MCSPARSE: A parallel sparse unsymmetric linear system solver

K.A. Gallivan, B.A. Marsolf, H.A.G. Wijshoff

RUU-CS-91-36
September 1991

Utrecht University

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

MCSPARSE: A parallel sparse unsymmetric linear system solver

K.A. Gallivan, B.A. Marsolf, H.A.G. Wijshoff

Technical Report RUU-CS-91-36
September 1991

Department of Computer Science
Utrecht University
P.O.Box 80.089
3508 TB Utrecht
The Netherlands

ISSN: 0924-3275

MCSPARSE: A Parallel Sparse Unsymmetric Linear System Solver[*]

K. A. Gallivan[†] B. A. Marsolf[†] H. A. G. Wijshoff[‡]

Abstract

In this paper, an unsymmetric 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 resulting linear systems are not symmetric. In other application areas, such as linear programming, optimization problems, directed network problems, and simulation problems the resulting linear systems are even unsymmetric in structure. For these arbitrary unsymmetric 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 completely resolved until information from the actual factorization is available, i.e., some decisions must take place

[*]This research was supported by the National Science Foundation under Grants No. US NSF-MIP-8410110, US NSF CCR-8717942, by the Department of Energy under Grant No. DE-FG02-85ER25001 and by the CSRD Affiliates Program.

[†]Center for Supercomputing Research and Development, University of Illinois, Urbana, Illinois 61801.

[‡]Department of Computer Science, Utrecht University, 3584 CH Utrecht, The Netherlands.

dynamically. As a result, for unsymmetric 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 multifrontal scheme [Duf86, DR83]. A multifrontal 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 [Ala88, Dav89, GSZ91]. 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 [AGL+87]. 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 fillin that is generated. Most sequential sparse matrix packages, such as MA28, used a simple strategy proposed by Markowitz [Mar57]. This strategy involves counting the number of nonzero elements in each column, c_j, and the number 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.

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, [OZ83]. 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 unsymmetric 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 [EGL+87, HR72]. 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, consdering 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 can increase fillin and reduce 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 MCSPARSE which allows adaptation to a wide range of multiprocessor architectures. In [GMW89] initial results with MCSPARSE are presented and more details can be found in [Mar91].

The paper is organized as follows. In Section 2 a global overview of the procedures in MCSPARSE is given. The details of the ordering H* are presented in Section 3. Casting is introduced and discussed from an algebraic point of view in Section 4. The details of the implementation of the factorization and solve phases of MCSPARSE are described in Section 5. Experimental results and conclusions are given in Sections 6 and 7.

2 Overview of MCSPARSE

This section presents an overview of the ordering and the solver and the relationship between the two. Though the two algorithms are independent, they were designed to complement each other.

2.1 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 phases.

The initial phase of the ordering, H0, attempts to adaptively find a weighted transversal. A transversal is defined by row and column permutations which ensure that all of the diagonal elements of the permuted matrix are nonzero. If such permutations do not exist then the matrix is structurally singular [DER86]. When determining this transversal H0 attempts to select elements for the diagonal that will enhance the stability of the factorization.

The second phase of the ordering applies Tarjan's Algorithm to determine the strongly connected components of the adjacency graph of the matrix. These strongly connected components determine the diagonal blocks of a block triangular form of the matrix.

The third phase of the algorithm, H1, uses the depth-first search of Tarjan's algorithm and other techniques to find separator sets. This algorithm is applied individually to the diagonal blocks that result from Tarjan's algorithm to reduce their size by creating a border for each block. At the end of the phase the individual border blocks are permuted to the end of the matrix to form a border for the entire matrix.

The final phase of the algorithm, H2, relies in part on nested dissection but also contains novel techniques for reducing the separator sets that exploit the unsymmetric nature of the matrix. H2 is applied individually to the diagonal blocks that result from the previous phase and are larger than some threshold.

The last three phases of the ordering determine a symmetric permutation of the matrix. This symmetric permutation when applied to the matrix will not remove any of the nonzero entries placed on the diagonal by H0. The structure of the matrix after the application of the permutations 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.

A preliminary algorithmic description of the H* ordering can be found in [Wij89].

2.2 Matrix Structure

The structure of the reordered matrix is shown in Figure 1. 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 1. This induced border structure is exploited during the factorization.

2.3 Factorization of the Matrix

The factorization of the matrix is performed in four stages. The first stage is the factorization of the diagonal blocks. (The issue of a factorization not existing for a diagonal block is discussed later.) 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 block D_i it will not perform any updates on the rows in block D_j. Nor will the pivots in the diagonal block D_j update any of the rows in diagonal 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. The elimination of a given border row by the pivots in the diagonal blocks must respect certain dependences. 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.

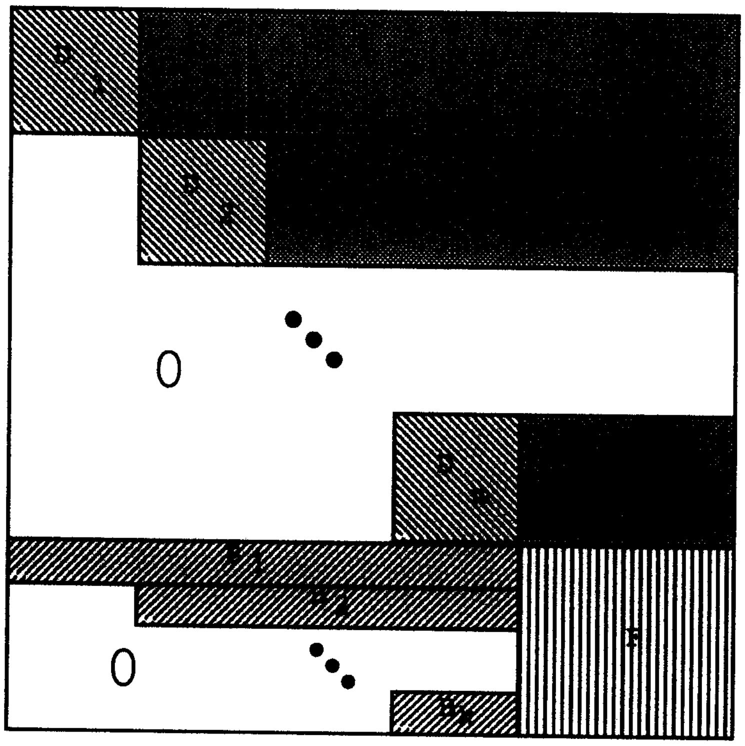


Figure 1: Bordered Block Upper Triangular Form

$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & x & & x & & x \\ 2 & & x & x & & \\ 3 & & & x & x & \\ 4 & x & & & x & \\ 5 & x & x & & & x \end{array}$$

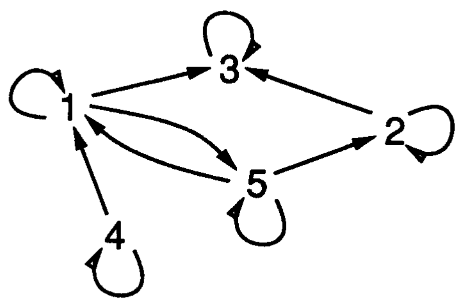


Figure 2: A 5 x 5 sparse matrix and its associated digraph.

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 an unsymmetric (square) sparse matrix A $(N \times N)$. 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.*

In other words A is the adjacency matrix of $G(V, E)$. An example of a 5×5 sparse matrix and its associated digraph are depicted in Figure 2.

The hybrid ordering H* is composed of two different types of orderings: unsymmetric and symmetric.

Definition 3.2 *An ordering of a sparse matrix is called* unsymmetric *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. Unsymmetric orderings are obtained by independent row and column interchanges of the matrix, with P representing the row interchanges and Q representing the column interchanges. So, where the unsymmetric orderings change certain properties of the sparse matrix, e.g., the eigenvalues, symmetric orderings maintain these. Also, if A is a diagonally dominant matrix, then after a symmetric ordering the resulting matrix will still be diagonally dominant, whereas an unsymmetric ordering destroys this property. The unsymmetric ordering is used to enhance the numerical properties of the factorization of the matrix. In H*, the symmetric orderings are used to obtain a bordered block triangular matrix.

In order to obtain the desired structure H* exploits the concept of a node separator set and 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 $x \in C$.

2. $x \in B$ and $y \in C$ and

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

3.2 H0

H0 is a transversal algorithm for permuting nonzero entries onto the diagonal using an unsymmetric ordering. The transversal algorithm has been modified to permute large elements to the diagonal in order to enhance the stability of the subsequent factorization.

3.2.1 The Transversal

The transversal ordering is a matching between the columns and the diagonals and could be found using many different algorithms. Algorithms for finding set representation [MH56] or solutions to the assignment problem[Kuh55] could be used to find the transversal. An alternative algorithm involves finding maximal matchings in bipartite graphs[HK73].

The algorithm chosen for the transversal is based on work of Duff and Gustavson [Duf81b, Duf81a, Gus76]. 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. The nonzero elements in column j are now searched to determine if there is an element whose row index i has not been considered as a means of determining a column interchange to extend the transversal. If such an element is found with a row index of, say, i, then the column to which i was assigned as a diagonal position becomes the current column, *cur_col*. Correspondingly, diagonal i is assigned to column j. The index i is then marked as having been considered during this step of the algorithm. Column j is saved as the parent column of column *cur_col* in a linked list of columns considered for possible interchange.

The algorithm then attempts to extend the transversal using *cur_col*. It first attempts to find a *easy* insertion as described previously. If an *easy* insertion is possible, the appropriate diagonal index is assigned to *cur_col* and current step of the algorithm terminates.

However, if an *easy* insertion is not possible, then the algorithm attempts to interchange *cur_col* with a column that has not yet been considered on this step. This is done by considering the elements in *cur_col* and extending the linked list as described above. If this extension is not possible the algorithm backtrack by replacing *cur_col* with its parent in the linked list of columns and continuing to scan the elements in the new *cur_col*.

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.

3.2.2 The Bounded Transversal

The transversal algorithm described above 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_{ij} to the diagonal if its value is within a bound, α, of the largest element in the column, i.e.,

$$\mid a_{ij} \mid \geq \alpha * \max_k(\mid a_{kj} \mid) \tag{1}$$

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. Then during the search phase, for both the *easy* insertion and the replacement insertions, an element will only be selected if it also 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 transversal cannot be found with the bound α, then the bound is loosened to allow more candidates and the algorithm is restarted. If after several attempts at loosening the bound 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.

3.3 Tarjan's Algorithm

Tarjan's algorithm finds the strongly connected components of the digraph associated with the matrix with time complexity linear in the number of nodes and edges. 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 finding a root node, *root*, and setting the current node equal to the root node, ($current = root$). The current node is then placed on the stack and marked as being processed. In addition, a pointer is kept for each node on the stack that indicates the lowest position on the stack reachable from that node via some path. This pointer is initialized to the node's position when initially placing a node on the stack ($low_{current} = pos(current)$).

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 pointer of node *current* and the low pointer of node y ($low_{current} = min(low_{current}, low_y)$). The algorithm now goes on to the next edge from node *current*. If node y is not on the stack, then it has been removed earlier and the algorithm goes on to the next edge from node *current*.

If the node y has not been processed, then it must be added to the stack initializing its low pointer to its position on the stack and saving a pointer to its predecessor, node *current*. The current node is now set to be the new node, ($current = y$), 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 comparing the position of the current node $low_{current}$. 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 ($low_{pred} = min(low_{pred}, low_{current})$). The predecessor is then taken to be the current node ($current = pred$) and the search of the predecessor's edges is resumed.

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

3.4 H1 Algorithm

A disadvantage of Tarjan's algorithm is that most sparse matrices do not allow a nice decomposition into strongly connected components. A typical case is a matrix 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 form a 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, α, on the stack two additional pointers are required. The first, denoted $nlow_\alpha$, is a pointer to the position of the node lowest on the stack that can be reached from α by a single edge. The second, denoted $blow_\alpha$, 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 α. 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_α for a node α 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 *blow* pointer is used to define an initial quasi-separator set consisting of the nodes on the stack from $blow_{current}$ to $pos(current) - 1$.

