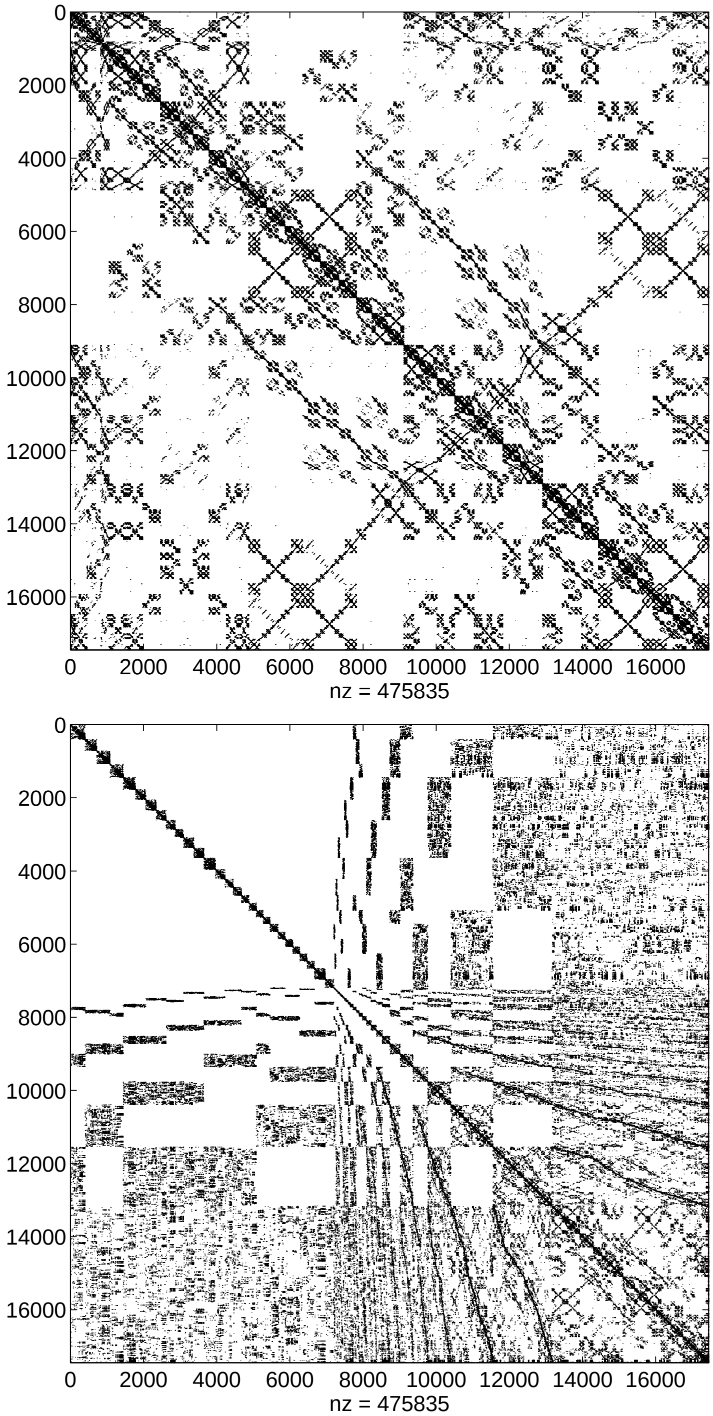


Figure 12: GISMONDI, before and after reordering.