[12] G. Miller, S. Teng, W. Thurston, and S. Vavasis, *Automatic mesh partitioning*, in Workshop on Sparse Matrix Computations: Graph Theory Issues and Algorithms, Institute for Mathematics and Its Applications, Springer-Verlag, 1992.

[13] G. L. Miller, S. Teng, and S. A. Vavasis, *A unified geometric approach to graph separators*, in Proceedings of the 32nd Annual Symposium on Foundations of Computer Science, IEEE, 1991, pp. 538–547.

[14] G. L. Miller and W. Thurston, *Separators in two and three dimensions*, in Proc. 22nd Ann. ACM Symp. Theory of Comput., New York, 1990, ACM, pp. 300-309.

[15] A. Pothen, H. D. Simon, and K.-P. Liou, *Partitioning sparse matrices with eigenvectors of graphs*, SIAM J. Matrix Anal. Appl., 11 (1990), pp. 430-452.

[16] J. D. Ullman, *Computational Aspects of VLSI*, Computer Science Press, Rockville, Maryland, 1984.

[17] S. A. Vavasis, *Automatic domain partitioning in three dimensions*, SIAM J. Sci. Stat. Comput., 12 (1991), pp. 950-970.

relatively primitive and cumbersome parallel I/O facilities currently available. Eventually the algorithm we have developed will be integrated into an overall distributed parallel software environment, such as a structural analysis package, so that the problem can be generated and solved in place on the parallel machine, with problem size limited only by the total memory available on the entire ensemble of processors. Our preliminary results with much smaller problems encourage us to expect the CND algorithm to be very effective in such an environment.

8. Future Work. We are encouraged by our results to date, but a considerable amount of work remains to be done along these lines. More extensive experimentation is needed, both in solving much larger and more diverse problems and in comparing the results with other competing algorithms. The ordering algorithm could be extended in several ways. For example, it may compute a separator that is unnecessarily large, and it would be desirable to reduce the separator to one of minimal size. We would also like to experiment with random sampling techniques to reduce the computational cost of the algorithm. Another area for further research is the use of rotations, conformal mappings, or other transformations of the input graph that might enhance the effectiveness of the Cartesian nested dissection algorithm. The algorithm could also be generalized to handle problems in three dimensions.

We are currently engaged in using the notion of Cartesian separators to design an algorithm for directly computing a suitable ordering for a nonsymmetric sparse matrix A without first computing the structure of $A^T A$. Of course, the ultimate goal is to solve large sparse systems of linear equations, so development of complementary algorithms for the subsequent numerical phases of the computation must also be completed. Finally, the entire suite of algorithms needs to be integrated into a usable software library format, and also integrated into software packages for specific applications areas, such as finite element structural analysis.

9. Acknowledgement. We wish to thank John Gilbert and Esmond Ng for helpful comments that improved the presentation of this paper.

REFERENCES

[1] M. J. Berger and S. H. Bokhari, *A partitioning strategy for nonuniform problems on multiprocessors*, IEEE Trans. Computers, C-36 (1987), pp. 570–580.

[2] G. C. Fox et al., *Solving Problems on Concurrent Processors*, vol. 1, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1988.

[3] J. A. George, *Nested dissection of a regular finite element mesh*, SIAM J. Numer. Anal., 10 (1973), pp. 345–363.

[4] J. A. George and J. W.-H. Liu, *An automatic nested dissection algorithm for irregular finite element problems*, SIAM J. Numer. Anal., 15 (1978), pp. 1053–1069.

[5] ———, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1981.

[6] J. R. Gilbert and E. Zmijewski, *A parallel graph partitioning algorithm for a message-passing multiprocessor*, Internat. J. Parallel Programming, 16 (1987), pp. 427–449.

[7] M. T. Heath, E. Ng, and B. W. Peyton, *Parallel algorithms for sparse linear systems*, SIAM Review, 33 (1991), pp. 420–460.

[8] B. Hendrickson and R. Leland, *An improved spectral graph partitioning algorithm for mapping parallel computations*. manuscript in preparation, 1992.

[9] R. Lipton and R. Tarjan, *A separator theorem for planar graphs*, SIAM J. Appl. Math., 36 (1979), pp. 177–189.

[10] J. W.-H. Liu, *Modification of the minimum degree algorithm by multiple elimination*, ACM Trans. Math. Software, 11 (1985), pp. 141–153.

[11] ———, *The role of elimination trees in sparse factorization*, SIAM J. Matrix Anal. Appl., 11 (1990), pp. 134–172.

Table 6

Time in seconds for ordering regular grids.

P	G100	G200	G300	G400
1	2.4	12.3	36.7	
2	2.1	8.3	24.9	
4	1.1	5.1	12.0	22.8
8	0.6	2.6	6.9	11.2
16	0.4	1.6	3.6	5.9
32	0.3	1.0	2.0	3.5
64	0.3	0.7	1.3	2.1
128	0.3	0.5	0.9	1.4

Table 7

Time in seconds for ordering L-shaped graphs.

	CND-bal			CND-opt		
P	L3	L6	L12	L3	L6	L12
1	9.1			20.0		
2	5.9	14.6		10.1	19.8	
4	4.0	8.9	15.2	6.9	13.2	25.7
8	2.1	4.4	8.5	4.4	8.7	19.1
16	1.3	2.5	4.7	3.0	5.5	11.1
32	0.9	1.6	3.0	2.3	3.7	8.9
64	0.7	1.1	2.0	1.8	2.8	6.2
128	0.6	0.9	1.6	1.5	2.4	5.0

TABLE 3
Millions of floating-point operations to compute L.

Problem	CND-bal	CND-opt	AND	MMD
L3	22	14	24	13
L6	49	35	55	27
L12	278	120	219	66

using a given ordering. Both measures are rather pessimistic, however, in that they not take into account all of the available sources of parallelism, nor do they acc for differences in the ability to exploit dense matrix kernels in the computatio Nevertheless, we see that CND-opt produces shorter elimination trees than AND or MMD, and the critical cost for CND-opt is also very competitive with the other orderings. We expect the elimination trees produced by CND-bal to be very well balanced, but the larger separators incurred can cause the total height of the tree the critical cost to be significantly higher than those for the other three orderin

TABLE 4
Elimination tree height.

Problem	CND-bal	CND-opt	AND	MMD
L3	632	441	581	580
L6	672	668	675	915
L12	1626	995	1444	1397

TABLE 5
Work along critical path.

Problem	CND-bal	CND-opt	AND	MMD
L3	11	2.7	11	3.0
L6	13	6.8	21	4.6
L12	134	31.0	77	13.0

Tables 6 and 7 show the ordering times for the CND algorithm using various numbers of processors P on an iPSC/860 hypercube multicomputer. The blank entries in the tables indicate cases that were not run because the problem would not fit in memory for that number of processors. We cannot give comparative results for AND and MMD, since they are not parallel algorithms. In Table 6 we show results only for CND-bal, since it already produces ideal orderings for square grids, and her there is no need to use the optimal criterion. As expected for any fixed problem size, we see a diminishing gain as more processors are used. Yet, in light of o previous experience with sparse matrix algorithms on such parallel machines, we fi it encouraging that we continue to see any speedup at all as we reach as many as 128 processors. In particular, these results suggest that communication costs ar growing unreasonably as the number of processors increases.

It should be noted that all of these test problems are relatively small, as even largest problems still fit on only four processors. The size of our test problems was ited by the logistic difficulties of generating large problems, transferring them a national networks, and getting them into and out of the parallel machines through t

Table 1
Description of test problems.

Problem	N	M
G100	10,000	19,000
G200	40,000	79,000
G300	90,000	179,400
G400	160,000	319,200
L3	12,864	37,983
L6	25,728	76,086
L12	42,880	127,170

of $\alpha = 1/3$. The latter choice for α is heuristic; it is simply intended to give th algorithm some freedom to reduce the separator size, yet not allow the splitting of graph to become too skewed. We note that this value has also sometimes been used in theoretical work on graph separators [9]. CND-bal does not require estimation or optimization of the separator size, and hence is less costly to compute than CND opt. CND-bal should produce well balanced subgraphs but may suffer a great deal of fill. CND-opt, on the other hand, incurs much less fill but may not maintain good balance. As mentioned earlier, we have also implemented a hybrid algorithm that uses CND-opt for the highest levels of nested dissection in order to keep those cri separators small, then switches over to the cheaper CND-bal for the remaining leve of dissection. We do not provide results for this hybrid approach, however, as th simply fall between those for pure CND-opt and CND-bal, mimicking one or the other more closely depending on the crossover point chosen for switching criteria. comparison with CND-bal and CND-opt, we also give results for two well known serial ordering algorithms, Automatic Nested Dissection (AND) [5] and Multiple Minimum Degree (MMD) [10].

