



ELSEVIER

Parallel Computing 22 (1996) 1291–1333

PARALLEL
COMPUTING

Solving large nonsymmetric sparse linear systems using MCSPARSE

K.A. Gallivan^{a,*}, B.A. Marsolf^{a,1}, H.A.G. Wijshoff^{b,2}

^a Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign,
Urbana, IL 61801, USA

^b High Performance Computing Division, Department of Computer Science, Leiden University, Leiden, The
Netherlands

Received 2 June 1994; revised 12 October 1994, 8 May 1995, 28 May 1996

Abstract

In this paper, the methods and implementation techniques used for the nonsymmetric sparse linear system solver, MCSPARSE on the Cedar system are described. A novel reordering scheme (H^*) upon which the solver is based is presented. The tradeoffs discussed include stability and fill-in control, hierarchical parallelism, and load balancing. Experimental results demonstrating the effectiveness of the solver with respect to each of these issues are presented. We also address the implications of this work for other parallel processing systems.

Keywords: Linear algebra; Sparse linear systems; Cedar system; Implementation; Parallel processing systems

1. Introduction

Several techniques have been proposed to solve large nonsymmetric sparse systems of linear equations on parallel processors. In this paper, we present a new method for solving such linear systems using novel reordering and pivoting schemes; we also discuss the issues involved in mapping this method onto a parallel processor. The

* Corresponding author.

¹ Supported by the National Science Foundation under Grant No. US NSF CCR-9120105 and by ARPA under a subcontract from the University of Minnesota of Grant No. ARPA/NIST 60NANB2D1272.

² Supported by Esprit DGXIII Grants ASPCA and APPARC (No. 6634).

discussion focuses on the tradeoffs involved with an implementation on the Cedar system [42], a multicluster architecture (four clusters, each with eight processors), but will also include comments on the applicability of the techniques to other systems.

A key task which determines the effectiveness of these techniques is the identification and exploitation of the computational granularity appropriate for the target multiprocessor architecture while maintaining the stability and sparsity of the factorization. For example, the architecture of the Cedar system requires the exploitation of multiple levels of parallel task granularity due to the cluster-based organization. In addition, careful consideration of data layout is necessary to effectively use the hybrid memory system comprising private cluster memories and a shared global memory.

Many approaches for solving nonsymmetric sparse systems in parallel have been investigated. One of these, the multifrontal scheme [4,7,12,13,39], has been used for both symmetric and nonsymmetric systems. A multifrontal scheme constructs a directed acyclic graph (called an *assembly DAG*) to organize the parallel work. A node in the assembly DAG represents a certain computation, which may include handling the information from the node's successors and performing some pivot eliminations. All nodes without successors in the assembly DAG may be computed in parallel, while internal nodes can be computed only after their successors have completed. A pool of the available work (i.e., the nodes in the assembly DAG that can be computed) is maintained in shared memory. When any process needs work, it retrieves a node from the pool. After all the successors of a node have finished, the node is then placed in the pool of available work. This approach, if correctly organized, can provide large and medium grain parallelism. However, the method tends to work well when the pivot sequence is constrained to the frontal matrix – which can cause stability concerns. In order to improve stability, a lost pivot recovery strategy has been added to multi-frontal schemes [31].

Another approach to parallel sparse solvers exploits the dynamic identification and application of parallel pivots [1,9,19,45]. At each stage, a set of pivots that can be applied in parallel is constructed and the appropriate updates performed. These codes typically concentrate on medium and fine grain parallelism and tend to be most efficient on a moderate number of processors with fairly tight coupling. There is also previous work on performance improvements of direct sparse solvers on vector supercomputers [4]. The results indicate that vectorization can sometimes be used to improve the performance. Both of these approaches can be used as part of an algorithm which exploits multiple levels of parallelism.

Tearing techniques have been proposed to expose large grain structure and parallelism by reordering the matrix into a bordered block triangular matrix [15,21,33]. This effectively partitions the problem into small subproblems (the diagonal blocks) and then eliminates all connections between the subproblems (the border blocks). Unfortunately, the associated factorization routines are often unable to preserve stability and sparsity without destroying this structure.

The approach taken in this paper uses the H^* ordering to identify a priori large and medium grain parallelism and to restructure the matrix into bordered block upper triangular form. This is used in combination with a factorization routine, utilizing the pivoting technique *casting*, which preserves this structure while maintaining stability

and sparsity at acceptable levels. We describe the implementation and performance tradeoffs for the multicluster solver MCSPARSE on the Cedar system. Specifically, we address the issues of stability and fill-in control, hierarchical parallelism, and load balancing.

The paper is organized as follows. We first review other reordering methods and then present the H^* ordering, followed by an overview of the MCSPARSE solver. Each of the issues listed above is then considered in turn with their performance tradeoffs on Cedar. The implications which these issues have on other parallel processors will be discussed. Finally, conclusions and the applicability of the results to further work on solvers such as MCSPARSE are presented.

2. Comparison of different approaches

During the introduction, the H^* ordering for transforming a matrix into bordered block upper triangular form was described as novel. This is not to say that the use of the bordered triangular form, or the bordered block triangular form, for solving sparse nonsymmetric systems is a new idea.

Research on orderings for transforming matrices into the bordered triangular form has been done using graph theory methods to find the minimal essential set [6,41]. These methods rely on the fact that the sparse system is positive definite, so that pivots can be chosen from the elements of the diagonal without losing stability. In the case of nonsymmetric systems, which are not necessarily positive definite, these methods are not always successful.

For nonsymmetric systems, the bordered *block* triangular form is preferable when pivot selection is restricted to within the diagonal blocks since the overall structure of the system is not destroyed. Several different methods for finding the bordered block triangular form have been proposed. Partitioning and tearing methods [43] can be used, and algorithms such as P^4 [33], P^5 [14], the Hierarchical Partition by Lin and Mah [37], and the level set algorithm by Arioli and Duff [2] were introduced for ordering the matrix into the desired form. Although these methods are rather successful at transforming the system into the bordered block triangular form, the associated factorization phases lack stability when the pivot search is constrained to the diagonal block; therefore, these methods are not recommended for use on general nonsymmetric systems.

In the remainder of this section, we briefly describe the major steps of the algorithm and relate them to previous work. Within MCSPARSE, the necessary provisions are made to guarantee a suitable level of stability within the factorization phase. First, the initial phase of H^* is used to transfer relatively large elements of the matrix to the diagonal. This transformation is based on the transversal algorithm, which is also used in the level set algorithm of Arioli and Duff. The main difference, however, is that in the level set algorithm the transversal is not constrained to contain relatively large elements.

After this initial phase, H^* proceeds by reordering the system into the desired form while preserving the initial diagonal structure via the use of symmetric permutations.

This is in contrast to the methods on which P^4 , P^5 , and the Hierarchical Partition rely. Symmetric orderings are, of course, not as flexible as nonsymmetric orderings, and the dimensions of the diagonal and border blocks in the resulting permuted matrix might not be as small. This can be observed in the results of the level set algorithm in Table 2. H^* mitigates this difficulty by using different basic algorithms (i.e., Tarjan's algorithm and nested dissection) in successive ordering phases designed to complement each other. As shown below, the complementary nature of the phases results in a significant increase in the power of the symmetric permutations.

In the factorization phase, precautions have to be taken to guarantee a reasonably stable solution method. P^5 , the Hierarchical Method, and the level set algorithm guarantee structurally nonsingular blocks. However, these methods are still potentially unstable. Iterative refinement can be used to improve the instability (see [2]). In our method, stability is guaranteed by allowing pivots to be taken within the diagonal blocks as well as the border. This was also attempted with the P^4 ordering [3]; however, the overhead incurred prevented this approach from being competitive with other direct solvers. Within MCSPARSE, border pivoting relies upon a symmetric permutation, referred to below as casting, which minimizes the associated overhead. Also, because the initial phase of the ordering moves large elements to the diagonal, the amount of casting can be significantly reduced (casting still may be necessary because the magnitudes of the elements may diminish as the factorization proceeds). This approach enables MCSPARSE to be competitive with other direct solvers, see Section 8.

Descriptions of the algorithms used within H^* are presented in the next section. A preliminary algorithmic description of the H^* ordering is in [47].

3. The H^* ordering

3.1. Background

The interpretation of the actions of H^* depends upon the notion of a graph associated with a sparse matrix.

Definition 3.1. Given a nonsymmetric $(N \times N)$ sparse matrix A , the digraph associated with A is defined to be the graph $G(V, E)$ with $|V| = N$ such that $(i, j) \in E$ if and only if $a_{i,j}$ is a non-zero entry in A .

The hybrid ordering H^* is composed of two different types of orderings: nonsymmetric and symmetric. Nonsymmetric orderings are obtained by independent row and column interchanges of the matrix, and can change certain properties of the sparse matrix that are preserved by symmetric orderings (e.g., eigenvalues and diagonal dominance). A nonsymmetric ordering, therefore, can be used to enhance the numerical properties of the factorization of the matrix if the values in the matrix are considered when determining the row and column orderings. In H^* , an initial nonsymmetric ordering is used to enhance the numerical properties of the factorization, and subsequent symmetric orderings are used to obtain a bordered block triangular matrix.

In order to obtain the desired structure, H^* exploits the concepts of a node separator set and a quasi-separator, a generalization applicable to directed graphs, which are defined as follows.

Definition 3.2. Given a graph $G = (V, E)$, a *node separator* set S of G is a subset of V such that there exists sets B and C with

- (a) B , C , and S disjoint,
- (b) $B \cup S \cup C = V$, and
- (c) there exist no edges $(x, y) \in E$ with
 - (1) $y \in B$ and $x \in C$, and
 - (2) $x \in B$ and $y \in C$.

If (c.1) is fulfilled but (c.2) is not, the set S is a *quasi-separator*.

There are four phases in the hybrid ordering H^* . The first phase, H_0 , is a nonsymmetric ordering which permutes onto the diagonal the largest elements available at each decision point of the production of the transversal. This guarantees that all elements on the diagonal are nonzero. The second phase consists of applying Tarjan's algorithm to transform the matrix into triangular block form. The third phase, H_1 , is applied to each diagonal block produced by Tarjan's algorithm that is considered too large. H_1 attempts to change each of these blocks into bordered block triangular form via a modified Tarjan's algorithm. The last phase, H_2 , is also applied to only the large diagonal blocks remaining in the matrix to change them into bordered block triangular form via a modified dissection algorithm. After H_1 and H_2 have been applied, the borders that were generated for each of the large diagonal blocks are combined to form a single border, thereby converting the entire matrix into the bordered block upper triangular form. Tarjan's algorithm, H_1 , and H_2 are all symmetric orderings.

It is important to note that all ordering phases, with the exception of Tarjan's, are heuristics in the sense that they cannot be proven optimal. Since each of these heuristics is based on complementary techniques, we need four phases in H^* . Also, as a direct consequence, these heuristics are governed by certain threshold parameters: α in H_0 ; T_{done} , T_{long} , T_{minb} , T_{maxb} , and T_{maxsep} in H_1 ; and β and T_{remain} in H_2 . The number of threshold parameters might seem somewhat excessive (for instance for the H_1 phase) but, as can be seen below, they are all functional and, as such, are necessary. Although each of these parameters can be individually controlled by the user, in our implementation we used a fixed, problem-independent set of values (see Section 3.6).

3.2. H_0

H_0 is a transversal algorithm for permuting nonzero entries onto the diagonal using a nonsymmetric ordering. The transversal algorithm has been modified to permute large elements to the diagonal in an attempt to enhance the stability of the subsequent factorization.

The transversal ordering is a matching between the columns and the diagonal positions of the matrix, which can be found using many different algorithms. Algorithms

for finding set representation [32] or solutions to the assignment problem [35] could be used. An alternative algorithm involves finding maximal matchings in bipartite graphs [34].

H0 is based on work by Duff and Gustavson [10,11,30]. The algorithm uses a depth-first search of the matrix to determine a series of column interchanges. The algorithm creates a transversal by assigning a unique diagonal position to each column of the matrix. These assignments determine a column permutation which places nonzero elements on the diagonal.

At step j , the algorithm has a transversal for columns 1 through $j - 1$ and attempts to extend the transversal to include column j . The algorithm first determines if an *easy* insertion is possible. An *easy* insertion occurs when column j has a nonzero element in row i where diagonal i is not currently assigned to another column. To determine if an *easy* insertion is possible, a sequential search is made of the nonzero elements in column j . If the nonzero element in row i is in a row whose index is not one of the currently assigned diagonal positions, then diagonal i is assigned to column j , the search is stopped, and the algorithm proceeds to column $j + 1$. If an *easy* insertion is not possible, then the algorithm must determine if an insertion can be realized by a suitable permutation of columns 1 through j (backtracking).

The algorithm continues until either an *easy* insertion is made, in which case the algorithm can proceed to the next column, or until it has considered all possible insertions for column j . If at any stage it is not possible to extend the transversal, then the matrix is structurally singular and there is no permutation to make all the diagonal entries nonzero.

This transversal algorithm was modified to enhance the chances of a stable factorization of the matrix with pivots selected from the diagonal blocks. The enhanced version of the algorithm attempts to place *large* elements along the diagonal. This is accomplished by permuting an element a_{ij} to the diagonal only if its value is within a bound, $0 \leq \alpha \leq 1$, of the largest element in the column, i.e.,

$$|a_{ij}| \geq \alpha * \max_k (|a_{kj}|). \quad (1)$$

Only a few changes to the transversal algorithm are required to support this enhancement. An initial step is added to the algorithm to find the maximum absolute value in each column. During the search phase, for both the *easy* insertion and the replacement insertions, an element will be selected only if it satisfies (1). Also, instead of taking the first element that is found by the search, the algorithm searches through all the possible elements and uses the element with the largest absolute value.

The algorithm starts with an initial bound $\alpha = 0.1$, which corresponds to the stability factor typically used in sparse direct solvers, and tries to find a transversal. If a satisfactory bounded transversal cannot be found, then an estimate of the necessary bound is made by examining the columns where the current bound failed. The values in each failed column are examined to determine the maximum value of the bound that would have allowed an insertion to take place for that column. The new bound is then set to the minimum of the bound estimates from all the failed columns and the algorithm is restarted. If a bound less than a preset limit is tried and a transversal is still not found,

then the bound is totally eliminated and the bounded transversal algorithm finds any transversal. However, even with the bound removed, the algorithm still tries the elements with the largest absolute value first. The performance of the H0 algorithm relies upon the ability to quickly find an adequate bound for the transversal.

