PARALLEL ORDERING USING EDGE CONTRACTION *

PADMA RAGHAVAN

{TECHNICAL REPORT: CS-95-293} †

Abstract. Computing a fill-reducing ordering of a sparse matrix is a central problem in the solution of sparse linear systems using direct methods. In recent years, there has been significant research in developing a sparse direct solver suitable for message-passing multiprocessors. However, computing the ordering step in parallel remains a challenge and there are very few methods available. This paper describes a new scheme called parallel contracted ordering which is a combination of a new parallel nested dissection heuristic and any serial ordering method. The new nested dissection heuristic called Shrink-Split ND (SSND) is based on parallel graph contraction. For a system with N unknowns, the complexity of SSND is $O(\frac{N}{P}\log P)$ using P processors in a hypercube; the overall complexity is $O(\frac{N}{P}\log N)$ when the serial ordering method chosen is graph exploration based nested dissection. We provide extensive empirical results on the quality of the ordering. We also report on the parallel performance of a preliminary implementation on three different message passing multiprocessors.

Key words. parallel algorithms, sparse linear systems, fill-in, ordering, sparse matrix factorization, nested dissection, parallel nested dissection

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

1. Introduction.

Consider the problem of solving a very large system of linear questions using Gaussian elimination when the coefficient matrix is sparse. The goal is to utilize sparsity to achieve low complexity for the solution process. In this regard it is a well known fact that the system must be reordered to incur low fill-in (original zeroes that become nonzero) during Gaussian elimination. When the coefficient matrix A is symmetric positive definite, the solution process (using Cholesky factorization) is organized into four distinct steps [9]:

- Ordering: Find a permutation matrix P so that PAP^T has a sparse Cholesky factor L.
- Symbolic factorization: Determine the structure of L and set up data structures.
- Numeric factorization: Compute L; $PAP^T = LL^T$.
- Triangular Solution: Solve the triangular systems $Ly = Pb$ and $L^T z = y$ to compute $x = P^T z$.

In the last several years there has been significant interest in a fully parallel solution using very large message passing multiprocessors. Although effective parallelization of symbolic and numeric factorization is well understood, computing orderings in parallel remains a very challenging task for which there are but very few methods available.

Most earlier work on parallel sparse matrix factorization has addressed the numeric factorization step which has the greatest serial complexity [12]. However, the serial complexity of the ordering step is such that if it were not parallelized it would soon become a bottleneck. For example, for a class of $N \times N$ sparse matrices, the serial complexity of nested dissection is $O(NlogN)$ while that of the numeric factorization is $O(N^{3/2})$. Furthermore, performing the ordering in serial makes it impractical to

---

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

† Department of Computer Science, The University of Tennessee, 107 Ayres Hall, Knoxville, TN 37996-1301

have fully parallel sparse direct solver be the main compute kernel in any large scale parallel scientific application.

In the serial case, the key purpose of ordering is to reduce fill-in (the problem of minimizing fill if NP-hard) during numeric factorization. The ordering also determines the column dependencies in computing the Cholesky factor; the dependencies can be expressed as an "elimination tree" representing the large-grain functional or task parallelism available in numeric phase. Of the several ordering heuristics that have been studied, the Multiple Minimum Degree [7, 9] ordering tends to incur the least fill over a wide class of problems but the elimination trees tend to be tall and unbalanced. On the other hand, the class of nested dissection [9] methods (which typically incur more fill-in than MMD) lead to elimination trees that are short and better balanced. So nested dissection orderings seem more appropriate in the parallel context. A related fact is that nested dissection methods are divide and conquer methods and so task parallelism during ordering can be utilized by parallel divide and conquer.

In this paper we develop a new parallel nested dissection scheme called Shrink-Split nested dissection (SSND). We apply SSND till we reduce the ordering problem to that of ordering as many disjoint submatrices as the total number of available processors. We show that SSND has the parallel complexity of $O(\frac{N}{P}\log P)$ on P processors of a hypercube. After SSND, we can essentially apply any serial ordering scheme independently on each processor and we choose to apply Multiple Minimum Degree. The overall ordering is called Parallel Contracted Ordering (PCO). We show that PCO is effective both in reducing fill and achieving good parallel performance. We begin with a brief review of background and related work in Section 2 and then present our main contributions in Sections 3 and 4. We develop our algorithm in Section 3 and analyze its parallel complexity in Section 3.1. Section 4 contains empirical results and is organized into two parts; the first relates to fill reduction while the second is on parallel performance. Conclusions are presented in Section 5 along with a discussion of related work.

2. Background and Related Research. We assume familiarity on the part of the reader with basic graph theoretic notation and key ideas in sparse matrix computations [9]. We use $G(A)$ to denote the graph of the $N \times N$ symmetric positive definite matrix A; $G(A) = (V, E)$ is undirected, vertices in V correspond to each row or column of A and E contains an edge between vertices i and j whenever $A_{i,j} \neq 0$. This graph can be used both to mimic the changes to the sparsity structure of A in the course of eliminating unknowns, and to predict fill and thereby the sparsity of L for a given numbering of vertices. The ordering methods Multiple Minimum Degree and Nested Dissection can be easily explained in terms of this graph.

Consider $G(A)$; eliminating the first unknown, will cause fill which is modeled by making all vertices adjacent to vertex 1 pairwise adjacent, fill edges added to make the set of vertices adjacent to vertex 1 into a clique. This resulting clique is of small size if the degree of the vertex being eliminated is low. The "minimum degree" strategy is that of locally reducing fill by picking the vertex to be eliminated as the one with fewest neighbors. In rather oversimplified terms, given $G(A)$, a new numbering of vertices in V is computed by picking the vertex with the smallest degree and assigning it the lowest unused number. Then the process of eliminating the unknown corresponding to this vertex is mimicked by adding fill edges to make the adjacency set of this vertex into a clique. The numbered vertex is removed from the graph which is then used for yet another step. This process is repeated till all vertices are numbered. Multiple Minimum Degree(MMD) developed by Alan George and Liu

[7, 9] is a highly sophisticated implementation of the "minimum degree" heuristic. The MMD implementation invariably obtains orderings with remarkably low fill. On the other hand, the elimination tree tends to be tall and unbalanced and so would conceivably require a new data partitioning strategy for the numeric phase. Finally, it does not seem clear how to parallelize this ordering method itself; Rothberg has recently studied this problem in [21].

Alternatively, paths in $G(A)$ have special significance for edges in the graph of L $G(L)$ or in other words nonzeroes in L. Given an ordering of vertices in V, there is an edge between vertices k, l in $G(L)$ if and only if there is a path between k and l in $G(A)$ consisting of vertices numbered lower than k and l. The class of nested dissection methods originally developed by Alan George relates to this characterization. Let V_s be a set of vertices (called a separator) whose removal, along with all edges incident on vertices in V_s, disconnects the graph into two remaining subgraphs, $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. If the matrix is reordered so that the vertices within each subgraph are numbered contiguously and the vertices in the separator are numbered last, then there cannot be any edges in $G(L)$ between vertices in $V1$ and $V2$. If the matrix is permuted accordingly, it will have the following form:

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

In this form, the zero blocks are preserved and factorization if A_1 and A_2 can proceed independently and thereby in parallel. This idea can be applied recursively thereby nesting the dissections; after $\log P$ stages, note that there will be a total of P disjoint subgraphs (submatrices). The large grain parallelism for the numeric step is defined clearly by this dissection process. The fill-in relates to the size of the separators and in broad terms, the smaller the separator, the lower the fill.