Tables 2 and 3 compare the orderings with respect to sparsity preservation by considering the resulting number of nonzeros in the Cholesky factor L and the tot number of floating-point operations required to compute L. There is no need for a sparsity comparison for the regular grids, since CND-bal produces theoretica ideal orderings for such problems. For the L-shaped problems, we see that CND-bal compares well with AND, and that CND-opt compares reasonably well with MMD, which is usually considered the best heuristic known for irregular problems.

Table 2
Thousands of nonzeros in Cholesky factor L.

Problem	CND-bal	CND-opt	AND	MMD
L3	462	401	458	381
L6	957	858	949	779
L12	2444	1819	2112	1476

Tables 4 and 5 compare the orderings with respect to two theoretical measures of parallelism, namely the height of the elimination tree (see [11] for definition) and the work, measured in millions of floating point operations, along the critical p in the elimination tree (essentially tree height weighted by the number of floati point operations at each node). These measures have commonly been used to give a rough idea of the potential running time of parallel sparse Cholesky factorizat that each processor holds at most cN/P vertices and cM/P edges, where c is a small constant. In the remainder of this section, the letter c is used to denote a suit… constant.

We estimate the communication complexity in terms of the number N_{msgs} of messages communicated by each processor. Communication is limited to the distribute… phase comprising D levels of nested dissection, where $D \approx \log_2(tP)$ and t is a small constant. At each level of distributed nested dissection, a few accumulation, casc… and global aggregation operations are performed. Each of these operations invol… $\log_2 P$ messages per processor. Over D levels, this amounts to $O((\log_2 P)^2)$ messages per processor. Since redistribution is simply a variant of pairwise accumulatio… also requires $\log_2 P$ messages. Accordingly,

$$N_{msgs} \leq c\,(\log_2 P)^2.$$

To estimate the computational complexity, we observe that the cost of a single level of nested dissection is proportional to the maximum number of edges on a processor, excluding the overhead associated with pairwise accumulation, cascading … global aggregation operations. The one-time cost of redistribution must also be ta… into account. But for these exceptions, the cost of nested dissection would amou… to $c(M/P)\log_2 N$. The overhead associated with cascading and global aggregation operations is proportional to the amount of information communicated. For these op… erations, the lists communicated contain a few values for each graph at that level … nested dissection. The communication volume is $c \log_2 P\,|G_l|$ for each level l. Since $|G_l|$ doubles for each successive level of nested dissection, the communicatio… volume is given by

$$c\,(\log_2 P)\,\{1 + 2 + 4 + 8 + \cdots + tP\} \leq 2\,c\,t\,P(\log_2 P).$$

¿From this result it can be seen that the associated cost is $P \log_2 P$. Using cascading accumulation (without explicit merging at each stage) results in $O(1)$ overhead for ea… pairwise communication step. At the end of the $\log_2 P$ steps, each processor π_k must merge count information over values in $L(\pi_k)$. Recall that the sets $L(\pi_k)$ were chosen so that each contains approximately N/P vertices. Therefore, the cost of mergin… is proportional to N/P. Likewise, there is only a constant overhead associated wi… the redistribution operation, since a processor simply forwards a portion of a rec… message. Following redistribution, new data structures must be set up on each pro… cessor for use in further processing, but this work is perfectly parallel and s… more or less evenly across the processors. Thus, the parallel arithmetic complexi… $O((M/P)\log_2 N)$.

7. Test Results. In this section we present some empirical test results for the parallel Cartesian nested dissection algorithm. In Table 1 we show the number o… vertices and edges for two types of test problems. The first type, labeled Gxxx, a… regular square grids of the given size; for example, G100 is a 100×100 square grid. … second type, labeled Lx, are L-shaped finite element problems generated by ANSYS, which is a standard commercial software package for finite element analysis. Thes… L-shaped graphs are quite irregular.

We give test results for the Cartesian Nested Dissection (CND) algorithm using two different options. By CND-bal we mean the CND algorithm using only the "exact" balance criterion $\alpha = 1/2$, and by CND-opt we mean the CND algorithm using the approximately optimal separator size within the balance range permitted by a valu…

the original graph G, to produce a complete nested dissection ordering in at mos $\log(|V|)$ steps. In a distributed parallel setting, however, it may be advantageou not to follow this process all the way to the end, since each step requires a signifi amount of communication. Instead, the dissection process can be stopped as soon a a level has been reached at which there are at least as many subgraphs as processors The data can then be reorganized to place whole subgraphs on each processor, so that a serial ordering algorithm can be applied to the remaining subgraphs on each processor from that point on, with no further communication required. We now describe such a two-phase, hybrid approach in greater detail.

The first phase of the hybrid algorithm consists of carrying out the first D levels of Cartesian nested dissection as described earlier, where D is the first level at the number of subgraphs is at least tP, with $t \geq 1$ a parameter specified by the user. The choice $t = 1$ yields less overall communication, since it shifts more the work to the second, communication-free phase. However, a choice of $t > 1$, by producing more subgraphs than the number of processors, may allow more flexibility in achieving a good load balance across processors during the second phase. Thus there is a problem-dependent trade-off in choosing a value for t. Whatever the choi for t, after D steps the Cartesian nested dissection process is stopped, and we m then redistribute the problem data so that each subgraph is assigned in its entir to only one processor. This redistribution step requires a significant amount of gl communication, which must be taken into account in assessing the total cost of th hybrid algorithm.

The necessary redistribution of problem data can be accomplished by a variant of the pairwise accumulation algorithm described earlier. In our earlier use of pai accumulation, we used the blocks of coordinates, $L(P-1), \ldots, L(0)$, as a means of organizing the accumulation so that at each step of dimensional exchange the computation would be shared among processors and the resulting data would be assigned to processors in a systematic way. For purposes of redistributing probl data between the global and local phases of the hybrid ordering algorithm, numeric accumulation is not required, but we can still use the same organization as pairwi accumulation to direct the flow of data to the necessary destinations. Specifically let the list of subgraphs to be redistributed play the same role that the coordin blocks played previously.

Let $\mathcal{G}_D = \{G_1, \ldots, G_q\}$ be the set of subgraphs after level D of nested dissection. We partition $\mathcal{G}_D$ into P subsets of graphs given by $\mathcal{G}(0), \ldots, \mathcal{G}(P-1)$. We use the symbol $list(\mathcal{G}(k))$ to denote information (Cartesian labels of vertices an edge lists) pertaining to all graphs in the set $\mathcal{G}(k)$. The structure of $list(\mathcal{G}(k))$ of the form $[<G_i \cdots\cdots>, \ldots, G_j < \cdots\cdots>, \ldots]$, where G_i and G_j are in $\mathcal{G}(k)$ and $i < j$. Merging of any two such lists takes time proportional to the sum of their sizes since the information is in increasing order of graph-ids. Concatenat $list(\mathcal{G}(0)), \ldots, list(\mathcal{G}(P-1))$ yields $list(\mathcal{G}(0), \ldots, \mathcal{G}(P-1))$. Redistribution of vertex information can thus be accomplished by using the above lists in the pairwis accumulation algorithm with $\mathcal{G}(0), \ldots, \mathcal{G}(P-1)$ instead of $L(0), \ldots, L(P-1)$. Once the redistribution step has been completed, then each processor π_k can apply a serial algorithm to order each subgraph in $\mathcal{G}(k)$ without any further communication among processors.

6.1. Parallel Complexity. We now provide estimates of the communication and computational complexity of the parallel Cartesian nested dissection algori for a graph $G = (V, E)$ with N vertices and M edges using P processors. We assume

dinate values satisfying the balance condition. We can then use a similar three-st
process to compute a value for each subgraph that minimizes η. Let $\mathcal{E} = \{E_1, \ldots, E_m\}$
the collection of edge sets of the subgraphs at this level of nested dissection. Th
to be accumulated are initially of the form $count_l(\pi_k)$ for processor π_k. At the
end of pairwise accumulation phase, π_k has $count_l(\mathcal{E}(k))$. Recall that the aim
is to compute $\kappa(i)$, the number of edges that straddle i in some subgraph G_j, using
the equation $\kappa(i) = \kappa(i-1) + \beta(i-1) - \varepsilon(i)$. Let r be the largest value in
$L(\pi_{k-1})$ for some graph $G_l \in \mathcal{G}_l$. Processor π_k requires the value $\kappa(r)$ of π_{k-1} to
compute $\kappa(i)$ for $i \in L(k)$. The cumulative count list for processor π_k consists of the
number of edges that cross its largest value in each subgraph. These cumulative cou
lists are cascaded as before. After cascading is complete, each processor π_k computes
the estimate η for each value in $L(k)$ and selects the one with a minimum value as
the local minimum. A global minimum is computed over all local minima by using
the same aggregation process as before, except that now the operation for combinin
information is selecting the minimum value rather than taking the set union.