3.3. Tarjan's algorithm

Tarjan's algorithm [44] finds the strongly connected components of the digraph associated with the matrix in time complexity linear to the number of nodes and edges.³ Renumbering the nodes of the digraph, corresponding to the decomposition of the graph into strongly connected components, yields a symmetric ordering that transforms the matrix into a block upper triangular form.

The strongly connected components are found with a depth-first search of the nodes using a stack to maintain the nodes whose processing has not been completed. The algorithm starts by setting the current node equal to an unprocessed node, placing it on the stack, and marking the node as being processed. In addition, a pointer, *low*, is kept for each node on the stack, indicating the lowest position on the stack reachable from that node. This pointer is initialized to the node's position on the stack.

Each edge, (*current*, *y*), originating from node *current* is considered in turn. If node *y* has already been processed, then it is checked to see if it is still on the stack. If it is, the *low* pointer of node *current* is set to the minimum of the *low* pointers for nodes *current* and *y*. If node *y* is not on the stack, then it has been removed earlier and can be skipped. The algorithm now goes on to the next edge.

If the node *y* has not been processed, then it is added to the stack, initializing its *low* pointer to its position, and saving a pointer to its predecessor, node *current*. The current node is now set to be the new node and a depth-first search of its edges begins.

Once all of the edges from the current node have been processed, the algorithm examines the *low* pointer for the current node to determine if a strongly connected component has been found. If $low_{current}$ equals the node's position on the stack, then a strongly connected component has been found, including the current node and all the nodes above it on the stack, which are then removed from the stack. If $low_{current}$ does not equal the node's position on the stack, then the *low* pointer of its predecessor is set to the minimum of the $low_{current}$ and the *low* pointer of the predecessor. The predecessor is then taken to be the current node and the search of the predecessor's edges is resumed.

Once all of the nodes that can be reached from the root node have been processed, the algorithm starts over with a new node that has not been processed. When all nodes have been processed, the algorithm terminates.

³ A description of this algorithm is included in this paper so that the modifications on which H1 relies can be properly discussed.

3.4. H1

A problem with most sparse matrices is that they do not allow a decomposition into strongly connected components that evenly distributes the nodes and, therefore, Tarjan's algorithm, by itself, will not provide a suitable decomposition. A typical case is a matrix whose associated digraph contains a large cycle. The third phase of H^* , H1, addresses this problem. It is based on Tarjan's algorithm and extracts from the digraph associated with the matrix a small set of nodes such that the remaining graph allows a better decomposition into strongly connected components. During the H1 phase, the size of each potential strongly connected component is monitored during its construction, and, whenever the size grows too large, an attempt is made to delete a small number of nodes from the graph such that the strongly connected component will not grow any further. H1 is applied to each diagonal block that is larger than a threshold, T_{done} , resulting from Tarjan's algorithm. When possible, each diagonal block is separated into two or more smaller blocks and a quasi-separator set. The union of these quasi-separators is placed in the border for the entire matrix.

H1 uses the same depth-first search as Tarjan's algorithm for placing nodes on the stack (as described in the previous section). However, for each node, x , on the stack, two additional pointers are required. The first, denoted $nlow_x$, is a pointer to the position of the lowest node on the stack that is reachable by a single edge from x . The second, denoted $mlow_x$, is a pointer to the position of the lowest node on the stack that is reachable by a single edge from any of the nodes higher than x on the stack. When a new node is placed on the stack, both of these pointers are initialized to the position of the new node.

In Tarjan's algorithm, the value of low_x for a node x indicates a lower bound for the size of the strongly connected component being constructed. Whenever this size is less than T_{done} , H1 proceeds identically to Tarjan's. However, when this threshold is exceeded, the $mlow_{current}$ pointer is used to define an initial quasi-separator set consisting of the nodes on the stack from $mlow_{current}$ to $pos(current) - 1$ (where $pos(j)$ indicates the position of node j on the stack).

Throughout the algorithm, whenever an edge to a node y , ($current, y$), is encountered such that $mlow_{current} - pos(y) \geq T_{long}$ for some threshold value T_{long} , the node $current$ is identified as having a *long* edge which increases the size of the quasi-separator set by an unacceptable level. So, in order to minimize the size of the quasi-separator set, the pointer $mlow_{current}$ is not updated with the position of the node y ; rather, the node $current$ itself is marked as a node to later consider moving into the quasi-separator set. This potentially increases the quasi-separator set by one node as opposed to keeping the current node in the strongly connected component and including all of the nodes from $\min(mlow_{current}, pos(y))$ to $pos(current) - 1$ in the quasi-separator set. The pointer $nlow_x$ is maintained for the current node and the nodes above it on the stack to allow the actual transfer of the marked nodes into the quasi-separator set. Whenever the initial quasi-separator set is constructed, as described above, it is augmented with the nodes which have been marked as having long edges.

In the implementation of H1, the pointers $nlow$ and $mlow$ are updated in a manner similar to that used to update low_x in Tarjan's algorithm. When an edge pointing to a

node y that is lower on the stack than the current node is encountered during the depth-first search, the pointers are updated as follows:

$$low_{current} = \min(low_{current}, low_y),$$

$$nlow_{current} = \min(nlow_{current}, pos(y)),$$

and the pointer $mlow_{current}$ is not updated.

When moving down in the stack to resume the examination of the edges of the predecessor of the current node (denoted below with the subscript $prev$), the updates performed are:

- If $mlow_{current} - nlow_{current} < T_{long}$ then $mlow_{current} = \min(mlow_{current}, nlow_{current})$,
- $mlow_{prev} = \min(mlow_{prev}, mlow_{current})$, and
- $low_{prev} = \min(low_{prev}, low_{current})$.

Note that the decision of whether a node has a long edge is postponed until all of the edges of the node have been examined. This implies that only the longest edge of a node, represented by $nlow$, is used to decide whether the node is moved to the quasi-separator.

After these updates, the decision is made as to whether: no action is required, in the case that a true strongly connected component has been found ($low_{current} = pos(current)$); or, the threshold on the size of the strongly connected component has been exceeded. In the last case, an attempt is made to reduce the size of the strongly connected component. The nodes are divided into three sets: the new block, a border block, and the remaining block. The new block includes the current node and the nodes above it on the stack. The border block contains the nodes starting from $mlow_{current}$ to $pos(current) - 1$. As noted above, the border block is augmented with any nodes marked as having a long edge in the new block. The bordered block is accepted only if:

- The new block is greater than a minimum size, T_{minb} , and smaller than a maximum size, T_{maxb} .
- The size of the augmented quasi-separator set relative to the size of the new block is less than T_{maxsep} .

If the bordered block is accepted, all three blocks are removed with the nodes in the remaining block marked as *still to be considered*. A new starting node is found and the algorithm restarts on the nodes yet to be considered.

If a true strongly connected component has been found, or if the strongly connected component under construction is still less than its allowed size, the same actions are taken as in Tarjan's algorithm.

When all of the nodes that can be reached from the starting node have been processed, the algorithm selects a new root node that has not been processed and continues. When all of the nodes have been processed, the last block will empty the stack and the algorithm is finished. It is important to note that H1 attempts to reduce all diagonal blocks to smaller than the threshold T_{done} ; however, the restriction (T_{maxsep}) on the size of the separator sets allowed might cause the size of some of the diagonal blocks to still be *greater than* T_{done} after H1 has been applied.

An example of how the H1 algorithm finds a quasi-separator set can be found by the

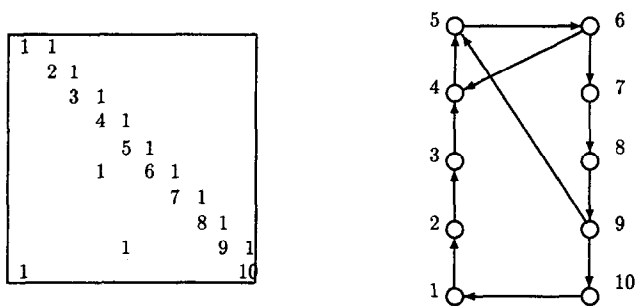


Fig. 1. A 10×10 sparse matrix and its associated digraph.

application of the H1 algorithm to the 10×10 sparse matrix in Fig. 1. The associated directed graph for the 10×10 matrix is also included in this figure.

Fig. 2 is the current state of the algorithm when it has just completed all the edges from node 6. The current block of completed nodes contains nodes 6, 7, 8, 9 and 10. There are three back edges from the nodes in the block; these are the edges $\{10, 1\}$, $\{9, 5\}$, and $\{6, 4\}$. The back edge $\{10, 1\}$, however, was determined to be a *long* edge and is not included in determining the size of the quasi-separator set. Therefore, for node 6, the one edge low pointer for the node points to node 4 ($nlow_6 = 4$); and the one edge low pointer for the nodes above node 6 points to node 5 ($mlow_6 = 5$). This yields an initial bordered block size of 5, a quasi-separator size of 2, and a remaining block size of 3.

Assuming the block sizes meet the necessary constraints, a search for the *long* back edges is made. This search finds the edge $\{10, 1\}$ and places node 10 in the quasi-separator set. The current block size becomes 4, the quasi-separator set becomes 3, and the remaining block stays at 3. The current block contains nodes 6, 7, 8, and 9, and the

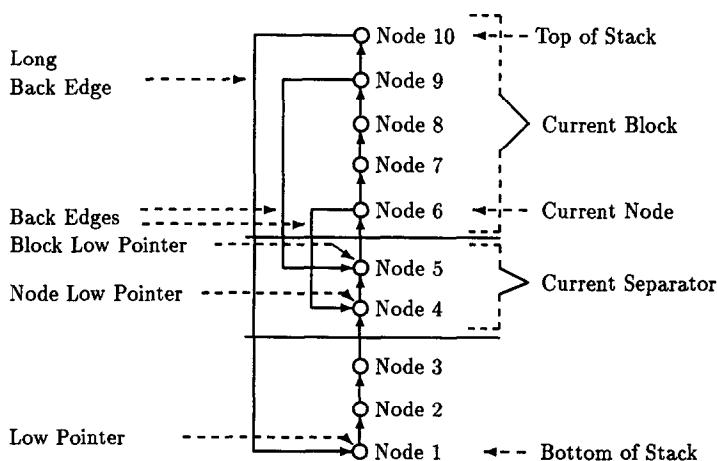


Fig. 2. H1 stack.

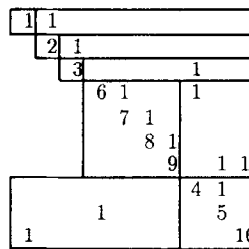


Fig. 3. Reordered matrix.

quasi-separator block contains the nodes 4, 5, and 10. Next, H1 is applied to the remaining nodes, which results in the three independent blocks 1, 2, and 3. The reordered matrix that H1 produces is shown in Fig. 3.

3.5. H2

The H1 algorithm described above approaches the problem of creating quasi-separator sets starting from an algorithm that is clearly intended for structurally nonsymmetric systems (Tarjan's algorithm). It is also possible to approach the problem starting from the standard techniques used to produce separator sets for structurally symmetric matrices (e.g., nested dissection [24,26]).

As in the standard approaches, the ordering H2 starts with the construction of separator sets for the adjacency matrix of $A + A^T$. In our implementation of H2, we used a straight forward implementation of automatic nested dissection [27]. However, other initial orderings could have been used, such as one-way dissection [25], more sophisticated implementations of automatic nested dissection [38], or the graph bisection heuristics proposed in [36].

H2 is applied only to the diagonal blocks that H1 failed to reduce to a size less than a user-specified threshold, T_{done} . The algorithm starts with the graph $(G = (V, E))$ that is associated with the adjacency matrix of the diagonal block under consideration, $M = (A + A^T)$, with the self-edges generated by the diagonal elements removed and where E_A represents the directed edges from A present in G . Before starting the dissection, the nodes are examined to determine if any have a large number of edges. If the number of edges connected to the node is greater than β , the node is placed into the border and removed from further consideration. A limit, β_{limit} , is placed on the number of nodes that will be placed in the border from any particular diagonal block by using this test. The values used for β and β_{limit} are based upon our experiences with the nonsymmetric matrices in the Harwell–Boeing collection.

Nested dissection generates a submatrix of bordered block form. However, since the objective of the H2 ordering is to bring the submatrix into bordered upper triangular block form, nested dissection is too restrictive and the constraints on the separator set can be relaxed. This fact is exploited by the H2 ordering. After each stage when a separator set S is constructed, H2 reduces the number of nodes in the separator set by allowing additional fill-in to be created in the upper triangular part of the submatrix, thereby producing a quasi-separator set.

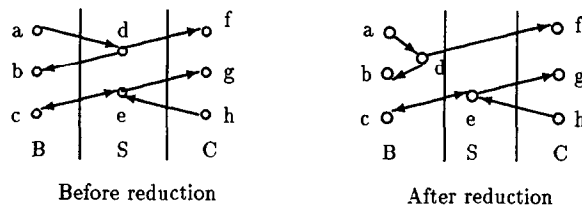


Fig. 4. Reduction of the separator set.

After a separator set S has been produced by the version of automatic nested dissection mentioned above, the graph G has been decomposed into a separator set S and two disjoint sets B and C . H2 attempts to reduce the size of S by looking only at the directed edges from A , E_A , and moving nodes out of S into either B or C as long as there are no edges from nodes in C to nodes in B . Edges are allowed from nodes in B to nodes in C . More formally, the reductions can be described as follows:

- (1) If there exists no edge $(y, x) \in E_A$ such that $y \in S$ and $x \in B$, then y may be moved to C .
- (2) If there exists no edge $(z, y) \in E_A$ such that $y \in S$ and $z \in C$, then y may be moved to B .

An example of the reduction of the separator set can be seen in Fig. 4. The node d may be moved into B since there is no edge from any node in C directed to the node d in S . The node e may not be removed from S since it does not meet the requirements for either of the reductions; and moving it out of S would destroy the desired structure.

An optimization to the reduction above involves moving nodes from B to C , or from C to B , so that the first two reductions can be applied to nodes for which the conditions of the reductions were not met with the initial contents of B and C . This is implemented by following the initial reductions with two enhancement phases.

The first phase consists of moving nodes from B to C together with applying the initial reduction techniques. A set of nodes $D \subset B$ is moved to set C if all of the following conditions are met:

- (1) There are no edges $(d, b) \in E_A$ where $d \in D$ and $b \in B$.
- (2) There exists $R \subseteq S$ such that there are edges $(y, d) \in E_A$ where $d \in D$ and $y \in R$; and there are no edges $(y, b) \in E_A$ where $y \in R$ and $b \in (B - D)$.
- (3) The size of the remaining part of set B is greater than the minimum size, $|B - D| > T_{remain}$.

After D is moved from B to set C , the initial reduction techniques on the separator set are repeated.