Throughout the algorithm, whenever an edge to a node y is encountered such that $pos(y) - blow_{current} \geq T_{long}$ 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 to an unacceptable level. So, in order to minimize the size of the quasi-separator set, the pointer $blow_{current}$ 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 $min(blow_{current}, pos(y))$ to $pos(current) - 1$ in the quasi-separator set. The pointer $nlow_\alpha$ 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 *blow* are updated in a manner similar to that used to update low_α 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 $blow_{current}$ 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 $blow_{current} - nlow_{current} < T_{long}$ then
    $blow_{current} = \min(blow_{current}, nlow_{current})$
end if
$blow_{prev} = \min(blow_{prev}, blow_{current})$
$low_{prev} = \min(low_{prev}, low_{current})$
```

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 latter 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 $blow_{current}$ to $pos(current) - 1$. As noted

$$\begin{array}{|cccccccccc|}
\hline
1 & 1 & & & & & & & & \\
 & 2 & 1 & & & & & & & \\
 & & 3 & 1 & & & & & & \\
 & & & 4 & 1 & & & & & \\
 & & & & 5 & 1 & & & & \\
 & & & 1 & & 6 & 1 & & & \\
 & & & & & & 7 & 1 & & \\
 & & & & & & & 8 & 1 & \\
 & & & & 1 & & & & 9 & 1 \\
1 & & & & & & & & & 10 \\
\hline
\end{array}$$

Figure 3: Nonzero Entries in Matrix

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 considered. A new starting node is found and the algorithm restarts on the nodes yet to be completed.

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 reach from the starting node have been processed, then the algorithm starts over looking for a new root node that has not been processed. 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 matrix in Figure 3. The associated directed graph for this matrix is in Figure 4.

Figure 5 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 it 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 ($blow_6 = 5$). This would give an initial bordered block size of 5, a quasi-separator size of 2, and a remaining block size of 3.

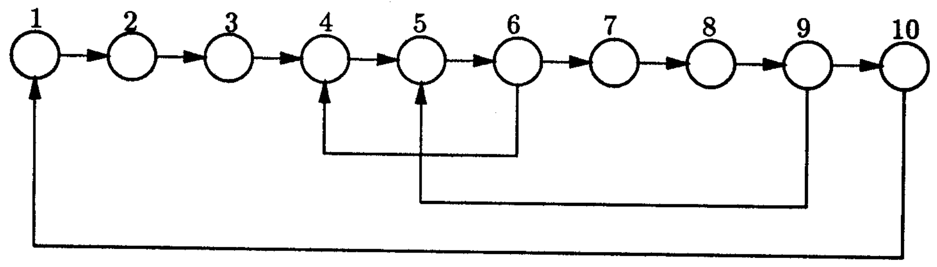


Figure 4: Associated Directed Graph for the Matrix

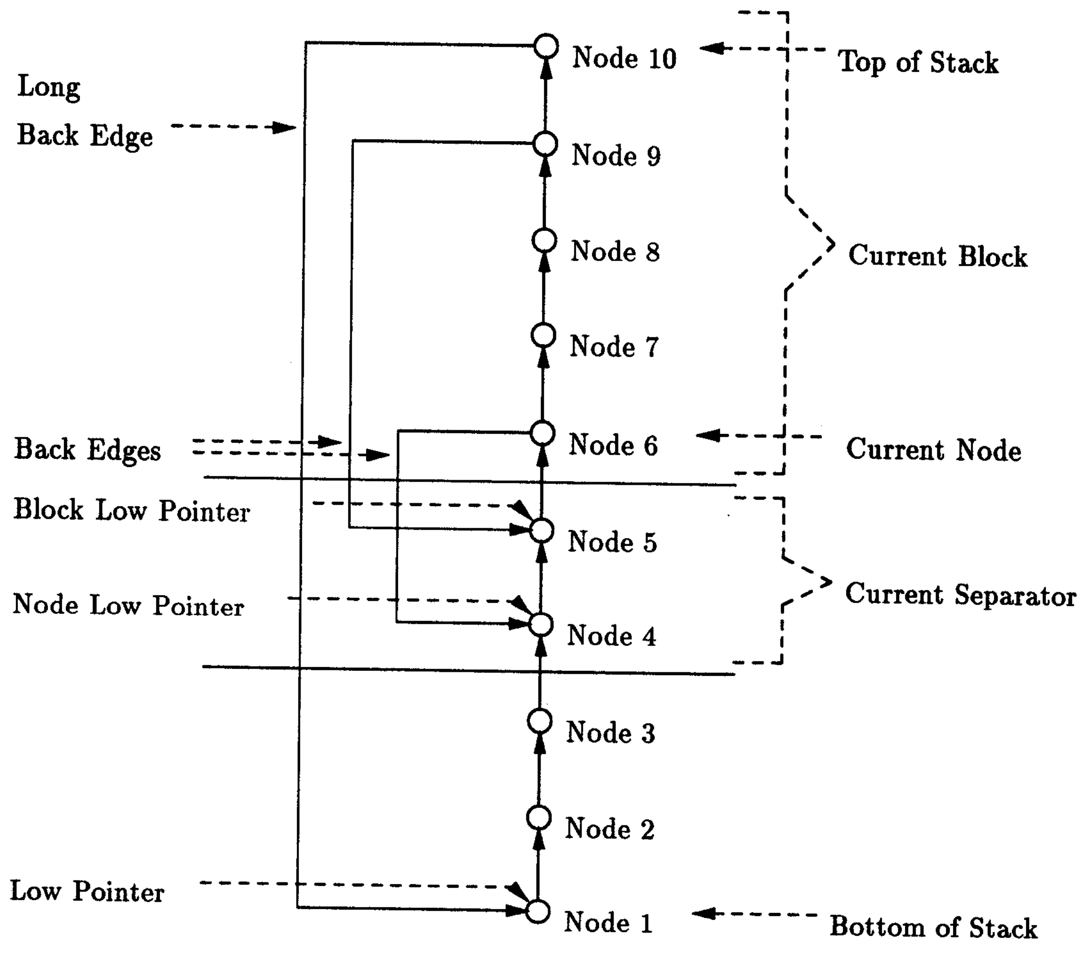


Figure 5: H1 Stack

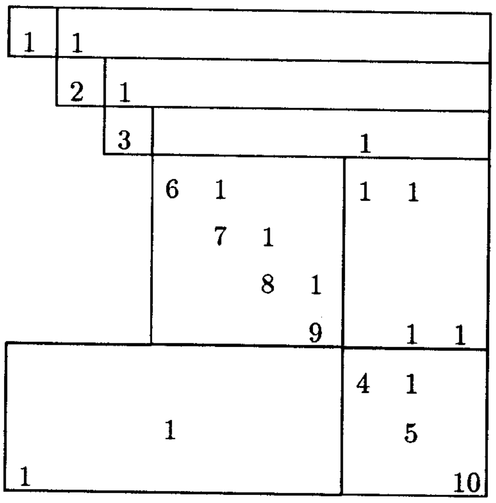


Figure 6: Reordered Matrix

Assuming the block sizes meet the necessary constraints, then the search would be made for the *long* back edges. This search would then find the edge {10,1} which would place node 10 in the quasi-separator set. The bordered block size would become 4, the quasi-separator set would become 3, and the remaining block would stay at 3. The bordered block would contain nodes 6, 7, 8, and 9. The quasi-separator block would contain the nodes 4, 5, and 10. The H1 algorithm would be applied to the remaining nodes which would result in the three independent blocks 1, 2, and 3 The reordered matrix resulting from the H1 algorithm is in Figure 6.

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 unsymmetric 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 [Geo73, GL78].

The ordering H2 starts with the construction of separator sets of the adjacency matrix of $A + A^T$ as in the standard approaches. For the implementation of H2 we used a straightforward implementation of automatic nested dissection [GL81]. However, other initial orderings could have been used such as one-way dissection [Geo80], more sophisticated implementations of automatic nested dissection [LRT79], or the graph bisection heuristics as proposed by [LL87]. In fact, the initial algorithm used for finding these separator sets does not appear to be very important.

Nested dissection preserves structural symmetry and therefore results in a matrix with an arrowhead form. However, since the objective of the H2 ordering is to bring the matrix into bordered upper triangular block form, the use of nested dissection only is too restrictive and the constraints on the separator set can be relaxed. It is this fact which is exploited by the H2 ordering. After each stage when a separator set S is constructed the ordering 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 matrix and thereby creating a quasi-separator set. The H2 algorithm is only applied to diagonal blocks produced by H1

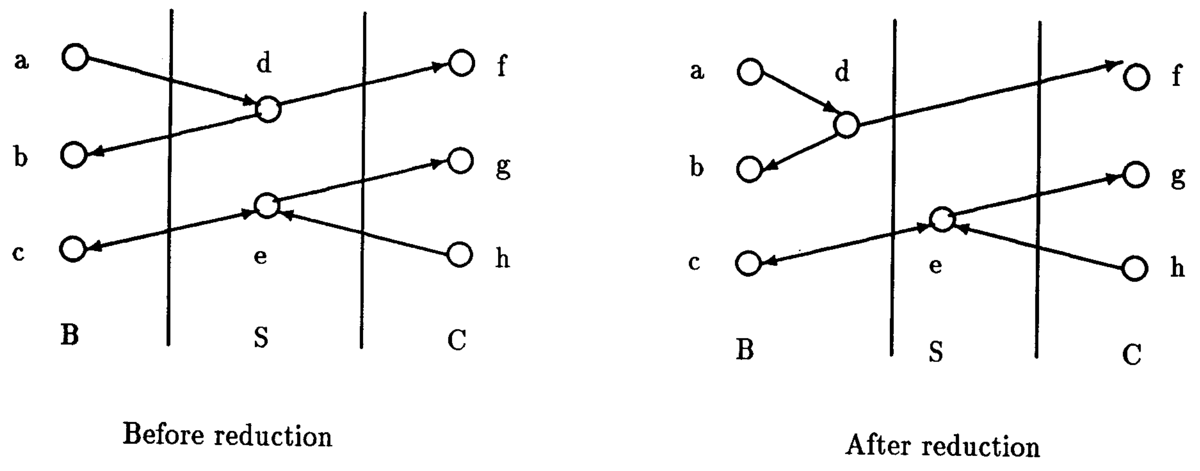


Figure 7: Reduction of the Separator Set

that are greater than a 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 $M = (A + A^t)$ and then removes the self-edges generated by the diagonal elements. Before the algorithm starts the dissection, it first examines the nodes to determine if any of the nodes 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, then the node is immediately placed into the border. 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. This limit is usually 7% of the nodes in the diagonal block.

After a separator set S has been produced by the version automatic nested dissection mentioned above has decomposed the graph G into a separator set S and two disjoint sets B and C the algorithm attempts to reduce the size of S by the following reductions:

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 7. In this example eight nodes were initially divided into three sets. The left set B contains the three nodes, a, b, and c. The separator set S has two nodes, d and e. The remaining three nodes, f, g, and h, are in the right set C. 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.

An optimization to the reduction algorithm involves moving nodes from B to C, or C to B, so that the first two reductions can be applied to nodes for which the conditions of the reductions were not met. 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 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, $\mid B - D \mid > T_{remain}$.

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

The second enhancement phase consists of moving a set of nodes from C to B together with applying the initial reduction techniques. A set of nodes $D \subset C$ is moved to B if the following conditions, symmetric with those above, 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, $\mid C - D \mid > T_{remain}$.

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

An example of this enhancement is provided in Figure 8. In this example none of the reductions may be applied to the initial separator set. However, the node f may be moved from set C to set B. After node f is moved to set B then the separator set may be reduced by moving the node d from set S into set 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}.

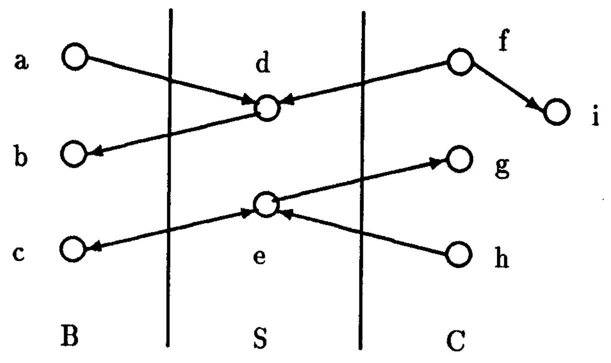
3.6 Results for H*

This section presents the results for the hybrid ordering H* that were collected on an Alliant FX/80. These results include border size, diagonal block sizes and performance results which include the ordering time.

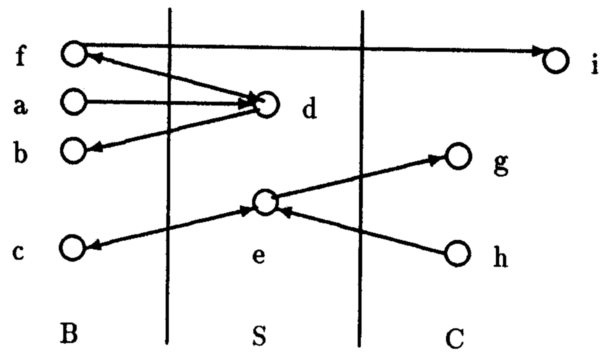
3.6.1 Test Matrices

The tests were conducted using matrices from the Harwell-Boeing test collection. All the matrices chosen were from the real, unsymmetric, assembled (RUA) test set. The RUA set used has 95 matrices, of which three are structurally singular and are not considered.

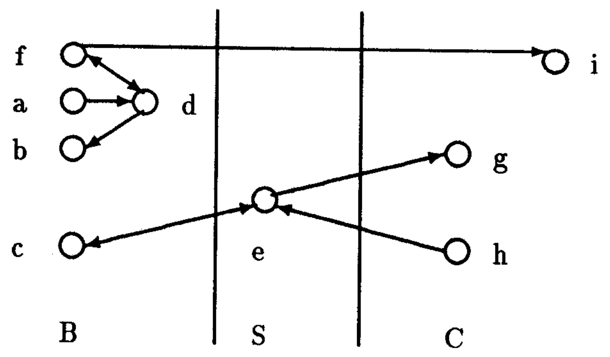
Because H* is meant to identify large grain parallelism, results will be presented only for the matrices which have at least 1,000 rows. Table 1 contains the name of the matrices, as well as the number of rows and number of nonzeros in each of the matrices. Also included in the table are the density for each matrix, calculated as the number of nonzero elements divided by the number of rows squared, and the average number of nonzero elements per row. The last value in the table is a measure of the symmetry in the structure of the matrix. The symmetry is calculated as the fraction of the elements of the matrix for which if $a_{i,j}$ is a nonzero element, then $a_{j,i}$ is also a nonzero element in the matrix.



Before reduction



After enhancement



After reduction

Figure 8: Enhanced Separator Set Reduction

Matrix Name	Number of Rows	Number of Nonzeros	Density	Elements Per Row	Symmetry
gaff1104	1104	16056	0.0131	14.5	1.0000
gemat11	4929	33185	0.0013	6.7	0.0017
gemat12	4929	33111	0.0013	6.7	0.0017
gre_1107	1107	5664	0.0046	5.1	0.1954
hwatt1	1856	11360	0.0032	6.1	0.9887
hwatt2	1856	11550	0.0033	6.2	0.9835
lns3937a	3937	25407	0.0016	6.4	0.8686
lns_3937	3937	25407	0.0016	6.4	0.8686
mahistlh	1258	7682	0.0048	6.1	0.0302
nnc1374	1374	8606	0.0045	6.2	0.8355
or678lhs	2529	90158	0.0140	35.6	0.0729
orsirr_1	1030	6858	0.0064	6.6	1.0000
orsreg_1	2205	14133	0.0029	6.4	1.0000
pores_2	1224	9613	0.0064	7.8	0.6614
saylr4	3564	22316	0.0017	6.2	1.0000
sherman1	1000	3750	0.0037	3.7	1.0000
sherman2	1080	23094	0.0197	21.3	0.6862
sherman3	5005	20033	0.0007	4.0	1.0000
sherman4	1104	3786	0.0031	3.4	1.0000
sherman5	3312	20793	0.0018	6.2	0.7803
west1505	1505	5445	0.0024	3.6	0.0020
west2021	2021	7353	0.0018	3.6	0.0039

Table 1: RUA Matrices with at Least 1000 Rows

Matrix Name	Transversal Bound	Transversal Time
gaff1104	1E+2	0.499
gemat11	1E+2	0.685
gemat12	1E+3	1.622
gre_1107	1E+1	1.595
hwatt1	1E+1	0.099
hwatt2	*	0.823
lns3937a	*	33.021
lns_3937	*	14.519
mahistlh	*	2.870
nnc1374	*	7.351
or678lhs	1E+2	1.476
orsirr_1	1E+1	0.058
orsreg_1	1E+1	0.120
pores_2	1E+4	2.516
saylr4	1E+1	0.191
sherman1	1E+1	0.045
sherman2	*	17.945
sherman3	1E+1	0.223
sherman4	1E+1	0.048
sherman5	1E+3	1.672
west1505	1E+6	7.771
west2021	1E+6	13.547

Table 2: H0 (Transversal) Statistics for Large RUA Matrices

3.6.2 Transversal Results

Table 2 contains the results for the application of the H0 portion of the ordering to the large matrices. The table contains the user process time in seconds on an Alliant FX/80, required to find the transversal and the transversal bound. 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,

$$\mid a_{i,i} \mid * \alpha \geq \max_{1 \leq j \leq n} \mid a_{j,i} \mid \tag{2}$$

for $1 \leq i \leq n$. When the tests were run, the initial value of α was 1E+1 and the maximum value of α was 1E+5. If the transversal could not be found for a given value, α was increased by a factor of ten,

$$\alpha = \alpha * 10 \tag{3}$$

until it became greater than the maximum value. If the maximum value was exceeded without finding a transversal, the transversal was tried without the bound. For some of the matrices the transversal bound is given as '*', this indicates that the H0 algorithm could not find a bounded transversal within the given limits and that H0 resorted to an unbounded transversal search.

3.6.3 Comparison of Orderings

Tests were performed on the large RUA matrices using six different combinations of the orderings. The combinations are:

H*: This test used the entire H* ordering, including the H0, Tarjan's, H1, and H2 orderings.

Tarjan: This test only used the H0 transversal and then Tarjan's ordering. This test was chosen because it represented the ordering used by MA28.

H2: This test only used the H0 transversal followed by the H2 ordering. This test was chosen to show the benefit of using Tarjan's and the H1 algorithm before applying the H2 algorithm.

No H0: This test calculated a transversal without a bound, and then applied the other orderings, Tarjan's, H1, and H2. This test was chosen to allow the comparison of the unbounded transversal with the bounded transversal of the H0 algorithm.

No H2: This test eliminated the H2 ordering. This test used the H0 transversal followed by Tarjan's and the H1 orderings. This test was chosen to show the effect of not having the H2 ordering.

Nored: This test uses the H* ordering with the H2 ordering replaced by a standard nested dissection algorithm for symmetric matrices applied to $A + A^T$ where A is any diagonal block. This test was chosen to allow the comparison of nested dissection with the enhanced dissection of the H2 algorithm.

The H0 ordering was run as described above. The threshold parameters used for H1 are $T_{long} = 5$, $T_{minb} = .05N$, $T_{maxb} = .75N$, and $T_{maxsep} = .05 newblock\_size$, where N is the dimension of the entire matrix. A value of $T_{remain} = .004N$ was used in H2. The parameter T_{done} was set to $.1N$. These values were obtained empirically with tests on the Harwell-Boeing matrices and can be easily adapted to other matrices.

The user process times, in seconds on the Alliant FX/80, required by the various orderings are presented in Table 3.

Table 4 contains the number of border rows for each of the different orderings with the large RUA matrices. The column labeled "Rows" indicates the number of rows in the matrix. When examining this table it is important to realize that having a small border is important, but that the ordering must also generate small diagonal blocks. Tarjan's ordering always produces zero border rows, and therefore is not included in the table, but the block sizes are often too large and/or nonuniform to be useful in a parallel processing environment. When looking at the border sizes, one must also examine the largest block size that resulted from the ordering, see Table 5.

The third table of results for the large matrices contains the size of the largest diagonal block after the ordering was applied. The block sizes are contained in Table 5. The column labeled "Rows" indicates the number of rows in the matrix. These blocks sizes can be combined with the border sizes from Table 4 to determine the effectiveness of the ordering in producing a bordered block upper triangular form.

When comparing the orderings by examining the size of the border and the size of the largest diagonal block, it can also be useful to look at the ratio between the two. Table 6

Matrix	H*	Tarjan	H2	No H0	No H2	Nored
gaff1104	2.668	1.076	2.319	2.441	1.242	2.640
gemat11	6.647	2.101	6.718	4.344	2.645	6.680
gemat12	6.619	3.025	7.449	5.054	4.013	6.524
gre_1107	3.473	1.830	3.193	3.145	1.913	3.264
hwatt1	3.411	0.568	3.218	3.321	0.837	3.199
hwatt2	3.605	1.289	3.780	2.768	1.600	3.483
lns3937a	39.359	33.860	40.629	10.176	34.444	39.389
lns_3937	22.352	15.595	22.482	9.881	16.121	22.278
mahistlh	4.054	3.212	4.373	1.617	3.243	3.883
nnc1374	9.650	7.714	9.439	2.920	7.803	9.510
or678lhs	12.036	4.468	13.526	10.007	5.110	10.466
orsirr_1	1.706	0.332	1.499	1.726	0.481	1.616
orsreg_1	4.381	0.692	4.060	4.304	0.938	4.110
pores_2	4.992	2.874	4.775	2.577	3.003	4.799
saylr4	6.902	1.121	6.061	6.691	1.803	6.664
sherman1	0.979	0.234	0.865	0.980	0.318	0.941
sherman2	21.054	18.783	21.367	4.051	18.871	20.633
sherman3	4.926	1.260	4.284	4.929	1.824	4.830
sherman4	0.803	0.254	0.720	0.815	0.315	0.776
sherman5	5.063	2.618	4.706	4.159	2.904	5.072
west1505	8.874	8.030	9.629	1.368	8.036	8.927
west2021	14.771	14.031	15.619	1.914	14.043	14.911