The process of obtaining a set of separating values for all subgraphs in $\mathcal{G}_l$ in
given coordinate dimension is now complete. A similar process is used to compute a
separating level for each subgraph in the other coordinate dimension. Each proces
can then determine the final separating value for each subgraph by making a local
comparison of the computed separating values in each coordinate dimension. We
denote the set of separating values for the subgraphs in $\mathcal{G}_l$ by $\sigma = \{s_1, \ldots, s_r\}$.

5.2. Constructing Separators in Parallel. Having determined a separating
value s_i, we must now construct a separator V_{s_i} for each subgraph $G_i \in \mathcal{G}_l$. Recalling
our earlier discussion, this requires that we compute the set U_{s_i} of vertices in G_i
with coordinate s_i, the set E_{s_i} of edges in G_i that straddle the value s_i, and the
correction set C_{s_i}. Using its vertex lists and group trees, a given processor π_k can
compute the subsets $U_{s_i}(\pi_k)$, $E_{s_i}(\pi_k)$, $C_{s_i}(\pi_k)$, and $V_{s_i}(\pi_k) = U_{s_i}(\pi_k) \cup C_{s_i}(\pi_k)$, but
communication would be required to compute the complete sets. Such non-disjoint
set unions could be computed by a dimensional exchange process analogous to those
we have already seen, but we can avoid some of the overhead that would be required
by taking a different approach in which the processors cooperate to number their
portions of each separator without ever forming the set union explicitly.

Since the numbering of vertices within a single separator is arbitrary, we ado
the convention that the vertices in $V_{s_i}(\pi_k)$ are numbered after those in $V_{s_i}(\pi_{k-1})$
for $0 < k < P$. To determine the range of numbers to use for its portion, each
processor π_k needs to know the total size of the subsets of each V_{s_i} on processors
$\pi_0, \ldots, \pi_{k-1}$. This can be accomplished using the previous cascade algorithm, with
$|V_{s_1}(\pi_k)|, \ldots, |V_{s_r}(\pi_k)|$ as the set of values to be cascaded at processor π_k. At the end
of the cascade step, processors number the vertices in their portions of each separ

The fact that the union of $E_{s_i}(\pi_k)$ over all processors is not explicitly constructed
may result in a separator that is somewhat larger than strictly necessary. In the se
case the correction set C_{s_i} is computed based on $E_{s_i} = \bigcup_{k=0}^{P-1} E_{s_i}(\pi_k)$, whereas in the
parallel case each processor computes $C_{s_i}(\pi_k)$ based on $E_{s_i}(\pi_k)$. Consider an edge
$(u, v) \in E_{s_i}(\pi_j)$ and another edge $(u, w) \in E_{s_i}(\pi_k)$. In the serial case, the common
vertex u could be selected to cover both edges, but in the distributed case a differ
vertex may be selected from each edge, thereby increasing the size of the separato

6 Parallel Cartesian Nested Dissection The algorithm given in the previ-
ous section computes a set of separators for all of the subgraphs at a given level
nested dissection. Thus, the algorithm could be applied repeatedly, beginning w

Computation of the required cumulative counts is an example of a parallel prefix computation, which can be implemented in a number of ways, with the best choice dependent on the interconnection network among the processors. Once again, we illustrate with an implementation, which we refer to as *cascading*, that is appropriat for a hypercube network using a form of dimensional exchange. Each processor π_k initially holds its cumulative count list $ccount_l(\mathcal{V}, L(k))$. During the successive steps of the cascading process, each processor maintains two lists of cumulative co one list to be kept and the other to be propagated further to other processors. Th list to be retained contains cumulative counts corresponding to blocks of coordi values smaller than $L(k)$ for each subgraph, and hence is initially empty. The list be propagated differs from the retained list in that it includes cumulative counts all blocks of coordinate values that the processor has seen thus far, and hence ini consists of $ccount_l(\mathcal{V}, L(k))$.

The cascading process requires $d = \log_2 P$ steps. In the first step of dimensional exchange, pairs of processors whose processor numbers differ in the least significan exchange their propagated cumulative count lists and merge the information receiv into the two lists to be kept and propagated. This first set of exchanges takes pla within 1-dimensional subcubes between consecutively numbered processors, say π_k and π_{k-1}, where k is odd. After the first exchange, the retained list on the lower numbered processor in each pair, π_{k-1}, remains empty, while that on the higher numbered processor, π_k, becomes $ccount_l(\mathcal{V}, L(k-1))$. The propagated list on both processors becomes $ccount_l(\mathcal{V}, L(k-1), L(k))$. At the next step, exchanges take place within subcubes of dimension 2. Each processor π_k exchanges its cumulative list to be propagated with $neighbor(k, 2)$. Again, take π_k to be the higher numbered processor of the pair, so that the other processor is π_{k-2}. Processor π_k receives a list of the form $ccount_l(\mathcal{V}, L(k-3), L(k-2))$, so that, after merging, its retained list is updated to become $ccount_l(\mathcal{V}, L(k-3), L(k-2), L(k-1))$ and the list to be propagated becomes $ccount_l(\mathcal{V}, L(k-3), L(k-2), L(k-1), L(k))$. The lower numbered processor of the pair need update only its propagated list, which become the same as that of the higher numbered processor, since both have seen the same blocks at this point.

This exchange process continues over subcubes of successively higher dimensio After i steps processor π_k contains a propagated list of the form $ccount_l(\mathcal{V}, L(k - 2^i + 1), \ldots, L(k+1))$ and a retained list of the form $ccount_l(\mathcal{V}, L(k - 2^i + 1), \ldots, L(k))$. The process terminates after d steps, at which point every processor has a retain list of the form $ccount_l(\mathcal{V}, L(0), \ldots, L(k-1))$, which is the desired result. The cascading process just described requires $\log_2 P$ communication steps and $P \log_2 P$ messages. An alternate implementation of parallel prefix can reduce the number of messages required, but it does not reduce the number of steps and requires non-neighbor communication in a hypercube.

Once cumulative counts have been cascaded, each processor can now determine, for each subgraph, the set of values within block $L(k)$ that satisfy the balance con tion. These sets of values must then be aggregated over all processors to arrive at full set of values satisfying the balance condition for each subgraph. This aggreg of sets can again be computed by a dimensional exchange process having d steps, at step i of which each processor exchanges information with its neighbor in the i dimension and the information received is combined with previous information by se union.

For each subgraph G_l, the above three-stage process determines a block of coorfor a block of coordinate values. Let L denote the set of coordinate values along a gi dimension over all subgraphs G_i, and let L be partitioned into P contiguous blocks of values, $L(0), \ldots, L(P-1)$, such that each block covers about the same number of vertices (which is always possible for reasonably well behaved graphs). Processo π_k will be responsible for accumulating the counts for each value in block $L(k)$ for all i. Let $\mathcal{V} = \{V_1, \ldots, V_t\}$. Initially, a given processor π_k contains $count(\pi_k, \mathcal{V}, L)$, and we want it to end up with $count(L(k), \mathcal{V})$. In other words, each processor initially has counts over all the coordinate values, but only for its own portion of each subgra whereas we want it to contain the counts over each entire subgraph, but only for it assigned block of coordinate values.

The best implementation of such a global information exchange operation depends on the interconnection network among the processors. Here we will illustra one possible implementation, which we term *pairwise accumulation*, that is suitable for a hypercube network (or any network that contains a hypercube or can emulate a hypercube efficiently). The algorithm is based on dimensional exchange. For simplic ity, assume that P is a power of two, and let $d = \log_2 P$. Let $neighbor(\pi_k, l)$, $1 \leq l \leq d$, denote the processor whose processor number differs from that of π_k in the lth least significant bit. The algorithm has d steps. In the first step, each process π_k, $0 \leq k < P/2$, sends $count(\pi_k, \mathcal{V}, L(P/2), \ldots, L(P-1))$ to $neighbor(\pi_k, d)$. Conversely, each processor π_k, $P/2 \leq k < P$, sends $count(\pi_k, \mathcal{V}, L(0), \ldots, L(P/2-1))$ to $neighbor(\pi_k, d)$. In other words, the processors in the lower and upper halves of the hypercube exchange counts for the upper and lower halves of blocks, so that th lower blocks end up on the lower processors, and the upper blocks end up on the upper processors. Each processor merges incoming information into its subgraph l corresponding to the appropriate set of coordinate values. This process is appl recursively to the two subcubes of dimension $d-1$, and so on, so that after d steps each processor has the desired information, namely, processor π_k contains counts over all subgraphs for the kth block of coordinate values.