Symmetric conditions can be defined to allow the motion of a set of nodes from C to B before repeating the initial reduction techniques. A set of nodes $D \subset C$ is moved to B if all of the following conditions are met:

- (1) There are no edges $(c, d) \in E_A$ where $d \in D$ and $c \in C$.
- (2) There exists $R \subseteq S$ such that there are edges $(d, y) \in E_A$ where $d \in D$ and $y \in R$; and there are no edges $(c, y) \in E_A$ where $y \in R$ and $c \in (C - D)$.
- (3) The size of the remaining part of set C is greater than the minimum size, $|C - D| > T_{remain}$.

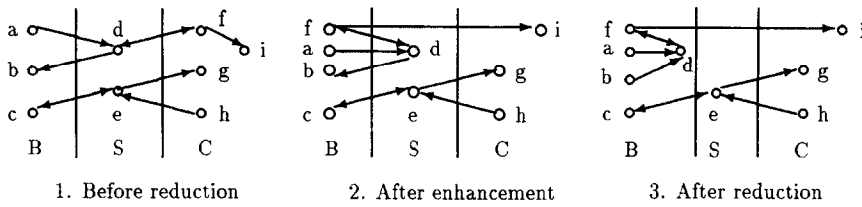


Fig. 5. Enhanced separator set reduction.

If all of these conditions are met, then the set D can be moved from C to B and the initial reduction techniques can be applied.

An example of this enhancement is provided in Fig. 5. None of the reductions may be applied to the initial separator set. However, the node f can move from C to B and, as a result, S can be reduced by moving the node d into B .

After the separator set has been reduced, it is removed from the graph and the algorithm is applied recursively to the two sets B and C until the resulting blocks are less than the desired maximum block size, T_{done} .

3.6. Results for H^*

This section presents the results for the hybrid ordering H^* that were collected on one cluster of Cedar, an Alliant FX/8. These results include border size, diagonal block sizes, and performance results which include the ordering time. The interested reader should consult [21] for details concerning the tuning of the heuristics that produced the data presented below.

In this paper, we restrict ourselves to the description of the following parameter settings which were used for each of the phases of H^* . The choice for each of these parameters is based on the arguments below, utilizing the following definitions:

- N = the size of the original matrix;
- $bsize_{orig}$ = the size of the block at the beginning of this algorithm phase; and
- $bsize_{found}$ = the size of the new block which has been found at the current stage of the algorithm.

H0 parameters.

- $\alpha_{initial} = 10^{-1}$. This value was chosen since it is the typical value used for the stability bound in sparse direct solvers.
- $\alpha_{min} = 10^{-5}$. The selection of this value is a tradeoff between performance and stability. The value chosen was based upon the experiments conducted with the test matrices.

H1 parameters.

- $T_{done} = N/10$. This value is based upon the target architecture. To achieve a good load balance on the four clusters of Cedar, a target of at least 2 blocks per cluster was set. With this parameter value, the size of the border can be up to 20% of the matrix and there should still be at least 8 blocks to be divided among the clusters.

- $T_{long} = 5$. This value was chosen based upon the experiments with the test matrices.
- $T_{minb} = N/20$. The value chosen for the minimum block size is $1/2$ of the maximum block size T_{done} . This value was chosen to keep the total number of blocks from growing too large.
- $T_{maxb} = bsize_{orig} * 3/4$. This value is used, in conjunction with T_{maxsep} , to make sure that the size of the block left after removing the new diagonal block and the separator block is adequate. These values are set such that after removing the rows, at least 20% of the rows in the original block will be left in the remaining block. The values chosen were based upon the experiments conducted with the test matrices.
- $T_{maxsep} = bsize_{found}/20$. This value is used to control the size of the border and is used in conjunction with T_{maxb} to control the size of the remaining block.

H2 parameters.

- $T_{done} = N/10$. This is the same parameter as is used by H1.
- $\beta = N/10$. This value was chosen based upon the experiments with the test matrices.
- $\beta_{limit} = bsize_{orig}/15$. This value was chosen based upon the experiments with the test matrices.
- $T_{remain} = N/25$. This value is 40% of the value used for T_{done} . This guarantees that when a block is divided into two new blocks, each block is at least 40% of T_{done} , with a separator block that is at most 20% of T_{done} . This is done to keep the total number of blocks from growing too large.

The tests were conducted using matrices from the Harwell–Boeing test collection. All the matrices chosen were from the real, nonsymmetric, assembled (RUA) set. The RUA set has 95 matrices, of which three are structurally singular and are not considered. Because H^* is meant to identify large grain parallelism, results for H^* will be presented for only fourteen of the matrices having at least 1,000 rows.

Table 1 contains the results for the application of the H^* ordering. This table contains: the transversal bound, α , which was found by H0; the total time for the H^* ordering (user process time in seconds) on one processor of Cedar; the total number of diagonal blocks after the ordering; the number of rows in the border; and the order of the largest diagonal block. The use of an unbounded transversal is indicated in Table 1 by a “*” in the bound column.

As can be seen from these tests, matrices tend to generate a large number of blocks. Five of the matrices generate over 900 blocks, whereas only four of the matrices generate less than 25 blocks. These small blocks are generated by Tarjan’s algorithm, which finds many small blocks of either one or two nodes. Methods for dealing with these small blocks efficiently will be discussed in Section 7.

The results of H^* can be compared with the related orderings produced by the P^4 algorithm [33], the P^5 algorithm [14], and the level set algorithm by Arioli and Duff [2] on a subset of the Harwell–Boeing matrices, the results of which are available in the literature [2,3]. This subset comprises the Grenoble matrices and the Westerberg’s matrices. The matrices range in order from 67 to 2021.

Table 1
H* Statistics for large RUA matrices

Matrix name	Rows	Non-zeroes	H0 bound	Total time	Total blocks	Border rows	Max. block
gaff1104	1104	16056	10^{-4}	2.14	190	202	108
gemat11	4929	33185	10^{-2}	4.04	437	348	404
gre_1107	1107	5664	10^{-1}	3.65	23	324	103
hwatt2	1856	11550	10^{-8}	2.23	142	430	158
mahistlh	1258	7682	*	1.45	930	74	124
nnc1374	1374	8606	10^{-9}	3.26	91	244	130
or678lhs	2529	90158	10^{-6}	7.20	2000	355	170
orsreg_1	2205	14133	10^{-1}	3.07	15	438	160
pores_2	1224	9613	10^{-5}	2.53	21	245	105
saylr4	3564	22316	10^{-1}	5.15	22	634	333
sherman2	1080	23094	10^{-7}	5.40	220	352	102
sherman3	5005	20033	10^{-1}	4.09	2119	423	394
sherman5	3312	20793	10^{-6}	4.70	1680	303	310
west2021	2021	7353	10^{-6}	5.89	1261	93	188

Table 2 shows the number of rows in the border of the matrix after the application of the algorithms and the size of the largest diagonal block remaining in the matrix after the application of the orderings. A value of “N.A.” indicates the result was not available in the literature. The results from P^5 are omitted from this table since, as indicated in [2], P^5 usually generates blocks of size 1 or 2, with an occasional block of size 3.

Table 2
Number of rows in border and largest diagonal block size

Matrix	Order	Border size				Largest block		
		H*	P^4	P^5	Level set	H*	P^4	Level set
gre_115	115	33	15	15	18	10	< 3	56
gre_185	185	86	28	28	52	16	< 3	69
gre_216	216	73	24	25	53	19	5	82
gre_216	216	70	24	25	N.A.	11	5	N.A.
gre_343	343	102	42	52	65	33	9	138
gre_512	512	148	50	55	106	49	5	211
gre_1107	1107	324	100	113	126	103	4	447
west0067	67	25	11	13	12	6	14	26
west0132	132	15	3	4	6	13	10	< 3
west0156	156	3	3	4	4	12	4	2
west0167	167	7	3	4	4	15	14	30
west0381	381	103	52	53	81	38	18	126
west0479	479	85	38	42	45	41	4	69
west0497	497	35	18	20	12	48	18	15
west0655	655	99	54	66	62	65	4	102
west0989	989	69	77	84	106	85	4	48
west1505	1505	79	116	127	112	145	4	79
west2021	2021	93	160	175	156	188	4	455

The comparison of the orderings with respect to the resulting block sizes must be interpreted with care since, in the final analysis, we are interested in their efficacy in terms of computing time when coupled with a parallel system solver. Nevertheless, some relevant points can be made.

Clearly, the P^4 and P^5 orderings produce smaller borders as well as smaller diagonal blocks than H^* and the level set algorithm. This is not surprising given that they use nonsymmetric permutations, which are more flexible. Unfortunately, the small diagonal blocks and border have less than satisfactory properties when coupled with a factorization algorithm. As Arioli and Duff point out in [2], the small diagonal block sizes can cause difficulties with both parallelism and the ability to choose stable pivots when the pivot searches are constrained to the diagonal blocks. Further attempts to improve stability via pivoting produced a prohibitive cost and the use of simple iterative refinement did not result in satisfactory recovery of accuracy [3]. Since MCSPARSE would suffer similar stability problems if the pivot selection was constrained to just the diagonal blocks, we will present an alternative method for maintaining stability.

The difficulties in the coupling of P^4 and P^5 with a stable factorization method motivated Arioli and Duff to consider other methods, including the level set ordering. It, like H^* , uses *symmetric* permutations. In general, the level set algorithm creates smaller borders but significantly larger diagonal blocks than H^* .

From this comparison, we see that H^* produces a reasonable compromise of diagonal blocks large enough to serve as the basis for a pivoting strategy and the exploitation of multiple levels of parallelism without becoming too large, at the cost of a somewhat larger border.

A few comments about the reuse of the ordering from H^* are in order. Though the H^* ordering uses values from the matrix to determine the transversal, this does not prevent the ordering from being reused for other matrices with the same structure. The weighted transversal is used in an attempt to improve the stability, but the transversal will be valid for any matrix with the same structure. Furthermore, it is possible that matrices with the same structure may also have similar values, allowing the weighted transversal to still have a positive effect.

4. MCSPARSE overview

The MCSPARSE solver was designed to be a large grain parallel, sparse, nonsymmetric, direct solver for the Cedar architecture that complements the H^* ordering.

There are many ways to solve a system of equations defined by a matrix with the structure in Fig. 6. We have chosen to implement a version which isolates the operations on each type of subblock – D_i 's, C_i 's, and B_i 's – and the coupling block F . This produces a modular algorithm which lends itself to parallelism and facilitates the modification of the solver to function as a parallel preconditioner in a direct/iterative hybrid solver [17,48]. For a discussion of another large grain approach which does not isolate the border, diagonal, and off-diagonal blocks during the factorization, see the discussion of LORA in [20].

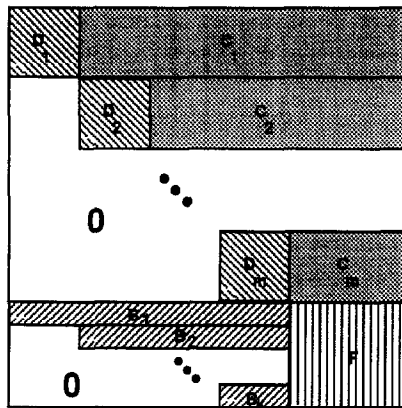


Fig. 6. Bordered block upper triangular form.

It should be noted that though the H^* ordering placed the rows into specific diagonal blocks, H^* did not determine the final pivot sequence. Pivoting can be done within the diagonal block to improve stability or fill-in without affecting the overall structure of the matrix. Furthermore, *casting* can be used to perform pivoting between the diagonal blocks and the border.

The factorization in MCSPARSE is performed in four stages.

- I. The first stage is the factorization of the diagonal blocks. Pivoting is used within each diagonal block D_j to find pivot elements which satisfy both stability and fill-in constraints. First, candidate pivots are selected using a modified Markowitz criteria that estimates expected off-diagonal block C_j and border block fill-in (see Section 6). The pivot $p_{k,m}$, which has the lowest count, is then checked for stability. This test consists of two parts. First, the standard test is made, which is used in most direct solvers. It compares the magnitude of the candidate to the other elements in the column of the diagonal block (i.e., $|p_{k,m}| \geq \mu \times \max_k |a_{k,m}|$). All experiments in this paper use $\mu = 0.1$. Second, a check is made to see if the column is to be *cast* to the border (see Section 5). Due to the structure of the matrix, the reductions of the D_i 's can be performed simultaneously. Two implementations of the diagonal block factorization are provided. The first form uses one processor to perform the factorization on a diagonal block with blocks done in parallel; and, the second form uses all the processors in one of the four clusters of Cedar to perform the factorization. During the execution, the decision of which form to use is based upon the size of the diagonal block and the number of diagonal blocks to be factored by a given cluster.
- II. In the second stage, after each D_i is reduced, the resulting lower triangular transformation L_i can be applied to the off-diagonal block C_i .
- III. Next, the border blocks, B_i 's, are eliminated and the associated rows in F are updated using the upper triangular transformations, U_i 's, associated with the L_i 's and the updated C_i 's. During the elimination, the size of the pivots and the elements being eliminated are monitored to control error growth (see Section 5).

- IV. The final stage uses level-3 BLAS-based dense matrix techniques to perform the factorization of the updated coupling block F ; multiple clusters are used if needed.

5. Stability issues

A key problem for solvers such as MCSPARSE is maintaining the stability of the factorization while utilizing a large grain structure within the system. While the bordered block upper triangular form provides a large grain structure for the exploitation of parallelism, it also constrains the set of potential pivots during the first and third stages of the algorithm. In this section, we review the pivoting strategy used in MCSPARSE and present results demonstrating its ability to maintain reasonable accuracy in the solution.

Casting. The typical implementation of a large grain solver applies Gaussian elimination to each diagonal block to calculate a local LU factorization. These local factorizations are then used, with the same pivot sequence, to eliminate the nonzero elements in the border. An implementation such as this has several problems. Though the system may be well-conditioned and nonsingular, there is no guarantee the diagonal blocks will be the same. In addition, even though the local factorizations may exist and be accurately computed, the pivot choices may result in substantial error growth when applied to the rows in the border.

The usual method for generating a stable factorization is to use a global pivoting strategy, which allows row and/or column permutations within the entire active portion of the matrix. Such permutations, however, may destroy the bordered block upper triangular form upon which the large grain parallel solver relies. Permutations which do not destroy this structure include those which are done within a diagonal block and those which exchange a row in a diagonal block with a row in the border.