Nested dissection algorithms differ primarily in the heuristics used for choosing separators. An approach called Automatic Nested Dissection [8] involves first finding a "peripheral" vertex, generating a level structure based on the connectivity of the graph, and then choosing a "middle" level of vertices as the separator. More recent heuristics for computing graph separators include spectral methods [20], methods based on geometric projections and mappings [10, 19] and graph contraction based heuristics [4, 15].

Parallel nested dissection requires an effective parallel method for computing a separator. If such a method is available then parallel divide and conquer would naturally provide functional or task parallelism. For example, if a separator can be computed effectively in parallel for the entire graph G using all P processors, then at the next step, at expense of exchanging some data, G_1 and G_2 could be gathered onto processor subsets of size $P/2$ which could then proceed with the recursion in parallel.

The problem of computing separators in parallel is related to a fundamental one for distributed computing, that of data (graph) partitioning to provide locality. If the data or computation to be mapped to processors in a parallel machine is represented as a graph then the "graph partitioning" problem is to find a small *edge* separator to split the graph into parts. Recursive application will lead to a multiway partition. Vertex and edge separators are closely related but for ordering sparse matrices (nested dissection) we need vertex separators and a special numbering. We will discuss ideas used in graph partitioning methods as they relate to our work but a direct comparison with them is not viable.

Breadth first search on sparse graphs has limited parallelism and a scalable implementation would not be possible unless the graph were already partitioned among processors; obviously such a partitioning requires the computation of separators. The same is true of spectral methods; the sparse matrix (and thereby its graph) must be partitioned among processors to enable effective parallel sparse matrix vector multiplication.

The first parallel nested dissection method was developed by Gilbert and Zmijewski [11, 22] and was based on the Kernighan-Lin heuristic. However, the method requires storage proportional to the number of edges in $G(A)$ at a single processor. The method uses the functional parallelism in parallel divide and conquer. The first separator was computed essentially on a single processor the next two independently and in parallel on two processors and so on. Our SSND uses a parallel divide and conquer approach but it also parallelizes the task of computing a single separator.

The second parallel method called Cartesian Nested Dissection (CND) was developed by Heath and Raghavan [14]. CND is suitable for sparse matrices associated with geometric information. For such matrices, the geometric information is used to compute a separator in an efficient "data-parallel" manner; the implementation does not require an initial locality preserving partitioning of the graph to the processors. CND has been shown to be a fast and scalable ordering method [14].

Our SSND algorithm was motivated by recent developments in serial ordering and graph partitioning techniques. Bui and Jones[4] developed a new nested dissection scheme based on heuristic for computing separators using graph contraction and the Kernighan-Lin method. Hendrickson and Leyland [15] present a multilevel graph partitioning scheme where they used graph contraction and the spectral method to compute edge separators. Yet another related work is that of Barnard and Simon [1] for graph partitioning using recursive spectral bisection.

In their algorithms, both Bui and Jones [4] and Hendrickson and Leland [15] compute a sequence of "contracted" graphs. The first graph in the sequence is the original one, the second is of half the size and so on till the last one (coarsest) has a small number (≈ 100) of vertices. A bisection is first computed in the coarsest graph, next it is projected to the preceding one and refined further to compute a bisection, this bisection is in turn projected to the next graph and so on till a bisection of the original graph is produced. In short, to compute a single separator, a sequence of contracted graphs is produced and a separator is computed in the coarsest graph and refined through the sequence to obtain final separator. This process of contracting the graph and refining separators is applied recursively to compute a nested dissection ordering. These methods with contracted graphs were aimed at improving the quality of the separator and indeed resulted in separators of significantly smaller size [4, 15]. In this work, we use the graph contraction idea but in parallel and for a different purpose. The contracted graphs are used to enable computing a separator in parallel and not to "improve" the quality of the separator. Secondly, the sequence of contracted graphs is also used to enable parallel divide and conquer. Finally, the graph contraction is performed exactly once and is used to compute all the separators.

Concurrent with this work, Barnard and Simon [2] and Karypis and Kumar [16] have developed parallel formulations of multilevel partitioning methods. Barnard and Simon parallelize their earlier recursive spectral scheme [1] for graph partitioning. Karypis and Kumar also parallelize the multilevel approach and develop a parallel nested dissection scheme. Their graph contraction method differs from than in SSND; the heuristic for computing a separator is also different. Our work differs from both these schemes in many respects but it does share a common feature, that of using parallel graph contraction. PCO is implemented on message passing distributed memory machines such as the Intel Delta while the algorithms in [2, 16] are on the Cray T3D using "shmem," the shared memory library. Unlike the other schemes, PCO uses contracted graphs to compute a separator in parallel, not to refine and improve the quality of the separator. The graph contraction is performed exactly once in PCO and then used to apply the parallel divide and conquer; in the other schemes, graph contraction is performed at every stage in the recursion. The heuristics used for computing the separator in PCO is different from those used in the other two scheme. Further comparison with the multilevel nested dissection scheme of Karypis and Kumar is provided in Section 4.

3. Algorithms. Our PCO algorithm has two phases, a distributed phase followed by a local phase. In the distributed phase processors cooperate to compute a nested dissection ordering; we call the algorithm Shrink-Split ND (SSND). In the local phase any serial ordering scheme can be used independently on each processor. We use Multiple Minimum Degree (MMD) in our implementation. In this section we will describe our SSND heuristic.

We now introduce notation that is used in the rest of the paper. We consider $G(A)$, the graph of the matrix A with a total of N vertices; we assume that the total number of edges M is a small constant times N. We use P to denote the total number of processors and we use π_j to denote the j-th processor for $j = 0, 1, \cdots (P-1)$. For any graph $G = (V, E)$, $G(p)$ denotes a distributed representation of G over some p processors. In such a representation, the vertex set is partitioned over the processors and every vertex is owned by a distinct processor. Each processor owns roughly $|V|/p$ vertices and the edges incident on these vertices; no "locality" is assumed. For graphs G and $\hat{G}$, we use $G(p/2)$ and $\hat{G}(p/2)$, to denote that each graph is distributed over processor subsets of size $p/2$ and that these processor subsets are *disjoint*. Initially, $G_0 = G(A)$ is assumed distributed over all P processors and the algorithm starts with $G_0(P)$.

SSND uses an efficient parallel graph contraction scheme aimed at reducing the size of $G(A)$ to a small number; this step is called `shrink`. This step produces a sequence of graphs $G_0(P), G_1(P) \cdots, G_k(P)$ with $k = \log_2 P - 1$. The number of vertices in G_{i+1} is roughly half that in G_i and the last graph G_k is the smallest with less than N/P vertices. The last graph $G_k(P)$ and the first $G_0(P)$ are used in a next step called `split`, to compute a separator and produce a bisection in each graph in the sequence. The bisected shrunk graphs are used to "nest" the dissections, i.e, apply `split` recursively using select shrunk subgraphs to generate a set of $P-1$ separators and hence P subgraphs. This completes the distributed phase; each of the subgraphs is then mapped to a processor which applies MMD in a local phase. Using this notation introduced earlier, we provide a brief outline of the SSND algorithm in Figure 1.