In order that a processor can merge lists in time proportional to the sum of thei sizes, we structure the list $count(\mathcal{V}, L)$ as

$$[\, l_1, <G_i, count>, \ldots, <G_j, count>, \ldots, l_2, <G_i, count>, \ldots, <G_j, count>, \ldots]$$

where l_1 is the smallest value in L, followed by a list of $<graph-id, count>$ pairs. The graph-id, count pairs are listed in increasing order of graph-id numbers. Sim information is listed for the next value, l_2, and so on.

The pairwise accumulation process described above effectively spreads the wor of accumulating counts for the coordinate values across all of the processors, bu must still traverse the resulting count lists and compute the cumulative vertex co in order to determine a separating value for each subgraph. The set of coordinat values spanned by an individual subgraph G_i may intersect more than one block of values $L(k)$, and hence the corresponding count lists may be spread over multiple processors. Thus, the necessary list traversals and cumulative vertex counts require further interprocessor communication. For a given subgraph G_i, let $L_i = \{l_i, \ldots, r_i\}$ be the ordered set of coordinate values spanned by G_i, and let $\forall k$ $L_i \cap L(k) = \{l_i(\pi_k), \ldots, r_i(\pi_k)\}$. In order to determine if a separating value lies within $L_i \cap L(k)$, processor π_k requires a cumulative count of vertices in G_i for all previous coordinate values, $l_i, \ldots, l_i(\pi_k) - 1$. We denote this cumulative count by $cumcount(V_i, L(0), \ldots, L(k-1))$. Processor π_k requires cumulative counts over all subgraphs in $\mathcal{V}$, which we denote by the list $cumcount(\mathcal{V}, L(0), \ldots, L(k-1))$.

The data distribution described above results in each processor's having verti
and edges at almost all coordinate values, but not having all of the vertices and ed
associated with any one coordinate value. As a consequence, in determining a sep
arating value, vertex and edge count lists must be accumulated over all processo
and traversed in increasing order of coordinate values to identify a separating v
satisfying a balance condition and/or minimizing η, and finally this computed sepa
rating value must be disseminated to all processors. Obviously, these steps requ
several phases of interprocessor communication, as well as a significant amount
computation. For effective parallelization, we will distribute lower order costs
as computing separating values over subgraphs at a given level of nested dissecti
as well higher order costs, such as constructing the vertex and edge count lists, a
all of the processors, and will also try to minimize communication costs.

In dealing with distributed data structures, we will adopt the notation that t
portion of a given entity that resides on processor π_k will be indicated by appending
(π_k) to the usual notation for the global object in question. Thus, for example, $V_i(\pi_k)$
denotes the portion of vertices in subgraph G_i that reside on processor π_k, and $list_i(\pi_k)$
denotes the portion of the given list residing on processor π_k.

5.1. Computing Separating Values in Parallel. We now describe the pro-
cess of computing separating values in parallel. As we will soon see, this computat
requires the same global communication pattern for each subgraph at a given level o
nested dissection. For many distributed memory parallel computers, the start-up c
for communication is relatively high, and therefore it pays to minimize the number
messages required to send a given volume of data. For this reason, we will concatena
together all of the data to be exchanged among processors over all of the subgraph
at a given level of nested dissection, so that a single set of communications will s
for computing all of the separating values. Grouping communications in this manne
represents a substantial saving over computing the separating value for each subgr
individually, which would incur a separate round of communication for each. This i
one reason we chose not to use an explicitly recursive formulation of the algorit
since a depth first approach would not permit us to handle an entire set of subgraphs
at a given level at once.

As we have seen, the determination of appropriate separating values requires no
counts for each of a series of coordinate values. In a parallel setting, the nece
count information is distributed over all of the processors. Thus, for each coord
value, the counts must be accumulated across the processors, the resulting separat
values computed, and this information must then be made available to all of the
processors. These three steps are required for each subgraph in $\mathcal{G}_l = \{G_1, \ldots, G_r\}$ at
a given level l of nested dissection, and each step requires global communication
reduce the number of messages, and hence the total communication start-up overhead
we will combine all of the relevant data for all of the subgraphs at a given level
each communication step. Of course, for good parallel efficiency, we must also shar
the computational work among all of the processors as well.

We first consider the process of accumulating count information across all proce
sors. For each coordinate value j and each subgraph $G_i = (V_i, E_i)$ at the given level,
we need to compute

$$count(V_i, j) = \sum_{k=0}^{P-1} count(V_i(\pi_k), j)$$

We will allocate this work among the processors by making each processor responsibl which is much smaller than $|V|$. The cost of separating G_i is therefore of the form $c_s|E_i|$, where c_s is a small constant. The cost of separating all subgraphs at level l of nested dissection is therefore given by

$$c_s \sum_{G_i \in \mathcal{G}_l} |E_i| \le c_s M$$

It remains to estimate the costs of forming working lists and group trees for ea resulting subgraph. Each list for G_i can easily be decomposed into two lists, one for each resulting subgraph, in time proportional to the length of the list. This is pos since it can be decided which subgraph an entity belongs to by a simple comparison of the appropriate coordinate value with the separating value. Such a decompositi will yield lists that are still in increasing order of the respective coordinat since the original sorted order is not affected by deletions. Accordingly, this c $O(|E_i|)$. A group tree for G_i can be decomposed into group trees for each of the resulting subgraphs. Each interval in a group tree for G_i can be easily determined if the interval is in G_{i1} or G_{i2} by comparing it with the separating value. The interval can then be added to the appropriate group tree. Including th overhead of allocating and initializing groups, the cost is proportional to $|E$ Over all subgraphs G_i in $\mathcal{G}_l$, the total cost of updating lists and group trees is therefore $c_u M$, where c_u is a small constant.

¿From the above paragraphs it follows that the cost of one level of nested dis-section is $c_l M$ where c_l is a constant. Thus, a single initialization step followed by at most $\log_2 N$ levels of nested dissection results in a serial time complexity of $O(M \log N)$.

5. Computing Separators in Parallel. We now adapt the Cartesian separator algorithm for use on a distributed memory parallel computer. Our goal will be to distribute the computation evenly across the processors while keeping the volume a frequency of interprocessor communication low. For the resulting parallel algorit be scalable, both higher and lower order costs should be shared among all processo and all data structures should be distributed across all memories. The distribu parallel algorithm will have the same general form as the serial algorithm, but t work of forming lists and counting and searching will be shared by all of the process In effect, each processor will own a portion of the data and will be responsible for counting or searching involving that portion. Coordinating such joint activities a the processors and reporting the results will obviously require some interproce communication, but we try to limit this for good efficiency.

Let the number of processors be P. We assume that the set of vertices V of the original graph is distributed among the processors so that each processor h approximately $|V|/P$ vertices. The set of edges E is distributed among the processor so that each edge is assigned to a processor holding one of the two vertices at i endpoints. This may not result in an even distribution of edges for all graphs, but most graphs arising in practice, such as finite element graphs, the number of edges each processor will be at most a constant times $|E|/P$. In mapping the problem data to processor memories, we make no assumption that locality is preserved, nor do we assume any correlation between the topology of the graph and the topology of the processor interconnection network. Indeed, the parallel algorithm we propose te to perform best with a random data distribution, since such a distribution tends balance the computational load in forming and searching the various lists require discuss the distributed parallel implementation below. Thus, rather than a typi depth-first approach resulting from explicit recursion, we instead take a breadth- approach, dealing with all of the subgraphs at a given level of dissection before mo on to the next level.

We introduce some notation here that we will find useful later on in formulating the parallel algorithm. For any given level l of the nested dissection process, w $\mathcal{G}_l$ denote the set of subgraphs of the initial graph at level l. We begin at level with $\mathcal{G}_0 = \{G\}$, where $G = (V, E)$ is the graph of the given sparse matrix to be ordered. The vertices and edges of G are scanned to construct the working vertex and edge lists, $list(V, x)$, $list(V, y)$, $list(E, x)$, and $list(E, y)$, and these lists a in turn to generate the corresponding count lists. A separating coordinate valu and Cartesian separator V_s are then computed for G as described previously, which yields two subgraphs G_1 and G_2. The vertices in the separator V_s are numbered $|V| - |V_s| + 1$ through $|V|$, completing level 0 of the dissection process. At level 1, we apply the Cartesian separator algorithm to each of the two subgraphs in $\mathcal{G}_1 = \{G_1, G_2\}$. Working lists are constructed for each subgraph, and separating coordinate values s_1 and s_2 and corresponding Cartesian separators V_{s_1} and V_{s_2} are computed. The vertices in the two separators are numbered and the four remaining subgraphs are then similarly processed at level 2, and so on. This process continues until vertices in the original graph have been numbered. At most $\log_2(|V|)$ levels of nested dissection are required to number all of the vertices, since the lth level result 2^l subgraphs.