Table 3: Ordering Times for Large RUA Matrices

Matrix	Rows	H*	H2	No H0	No H2	Nored
gaff1104	1104	232	232	250	0	308
gemat11	4929	409	544	301	0	651
gemat12	4929	354	557	336	187	447
gre_1107	1107	337	337	320	0	545
hwatt1	1856	403	401	368	0	369
hwatt2	1856	405	435	476	75	417
lns3937a	3937	582	529	487	158	867
lns_3937	3937	494	525	499	0	794
mahistlh	1258	136	185	191	0	317
nnc1374	1374	226	226	219	0	449
or678lhs	2529	398	253	307	0	571
orsirr_1	1030	322	322	322	0	342
orsreg_1	2205	458	458	438	0	444
pores_2	1224	300	300	276	0	539
saylr4	3564	634	634	634	0	636
sherman1	1000	147	147	147	0	152
sherman2	1080	332	390	366	0	438
sherman3	5005	423	423	423	0	452
sherman4	1104	103	103	103	0	103
sherman5	3312	280	280	287	0	337
west1505	1505	95	117	104	0	400
west2021	2021	131	179	133	63	385

Table 4: Border Rows for Large RUA Matrices

Matrix	Rows	H*	Tarjan	H2	No H0	No H2	Nored
gaff1104	1104	94	460	110	88	460	79
gemat11	4929	488	4578	492	460	4578	444
gemat12	4929	476	4553	482	486	2496	474
gre_1107	1107	104	1107	104	109	1107	96
hwatt1	1856	101	1728	153	124	1728	124
hwatt2	1856	181	1792	128	128	1300	177
lns3937a	3937	364	3558	360	373	2477	264
lns_3937	3937	328	3558	338	319	3558	369
mahistlh	1258	115	589	125	124	589	100
nnc1374	1374	135	1318	136	134	1318	109
or678lhs	2529	252	1830	252	252	1830	252
orsirr_1	1030	101	1030	101	101	1030	100
orsreg_1	2205	196	2205	196	160	2205	160
pores_2	1224	106	1224	106	111	1224	100
saylr4	3564	333	3564	333	333	3564	333
sherman1	1000	65	681	100	65	681	63
sherman2	1080	102	870	101	95	870	90
sherman3	5005	394	2830	500	394	2830	394
sherman4	1104	87	546	110	87	546	87
sherman5	3312	254	1638	331	303	1638	260
west1505	1505	144	1099	147	144	1099	133
west2021	2021	187	1500	193	174	665	161

Table 5: Largest Diagonal Block for Large RUA Matrices

Matrix	Rows	H*	H2	No H0	No H2	Nored
gaff1104	1104	2.468	2.109	2.841	0.000	3.899
gemat11	4929	0.838	1.106	0.654	0.000	1.466
gemat12	4929	0.744	1.156	0.691	0.075	0.943
gre_1107	1107	3.240	3.240	2.936	0.000	5.677
hwatt1	1856	3.990	2.621	2.968	0.000	2.976
hwatt2	1856	2.238	3.398	3.719	0.058	2.356
lns3937a	3937	1.599	1.469	1.306	0.064	3.284
lns_3937	3937	1.506	1.553	1.564	0.000	2.152
mahistlh	1258	1.183	1.480	1.540	0.000	3.170
nnc1374	1374	1.674	1.662	1.634	0.000	4.119
or678lhs	2529	1.579	1.004	1.218	0.000	2.266
orsirr_1	1030	3.188	3.188	3.188	0.000	3.420
orsreg_1	2205	2.337	2.337	2.737	0.000	2.775
pores_2	1224	2.830	2.830	2.486	0.000	5.390
saylr4	3564	1.904	1.904	1.904	0.000	1.910
sherman1	1000	2.262	1.470	2.262	0.000	2.413
sherman2	1080	3.255	3.861	3.853	0.000	4.867
sherman3	5005	1.074	0.846	1.074	0.000	1.147
sherman4	1104	1.184	0.936	1.184	0.000	1.184
sherman5	3312	1.102	0.846	0.947	0.000	1.296
west1505	1505	0.660	0.796	0.722	0.000	3.008
west2021	2021	0.701	0.927	0.764	0.095	2.391

Table 6: Ratio of the Border Rows to the Largest Diagonal Block

contains the ratio of the number of rows in the border to the number of rows in the largest diagonal block.

Since the ordering attempts to provide the smallest diagonal blocks with the fewest number of rows in the border, it is useful to look at the sum of the number of rows in largest diagonal block and the border. Table 7 contains the fraction of rows that are in the largest diagonal block and in the border for each of the matrices,

$$f = \frac{\textit{border rows} + \textit{largest diagonal block rows}}{\textit{total rows}}. \tag{4}$$

The last row in the table contains the averages for each of the orderings. As can be seen from this table, the two best ordering strategies for producing the smallest diagonal blocks with the fewest number of rows in the border are the H* ordering and the No H0 ordering. For both of these ordering, on the average, approximately one-fourth of the rows in the matrix are in either the largest diagonal block or the border. By comparison, the Tarjan ordering has four-fifths of the rows in the largest diagonal block, on average (there is no border for the Tarjan ordering so all rows in this table for the Tarjan ordering are in the largest diagonal block).

Instead of just adding the border rows and diagonal block rows, the values could have been weighted by some α to place a greater significance on either the border or the largest

Matrix	H*	Tarjan	H2	No H0	No H2	Nored
gaff1104	0.295	0.417	0.310	0.306	0.417	0.351
gemat11	0.182	0.929	0.210	0.154	0.929	0.222
gemat12	0.168	0.924	0.211	0.167	0.544	0.187
gre_1107	0.398	1.000	0.398	0.388	1.000	0.579
hwatt1	0.272	0.931	0.298	0.265	0.931	0.266
hwatt2	0.316	0.966	0.303	0.325	0.741	0.320
lns3937a	0.240	0.904	0.226	0.218	0.669	0.287
lns_3937	0.209	0.904	0.219	0.208	0.904	0.295
mahistlh	0.200	0.468	0.246	0.250	0.468	0.331
nnc1374	0.263	0.959	0.263	0.257	0.959	0.406
or678lhs	0.257	0.724	0.200	0.221	0.724	0.325
orsirr_1	0.411	1.000	0.411	0.411	1.000	0.429
orsreg_1	0.297	1.000	0.297	0.271	1.000	0.274
pores_2	0.332	1.000	0.332	0.316	1.000	0.522
saylr4	0.271	1.000	0.271	0.271	1.000	0.272
sherman1	0.212	0.681	0.247	0.212	0.681	0.215
sherman2	0.402	0.806	0.455	0.427	0.806	0.489
sherman3	0.163	0.565	0.184	0.163	0.565	0.169
sherman4	0.172	0.495	0.193	0.172	0.495	0.172
sherman5	0.161	0.495	0.184	0.178	0.495	0.180
west1505	0.159	0.730	0.175	0.165	0.730	0.354
west2021	0.157	0.742	0.184	0.152	0.360	0.270
Average	0.252	0.802	0.264	0.250	0.746	0.314

Table 7: Fraction of Rows in the Largest Diagonal Block and the Border

Matrix	Rows	H*	Tarjan	H2	No H0	No H2	Nored
gaff1104	1104	192	184	83	192	184	192
gemat11	4929	372	352	20	372	352	377
gemat12	4929	416	377	31	402	392	416
gre_1107	1107	16	1	16	13	1	13
hwatt1	1856	146	129	24	146	129	148
hwatt2	1856	87	65	19	79	73	88
lns3937a	3937	383	351	36	371	361	385
lns_3937	3937	371	351	30	371	351	371
mahistlh	1258	681	670	106	678	670	681
nnc1374	1374	73	57	17	71	57	69
or678lhs	2529	1187	700	103	1100	700	1202
orsirr_1	1030	13	1	13	13	1	11
orsreg_1	2205	16	1	16	15	1	15
pores_2	1224	14	1	14	12	1	11
saylr4	3564	22	1	22	22	1	23
sherman1	1000	333	318	236	333	318	334
sherman2	1080	221	211	12	222	211	218
sherman3	5005	2119	2111	1620	2119	2111	2119
sherman4	1104	566	559	457	566	559	566
sherman5	3312	1681	1675	1351	1682	1675	1681
west1505	1505	426	405	27	424	405	441
west2021	2021	589	522	48	552	575	588

Table 8: Number of Diagonal Blocks for Large RUA Matrices

diagonal block,

$$p = \frac{(1-\alpha) \times \textit{border rows} + \alpha \times \textit{largest diagonal block rows}}{\textit{total rows}}. \tag{5}$$

Such a weighting could be used to represent the difference in the work done to the border rows and the diagonal block rows. The difference in work being the border rows will be used in a sequential updates and the factorization of a single block, where as the largest diagonal block will be factored in parallel with the other diagonal blocks.

The size of the border and the largest diagonal block are not enough to provide a complete understanding of how well an ordering is working. Other important factors are the uniformity and number of the diagonal blocks. Table 8 contains the number of diagonal blocks generated by each of the orderings.

One final set of values for comparing the different orderings is the average size of the diagonal blocks. This can be combined with the size of the largest diagonal block to determine if the ordering results in blocks near the maximum size or if a large number of blocks over a wide range of sizes are generated. The large number of blocks of size one, however, greatly influences the average size of the blocks. (These blocks of size one are generated by Tarjan's algorithm and appear in any of the orderings which include Tarjan's algorithm.) Therefore, the average block size will only be calculated for the blocks greater than a size of one. Table 9 contains the average block sizes.

Matrix	Rows	H*	Tarjan	H2	No H0	No H2	Nored
gaff1104	1104	57	231	61	56	231	51
gemat11	4929	245	4578	230	213	4578	206
gemat12	4929	149	4553	141	233	544	146
gre_1107	1107	54	1107	54	65	1107	46
hwatt1	1856	77	1728	80	96	1728	96
hwatt2	1856	91	1792	83	93	855	91
lns3937a	3937	67	120	125	65	104	55
lns_3937	3937	66	120	126	67	120	61
mahistlh	1258	41	589	17	49	589	33
nnc1374	1374	68	1318	71	73	1318	86
or678lhs	2529	79	1830	26	31	1830	152
orsirr_1	1030	54	1030	54	54	1030	62
orsreg_1	2205	109	2205	109	117	2205	117
pores_2	1224	71	1224	71	86	1224	62
saylr4	3564	172	3564	172	172	3564	171
sherman1	1000	38	228	48	38	228	37
sherman2	1080	48	870	57	45	870	71
sherman3	5005	247	1448	270	247	1448	244
sherman4	1104	55	546	61	55	546	55
sherman5	3312	194	1638	211	192	1638	185
west1505	1505	43	551	55	47	551	34
west2021	2021	77	1500	70	64	346	75

Table 9: Average Size of the Diagonal Blocks with Size Greater than 1

3.7 Observations

By examining the results presented in the previous section, several observations can be made about the effectiveness of the different orderings.

One problem with Tarjan's ordering is that many sparse matrices do not decompose into a large number of equal-sized strongly connected components. As a result, this ordering can result in large diagonal blocks. For the test matrices used, the largest diagonal block contained over half of the rows in the matrices for all but three of the twenty-two matrices. In five cases, the entire matrix was contained in one diagonal block.

In addition, Tarjan's ordering can result in a large number of diagonal blocks. Combining the size of the largest diagonal block with the number of diagonal blocks means that the ordering tends to generate a single large block and a large number of very small blocks. This disparity in block sizes can cause load balancing trouble for parallel processors, and indicates why Tarjan's ordering, by itself, produces orderings that are ill-suited for large-grain parallel processing by diagonal blocks.

The No H2 ordering attempts to enhance Tarjan's ordering by the addition of the H1 algorithm. The H1 ordering, however, only partially succeeded on four of the twenty-two matrices. During the four successes the size of the largest diagonal block was reduced between 28% and 44%. The No H2 ordering suffers from the same problems Tarjan's ordering, the large disparity in block sizes. Therefore, the conclusion about this ordering is the same as for Tarjan's ordering, that, by itself, it produces orderings that are ill-suited for large-grain parallel processing by diagonal blocks. Further work in this area has already shown that the effectiveness H1 ordering can be improved by removing rows which are above a specified density threshold as is done in the H2 ordering [Wan91].

The H2 ordering is applied recursively until the diagonal block size is reduced to a specified level. As a result, the size of the largest diagonal block for the H2 ordering may be selected when the test is run. Since the block size can be adjusted, the more important feature of the ordering is the size of the border. By comparison with the H* ordering, the H2 ordering results in a larger border for seven of the twenty-two matrices and a smaller border for only three of the matrices. A further comparison of the H2 and H* orderings show that the H2 ordering is slower for eleven of the matrices and faster for the other eleven. Therefore, since the H2 ordering produces the same or larger borders in roughly the equivalent amount of time as the H* ordering, the H* ordering would be considered better than the H2 ordering.

The Nored ordering eliminated the enhancements to the nested dissection within the H2 ordering. When comparing the size of the border from the Nored ordering to the H* ordering, the Nored ordering has a larger border for 19 of the matrices and a smaller border for only 2 of the matrices. This indicates that the enhancements to nested dissection within the H2 ordering do result in a smaller border.

The No H0 ordering used an unbounded transversal instead of the bounded transversal of the H* ordering. This resulted in the No H0 ordering being significantly faster than the H* ordering for eleven of the matrices, and being roughly the same or slightly faster for the other eleven. When comparing border rows, the No H0 ordering had a smaller border for nine of the matrices and a larger border for nine of the matrices. The comparison of the No H0 ordering and the H* ordering would seem to indicate that the No H0 ordering is better for its ability to produce similar results in less time. However, the main benefit of the H0 ordering is the stability of the ordering which can only be measured with the help of a solver. Therefore, the final comparison between the H* ordering and the No H0 ordering will be delayed until after the solver has been discussed.

As presented here, the H* ordering can do at least as good as the orderings that were combined to form the H* ordering when looking at the resulting border and diagonal blocks.

One factor that can cause difficulty for the H* ordering is the amount of the symmetry within the matrix. For the 6 matrices which have a symmetry < 0.1000, the average border size is 9.0%. However, for the other 16 matrices, with the symmetry > 0.1000, the average border size is 19.0%, with a maximum of 31%.

Another factor that appears to result in large borders is the density of the matrix. For the 3 matrices that have more than 10 elements per row the average border size is 22.6%. This appears to be true whether the matrix has symmetric structure or not.

The hybrid ordering H* succeeds in ordering matrices into bordered block upper triangular form with smaller diagonal blocks and a smaller border than the separate orderings that were combined to form the hybrid ordering. The difficulty within the ordering, however, is the handling of matrices with symmetric structure or those that are less sparse.

4 Stability Issues

4.1 General considerations

The major problem with a large grain parallel solver is maintaining the stability of the entire system while only working with pivot selection constrained to a particular subsystem, i.e., 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 nonsingular. The difficulties in addressing these issues have prevented tearing techniques from being employed 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 [Dav89]. 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 unsymmetric 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.

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 unsymmetric solvers. We will, however, allow the size of the border to increase during the factorization. In doing so we would also like to restrict any unsymmetric permutations (cf. definition 3.2) 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 unsymmetric 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* $(1,2,\cdots,i-1,i,i+1,\cdots,n) \rightarrow (1,2,\cdots,i-1,i+1,\cdots,n,i)$ *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:

```
$i = 1$
castnumb = 0
for $k = 1$ to $N$
    foreach $a_{ji}, j > i$
        if $p_{ii}$ is stable for $a_{ji}$
            then  eliminate $a_{ji}$
            else  cast $p_{ii}$ $(A \leftarrow perm(A))$
                  castnumb = castnumb +1
                  goto end
        endif
    endforeach
```

```
            $i = i + 1$
  end:
        endfor
        for $i = N-$ castnumb $+1$ to $N$
            find a stable pivot $p_{ii}$
            foreach $a_{ji}, j > i$
                eliminate $a_{ji}$
            endforeach
        endfor
```

The last set of nested loops in this procedure factors the diagonal block which relates the pivots which were cast during the initial part of the factorization. This portion of the procedure has been left vague intentionally since there is a certain amount of freedom as to how to finish the factorization. In general, unsymmetric pivoting within this block may be needed to guarantee the existence and stability of the factorization. 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. For example, if all of the diagonal elements are 0 all of the pivots are cast and the entire factorization is performed by the second phase of the procedure. This can be improved by allowing some unsymmetric (local) permutations to place a stable pivot on the diagonal and thereby reduce casting. This approach is particularly appropriate for bordered block upper triangular matrices and will be discussed in more detail later.

Applying the procedure above to a matrix A can be characterized algebraically as

$$L_c^{-1} P_c \left(S_N L_N^{-1} \cdots S_1 L_1^{-1} A S_1^T \cdots S_N^T \right) Q_c^T = U,$$

where

- U is an upper triangular matrix.
- L_k^{-1} is the elementary lower triangular matrix which performs the stable eliminations on the k-th iteration of the initial phase of the procedure.
- S_k is either I or the shift permutation needed when casting is done on the k-th iteration.
- P_c, Q_c, and L_c are the row interchanges, column interchanges and lower triangular factor produced in the second phase of the algorithm which are required for stability and existence of the factorization. Note that they have the following structure

$$P_c = \begin{pmatrix} I & 0 \\ 0 & \tilde{P} \end{pmatrix}, \; Q_c = \begin{pmatrix} I & 0 \\ 0 & \tilde{Q} \end{pmatrix}, \; L_c = \begin{pmatrix} I & 0 \\ 0 & \tilde{L} \end{pmatrix}$$

  where the order of $\tilde{P}$,$\tilde{Q}$, and $\tilde{L}$ is the number of pivots cast.

If we let,

$$\begin{aligned}
E_k^{-1} &= S_N \cdots S_k L_k^{-1} S_k^T \cdots S_N^T \\
P &= \prod_{k=N}^{1} S_k \\
E &= \prod_{k=1}^{N} E_k \\
Q &= Q_c P \\
T &= E P_c^T
\end{aligned}$$

then the decomposition can be written

$$PAQ^T = TL_cU.$$

In general, T is not a unit lower triangular matrix so the algorithm does not necessarily produce a standard triangular factorization of a reordered version of A. More of the structure of the factorization can be seen by considering a reorganization of the procedure which is mathematically equivalent.

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

As before, the last set of nested loops corresponds to the factorization of the diagonal block relating all of the cast pivots (possibly requiring an unsymmetric 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.

Let us consider k-th pass (from **begin** to **end** label) through the unresolved columns. If we assume that there are t unresolved columns considered, the action of this pass can be expressed as the application to the current updated form of A of the matrix

$$S_{t,k}E_{t,k}^{-1}\cdots S_{1,k}E_{1,k}^{-1},$$

where the $S_{i,k}$ are the shift permutation matrices and the $E_{i,k}^{-1}$ are elementary lower triangular matrices that perform the stable eliminations for the columns. This matrix can be rewritten as

$$\begin{pmatrix} L_k & B_k \\ 0 & \hat{L}_k \end{pmatrix} S_{t,k}\cdots S_{1,k}$$

with L_k a unit lower triangular matrix and $\hat{L}_k$ a unit lower triangular matrix of dimension castnumb. The zero entries in the lower left block arise because the elimination of the elements in the rows of cast pivots is delayed to the next pass. This fact enables us to define a permutation matrix P_k for this pass, representing the row permutation $(1,2,3,\cdots,N-\text{castnumb}+1,\cdots,N) \rightarrow (N-\text{castnumb}+1,\cdots,N-1,N,1,2,3,\cdots,N-\text{castnumb})$ such that

$$P_k\begin{pmatrix} L_k & B_k \\ 0 & \hat{L}_k \end{pmatrix} P_k^T = \tilde{L}_k^{-1}$$

with $\tilde{L}_k$ a unit lower triangular matrix. Note that the first algorithm does not delay these eliminations and the structure cannot be exploited. Let $\hat{S}_k = \prod_{i=t}^{1} S_{i,k}$ then the action of the k-th pass is to apply the matrix

$$P_k^T\tilde{L}_k^{-1}P_k\hat{S}_k$$

Let m be the number of passes through the unresolved columns needed to complete the first phase. Then the procedure can be characterized algebraically as follows.

$$L_c^{-1}P_c\left(P_m^T\tilde{L}_m^{-1}\tilde{C}_{m-1}^T\tilde{L}_{m-1}^{-1}\cdots\tilde{C}_1^T\tilde{L}_1^{-1}P_1\hat{S}_1AC_1^T\cdots C_m^T\right)Q_c^T = U,$$

where

- U is an upper triangular matrix.
- $\tilde{L}_k^{-1}$ is as described above.
- C_k is either I or the permutation that casts all of the pivots marked for casting in the k-th pass, i.e., $C_k = \prod_{i=1}^{t} S_{i,k}$, where the $S_{i,k}$ are defined above for the k-th pass.
- $\tilde{C}_k^T = P_{k+1}\hat{S}_{k+1}P_k^T$ for $k > 1$.
- P_c, Q_c, and L_c are the row interchanges, column interchanges and lower triangular factor produced in the second phase of the algorithm which are required for the stability and existence of the factorization. They have the same structure as those produced for the first procedure but are not necessarily the same matrices.

Let $\hat{Q}^T = C_1^T \cdots C_m^T Q_c^T$, $\tilde{C}_m = P_m P_c^T$ and $\hat{P} = P_1 \hat{S}_1$ then the factorization produced by this form of the algorithm is

$$\hat{P} A \hat{Q}^T = \tilde{L}_1 \tilde{C}_1 \cdots \tilde{L}_{m-1} \tilde{C}_{m-1} \tilde{L}_m \tilde{C}_m L_c U.$$

It can be seen from this form that although a standard LU factorization is not produced the resulting decomposition can be expressed as a series of m matrix pairs each comprising a unit lower triangular matrix and a permutation matrix, followed by a single unit lower triangular matrix and finally an upper triangular matrix. The number of these pairs, m, could be as large as the dimension of the original matrix A. However, as is seen below, the amount of casting and, therefore, the number of passes required, can be considerably reduced when a suitable implementation is used.

Since casting is not allowed in the second phase of both algorithms above, the following observation can be made,

Proposition 4.1 *For every column in the resulting factorization at most two pivot elements are used to eliminate all of the nonzero elements.*

The first pivot is the diagonal element that is compared against all of the elements that it is used to eliminate. The second pivot choice is made during factorization of the diagonal block relating the cast pivots. The use of an unsymmetric permutation to factor the block allows the consideration of the size of any other elements in the column which must be eliminated when selecting the pivot element. There are two main benefits from this approach. We avoid the common problem in tearing techniques of using a pivot to eliminate elements whose relative size has not been examined (and therefore not controlling the size of the elements in the factors). Also, we avoid having to make a number of comparisons and possible interchanges per column that is proportional to the number of elements to be eliminated (which can improve the efficiency of implementation). In comparison, pairwise pivoting can make up to $n/2$ comparisons to eliminate the first column of a dense matrix.

Given Proposition 4.1, if, in both phases, the condition for a pivot to be considered 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.

Proposition 4.2 *The growth factor for the factorization algorithms above applied to a dense matrix $A \in \Re^N$ 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 [Bun74, Gea75]. 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.

5 MCSPARSE Factorization/Solve Description

5.1 Stability control

In this section, the techniques used to control the stability of the factorization are discussed as well as the incorporation of the casting into MCSPARSE.

5.1.1 Global considerations

When factoring a bordered block upper triangular matrix, a pivot may be cast during the parallel factorization of the diagonal blocks or during the elimination of the border. In both cases, the completion of the elimination of the pivot column is delayed until the factorization of the diagonal border block. For example, if for column i all of the subdiagonal elements in the diagonal block were eliminated and the pivot is cast during the elimination of the elements in the border in column i then the nonzero elements in the border are permuted to the upper triangular part of the border diagonal block. Note that these elements do not then require elimination unless further casting or unsymmetric pivoting during the factorization of the border diagonal block move them back into the lower triangular part of the matrix. During the diagonal border block factorization no casting is allowed. This deviates from the description above. There casting is prohibited for pivots selected in a row or column that contains another element that was selected as a pivot and cast earlier in the factorization. In MCSPARSE this prohibition is extended to include rows and columns which are placed in the border by H*. This modification does not affect the conclusions of the previous sections. In order to differentiate between casting performed during the elimination of elements in the diagonal blocks and casting performed during the elimination of elements of the border or elements of rows placed in the border due to casting an earlier pivot, the former will be referred to as diagonal casting and the latter border casting.

In the remainder of this section we discuss the local pivoting strategy and the tests used to determine whether or not a pivot is to be cast to the border.

5.1.2 Local pivoting

As mentioned above, it is possible to use local unsymmetric pivoting to reduce the amount of casting required in the first phase of the algorithm. Furthermore, it is necessary to use unsymmetric pivoting in the factorization of the border diagonal block to ensure the existence and stability of the overall factorization. MCSPARSE exploits the bordered block upper triangular structure and performs unsymmetric permutations in each of the diagonal blocks to select locally stable pivots and reduce the amount of casting required. A locally stable pivot is one which is within a specified factor of the maximum magnitude of elements of the diagonal block in the pivot row or column. MCSPARSE uses the test within the pivot column, i.e., for some $\gamma \leq 1$,

$$| a_{i,j} | \geq \gamma * \max_{k \in Active} (| a_{k,j} |). \tag{6}$$

The global stability of the factorization is then ensured by the border casting and the unsymmetric pivoting in the border diagonal block factorization. These unsymmetric permutations are also used to control fill-in in the diagonal blocks. A modified version of the Markowitz count is used to assess the fill-in potential of an element. The stability and sparsity constraints are combined by choosing the element with the smallest modified Markowitz count that is also a locally stable pivot. (See Section 5.3 for implementation details.)

The border diagonal block can be treated as a dense or sparse matrix as its size warrants. When it is treated as dense partial pivoting is used and when it is treated as sparse a pivot search similar to the diagonal blocks is used.

5.1.3 Diagonal casting

Since a local unsymmetric permutation is used within each of the diagonal blocks during the elimination of their subdiagonal it is always possible, with one exception, to determine a locally stable pivot element. The one exception is when a row or column of all 0's is created in the portion of the matrix being eliminated. In this case, the pivot must be cast and the elimination of the column is completed in a stable manner during the border diagonal block factorization. (As a result, the singularity of the diagonal blocks is irrelevant when casting is used.) Even though casting is not required when a nontrivial row or column is present, there are some situations that it is very likely that the locally stable pivot will be cast during the elimination of the border. It is more efficient to anticipate this border casting and mark the pivots during the elimination of the diagonal blocks. Two tests are used for the anticipatory diagonal casting.

If the locally stable pivot selected is *small* it is cast to the border since it is very likely that it will not satisfy the stability requirements needed to use it to eliminate the elements in the column that are in the border. Specifically, if the test

$$| a_{i,j} | \quad < \quad \alpha \tag{7}$$

is true where $a_{i,j}$ is the selected pivot, then column j and row i are permuted to the border. Recall that the potential pivot element has already been chosen to be within a certain factor of the largest element in the column so this test assesses the overall magnitude of the potential pivot column. Of course, the threshold value used is dependent upon the matrix. The threshold could be calculated as a fraction of a matrix norm such as $\|A\|_\infty$ or $\|D_i\|_\infty$. For matrices where the size of the largest elements are 10^2 or 10^3 the experiments discussed later show a threshold of 10^{-5} to be reasonable. The effects of different thresholds are in the parameter study in Section 6.1.

Even though the pivot passes the first test it still may be possible to anticipate border casting. If the border elements in the column were already updated with all appropriate previous pivots the border casting decision could be made. Of course, this is impossible since the border elimination takes place after the factorization of the diagonal blocks. The original elements of the border are available, however, and the second test for diagonal casting compares the absolute value of the selected pivot with the maximum norm of the original border elements in the pivot column. If the test

$$| a_{i,j} | \quad < \quad \beta * \max_{k \,\in\, border} (| a_{k,j} |), \tag{8}$$

where $a_{i,j}$ is the selected pivot, is true column j and row i are permuted to the border. Of course, border casting may still result due to updates to the original border elements and fill-in in the border.

5.1.4 Border Casting

Border casting involves casting a pivot element during the elimination of the border because the relative difference between the magnitude of the border element and pivot element is too large. The test

$$| a_{j,j} | \quad < \quad \epsilon * | a_{k,j} | \tag{9}$$

is checked before pivot $a_{j,j}$ is used to eliminate border element $a_{k,j}$. If the test is true, then pivot $a_{j,j}$ is cast to the border.

A major difference between diagonal and border casting is that in the former, the pivot element is cast before being used to eliminate any elements, in the latter this is not the case. In border casting, the cast pivot element has already been used to eliminate elements within the diagonal block and it may have been used to eliminate elements within other border rows. It is this casting of a pivot that has already been used to eliminate some elements that prevents the production of an LU factorization of A. For example, consider rows i and m within the same diagonal block. Suppose row i is used to eliminate an element in row m within the diagonal block, and then is permuted to the border during the border elimination. Now suppose that after row i is cast, row m must be used to eliminate one of the new border elements in row i. As a result, row i has a multiplier from row m and row m has a multiplier from row i which precludes an LU factorization. This situation is similar to that encountered when using pairwise pivoting to maintain stability and parallelism.

5.2 Fill-in control

In the present implementation of MCSPARSE, during the first stage of the solver a processor has access only to the information for the particular diagonal block that it is factoring at the time, i.e., the application to the off-diagonal block C_i of the transformations which eliminate subdiagonal elements in D_i are delayed. Therefore, a method to control fill-in had to be developed which when applied within the diagonal blocks would also effectively control the fill elements in the entire matrix. This involves the use of estimates of the effect of local decisions on the global amount of fill-in similar to that used in a priori methods.

MCSPARSE relies on a modification to the Markowitz count to control fill-in. The Markowitz count for an element $a_{i,j}$ is the product of the number of elements in the row and the column, each minus one.

$$\mu_{i,j} \quad = \quad | \, row_i - 1 \, | * | \, column_j - 1 \, | \tag{10}$$

These counts must be updated throughout the course of the factorization. In the case of MCSPARSE, such information can be maintained within the diagonal blocks but not across the entire matrix. A modified version of the Markowitz count, which exploits some information on the matrix outside the diagonal block, was derived to control fill-in when eliminating the diagonal blocks . When calculating the Markowitz count for an element, the row count is modified to include estimates of the number of elements within the part of the row that are in the off-diagonal blocks and similar estimates of the number of elements in the border of a column are used to modify the column count.

The first modification to the Markowitz count was to include the number of elements in the off-diagonal portion of the row. However, to control the effect of the off-diagonal elements a scaling factor is applied to the off-diagonal row count before it is added to the diagonal row count.

$$row\_len_i \quad = \quad row\_diag_i + row\_mult * row\_off\_diag_i \tag{11}$$

This scaling factor allows the off-diagonal elements to be weighted to provide a smaller or greater influence in the pivot selection.

A problem with the addition of the count of the off-diagonal elements is that the update of the elements in the off-diagonal portion of the row is not being performed. Therefore, the count of elements cannot being updated to exactly account for the application of the previous pivots. An enhancement to the off-diagonal count was to increase the off-diagonal count for a row each time the row was modified. The off-diagonal count is increased by the off-diagonal count for the pivot row, scaled by some factor.

$$\mathit{row\_off\_diag}_i = \mathit{row\_off\_diag}_i + \mathit{ofactor} * \mathit{row\_off\_diag}_{pivot} \qquad (12)$$

The most pessimistic estimate of fill is to use an *ofactor* of 1 which assumes that all of the elements in the pivot row form new fill elements in the target row. If an *ofactor* value of 0 is used then this assumes that the pivot row generates no new fill elements in the target row. The value of the off-diagonal count was also constrained to prevent it from becoming larger than the maximum possible off-diagonal count for the row. This constraint was used to prevent the off-diagonal count from overshadowing the diagonal block row count within the modified Markowitz count.

The second modification to the Markowitz count was the addition of the number of elements in the column within the border rows. These elements were included in an attempt to limit the number of fill elements generated during the border update. As with the row counts, a scaling factor is applied to the border column count when it is added to the diagonal block column count.

$$\mathit{col\_len}_j = \mathit{col\_diag}_j + \mathit{col\_mult} * \mathit{border\_col\_diag}_j \qquad (13)$$

This scaling factor allows the border column elements to be weighted to provide a smaller or greater influence in the pivot selection.

The addition of the border column elements has the same problem as the addition of the off-diagonal column elements, the count is not being updated during the diagonal block factorization. This is handled in a manner similar to the row counts. The border column count is increased by the border column count for the pivot column, scaled by some factor.

$$\mathit{border\_col}_j = \mathit{border\_col}_j + \mathit{bfactor} * \mathit{border\_col}_{pivot} \qquad (14)$$

As above, the most pessimistic estimate of fill is to use an *bfactor* of 1 which assumes that all of the elements in the pivot column form new fill elements in the target column. If an *bfactor* value of 0 is used then this assumes that the pivot column generates no new fill elements in the target column. The value of the border column count is also constrained to prevent the count from becoming larger than the number of rows in the border.

5.3 Implementation

In this section, we discuss the present implementation of MCSPARSE for the Cedar multiprocessor [Sta91, KDLS86, Yew86]. It is assumed that both the matrix A and the right-hand side vector b of the system to be solved are available. The solver is composed of two phases. The first phase calculates the factorization of the reordered system A; and the second phase calculates the solution of $Ax = b$ based on the factorization. Though these are two separate phases logically, the implementation of the solver discussed exploits the fact that they can be overlapped.

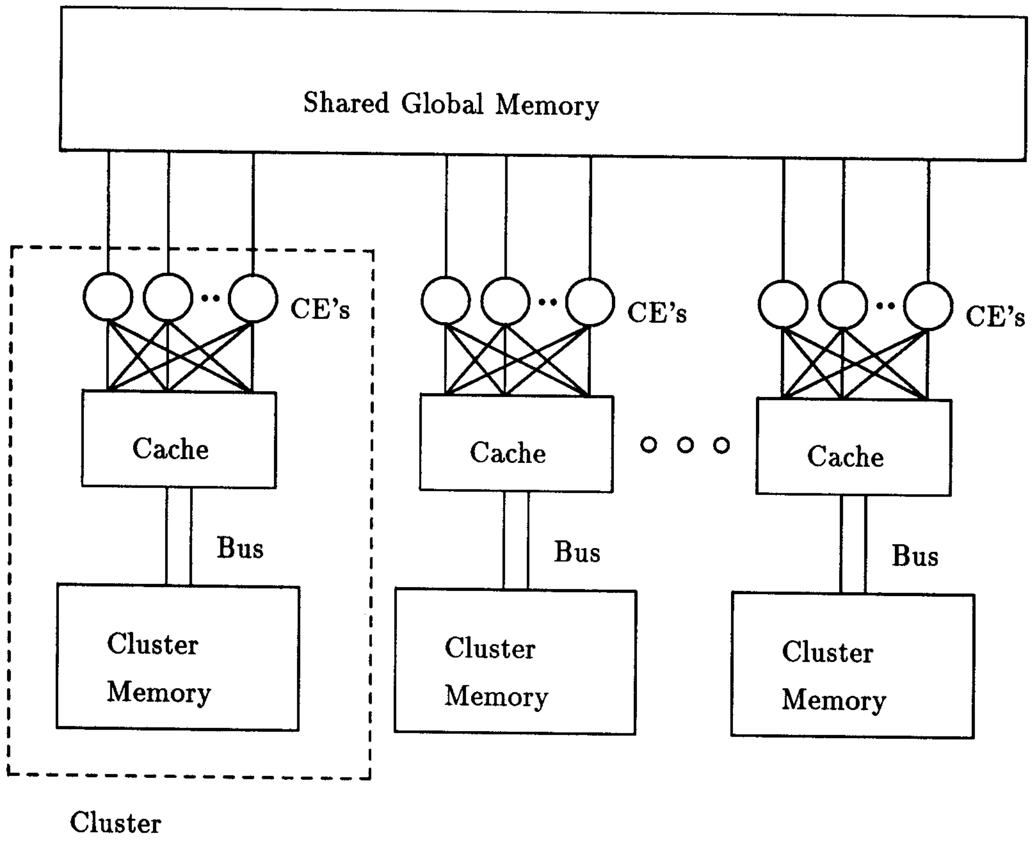


Figure 9: Cedar Architecture

The factorization of the matrix is performed by first eliminating all of the subdiagonal elements in the diagonal blocks D_i. Due to casting, the sizes of the diagonal blocks may change from those produced by H*. For simplicity, we will refer, incorrectly, to the elimination of subdiagonal elements and attendant casting as the calculation of an LU factorization of the diagonal blocks. Each off-diagonal block, C_i, is then updated with the transformations that eliminated the subdiagonal elements of the associated diagonal block D_i. The forward solve for the diagonal blocks is performed next. The next stage is the update of the border rows by the diagonal blocks. This stage requires synchronization as each border row (or block of border rows) must be passed sequentially through the diagonal blocks to be updated. When the border rows have all been updated, the factorization of the border diagonal block is calculated using one or more clusters depending on the size of the system. The forward and backward solve for the border rows are then performed. The final step is the backward solve which implies some synchronization.

5.3.1 Cedar

The Cedar architecture is a cluster-based multivector processor. It comprises a small number of clusters, presently four, connected to a shared global memory via a two stage Omega network. A diagram of the architecture is in Figure 9.

Each cluster has a small number of vector processors (≤ 8) which share an hierarchical cluster memory. The clusters are Alliant FX/8's modified to allow access to Cedar's shared global memory. The processors within a cluster are connected by a concurrency control bus that allows fast synchronization and is used for medium and fine grain parallelism. The first level of the cluster memory system is a 4-way interleaved, hardware-managed cache shared by the processors within a cluster. The cache is connected to the larger cluster memory via a high-speed bus.

Each processor is connected to the global interconnection network via a private port and has a prefetch unit which allows block fetches from global memory to offset the latency of the network. The interconnection network between the processors and the global memory consists of two unidirectional Omega networks (one network to memory the other from memory). All data communication between clusters is through the global memory.

Efficient use of this architecture requires the exploitation of large grain parallelism across clusters as well as medium and fine grain parallelism within each cluster. Due to the hierarchical nature of the memory system, careful consideration must be given to data placement and the exploitation of data prefetch and data locality are crucial performance considerations.

5.3.2 Description of Factorization Phase

The factorization algorithm is presented in Figure 10.