The `shrink` step produces a sequence distributed "shrunk" graphs. It starts with $G_0(P)$, the original graph distributed on P processors. During `shrink`, the next graph in the sequence is obtained by performing a small number of parallel edge contraction steps. Consider computing $G_{i+1}(P)$ from $G_i(P)$. The set of processors of size P is partitioned into $P/2$ processor pairs. Each processor pair computes a set of edges that form a matching (i.e., no two edges share a common end point) and the vertices at end points of these edges are "owned" by the two processors. Now the union of such matchings over all $P/2$ processor pairs (denoted by $\dot{M}$) is also a matching because the

Parallel `shrink-split` ND:

- `shrink`: parallel edge-contract to obtain a sequence of distributed shrunk graphs:
  $G_0(P), G_1(P), \cdots G_k(P)$, $k = \log_2(P) - 1$
- assign $p \leftarrow P$
- `split` using $G_0(p) \cdots G_k(p)$:
  1. separate $G_k(p)$ and project to $G_0(p)$
     separate $G_0(p)$ into $\tilde{G}_0(p)$ and $\hat{G}_0(p)$
  2. mark split through shrunk graphs to get two sequences:
     $\tilde{G}_0(p), \tilde{G}_1(p), \cdots \tilde{G}_{k-1}(p)$
     $\hat{G}_0(p), \hat{G}_1(p), \cdots \hat{G}_{k-1}(p)$
  3. redistribute to place each sequence on disjoint processor subsets of size $p/2$ to get:
     $\tilde{G}_0(p/2), \tilde{G}_1(p/2), \cdots \tilde{G}_{k-1}(p/2)$
     $\hat{G}_0(p/2), \hat{G}_1(p/2), \cdots \hat{G}_{k-1}(p/2)$
- `nest`: while $k > 0$
  1. assign $k \leftarrow k - 1$;
  2. assign $p \leftarrow p/2$;
  3. do in **parallel** on disjoint processor subsets of size p
     `split` using $\tilde{G}_0(p), \tilde{G}_1(p), \cdots \tilde{G}_k(p)$
     `split` using $\hat{G}_0(p), \hat{G}_1(p), \cdots \hat{G}_k(p)$

FIG. 1. *Summary of key steps in* `Shrink-split` *ND;* G_0 *is original graph and* P *is the total number of processors*

vertex to processor assignment is a partition. Next each edge $(v, w) \in \dot{M}$ is replaced a by single vertex whose adjacency set is the union of those of v and w. Let the graph contracted using $\dot{M}$ be $\dot{G}(P)$; if $1/2|G_i(P)| < |\dot{G}(P)| < (1/3)|G_i(P)|$, then $\dot{G}(P)$ is made the new $G_{i+1}(P)$. However, if the size criterion is not satisfied, the process of computing a matching is repeated using $\dot{G}(P)$. The new matching $\ddot{M}$, is obtained using a different partition of the processors into pairs. The processor pair partition is computed in a greedy fashion to maximize the number of edges common to any pair and this in turn helps to get larger sized matchings over all processors. This processor pairing technique along with the fact that most graphs have average degree greater than 4 helps to cut down the number of matching steps. Typically, we have observed two or three intermediate steps in reducing $G_i(P)$ to the next $G_{i+1}(P)$. At the end of `shrink`, a sequence of $log_2 P$ distributed shrunk graphs are available:

$$G_0(P), G_1(P), \cdots, G_k(P), \quad k = log_2 P - 1.$$

Note that a vertex in $G_i(P)$ could contain a pair of vertices in $G_{i-1}(P)$ and hence a group of vertices in the original graph $G_0(P)$.

The `split` step bisects and redistributes a subsequence of shrunk graphs to allow parallel divide and conquer. For example, given $G_0(p), G_1(p), \cdots G_l(p)$, $l \leq log_2 P - 1$, and a p processor subset, `split` will bisect each graph $G_i(p)$ (except the last one) in the sequence to produce two sequences:

$$\tilde{G}_0(p), \tilde{G}_1(p), \cdots \tilde{G}_{l-1}(p) \text{ and } \hat{G}_0(p), \hat{G}_1(p), \cdots \hat{G}_{l-1}(p).$$

The processor subset is then partitioned into two subsets of size $p/2$ and the graphs are redistributed so that all graphs in each sequence are distributed over exactly one of the two processor subsets. Now `split` can be applied in parallel in each of the two processor subsets to carry the recursion one step further. The recursion ends on computing $P-1$ separators to dissect the graph into P subgraphs. By the parallel divide and conquer nature of `split`, each of the processors ends up having exactly one of the P subgraphs. The `split` step has three parts, "separate," "mark," and "redistribute." We will describe each one for the first time `split` " is invoked with the sequence of subgraphs from $G_0 \cdots G_k$ using all P processors, i.e., $G_0(P), G_1(P), \cdots G_k(P), k = \log_2 P - 1$.

To motivate our "separate" procedure we now relate the size of a separator in the contracted graph to that of the separator in the original graph. Consider the class of N vertex planar graphs which have of separators of size $c\sqrt{N}$ (c a small constant) [18]. Because planarity is preserved under edge contraction, each graph in the sequence of graphs defined at the end of `split`, i.e., $G_0, G_1, \cdots G_k$ satisfies the planarity condition. If we assume that each G_{i+1} has at most twice the number of vertices in G_i, the separator in G_i is of size $c\frac{\sqrt{N}}{2^{i/2}}$. The separator S_k in G_k is of size $c\frac{\sqrt{N}}{\sqrt{P}}$. Consider expanding S_k in G_k in terms of the original vertices, i.e., by replacing each vertex by the set of vertices in G_0 that it represents. This expanded set can have at most P times the size of S_k, that is at most $c\sqrt{P}\sqrt{N}$ vertices. This set forms a separator in G_0. It is obviously larger than the best separator in G_0 but at worst only by a factor of $\sqrt{P}$ whereas the size of the graph G_0 is roughly P times that of G_k. We use the expanded set of S_k (the separator in G_k) to compute a separator in G_0. In our experiments, the expansion leads to a set that is only about $4-8$ times the size of the final separator.

First the "separate," step; it ultimately computes a vertex separator in G_0. The the smallest graph $G_k(P)$ is first accumulated onto one processor, designated the leader of the group of processors. This is indeed feasible because the size of the graph is small enough. Next, the leader processor can apply any sequential algorithm to compute a separator in G_k. In our current implementation, we use the inexpensive level search based AND heuristic of Sparspak [8, 5]. Now each vertex in G_k is marked to be either in part A_k, B_k, or the separator S_k. This partition in G_k is projected to the graph at level 0, i.e., G_0. Recall that each vertex v in G_k represents a group of vertices in G_0; if $v \in A_k$ then all vertices in the group are assigned to part A. Likewise, if v is in $B_k(S_k)$, the group is assigned to part $B(S)$. Now A, B, S form a partition of G_0, S is a vertex separator but it tends to be large. Let A and B be represented by single vertices a and b. Consider the graph induced by the vertex set $\{S, a, b\}$; the edges include all those incident on S but with a minor modification. All those edges connecting a vertex v to those in A are replaced by a single edge (v, a), likewise edges connecting a vertex v to those in B are replaced by (v, b). This graph is bisected again using a level search approach. A level search is started from vertex a; if a vertex v is encountered which is adjacent to b, the vertex is not included in continuing the level search, instead it is included in the separator S_a; all other vertices in the explored in the level search are marked to be in part A. This process leads to a separator S_a of a certain size along with two parts. The same technique is applied starting with vertex b to compute a separator S_b. The smaller of S_a and S_b is selected as the final separator S_0 in $G_0(P)$. The vertices in S_0 are numbered as dictated by the nested dissection algorithm. Now there is a partition A_0, B_0 and S_0 in G_0.