4.1. Serial Complexity. We now estimate the serial time complexity of the foregoing Cartesian nested dissection algorithm. Consider a Cartesian labeled g $G = (V, E)$ with N vertices and M edges. We assume that any subgraph of G has at least as many edges as vertices. To compute the cost of ordering G we compute bounds for the cost of initialization and the cost of each level of dissection.

In the initialization step, vertices are sorted in increasing order of both x y coordinate values. The complexity of this step is $O(N \log_2 N)$ using a heapsort. These sorted lists are used to construct the working lists $list(V, x)$ and $list(V$ which requires time proportional to N, the length of the lists. The sorted list vertex coordinates are also used to construct the edge lists $list(E, x)$ and $list(E,$ time proportional to M. A group tree is constructed for each dimension by mapping edges to intervals. Each group tree can have at most N groups. Entering an interva into a group tree takes time proportional to $\log_2 N$. The cost of forming group trees is therefore proportional to $M \log_2 N$. The overall cost of the initialization step is therefore $O(M \log_2 N)$.

The cost of separating a subgraph $G_i = (V_i, E_i)$ is given by the sum of the costs of computing a separating value and then constructing and numbering the correspondin separator. Computing a separating value that satisfies the balance condition requi the formation and traversal of the vertex count lists $count_i(x)$ and $count_i(y)$. The cost of forming these lists is proportional to $|V_i|$. The cost of traversing them depends on the number of actual coordinate values in the graph, which is obviously at most $|V_i|$ Computing the estimate η for the separator size requires the formation and travers of the edge count lists $Ecount_i(x)$ and $Ecount_i(y)$, resulting in cost proportional to $|E_i|$. Computing the set of edges that straddle the separating value involves searchi one of the group trees and deleting edges selected. This can be accomplished in ti proportional to $\log_2(M)$ and the number of edges found. Computing and numbering the actual separator can be performed in time proportional to the size of the separa integers q and r define a group $g_{q,r}$. We would, for example, have $g_{48,64}$ and $g_{48,56}$ but not $g_{56,48}$. The group $g_{q,r}$ consists of intervals that have left endpoints greater than or equal to q, right endpoints less than r, and straddle the midpoint $m = (q + r)/2$; i.e., the left endpoint is at most m and the right endpoint is at least m. The inter within $g_{q,r}$ are arranged in a list that is threaded in two directions by two independent linked lists. One of the linked lists is in *increasing* order of the *left* endpoints and th other is in *decreasing* order of the *right* endpoints of intervals in $g_{q,r}$. Such a dual threading is required to enable efficient searching as described below.

A group tree is a complete binary tree whose vertices are interval groups. Consid an interval $[a, b]$ such that a and b are consecutive multiples of a power of two. The group tree $GT[a, b]$ for the interval $[a, b]$ is defined recursively by taking $g_{a,b}$ as root, and given a vertex $g_{q,r}$, its left child is $g_{q,(q+r)/2}$ and its right child is $g_{(q+r)/2,r}$. Given a group tree $GT[a, b]$, it is easy to find members that cross a given point s. The search is started at the root with $g_{a,b}$. The following actions are applied recursively at each vertex $g_{q,r}$, whose midpoint is $m = (q + r)/2$.

case $s = m$: All intervals in the group $g_{q,r}$ straddle s. Furthermore, intervals in descendants of $g_{q,r}$ cannot contain any intervals of interest, and thus the recursion terminates.

case $s < m$: Each interval in $g_{q,r}$ whose left endpoint is not larger than s must straddle s. Such intervals can be found in time linear in the number of matches, a the intervals are threaded in increasing order of left endpoints. The left child o $g_{q,r}$, namely the group $g_{q,(q+r)/2}$, must then also be searched.

case $s > m$: Each interval in $g_{q,r}$ whose right endpoint is not smaller than s must straddle s. Such intervals can be found in time linear in the number of matches, a the intervals are threaded in decreasing order of right endpoints. The right chil $g_{q,r}$, namely the group $g_{(q+r)/2,r}$, must then also be searched.

The time complexity of the above search process can be estimated by noting that the height of the group tree is $\log_2(b-a)$, so that at most $\log_2(b-a)$ groups need be searched. Within each group, the time spent is linear in the number of matches, du to the dual threading. Hence, the cost of a single search is at most $\log_2(b-a) + k$, where k is the actual number of matches.

The group tree search technique outlined above is immediately applicable to com puting the set of E_s edges that straddle the separating value s: we simply associate each edge in the subgraph with the interval whose endpoints are the coordinate valu in the given dimension of the corresponding pair of vertices. The two resulting gr trees (one for each coordinate dimension) are formed initially for the entire grap and thereafter can be modified easily for use in the searches at successive level nested dissection. Not counting this initialization cost, the cost of finding the dle edges for a given subgraph G_i using group tree search is then proportional to $\log |V_i|$ plus the number of edges found. This is a substantial improvement over the cost of the simpler algorithm described earlier, which is linear in $|E_i|$.

4. Cartesian Nested Dissection Having described an algorithm for computing a Cartesian separator for a given graph, we can use the algorithm repeatedly to derive an algorithm for Cartesian nested dissection to order a sparse matrix. most natural way to implement such an algorithm is to invoke the separator algorithm recursively on successively smaller subgraphs of the initial graph. We do not take an explicitly recursive approach, however, for reasons that will become clear when a label indicating the graph to which the information pertains, then counts c_i and c of vertices in G with coordinate value i, etc. The vertex count list *count*(V, x) traversed in increasing order and the cumulative count of vertices incremented un the first value is found, say a, that satisfies the balance condition. Traversal of list then continues until a value is found at which the balance condition is no lon satisfied; we denote by b the last value at which the balance condition was still satis Alternatively, depending on which would give the smallest expected running time, could instead be found by traversing the vertex count list in decreasing order fr the top. In either case, we will have identified the block $[a, b]$ of potential separa values, all of which satisfy the balance condition.

We must now compute the estimate $\eta(i)$ for each value $i \in [a, b]$. Let (u, v) be an edge in E, with $x(u) \leq x(v)$. Such an edge can be thought of as beginning at $x(u)$ and ending at $x(v)$. Let $\beta(i)$ and $\varepsilon(i)$ denote the number of edges that begin and end, respectively, at i. Edges in E are maintained in an *edge list*, denoted by *list*(E, x), in increasing order of the x coordinates of their associated vertice edge (u, v) is entered into the ordered edge list at positions given by $x(u)$ and $x($ where $x(u) < x(v)$, and marked respectively as a begin and an end entry. The edge list is traversed to compute an *edge count list*, denoted by *count*(E, x), of the form $[G, <i, \beta(i), \varepsilon(i)>, <j, \beta(j), \varepsilon(j)>]$. We now let $\kappa(i)$ be the number of edges that cross i. Observe that $\kappa(i) = \kappa(i-1) + \beta(i-1) - \varepsilon(i)$. This fact is used to compute $\kappa(i)$ for each value in the block $[a, b]$ by traversing the edge count list *count*$(E,$ We note also that the size of the initial approximation U_i to the separator, $|U_i|$, can be computed for each coordinate value i by scanning the vertex count list *count*(V, x) Finally, we note that for each coordinate value i, our estimate for the final separa size is given by $\eta(i) = |U_i| + \kappa(i)$. Having computed the value of $\eta(i)$ for each $i \in [a, b]$, we select the coordinate value s with the minimum value of $\eta(s)$ as the separating value for that dimension. A separating value is similarly computed for the othe coordinate dimension, and the one yielding the smaller estimated separator size selected as the separating value for computing a Cartesian separator.

3.2. Constructing a Separator. Having chosen a separating value s in one of the coordinate dimensions, we now proceed to construct a Cartesian separator. Agai for definiteness, assume that we have chosen the x coordinate dimension. According t our earlier definition, the desired separator is the union of the initial approximate separator U_s and the correction set C_s. The set U_s is easily computed using the vertex list *list*(V, x). The construction of the correction set C_s requires that we compute the set E_s of edges that straddle the separating value s. A simple way to search for these straddle edges would be to traverse the edge list *list*(E, x) in increasing order u value s. For each beginning edge (u, v), with $x(u) \leq x(v)$, we add the edge to E_s if $x(v) > s$. Upon reaching value s in traversing *list*(E, x), we have completed the computation of E_s. We initialize the set W_s to be U_s, then for each edge in E_s such that neither of its endpoints is already in W_s, we add to W_s one of those endpoints. The choice of which endpoint to include in W_s can be made arbitrarily, or the choice can be governed by requiring that the balance condition be maintained.