```
Concurrent loop over diagonal blocks
    Factor diagonal block
Concurrent loop over off-diagonal blocks
    Update off-diagonal block
Loop over border blocks
    Loop over diagonal blocks
        Update border block with diagonal block
Factor border diagonal block
```

Figure 10: Factorization Algorithm

Due to the structure of the matrix the factorizations of the diagonal blocks are independent and may proceed in parallel. An outline of the algorithm for a single diagonal block factorization is provided in Figure 11.

The first step in the algorithm is the initialization of the data structures for the factorization. When the ordering is successful the diagonal blocks are relatively small and they can be stored in a dense array during their factorization. Of course, the factorization still exploits the fact that the matrices are sparse to reduce the amount of work. It is straightforward to replace the factorization routine with one that also uses a more conventional sparse storage scheme when the diagonal blocks are large enough to warrant it. The size of the diagonal blocks is also used to classify each as to whether or not an entire cluster should be used to produce its factorization. During the factorization of the diagonal blocks, a particular cluster processes the blocks assigned to it in two passes. The first pass processes in parallel all of the diagonal blocks that are classified as needing only

```
Initialize the data structures
Loop over the number of rows
        Find best pivot element
        If pivot bad then
                Cast pivot
        Else
                Remove pivot from data structures
                Loop over remaining rows
                        Apply pivot to row
        End if
Extract factorization
```

Figure 11: Diagonal Block Factorization Algorithm

a single processor. The second pass handles in turn each of the blocks that were classified as requiring the entire cluster.

The nonzero elements are placed in the dense array according to the row and column number for each element. In addition, the column indices of all the nonzero elements within a row are saved and the row indices of all the nonzero elements within a column are saved. These indices for the rows (or columns) will allow the nonzero elements within a row (column) to accessed without scanning the row (column) searching for the nonzero elements.

After all the elements are placed in the diagonal block, the lists of the active columns and the active rows are initialized to include all the columns and rows in the diagonal block respectively. Also, an array of linked lists is created such that each row (column) is placed in the list that contains rows (columns) of the same length. Included in the length of the column are the elements within the diagonal block and the number of elements in the border that were in the column at the start of the solver. Similarly, the length of the row is the sum of the number of elements in the row in the diagonal block and the number of elements that are in the off-diagonal portion of the row. (More details on the calculation of the row lengths and the column lengths are provided in Section 5.2.)

After the initialization, the algorithm enters a loop where in each iteration of the loop it attempts to find a pivot element and apply the pivot element to the remaining portion of the block. The search is made to find a pivot that will preserves the stability and the sparsity of the factorization. In order to preserve the sparsity of the system, the pivot search calculates Markowitz counts using the modified row and column counts discussed earlier. The search looks for the element with the smallest Markowitz count that also satisfies a stability bound. The stability bound requires that the absolute value of the pivot element, $a_{i,j}$, be within some $\gamma \leq 1$ times the maximum absolute value within the column, cf. Equation (6).

To speed up the search for the smallest Markowitz count, the rows and columns are examined in increasing order by length. First the elements in all rows of length one will be examined. Next the elements in all columns of length one will be examined. This will be followed by all rows of length two and then all columns of length two. This will continue until it is known that all elements that have not been examined must have a larger Markowitz count. The minimum Markowitz count for the unexamined elements in rows or columns of length j is $(j-1)*(j-1)$ (since the search is in increasing order of length). Therefore, if $(j-1)*(j-1)$ is greater than the minimum Markowitz count that was already found there is no need to continue the search. This pivot search strategy is modeled after MA28.

In the case that two possible pivot elements are found to have the smallest Markowitz count and both meet the stability bound, then the element with the absolute value closest to the maximum absolute value within the column is chosen as the pivot element.

If the search fails to find a pivot element then the columns that remain in the active list of columns are cast to the border (as well as the rows remaining in the active list of rows.) During the diagonal block factorization the row and column are only marked to be cast. The actual transfer of the elements to the border will occur during the update of the border blocks.

If a pivot element is found, the diagonal casting checks are applied. If the pivot element fails these checks then the row and column containing the pivot element are marked for casting to the border and they are removed from the active lists. The border column count for each of the columns where there is an element must be updated. The linked lists that group together rows and columns with the same number of elements are also updated.

When a pivot element is accepted, the row permutation and column permutation are updated to save the order of the pivots and the linked lists and active row and column lists are altered. Before the pivot row is applied, the number of fill-in elements in the diagonal block portion of each target row is calculated and an estimate of the number of fill-in elements in the off-diagonal portion of the row is made as discussed in Section 5.2. The row is then moved to the correct linked list for its new element count. After the pivot row has been applied to all the target rows, the column counts of the columns that have been updated are recalculated and the columns moved to the correct linked lists.

As with the diagonal blocks, the updates of the off-diagonal blocks, C_i, are independent and can be performed in parallel. A loop is made, in the appropriately permuted order, over all the rows in the block, updating them with nonzero portion of L for the corresponding row in the diagonal block and the appropriate off-diagonal rows, that have already been processed. The algorithm for this update is presented in Figure 12.

```
Loop over all rows
    Loop over all previous rows
        Update row with previous row
```

Figure 12: Off-diagonal Block Update Algorithm

At step i in the algorithm, the i^{th} pivot row is found, row l. For all the nonzero L elements in the diagonal portion of row l, the off-diagonal portion of row l is updated by the L element and the corresponding row. If the nonzero L element is in column j, then row l is updated as follows,

$$\begin{aligned}&\forall k \in \mathit{offdiag}_j\\ &\qquad \mathit{offdiag}_{l,k} = \mathit{offdiag}_{l,k} - (L_{l,j} * \mathit{offdiag}_{j,k}). \qquad (15)\end{aligned}$$

Each row within the border must be updated using the diagonal and off-diagonal blocks to eliminate the elements in the row corresponding to the pivot elements in the diagonal blocks. A border block consists of several rows within the border grouped together for the sake of efficiency. The updates to the border blocks are independent and can proceed in parallel.

The border updates eliminate the elements in the border blocks corresponding to the pivot elements in the diagonal blocks. These updates by the diagonal blocks must be performed in the order of the diagonal blocks to correctly eliminate the elements in the border blocks. This means that before a border block can be updated by D_i it must have already been updated by all D_j, $1 \leq j \leq i - 1$ such that the present updated form of the row has a nonzero element in at least one of the columns in D_j. If the minimum column number for all the rows in border block i is column j, and if column j is in diagonal block k, then the first diagonal block that needs to update border block i is diagonal block k. Diagonal blocks 1 through $k - 1$ do not need to eliminate any of the elements in the border block. This allows the minimum column indices for the border blocks to be used to determine the set of border blocks whose updates can be started in parallel. The algorithm for the update of the border block is provided in Figure 13

```
Retrieve border block to be updated
While diagonal block on this cluster
        or in global memory
        Retrieve diagonal block for update
        Retrieve off-diagonal block for update
        Concurrent loop over rows in border block
                Update border elements with diagonal block
                Update border elements with off-diagonal block
        Cast all necessary rows/columns into border block
Send border block to next cluster
```

Figure 13: Border Block Update Algorithm

In order to update border block B_j by diagonal block D_k, a loop is made over all the rows in border block B_j. For each row i in block B_j the following actions are performed. The row i is separated into two parts, the diagonal portion corresponding to the elements in the columns in diagonal block D_k and the off-diagonal portion consisting of the elements in the remaining columns. A loop is then made over the pivot columns in diagonal block D_k, in the order in which the pivots were applied in the diagonal block. The first pivot is applied, if necessary, followed by the second pivot, and so forth. The pivot is applied only if there is a nonzero element in the border row in the same column as the pivot element. If the k^{th} pivot is column j and there is a nonzero element in row i in column j, then the element $a_{i,j}$ must be eliminated by the k^{th} pivot row, row m,

$$\begin{aligned} \mu_{i,j} &= a_{i,j}/a_{m,j} \\ \forall n \in U_m &\neq j \\ a_{i,n} &= a_{i,n} - \mu_{i,j} * a_{m,n}. \end{aligned} \tag{16}$$

The values of the multipliers $\mu_{i,j}$ are also saved. During these updates the border casting checks are made. Once a diagonal block row is marked for border casting it is no longer used for updates. After the entire diagonal block portion of the border row is eliminated, the off-diagonal portion of the row is updated using the values of the saved multipliers.

Once all of the rows in the border block have been eliminated, all rows within the diagonal block that were marked for casting are actually cast to the border block. The rows cast to the border block may contain elements that must be eliminated by the remaining rows in the diagonal block. Therefore, the border block is sent through the border update algorithm a second time with the same diagonal block.

After a border block has been updated with no rows being cast during the update, the diagonal block is checked to see if any rows cast during the diagonal block factorization have not been moved to a border block. The cast rows will now be added to the border block and the rows will be marked as having been moved so that no other border updates will try to move the rows into a border block (requiring synchronization). Since the diagonal block factorization correctly eliminated all the elements in the cast rows corresponding to the diagonal block, it is not necessary to send the border block back to be updated again after these rows have been cast to the border block. The border block is now ready to be updated by the next diagonal block.

After all of the border blocks have been updated by the appropriate diagonal blocks, the factorization of the border diagonal block may be performed. When the border diagonal block is treated as dense, this factorization is a BLAS3-based block LU decomposition with partial pivoting. Multiple clusters, and therefore global memory, may be used depending on the size of the block. The algorithm for this factorization is in Figure 14.

```
Loop over rows in border diagonal block
    Factor next $\omega$ columns with partial pivoting
    Update remaining portion of the $\omega$ rows
    Update remaining portion of border diagonal block
        using the $\omega$ columns and $\omega$ rows
```

Figure 14: Border Diagonal Block Factorization Algorithm

5.3.3 Description of Solution Phase

The method for calculating the solution of the system is in part determined by the structure of the matrix, in a similar manner as the structure determined the factorization. With the independent diagonal blocks, the forward solves for the diagonal blocks can be done concurrently. After the forward solves for the diagonal blocks, the forward solve for the border rows is performed, followed by the backward solve for the border rows. The subsequent backward solve of the diagonal blocks must be performed sequentially, starting from the last diagonal block and working towards the first diagonal block. (This portion of the implementation can be improved by exploiting some of the finer grain structure of the matrix U.)

The algorithm for the solution of the matrix is in Figure 15.

The algorithm for the forward solve is in Figure 16.

```
Concurrent loop over all diagonal blocks
    Forward solve with diagonal block
Forward solve with border
Backward solve with border
Loop over all diagonal blocks in reverse order
    Backward solve with diagonal block
```

Figure 15: Solution Algorithm