For example, pairwise pivoting could be used to eliminate the rows of the border in parallel [8]. This preserves not only the general structure, but also the number of rows in each of the diagonal blocks and the border. There are some drawbacks, however. If the matrix was, in fact, symmetric in pattern, then the symmetry has been destroyed. Pairwise pivoting can also permute the relatively dense rows that tend to appear in the border into the diagonal blocks. This can cause unnecessary fill-in in the border and, if the border elimination and diagonal block factorization have been intermixed, in the diagonal and off-diagonal blocks as well. The fact that potentially all of the border rows eliminated by a diagonal block will require interchanges implies that the overall bound on the growth factor of the elimination is larger than that for strategies having only one or two comparisons per pivot column or row. The complexity of the synchronization during the factorization (particularly if the strategy of *S-Blocks* discussed below is used) increases considerably compared to the one adopted by MCSPARSE. Finally, the resulting factorization cannot be characterized as simply as those that result from MCSPARSE's strategy.

Other strategies discussed in the literature have resulted in solvers with either unacceptable cost or stability control (e.g., [2,3]). The strategy we prefer is one which preserves, up to a point, the overall structure of the matrix while allowing the

implementation of a global pivoting strategy that yields a factorization with stability similar to more conventional nonsymmetric solvers.

The strategy used in MCSPARSE, called *casting*, is described in detail for both arbitrary and bordered block upper triangular matrices in [16,21,40]. We review it briefly here.

Casting is a symmetric permutation that decreases the size of the diagonal block containing the cast pivot by moving a row and column into the border, thereby increasing its size by one. More formally:

Definition 5.1. A pivot p_{ii} is said to be *cast* if the system is permuted by the column permutation

$$(1, 2 \dots i-1, i, i+1 \dots n) \rightarrow (1, 2 \dots i-1, i+1 \dots n, i)$$

followed by an identical row permutation.

Casting is used in the first stage (the diagonal block factorization) and the third stage (the border row updates) of the algorithm. Casting in the first stage, referred to as *diagonal casting*, is not required to maintain the existence and stability of the factorization, but it can help maintain stability and enhance performance. At this point, casting is effectively delaying the elimination of the row and column until all the diagonal blocks have been processed. As in pairwise pivoting, local pivots can be used to bound the elements of the transformation matrices even if the submatrix being transformed is singular, or nearly so (0 columns are merely left untouched). However, when pivoting with the elements in the border is highly likely, casting can simplify matters. For example, if an entire pivot column is small (or 0), either absolutely or relative to a norm of the submatrix or the entire matrix, then pivoting is very likely (or must be done) during the border elimination. This pivoting will be done via the second type of casting discussed below and requires fairly complex synchronization. All such pivots in the diagonal blocks are marked as *cast* when encountered and can be permuted to the border at a convenient time in the algorithm (e.g., after the first stage).

In addition to reducing the complexity of the synchronization needed, diagonal casting also allows the elimination of the associated column to be completed with only a single pivot choice, as opposed to border casting which requires two pivots, and thus helps keep the growth factor down. More formally, after a candidate pivot element, $p_{k,k}$, has been selected based on the modified Markowitz count and the magnitude check relative to other elements in the column of the diagonal block, its magnitude is also checked relative to a diagonal casting parameter δ_1 . For simplicity, we assume that a single δ_1 is used. It could, of course, be selected based on local criteria and be different for each D_i (e.g., $\|D_i\|_\infty$). If $|p_{k,k}| < \delta_1$, then it is marked as *cast* and the k th column is considered no further in the transformation of the associated diagonal block. A simple modification to this inequality can be made to aid the anticipatory nature of diagonal casting. One could also compare $|p_{k,k}|$ to the element with the largest magnitude in that column in the border (i.e., if $|p_{k,k}| < \delta_1$ or $|p_{k,k}|/|b_k| < \delta_2$, where b_k is the element with the largest magnitude in the k -th column of the border in the original matrix, then it is cast). Of course, this does not take into account fill-in elements nor changes due to the updates during elimination.

Since the nature of diagonal casting is anticipatory, parameter choice is crucial to its success. If the parameters are too conservative, performance may degrade due to excessive growth of the border from elements where the anticipation of pivoting with the border was wrong. On the other hand, if the parameters are too optimistic, very little diagonal casting is done and all the pivoting must be completed during the elimination of the border in a more costly manner. See below where the preferred choices for these parameters are described.

Casting in the third stage is called *border casting* and is necessary for the existence and stability of the factorization. In the border update, a pivot selected in stage 1 is used to eliminate as many elements of that column in the border as possible. A stability check similar to that used to select the pivot is made before each attempt at eliminating a border element. If the ratio of the magnitude of the pivot element to the magnitude of the border element is less than some parameter δ_3 , the pivot is marked as cast. Of course, many border rows could be applying the same pivot simultaneously, so when the actual casting is performed depends upon the form of parallelism used during the elimination of the border.

The elements of the column in the border that were not eliminated, and the pivot element itself, add a column to the diagonal border block F . As a result, these elements are eliminated with no more than one additional pivot selection during the factorization of F .

Note that casting a pivot from a diagonal block into the border also implies that a portion of that row is added to the off-diagonal portion of the border. So, in addition to the second pivot selected for the cast column during the factorization of F , we have further eliminations to be done on the cast column during the border factorization. Since this row and column have already been used as a pivot row and column in stage 1, it is not possible for the computed factorization to be written as the LU factorization of a permutation of the original matrix. However, since there are at most two pivots per column, the factorization can be characterized algebraically in a manner very similar to the standard LU factorization [22]. This simplifies the way in which the factorization can be saved and used to solve systems with right-hand sides supplied later [21].

Given that there can be at most two pivots per column, if the condition for a pivot to be considered as a *stable* pivot is taken to be that of partial pivoting (i.e., the pivot is larger in magnitude than the elements eliminated), then the growth factor in the error analysis of the factorization is easily derived. The growth factor for the factorization algorithms above applied to a dense matrix $A \in \mathbb{R}^N$ is bounded by $2 \times 2^{N-1} = 2^N$. The extra factor of 2 is due to the two pivots per column. In the case of sparse matrices, the growth factor bound can be reduced significantly and depends upon the sparsity of the matrix [5,23]. When the partial pivoting conditions are relaxed, or another pivoting strategy is combined with casting, the bound on the growth factor is easily deduced from modifications to the standard analysis.

It is common for most direct factorization algorithms to use a simple iterative refinement postprocessing step to mitigate any accuracy problems caused by the tradeoffs between sparsity and stability. MCSPARSE also makes use of this technique. However, when the tradeoffs between stability and sparsity become too large, MCSPARSE can be joined with more sophisticated iterative methods. In addition, the MCSPARSE

structure can be used with incomplete factorization methods to form an iterative solver [20].

Stability results. In order to investigate the efficacy of the casting-based pivoting strategy, MCSPARSE was used to solve systems defined by the large matrices (≥ 1000 rows) from the RUA portion of the Harwell–Boeing test set. A comparison was made between MCSPARSE using both border and diagonal casting, MCSPARSE using both border and diagonal casting and iterative refinement, and MA28 using the stability parameter setting of $\mu = 0.1$ and iterative refinement. The tests were performed using one processor of Cedar. Within MCSPARSE, the following stability parameter settings were chosen.

- $\delta_1 = 10^{-5}$. This value was found to work reasonably well for the test matrices. However, for four of the matrices (hwatt_1, hwatt_2, nnc1374, and sherman3), this type of casting was found to cast too many rows and was disabled for these matrices only.
- δ_2 . Casting relative to the original elements in the border was found not to improve the stability and, therefore, this parameter was not used.
- $\delta_3 = 10^{-6}$. By experimentation, this value was found to work well for most matrices. However, for nnc1374, this bound cast too many rows and was changed to $\delta_3 = 10^{-9}$.

Table 3
Comparison of the relative errors for MCSPARSE and MA28

Matrix	MCSPARSE		MA28
	Without iterative refinement	With iterative refinement	With iterative refinement
gaff1104	$2.0 * 10^{-7}$	$2.0 * 10^{-7}$	$4.1 * 10^{-7}$
gemat11	$5.1 * 10^{-11}$	$5.1 * 10^{-11}$	$2.3 * 10^{-11}$
gemat12	$4.0 * 10^{-7}$	$3.1 * 10^{-11}$	$4.4 * 10^{-11}$
gre_1107	$6.7 * 10^{-6}$	$9.0 * 10^{-10}$	$1.1 * 10^{-9}$
hwatt_1	$1.8 * 10^{-14}$	$1.8 * 10^{-14}$	$7.2 * 10^{-15}$
hwatt_2	$4.8 * 10^{-14}$	$4.8 * 10^{-14}$	$2.7 * 10^{-7}$
mahisth	$1.2 * 10^{-9}$	$6.8 * 10^{-12}$	$6.3 * 10^{-14}$
nnc1374	$5.8 * 10^1$	$5.9 * 10^{-4}$	$1.7 * 10^{-5}$
or678lhs	$3.0 * 10^{-14}$	$3.0 * 10^{-14}$	$2.8 * 10^{-12}$
orsirr_1	$2.1 * 10^{-13}$	$2.1 * 10^{-13}$	$1.5 * 10^{-13}$
orsreg_1	$2.2 * 10^{-13}$	$2.2 * 10^{-13}$	$1.5 * 10^{-13}$
pores_2	$3.5 * 10^{-7}$	$5.4 * 10^{-13}$	$4.6 * 10^{-11}$
saylr4	$1.2 * 10^{-11}$	$1.2 * 10^{-11}$	$7.8 * 10^{-12}$
sherman1	$1.5 * 10^{-13}$	$1.5 * 10^{-13}$	$5.0 * 10^{-14}$
sherman2	$1.3 * 10^{-3}$	$1.3 * 10^{-11}$	$3.1 * 10^{-9}$
sherman3	$5.8 * 10^{-13}$	$5.8 * 10^{-13}$	$5.8 * 10^{-13}$
sherman4	$3.4 * 10^{-15}$	$3.4 * 10^{-15}$	$2.0 * 10^{-14}$
sherman5	$6.2 * 10^{-8}$	$7.5 * 10^{-15}$	$1.5 * 10^{-13}$
west1505	$4.1 * 10^{-8}$	$2.5 * 10^{-10}$	$1.2 * 10^{-9}$
west2021	$4.8 * 10^{-7}$	$1.6 * 10^{-10}$	$1.4 * 10^{-9}$

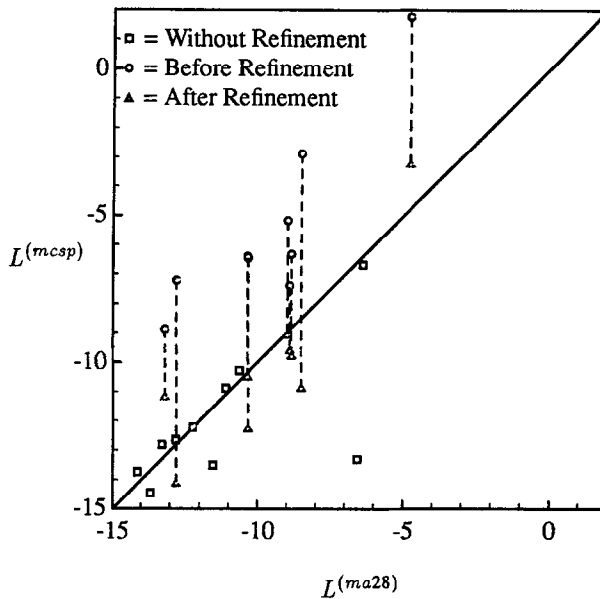


Fig. 7. The log of the relative error for MCSPARSE (L^{mcsp}), with and without iterative refinement, as compared to the log of the relative error for MA28 (L^{ma28}).

For both MCSPARSE and MA28, the iterative refinement stopping criterion were set to achieve high stability. The refinement continued until the estimate of the relative error was reduced to less than 10^{-14} , or until the convergence rate had slowed such that the decrease in the maximum norm of the correction vector was less than 1%. The results of these tests are presented in Table 3.

As can be seen from the table, MCSPARSE without iterative refinement produces a solution that is essentially as good or better than MA28 for eight of the twenty matrices. Indeed, for one of the matrices, *hwatt2*, the local pivot searches of MCSPARSE yield an error considerably better than MA28 – $E^{mcsp} = 10^{-14}$ and $E^{ma28} = 10^{-7}$. Of the other twelve matrices, iterative refinement improves the MCSPARSE solution to at least as good as that of MA28 for seven. Five of the matrices do not yield results better than MA28, despite the use of iterative refinement. For four of these, $E^{mcsp} \leq 10^{-11}$ so that the solution, even though not as good as MA28, is very accurate. For the other matrix, *nnc1374*, $E^{mcsp} = 10^{-4}$, which is not satisfactory. However, note that $E^{ma28} = 10^{-5}$, indicating that the matrix is a fairly difficult problem.

To better show the stability results, these results are compared graphically in Fig. 7. The log of the relative error when solved with MCSPARSE is plotted against the log of the relative error when solved with MA28. If iterative refinement was not needed for the MCSPARSE solution, or if it did not improve the relative error, then only one mark, the “Without Refinement” mark, is plotted for the matrix. If iterative refinement was used to improve the relative error of the MCSPARSE solution, then both relative errors, with and without refinement, are plotted and connected by a vertical line.

For all matrices where MCSPARSE ends up below the diagonal line, the MCSPARSE error is better than the MA28 error. As can be seen from the graph, for only two matrices, nnc1374 (which is discussed above) and mahist1h (where the MCSPARSE error of 10^{-11} is already considered good), is the MCSPARSE error less than the MA28. The MCSPARSE error is close to, or better than, the MA28 error for the other eighteen matrices.

6. Methods for fill-in control

Fill-in control is more difficult for a large grain parallel solver than for its medium grain counterparts due to the reduced amount of information available when performing the pivot selection in each subsystem. This is due to the fact that the number of fill-in elements generated outside of the subsystem is not known. To improve the overall effectiveness of the local method, estimates of the amount of fill-in generated outside the subsystem may be incorporated into the pivot selection criteria.

Most dynamic methods of pivot selection for nonsymmetric systems are based on the Markowitz count of an element. The Markowitz count for the element $a_{i,j}$, $M(i, j)$, is defined as follows:

$$M(i, j) = (r_i - 1) * (c_j - 1),$$

where

r_i = the number of nonzero elements in row i and

c_j = the number of nonzero elements in column j .

The pivot is taken to be the element within the active portion of the system with the smallest Markowitz count ($pivot = a_{m,n}$ where $M(m, n) = \min(M(i, j)) \forall i, j$) and which satisfies acceptable stability constraints. These counts must be updated as the factorization proceeds by removing from the counts elements in the pivot row and pivot column as well as adding any fill-in produced when applying the associated pivot.

