

The parallel solution of nonsymmetric sparse linear systems using the H^* reordering and an associated factorization.

K. A. Gallivan* B. A. Marsolf*
Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign
Urbana, Illinois
61801

H. A. G. Wijshoff†
High Performance Computing Division
Department of Computer Science
Leiden University
Leiden, The Netherlands

Abstract

In this paper, a nonsymmetric sparse linear system solver based on the exploitation of multilevel parallelism is proposed. One of the main issues addressed is the application of tearing techniques to enhance large grain parallelism in a manner that maintains reasonable stability. This is accomplished by a combination of a novel reordering technique (H^*) and pivoting strategy. The large grain parallelism exposed by the reordering is combined with medium (various parallel row updates strategies) and fine grain (vectorization) parallelism to allow adaptation to a wide range of multiprocessor architectures. Experimental results are presented which show the effectiveness of the reordering, as well as the stability and efficiency of the solver.

1 Introduction

Several techniques have been proposed to solve large sparse systems of linear equations on parallel processors. 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. Many algorithms assume special properties such as symmetric positive definiteness or exploit knowledge of the application from which the system arises, e.g., finite element problems. These properties can be exploited in the a priori identification of parallelism, preservation of sparsity and guaranteeing stability. These decisions can be done statically before the factorization is performed, e.g., the symbolic factorization techniques and orderings of many direct solvers for positive definite systems.

In many applications, such as device simulation, computational fluid dynamics, circuit simulation, and structural mechanics, the values in the resulting linear systems are not

symmetric, though the structure of the system is symmetric. In other application areas, such as linear programming, optimization problems, directed network problems, and simulation problems, the resulting linear systems are even nonsymmetric in structure. For these arbitrary nonsymmetric systems the exploitation of parallelism while maintaining stability and sparsity becomes extremely difficult. This is due to the fact that the requirements are often contradictory and cannot be totally resolved until information from the factorization is available, i.e., some decisions must take place dynamically. As a result, for nonsymmetric systems on a range of parallel architectures it is often necessary to carefully mix a priori static and dynamic runtime decisions.

One approach that has been tried for parallel sparse system solvers is the multi-frontal scheme [11, 13]. A multi-frontal scheme constructs an elimination tree to organize the parallel work. A node in the tree represents a certain computation, which may include handling the information from the node's children and performing some pivot eliminations. All leaf nodes of the tree may be computed in parallel, while internal nodes can only be computed after their children have completed. A pool of the available work, the nodes in the tree that can be computed, is maintained in shared memory. When any process needs work it retrieves a node from the pool. After all the children of a node have finished, the parent node is then placed in the pool of available work. This approach if organized correctly can provide large and medium grain parallelism. However, the method tends to work well on matrices with a near-symmetric structure and the pivot sequence is constrained.

Another approach to parallel sparse solvers exploits the dynamic identification and application of parallel pivots [1, 7, 19]. At each stage these algorithms construct a set of pivots that can be applied in parallel and perform the appropriate updates. 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 synchronization. There is also previous work on performance improvements of direct sparse solvers on vector supercomputers [5]. 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.

An important part of any sparse solver is the algorithm for controlling the amount of fill-in that is generated. Most sequential sparse matrix packages and, in particular, MA28, use a strategy which is based on technique proposed by Markowitz [34]. This strategy involves counting the number of nonzero elements in each column, c_j , and the num-

*Supported by the National Science Foundation under Grant No. US NSF CCR-9120105, by the Department of Energy under Grant No. DE-FG02-85ER25001 and by ARPA under a subcontract from the University of Minnesota of Grant No. ARPA/NIST 60NANB2D1272.

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

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association of Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

ber of nonzero elements in each row, r_i , and then choosing the pivot node to be the element $a_{i,j}$ where the product $(c_j - 1) * (r_i - 1)$ is the minimum over all possible pivot candidates. Various modified forms of this strategy that limit the number of elements considered are possible.

The final aspect of pivot selection is the maintenance of stability. Typically, this is done by choosing a pivot element that is within a specified multiple of the largest element in the pivot row or pivot column or the active part of the matrix depending on the efficiency of these tests given the data structures assumed. (A discussion of stability can be found in [12].)

The stability and sparsity tests for pivot selection are often contradictory and most strategies involve some combination of the two, e.g., the generalized Markowitz strategy, [35]. Parallel solvers add a third constraint to pivot selection. For the medium and fine grain algorithms mentioned above, these three constraints can be considered in a reasonably straightforward way potentially with respect to the entire active portion of the matrix. The exploitation of larger grain parallelism, however, often imposes a static decomposition on the structure of the matrix which further constrains pivot selection.

The effect of these constraints, for nonsymmetric problems, can be seen by considering tearing techniques. These have been proposed to expose large-grain structure and parallelism by reordering the matrix into a bordered block triangular matrix [15, 27]. 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. For example, considering the entire active portion of the matrix during a pivot search can easily destroy the block structure. On the other hand, limiting the search to a particular block, which can reduce the fill-in within that block, can increase the fill-in for the overall matrix and reduce the accuracy of the solution.

The approach taken in this paper uses a novel ordering technique, H^* , to identify a priori large and medium grain parallelism by creating a bordered upper triangular structure and a factorization routine which preserves this structure while attempting to maintain stability and sparsity at acceptable levels. A technique referred to as *casting* is used to control the stability of the factorization. The large and medium grain parallelism (parallel subsystems of various sizes) exposed by H^* is combined with medium (various parallel row updates strategies) and fine grain (vectorization) parallelism to form a multi-grain parallel solver which allows adaptation to a wide range of multiprocessor architectures. A multi-cluster version of the solver, MCSPARSE, has been implemented and analyzed on the Cedar system [37]. Initial results with MCSPARSE were presented in [16] and more details of the implementation and its tuning can be found in [20].

The paper is organized as follows. In Section 2 a comparison between our approach and other methods is presented. The details of the ordering H^* are presented in Section 3. Casting is introduced in Section 4. In Section 5 an overview of the procedures in MCSPARSE is given. Experimental results and conclusions are given in Sections 6 and 7.

2 Comparison of Different Approaches

During the introduction, the H^* ordering for transforming a matrix into bordered block upper triangular form was de-

scribed as novel. This is not to say the use of the bordered triangular form or the bordered block triangular form for solving sparse nonsymmetric systems is a new idea.