We illustrate the `shrink` step as well as the the "separate" procedure of `split` for a small 5×5 grid graph in Figure 2.

Now to the "mark" step. The sets A_0 and B_0 in G_0 are used to mark out two subgraphs $\hat{G}_1$ and $\tilde{G}_1$ from G_1. These are used in turn to mark $\hat{G}_2$ and $\tilde{G}_2$ and so on till $\hat{G}_{k-1}$ and $\tilde{G}_{k-1}$. So now there are two sequences of shrunk graphs on all P processors:

$$\tilde{G}_0(P), \tilde{G}_1(P), \cdots \tilde{G}_{k-1}(P), \text{ and } \hat{G}_0(P), \hat{G}_1(P), \cdots \hat{G}_{k-1}(P)$$

Finally, the "redistribute" steps distributes the two sequences of subgraphs so the next step in nested dissection can proceed independently in two disjoint processor subsets. The set of P processors is partitioned into two halves of size $P/2$ each. The i-th processor in one half is paired with the i-th in the other. In each pair, a processor exchanges its portion of each $\hat{G}_i$ for its partner's portion of $\tilde{G}_i$. At the end of such exchanges the two sequences of shrunk graphs reside on disjoint processor subsets of size $P/2$; i.e., we achieve:

$$\tilde{G}_0(P/2), \tilde{G}_1(P/2), \cdots \tilde{G}_{k-1}(P/2), \text{ and } \hat{G}_0(P/2), \hat{G}_1(P/2), \cdots \hat{G}_{k-1}(P/2)$$

Notice that G_k is not used after `split` but since G_{k-1} has been bisected, each half has roughly the same size as the original G_k. The dissection is "nested" by applying `split` as described above but independently on each of the two processor subsets using $\tilde{G}_0(P/2), \tilde{G}_1(P/2), \cdots \tilde{G}_{k-1}(P/2)$ and $\hat{G}_0(P/2), \hat{G}_1(P/2), \cdots \hat{G}_{k-1}(P/2)$. After some $\log_2 P$ `split` steps, each processor group will have exactly one processor containing one subgraph. This completes SSND.

3.1. Parallel Complexity. We now derive expressions for the parallel complexity of SSND. Let P denote the total number of processors connected in a hypercube with cut-through routing. We assume that t_s is the startup cost, t_w is the per word cost, and that the per-link cost is negligible. In this section we use c_i, where i is a small integer to denote a suitable constant.

Recall that N is the total number of vertices in $G(A)$ and that the total number of edges is assumed to be a small constant times N. The vertices are partitioned among the processors and each processor is assigned edges incident on its vertices. Furthermore, each processor has no more than $N/P + c_0$ vertices. We also assume that $N/P \geq P \log_2 P$; this is justified given the large per processor memory of recent MIMD machines. For a graph G_i, we use N_i to denote the number of vertices and once agaian we assume the number of edges is a small constant times this value. If G_i is distributed over a set of processors and π_j belongs to this set, then we use $N_i(\pi_j)$ to denote the number of vertices of G_i on processor π_j.

Complexity of `shrink`. Recall that `shrink` produces a sequence of $k = \log_2 P - 1$ subgraphs of reduced size:

$$G_0(P), G_1(P), \cdots, G_k(P).$$

Now to the cost of computing $G_{i+1}(P)$ using $G_i(P)$. This is the cost of computing matchings using processor pairs. Although in our experience only a small constant number of intermediate matching steps are required, in the worst case each processor may need to communicate with all others. The processor pairing can be chosen so that all paths in every communication step are congestion free. The total computation

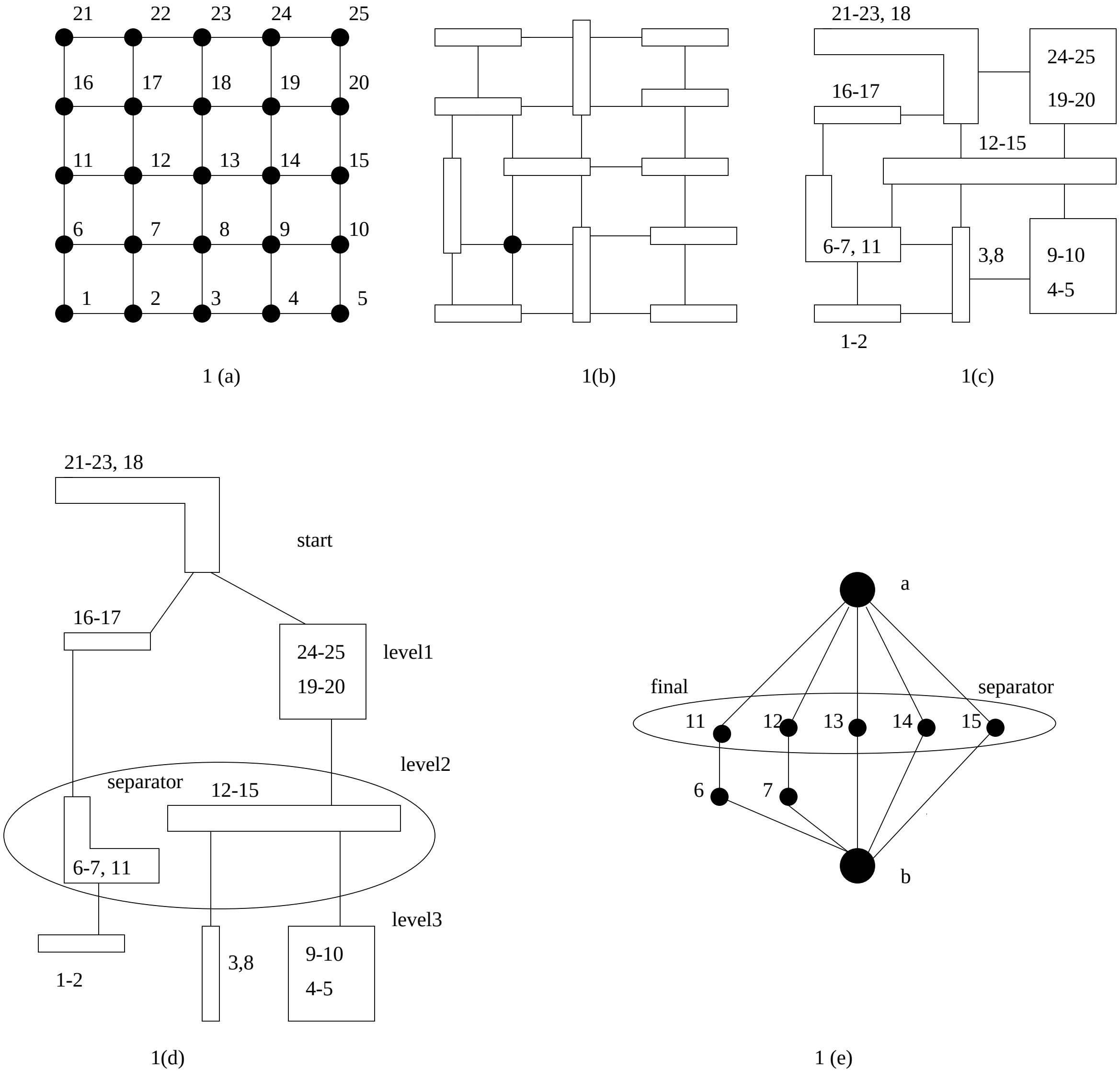

FIG. 2. *Shrink using* 5×5 *grid: 1(a)* G_0*, 1(b)* G_1*, 1(c)* G_2*; Separate: 1(d) level search in* G_2*, 1(e) computing separator in* G_0