This count can be easily modified to use estimates of the amount of fill-in outside of the diagonal blocks as follows:

$$M(i, j) = (r_i - 1 + \beta * or_i) * (c_j - 1 + \gamma * oc_j),$$

where

r_i = the number of nonzero elements in row i inside the subsystem,

or_i = estimate of the number of nonzero elements in row i outside the subsystem,

c_j = the number of nonzero elements in column j inside the subsystem, and

oc_j = estimate of the number of nonzero elements in column j outside the subsystem.

The multipliers β and γ are used to vary the influence of the estimates on the Markowitz counts. As with the original Markowitz counts, the number of elements in the rows and columns inside the subsystem must be updated whenever a pivot is applied.

But now, with the modified Markowitz counts, the estimates for the number of elements outside the subsystem must also be updated.

The estimates are initialized to the number of elements outside the subsystem for each row and column in the original matrix. Each time a pivot element is chosen, the estimates for the pivot row and pivot column are used to update the estimates for the target rows and target columns. Specifically, the updates for the target rows and columns are of the form:

$$or_i = or_i + \alpha * or_{pivot}, \quad oc_j = oc_j + \omega * oc_{pivot},$$

where $0 \leq \alpha, \omega \leq 1$. When α and ω are 0, this represents no fill-in elements being generated outside the subsystem. When α and ω are 1, this represents all elements in the pivot row generating a fill-in element in the target row. In the current version of the MCSPARSE, the fill-in parameters were chosen to be:

- $\alpha = 1.0$, to perform a worse case estimate for the fill-in generation in the off-diagonal block.
- $\beta = 1.0$, to equally weight the fill-in in the off-diagonal block with the fill-in in the diagonal block.
- $\omega = 1.0$, to perform a worse case estimate for the fill-in generation in the border.
- $\gamma = 1.0$, to equally weight the fill-in in the border with the fill-in in the diagonal block.

Fill-in results. Tests were run with MCSPARSE to determine the effectiveness of the modified Markowitz count and to compare the fill-in performance of MCSPARSE to a

Table 4
Fill-in ratio comparison of MCSPARSE and MA28

Matrix	$R^{(mcsp)}$	$R^{(ma28)}$	$R^{(mcsp)} / R^{(ma28)}$
hwatt_1	16.69	21.79	0.77
sherman3	15.52	18.39	0.84
hwatt_2	21.03	21.39	0.98
sherman4	4.99	4.91	1.02
saylr4	24.52	20.22	1.21
sherman1	7.47	5.40	1.38
gaffl104	6.64	3.89	1.71
sherman2	19.07	10.44	1.83
orsreg_1	18.46	9.66	1.91
sherman5	14.77	6.35	2.33
orsirr_1	18.83	6.56	2.87
pores_2	9.82	3.03	3.24
nnc1374	22.97	5.05	4.55
mahistlh	1.62	0.36	4.56
gre_1107	32.37	6.61	4.90
west1505	3.10	0.49	6.34
west2021	3.50	0.48	7.30
gemat11	4.24	0.56	7.54
gemat12	6.81	0.56	12.17
or678lhs	2.45	0.15	15.91

standard sparse solver, MA28. These tests were run using the same 20 matrices that were used for the stability experiments in Section 5 and were conducted on one processor of Cedar.

Table 4 contains, for each matrix, the ratio of the number of fill-in elements to the number of elements in the original matrix for MA28 using $u = 0.1$ and MCSPARSE, denoted $R_i^{(ma28)}$ and $R_i^{(mcsp)}$ respectively, as well as the ratio $R^{(mcsp)}/R_i^{(ma28)}$. Two lines were drawn on the table to help describe how MCSPARSE compares to MA28. The matrices above the upper line are the matrices where the MCSPARSE fill-in is less than or equal to the MA28 fill-in. The matrices between the upper line and the lower line are the matrices where the MCSPARSE fill-in is greater than the MA28 fill-in, but not more than twice the MA28 fill-in.

For three of the matrices, MCSPARSE is doing at least as well as MA28, and for another six of the matrices, MCSPARSE is generating no more than twice the fill-in of MA28. For these nine matrices, then, MCSPARSE seems to be doing a satisfactory job of controlling fill-in. For the other eleven matrices, however, MCSPARSE does not appear to be doing as well.

One reason MCSPARSE appears to generate more fill-in than MA28 is the use of dense storage for the border diagonal block in MCSPARSE. In the counts above, the border diagonal block was assumed to be fully dense. Since these elements are processed using a high-performance parallel dense solver, they do not contribute as much per element to the overall execution time as the sparse elements, and care should be taken with the comparison. A switch to dense techniques can also be done dynamically during the

Table 5
Fill-in ratio comparison of MCSPARSE and Y12M

Matrix	R^{mcsp}	$R^{(y12m)}$	$R^{(mcsp)}/R^{(y12m)}$
sherman3	15.52	43.19	0.36
sherman4	4.99	13.14	0.38
gaffl104	6.64	16.78	0.40
saylr4	24.52	59.28	0.41
hwatt_1	16.69	37.19	0.45
orsreg_1	18.46	40.22	0.46
pores_2	9.82	17.02	0.58
hwatt_2	21.03	34.90	0.60
sherman1	7.47	11.70	0.64
sherman5	14.77	22.95	0.64
orsirr_1	18.83	21.70	0.87
or678lhs	2.45	2.25	1.09
nnc1374	22.97	18.64	1.23
gemat11	4.24	3.07	1.38
sherman2	19.07	12.85	1.48
gre_1107	32.37	21.70	1.49
west2021	3.50	2.00	1.76
mahistlh	1.62	0.85	1.91
west1505	3.10	1.57	1.98
gemat12	6.81	3.25	2.09

Table 6

Comparison of the total element counts after factorization for MCSPARSE and Y12M

Matrix	MCSPARSE				Y12M			
	Sparse elems	Dense block	Dense elems	Total elems	Sparse elems	Dense block	Dense elems	Total elems
gaff1104	56560	254	64516	121076	15059	520	270400	285459
gemat11	52664	349	121801	174465	36616	313	97969	134585
gemat12	78632	426	181476	260108	36787	322	103684	140471
gre_1107	83691	324	104976	188667	9545	345	119025	128570
hwatt_1	65550	368	135424	200974	24221	640	409600	433821
hwatt_2	69300	430	184900	254200	26484	623	388129	414613
mahistlh	14320	76	5776	20096	6275	89	7921	14196
nnc1374	51176	370	136900	188076	15766	391	152881	168647
or678lhs	184643	355	126025	310668	43832	499	249001	292833
orsirr_1	25798	332	110224	136022	13536	377	142129	155665
orsreg_1	79346	438	191844	271190	36499	739	546121	582620
pores_2	25781	282	79524	105305	16422	396	156816	173238
saylr4	166530	634	401956	568486	63881	1132	1281424	1345305
sherman1	10147	147	21609	31756	5205	206	42436	47641
sherman2	171154	519	269361	440515	25048	543	294849	319897
sherman3	154186	423	178929	333115	42578	918	842724	885302
sherman4	12066	103	10609	22675	4248	222	49284	53532
sherman5	201848	359	128881	330729	23280	689	474721	498001
west1505	11234	100	10000	21234	5456	92	8464	13920
west2021	17062	117	13689	30751	7264	121	14641	21905

factorization rather than being confined to the predefined dense border diagonal block. A parallel MA28-like solver which exploits such a technique is the parallel version of Y12M developed for the Alliant FX/8 [19].

The fill-in ratios for Y12M, $R_i^{(y12m)}$, are presented and compared to those for MCSPARSE in Table 5. Again, a line has been drawn in the table to help compare the two solvers. The matrices above the line are those where MCSPARSE had fewer fill-in elements than Y12M; and, the matrices below the line are those where Y12M had fewer fill-in elements. As can be seen from this comparison, MCSPARSE had fewer fill-in elements for eleven of the matrices, and Y12M had fewer fill-in elements for nine of the matrices. Of the nine matrices where Y12M had fewer fill-in elements, the MCSPARSE fill-in rate was not more than twice the Y12M fill-in rate for eight of these matrices. The remaining matrix was just barely more than twice the fill-in rate of Y12M.

Since MCSPARSE and Y12M both perform a switch to dense matrix techniques, a simple comparison of the overall fill-in ratios is not complete enough. In Table 6, the numbers of elements in the dense and sparse portions of the matrices after factorization are separated. MCSPARSE tends to have more sparse elements than Y12M, which is also expected due to the fact that MCSPARSE preserves the block structure of the reordered matrix and therefore has less flexibility. Also note that for seventeen of the matrices, MCSPARSE has a smaller dense block than Y12M.

In reviewing the fill-in results for MCSPARSE, several observations relative to the effect

on final performance can be made. When comparing the number of fill-in elements between MCSPARSE and MA28, one should recall that MCSPARSE is a parallel solver and that MA28 is a sequential solver. A sequential solver attempts to keep the number of fill-in elements as small as possible in order to minimize the number of floating point operations. A parallel solver, however, can achieve better performance when generating more fill-in elements as long as sufficient parallelism is available in both the machine and computational dependencies. By using a reordering that allows the fill-in to be constrained to certain areas of the matrix structure, large grain parallelism can also be exploited with minimal synchronization cost. Furthermore, on a high-performance machine, dense matrix structures and techniques can be utilized more efficiently than sparse techniques for certain problem sizes. As a result, MCSPARSE on a parallel computer can outperform MA28 and Y12M even when generating significantly more fill-in elements. When comparing the number of fill-in elements between MCSPARSE and a solver with a switch to dense matrix capabilities, Y12M, the fill-in results appear favorable, with MCSPARSE generating fewer fill-in elements for just over half of the matrices.

Given that the fill-in affects the performance of the solvers in many different ways, a fill-in comparison is only one part of the performance comparison between solvers. In Section 8, the comparison between the solvers will be continued using the timing results of experiments on Cedar.

7. Load balancing

The major performance problem with a large grain solver is the difficulty in achieving a good load balance, which can also accommodate the way a sparse matrix may change during the factorization. The fill-in elements generated by the factorization may be unevenly distributed, leaving some rows unaffected while making other rows dense. In addition, matrices with similar structure but different numerical values can generate different fill-in patterns as each matrix may need its own pivot ordering for stability reasons.

Reblocking. The first step of the load balancing procedure is to determine what objects are to be partitioned. The bordered block upper triangular form consists of a series of diagonal blocks with corresponding off-diagonal blocks and a border. The solver may use the set of diagonal blocks provided to it, or it can combine consecutive diagonal blocks to form larger blocks.

Reblocking consists of taking the original diagonal blocks from the ordering,

$$D_j^{(o)} = \{\text{all rows } r_k \text{ in diagonal block } j\},$$

and combining consecutive diagonal blocks to form new blocks $D_i^{(r)}$ such that

$$D_i^{(r)} = D_j^{(o)} \cup D_{j+1}^{(o)} \cup \dots \cup D_{j+k}^{(o)}.$$

Reblocking can be done to make the size of the diagonal blocks, $|D_i^{(r)}|$, more uniform.

Whether or not the solver should combine diagonal blocks is dependent upon the target machine. When using a few large blocks, less communication may be required,

but balancing the load at the block level may be more difficult. Smaller blocks may be easier to load balance at the block level, but the increased number of blocks may require more communication. A machine with many processors may require a large number of small blocks to keep the processors busy, and a machine with fewer processors may be faster with fewer blocks.

Work count. The next load balancing step involves assigning the diagonal blocks to the clusters. To perform this assignment, the algorithm should consider how much work will be done with the diagonal block. This includes both the work that will be done to calculate the local factorization for the diagonal block, and the work that will be done to apply the local factorization to the border. The information available for estimating the work includes the number of rows in the diagonal block, the number of nonzero elements in the diagonal block, and the original number of border rows the diagonal block needs to update. (The diagonal block may actually update more border rows due to rows being cast into the border.)

Two different work estimates have been developed for the partitionings. The first work estimate combines the number of rows in the diagonal block with the number of border rows to be updated. For $A \in \mathbb{R}^{n \times n}$, define border row b_j as:

$$b_j = \{c_k \mid a_{b_j, c_k} \neq 0\}.$$

The border rows which need to be updated by diagonal block i are defined as

$$U_i = \{b_j \mid b_j^{(\min)} \leq d_i^{(\max)}\}$$

where

$$b_j^{(\min)} = \min_j c_j \in b_i \quad \text{and} \quad d_i^{(\max)} = \max_j r_j \in D_i^{(r)}.$$

In other words, U_i is the set of border rows such that each row has a nonzero to the left of $d_i^{(\max)}$ and D_j is assumed to update all border rows in U_i . The work count $w_i^{(rb)}$ for diagonal block $D_i^{(r)}$ can therefore be defined as:

$$w_i^{(rb)} = |D_i^{(r)}| * |U_i|.$$

This estimate is simple, yet it attempts to weight both the size of the diagonal block and the number of border rows updated. This work estimate is used to assign nearly equal amounts of estimated work to each cluster during the partitioning.

One problem with this work estimate is it assumes that diagonal blocks with a similar number of rows will be factored in roughly the same amount of time. This assumption ignores how the diagonal blocks were generated. Though the new diagonal blocks may have a similar number of rows, the amount of work to factor these new diagonal blocks may vary greatly due to the differences in the size of the original diagonal blocks which were combined to form the new diagonal block.

A second work count was developed to combine the size of the new diagonal block with the information about the original diagonal blocks. The largest diagonal block from the ordering contained within a reblocked diagonal block is found as

$$|D_{i, \max}^{(o)}| = \max_k |D_k^{(o)}| \quad \forall D_k^{(o)} \subseteq D_i^{(r)}.$$

The work count $w_i^{(or)}$ for diagonal block $D_i^{(r)}$ is calculated as

$$w_i^{(or)} = |D_{i,\max}^{(o)}| * |D_i^{(r)}|.$$

One factor these work estimates ignore, however, is the sequential nature of the border updates. (A border row can only be updated by one diagonal block at a time.) The partitioning algorithm can incorporate whichever diagonal block each border row starts with to determine where the most parallel work is available.

Static partitioning. Four different partitioning algorithms have been developed using the work estimates described above.

Partition $\mathcal{P}1$: This algorithm assigns the diagonal blocks associated with the start of a border row to the Cedar clusters in a round robin fashion. As a result, the diagonal blocks are interleaved among the clusters. The remaining diagonal blocks are distributed in an effort to keep consecutive diagonal blocks assigned to the same cluster, while keeping the amount of work assigned to each cluster equal. The work count $w_i^{(rb)}$ is used.

Partition $\mathcal{P}2$: This algorithm divides the diagonal blocks associated with the start of a border row into groups, where each group has the same number of blocks and the blocks are kept in order (i.e., the first cluster will receive the first n blocks). The remaining diagonal blocks are distributed in an effort to keep consecutive diagonal blocks assigned to the same cluster, while keeping the amount of work assigned to each cluster equal. The work count $w_i^{(rb)}$ is used.