Research into orderings for transforming matrices into the bordered triangular form has been done using graph theory methods for finding the minimal essential set [6, 36]. These methods rely on the fact that the sparse system is positive definite, so that diagonal pivots can be used without deteriorating the stability of the solution method. In 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 as it allows pivot selection within the diagonal blocks without destroying the overall structure of the system. Several different methods have been proposed for finding the bordered block triangular form. Partitioning and tearing methods [38] can be used, and algorithms such as P^4 [27], P^5 [14], the Hierarchical Partition by Lin and Mah [31], 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 for transforming the system into the bordered block triangular form, the associated factorization phases lacked stability and, therefore, are not recommended to be used for 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 taken 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 presented later. The main difference, however, is that in the level set algorithm the transversal is not constrained to contain relatively large elements, but just nonzero 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 resulting structure of the system might not have as small of diagonal block and border block sizes. This can be observed in the results of the level set algorithm. 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 is shown below the complementary nature of the phases results in a significant increase in the power of the symmetric permutations.

In the factorization phase, provisions have to be taken to guarantee a reasonably stable solution method. P^5 , the Hierarchical Method, and the level set algorithm guarantee structurally non-singular blocks. However, these methods are still potentially unstable. Iterative refinement could be used to improve the stability of these methods, 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 reduced significantly. This approach enables MCSPARSE to be

competitive with other direct solvers, (see Section 6).

Descriptions of the actual algorithms used within the H* algorithm are presented within the next section. A preliminary algorithmic description of the H* ordering is in [41].

3 The Hybrid 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.

Definition 3.2 *An ordering of a sparse matrix is called nonsymmetric if the ordering can be represented by*

$$\tilde{A} = PAQ^T,$$

with P and Q permutation matrices. If $P = Q$ the ordering is called symmetric.

Note that symmetric orderings have the property that the associated graphs of A and \tilde{A} are isomorphic, i.e., only the numbering of the nodes differs. Nonsymmetric orderings are obtained by independent row and column interchanges of the matrix represented by P and Q respectively. So, where the nonsymmetric orderings change certain properties of the sparse matrix, e.g., eigenvalues and diagonal dominance, symmetric orderings maintain these. The 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 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 3.3 *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, H0, is a nonsymmetric ordering which permutes the largest elements available at each decision point of the production of the transversal onto the diagonal. The second phase consists of applying Tarjan's algorithm to transform the matrix into triangular block form. The third phase, H1, is applied to each diagonal block produced by Tarjan's algorithm that are considered too large. H1 attempts to change each of these blocks into bordered block triangular form via a modified Tarjan's algorithm. H2, the last phase,

is also only applied to the large diagonal blocks remaining in the matrix to change them into bordered block triangular form via a modified dissection algorithm. The last three phases, Tarjan's, H1, and H2 are all symmetric orderings.

3.2 H0

H0 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 order to enhance the stability of the subsequent factorization.

The transversal ordering is a matching between the columns and the diagonal positions of the matrix and can be found using many different algorithms. Algorithms for finding set representation [33] or solutions to the assignment problem [29] could be used. An alternative algorithm involves finding maximal matchings in bipartite graphs [28].

H0 is based on work of Duff and Gustavson [9, 10, 26]. 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 each step j , the algorithm has a transversal for columns 1 through $j-1$ and tries 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 currently not 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 column 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, 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 only permuting an element $a_{i,j}$ to the diagonal if its value is within a bound, α , of the largest element in the column, i.e.,

$$|a_{i,j}| * \alpha \geq \max_k (|a_{k,j}|) \quad (1)$$

Only a few changes to the transversal algorithm are required to support the 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 only be selected if it meets the bound of Equation 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 α and tries to find a transversal. If a satisfactory bounded transversal

cannot be found, then an estimate of what bound is necessary is made by examining the columns where the current bound failed. The bound is then set to this estimate and the algorithm is restarted. If a bound greater than a preset limit is tried and a transversal is still not found, then the bound is eliminated totally 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 H0 algorithm relies upon the ability to quickly find an adequate bound for the transversal.

3.3 Tarjan's Algorithm

Tarjan's algorithm [39] finds the strongly connected components of the digraph associated with the matrix with time complexity linear in the number of nodes and edges.¹ A renumbering of the nodes of the digraph corresponding to the decomposition of the graph into strongly connected components yields a symmetric ordering which 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 active nodes. 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 that indicates 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.

When all of the edges from the current node have been processed, then the algorithm checks to determine if a strongly connected component has been found by examining the position of the current node. 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.

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

3.4 H1 Algorithm

A problem with most sparse matrices is that they do not allow a nice decomposition into strongly connected components and, therefore, Tarjan's algorithm, by itself, will not provide a suitable decomposition. A typical case is a matrix

¹ A description of this algorithm is included in this paper so that the modifications on which H1 relies can be discussed properly.

whose associated digraph contains a large cycle. The third phase of H*, the H1 algorithm, addresses this problem. It is based on Tarjan's algorithm and extracts from the digraph 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 potentially 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. The H1 algorithm is applied to each diagonal block resulting from Tarjan's algorithm that is larger than a threshold, *T_{done}*. Each diagonal block is separated, when possible, into two or more smaller blocks and a quasi-separator set. The union of these quasi-separators are placed in the border for the entire matrix.

The H1 algorithm 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 node lowest on the stack that can be reached from *x* by a single edge. The second, denoted *mlow_x*, is a pointer to the position of lowest node on the stack that can be reached by a single edge from any of the nodes higher on the stack than *x*. 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 some threshold, *T_{done}*, the H1 algorithm 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*.