In the worst case, the computational cost of this simple algorithm for finding straddle edges is proportional to the number of edges in the subgraph. This cos can be reduced by using the concept of a *group tree* [16], which enables more efficient searching for intervals that contain a given point s. A group tree is based on the not of interval groups in a given coordinate dimension. An interval group is specified a pair of integers that are consecutive multiples of the same power of two; two suc (assuming that vertices are chosen appropriately for the correction set C the initial balance). Determining the size of a Cartesian separator, on the ot hand, is more difficult, since the set of vertices U_s with coordinates equal to s is merely an initial approximation that must be augmented by the correction set C_s, whose size is not so easily determined. In seeking a small separator we will, efficiency, merely estimate the eventual separator size rather than compute it exact For a given coordinate value s, we define the quantity

$$\eta(s) = |U_s| + |E_s|,$$

where the sets U_s and E_s are as defined previously. Clearly, $\eta(s)$ is an upper bound on the separator size; it may be an overestimate because a single vertex may "cover" mor than one straddle edge in E_s, so that $|C_s|$ may be smaller than $|E_s|$. Nevertheless, $\eta(s)$ is sufficiently accurate for our purposes, and we will use it as an estimate for separator size in seeking an approximate minimum.

The desired balance between the two subgraphs resulting from a single dissectio is given by a user-specified quantity, α, $0 < \alpha < 1$, which is interpreted as a limi on the relative proportion between the sizes of the two subgraphs. Specifically, require that the separating value s be chosen so that

$$\alpha|V| \leq |U_1|, \ |U_2| \leq (1-\alpha)|V|.$$

A value of $\alpha = 1/3$, for example, means that one subgraph can be at most twice the size of the other. There may be many potential separating values that satisfy th balance condition, with some values resulting in smaller separators than others. choose the value s that minimizes the estimate $\eta(s)$ for the separator size. We hand the special case $\alpha = 1/2$ separately, since it requires perfect balance (as close possible) regardless of the resulting separator size, and hence the estimate $\eta(s)$ not be computed.

We illustrate these concepts for the example of Figure 3, working with the x dimension. If $\alpha = 1/3$, then a separating value of either $s = 3$ or $s = 4$ satisfies the balance criterion. Calculating the estimated separator size for each of these va we get $\eta(3) = 3$ and $\eta(4) = 2$, so that we would choose $s = 4$ as the best separating value in this case. If $\alpha = 1/5$ instead, then any separating value in the interval [would satisfy the balance criterion, but the estimated separator sizes would stil $s = 4$ as the best choice.

We now sketch an algorithm for computing a separating value that minimizes the approximate separator size subject to the specified balance constraint. This atively simple serial algorithm serves to introduce appropriate terminology, not and data structures, providing a framework for our subsequent development of a dis tributed parallel algorithm. For definiteness, assume that we are working with t x coordinate dimension; similar definitions and procedures are also applicable to y dimension. In general, we process both dimensions in the same fashion and use whichever yields the smaller separator. When this procedure for computing separa tors is used repeatedly in nested dissection, a different coordinate dimension may selected at each stage.

For a given graph $G = (V, E)$, the vertices in V are maintained in a *vertex list*, which we denote by $list(V, x)$, in increasing order of their x coordinate values. T vertex list is traversed to compute a *vertex count list*, denoted by $count(V, x)$, of counts of vertices in G at each coordinate value, in increasing order in the x dimension. vertex count list has the form $count(V, x) = [\langle c_j, j\rangle]$, where G is chosen in one of the two coordinate dimensions, say x. We will refer to s as a “sepa-rating value” because it will be used to dissect the graph along the given coordin dimension. Let U_1, U_2, and U_s be the sets of all vertices whose x coordinate is less than s, greater than s, and equal to s, respectively. This partitioning of the no in the graph does not necessarily give us a vertex separator, because there may st be paths connecting vertices in U_1 and U_2. However, any such path must contain an edge that “straddles” the separating value s. Let E_s be the set of all such straddle edges, i.e.,

$$E_s = \{(u_1, u_2) \in E : u_1 \in U_1,\ u_2 \in U_2\}.$$

For each edge $(u_1, u_2) \in E_s$, arbitrarily select one of its two associated vertices for inclusion in the set C_s, which we refer to as the “correction set” for V_s. We now define the following sets:

$$V_1 = U_1 \setminus C_s, \quad V_2 = U_2 \setminus C_s, \quad V_s = U_s \cup C_s.$$

The set V_s is a vertex separator for the graph, since each vertex in V_1 is connected only to vertices in V_s or other vertices in V_1, and similarly for V_2. We refer to such a separator as a “Cartesian separator;” henceforth, when we use the term separator w will mean a Cartesian separator.

We illustrate these concepts for the example of Figure 3. Using $s = 3$ as a separating value in the x dimension, we get the initial sets

$$U_1 = \{1, 2, 6, 8, 12, 13, 14\}, \quad U_2 = \{3, 4, 5, 7, 10, 11\}, \quad U_s = \{9, 15\}.$$

The set of straddle edges is the singleton set $E_s = \{(2, 3)\}$. Choosing one of the endpoints of this edge, say node 2, we get the correction set $C_s = \{2\}$. Thus, the final subgraphs and separator are given by

$$V_1 = \{1, 6, 8, 12, 13, 14\}, \quad V_2 = \{3, 4, 5, 7, 10, 11\}, \quad V_s = \{2, 9, 15\}.$$

It is not difficult to devise graphs for which even the best Cartesian separato is much larger than necessary. For example, a one-dimensional grid wound into a spiral in the plane will be cut many times by any bisecting line, but can be separat evenly by removing a single vertex. Similarly, a planar graph consisting of n conc tric squares whose corresponding corners are connected can be separated evenly b removing only four vertices, yet any bisecting line will cut $2n$ edges, giving a se rator of size n. However, we have found Cartesian separators to be very effective f separating graphs that arise in practice. In the next sections we proceed to dis the two main subproblems in computing a Cartesian separator:

- Determining an appropriate choice for the separating value s,
- Determining the correction set C_s

3.1. Choosing a Separating Value. As we observed earlier, the two main criteria for choosing a separator are that the separator be small and that the resul subgraphs be well balanced (i.e., about equal in size). These criteria are genera conflict, so there is a tradeoff between them. In choosing a separating value s fo computing a Cartesian separator in a given dimension, the balance between the size of the resulting subgraphs is determined by the relative numbers of vertices hav coordinates less than s or greater than s in that dimension. Thus, we can attai any desired degree of balance, including optimal balance, simply by counting verti

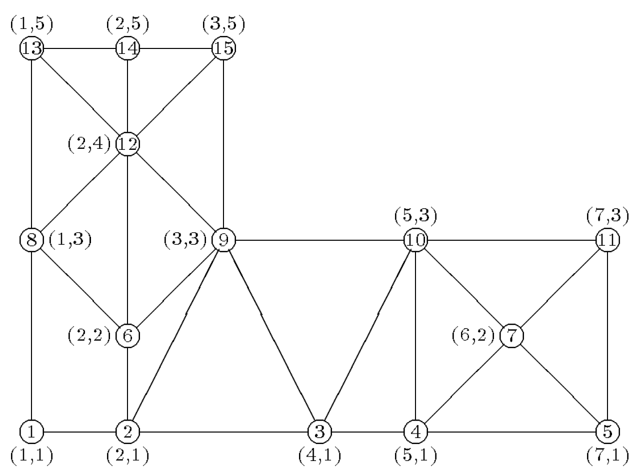


FIG. 3 . *Finite element graph with Cartesian coordinates of nodes shown.*

especially *before* the graph has been partitioned so that data locality can be main-tained (i.e., contiguous pieces are assigned to individual processors, and "ne pieces assigned to "nearby" processors). More recent heuristics for computing gr separators include spectral methods [8, 15] and methods based on geometric projec-tions and mappings [12, 13, 14, 17]. These may have greater potential for parallel implementation, but this has yet to be demonstrated in practice. An explicitly pa allel implementation of the Kernighan-Lin algorithm for computing graph separato can be found in [6].

In this paper we present another new approach to computing separators, one that is designed to be effective in a distributed parallel environment. Its principal fe are the use of Cartesian coordinates for the vertices, its lack of dependence on i data locality, and the control it provides over both the size of the separator and balance of the resulting pieces. This technique is used recursively to produce a n dissection ordering. A somewhat similar "recursive bisection" approach, based on geographic locations of points or particles, has also been used in other contexts, as domain decomposition [1] and load balancing of parallel computations [2, 4, 30]. However, these efforts have been concerned primarily with the numerical balance o the partitioning rather than the interconnectivity among the points, if any, or sizes of the separators used.