Partition $\mathcal{P}3$: This algorithm assigns a consecutive group of diagonal blocks to each cluster, where each group has about the same work amount. The work count $w_i^{(rb)}$ is used.

Partition $\mathcal{P}4$: This algorithm assigns a consecutive group of diagonal blocks to each cluster, where each group has about the same work amount. The work count $w_i^{(or)}$ is used.

Each of these partitionings makes slightly different assumptions about how to balance the work load. For example:

- Partition $\mathcal{P}1$ assumes the border rows can be cheaply transferred between clusters when updating the border rows with the diagonal blocks.
- Partition $\mathcal{P}2$ assumes the cost of transferring the border rows is high and therefore attempts to keep the number of transfers low.
- Partition $\mathcal{P}3$ assumes that balancing the parallel border work is not important.
- Partition $\mathcal{P}4$ assumes that balancing the parallel border work is not important, but that the origin of the diagonal blocks is important.

All of these partitionings have a similar problem, which is a bottleneck caused by the sequential nature of the updates. Since a border row must be updated in order by the diagonal blocks, the last diagonal block will be the last block to update each border row. After all the other clusters have finished their updates, the cluster containing the last diagonal blocks will still be doing updates; and, since the border rows have had repeated updates already applied, it is likely the border rows will be denser.

Dynamic load balancing. One solution to this bottleneck is to allow other clusters to help with the updates using the last diagonal blocks. By placing the local factorizations

of the last diagonal blocks into global memory, the other clusters can share the work to be done by these blocks, called *S-Blocks*. Any cluster which has finished its work can help by applying the *S-Blocks* to the border rows which still need updated. This method has been implemented within MCSPARSE to help with the load balancing problem.

Processor load balancing. One last load balancing question is the determination of how many processors should be used to factor a diagonal block. A cluster is assigned a set of diagonal blocks and a number of processors which it may use to factor the blocks. The cluster may factor a block using all of its processors, or it may factor multiple blocks at a time using one processor per diagonal block. The goal for the cluster is to factor all of the blocks in the minimum amount of time. It is also important to realize that the processors within a cluster will share the cache on the cluster.

A threshold, σ , is set by the user to indicate the maximum size of a diagonal block to be factored with one processor. The factorization of any diagonal block with a size greater than σ uses all processors in a single cluster.

Though it may be more efficient to solve the small blocks in parallel, if there are fewer than $\eta * P$ blocks available, where P is the number processors, then $(1 - \eta) * P$ processors would be idle during the factorization. In this case, it may be quicker to use the parallel solver to factor each of the diagonal blocks in turn as it can utilize more processors. The number of diagonal blocks assigned to the cluster below the threshold are found,

$$D_{small} = \{D_i \mid |D_i| \leq \sigma\}.$$

If $|D_{small}| \geq \eta * P$, then the diagonal blocks in D_{small} will be factored with one processor per diagonal block. However, if $|D_{small}| < \eta * P$, then the diagonal blocks in D_{small} will be factored using all P processors per diagonal block. All other diagonal blocks assigned to the cluster are factored in turn using all processors in the cluster.

Currently the parameters are set to:

- $\sigma = 100$. This is based upon the parallel performance of the processors within a Cedar cluster. Due to the overhead in the parallel diagonal block factorization, it is efficient to solve a diagonal block on multiple processors only when its size is greater than 100. For diagonal blocks whose size is less than 100, it is more efficient to solve multiple blocks in parallel.
- $\eta = 1/3$. This parameter is also based upon the parallel performance of the processors within a Cedar cluster and the parallel performance of the diagonal block factorization. Therefore, as long as $1/3$ of the processors will be busy, the small diagonal blocks will be solved in parallel.

Load balancing results. This section presents the results from testing both the static and dynamic load balancing methods. The static load balancing methods, \mathcal{P}_i 's, are compared first and then the effects of *S-Blocks* will be examined. In addition, during all testing both reblocking and processor load balancing were performed.

To determine the effectiveness of the different partitionings, the four partitionings were tested on Cedar, using all four clusters and eight processors per cluster, and compared against MCSPARSE on one cluster using eight processors. Because these

Table 7

The speedup of the 4 cluster code over the 1 cluster code for the different partitionings \mathcal{P}

\mathcal{P} Matrices ordered by sequential factorization time																				
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	1.3	1.2	1.3	1.3	1.7	1.6	1.4	1.5	1.9	1.5	1.8	1.9	2.0	2.5	2.9	2.6	2.1	2.5	2.4	2.5
2	1.8	1.6	1.2	1.6	1.9	1.6	1.5	1.7	2.2	1.5	2.0	1.8	2.6	3.1	2.5	2.7	2.0	2.8	2.8	2.8
3	1.2	1.5	1.1	1.5	1.5	1.5	1.6	1.9	2.1	1.4	1.5	1.5	2.6	1.8	2.0	2.0	1.8	1.9	1.6	1.7
4	1.6	1.4	1.8	1.4	1.6	1.7	1.6	1.8	2.3	2.1	2.1	2.1	2.5	2.6	3.0	2.7	2.2	3.0	3.1	2.6

partitionings are designed to balance the work load between clusters and to address intercluster issues, the cluster speedup provides an indication of how well each partitioning performs. In Section 8, comparisons will be made for MCSPARSE on 1 and 32 processors, as well as comparisons to other one-cluster solvers.

The four cluster factorization times were used with the factorization time for the single cluster code to calculate the cluster speedups, with a maximum possible speedup of 4. These tests were performed for the 20 large matrices, and the number of *S-Blocks* was chosen to be $1/2$ of the diagonal blocks (this choice is based upon the results of the dynamic load balancing tests presented below). The resulting speedups are presented in Table 7, with the best speedup for each matrix highlighted. (The matrices are ordered as they appear in Table 9.)

Though all of these partitionings have their advantages, for our target machine Partition $\mathcal{P}2$ was anticipated to be the best since it considered the overhead of transferring data between clusters. As can be seen from the data, however, partition $\mathcal{P}4$ provides better results for more matrices than $\mathcal{P}2$ and $\mathcal{P}3$. Given that $\mathcal{P}3$ and $\mathcal{P}4$ are better than $\mathcal{P}2$ for 12 of the matrices, this shows that the other overheads are just as important as the data transfer between clusters. Because of the different work counts used by partitions $\mathcal{P}2$ and $\mathcal{P}4$, these results also suggest there are two different load balancing problems. For some matrices it is more important to balance the border updates, and for other matrices it is more important to balance the diagonal block factorizations. It should be noted that during the testing of the different partitionings, half of the diagonal blocks were *S-Blocks*. This indicates that the partitionings have a dramatic performance effect even when dynamic load balancing is used during the border update.

To investigate the effects of the dynamic load balancing with *S-Blocks*, tests were run with MCSPARSE on the four clusters of Cedar using partition $\mathcal{P}4$ (since it provided the best performance during the static load balance tests) with a varying number of *S-Blocks*. The number of *S-Blocks* used during the tests was calculated as a fraction of the total number of diagonal blocks. (Where $S = \{\text{the set of } S\text{-Blocks}\}$ and $D = \{\text{the set of diagonal blocks}\}$, the fraction of blocks which are *S-Blocks* is calculated as $|S|/|D|$.) The factorizations of the twenty large matrices were calculated using no *S-Blocks* and with $|S|/|D| = 1/4, 1/3$, and $1/2$. After the tests were run, the speedups were calculated by comparing the factorization time to the time for the one cluster code using eight processors. The speedups are presented in Table 8, with the best speedup per matrix being highlighted. (The matrices are ordered as they appear in Table 9.)

Table 8

The speedup of the 4 cluster code over the 1 cluster code for the different number of *S-Blocks*

$ S / D $	Matrices ordered by sequential factorization time																			
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	1.2	0.8	1.5	1.6	1.4	1.2	1.1	1.7	1.5	1.7	1.9	2.0	1.3	2.4	2.7	2.4	1.4	2.2	2.4	2.4
1/4	1.4	1.4	1.7	1.4	1.5	1.5	1.4	1.4	1.8	2.0	1.8	2.1	1.3	2.6	3.0	2.5	1.4	2.8	2.3	2.6
1/3	1.4	1.2	1.7	1.4	1.7	1.5	1.3	1.8	1.9	2.0	2.2	2.2	2.5	2.6	2.9	2.6	1.3	2.9	3.2	2.8
1/2	1.6	1.4	1.8	1.4	1.6	1.7	1.6	1.8	2.3	2.1	2.1	2.1	2.5	2.6	3.0	2.7	2.2	3.0	3.1	2.6

This table shows that the use of *S-Blocks* helps the performance for 19 of the 20 matrices, as compared to using zero *S-Blocks*. Therefore, using *S-Blocks* does not hurt. The performance degradation which results from not using *S-Blocks* is drastic for several of the matrices, with two of the matrices (numbers 13 and 17) having their time increased by about 40% when *S-Blocks* are not used. Of the different parameter values, using $|S|/|D| = 1/2$ provided the best result for 14 of the 20 matrices. Whereas the next best value, $|S|/|D| = 1/3$, provided the best results for only 8 of the 20 matrices.

Tests were not performed with more than half of the diagonal blocks being *S-Blocks* for architectural reasons. As more *S-Blocks* are used, more shared global memory must be allocated to hold the blocks. When solving larger and larger problems, however, the memory available for the *S-Blocks* will decrease, assuming the border size and/or density increases. For *S-Blocks* = 1, MCSPARSE essentially defaults to a shared memory approach running in global memory which, due to memory size constraints and communication costs, would not be efficient on the clustered Cedar architecture.

These results show the importance of determining the correct load balancing strategy for the individual problem being solved, with a combination of dynamic and static load balancing methods necessary for MCSPARSE to perform well on Cedar. Note that though it is convenient to implement *S-Blocks* using the shared global memory of Cedar, the global memory is not required. *S-Blocks* can be implemented on a distributed memory machine by replicating the *S-Blocks* on the nodes, although this does alter the choice of diagonal blocks placed in the set of *S-Blocks* (i.e., the set will no longer be contiguous).

8. Hierarchical parallelism

In order to achieve acceptable performance on a target machine, a solver must be able to effectively match the architectural resources (e.g., vector units, multiple processors and hierarchical memories) to the correct level of algorithmic parallelism. If the machine has only vector units, then fine grain parallelism is sufficient; if the machine consists of several tightly coupled processors, then medium and fine grain parallelism suffice; or, given multiple loosely coupled processors or clusters of processors, large grain parallelism is necessary.

More complex architectures often possess all three levels of parallel coupling and a complex memory system and thereby require the simultaneous exploitation of multiple

levels of algorithmic parallelism. Cedar is such an architecture. It consists of processor clusters loosely coupled through a shared memory. Each cluster contains eight tightly coupled vector processors which share a cache, and a local cluster memory which is not accessible by any processor outside the cluster.

The mapping of the bordered block upper triangular form of the matrix onto the Cedar architecture to utilize the multiple levels of parallelism is straightforward. The diagonal blocks map onto the cluster for large grain parallelism; the rows map onto the processors for medium grain parallelism; and the elements within a row map to the vector units of the processors for fine grain parallelism. This mapping demonstrates one way to utilize the features of the architecture, but this is not the only possible mapping. When implementing the solver, several assumptions about the mapping must be examined. The mapping assumes there are enough rows in the diagonal block for the processors of the cluster to be effective. The current H^* ordering [40], however, usually generates a large number of small diagonal blocks containing between one and four rows. Since a full Cedar cluster contains eight processors, such diagonal blocks would be unable to use more than half of the processors. A method for overcoming this problem was described in Section 7. Hierarchical parallelism, in combination with these methods, allows the solver to take advantage of the available resources of the Cedar architecture when operating on the diagonal blocks. The solver, however, must operate on more than just the diagonal blocks. The factorization of the border presents a similar performance problem.

The border consists of multiple rows which contain the connections between the diagonal blocks. Each nonzero border element in an off-diagonal column must be eliminated by applying the local factorization of the corresponding diagonal block. For example, consider a border row i with nonzero elements in columns j , k , and m . If column j is in diagonal block x , column k is in diagonal block y , and column m is in diagonal block z , then diagonal blocks x , y , and z must be applied to row i . In addition, any fill-in elements generated in the off-diagonal border columns by the application of the diagonal blocks must also be eliminated.

If $j < k < m$, then diagonal block x is the first diagonal block which must be applied to the border row. There is no need to apply any diagonal block before block x to the border row because there are no nonzero elements before column j that need to be eliminated. A logical grouping of the border rows, therefore, is to group the border rows together into blocks such that all the border rows in a block have their first nonzero element in a column within the same diagonal block. With respect to the example, border row i would be placed in a border block with all the other border rows that need diagonal block x as the first diagonal block applied to the rows.

Having grouped the border rows into blocks, the parallel mapping onto the Cedar architecture is straightforward. A border block is associated with a particular cluster, a border row with a processor, and the elements of a row with the vector units. As a result, when the border blocks can be grouped such that they contain a number of rows that is a multiple of the processors available in a cluster, or a much larger number, reasonable efficiency is achieved.

This mapping allows multiple border blocks to be updated in parallel, one border block per cluster, and multiple border rows to be updated in parallel, one per processor.

Within a cluster, the rows in the border block are placed in a work queue. A processor in the cluster removes a border row from the queue, updates the row with the appropriate diagonal blocks on the cluster, and then returns the row to the border block. The processor must check a flag for each diagonal block row before it is used to determine if it has been cast and, if it has been cast, proceed to the next update within the border row (this is a simple read, not an indivisible access). If, during the update, a processor decides to cast a diagonal block row instead of updating the border row, it must notify the other processors in the cluster of its decision. Since the border row updates are not synchronized, the other processors in the cluster may have already used the cast row to update some other set of border rows. This results in the nondeterministic application of a diagonal block row before it is cast to the border rows within the cluster. In fact, when using *S-Blocks*, several clusters can be using the same diagonal block to concurrently update multiple border blocks. If a row within an *S-Blocks* is cast, the nondeterministic application of the cast row occurs across multiple border blocks.

Though the applications of the diagonal block rows are not synchronized, the casting of a diagonal block row is synchronized to avoid multiple copies of the row being cast into the border (possibly into different border blocks). Some arbitration strategy is needed to identify which of the potentially many processors that decided to cast the diagonal block row will actually perform the casting operation. This is easily accomplished by making the indication of the intention to cast a row, mentioned above, via an indivisible update of the flag associated with the row. The first processor to succeed in updating the cast flag for the row is responsible for actually casting it into the border block by performing the appropriate data structure updates.