cost is no more than that of each processor examining all its vertices and edges. The communication cost is bounded by that of the worst case scenario, all to all personalized communication with any processor π_j communicating at most a small constant times $N_i(\pi_j)$ integers to all other processors. The cost of this collective communication is $t_s(P-1) + t_w N_i(\pi_j)$. Under the assumption that each processor has at most some $N_i/P + c$ vertices, the total cost of the reducing $G_i(P)$ to $G_{i+1}(P)$ is:

$$c_1 \frac{N_i}{P} + t_w \frac{N_i}{P} + t_s(P-1).$$

Recall that N_{i+1}, the number of vertices in G_{i+1}, is roughly $(1/2)N_i$ and that the sizes of the shrunk graphs form a geometric progression. As a consequence, the total cost over all $k = log_2 P$ steps of `shrink` is at most:

$$cost(\texttt{shrink}) \quad = \quad 2c_1 \frac{N}{P} + 2t_w \frac{N}{P} + t_s(P-1)\log_2 P$$

Thus, the worst case complexity of `shrink` is $O(N/P)$.

Complexity of `split`. `Split` is composed of separate, mark, and redistribute steps. Furthermore, `split` itself is applied recursively, starting at $k = log_2 P - 1$ with P processors, next with $k-1$ and $P/2$ processors in each of two processor groups and so on. Let $\texttt{split}(i,p)$ denote `split` using a processor subset of size p on a subsequence of graphs $G_0(p), G_1(p), \cdots G_i(p)$. The graphs $G_0(p), G_1(p), \cdots G_i(p)$ are not necessarily the same as the original sequence of shrunk graphs but could represent a subsequence at any stage of the dissection. Let $cost(\texttt{split}, i, p)$ denote the cost of $\texttt{split}(i,p)$.

Consider $\texttt{split}(i,p)$; first of all, $G_i(p)$ and p processors are used to compute a separator and this separator is projected to $G_0(p)$, to compute a partition of $G_0(p)$. Finally, the partition in marked through the graphs $G_1(p), G_2(p) \cdots G_{i-1}(p)$ and these are redistributed so that there are two resulting sequences on two processor subsets each of size $p/2$.

The process of separating $G_i(p)$ requires a gather to the leader processor; each processor has roughly N_i/p units to send to the leader processor. The communication cost is at most $t_s\ log_2 P + t_w N_i/p$. The projection to level zero requires at worst an all to all personalized communication of cost $t_s(p-1) + t_w N_0/p$. We assume that the cost of bisecting the small graph induced by the separator is no more than the cost of the earlier gather step. The overall cost of separate denoted by $cost_{sep}$ is then approximately $c_2 N_0/p + t_s(p-1)$.

The cost of marking is mainly that of communicating the partition information through the sequence of graphs. At each graph, it can cost at worst an all to all personalized communication. For a graph $G_l(p), 0 \leq l \leq i$, the cost is $c_3(N_l/p) + 2t_w(N_l/p) + t_s(p-1)\log_2 p$. The sizes of the graph are halved starting at N_0 and if we assume that each graph is distributed in a balanced manner over the p processors, the total cost ($cost_{mark}$) is no more than $2c_3\frac{N_0}{p} + 2t_w\frac{N_0}{p} + t_s(p-1)\log_2 p$.

The redistribute step requires at most exchanging half the data at a processor with its partner; hence the cost is at most $c_4 N_l/p$ for $G_l(p), 0 \leq l \leq i$. Again given the sizes of the graphs in the sequence are repeatedly halved, $cost_{redist}$ is $2c_4 N_0/p$. Thus the total cost of $\texttt{split}(i,p)$ is :

$$cost(\texttt{split}, i, p) \quad = \quad c_4 \frac{N_0}{p} + t_s(p-1)\ log_2 P$$

Now the total cost over all nested calls to `split` is:

$$cost(\texttt{split}) \quad = \quad \sum_{i=0}^{\log_2 P-1} cost(\texttt{split}, i+1, 2^{i+1})$$

At $\texttt{split}(\log_2 P, P)$, the size N_0 is actually N. At the next step, there are two parts in the original graph each roughly half the size. So at $\texttt{split}(\log_2 P-1, P/2)$, the size $\hat{N}_0$ is $N/2$. However, the number of processors is also being halved. So at $\texttt{split}(\log_2 P, P)$, the size per processor is N/P, and at $\texttt{split}(\log_2 P-1, P/2)$, the size per processor is again N/P. The recursion is applied at most $\log_2 P$ times and so the sum is bounded by:

$$cost(\texttt{split}, i, p) \quad = \quad \sum_{0}^{i=\log_2 P-1} c_3\frac{N}{P} + t_s 2^{i+1}(i+1)$$

This in turn reduces to:

$$cost(\texttt{split}) \quad = c_4\tfrac{N}{P}\log_2 P + c_5 P\log_2 P$$

Combining both $cost(\texttt{shrink})$ and $cost(\texttt{split})$ the total cost of SSND is:

$$cost(\texttt{ssnd}) \quad = \quad c_6\frac{N}{P}\log_2 P + c_7 P\log_2 P \qquad (1)$$

The overall complexity of PCO depends on the local ordering scheme. The serial complexity of nested dissection orderings is $O(n\log_2 n)$ for a graph with n vertices [18]. For such a local ordering scheme, each processor has a subgraph with roughly N/P vertices; hence the cost is $O(\frac{N}{P}\log_2\frac{N}{P})$. The total parallel cost is therefore $O(\frac{N}{P}\log_2 N + P\ log_2 P)$. Given our assumption that $P\ log_2 P$ is smaller than N/P we have a scalable parallel ordering algorithm for general sparse matrices.

4. Empirical Results. We now present empirical results on the performance of PCO. We first examine the effectiveness of PCO in preserving sparsity by comparing the fill incurred by PCO and other well established schemes. Next, we discuss the parallel performance of PCO on three different distributed memory machines.

The amount of fill incurred is a good measure of the effectiveness of any ordering scheme. Our test suite has a total of twenty sparse problems ranging in size from four and a half thousand to about forty thousand unknowns. We use four different ordering strategies to compare the fill incurred. The test problems are described in the first three columns of Table 1.

We compare the fill resulting from PCO with that for three sequential methods and two forms of the parallel Cartesian Nested Dissection (CND) [14] method. The three serial methods we use are: Multiple Minimum Degree (MMD) [9], Automatic Nested Dissection (AND) [8], and Spectral Nested Dissection (SND) [20]. For both MMD and AND we use the Sparspak [5] implementation; we use the SND code distributed by Pothen and Wang. In practice, the serial Multiple Minimum Degree (MMD) method is known to produce the least fill of any general purpose ordering heuristic for the vast majority of sparse problems and so comparing against it is a good test for PCO. AND is generally not as effective as MMD in limiting fill, but provides a widely recognized point of comparison to ensure that PCO is at least in the right range with respect to preserving sparsity. AND is one of the fasted serial

TABLE 1

Description of test problems and comparison of fill-in; All values are in thousands.