2.2. Cartesian Representation. One motivation for our use of a Cartesian representation of the graph is to make the data "self identifying." This will be im tant when we consider implementing the algorithm on distributed memory parallel computers. In particular, the data can be scattered randomly across the local mem ories of the processors, yet we can still tell where (geographically) any given pi data lies within the overall problem, without needing any communication to establi context. In effect, this approach makes the distributed memory "content address-able," thereby reducing much of the problem of computing separators to relativel simple counting and searching operations, which can be done very effectively in distributed manner.

For each vertex $v \in V$ we assume that we are given a pair of Cartesian coordinates, which we denote by $x(v)$ and $y(v)$, representing the horizontal and vertical coordina directions, respectively, in the Euclidean plane. One might wish to apply a rotat to the coordinate system to place the graph into some more advantageous orientatio we assume that this has already been done, if desired. One possible way to determin a good orientation would be to compute the axis of minimum inertia of the vertices as a collection of points in the plane.

As will be seen shortly, the efficiency of our method depends on both the range and the "occupancy rate" of the possible coordinate values in each dimension. Ther fore, we "integerize" the original "natural" coordinate values by sorting them in dimension and then reassigning consecutive integer values to distinct coordinate ues in sequence. The basis for this strategy is to try to minimize the range of val while ensuring that all coordinate values are actually used, since unused values waste space and time in our algorithms. Such an operation may significantly dis-tort the original metric geometry of the graph, but it does not change its topologi structure, thereby enhancing efficiency while retaining the effectiveness of our proach in finding good separators. Figure 3 shows our example graph with Cartesian coordinates for the nodes.

3. Cartesian Separators. We now describe our strategy for computing a ver-tex separator in a Cartesian labeled graph $G = (V, E)$. Let s be a coordinate value

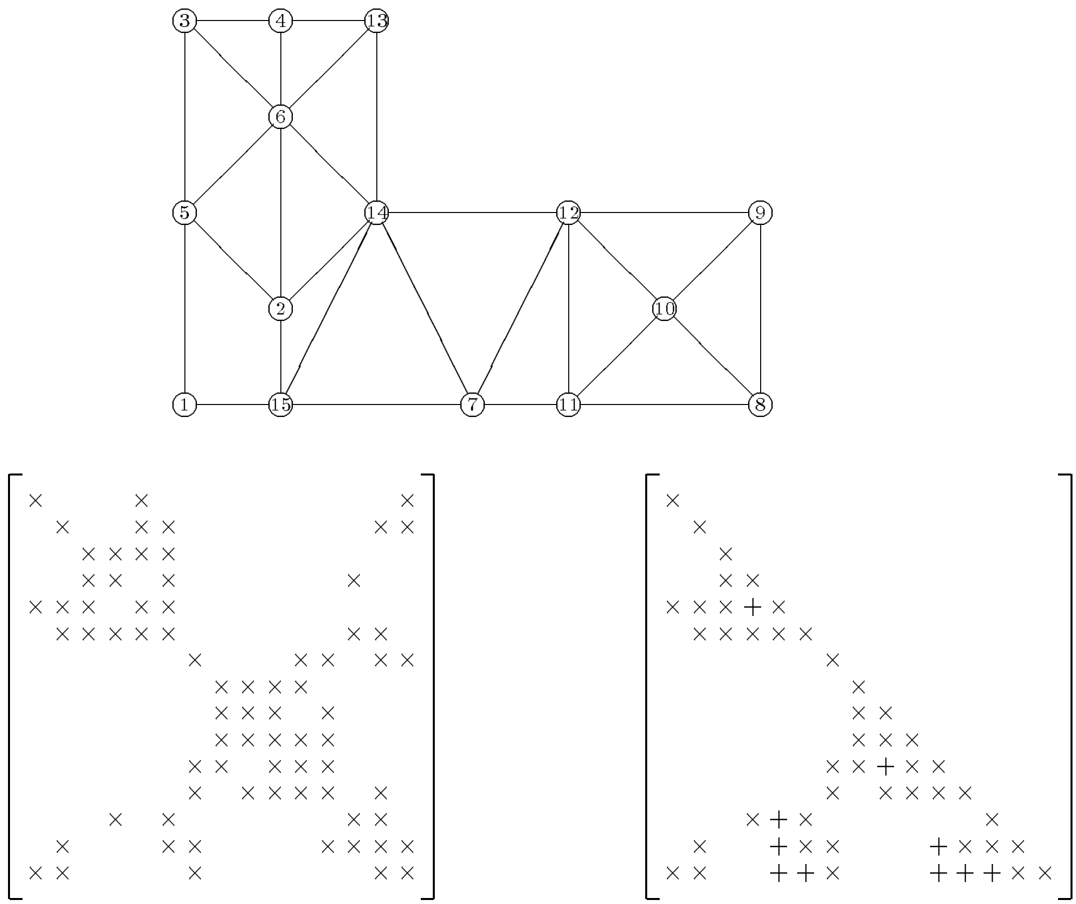


FIG. 2. *Finite element graph reordered by nested dissection (top) and the nonzero patterns of the corresponding sparse matrix (left) and its Cholesky factor (right), with fill indicated by* $+$.

If the matrix is reordered so that the vertices within each subgraph are numbere contiguously and the vertices in the separator are numbered last, then the matrix w have the following bordered block diagonal form:

$$A = \begin{bmatrix} A_1 & 0 & S_1 \\ 0 & A_2 & S_2 \\ S_1^T & S_2^T & A_s \end{bmatrix}.$$

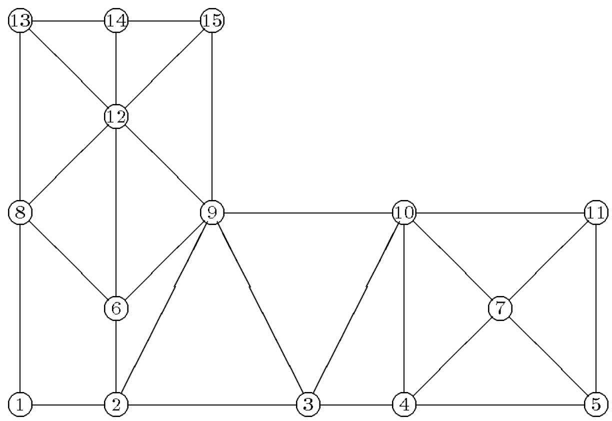
The significance of the above partitioning of the matrix is twofold: first, the zero bl are preserved in the factorization, thereby limiting fill; second, factorization matrices A_1 and A_2 can proceed independently, thereby enabling parallel execution on separate processors. This idea can be applied recursively, breaking each subgr into smaller and smaller pieces with successive separators, giving a nested sequ of dissections of the graph that inhibit fill and promote concurrency at each level

Figure 2 shows our original example reordered by nested dissection. In the subs quent Cholesky factorization, the reordered matrix suffers considerably less fill with the original ordering, and also permits greater parallelism. For example, col 1, 2, 3, 7, and 8 of the Cholesky factor depend on no prior columns, and hence can be computed simultaneously, whereas in the original ordering every column of th Cholesky factor depends on the immediately preceding column.

The effectiveness of nested dissection in limiting fill depends on the size of t separators that split the graph, with smaller separators obviously being better. highly regular, planar problems (e.g., two-dimensional finite difference or finite ement grids), suitably small separators can usually be found [9]. For problems in dimensions higher than two, or for highly irregular problems with less localized nectivity, nested dissection tends to be less effective, but so do most other orde heuristics, which explains why iterative methods are often preferred over direct m ods in such circumstances. In this paper we will focus on problems for which an embedding of the graph in the two-dimensional Euclidean plane is given, but whose graph is not necessarily planar. Such a problem might result, for example, fro two dimensional finite element structural analysis. Indeed, our test problems are tained from standard commercial structural analysis packages, which routinely sup Cartesian coordinates for the vertices. Our approach appears to generalize to th dimensions in a reasonably straightforward manner, but such an implementation has not yet been done, and its effectiveness in such a setting remains to be demonstrate

In addition to the size of a separator, the relative sizes of the resulting subgra also important. Maximum benefit from the divide-and-conquer approach is obtained when the remaining subgraphs are of about the same size; an effective nested dissecti algorithm should not permit an arbitrarily skewed ratio between the sizes of the pie In a parallel setting, this criterion takes on additional significance in that it determines the load balance of the computational subtasks assigned to individu processors. Thus, the algorithms we develop will take into account both size an balance in choosing separators.