In order to maintain efficiency, the processor that acquires responsibility for casting the row immediately places it into the work queue associated with the current border block being processed by the cluster. This allows any idle processor on the cluster to begin the update of the cast row while the casting processor finishes the update of its border row. As a result, synchronization is only required for removing a border row from the work queue and for placing the row back into the border block after it has been updated by all the appropriate diagonal blocks on the cluster, or when a processor is involved in casting a row. The resulting pivoting/synchronization strategy, when combined with the anticipatory diagonal casting discussed above, preserves the structure of the matrix while considerably reducing the amount and complexity of synchronization as compared to other strategies. For example, pairwise pivoting requires synchronization before each elimination, as the processor acquires the pair of rows needed for a single elimination operation.

Parallelism results. In the previous sections, the experimental results demonstrated that MCSPARSE is competitive in terms of stability and is potentially competitive in terms of fill-in with other parallel solvers that exploit dense matrix techniques when appropriate. In this section, we present the performance of MCSPARSE in terms of parallel execution time on various configurations of the Cedar multiprocessor for the 20 large matrices from the RUA section of the Harwell–Boeing test set used earlier.

Table 9 contains the single-user mode wall-clock time in seconds for three different codes on various configurations of Cedar. The first code is MA28, which does little to

Table 9

Factorization times sorted by the MCSPARSE execution time on one processor

Matrix	MCSPARSE			MA28	Y12M		$T_8^{(y12m)} / T_8^{(mcsp)}$
	$T_1^{(mcsp)}$	$T_8^{(mcsp)}$	$T_{32}^{(mcsp)}$		$T_1^{(y12m)}$	$T_8^{(y12m)}$	
1. mahistlh	3.23	0.74	0.46	3.78	5.59	1.19	1.60
2. sherman1	3.77	0.75	0.55	6.55	4.01	1.16	1.54
3. sherman4	4.11	0.94	0.62	4.97	4.05	1.17	1.24
4. west1505	4.20	1.04	0.75	2.16	3.85	1.10	1.05
5. west2021	6.87	1.69	1.10	2.92	5.49	1.46	0.86
6. pores_2	11.15	2.19	1.34	20.03	17.51	4.53	2.06
7. orsirr_1	13.73	2.74	1.67	26.52	12.43	3.81	1.39
8. nnc1374	19.98	4.18	2.50	28.59	11.68	4.29	1.02
9. gre_1107	20.53	3.81	1.73	29.42	11.47	2.89	0.75
10. gaffl104	24.56	5.47	2.58	56.44	25.62	6.71	1.22
11. hwatt_1	27.78	5.78	2.73	83.87	41.61	11.17	1.93
12. hwatt_2	33.57	6.40	3.08	79.68	41.68	11.15	1.74
13. or678lhs	39.06	7.45	3.02	67.03	130.26	11.87	1.59
14. gemat11	40.32	11.01	4.21	12.79	28.47	6.48	0.58
15. orsreg_1	41.77	9.88	3.19	121.04	67.40	17.05	1.72
16. gemat12	57.12	14.27	5.53	15.19	28.98	6.66	0.46
17. sherman2	78.42	14.49	6.26	565.07	33.82	8.97	0.61
18. saylr4	114.19	23.58	7.79	313.46	193.08	42.45	1.80
19. sherman5	125.69	28.59	9.73	318.37	57.46	12.78	0.44
20. sherman3	127.78	28.53	10.47	191.81	108.71	24.88	0.87

expose exploitable parallelism. It does make use of an initial reordering of the matrix to create a block triangular form which, along with pivot selection to maintain sparsity in the factorization, keeps the number of operations low. For MA28, the table contains the factorization times using $\mu = 0.1$ and $\text{nsrch} = 4$, for the improved pivot search (i.e., at most four rows were searched when selecting a pivot) as compared to the classical MA28 pivot search that could consider all remaining columns and rows. Postprocessing with iterative refinement is used to enhance the accuracy of the MA28 solution. The times listed in the table are for MA28, restructured for parallelism by the Alliant compiler running on all eight processors in a single Cedar cluster. Unfortunately, they are virtually identical to those for MA28 on a single processor. This, of course, is merely a confirmation of the well-known fact that current restructuring technology is not advanced enough to handle such complex code constructs, although promising research continues on the topic. It is therefore important that an efficient eight processor code for a single Cedar cluster is used in the comparison with MCSPARSE.

In order to exploit the parallel and memory resources of the single cluster, it is necessary to alter the pivot search and application procedures as well as to switch to dense techniques and exploit high performance library code designed specifically for one cluster of Cedar. Fortunately, the parallel Y12M code used in the earlier comparisons exploits all of these techniques [19].

The Y12M times were obtained using the version of the code which searches, in parallel, for a set of nearly independent pivots; applies the pivots in parallel; switches to

high performance BLAS3-based dense matrix techniques when appropriate; and uses postprocessing with iterative refinement to enhance the accuracy of the solution. The stability factor for Y12M was 10 (this is basically the same as $\mu = 0.1$ in MA28). The pivot search has various options depending on the number of processors used and the degree of sparsity maintained in the active portion of the matrix (before the switch to dense techniques). The best wall-clock time observed for the different options is listed for each matrix. This code was developed and hand-tuned for parallel execution on one cluster of Cedar and exploits the same linear algebra kernels that are used as the basis for the multicluster dense solver in MCSPARSE. The parallel Y12M represents a reasonably aggressive use of parallelism in a sparse solver for an arbitrary nonsymmetric system on a moderate number of tightly coupled processors. When its techniques are extended to multicluster parallelism, however, the mismatch in parallel granularity implied by the algorithm, and required by the parallel control overhead on Cedar, yields very little speedup for the moderately sized matrices in the test suite. As a result, techniques such as MCSPARSE must be used to extend the speedup due to parallelism across clusters.

The MCSPARSE results are the wall-clock time for the factorization, solve, and iterative refinement phases and were performed using $\mu = 0.1$; the pivot search in the diagonal block was allowed to search all rows in the block if necessary. This speedup in the factorization time comes with a price, however. The time to reorder the matrix for multicluster factorization is nontrivial, and reuse of the ordering on several matrices is typically needed to amortize its cost for the moderate system orders considered here. The four cluster results presented in Table 9 are for the version which uses the hierarchical parallelism described earlier. A comparison with a multicluster MCSPARSE version which does not exploit the hierarchical parallelism is presented later.

The labels in Table 9 are of the form $T_p^{(code)}$, where *code* indicates the program timed and p indicates the number of processors. The hardware configurations represented by different p values are: $p = 1$ one processor execution, using cluster memory; $p = 8$ single cluster (eight tightly coupled processors); and $p = 32$ four clusters of eight processors each.

The timing data illustrates the remarkable fact that MCSPARSE can be a very effective solver on a single processor and a single cluster assuming the cost of the H^* ordering is not considered or that it can be amortized over several factorizations. As expected, the single processor results are a mixed set where each of the three solvers is best for some set of matrices. However, in general, MCSPARSE is the best, or a reasonably close second, in sixteen cases.

More surprising is the fact that even though the structure of the reordered matrix was mainly intended to expose large grain multicluster parallelism, MCSPARSE is fairly competitive with Y12M on eight processors in a single cluster. This can be seen by examining the ratio $T_8^{(y12m)}/T_8^{(mcsp)}$ (i.e., the speedup of MCSPARSE over Y12M on a single cluster given in the table). For all but four of the problems, MCSPARSE is either better than or very close to Y12M (within 25%) in execution time.

The main purpose of developing MCSPARSE was the need for a multicluster direct sparse solver on Cedar. As noted above, the medium and fine grain parallelism, along with the unstructured dynamic pivot search strategies of Y12M, do not extend well beyond one cluster. The effectiveness of the multicluster version of MCSPARSE, which

Table 10

Speedup of MCSPARSE over the one processor version and one cluster version on Cedar

Matrix	$T_1^{(mcsp)} / T_8^{(mcsp)}$	$T_1^{(mcsp)} / T_{32}^{(mcsp)}$	$T_8^{(mcsp)} / T_{32}^{(mcsp)}$
1. mahistlh	4.36	7.02	1.60
2. sherman1	5.02	6.85	1.36
3. sherman4	4.37	6.62	1.51
4. west1505	4.03	5.60	1.38
5. west2021	4.06	6.24	1.53
6. pores_2	5.09	8.32	1.63
7. orsirr_1	5.01	8.22	1.64
8. nnc1374	4.77	7.99	1.67
9. gre_1107	5.38	11.86	2.20
10. gaff1104	4.48	9.51	2.12
11. hwatt_1	4.80	10.17	2.11
12. hwatt_2	5.24	10.89	2.07
13. or678lhs	5.24	12.93	2.46
14. gemat11	3.66	9.57	2.61
15. orsreg_1	4.22	13.09	3.09
16. gemat12	4.00	10.32	2.58
17. sherman2	5.41	12.52	2.31
18. saylr4	4.84	14.65	3.02
19. sherman5	4.39	12.91	2.93
20. sherman3	4.47	12.20	2.72

exploits the hierarchical parallelism described earlier, can be seen from the speedups presented in Table 10. The processor and cluster speedups for MCSPARSE on one and four clusters are presented. These are relative to MCSPARSE on one processor and relative to MCSPARSE on one cluster respectively. The second is the main measure of the success of MCSPARSE given the design goal above.

The speedup for MCSPARSE on one cluster relative to its execution time on one processor is between four and five for all but one problem. This indicates that while reasonable parallel performance has been achieved for both MCSPARSE and Y12M in one cluster, scaling Cedar with more processors in a cluster is probably a losing proposition for the moderately-sized problems of the test suite. This leaves the question of how effective MCSPARSE can be if Cedar is scaled up in the number of clusters (the original design goal).

The second and third columns contain the speedups which address this issue relative to one processor and one cluster respectively. Both show a difference between the first eight to ten smaller problems (in terms of one processor execution time), with the speedups relative to one processor execution illustrating it more strikingly. Within this column, the speedups range from 5.60 to 14.65, but the first ten matrices have an average speedup of 7.82, and only one matrix with a speedup greater than 10. The last ten matrices all have speedups greater than 9.5, with an average of 11.93. The cluster speedups in the third column have similar trends, but indicate the preferred scaling of Cedar. The cluster speedups for the matrices range from 1.36 up to 3.09 out of 4; the first eight matrices, however, all have speedups less than 2 for less than 50% cluster

efficiency, while the last twelve matrices all have speedups greater than 2 with most of them having cluster efficiency near 60% or higher.

The experiments clearly show that MCSPARSE has achieved the goal of extending the performance improvement possible via parallelism when solving a nonsymmetric sparse linear system of moderate size on a cluster-based architecture.

The above results use the version of MCSPARSE which exploits the hierarchical parallelism available in factoring the matrices reordered by H^* . It is possible, however, to use multiple clusters in a non-hierarchically parallel manner in a much less complex version of MCSPARSE. The next set of results demonstrate that the added complexity is worth the effort and, in fact, necessary to achieve reasonable multicluster speedup. To test non-hierarchical parallelism, an experimental version of MCSPARSE was developed which viewed Cedar as a “flat” shared memory machine with 32 processors. This was accomplished by executing the parallel loops across all 32 processors and using the shared global memory in a manner similar to that used for the cluster memory when executing the hierarchical version of MCSPARSE on a single cluster.

Fig. 8 contains a graph which shows the four cluster speedups for the hierarchical code and the flat code. The times $T_8^{(mcsp)}$, $T_{32}^{(mcsp)}$, and $T_{32}^{(flat)}$ are all factorization and solve times (i.e., the time to perform iterative refinement has been removed from both codes).

The results clearly show that just using H^* to expose the bordered block triangular form is not enough. Hierarchical parallelism exploitation must be designed into the code. The cluster speedup for the hierarchical code ranges from 1.5 to 3.1, with a mean speedup of 2.2, while the flat code cluster speedup ranges from 0.7 to 2.2, with a mean

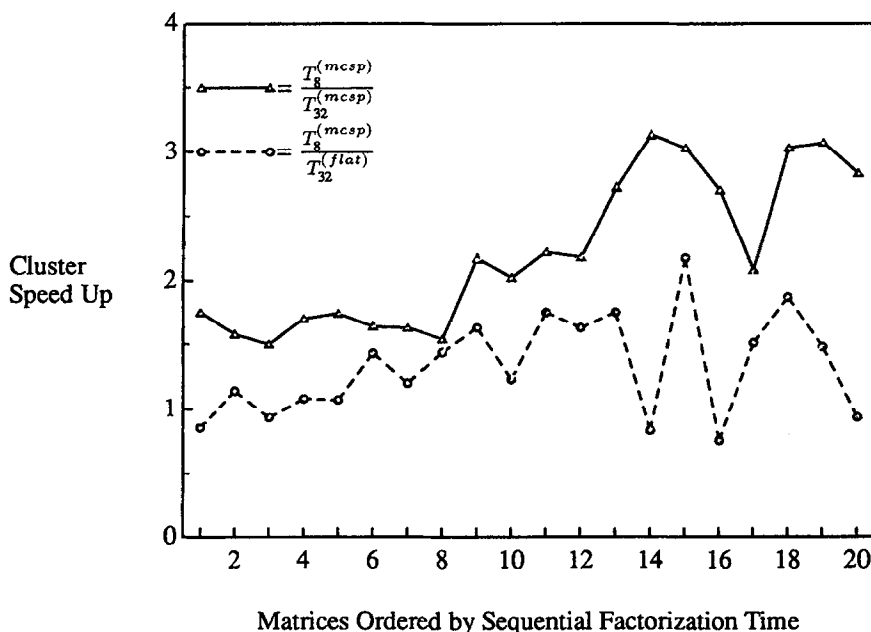


Fig. 8. Comparison of hierarchical and flat parallelism.

Table 11

Performance comparison of the sequential and parallel H^* ordering

Matrix	$T^{(H^*)}$	$T_8^{(H^*)}$
1. mahistlh	1.45	0.68
2. sherman1	0.75	0.50
3. sherman4	0.69	0.46
4. west1505	2.47	0.58
5. west2021	5.89	0.85
6. pores_2	2.53	1.21
7. orsirr_1	1.22	0.76
8. nnc1374	3.26	1.21
9. gre_1107	3.65	1.72
10. gaff1104	2.14	1.25
11. hwatt_1	2.31	1.46
12. hwatt_2	2.23	1.49
13. or678lhs	7.20	3.88
14. gemat11	4.04	2.87
15. orsreg_1	3.07	2.13
16. gemat12	4.50	3.33
17. sherman2	5.40	2.59
18. saylr4	5.15	3.70
19. sherman5	4.70	2.36
20. sherman3	4.09	3.00

speedup of 1.3. In fact, the flat code on 32 processors is slower than the 8 processor code for five of the twenty matrices.