Throughout the algorithm, whenever an edge to a node *y* is encountered such that *pos(y) - mlow_{current} ≥ 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 to 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 for consideration later in the algorithm as a node to be moved 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 that points 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:

$$\begin{aligned} low_{current} &= \min(low_{current}, low_y), \\ nlow_{current} &= \min(nlow_{current}, position(y)), \end{aligned}$$

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
     $mlow_{current} = \min(mlow_{current}, nlow_{current})$ 
end if
 $mlow_{prev} = \min(mlow_{prev}, mlow_{current})$ 
 $low_{prev} = \min(low_{prev}, low_{current})$ 

```

Note that the decision of whether or not 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 or not the node is moved to the quasi-separator.

After these updates the decision is made as to whether: no action is required, 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 in the new block that have been marked as having a long edge. The bordered block is only accepted 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.

An example of how the H1 algorithm finds a quasi-separator set can be found by the application of the H1 algorithm to the 8×8 sparse matrix in Figure 1. The associated directed

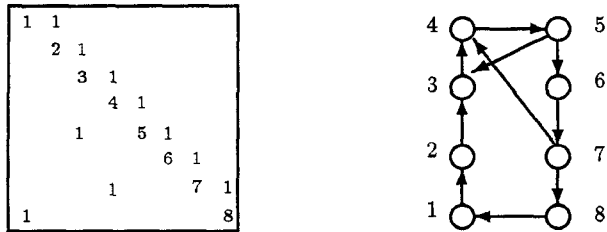


Figure 1: A 8×8 sparse matrix and its associated digraph

graph for the 8×8 matrix is also included in this figure.

Figure 2 is the current state of the algorithm when it has just completed all the edges from node 5. The current block of completed nodes contains nodes 5, 6, 7, and 8. There are

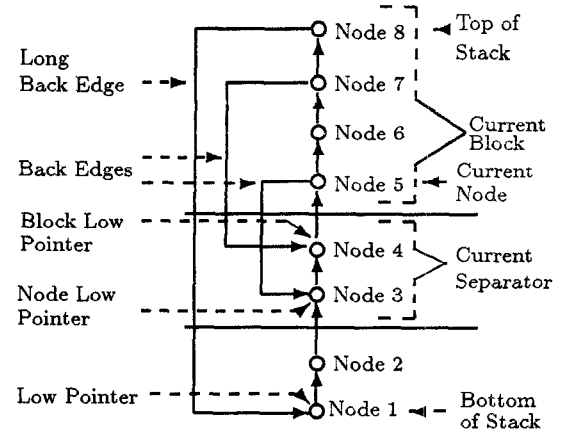


Figure 2: H1 Stack

three back edges from the nodes in the block, these are the edges $\{8,1\}$, $\{7,4\}$, and $\{5,3\}$. The back edge $\{8,1\}$ however was determined to be a *long* edge and it is not included in determining the size of the quasi-separator set. Therefore, for node 5 the one edge low pointer for the node points to node 3 ($nlow_5 = 3$); And the one edge low pointer for the nodes above node 5 points to node 4 ($mlow_5 = 4$). This yields an initial bordered block size of 4, a quasi-separator size of 2, and a remaining block size of 2.

Assuming the block sizes meet the necessary constraints, the search for the *long* back edges is made. This search finds the edge $\{8,1\}$ and places node 8 in the quasi-separator set. The bordered block size becomes 3, the quasi-separator set becomes 3, and the remaining block stays at 2. The bordered block contains nodes 5, 6, and 7. The quasi-separator block contains the nodes 3, 4, and 8. Next, H1 is applied to the remaining nodes which results in the two independent blocks 1 and 2. The matrix that H1 produces is shown in Figure 3.

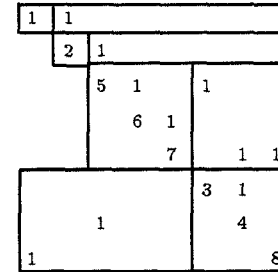


Figure 3: Reordered Matrix

3.5 H2 Algorithm

The H1 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 of transforming the matrix to block upper triangular form starting from the standard techniques used to produce separator sets for structurally symmetric matrices, e.g., nested dissection [22, 25].

As in the standard approaches, the ordering H2 starts with the construction of separator sets of the adjacency ma-

trix of $A + A^T$. In our implementation of H2 we used a straight-forward implementation of automatic nested dissection [24]. However, other initial orderings could have been used such as one-way dissection [23], more sophisticated implementations of automatic nested dissection [32], or the graph bisection heuristics proposed in [30].

The H2 algorithm is only applied to diagonal blocks produced by H1 that are greater than a user-specified threshold, T_{done} . The algorithm starts with the graph $(G = (V, E))$ associated with the unscaled symmetric part of the diagonal block under consideration, $M = (A + A^T)$, with the self-edges generated by the diagonal elements removed. 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 , where β is usually 10% of the rows in the original matrix, the node is placed into the border and removed from further consideration. A limit is placed on the number of nodes that will be placed in the border from any particular diagonal block by using this test. In our implementation, this limit is usually 7% of the nodes in the diagonal block. Our experience with the RUA matrices has shown the values of 10% and 7% to provide reasonable performance.

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 only 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.

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 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 $(y, x) \in E$ such that $y \in S$ and $x \in B$ then y may be moved to C .
2. If there exists no edge $(z, y) \in E$ 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 Figure 4. Since there is no edge from any node in C directed to the node d in S , then d may be moved into B . The node e may not be removed from S since it does not meet the requirements for either of the reductions and moving out of S would destroy the desired structure.

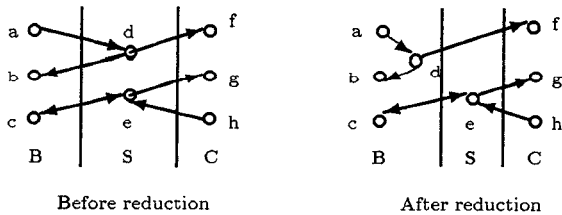


Figure 4: Reduction of the Separator Set

An optimization to the reduction above involves moving nodes from B to C , or C to B , so that the first two reduc-

tions 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 $(d, b) \in E$ where $d \in D$ and $b \in B$.
2. There exists $R \subseteq S$ such that there are edges $(y, d) \in E$ where $d \in D$ and $y \in R$; and there are no edges $(y, b) \in E$ 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 separator set are repeated.

Symmetric conditions can be defined that 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 $(c, d) \in E$ where $d \in D$ and $c \in C$.
2. There exists $R \subseteq S$ such that there are edges $(d, y) \in E$ where $d \in D$ and $y \in R$; and there are no edges $(c, y) \in E$ 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, $|C - D| > T_{remain}$.

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 Figure 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 .

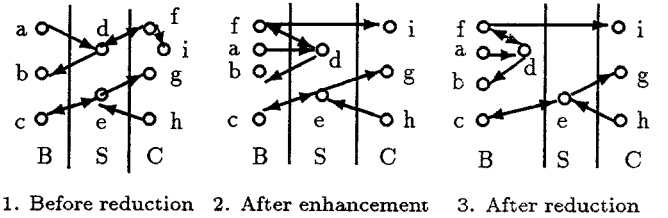


Figure 5: Enhanced Separator Set Reduction

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 [20] for many more details concerning the tuning of the heuristics that produces the data presented below.

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 only be presented for fourteen of the matrices which have at least 1,000 rows.

Table 1 contains the results for the application of the H^* ordering to the large matrices. This table contains the transversal bound which was found by H^* , the total time for the H^* ordering (user process time in seconds), the total number of diagonal blocks after the ordering, the number of rows in the border, and the largest diagonal block.