```
Loop over all rows
    Update rhs with diagonal block elements
    Update rhs with off-diagonal block elements
```

Figure 16: Forward Solve Algorithm

Since it is possible for a diagonal block row that was used as a pivot row during the diagonal block factorization to be cast during the border block updates, the present version of the solver opts for simplicity and performs the forward solve of the row before the row is cast. As a result, in the present implementation, the diagonal block forward solve algorithm is applied before the border block update algorithm.

After the forward solves have been calculated for all the diagonal blocks, and the factorization of the diagonal blocks is complete, the forward solve for the border rows may be performed. The backward solve for the border and for the diagonal blocks follow using similar algorithms.

5.4 Load balancing

The efficiency of the parallel factorization and solver is directly related to the effectiveness of obtaining a well balanced load for the different processors. The first part of the problem is to distribute the diagonal blocks for an even distribution of work during the diagonal block factorization and the off-diagonal updates. The second part of the problem is to keep all of the clusters busy during the update of the border blocks.

The first part of the problem can be solved with an even distribution of work to the clusters. However, as fill-in is controlled by heuristics, the amount of work to be performed for the elimination of a diagonal block is not exactly predictable. In practice a reasonable work estimation for the diagonal block factorization can be realized, because the amount of work does not depend on the position of the diagonal block in the system and most of the diagonal blocks will exhibit a banded structure confining fill-in within each band. This latter property is caused by the H2 phase of the ordering H*, which relies on levelization of the associated graph thereby creating banded subsystems (see Section 3.5).

A suitable load balance during the border update is much more difficult to realize. First, contrary to the diagonal block factorizations, the tasks to be performed are dependent upon each other, see previous section. Second, because of the distribution of data, the distribution of the parallel tasks for the diagonal block factorizations will automatically induce a distribution of work for the border update. This is caused by the fact that, whenever a cluster (processor) finishes its task of factoring a particular diagonal block, keeping the L and U factors in cluster memory of this cluster will prevent unnecessary data movement. However, a block of the border can only be updated by its corresponding diagonal block. So, in order to minimize data movement between the different clusters a border block has to be updated by the cluster which holds the corresponding diagonal block factors. Another problem related to the minimization of data movement is caused by the fact that when a diagonal block finishes the update of a border block the border block must then be updated by the next consecutive diagonal block. If the next diagonal block is on the same cluster, then the border block can be left in cluster memory and updated again. However, when the next diagonal block is assigned to another cluster, the border block must by moved into global memory so that the other cluster can copy it into its cluster memory. So, it is preferable to have adjacent diagonal blocks assigned to the same cluster. Thirdly, the amount of work involved for the update of a particular block of the border is dependent on the position of that block in the border. For instance, for the rightmost columns of a border block the fill-in generated during the elimination of all previous row elements will be substantial, while for the initial columns in the block this is not the case. Further, the staircase form of the border causes the position of the diagonal block within the matrix to determine the number of border rows the diagonal block will update. A diagonal block will not update the border rows where the minimum column element within the border row is greater than the highest column within the diagonal block. Therefore, the first diagonal blocks may not update all the border blocks where as the last diagonal blocks are almost guaranteed to update all the border blocks.

In this section we describe how the above mentioned constraints were dealt with.

5.4.1 Block Size

The diagonal blocks and the border blocks from the **H*** ordering (which are called ordered blocks) vary in size, from small blocks of one row to large blocks containing up to 10% of the rows in the matrix. The disparity of block sizes of the ordered diagonal blocks and the ordered border blocks have a major impact on the load balance. The presence of both large and small diagonal blocks has advantages as well as disadvantages. Small diagonal blocks provide more fine grained tasks making it easier to maintain a reasonable load balance. However, small sized diagonal blocks increase the number of these blocks and the number of times a block of border rows is updated.

Large diagonal blocks provide the advantage of reducing the number of border block updates. However, large diagonal blocks reduce the number of blocks and therefore reduce the number of possible distributions. With large diagonal blocks it may not be possible to achieve a reasonable load balance, and the imbalance caused by one or two blocks will be much greater than with the small diagonal blocks.

The different sizes, meaning the number of rows, of the border blocks provide similar tradeoffs during the border update. Large border blocks reduce the number of border block updates by increasing the number of border rows that will be updated at a time. However, the large border blocks reduce the amount of work that can be distributed among the clusters. Small border blocks increase the number of border block updates to be performed, but at the same time increase the amount of work that cluster can share.

The disparity in block sizes from the ordering results in problems for the solver as it attempts to find a load balance for good performance. To improve the performance, the solver regroups the ordered diagonal blocks and the border rows to form new blocks which are called partitioned blocks. The solver combines small ordered diagonal blocks into larger partitioned diagonal blocks (or small ordered diagonal blocks may be combined with other large ordered diagonal blocks to form partitioned diagonal blocks). This allows the solver to eliminate the smaller, ordered diagonal blocks and results in a new, smaller set of partitioned diagonal blocks that are larger and closer to the same size.

The differences in the size of the ordered border blocks results in the solver having trouble achieving a good load balance within a cluster during the border update. To handle this imbalance the solver eliminates the ordered border blocks and combines the border rows to create partitioned border blocks that are equivalent in size. This allows the solver to create border blocks that will keep all the CE's on a cluster busy during the update.

The solver repartitions the ordered diagonal blocks and the ordered border blocks into the partitioned diagonal blocks and the partitioned border blocks before doing anything else with the blocks. As a result, the remainder of the solver only deals with the partitioned diagonal blocks and the partitioned border blocks. Therefore, in the discussion of the operations of the solver the references to diagonal blocks refer to the partitioned diagonal blocks and the references to the border blocks refer to the partitioned border blocks.

The results for changing the partitioned block sizes are in Section 6.3.1.

5.4.2 Interleaved

The first static load balance scheme that was tried was to interleave the diagonal blocks that could perform border updates among the clusters. Each border block is checked to determine which diagonal block can perform the first update on the border block. The diagonal blocks that are able to perform a first update are distributed in an interleaved fashion among the clusters. The remaining diagonal blocks are distributed to the clusters in an attempt to even the amount of work to be done be each cluster.

To estimate the amount of work required for each diagonal block during the border updates, a work count is calculated as the summation of the border rows to be updated by the diagonal block times the number of rows in the diagonal block. The amount of work assigned to each cluster is calculated as the summation of the work counts for the diagonal blocks assigned to the cluster. Each of the remaining diagonal blocks is assigned to the same cluster as the diagonal block preceding it as long as the even work distribution is not violated. When the work count for the cluster exceeds its limit, the diagonal block is assigned to the next cluster which does not exceed its work count.

This distribution scheme attempts to create a pipeline of border updates to prevent one cluster from doing all of the work at the end of the border update phase. However, its major problem is that the number of times a border block must be copied between clusters is at least as large as the number of diagonal blocks with parallel work. The amount of overhead required when there are a large number of diagonal blocks with parallel work made this scheme impractical.

See Section 6.3.2 for the results of testing the interleaved scheme.

5.4.3 Even Division of Parallel Work

The second static load balance scheme consists of evenly dividing the parallel work into sets such that there is one set for each cluster. Each set contains the same number of diagonal blocks, and the diagonal blocks within each set are adjacent. The remaining diagonal blocks are assigned to the clusters in the same fashion as the interleaved distribution so that each cluster had about the same work count.

This scheme attempted to reduce the number of times a border block would be copied between clusters. However, with the even distribution of the work, some extra copies could be required, but not as many as were required for the interleaved scheme. The problem with this scheme is that the cluster that received the last diagonal blocks would usually take a lot longer to finish than the other clusters, however, it provided much better results than the interleaved scheme.

See Section 6.3.2 for the results of testing this scheme.

5.4.4 Evenly Divide Work

The final attempt at a static load balance scheme is to ignore the parallel work and evenly divide the work count among the clusters. Each cluster is assigned one set of consecutive diagonal blocks, with each set of diagonal blocks having about the same work count.

This method tries to even out the work assigned to each cluster and also guarantees that the number of times each border block is copied between clusters is at most $C - 1$ where C is the number of clusters. However, as expected, the amount of parallelism in the border update is too constrained to achieve an efficient parallel execution. Also, the work count tends to underestimate the amount of work that is required by the later diagonal blocks relative to the initial diagonal blocks since the work count does not account for the increasing density in the rows processed by the last diagonal blocks.

This scheme did not do as well as the scheme for evenly dividing the parallel work. The actual test results for this scheme are in Section 6.3.2.

5.4.5 Dynamic Load Balancing - S Blocks

This section discusses the dynamic load balancing scheme that was added to the solver to enhance the efficacy of the static schemes. The main result of studying the static load balancing schemes is that a static method is unable to determine where to distribute the diagonal blocks so that an efficient parallel computation is obtained for the border update and at the same time to minimize the number of times a border block is copied between clusters.

A dynamic scheme was designed to redistribute the work at the end of the border update, where the largest load imbalance was observed. In this strategy, the last s diagonal blocks in the matrix, referred to as the S blocks, are placed in global memory after their diagonal block factorization and the off-diagonal updates are completed. With the S blocks in global memory, any cluster that has a border block to be updated by an S block can retrieve the S block and perform the update. A global work queue is used within the border update to hold the border blocks that need to be updated by the S blocks. After a cluster has finished the work assigned to its local work queue, the cluster can retrieve border blocks from the global work queue and perform the border updates using the S blocks. Instead of having one cluster responsible for performing the last updates on all of the border blocks, this strategy allows any of the clusters to perform the last updates on the border blocks.

During the border update phase a cluster waits for border blocks to be placed in its local work queue. (This queue is in global memory and the border block is placed there by the cluster that performed the preceding update.) When the cluster finds a border block in its local work queue it retrieves the border block and performs the update using the appropriate diagonal block. After performing one update, the cluster checks to see if the next update to be performed is by a diagonal block assigned to it. If the next diagonal block is assigned to it, the cluster performs the next update. The cluster keeps the border block until the next diagonal block needed for an update is assigned to some other cluster.

At this point in the static strategies the cluster would send, via global memory, the border block to the cluster that was assigned the next diagonal block. In this strategy, if the next diagonal block is an S block then the cluster can keep the border block, retrieve the S block from global memory, and perform the next update. The cluster can keep performing updates with the S blocks until all of the updates required by the border block have been performed.

There is, however, a problem with allowing a cluster continue performing updates on a border block B_i with the S blocks. If there is a border block B_j for the cluster in its local work queue, no other cluster can update the border block B_j. However, the border block B_i that is being updated by the S blocks can be updated by any of the clusters. This may mean that one cluster is idle while there is local work for another cluster that is not being done. Therefore, when a cluster x is performing S block updates on a border block B_i and a border block B_j is placed in its local work queue, the cluster x places the border block B_i in the global work queue, rather than continuing the application of S blocks, so that if some other cluster y is idle, then cluster y can perform the updates on border block B_i. The cluster x then retrieves border block B_j from the local work queue and starts updating the border block.

In order to limit the number of times a border block is transferred between clusters, a cluster may not want to check its local work queue between every border block update by S blocks. It may be that once a cluster has a border block being updated by S blocks it may only want to check its work queue every other update or every third update. A parameter is available to set the number of updates a cluster performs before checking it local work queue in order to achieve better performance.