Label	N	$(1/2)\|A\|$	$\|L\|$					
			MMD	SND	AND	CND	CND-MMD	PCO
hammond	4.7	18.4	91	101	**146**	116	125	130
barth4	6.0	23.4	116	135	**200**	152	154	156
venkat	10.1	39.8	244	256	**340**	304	298	303
kall2	6.8	53.6	576	1498	**1629**	964	980	990
sphere6	16.3	65.5	**905**	685	681	686	720	660
barth4dual	11.4	28.3	125	158	212	**215**	185	179
dimitri3	11.1	43.9	272	289	**362**	347	352	**383**
shuttle	10.4	57.0	320	381	387	385	373	**447**
ghs1	14.9	58.2	214	322	**595**	383	323	392
barth5	15.2	61.4	372	400	**657**	448	600	560
gph1	15.6	62.2	463	498	531	**540**	542	500
gl1	17.3	68.8	435	488	646	**728**	688	602
gsq1	17.4	68.7	312	384	**483**	476	492	411
barth5dual	30.2	75.1	421	494	**648**	599	514	581
kall3	10.5	86.6	1659	2268	**3520**	1060	1260	1691
vaughan	29.6	111.4	1447	1935	1481	**4049**	1979	1380
gsq2	30.73	121.8	673	781	**942**	915	873	900
gl2	34.0	135.5	1052	1081	1478	**1558**	1380	1380
gph2	35.1	139.7	1205	1248	1359	**1441**	1459	1301
ghs2	39.3	152.8	615	1056	**2882**	867	856	1256

ordering schemes. SND compares well with MMD with respect to fill but has very large execution time. We also present fill-in results for parallel CND [14] in two forms: one in which the CND heuristic is applied throughout in both the distributed and local phase, and another (CND-MMD) in which CND is applied in the distributed phase with 32 processors followed by MMD in the local phase. For the results in Table 1, we used PCO with 32 processors. The fill increases by at most ten percent on increasing the number of processors to 128.

In Table 1, for a given problem, the value in bold font is that of the largest fill. As expected, MMD produces the least fill for almost all the problems. The fill incurred by SND is close to that of MMD while that of AND is the largest for several problems. On the other hand among the three serial ordering schemes, AND is by far the fastest, MMD takes more time than AND while SND takes considerably longer. Parallel CND is suitable for graphs associated with geometric information and when executed on one processor takes approximately twice the time required by AND. Hence its serial running time is low and it has been shown to parallelize well on distributed memory machines. However, it does tend to incur larger fill comparable to the AND heuristic. When used with MMD for the local phase (32 processors), the fill is reduced in most cases. Our PCO scheme incurs about the same fill in as the CND-MMD ordering and in most cases it incurs less fill than AND.

The behavior of PCO is not surprising. In broad terms, PCO is a combination of AND and MMD suitable for parallel machines and so the sparsity preservation aspect of PCO mimics a hybrid of both AND and MMD strategies. What is interesting is that the combination of graph contraction (`shrink`) followed by the simple `split` is effective; in our experiments we observed that the separators computed were in fact smaller than those computed by AND. We believe this is related to effect of graph contraction. Both Bui and Jones [4] as well as Hendrickson and Leland [15] observe that the separator size is considerably reduced when computed by successive refinement over a sequence of contracted graphs. In PCO, we do not use the contracted graphs for refinement but to simply enable computing a separator in parallel; we project the separator in the smallest shrunk graph to the original and then refine it. It appears that this two step process leads to a slight improvement in separator size over AND and in any case is no worse than applying AND on the original graph while allowing effective parallel implementation.

The design of a fully parallel sparse solver requires an effective parallel ordering scheme. Ideally, the fill incurred should compare well with the best serial scheme but in practice the time to compute the ordering is also an important consideration. For most sparse problems even though the asymptotic complexity of factorization is of higher order compared to the ordering, the actual factorization times tend to be very low for codes designed using BLAS. On the other hand, graph-based ordering methods with indirection and complicated data structures lead to a lower utilization of peak machine performance and it is not surprising to find that the ordering times are higher than that of factorization for small to medium problem sizes. Parallel ordering is even harder because there is no data locality to begin and parallel nested solves both the ordering and partitioning problem and involves significant data redistribution. There is certainly a trade-off between time to order versus the fill incurred. Our aim was to design a fast, reasonably simple, parallel nested scheme suitable for general problems, i.e., those for which geometry may not be available. PCO serves this purpose; the fill incurred is in the range marked by MMD and AND and the time to compute the ordering is typically $1.5 - 2$ times that of CND-MMD.

Parallel Performance. As derived in the earlier section, the parallel complexity of SSND is $O(N/P \log_2 P)$ and the memory requirement is $O(N/P)$. The complexity PCO is the same as that of parallel CND. We compare the actual execution times and speed-ups of PCO with that of CND for three different message passing multiprocessors, the Thinking Machines CM5, the Intel iPSC/860, and the Intel Touchstone Delta.

Despite the analytic measures of scalability, the actual speedups obtained on available message passing multiprocessors is far from ideal. There are several factors that account for this. First of all, most machines have a high communication to computation ratio, i.e., the time required to communicate one word is large multiple of that required for one unit of computing. Next, the interconnection network affects the performance and most collective communication operations such as broadcast, gather etc. have higher cost on the mesh compared to the hypercube. Thirdly, algorithms for hypercubes can be better adapted to meshes by careful mapping of processors in a hypercube to those in the mesh but no automated tools exist for this purpose. Recently, after we developed our code several message passing libraries have been developed [6, 3]; implementations in terms of these library routines should ease the problem of performance tuning. Finally, cache-effects and message buffering protocols also play a role in observed speed-ups. For example, with an effective short message protocol, sending several short messages may take less time than a single long message even though the total volume of communication may be the same. Both parallel CND and PCO are implemented in C with message passing extensions. In both codes the "hypercubic" versions of most collective communications are run as is on the mesh; no attempt has been made to recode for better performance on the mesh.

The three machines we use (TMC-CM5, Intel iPSC/860, Intel Delta) support the same basic MIMD computational paradigm with explicit message passing but differ in many significant ways. The CM5 has Sparc processors with 32 Mbytes of local memory in a fat-tree network. The fat-tree network can simulate a hypercubic network without any significant penalty [17] and with the low execution rate (5 Mflops) of the Sparc, the CM5 has the most favorable communication to computation ratio of the three machines. The Intel iPSC/860 has iPSC/860 processors with 8 Mbytes of local memory, connected in a hypercube. The Intel Touchstone Delta has the same iPSC/860 processors but with 16 Mbytes of local memory and a mesh interconnection network. Besides the interconnection network, the message protocols are also different on the two Intel multiprocessors.

We compare the parallel performance of PCO against that of CND. For the latter, we again use two versions, one with CND all through and the other (CND-MMD) with CND for a distributed phase and MMD for the local. The version of CND and CND-MMD we use for fair comparison is not the code available in public domain as part of a fully parallel solver [13]. The redistribution step after distributed CND has been modified to redistribute only the graph information while the version in the solver redistributes the numeric values as well as the right hand side vector for later numeric factorization. Recall that MMD is applied in a local phase for both PCO and CND-MMD. The local phase problem size is quite small for the test problems and MMD is a faster method than both CND and SSND for these local phase problems. This is not the case with larger problems, and MMD is slower than CND; for smaller problems when CND is used for the local phase, a sort step required to set up data structures is the dominant cost. We use four representative problems from our earlier test suite for studying the effect on execution times while successively doubling the number of processors from 16 to 128. Two of the problems "gl2" and "gsq2" are highly graded 2-D finite-elements; "gl2" is an L-shaped region while "gsq2" is a square region. Both problems are fairly small with about thirty thousand unknowns and about four times as many nonzeroes. The third problem is "barth5dual," a 3-D mesh while the fourth "g500" is a 500×500 model grid problem. For most problems with the same number of processors, the execution time of PCO is at about 1.5 to 2 times that of CND-MMD. This factor plays a significant role on execution times for the same problem but with increasing number of processors. On doubling the number of processors, the local phase size is roughly halved by applying another step of the relatively expensive distributed ordering.