Nested dissection algorithms differ primarily in the heuristics used for choos separators. A typical approach to automatic nested dissection for irregular gra [4] involves first finding a “peripheral” vertex, generating a level structure base the connectivity of the graph, and then choosing a “middle” level of vertices as t separator. Such an approach is difficult to implement efficiently on a distributed par allel computer for a number of reasons, including the necessary serialization of of the steps, and the communication required to assess the connectivity of the gra



$$\left[\begin{array}{ccccccccccccccc}
\times & \times & & & & & & \times & & & & & & & \\
\times & \times & \times & & & \times & & & \times & & & & & & \\
 & \times & \times & \times & & & & & \times & \times & & & & & \\
 & & \times & \times & \times & & \times & & & \times & & & & & \\
 & & & \times & \times & & \times & & & & \times & & & & \\
 & \times & & & & \times & & \times & \times & & & \times & & & \\
 & & & \times & \times & & \times & & & \times & \times & & & & \\
\times & & & & & \times & & \times & & & & \times & \times & & \\
 & \times & \times & & & \times & & & \times & \times & & \times & & & \times \\
 & & \times & \times & & & \times & & \times & \times & \times & & & & \\
 & & & & \times & & \times & & & \times & \times & & & & \\
 & & & & & \times & & \times & \times & & & \times & \times & \times & \times \\
 & & & & & & & \times & & & & \times & \times & \times & \\
 & & & & & & & & & & & \times & \times & \times & \times \\
 & & & & & & & & \times & & & \times & & \times & \times
\end{array}\right]
\qquad
\left[\begin{array}{ccccccccccccccc}
\times & & & & & & & & & & & & & & \\
\times & \times & & & & & & & & & & & & & \\
 & \times & \times & & & & & & & & & & & & \\
 & & \times & \times & & & & & & & & & & & \\
 & & & \times & \times & & & & & & & & & & \\
 & \times & + & + & + & \times & & & & & & & & & \\
 & & & \times & \times & + & \times & & & & & & & & \\
\times & + & + & + & + & \times & + & \times & & & & & & & \\
 & \times & \times & + & + & \times & + & + & \times & & & & & & \\
 & & \times & \times & + & + & \times & + & \times & \times & & & & & \\
 & & & & \times & + & \times & + & + & \times & \times & & & & \\
 & & & & & \times & + & \times & \times & + & + & \times & & & \\
 & & & & & & & \times & + & + & + & \times & \times & & \\
 & & & & & & & & & & & \times & \times & \times & \\
 & & & & & & & & \times & + & + & \times & + & \times & \times
\end{array}\right]$$

FIG. 1. *Example finite element graph (top) and the nonzero patterns of the corresponding sparse matrix (left) and its Cholesky factor (right), with fill indicated by +.*

promote concurrency in the factorization.

In this paper we are concerned with the problem of computing fill-reducing order-ings for symmetric positive definite sparse matrices that will enable efficient Chole factorization on large-scale, distributed-memory parallel computers. Perhaps the important consideration is that the ordering itself be computed in parallel on the s multiprocessor machine. Most previous work on parallel sparse matrix factorizat has focused on the more costly (and more easily parallelized) numeric phases, an has simply assumed that an appropriate and effective ordering could be precomputed on a serial machine (see [7] for a survey of this work). Such an approach is not scal-able, however, as any such serial phase will eventually become a bottleneck as t problem size and number of processors grow. We therefore seek a distributed parall ordering algorithm that can be integrated on the same machine with the subsequent parallel numeric computation and maintain reasonable efficiency over a wide range of parallel architectures and number of processors. Additional issues that will co us are the fill (and hence work and storage) that result from a given ordering, an also the resulting concurrency, load balance, and communication traffic in computin the Cholesky factor on such a parallel computer.

Designing an efficient, scalable, distributed ordering algorithm for sparse mat ces presents a formidable challenge. The best serial ordering algorithms have evo over an extended period of time and are extremely efficient. Much of this efficiency results from sophisticated data structures and algorithmic refinements that are d cult to extend to a distributed parallel setting. Moreover, many of these algorit involve inherently serial precedence constraints and have relatively little compu over which to amortize the communication necessary in a parallel implementation Perhaps most daunting of all, we seem to have a bootstrapping problem in that the efficiency of most distributed parallel algorithms depends on having a high degre of data locality, but we do not know how to partition our problem and distribute it across the processors until after we have an ordering. We therefore propose ordering algorithm that lends itself to a distributed parallel implementation w effectiveness does not depend on initial data locality.

2. Background Throughout this paper we will assume familiarity with numer-ous basic concepts in sparse matrix computations. Such background material can be found, for example, in the text [5]. In particular, we will use the standard graph model for sparse Gaussian elimination, which we explain briefly here. The graph of an $n \times n$ symmetric matrix A is an undirected graph having n vertices, with an edge between two vertices i and j if the corresponding entry a_{ij} is nonzero in the matrix. We use the notation $G = (V, E)$ to denote the vertex and edge sets, respectively, of a graph G. The structural effect of Gaussian elimination on the matrix is easily de-scribed in terms of the corresponding graph. The fill introduced into the matrix as result of eliminating a variable adds fill edges to the corresponding graph so that neighbors of the eliminated vertex become a clique. The elimination or factorizat process can thus be modeled by a sequence of graphs, each having one less vertex than the previous graph but possibly gaining edges, until only one vertex remains. small example graph and corresponding matrix A are shown in Figure 1. Also shown is the fill in the Cholesky factor L of the example matrix, where $A = LL^T$.

2.1. Nested Dissection Nested dissection is a divide-and-conquer strategy for ordering sparse matrices, originally due to Alan George [3]. Let a set S of vertices (called a separator) whose removal, along with all edges incident on vertices in V connects the graph into two remaining subgraphs, $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$.

A CARTESIAN PARALLEL NESTED DISSECTION ALGORITHM [*]

MICHAEL T. HEATH [†] AND PADMA RAGHAVAN [‡]

Abstract. This paper is concerned with the distributed parallel computation of an ordering for a symmetric positive definite sparse matrix. The purpose of the ordering is to limit fill and enhance concurrency in the subsequent computation of the Cholesky factorization of the matrix. We use a geometric approach to nested dissection based on a given Cartesian embedding of the graph of the matrix in Euclidean space. The resulting algorithm can be implemented efficiently on massively parallel, distributed memory computers. One unusual feature of the distributed algorithm is that its effectiveness does not depend strongly on data locality, which is critical in this context, since an appropriate partitioning of the problem is not known until after the ordering has been determined. The ordering algorithm is the first component in a suite of scalable parallel algorithms currently under development for solving large sparse linear systems on massively parallel computers.

Key words. parallel algorithms, sparse linear systems, ordering, Cartesian coordinates, nested dissection, Cholesky factorization

AMS(MOS) subject classifications. 65F, 65W

1. Introduction The ordering of the equations and unknowns in a sparse system of linear equations can have a dramatic effect on the computational work and storage required for solving the system by direct methods. The reason is that mos sparse systems suffer fill during the factorization process, that is, matrix entrie are initially zero become nonzero during the computation, and the amount of such fil depends strongly on the ordering of the rows and columns of the matrix. Thus, ordering sparse matrices for efficient factorization is an important step in solving m large-scale computational problems in science and engineering, such as finite ele structural analysis. In general, finding an ordering that minimizes fill is a very d cult combinatorial problem (NP-complete). Practical sparse factorization algori are therefore based on heuristically chosen orderings that are reasonably effecti limiting fill, but much less costly to compute than the true optimum. Some of the most commonly used ordering heuristics are minimum degree, nested dissection, and various schemes for reducing the bandwidth or profile of the matrix.

In addition to determining fill, the ordering also affects the potential paralle that can be exploited in factoring the matrix. These two considerations – reducing and enhancing parallelism – are largely compatible, but by no means coincident objec tives. Sparsity and parallelism are positively correlated to some extent, since s implies a lack of interconnections among matrix elements that often translates i computational subtasks that can be executed independently on different processor This relationship is extremely complicated, however, and parallel efficiency deper on many other considerations as well, such as load balance and communication traffic. Thus, for example, minimum degree is in many cases the most effective heuristic known for limiting fill, but may produce orderings for which the natural load balance is uneven in parallel factorization. As another example, band-oriented metho however effective they may or may not be in limiting fill, tend to inhibit rather tha

---

[*] This research was supported by the Defense Advanced Research Projects Agency through the Army Research Office under contract number DAAL03-91-C-0047.

[†] Department of Computer Science and National Center for Supercomputing Applications, University of Illinois, 405 N. Mathews Ave., Urbana, IL 61801.

[‡] National Center for Supercomputing Applications, University of Illinois, 405 N. Mathews Ave., Urbana, IL 61801.