When the border block needs to be updated by an S block it was earlier stated that the cluster then retrieves the S block from global memory to perform the update. However, it is not necessary to retrieve the S block every time, nor may the S block even need to be retrieved once. The cluster on which the factorization of the S block was performed has no need to retrieve the S block since it already has the block in cluster memory. The cluster can use its local copy of the S block and does not need to retrieve the block. In addition, once a cluster has retrieved an S block from global memory, it saves the S block in its cluster memory. As a result, a cluster only needs to retrieve an S block once. If the cluster later needs the same S block, it uses the copy it retrieved previously.

Due to the different loads on the different clusters, it may be that cluster x wants to use an S block before cluster y has calculated the factorization of the S block. Therefore, before a cluster can retrieve an S block from global memory it must make sure the S block is actually available. If the cluster needs to update a border block with an S block that is not yet available, it places the border block on the end of the global work queue and tries to find other work that it can do. If computations on cluster y complete much later than on cluster x, cluster x may not find any work that can be done until cluster y places the S blocks in global memory.

The results for the use of the S blocks are presented in the Section 6.3.3.

6 MCSPARSE Results

This section presents the results for the large-grain parallel sparse system solver, MCSPARSE. The experiments were performed on an Alliant FX/80 and the Cedar multiprocessor with various memory/processor configurations.

6.1 Stability Results

A parameter study was made using different combinations of the two diagonal casting techniques described earlier to determine the set of parameters which would result in the smallest relative errors. No border casting check was made. The following tests were conducted:

No Checks This test was run without any diagonal casting checks.

Absolute Check This test was run with the minimum value allowed for a pivot set at 10^{-5} ($\alpha = 10^{-5}$ in Equation (7)).

Relative Check This test was run with the tolerance on pivot size relative to the original border at 10^{-4}($\beta = 10^{-4}$ in Equation (8)).

Both This test was run with both diagonal casting checks made using the parameters above.

The tests were run using matrices from the Harwell-Boeing test collection. All the matrices chosen were from the real, unsymmetric, assembled (RUA) section of the collection. In all cases, the locally stable pivot test, Equation (6), used $\gamma = 0.1$. A total of 78 matrices were solved by at least one version of the solver. The following table, Table 10, compares the different sets previously described. The table lists the number of matrices each set of parameters correctly solved to with the given number of orders of magnitude. The number of matrices listed under the 0 column indicates that the parameter set solved that number of matrices to same order of the relative error as the best result achieved with the four versions of the code compared. The number of matrices listed under the 1 column indicates that the parameter set solved that number of matrices to within one order of magnitude of the best result. The columns labeled 2 and 3 are similarly defined. These tests were run on an Alliant FX/80 with 8 processors.

As can be seen from the table the main diagonal casting improvement results when performing the check against the magnitude of the pivot (and indirectly against the rest of the pivot column and overall norm of matrix). Using only the check of the pivot relative to the norm of the original border elements yielded results similar to no casting at all. The combination of the two types of casting produced the worst results of the four different combinations. Notice that the differences are not that great between the four versions. This is due to the fact that the H0 phase of H* ordering enhances stability for many matrices to a level such that little or no casting is required.

Test	0	1	2	3	Total
No Checks	52	11	2	1	66
Absolute Check	55	11	2	2	70
Relative Check	51	10	1	3	65
Both	48	13	0	3	64

Table 10: Casting Parameter Study

Test	New	Better	0	1	2	3	Total
Relative Casting	6	22	35	15	1	1	79

Table 11: Casting Results for Relative Casting

When the border casting was added to the solver, the accuracy was increased and the solver was able to solve more of the matrices. The tests run with the border casting also use the first diagonal casting check with $\alpha = 10^{-5}$. The border casting used a tolerance of 10^{-6} ($\epsilon = 10^{-6}$ in Equation (9)). Six matrices were solved with border casting that could not be solved with diagonal casting alone. However, five matrices that were solved before could no longer be solved. These failures were due to the amount of casting the solver attempted in order to obtain a more accurate result. The resulting border size exceeded a bound set in the implementation of MCSPARSE.

Table 11 shows the results when border casting is added and contains the same fields as the previous casting results table, with the addition of two new columns. The first column, New, indicates the number of new matrices that were solve. The second column, Better, indicates the number of matrices for which the accuracy of the solution improved compared to the solver with diagonal casting only. As can be seen from the table border casting improved the accuracy of the solver for 28 matrices and still solved a total of 79 matrices within 3 orders of accuracy of the best parameter set for the solver.

Another factor that influences the stability of the solver is the ordering of the matrix. The effect of different orderings on the stability of the solver can be seen by using the orderings that were presented in Section 3.6.3. Table 12 shows the relative maximum norm of the error,

$$error \;=\; \frac{\max_{1 \le i \le n}(\mid x_{calculated} - x_{known} \mid)}{\max_{1 \le i \le n}(\mid x_{known} \mid)} \tag{17}$$

for a subset of the large RUA matrices. The smallest error for each matrix is highlighted. The solver results are not provided for the **Tarjan** ordering or for the **No H2** ordering due to the inability of the solver to handle the size of the diagonal blocks for most of the matrices. The solver was configured to use diagonal casting with $\alpha = 10^{-5}$ and border casting with $\epsilon = 10^{-6}$.

As can be seen from this table the solution of different orderings of the same matrix can vary by up to six orders of magnitude for some matrices and have little effect on others. This implies that the ordering used in conjunction with the solver can have a significant effect on the stability of solution of some matrices.

It can also be seen that the effect of the bounded transversal on the stability of the matrices is considerable. For nine of the thirteen matrices the unbounded transversal resulted in the largest relative error in the solution. Of the other four matrices, the solutions of two resulted in the same error regardless of the ordering, and in the remaining

Matrix	**Hybrid**	**H2**	**No H0**	**Nored**
gaff1104	**.9E-06**	.2E-05	.1E-03	.1E-06
gemat12	.6E-07	**.3E-08**	.1E-07	.3E-07
gre_1107	.1E-05	.8E-06	.2E-04	**.5E-08**
mahistlh	**.2E-08**	**.2E-08**	.9E-07	.1E-07
orsirr_1	.1E-12	.1E-12	**.6E-13**	.5E-12
orsreg_1	**.1E-12**	**.1E-12**	**.1E-12**	**.1E-12**
pores_2	.3E-05	.4E-06	.2E-04	**.5E-08**
saylr4	**.1E-10**	**.1E-10**	**.1E-10**	**.1E-10**
sherman1	.1E-12	**.4E-13**	.1E-12	.1E-12
sherman4	.1E-13	.2E-13	**.2E-14**	.1E-13
sherman5	.2E-08	.8E-08	.1E-04	**.3E-10**
west1505	.2E-05	.2E-06	.3E-03	**.1E-06**
west2021	.5E-06	.7E-06	.1E-06	**.2E-07**

Table 12: Maximum Norm of the Relative Error for the Large RUA Matrices Using Different Orderings

two the error was one order of magnitude better than the other orderings. These results support the use of the bounded transversal in the ordering to improve the stability of the solver.

Further tests were conducted to examine the relationship between casting and the bounded transversal. In these tests, a set of matrices were ordered with and without the bounded transversal. The two orderings were then solved with and without casting. The casting methods used within the solver were the same as the previous test. Table 13 contains the results.

The results show that the effects of casting and the bounded transversal are different for different matrices. For matrices such as *orsirr_1*, *orsreg_1*, *saylr4*, *sherman1*, and *sherman4*, roughly the same results are achieved regardless of use of the transversal or casting. For *sherman5* and *west1505*, however, the solutions of the bounded orderings are the same regardless of the casting, and better than the unbounded transversal case, with or without casting. In contrast, *mahistlh* and *west2021* show much better stability with casting than without, but the transversal has little effect on these matrices. And *gemat12* provides better results as long as either casting or the bounded transversal is used. These comparisons show that, in practice, in order to provide the best stability for a wide range of sparse matrices both techniques should be used.

The stability results so far have compared different versions of the solver to indicate when the stability of the solver has improved. However, it is also important to compare the solver against other known solvers. For these comparisons the solver MA28, [Duf77], was chosen and 80 matrices were solved. MA28 was run with the stability factor (u) at 1.0 (the most stable value) and with a value of 0.1 (which is less stable but allows better control of fill-in). MCSPARSE was run with the diagonal casting ($\alpha = 10^{-5}$) and border casting ($\epsilon = 10^{-6}$). Table 14 compares the relative stability of the two solvers, showing the increase in the stability of MCSPARSE over MA28 for the RUA test matrices. The different columns indicate the difference in the order of the relative errors between the two solvers. If the MCSPARSE solver had a relative error of 10^{-10} and if the MA28 solver had a relative

Matrix	With Casting		Without Casting	
Matrix	Bounded	Unbounded	Bounded	Unbounded
gaff1104	.9E-06	.1E-03	.6E-06	.4E-05
gemat12	.6E-07	.1E-07	.2E-07	.1E-05
gre_1107	.1E-05	.2E-04	.3E-05	.5E-04
mahistlh	.2E-08	.9E-07	.1E-01	.2E-01
orsirr_1	.1E-12	.6E-13	.1E-11	.1E-12
orsreg_1	.1E-12	.1E-12	.1E-12	.1E-12
pores_2	.3E-05	.2E-04	.5E-12	.1E-03
saylr4	.1E-10	.1E-10	.1E-10	.1E-10
sherman1	.1E-12	.1E-12	.1E-12	.1E-12
sherman4	.1E-13	.2E-14	.6E-14	.2E-13
sherman5	.2E-08	.1E-04	.6E-08	.4E+10
west1505	.2E-05	.3E-03	.5E-05	.4E-03
west2021	.5E-06	.1E-06	.8E+01	.2E+00

Table 13: Maximum Norm of the Relative Error for the Large RUA Matrices Comparing the Bounded Transversal and Casting

Stability	$> +3$	+3	+2	+1	0	-1	-2	-3	< -3
$u = 1.0$	2	2	3	4	20	14	11	8	16
$u = 0.1$	4	1	6	7	21	12	11	5	13

Table 14: Casting Stability Compared to MA28

error of 10^{-8} then this would be considered an improvement of +2 orders of magnitude. Table 15 compares the stability for only fourteen of the large RUA matrices.

It can be seen that the two solvers MA28 and MCSPARSE are comparable with respect to relative error achieved. The table for all the matrices shows that for 64% of the matrices the difference between the solvers is within two orders of magnitude and for 78% of the matrices the difference between the solvers is within three orders of magnitude. When only looking at the large matrices the numbers are similar, for 50% of the matrices the difference between the solvers is within two orders of magnitude and for 79% of the matrices the difference between the solvers in within three orders of magnitude. Another factor in the comparison of the stability results is the nondeterminism of border casting.[1]

6.2 Fillin 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 in the matrices and the original number of elements in the matrices can be found in Table 1.

The tests were conducted by altering the choice of the values outside the diagonal block which are included in the modified Markowitz count. The following tests were conducted.

[1]The nondeterminism arises from the use of S blocks and the implementation choice of updating in parallel multiple rows of a border block with the same diagonal block

Matrix	MCSPARSE	MA28	
		$u = 1.0$	$u = 0.1$
gaff1104	.9E-06	.5E-06	.4E-06
gemat12	.3E-07	.5E-10	.1E-09
gre_1107	.3E-05	.4E-08	.6E-06
mahistlh	.1E-08	.1E-12	.1E-12
orsirr_1	.1E-12	.4E-12	.9E-12
orsreg_1	.1E-12	.4E-12	.4E-11
pores_2	.2E-05	.1E-09	.1E-09
saylr4	.1E-10	.2E-10	.8E-10
sherman1	.1E-12	.9E-13	.1E-11
sherman2	.1E+01	.6E-08	.7E-06
sherman4	.7E-14	.7E-14	.1E-10
sherman5	.4E-09	.2E-12	.2E-09
west1505	.3E-05	.1E-07	.5E-08
west2021	.5E-05	.4E-08	.3E-08

Table 15: Stability Comparison Between MCSPARSE and MA28

Test	*row_mult*	*ofactor*	*col_mult*	*bfactor*
Both	1.0	1.0	1.0	1.0
Row	1.0	1.0	0.0	1.0
Column	0.0	1.0	1.0	1.0
Neither	0.0	1.0	0.0	1.0

Table 16: Parameters for Sparsity Testing

Both This tests include both the off-diagonal row counts and the border column counts in the modified Markowitz count.

Row This test only included the off-diagonal row counts in the modified Markowitz count.

Column This test only included the border column counts in the modified Markowitz count.

Neither This test used neither the border column counts or the off-diagonal row counts. The Markowitz count was calculated from the elements within the diagonal block.

The parameter values used for the tests are in Table 16. The number of fill-in elements from the tests are in Table 17. 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 at 0.1. The *No Control* column indicates the number of fill-in elements MCSPARSE generated when operating without any fill-in control.

Clearly, the use of some sort of fill-in control is crucial for MCSPARSE. 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 available storage (due to selected use of dense structures) so that the cost of the extra work created by fill-in is significantly reduced. This can be seen in the experimental results of Section 6.

Matrix	MA28		No Control	Both	Row	Column	Neither
	$u = 1.0$	$u = 0.1$					
gaff1104	57626	62450	107407	91524	94317	88257	92268
gemat12	27374	18526	661352	132291	125124	133917	125418
gre_1107	39706	37411	246047	198616	194791	198745	199749
mahistlh	4624	2729	39490	21350	20076	21677	20771
orsirr_1	80683	44957	143431	81700	94703	101511	108215
orsreg_1	310517	136528	266218	240348	230888	219794	220971
pores_2	28545	29149	128724	61500	57870	62558	62239
saylr4	288438	451305	686714	533654	533351	461879	510048
sherman1	14860	20262	35440	24456	23241	22428	24158
sherman2	258300	241186	354402	320374	318746	316680	322516
sherman4	10756	18571	25502	18190	16351	18405	17078
sherman5	191177	131962	342410	195123	207527	208106	199759
west1505	3380	2664	51096	23652	20251	24766	22069
west2021	3944	3530	78052	32534	31764	31680	32375

Table 17: Fill Elements for Large RUA Matrices

6.3 Results of load balancing experiments

This section presents some results of load balancing experiments on a Cedar configuration of four clusters with two processors per cluster. For such tests, it is necessary augment the results on the large reordered Harwell-Boeing matrices with results on parameterized expansions of these matrices. This is due to the fact that many of the Harwell-Boeing matrices are too small to warrant a multicluster version of MCSPARSE. The expansions of the reordered matrices are obtained via a simple repetition strategy which preserves the bordered block upper triangular structure. For example, the fourth expansion of *bp_800* consists of four repetitions of the diagonal blocks along the diagonal followed by four repetitions of the border diagonal block. This includes four sections of the border rows and off-diagonal rows. Elements are also added at an average of five per row to each of the off-diagonal rows. This can be seen in Figure 17. The A block represents the diagonal blocks and part of the off-diagonal rows. The B block represents the off-diagonal portion of the border rows. The C block represents the off-diagonal columns with the border diagonal block. And the D block represents the border diagonal block. The area filled with the dashed lines indicates where the random fill-in elements were placed during the expansion.