The transversal bound is a scalar α such that the maximum value in a column is not more than α times the corresponding diagonal element, $|a_{i,j}| \leq \alpha \max_{1 \leq j \leq n} |a_{j,j}|$ for $1 \leq i \leq n$. When the tests were run, the initial value of α was $1E+1$. If this value of α was not adequate the H^* algorithm would estimate what α should be by looking at the columns where the current bound failed. The value of α would be increased to the estimate and the algorithm would retry. This would continue until either a transversal had been found or until a bound greater than $1E+5$ had been tried. If a bound greater than $1E+5$ was tried without finding a transversal, then the transversal was then tried without the bound. For some of the matrices the transversal bound is given as '*', this indicates that the H^* algorithm could not find a bounded transversal within the given limits and an unbounded transversal search was used.

Matrix Name	Rows	Non-zeroes	H^* Bound	Total Time	Total Blocks	Border Rows	Max Block
gafl104	1104	16056	$1E+4$	2.066	190	202	108
gemat11	4929	33185	$1E+2$	3.915	437	348	404
gre_1107	1107	5664	$1E+1$	3.494	23	324	103
hwatt2	1856	11550	$1E+8$	2.136	142	430	158
mahistlh	1258	7682	*	1.379	930	74	124
nnc1374	1374	8606	$1E+9$	3.126	91	244	130
or678lhs	2529	90158	$1E+6$	6.962	2000	355	170
orsreg_1	2205	14133	$1E+1$	2.930	15	438	160
pores_2	1224	9613	$1E+5$	2.409	21	245	105
saylr4	3564	22316	$1E+1$	4.966	22	634	333
sherman2	1080	23094	$1E+7$	5.179	220	352	102
sherman3	5005	20033	$1E+1$	3.942	2119	423	394
sherman5	3312	20793	$1E+6$	4.483	1680	303	310
west2021	2021	7353	$1E+6$	5.670	1261	93	188

Table 1: H^* Statistics for Large RUA Matrices

The results of H^* can be compared with the related orderings produced by the P^4 algorithm [27], the P^5 algorithm [14], and the level set algorithm by Arioli and Duff [2], on a subset of the Harwell-Boeing matrices for which the results are available in the literature [4, 2]. This subset comprises the Grenoble matrices and the Westerberg's matrices. The matrices range in order from 67 to 2021.

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. 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.

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

Clearly, the P^4 and P^5 orderings produce smaller bor-

Matrix	Order	Border Size				Largest Block		
		H^*	P^4	P^5	Level Set	H^*	P^4	Level Set
gre_1115	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

Table 2: Number of Rows in Border & Largest Diagonal Block Size

ders 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 accuracy [4].

The difficulties in the coupling of P^4 and P^5 and 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.

4 Stability Issues

4.1 General considerations

The major problem with a large grain parallel solver is maintaining the stability of the factorization while only working with pivot selection constrained to a particular subsystem, e.g., a diagonal block or border block of the reordered system. Typically, when using tearing techniques, codes apply Gaussian elimination to each of the diagonal blocks to calculate a local LU factorization. These factorizations are then used *without further pivoting* to eliminate the border nonzero elements. Even when such a factorization exists and is accurately computed, the pivot choices may cause substantial error growth when applied to the border rows. Additionally, there is no guarantee that the diagonal blocks are well-conditioned or even non-singular. The difficulties

in addressing these issues have prevented tearing techniques from being used in general matrix factorization packages.

In order to maintain stability it is necessary to apply a global pivoting strategy. This conflicts with the restrictions mentioned above, that are usually imposed in order to maintain the large grain structure of the matrix during the factorization. In general factorization routines, the global pivoting strategy usually involves making sure that a pivot element is within some factor of the maximum absolute value within the pivot row. In the case of border block upper triangular matrices such a strategy could lead to pivot choices which destroy the structure, e.g., the exchange with a column in the rightmost part of the matrix can result in the introduction of nonzero elements in the portion of the lower block triangular part of the matrix where zeros are desired. When stability control is combined with fill-in control, the pivot selection is done on the entire active portion of the matrix. Whenever a pivot is chosen outside the diagonal block being factored but not in the border, i.e., in one of the other diagonal blocks in the block upper triangular part, a row permutation is needed along with a column permutation. This row permutation also destroys the structure of the matrix.

Row permutations with the border, at the appropriate point in the factorization, do in fact preserve the bordered block upper triangular structure. For example, pairwise pivoting could be used to eliminate the rows of the border in parallel [7]. This preserves not only the general structure but the number of rows in each of the diagonal blocks and the border as well. (This is, of course, not true for structurally symmetric matrices where the bordered block upper triangular form is in fact an arrowhead form and any nonsymmetric permutation can potentially destroy structure.) There are some drawbacks, however. Pairwise pivoting can permute the relatively dense rows that tend to appear in the border into the diagonal blocks. This can increase fill-in during the factorization phase depending on exactly when the border is eliminated relative to the factorization of the rest of the diagonal block. 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 that have only one or two comparisons per pivot column or row. Finally, the complexity of the synchronization during the factorization and the application of the factorization to subsequent right-hand side vectors is nontrivial compared to other ways of handling the problem.

Other strategies discussed previously in the literature have resulted in solvers with either unacceptable cost or stability control, e.g., [2, 3, 4]. We would like to develop a strategy that preserves the overall structure of the matrix while allowing the implementation of a global pivoting strategy which yields a factorization with stability similar to more conventional nonsymmetric solvers, such as MA28 [8]. We will, however, allow the size of the border to increase during the factorization. In doing so we would also like to restrict any nonsymmetric permutations to the diagonal blocks of the block upper triangular part of the matrix and the diagonal block of the border.

4.2 Casting

The strategy used in MCSPARSE is based on a technique which combines standard nonsymmetric permutations for pivot selection within the diagonal blocks and symmetric permutations to facilitate the required global pivoting.

Definition 4.1 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.

Note that by definition casting a pivot is a symmetric permutation. Also note that in case of solving a bordered block upper triangular system whenever a pivot is cast the border size increases by one.

This casting can be incorporated into a factorization as follows:

```

acastnumb = 0
begin:
  i = 1
  castnumb = 0
  for k = 1 to N-acastnumb
    foreach  $a_{ji}, j > i$  and  $j \leq N$ -castnumb
      if  $p_{ii}$  is stable for  $a_{ji}$ 
        then eliminate  $a_{ji}$ 
      else cast  $p_{ii}$  ( $A \leftarrow \text{perm}(A)$ )
        castnumb = castnumb + 1
        goto end
      endif
    endforeach
    i = i + 1
  end:
  endfor
  if castnumb > 0 then
    acastnumb = acastnumb + castnumb
    go to begin
  endif
  for i = N - acastnumb + 1 to N
    find a stable pivot  $p_{ii}$ 
    foreach  $a_{ji}, j > i$ 
      eliminate  $a_{ji}$ 
    endforeach
  endfor

```

The last set of nested loops corresponds to the factorization of the diagonal block relating all of the cast pivots (possibly requiring a nonsymmetric permutation). The algorithm completes the first phase when all columns have either completed their eliminations or have cast the pivot element used for the column in the first phase. Note that the if statement which determines if castnumb is greater than zero will re-execute the first loops, if any pivots were cast, by jumping to the **begin:** label. The re-execution of the first loops is necessary to eliminate the values in the rows which contain the cast pivots. Values which have already been eliminated will be ignored so no redundant work will be performed.

Encountering 0 pivot elements in the initial part of the procedure does not cause problem since they will be cast and eliminated in the second phase of the factorization. The initial phase may cause some inefficiency since only diagonal elements are considered as pivots. This can be improved by allowing some nonsymmetric (local) permutations to place a stable pivot on the diagonal and thereby reduce casting.

5 Overview of MCSPARSE

5.1 Hybrid Ordering H*

As indicated above, the purpose of the ordering is to expose structure in the matrix that is not apparent to allow the exploitation of large and medium grain parallelism. H* attempts to achieve this goal and comprises four distinct

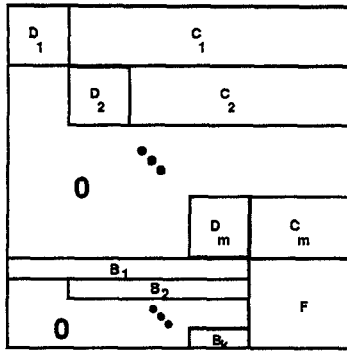


Figure 6: Bordered Block Upper Triangular Form

phases: H0, Tarjan's Algorithm, H1, and H2. These phases have been described in Section 3.

The structure of the matrix after the application of the permutations generated by the different phases is a bordered block upper triangular matrix. Further, the rows of the border are sorted based on the column index of their leftmost nonzero entry.

5.2 Matrix Structure

The structure of the reordered matrix is shown in Figure 6. Note that a block upper triangular form is assumed without losing generality. The interaction of diagonal blocks D_1 through D_m is confined to the off-diagonal blocks C_1 through C_{m-1} . The border diagonal block, F , comprises all of the separator sets produced by H*. Therefore, F interacts with all of the diagonal blocks through both the border and the off-diagonal blocks C_1 through C_m . The recursive nature of the production of the separator sets induces a block structure within the border. Specifically, the nonzeros of the rows belonging to a particular separator set from H1 or H2 are confined to the columns of the diagonal blocks which correspond to the block that was split by the separator. Note that the final sorting of the rows in the border does not affect this property and results in the staircase structure indicated in Figure 6. This induced border structure is exploited during the factorization.

5.3 Factorization of the Matrix

The factorization of the matrix is performed in four stages. The first stage is the factorization of the diagonal blocks. Complete 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 which uses an estimate of expected off-diagonal block C_j and border block fill-in². Then, out of these candidate pivots that pivot $p_{k,m}$ is chosen which has the lowest Markowitz count and satisfies $|p_{k,m}| \geq \mu \times \max_k |a_{k,m}|$, with $\mu = 0.1$ and the constraint $|p_{k,m}| > \alpha$, with $\alpha = 10^{-5}$. If a pivot can not be found satisfying the stability constraint then the column and corresponding row are cast. (The issue of a factorization not existing for a diagonal block is discussed Section 4.)

Since this matrix is bordered block upper triangular, there are no edges from diagonal block D_j to diagonal block

D_i , $\forall i, i < j$. Therefore, when a pivot is selected in diagonal D_i it will not perform any updates on the rows in block D_j . Nor will the pivots in the block D_j update any of the rows in block D_i . As a result, the LU factorization of the blocks can be performed in parallel. Similarly, after the diagonal blocks have been factored, the L factors can be used to update the off-diagonal blocks in parallel.

Next, the border blocks are eliminated using the diagonal blocks and the off-diagonal blocks. Again, the absolute values of the pivots chosen during the diagonal block factorizations are checked against the absolute values of the border elements. Whenever the absolute value of a pivot is smaller than $10^{-6} \times$ the absolute value of the border element to be eliminated, the column and corresponding row are cast to the border, see also Section 4.

The elimination of a given border row by the pivots in the diagonal blocks must respect certain dependencies. A diagonal block D_j cannot update a row in the border until after the row has been updated by all blocks D_i , $i < j$. However, the update of a row in the border is independent of the update to the other rows in the border. Therefore, a diagonal block D_j can update the rows of the border in parallel. The staircase structure of the border can be exploited to produce appropriate granularity for a particular processor. The staircase structure implies that the number of diagonal blocks involved in the initial updates is equal to the number of "stairs" in the border. This can be used to enhance the initial distribution of work and data (diagonal and off-diagonal blocks) across the processors.

6 MCSPARSE Results

This section presents summary results for the large-grain parallel sparse system solver, MCSPARSE, on the Cedar multiprocessor. For a more detailed look at the Cedar performance results for MCSPARSE see [21].

6.1 Stability Results