In Table 2, we provide parallel execution times for PCO, CND, and CND-MMD on each of the three machines. The values in the column labeled 16 contain execution times in seconds. In the remaining columns, the execution time is expressed as a fraction of that for 16 processors for a given problem. For example, for PCO and the problem "gl2" on the Intel Delta, the columns labeled 16 and 32 contain 10.7 and 0.75 to indicate that the execution time on 32 processors is .75 times that on 16 processors. For all methods we observe that:

Table 2

Execution Times on 16-128 processors of the Intel iPSC/860, TMC-CM5, and Intel Delta. Values for processors 32-128 are presented as a fraction of the time for 16 processors.

Problem Problem	Machine	Algorithm	16 (seconds)	32	64	12 8
gl2						
	TMC - CM5	PCO	31.5	.70	.44	.29
		CND-MMD	21.4	.72	.45	.29
		CND	29.5	.68	.40	.25
	iPSC/860	PCO	11.9	.70	.53	.38
		CND-MMD	8.7	.73	.52	.32
		CND	11.5	.65	.45	.27
	Intel/Delta	PCO	10.7	.75	.60	.50
		CND-MMD	7.9	.69	.45	.31
		CND	9.9	.67	.41	.27
gsq2						
	TMC - CM5	PCO	31.4	.62	.41	.28
		CND-MMD	20.8	.68	.42	.29
		CND	27.6	.65	.40	.24
	iPSC/860	PCO	11.9	.69	.53	.35
		CND-MMD	8.2	.73	.53	.36
		CND	11.5	.61	.39	.26
	Intel/Delta	PCO	10.9	.66	.55	.51
		CND-MMD	7.5	.69	.47	.38
		CND	10.2	.66	.42	.27
barth5dual						
	TMC - CM5	PCO	31.6	.68	.46	.36
		CND-MMD	20.0	.70	.46	.32
		CND	21.1	.68	.42	.28
	iPSC/860	PCO	12.9	.76	.54	.40
		CND-MMD	8.1	.76	.50	.35
		CND	8.5	.72	.46	.32
	Intel/Delta	PCO	11.7	.82	.56	.45
		CND-MMD	7.5	.87	.55	.38
		CND	7.9	.77	.47	.35
g500						
	TMC - CM5	PCO	52.4	.62	.43	.30
		CND-MMD	24.0	.65	.46	.32
		CND	28.6	.60	.34	.26
	iPSC/860	PCO	20.2	.65	.45	.34
		CND-MMD	10.4	.65	.50	.35
		CND	10.8	.60	.34	.32
	Intel/Delta	PCO	17.7	.62	.56	.34
		CND-MMD	9.2	.65	.45	.36
		CND	9.3	.65	.40	.34

- the best parallel performance is on the CM5 which has the most balanced communication to computation ratio as well as the fat-tree (hypercubic) network.
- performance on the Intel iPSC/860 hypercube is worse than that on the CM5 because of the higher communication to computation ratio.
- the relative decrease in execution time is the least on the Intel Delta because of the the mesh connectivity as well as the large communication to computation ratio. The effect of the mesh interconnection is more visible for PCO because it is more communication intensive.

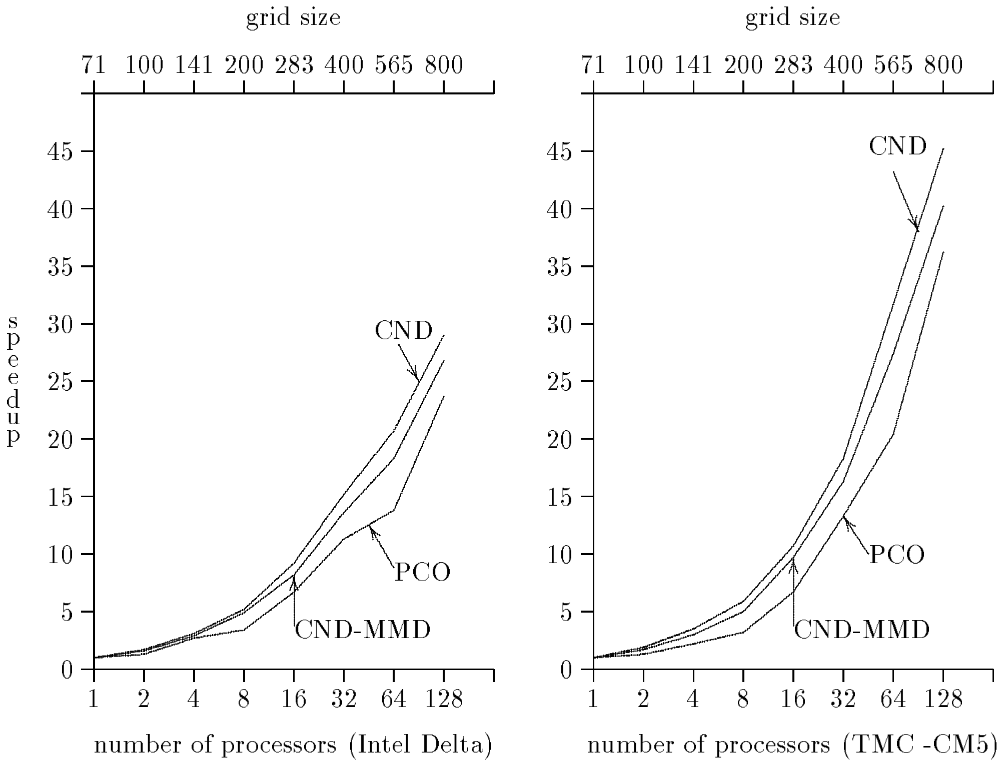

FIG. 3. *Scaled Speedup for a series of grid problems on the Intel Delta and TMC-CM5*

With respect to Table 2, if the speedup is linear in the number of processors (relative to the execution time on 16 processors), the values in columns labeled 32, 64 and 128 should be .5, .25 and .125. For PCO with 128 processors of the CM5, the time on 128 processors is about twice the ideal value for "gl2" and "gsq2," i.e., on 128 processors the execution time is about .28 times that on 16 processors. This is worse than the value of .24 observed for the same problems on the CM5 for CND but very close to the values of $.29 - .30$ observed for CND-MMD. For "barth5dual," the speedup of PCO is a little less than that of CND-MMD. This is in part because the **`shrink`** procedure takes slightly longer with a larger number of processors; we found that the number of intermediate steps in contracting from one level to the next remains the same for 16 through 64 processors but tends to increase by a total of $3-4$ for 128 processors. For "g500," the speedup of PCO is quite similar to that of CND but in absolute terms, the total execution time is larger than that of CND.