6.3.1 Block Size Results

The effect of the partitioned diagonal block size (the new blocks formed, at the start of the factorization, by regrouping the diagonal blocks that result from H*), when using the interleaved static load balancing scheme, can be seen in Figure 18. This graph shows the wall clock time to solve the fourth expansion of the *bp_800* as a function of the minimal diagonal block size. When the partitioned diagonal block size is too small, the number of diagonal blocks is very large and the overall amount of time to solve the matrix increases. However, when the diagonal block size is too large the number of blocks is very small and

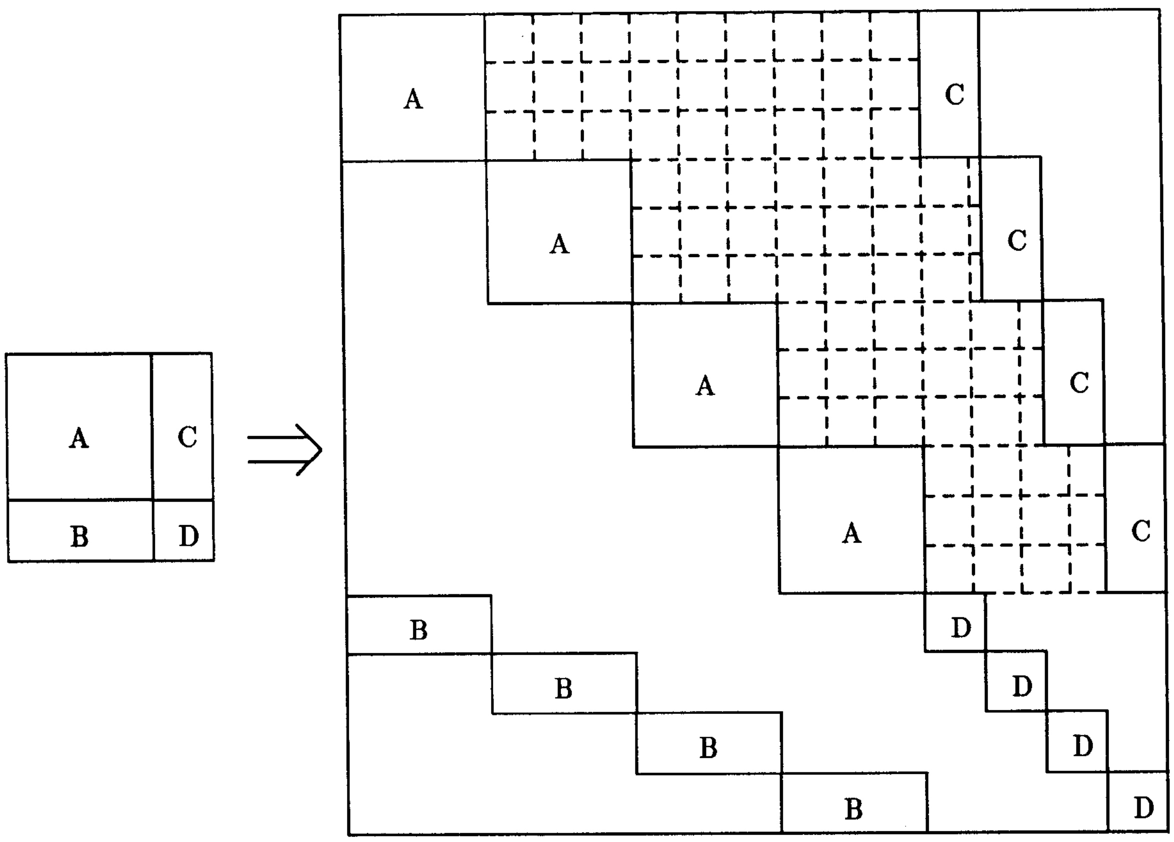


Figure 17: Fourth Expansion of *bp_800*

Matrix	One Cluster	Interleaved	Even Parallel	Even Work
bp_800	1.40	2.80	2.77	3.03
gemat12	33.17	21.70	21.04	22.47
saylr4	62.10	40.90	38.19	45.14
sherman2	58.58	32.32	30.22	39.20
sherman5	41.80	22.33	20.97	29.21
west1505	4.21	4.28	4.52	4.99
west2021	4.94	5.08	4.86	5.33

Table 18: Static Load Balance Solution Times

again the overall amount of time to solve the matrix increases due to a loss of parallelism.

The effect of the partitioned border block size (the border blocks formed by the solver regrouping the border blocks from the ordering) can be seen in Figure 19. This graph shows the wall clock time to solve the sixth expansion of *bp_800* as a function of the maximal border block size. As the border block size increases, the time to solve the system also increases. Of course, decreasing the partitioned border block to very small sizes causes an increase in execution time as well due to increased overhead of intercluster synchronization.

6.3.2 Static Load Balance Results

This section compares the static load balancing schemes. The three experiments were as follows:

Interleaved The application of the interleaved load balance algorithm.

Even Parallel The application of the even division of parallel work algorithm.

Even Work The application of the even division of work algorithm.

Table 18 presents the wall clock times for the solutions of the matrices using the different load balancing algorithms. The one cluster wall clock times are also provided.

These results show that the even division of parallel work is slightly better than the interleaved version, which are both much better than the version with the even division of work. The results show that for all but one of the example matrices the even division of parallel work provides the best performance. For the other matrix, *west1505*, the interleaved version actually provides better performance. The results show that the interleaved scheme is on average 3.43% slower than the even division of parallel work scheme. The even division of work scheme, however, is clearly the worst overall with performance 17.64% slower than the even division of parallel work scheme. This table also demonstrates the problem with the Harwell-Boeing matrices on a multicluster Cedar. The matrices *gemat12* and *sherman5* are the only ones large enough to show improved performance on multiple clusters relative to a single cluster. Based on these results, the default static scheme chosen for MCSPARSE is taken to be the even division of parallel work scheme.

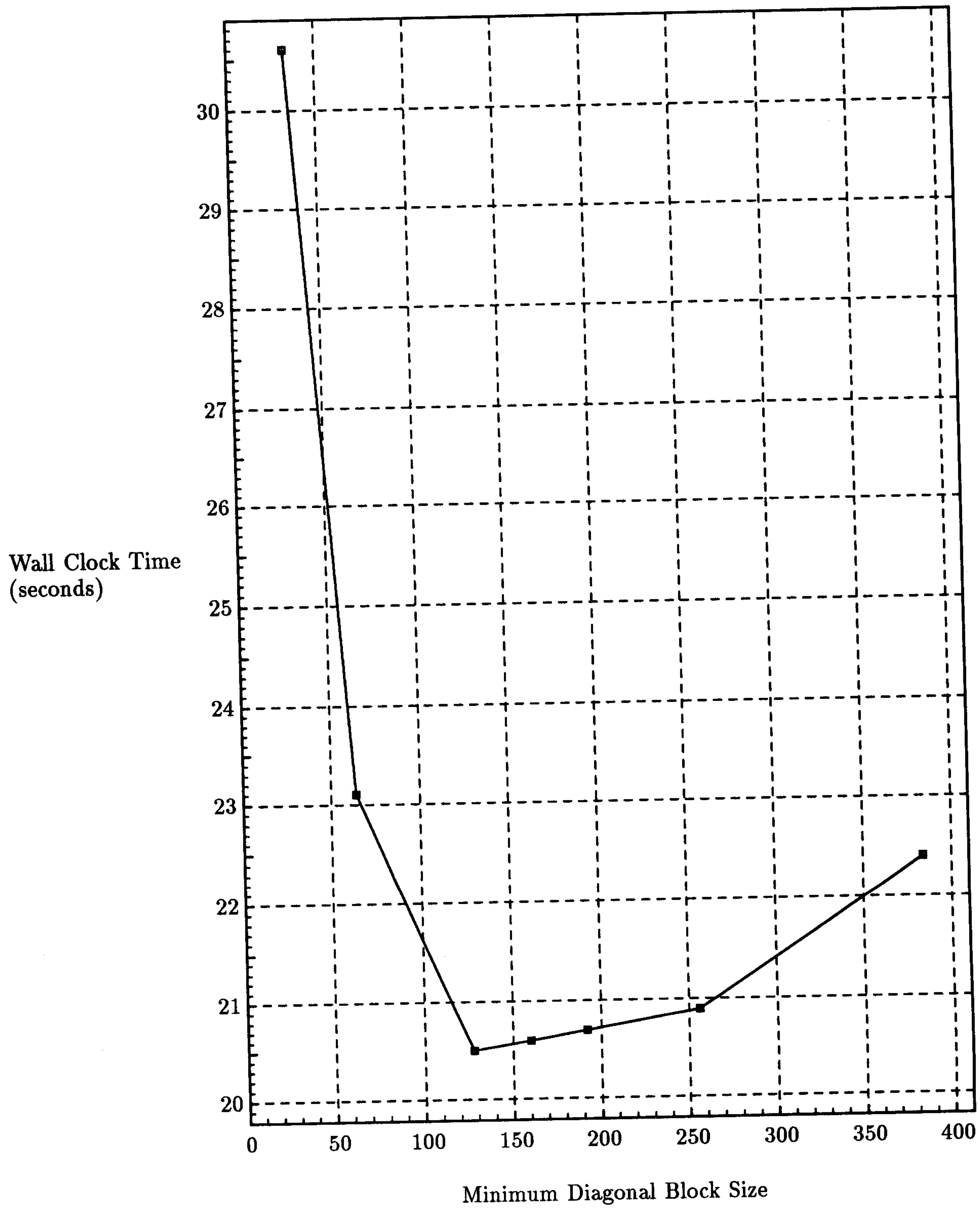


Figure 18: Solution Times for Fourth Expansion of *bp_800*

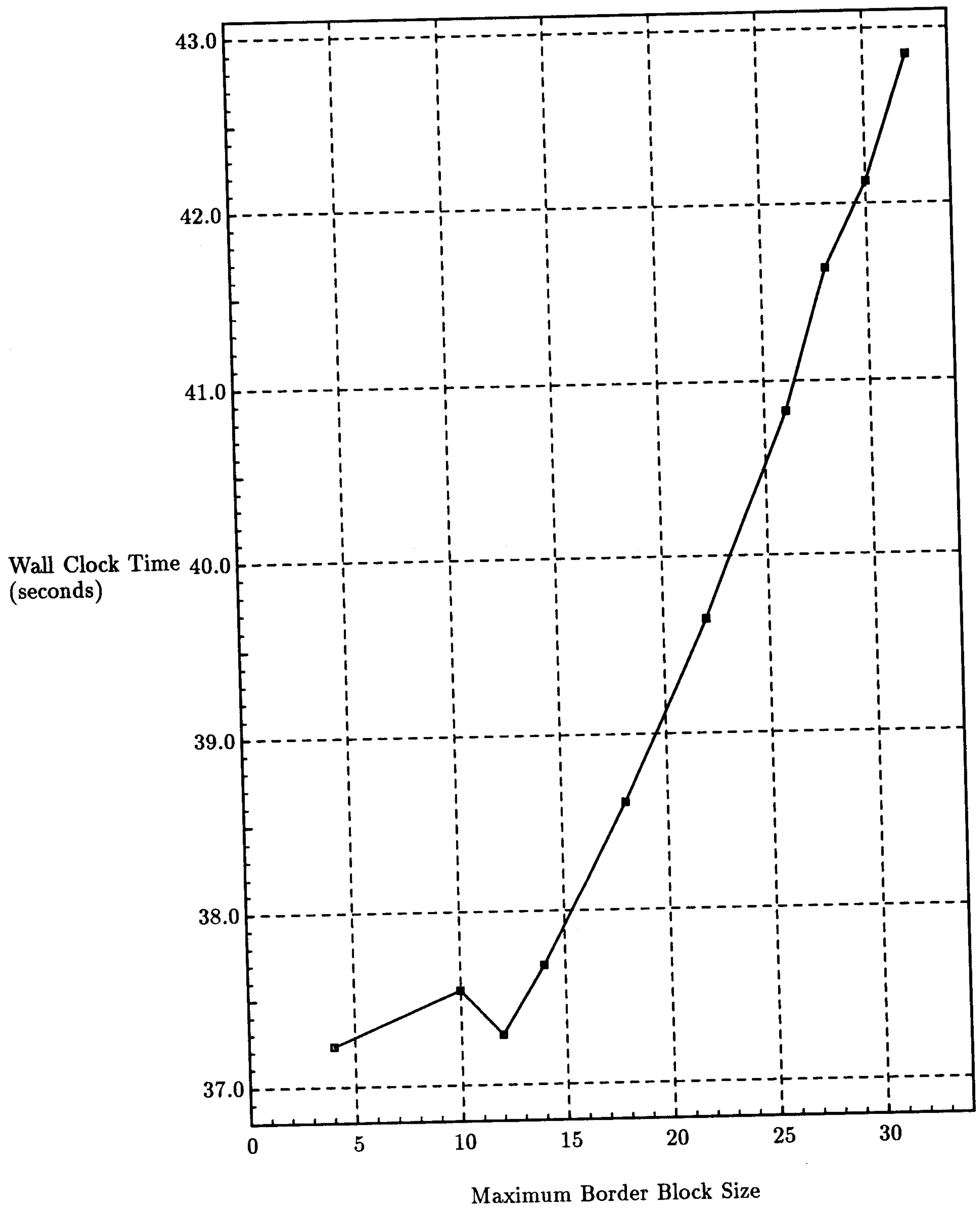


Figure 19: Solution of the Sixth Expansion of *bp_800*

Matrix	Interleaved		Even Parallel		Even Work	
	No S	With S	No S	With S	No S	With S
gemat12	21.70	20.53	21.04	18.77	22.47	20.98
saylr4	40.90	40.80	38.19	37.46	45.14	45.41
sherman2	32.32	32.33	30.22	30.11	39.20	39.30
sherman5	22.33	18.52	20.97	19.00	29.21	23.40
west1505	4.28	4.30	4.52	4.52	4.99	5.09
west2021	5.08	4.54	4.86	4.89	5.33	5.30

Table 19: Dynamic Load Balance Solution Time Comparisons

6.3.3 Dynamic Load Balance Results

The third expansion of the *sherman4* matrix was used to investigate the effect of using S blocks to enhance the performance of the static scheme. The speedup of MCSPARSE on four clusters over MCSPARSE on one cluster as a function of the number of S blocks is presented in Figure 20. It is seen that as the number of S blocks increases the speed up of the solver improves. However, after some point the addition of more S blocks decreases performance due to the fact that a more significant amount of intercluster synchronization and data movement is required.

The wall clock times for the solution, with and without S blocks, of the large Harwell-Boeing matrices used for the evaluation of the static load balancing schemes are shown in Table 19. (see Section 6.3.2). These results were collected without fine tuning the number of S blocks to achieve the best performance as was done in the previous test. A default value of five S blocks was used for all matrices. Even without tuning performance improvements were achieved for almost all the matrices. As expected, the improvements were the most significant for the matrices *sherman5*, which resulted in an average performance increase of 15.45%, and *gemat12*, which resulted in an average performance increase of 7.61%.

6.4 Cedar Performance Results

This section contains the performance results for MCSPARSE collected on a four cluster Cedar configuration, with each cluster comprising four processors. The times given in this section are wall clock times, in seconds, for the code running in single user mode.

As mentioned in the previous section, 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 large sized Harwell-Boeing matrices *gemat12*, *saylr4* and *sherman3*, together with the seventh to ninth expansion of *bp_800* were used to obtain performance measurements on the Cedar system. The results for these systems are summarized in Table 20.

This table shows clearly that as the size of the system increases, e.g., the seventh through the ninth expansion of *bp_800*, the speedup increases accordingly. It should be noted that for the eight and the ninth expansion the speedup for two clusters is superlinear, 2.4 and 2.7 respectively. This is caused by the fact that, when these systems were solved on one cluster, the data space required was larger than the cluster memory on one cluster,

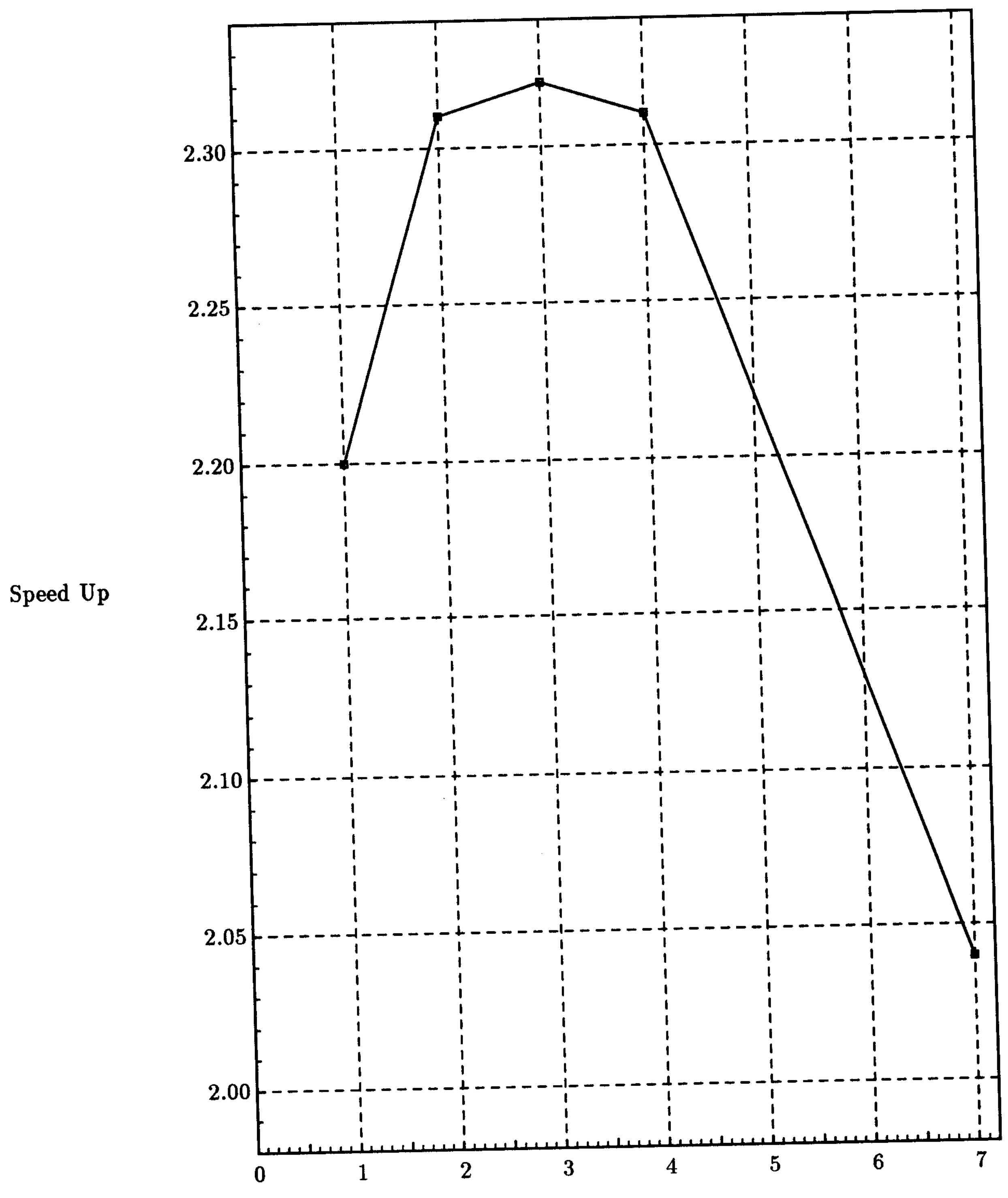


Figure 20: Solution of the Third Expansion of *sherman4*

Matrix	1 Cluster	2 Cluster		4 Cluster	
	Wall Time	Wall Time	Speed Up	Wall Time	Speed Up
bp_800.7	48.919	33.213	1.47	27.523	1.78
bp_800.8	98.175	40.967	2.40	34.848	2.82
bp_800.9	137.076	50.844	2.70	43.802	3.13
gemat12	36.434	17.819	2.04	17.947	2.03
saylr4	42.390	35.279	1.20	31.629	1.34
sherman3	37.054	26.265	1.41	25.106	1.48

Table 20: Solution Time for Large Matrices

which resulted in a significant number of page faults. When the systems were spread across more than one cluster, the cluster data space required was satisfied by the available cluster memory, the number of page faults decreased, and the super-linear speed up was obtained.

6.5 Alliant FX/80 Performance Results and MA28 Comparisons

In this section we give performance results for MCSPARSE on the Alliant FX/80, and compare its effectiveness against a known sequential sparse solver, MA28 [Duf77].

The solution times of the large matrices from the RUA collection for both the MCSPARSE and MA28 solvers are presented in Table 21. This table contains the user process times for the solutions as collected in single-user mode on the Alliant FX/80. The times for the MCSPARSE solver are presented for both one and 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 two stability constraints, $u = 1.0$ and $u = 0.1$. The single time presented for each 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/80, the speedup obtained for eight processors over one processor is significant. The Alliant FX/80 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 75 for *orsreg_1* using $u = 1$. 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 unsymmetric systems. The interested reader should see

Matrix	Hybrid	MCSPARSE 1CE		MCSPARSE 8CE		MA28	
	Reorder	Solution	Total	Solution	Total	$u = 1.0$	$u = 0.1$
gaff1104	2.634	23.864	26.498	5.803	8.437	50.714	85.148
gemat12	6.407	44.194	50.601	10.640	17.047	67.127	15.600
gre_1107	3.345	23.196	26.541	5.567	8.912	38.975	27.695
mahistlh	3.995	4.362	8.357	1.317	5.312	5.269	4.404
orsirr_1	1.698	11.305	13.003	3.624	5.322	94.902	23.622
orsreg_1	4.219	31.244	35.463	7.820	12.039	898.284	99.724
pores_2	4.933	9.474	14.407	3.007	7.940	28.889	27.777
saylr4	6.833	84.743	91.576	21.736	28.569	256.879	962.375
sherman1	0.973	2.948	3.921	1.026	1.999	5.653	11.466
sherman4	0.797	3.139	3.936	1.022	1.819	3.660	10.737
sherman5	5.036	60.838	65.874	13.118	18.154	705.007	284.739
west1505	8.814	4.937	13.751	1.340	10.154	7.940	6.385
west2021	14.657	5.405	20.062	1.668	16.352	13.206	11.117

Table 21: Solution Time Comparison Between MCSPARSE and MA28

[GSZ91] for the performance of the unsymmetric sparse code Y12M2.

7 Conclusions

A parallel solver for unsymmetric 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 the use of 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 that employed in MA28.

The H* ordering combines four different orderings, H0, Tarjan's algorithm for finding strongly connected components, H1 and H2, to transform a matrix into bordered block upper triangular form. Except for the H0 ordering all of these orderings are symmetric, which distinguishes this ordering 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 multivector processor, an Alliant FX/80, 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 multivector processors and to exploit their architectures efficiently. Initial results, [Wan91], 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 [GSZ90b, GSZ90a], 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

[AGL+87] 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. *Intl. J. Supercomputing Appl.*, 1(4):10–30, Winter 1987.

[Ala88] G. Alaghband. A parallel pivoting algorithm on a shared memory multiprocessor. In *Proceedings of the 1988 International Conference on Parallel Processing, Volume3: Applications and Algorithms*, volume 3, pages 177–180, 1988.

[Bun74] J. R. Bunch. Analysis of sparse elimination. *SIAM J. Numer. Anal.*, 11(5):847–873, October 1974.

[Dav89] T. Davis. A parallel algorithm for sparse unsymmetric LU factorization. Technical Report CSRD Report No. 907, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, 1989. PhD. thesis.

[DER86] Iain S. Duff, Albert M. Erisman, and John K. Reid. *Direct Methods for Sparse Matrices.* Oxford University Press, New York, 1986.

[DR83] I. S. Duff and J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Trans. Math. Software*, 9:302–325, 1983.

[Duf77] I. S. Duff. Ma28- a set of fortran subroutines for sparse unsymmetric linear equations. Technical Report Report AERE R8730, HMSO, London, 1977.

[Duf81a] I. S. Duff. Algorithm 575. permutations for a zero-free diagonal. *ACM Trans. Math. Software*, 7(3):387–390, September 1981.

[Duf81b] I. S. Duff. On algorithms for obtaining a maximum transversal. *ACM Trans. Math. Software*, 7(3):315–330, September 1981.

[Duf86] I. S. Duff. Parallel implementation of multifrontal schemes. *Parallel Computing*, 3:193–204, 1986.

[EGL+87] A. M. Erisman, R. G. Grimes, J. G. Lewis, W. G. Poole Jr., and H. D. Simon. Evaluation of orderings for unsymmetric sparse matrices. *SIAM J. Sci. Stat. Comput.*, 8(4):600–624, July 1987.

[Gea75] C. W. Gear. Numerical errors in sparse linear equations. Technical Report UIUDCS-F-75-885, Department of Computer Science, University of Illinois, Urbana, IL, 1975.

[Geo73] A. George. Nested dissection of a regular finite element mesh. *SIAM J. Numer. Anal.*, 10(2):345–363, April 1973.

[Geo80] A. George. An automatic one-way dissection algorithm for irregular finite element problems. *SIAM J. Numer. Anal.*, 17(6):740–751, December 1980.

[GL78] A. George and J. W. H. Liu. An automatic nested dissection algorithm for irregular finite-element problems. *SIAM J. Numer. Anal.*, 15:1053–1069, 1978.

[GL81] A. George and J.W. Liu. *Computer Solution of Large Sparse Positive Definite Systems.* Prentice Hall, 1981.

[GMW89] K. Gallivan, B. Marsolf, and H. Wijshoff. A large-grain parallel sparse system solver. In *Proc. Fourth SIAM Conf. on Parallel Proc. for Scient. Comp.*, pages 23–28, Chicago, IL, 1989.

[GSZ90a] K. Gallivan, A. Sameh, and Z. Zlatev. Parallel hybrid sparse linear system solver. *Computing systems in engineering*, 1:183–195, 1990.

[GSZ90b] K. Gallivan, A. Sameh, and Z. Zlatev. Solving general sparse linear systems using conjugate gradient-type methods. In *Proceedings of the 1990 International Conference on Supercomputing*, pages 132–139, New York, 1990. ACM Press. June 11-15, 1990, Amsterdam, The Netherlands.

[GSZ91] K. Gallivan, A. Sameh, and Z. Zlatev. Parallel direct method codes for general sparse matrices. Technical Report 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.

[Gus76] F.G. Gustavson. Finding the block lower triangular form of a matrix. In J.R. Bunch and D.J. Rose, editor, *Sparse matrix computations.* Academic Press, New York, 1976.

[HK73] J. E. Hopcroft and R. M. Karp. An $n^{\frac{5}{2}}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 2(4):225–231, December 1973.

[HR72] E. Hellerman and D. C. Rarick. The partitioned preassigned pivot procedure (p^4). In D. J. Rose and R. A. Willoughby, editors, *Sparse matrices and their applications.* Plenum, New York, 1972.

[KDLS86] D. Kuck, E. Davidson, D. Lawrie, and A. Sameh. Parallel supercomputing today and the Cedar approach. *Science*, 231:967–974, 1986.

[Kuh55] H. W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1):83–97, March 1955.

[LL87] C.E. Leiserson and J.G. Lewis. Orderings for parallel sparse symmetric factorization. In *Proc. Third SIAM Conf. on Parallel Proc. for Scient. Comp.*, pages 27–31, Los Angeles, CA., 1987.

[LRT79] R.J. Lipton, D.J. Rose, and R.E. Tarjan. Generalized nested dissection. *SIAM J. Numer. Anal.*, 16:346–358, 1979.

[Mar57] H. M. Markowitz. The elimination form of the inverse and its application to linear programming. *Management Science*, 3:255–269, April 1957.

[Mar91] B. Marsolf. Large grain parallel sparse system solver. Technical Report CSRD Report No. 1125, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, 1991. Master Thesis.

[MH56] Jr. M. Hall. An algorithm for distinct representatives. *The American Mathematical Monthly*, 63(10):716–717, December 1956.

[OZ83] O. Osterby and Z. Zlatev. *Direct methods for sparse matrices.* Springer, Berlin, 1983.

[Sta91] CSRD Staff. The Cedar Project. Technical Report CSRD Report No. 1122, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, 1991.

[Wan91] X. Wang. private communication, April 1991.

[Wij89] H. A. G. Wijshoff. Symmetric orderings for unsymmetric sparse matrices. Technical Report CSRD Report No. 901, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, 1989.

[Yew86] P. Yew. Architecture of the Cedar parallel supercomputer. Technical Report CSRD Report No. 609, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, 1986.