All of the discussion above assumed that the time required to run H^* to reorder the matrices was amortized over solving several systems. Table 11 contains the execution time for H^* on one processor of Cedar. Clearly, if this sequential reordering time is not assumed to be amortized, serious degradation in the ordering/factorization/solution can occur. One approach to improve this performance is to investigate the use of parallel versions of the algorithms. This can be done with various levels of complexity. A simple approach is to utilize the loop-based parallelism available in the algorithms described above. For instance, instead of trying one bound at a time in H0, a different bound can be tried on each processor in parallel. Within the H1 and H2 phases, the initialization of data structures, the updating of data structures, and the searches that occur can be done in parallel. Table 11 also contains the one cluster execution time, $T_8^{(H^*)}$, when utilizing the loop-based parallelism within H^* . Some moderate improvements are seen. Note that parallelizing H^* can change the resulting ordering of the matrices. For most matrices in the test suite, the change in the ordering has little effect on the performance of the solver, usually less than 10%. For the two matrices with larger degradation in time, the size of the border in the parallel H^* matrix is significantly larger than that from the sequential H^* . For one matrix, the factorization time decreases due to a smaller border from the parallel H^* and a reduction in the amount of casting.

The main problem with a parallel H^* is the depth-first search nature of the Tarjan and H1 phases. Wijshoff and Geschiere have considered more aggressively parallel forms of the phases of H^* , but a discussion of their results is beyond the scope of this

paper; the interested reader is referred to [28,29]. It is also possible to use a faster, but usually less successful, reordering strategy which results in a matrix that can be factored by a different parallel algorithm, or whose form can be made consistent with the bordered block triangular form assumed by MCSPARSE [20].

9. Conclusions

A new ordering technique H^* and an associated parallel factorization algorithm, MCSPARSE, for solving sparse nonsymmetric linear systems on a multicluster architecture have been presented. The design tradeoffs on the experimental Cedar system have been considered and the resulting performance demonstrated via experiments on the system for a suite of moderately-sized nonsymmetric systems from the Harwell-Boeing matrix collection.

The H^* ordering combines four different orderings to transform a matrix into bordered block upper triangular form: H_0 , Tarjan's algorithm for finding strongly connected components, H_1 , and H_2 . Except for H_0 , these orderings are symmetric, which distinguishes H^* from other tearing techniques. The effectiveness of the H^* ordering, in terms of producing small borders and improving the stability of the factorization, has been demonstrated.

The issues of stability and sparsity in MCSPARSE have been addressed successfully. For stability, the H^* ordering has been combined with a pivoting technique, casting, based on the delay of pivoting decisions in such a way that: the application of unstable pivots during the factorization is avoided; the overall structure of the matrix is preserved; complex synchronization policies are avoided; and multilevel parallelism is exploited. The restriction on the number of pivot elements per column results in a factorization that is LU-like but much simpler and possessing a worst-case growth factor less than the factorization and growth factors associated with a pairwise or blockwise pivoting strategy often used in parallel nonsymmetric sparse solvers. The use of this method, in conjunction with the application of iterative refinement, allows MCSPARSE to obtain stable factorizations comparable to standard robust factorization routines, such as that employed in MA28, used on sequential processors.

To improve sparsity while maintaining large grain parallelism, a modified version of the Markowitz count was developed which includes the fill-in generated inside the diagonal block and estimates the fill-in to be generated outside the diagonal block. Though generating more fill-in elements than the sequential methods, the improved performance of the large grain parallelism combined with an implicit switch to dense methods allows MCSPARSE to effectively use the parallel resources available in single and multiple cluster execution modes on Cedar.

Both static and dynamic methods of load balancing were implemented to improve the performance of MCSPARSE on Cedar. Reblocking is combined with multiple work counts and static partitioning to determine the initial allocation of diagonal blocks to clusters. Within a cluster, the diagonal blocks are evaluated to determine if a processor or a cluster should be used to factor the diagonal block. *S-Blocks* are used within the update of the border blocks to dynamically balance the work. While combinations of these methods were tested on the Cedar system, they are adaptable to other parallel systems.

The use of hierarchical parallelism was tested and compared to a flat multicluster solver for the matrices reordered by H^* . The version of MCSPARSE which exploits hierarchical parallelism was found to deliver reasonable performance improvements as the number of clusters increased, even for the moderately-sized problems in the test suite.

There are several other avenues of investigation left to pursue with respect to MCSPARSE. As noted earlier, a parallel implementation of the H^* ordering would improve the overall performance of the solver by mitigating the need to assume the ability to amortize the cost of H^* over several problems with similar structure. The code is adaptable to multivector and multicluster processors other than the single experimental Cedar system, and several potential improvements are possible. The most important is more fully exploiting the hierarchical structure of the border. The Cedar version of MCSPARSE essentially treats the border as a collection of submatrices at the same level. It is clear, however, from the H^* ordering, that there is a hierarchical relationship between the portions of the border produced at different stages of the algorithm.

At some point, the performance improvement due to the multilevel parallelism will decline and a hybrid strategy of direct and iterative methods similar to that used in a code based on the parallel Y12M code used in the experiments here should be considered [17,18]. Initial results, [46], indicate that MCSPARSE can be adapted to use a combination of positional dropping (i.e., ignoring a fill-in element due to its position in the matrix) and numerical dropping (i.e., ignoring a fill-in element because of its relative magnitude) to produce a preconditioner for iterative methods for nonsymmetric systems.

References

- [1] G. Alagband, Parallel pivoting combined with parallel reduction and fill-in control, *Parallel Computing* 11 (1989) 201–221.
- [2] M. Arioli and I.S. Duff, Experiments in tearing large sparse systems, in: M.G. Cox and S. Hammarling, eds., *Reliable Numerical Computation* (Oxford University Press, New York, 1990) 207–226.
- [3] M. Arioli, I.S. Duff, N.I.M. Gould and J.K. Reid, Use of the P^4 and P^5 algorithms for in-core factorization of sparse matrices, *SIAM J. Sci. Statist. Comput.* 11 (5) (1990) 913–927.
- [4] C.C. Ashcraft, R.G. Grimes, J.G. Lewis, B.W. Peyton and H.D. Simon, Progress in sparse matrix methods for large linear systems on vector supercomputers, *Internat. J. Supercomputing Appl.* 1 (4) (1987) 10–30.
- [5] J.R. Bunch, Analysis of sparse elimination, *SIAM J. Numer. Anal.* 11 (5) (1974) 847–873.
- [6] L.K. Cheung and E.S. Kuh, The bordered triangular matrix and minimum essential sets of a digraph, *IEEE Trans. Circuits Systems* 21 (5) (1974) 633–639.
- [7] T.A. Davis, Users' guide for the unsymmetric pattern multifrontal package (UMFPACK, Version 1.1), Tech. Rept. TR-95-004, CISE Department, University of Florida, 1995.
- [8] T.A. Davis and E.S. Davidson, Pairwise reduction for the direct, parallel solution of sparse unsymmetric sets of linear equations, *IEEE Trans. Comput.* 37 (12) (1988) 1648–1654.
- [9] T.A. Davis and P.C. Yew, A nondeterministic parallel algorithm for general unsymmetric sparse LU factorization, *SIAM J. Matrix Anal. Appl.* 11 (3) (1990) 383–402.
- [10] I.S. Duff, Algorithm 575, Permutations for a zero-free diagonal, *ACM Trans. Math. Software* 7 (3) (1981) 387–390.
- [11] I.S. Duff, On algorithms for obtaining a maximum transversal, *ACM Trans. Math. Software* 7 (3) (1981) 315–330.

- [12] I.S. Duff, Parallel implementation of multifrontal schemes, *Parallel Computing* 3 (1986) 193–204.
- [13] I.S. Duff and J.K. Reid, The multifrontal solution of indefinite sparse symmetric linear equations, *ACM Trans. Math. Software* 9 (1983) 302–325.
- [14] A.M. Erisman, R.G. Grimes, J.G. Lewis and W.G. Poole, A structurally stable modification of Hellerman–Rarick's P^4 algorithm for reordering unsymmetric sparse matrices, *SIAM J. Numer. Anal.* 22 (2) (1985) 369–385.
- [15] A.M. Erisman, R.G. Grimes, J.G. Lewis, W.G. Poole and H.D. Simon, Evaluation of orderings for unsymmetric sparse matrices, *SIAM J. Sci. Statist. Comput.* 8 (4) (1987) 600–624.
- [16] K. Gallivan, B. Marsolf and H. Wijshoff, A large-grain parallel sparse system solver, in: *Proc. 4th SIAM Conf. on Parallel Proc. for Scient. Comp.* Chicago, IL, (1989) 23–28.
- [17] K. Gallivan, A. Sameh and Z. Zlatev, Parallel hybrid sparse linear system solver, *Computing Systems in Engineering* 1 (1990) 183–195.
- [18] K. Gallivan, A. Sameh and Z. Zlatev, Solving general sparse linear systems using conjugate gradient-type methods, in: *Proc. 1990 Internat. Conf. on Supercomputing*, Amsterdam, The Netherlands (ACM Press, New York, 1990) 132–139.
- [19] K. Gallivan, A. Sameh and Z. Zlatev, Parallel direct method codes for general sparse matrices, in: Spedicato, Bertocchi and Vespucci, eds., *Proc. NATO ASI on Linear Systems*, 1991.
- [20] K.A. Gallivan, P.C. Hansen, T. Ostronsky and Z. Zlatev, A locally optimized reordering algorithm and its application to a parallel sparse linear system solver, *Computing* 54 (1) 39–67.
- [21] K.A. Gallivan, B.A. Marsolf and H.A.G. Wijshoff, MCSPARSE: A parallel sparse unsymmetric linear system solver. Tech. Rept. CSRD Report No. 1142, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, 1991.
- [22] K.A. Gallivan, B.A. Marsolf and H.A.G. Wijshoff, The parallel solution of nonsymmetric sparse linear systems using the H^* reordering and an associated factorization, in: *Proc. 8th ACM Internat. Conf. on Supercomputing* Manchester, England (1994) 419–430.
- [23] C.W. Gear, Numerical errors in sparse linear equations, Tech. Rept. UIUDCS-F-75-885, Department of Computer Science, University of Illinois, Urbana, IL, 1975.
- [24] A. George, Nested dissection of a regular finite element mesh, *SIAM J. Numer. Anal.* 10 (2) (1973) 345–363.
- [25] A. George, An automatic one-way dissection algorithm for irregular finite element problems, *SIAM J. Numer. Anal.* 17 (6) (1980) 740–751.
- [26] A. George and J.W.H. Liu, An automatic nested dissection algorithm for irregular finite-element problems, *SIAM J. Numer. Anal.* 15 (1978) 1053–1069.
- [27] A. George and J.W. Liu, *Computer Solution of Large Sparse Positive Definite Systems* (Prentice Hall, Englewood Cliffs, NJ, 1981).
- [28] J.P. Geschiere, Research on parallelizing the reordering phase of MCSPARSE, a large grain parallel sparse unsymmetric linear system solver, Master Thesis, Tech. Rept. INF/SCR-92-23. Department of Computer Science, Utrecht University, 1992.
- [29] J.P. Geschiere and H.A.G. Wijshoff, Exploiting large grain parallelism in a sparse direct linear system solver, Tech. Rept. 93-18, High Performance Computing Division, Leiden University, 1993.
- [30] F.G. Gustavson, Finding the block lower triangular form of a matrix, in: J.R. Bunch and D.J. Rose, eds., *Sparse Matrix Computations* (Academic Press, New York, 1976).
- [31] S.M. Hadfield and T.A. Davis, A distributed memory, multifrontal method for sequences of unsymmetric pattern matrices, in: *Proc. 1995 Internat. Conf. on Parallel Processing, Volume 3: Algorithms and Applications* (1995) 42–45.
- [32] M. Hall Jr, An algorithm for distinct representatives, *Amer. Math. Monthly* 63 (10) (1956) 716–717.
- [33] E. Hellerman and D.C. Rarick, The partitioned preassigned pivot procedure (P^4), in: D.J. Rose and R.A. Willoughby, eds., *Sparse Matrices and their Applications* (Plenum, New York, 1972).
- [34] J.E. Hopcroft and R.M. Karp, An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs, *SIAM J. Comput.* 2 (4) (1973) 225–231.
- [35] H.W. Kuhn, The Hungarian method for the assignment problem, *Naval Research Logistics Quarterly* 2 (1) (1955) 83–97.
- [36] C.E. Leiserson and J.G. Lewis, Orderings for parallel sparse symmetric factorization, in: *Proc. 3rd SIAM Conf. on Parallel Proc. for Scient. Comp.*, Los Angeles, CA (1987) 27–31.

- [37] T.D. Lin and R.S.H. Mah, Hierarchical partition – A new optimal pivoting algorithm, *Mathematical Programming* 12:260–278, 1977.
- [38] R.J. Lipton, D.J. Rose and R.E. Tarjan, Generalized nested dissection, *SIAM J. Numer. Anal.* 16 (1979) 346–358.
- [39] J.W.H. Liu, The multifrontal method for sparse matrix solution: Theory and practice, *SIAM Review* 34 (1) (1992) 82–109.
- [40] B. Marsolf, Large grain parallel sparse system solver, Master Thesis, Tech. Rept. CSRD Report No. 1125, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, 1991.
- [41] A. Sangiovanni-Vincentelli, An optimization problem arising from tearing methods, in: J.R. Bunch and D.J. Rose, eds., *Sparse Matrix Computations* (Academic Press, New York, 1976) 97–110.
- [42] Staff, The Cedar project, Tech. Rept. CSRD Report No. 1122, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, 1991.
- [43] D.V. Steward, Partitioning and tearing systems of equations, *SIAM J. Numer. Anal.* 2 (2) (1965) 345–365.
- [44] R.E. Tarjan, Depth-first search and linear graph algorithms, *SIAM J. Comput.* 1 (1972) 146–160.
- [45] A.F. van der Stappen, R.H. Bisseling and J.G.G. van de Vorst, Parallel sparse LU decomposition on a mesh network of transputers, *SIAM J. Matrix Anal. Appl.* 14 (1993) 853–879.
- [46] X. Wang, private communication.
- [47] H.A.G. Wijshoff, Symmetric orderings for unsymmetric sparse matrices, Tech. Rept. CSRD Report No. 901, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, 1989.
- [48] U. Meier Yang, Preconditioned iterative solver for nonsymmetric linear systems, Ph.D. Thesis, University of Illinois at Urbana-Champaign, 1994.