We next examine the speed-up of PCO when the problem size per processor is kept fixed and the number of processors is increased. We use a sequence of model grid problems for this purpose starting with 71×71 grid for a single processor and ending with the 800×800 grid for 128 processors. Once again, if T_i denotes the execution time using i processors, the scaled speedup is computed as $i \times \frac{T_1}{T_i}$. The scaled speedup is plotted for the Intel Delta and the CM5 in Figure 3. The plot shows the effect of the interconnection network; the mesh leads to lower speed-ups for all methods. It is conceivable that some of this effect will not be so significant if all the collective communication routines are optimized for the mesh. The speedup achieved by PCO is comparable to that of CND-MMD on both multiprocessors. The plot for PCO is not as smooth because the speedup drops for 8 and 64 processors. In these two instances there is an increase in the number of intermediate steps in contracting one graph to the next.

Recently, Karypis and Kumar [16] have implemented a parallel multilevel nested dissection (MLND) on the Cray T3D using the shared memory library. The differences between the their algorithm and PCO algorithm have been described earlier in Section 2. We now comment on differences in performance and complexity. Kumar and Karypis show that their method produces low fill-in (comparable to that of Multiple Minimum Degree) for some test problems; they attribute this to their "graph-growing" separator heuristic. Our PCO certainly has higher fill-in on average than MMD. Given the difference in computing paradigms (shared memory vs distributed memory) and the parallel multiprocessors, a direct comparison in performance is not viable. However, on the basis of execution times presented [16] it appears that the relative speed-up on going from 16 to 128 processors is higher for PCO compared to MLND. For MLND, the execution time on 128 processors (Table 4 in [16]) is in the range $0.5 - 0.6$ times that on 16 processors. On the TMC-CM5, the execution time for PCO on 128 processors is in the range $0.28 - 0.30$ times that on 16 processors. Furthermore, SSND has a lower parallel complexity of $O(\frac{N}{P} \log P)$ as well as a lower memory requirement of $O(N/P)$. MLND has parallel complexity of $O(\frac{N}{\sqrt{P}} \log P)$ and memory requirements are $O(\frac{N}{\sqrt{P}})$.

5. Conclusions. We have developed SSND, a new parallel nested dissection scheme of complexity $O(\frac{N}{P} \log P)$ for dissecting a graph with N vertices into P subgraphs. The overall ordering scheme called PCO uses Multiple Minimum Degree for ordering the P subgraphs independently in a local phase. If instead of Multiple Minimum Degree, a graph based nested dissection dissection method [18] is used, the overall complexity of PCO is $O(\frac{N}{P} \log N)$. The ordering shows good speed-ups on the Thinking Machines CM5, Intel iPSC/860, and the Intel Touchstone Delta. A scaled speedup of 35 is obtained for the model grid problem with 128 processors of the CM5. The fill-in obtained is within twice that of Multiple Minimum Degree and less than that of Automatic Nested Dissection for the problems in our test-suite.

In summary, PCO seems highly promising. We believe that the fill-in incurred by PCO can be substantially decreased by using a single parallel refinement step after SSND. In our initial experiments we have found that such a "post-refinement" step can reduce the separator sizes by as much as twenty to thirty five percent. We are in the process of effectively parallelizing this step. We believe that especially for message passing multiprocessors with high communication latency, a single "post-refinement" step is more viable than successive refinements as in the serial multilevel approach [4, 15]. Such a "post-refinement" step would be applicable after any P

way dissection scheme. A "post-refinement" also allows the movement of elements between more than two regions. Our initial experiments indicate that this helps to smooth out some of the larger separators and the imbalance in sizes of subgraphs resulting from successive bisection. We also believe that parallel performance on the mesh architectures would be substantially improved with an implementation using optimized collective communication schemes.

6. Acknowledgements. The author would like to thank T.N. Bui and B. Hendrickson for several useful comments. This research used the CM5 at the National Center for Supercomputing, the Intel IPSC/860 at the Oak Ridge National Laboratory, and the Intel Touchstone Delta at the Caltech Concurrent Supercomputing Consortium.

REFERENCES

[1] S. T. Barnard and H. Simon, *A fast multilevel implementation of recursive spectral bisection for partitioning unstructured meshes*, in Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing, R. F. Sincovec, D. E. Keyes, M. R. Leuze, L. R. Petzold, and D. A. Reed, eds., Philadephia, PA, 1993, SIAM Publications, pp. 711–718.

[2] ———, *A parallel implementation of multilevel recursive spectral bisection for partitioning unstructured meshes*, in Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing, D. H. Bailey, P. E. Bjorstad, J. R. Gilbert, M. V. Mascagni, R. S. Schreiber, H. D. Simon, V. J. Torczon, and L. T. Watson, eds., Philadephia, PA, 1995, SIAM Publications, pp. 627–632.

[3] M. Barnett, S. Gupta, D. Payne, L. Shuler, R. van de Geijn, , and J. Watts, *Interprocessor collective communication library (InterCom)*, in Scalable High Performance Computing Conference, Los Alamitos, CA, 1994, IEEE Computer Society Press.

[4] T. N. 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, R. F. Sincovec, D. E. Keyes, M. R. Leuze, L. R. Petzold, and D. A. Reed, eds., Philadephia, PA, 1993, SIAM Publications, pp. 445–456.

[5] E. Chu, A. George, J. Liu, and E. Ng, *Sparspak: Waterloo sparse matrix package user's guide for Sparspak-A*, Tech. Rep. CS-84-36, Department of Computer Science, Univ. of Waterloo, Waterloo, Ontario, Canada, 1984.

[6] M. P. I. Forum, *MPI: A message -passing interface standard*, Tech. Rep. CS-94-230, Computer Science Dept., Univ. of Tennessee, Knoxville, April 1994.

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

[8] 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.

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

[10] J. R. Gilbert, G. L. Miller, and S. H. Teng, *Geometric mesh partitioning: Implementation and experiments*, Tech. Rep. in preparation, Xerox Palo Alto Research Center, 1994.

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

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

[13] M. T. Heath and P. Raghavan, *Performance of a fully parallel sparse solver*, in Scalable High Performance Computing Conference, Los Alamitos, CA, 1994, IEEE Computer Society Press. submitted to International Journal of Supercomputer Applications.

[14] ———, *A Cartesian nested dissection algorithm*, SIAM J. Matrix Anal. Appl., 16 (1995), pp. 235–253.

[15] B. Hendrickson and R. Leland, *A multilevel algorithm for partitioning graphs*, Tech. Rep. SAND93-1301, Sandia National Laboratories, Albuquerque, NM 87185, 1993.

[16] G. Karypis and V. Kumar, *Parallel multilevel graph partitioning*, Tech. Rep. 95-036, Department of Computer Science, University of Minnesota, Minneapolis, MN, May 1995.

[17] C. E. Leiserson, *The network architecture of the connection machine CM-5*, in Fourth Annual ACP Symposium on Parallel Algorithms and Architectures, ACM Publications, 1992, pp. 272–285.

[18] R. J. Lipton, D. J. Rose, and R. E. Tarjan, *Generalized nested dissection*, SIAM J. Numer. Anal., 16 (1979), pp. 346–358.

[19] 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.

[20] 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.

[21] E. Rothberg, *A parallel implementation of the multiple minimum degree heuristic*. Presentation, Fifth Siam Conference on Applied Linear Algebra.

[22] E. Zmijewski, *Sparse Cholesky Factorization on a Multiprocessor*, PhD thesis, Department of Computer Science, Cornell University, August 1987.