To determine if MCSPARSE was stable, a comparison was made between the stability of MCSPARSE and MA28, [8], a standard sparse system solver. Both algorithms were used to solve the large test matrices from the Harwell-Boeing test collection and the relative maximum norm of the error,

$$\text{error} = \frac{\max_{1 \leq i \leq n} (|x_{\text{calculated}} - x_{\text{known}}|)}{\max_{1 \leq i \leq n} (|x_{\text{known}}|)} \quad (2)$$

was calculated for all of the solutions. For the comparison, MA28 was run with the stability factor³ (u) at 1.0 and with a value of 0.1. MCSPARSE was run with the diagonal casting ($\alpha = 10^{-5}$) and border casting ($\epsilon = 10^{-6}$). There are, of course, many combinations of casting that can be done. The combination used here and its parameters have been tuned via many experiments. The interested reader is directed to [20] for the details of the casting comparisons and tuning. Also, MCSPARSE was run both with and without iterative refinement (I.R.).

Table 3 compares the stability for fourteen of the large RUA matrices. For these matrices MCSPARSE degrades slightly, for 50% of the matrices the relative error for MCSPARSE is of the same order of magnitude or better than MA28 and for 78% of the matrices the difference between the solvers is

²The estimate of the number of fill elements generated outside the diagonal block is based on a worst case scenario. For further details about this estimate see [20].

³The stability factor u is used to restrict the pivot choices such that a pivot $a_{i,j}$ can only be used if $|a_{i,j}| \geq u \times \max_k |a_{k,j}| \forall k$ in the active portion of the matrix.

within four orders of magnitude. However, with the addition of a few steps of iterative refinement, MCSPARSE is able to solve 13 of the matrices with a relative error of at least the same order of magnitude as MA28 and the remaining matrix has a relative error of only one order of magnitude worse than MA28. This compares favorably to results in [4] where the augmentation of their limited pivoting strategy with iterative refinement did not produce satisfactory results.

Matrix	Relative Error				Fill-in ($\times 10^3$)		
	MCSPARSE		MA28		MCSP.	MA28	
	W/out I.R.	With I.R.	$u =$ 1.0	$u =$ 0.1		$u =$ 1.0	$u =$ 0.1
1 gaff1104	.1E-06	.1E-06	.5E-06	.4E-06	105	58	62
2 gemat11	.3E-10	.4E-11	.3E-10	.4E-10	143	28	19
3 gre_1107	.7E-06	.1E-09	.4E-08	.6E-06	183	40	37
4 hwatt_2	.4E-13	.4E-13	.6E-06	.2E-06	245	102	247
5 mahistlh	.1E-08	.6E-11	.1E-12	.1E-12	12	5	3
6 nnc1374	.5E-02	.5E-03	.2E-04	.1E-04	192	58	43
7 or678lhs	.5E-13	.3E-14	.5E-12	.9E-12	221	25	14
8 orsreg_1	.1E-12	.7E-13	.4E-12	.4E-11	261	311	137
9 pores_2	.1E-05	.6E-12	.1E-09	.1E-09	97	29	29
10 saylr4	.9E-11	.2E-11	.2E-10	.8E-10	548	288	451
11 sherman2	.1E-02	.3E-11	.6E-08	.7E-06	431	258	241
12 sherman3	.5E-12	.2E-12	.4E-12	.2E-09	311	190	368
13 sherman5	.6E-07	.2E-14	.2E-12	.2E-09	314	191	132
14 west2021	.1E-05	.1E-09	.4E-08	.3E-08	25	4	4

Table 3: Stability and Fill-in Comparison Between MCSPARSE and MA28

6.2 Fill-in Results

To determine if the modified Markowitz count was successful in reducing the amount of fill-in a number of tests were run using the RUA matrices from the Harwell-Boeing test collection. This section presents the results from the tests conducted with the large matrices (the matrices with at least 1,000 rows). The number of rows and original number of elements in the matrices can be found in Table 1.

The number of fill-in elements from the tests are in Table 3. This table also contains two other columns. The MA28 column indicates the number of fill-in elements generated by MA28 with the stability factor (u) at 1.0 and 0.1.

As expected, MA28 almost always produces less fill-in than MCSPARSE due to its more global pivot search. However, MCSPARSE benefits from a localization of the fill-in which allows for a more efficient exploitation of storage (due to selected use of dense structures) so that the cost of the extra work created by fill-in is significantly reduced.

6.3 Cedar Performance Results

This section summarizes the performance results for MCSPARSE collected on a four cluster Cedar configuration, with each cluster comprising eight processors. In fact, each cluster represents an Alliant FX/8 with increased cache size and the four clusters share a global memory accessed by an omega interconnection network[37]. Since MCSPARSE was designed to exploit large grain parallelism, the multi-cluster performance is examined first, followed by the one cluster performance. The times given in this section are wall clock times, in seconds, for the code running in single user mode.

The size of the system to be solved has to be fairly large in order to reduce the overhead associated with the exploitation of large grain parallelism to an acceptable level. So, only

the fourteen large-sized Harwell-Boeing matrices listed in Table 3 were used to obtain performance measurements on the Cedar system. The results for these systems are summarized in Table 4, which compares the time to run on four clusters, $T(4CL)$, (using all 32 processors) to the time to run on one cluster, $T(1CL)$, (using eight processors). For these tests, repeated runs were used to eliminate any effects the virtual memory system might have on the results.

Matrix	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$T(1CL)$	2.1	3.1	2.3	2.1	1.8	1.9	2.6	3.0	1.7	3.0	2.2	2.8	3.0	1.9
$T(4CL)$														

Table 4: Summary of MCSPARSE Multi-Cluster Performance on Cedar (Matrices Numbered as in Table 3)

As can be seen from these tests, the multi-cluster performance of Cedar resulted in a cluster speed up ranging from 1.7 up to 3.1 (out of a possible 4.0). For 10 of the 14 matrices, the speed up was at least 2 and for 4 of the matrices the speed up was at least 3. In the following section the one cluster performance of MCSPARSE is compared to the performance of MA28 (the results from these two sections can be combined to determine the standard speed up comparing one processor to 32 processors.)

6.4 Performance Comparison for MCSPARSE and MA28

In this section we give performance results for MCSPARSE on one cluster of Cedar, an Alliant FX/8, and compare its effectiveness against a known sequential sparse solver, MA28 [8].

The solution times of the large matrices from the RUA collection for both the MCSPARSE and MA28 solvers are presented in Table 5. This table contains the wall clock times for the solutions as collected in single-user mode on the one cluster of Cedar. The times for the MCSPARSE solver are presented for both one and eight processor runs. The times for MA28 are presented for eight processor runs.

When comparing the solution times for MCSPARSE against MA28, it is necessary to include the ordering time for the matrix along with the solution time. The columns labeled as *Total* contain the sum of the ordering time and the solution time. For MA28 the solution times are presented for the stability constraint $u = 0.1$ and using the improved pivot search algorithm with the pivot search being limited to four rows and four columns. The single time presented for the MA28 run contains both the ordering and solution time.

This table shows that, although MCSPARSE was not specifically designed to run efficiently on an Alliant FX/8, the speedup obtained for eight processors over one processor is significant. The Alliant FX/8 is a tightly coupled multiprocessor compared to the Cedar architecture for which MCSPARSE was intended. These results clearly indicate that the large and medium grain parallelism exploited by MCSPARSE does not entail an unnecessary amount of overhead or mismatch in load balance that would prevent reasonable performance on a tightly coupled architecture. Second, it can be observed that the time for performing the ordering H^* is less than the time needed for factoring and solving the system, though still proportional to the latter one. It should be noted, however, that the ordering was performed on one processor. The ordering time could be reduced significantly via a parallel implementation, which should be easy realizable due to the recursive nature of H^* . The comparison with MA28 shows that the performance improvement can vary considerably, but is substantial, e.g., a factor of 29 for

Matrix	Hybrid Reorder	MCSPARSE				MA28
		1CE		8CE		8CE
		Solve	Total	Solve	Total	$u = 0.1$
gaff1104	2.14	24.49	26.63	5.22	7.36	54.72
gemat11	4.04	40.14	44.18	10.96	15.00	11.41
gre_1107	3.65	20.53	24.18	3.76	7.41	28.46
hwatt.2	2.23	33.49	35.72	6.28	8.51	77.82
mahistlh	1.45	3.18	4.63	0.72	2.17	3.30
nnc1374	3.26	19.97	23.23	3.81	7.07	27.84
or678lhs	7.20	38.75	45.95	7.62	14.82	62.69
orsreg.1	3.07	41.69	44.76	9.61	12.68	117.79
pores_2	2.53	11.15	13.68	2.19	4.72	19.03
saylr4	5.15	114.06	119.21	23.51	28.66	306.07
sherman2	5.40	78.41	83.81	13.37	18.77	554.09
sherman3	4.09	127.64	131.73	26.85	30.94	187.87
sherman5	4.70	125.68	130.38	28.62	33.32	309.70
west2021	5.89	6.87	12.76	1.64	7.53	2.38

Table 5: Solution Time Comparison Between MCSPARSE and MA28

sherman2. The eight processor version of MA28 was produced via a restructuring compiler so there is clearly room for improvement in its performance. Nevertheless, the superiority of MCSPARSE is often large enough to indicate any performance increase via a redesign of MA28 to apply parallel pivots might still fall short. In any case, MCSPARSE often compares favorably with such a parallel pivots code for nonsymmetric systems. The interested reader should see [19] for the performance of the nonsymmetric sparse code Y12M2.

7 Conclusions

A parallel solver for nonsymmetric linear systems of equations, MCSPARSE, was introduced, which combines different granularities of parallelism. One of the main concerns addressed by MCSPARSE is the maintaining of stability and sparsity at acceptable levels while allowing large grain parallelism to be exploited. This is achieved by using a novel ordering technique H^* combined with a new technique, casting, which provides a mean to discard the application of unstable pivots during the factorization. This enables MCSPARSE to obtain stable factorizations which are comparable to standard factorization routines, such as MA28.

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

Casting has been described for general matrices and for the bordered block upper triangular form produced by H^* . For the latter matrices, casting maintains stability by using numerical information gathered during the factorization to adjust the diagonal blocks and the border produced by H^* . The particular implementation of diagonal block and border block casting used in MCSPARSE has been described and evaluated by comparison with MA28.

Multiple levels of parallelism are present and exploitable in MCSPARSE: very large-grain parallelism with several diagonal block factorizations and border block updates per cluster of processors; large-grain parallelism within a cluster when factoring a diagonal block per processor; medium-

grain parallelism when using the processors in one cluster to factor a single diagonal block or update a single border block; and fine-grain vectorization used within each processor. Experiments investigating the performance of MCSPARSE on both a tightly coupled multi-vector processor, an Alliant FX/8, and a more loosely coupled cluster-based architecture, a four cluster Cedar, have been reported and show the algorithm's effectiveness.

There are several avenues of investigation left to pursue with respect to MCSPARSE. A parallel implementation of the H^* ordering would improve further the overall performance of MCSPARSE. The code could be adapted to map its multilevel parallelism onto other multi-vector processors and to exploit their architectures efficiently. Initial results, [40], 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 [18, 17], to produce a preconditioner for conjugate gradient-like algorithms. Finally, the techniques used in MCSPARSE should be considered for use with more conventional approaches to solving systems with tearing techniques, e.g., exploiting the Sherman-Morrison-Woodbury formula.

References

- [1] ALAGHBAND, G. A parallel pivoting algorithm on a shared memory multiprocessor. In *Proceedings of the 1988 International Conference on Parallel Processing, Volume 3: Applications and Algorithms* (1988), vol. 3, pp. 177-180.
- [2] ARIOLI, M., AND DUFF, I. S. Experiments in tearing large sparse systems. In *Reliable Numerical Computation*, M. G. Cox and S. Hammarling, Eds. Oxford University Press, New York, 1990, pp. 207-226.
- [3] ARIOLI, M., DUFF, I. S., GOULD, N. I. M., AND REID, J. K. The practical use of the Hellerman-Rarick P^4 algorithm and the P^5 variant of Erisman et al. Tech. Rep. Report CSS 213, CSS Division, Harwell Laboratory, England, 1987.
- [4] ARIOLI, M., DUFF, I. S., GOULD, N. I. M., AND REID, J. K. Use of the P^4 and P^5 algorithms for in-core factorization of sparse matrices. *SIAM J. Sci. Stat. Comput.* 11, 5 (September 1990), 913-927.
- [5] ASHCRAFT, C. C., GRIMES, R. G., LEWIS, J. G., PEYTON, B. W., AND SIMON, H. D. Progress in sparse matrix methods for large linear systems on vector supercomputers. *Intl. J. Supercomputing Appl.* 1, 4 (Winter 1987), 10-30.
- [6] CHEUNG, L. K., AND KUH, E. S. The bordered triangular matrix and minimum essential sets of a digraph. *I.E.E.E. Transactions on Circuits and Systems CAS-21*, 5 (September 1974), 633-639.
- [7] DAVIS, T. A parallel algorithm for sparse unsymmetric LU factorization. Tech. Rep. CSRD Report No. 907, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, 1989. PhD. thesis.
- [8] DUFF, I. S. Ma28— a set of fortran subroutines for sparse unsymmetric linear equations. Tech. Rep. Report AERE R8730, HMSO, London, 1977.

- [9] DUFF, I. S. Algorithm 575. permutations for a zero-free diagonal. *ACM Trans. Math. Software* 7, 3 (September 1981), 387–390.
- [10] DUFF, I. S. On algorithms for obtaining a maximum transversal. *ACM Trans. Math. Software* 7, 3 (September 1981), 315–330.
- [11] DUFF, I. S. Parallel implementation of multifrontal schemes. *Parallel Computing* 3 (1986), 193–204.
- [12] DUFF, I. S., ERISMAN, A. M., AND REID, J. K. *Direct Methods for Sparse Matrices*. Oxford University Press, New York, 1986.
- [13] DUFF, I. S., AND REID, J. K. The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Trans. Math. Software* 9 (1983), 302–325.
- [14] ERISMAN, A. M., GRIMES, R. G., LEWIS, J. G., AND POOLE, W. G. A structurally stable modification of Hellerman-Rarick's P^4 algorithm for reordering unsymmetric sparse matrices. *SIAM J. Numer. Anal.* 22, 2 (April 1985), 369–385.
- [15] ERISMAN, A. M., GRIMES, R. G., LEWIS, J. G., POOLE, W. G., AND SIMON, H. D. Evaluation of orderings for unsymmetric sparse matrices. *SIAM J. Sci. Stat. Comput.* 8, 4 (July 1987), 600–624.
- [16] GALLIVAN, K., MARSOLF, B., AND WIJSHOFF, H. A large-grain parallel sparse system solver. In *Proc. Fourth SIAM Conf. on Parallel Proc. for Scient. Comp.* (Chicago, IL, 1989), pp. 23–28.
- [17] GALLIVAN, K., SAMEH, A., AND ZLATEV, Z. Parallel hybrid sparse linear system solver. *Computing systems in engineering* 1 (1990), 183–195.
- [18] GALLIVAN, K., SAMEH, A., AND ZLATEV, Z. Solving general sparse linear systems using conjugate gradient-type methods. In *Proceedings of the 1990 International Conference on Supercomputing* (New York, 1990), ACM Press, pp. 132–139. June 11–15, 1990, Amsterdam, The Netherlands.
- [19] GALLIVAN, K., SAMEH, A., AND ZLATEV, Z. Parallel direct method codes for general sparse matrices. Tech. Rep. CSRD Report No. 1143, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, 1991. To appear in Proceedings of NATO ASI on Linear Systems. Bertocchi, Spedicato and Vespucci, eds., 1991.
- [20] GALLIVAN, K. A., MARSOLF, B. A., AND WIJSHOFF, H. A. G. MCSPARSE: A parallel sparse unsymmetric linear system solver. Tech. Rep. CSRD Report No. 1142, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, 1991.
- [21] GALLIVAN, K. A., MARSOLF, B. A., AND WIJSHOFF, H. A. G. Solving large nonsymmetric sparse linear systems on Cedar. Tech. Rep. CSRD Report No. 1313, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, 1993.
- [22] GEORGE, A. Nested dissection of a regular finite element mesh. *SIAM J. Numer. Anal.* 10, 2 (April 1973), 345–363.
- [23] GEORGE, A. An automatic one-way dissection algorithm for irregular finite element problems. *SIAM J. Numer. Anal.* 17, 6 (December 1980), 740–751.
- [24] GEORGE, A., AND LIU, J. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice Hall, 1981.
- [25] GEORGE, A., AND LIU, J. W. H. An automatic nested dissection algorithm for irregular finite-element problems. *SIAM J. Numer. Anal.* 15 (1978), 1053–1069.
- [26] GUSTAVSON, F. Finding the block lower triangular form of a matrix. In *Sparse Matrix Computations*, J. Bunch and D. Rose, Eds. Academic Press, New York, 1976.
- [27] HELLERMAN, E., AND RARICK, D. C. The partitioned preassigned pivot procedure (P^4). In *Sparse Matrices and their Applications*, D. J. Rose and R. A. Willoughby, Eds. Plenum, New York, 1972.
- [28] HOPCROFT, J. E., AND KARP, R. M. An $n^{\frac{5}{2}}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.* 2, 4 (December 1973), 225–231.
- [29] KUHN, H. W. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly* 2, 1 (March 1955), 83–97.
- [30] LEISERSON, C., AND LEWIS, J. Orderings for parallel sparse symmetric factorization. In *Proc. Third SIAM Conf. on Parallel Proc. for Scient. Comp.* (Los Angeles, CA., 1987), pp. 27–31.
- [31] LIN, T. D., AND MAH, R. S. H. Hierarchical partition - a new optimal pivoting algorithm. *Mathematical Programming* 12 (1977), 260–278.
- [32] LIPTON, R., ROSE, D., AND TARJAN, R. Generalized nested dissection. *SIAM J. Numer. Anal.* 16 (1979), 346–358.
- [33] M. HALL, J. An algorithm for distinct representatives. *The American Mathematical Monthly* 63, 10 (December 1956), 716–717.
- [34] MARKOWITZ, H. M. The elimination form of the inverse and its application to linear programming. *Management Science* 3 (April 1957), 255–269.
- [35] OSTERBY, O., AND ZLATEV, Z. *Direct methods for sparse matrices*. Springer, Berlin, 1983.
- [36] SANGIOVANNI-VINCENTELLI, A. An optimization problem arising from tearing methods. In *Sparse Matrix Computations*, J. R. Bunch and D. J. Rose, Eds. Academic Press Inc., New York, 1976, pp. 97–110.
- [37] STAFF. The Cedar Project. Tech. Rep. CSRD Report No. 1122, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, 1991.
- [38] STEWARD, D. V. Partitioning and tearing systems of equations. *SIAM Journal Numerical Analysis* 2, 2 (1965), 345–365.
- [39] TARJAN, R. E. Depth-first search and linear graph algorithms. *SIAM J. Computing* 1 (1972), 146–160.
- [40] WANG, X. private communication, April 1991.
- [41] WIJSHOFF, H. A. G. Symmetric orderings for unsymmetric sparse matrices. Tech. Rep. CSRD Report No. 901, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, 1989.