A PARALLEL IMPLEMENTATION OF THE BLOCK-PARTITIONED INVERSE MULTIFRONTAL ZSPARSE ALGORITHM

YOGIN E. CAMPBELL* AND TIMOTHY A. DAVIS†

Technical Report TR-95-023, Computer and Information Sciences Department, University of Florida, Gainesville, FL, 32611 USA. October, 1995.

Key words. Symmetric multifrontal method, supernode, Takahashi equations, symmetric inverse multifrontal method, inverse frontal matrix, inverse contribution matrix, inverse assembly tree, Zsparse.

AMS (MOS) subject classifications. 05C50, 65F50, 65F05.

Abbreviated title. A parallel inverse multifrontal Zsparse algorithm

Abstract.

The sparse inverse subset problem is the computation of the entries of the inverse of a sparse matrix for which the corresponding entry is nonzero in the factors of the matrix. We present a parallel, block-partitioned formulation of the inverse multifrontal algorithm to compute the sparse inverse subset. Numerical results for an implementation of this algorithm on an 8-processor, shared-memory CRAY-C98 architecture are discussed. We show that for large problems we obtain efficiency ratings of over 80% and performance in excess of 1 Gflop.

1. Introduction. An efficient method to compute the sparse inverse subset (Zsparse) is important in practical applications such as the computation of short circuit currents in power systems or in estimating the variances of the fitted parameters in the least-squared data-fitting problem. (The sparse inverse subset (Zsparse), is defined as the set of inverse entries in locations corresponding to the positions of nonzero entries in the LDU factorized form of the matrix.)

In [3] Campbell and Davis introduced a new algorithm to compute Zsparse for symmetric matrices based on one of the equations presented by Takahashi, Fagan, and Chin in [11], and an inverted form of the symmetric multifrontal method. This paper discusses a parallel implementation of the block-partitioned form of the Zsparse algorithm. The target implementation platform is the CRAY-C98, a parallel-vector shared-memory machine. Extensions of the method (for arbitrary subsets) are discussed in [4]. See also [2].

We explicitly exploit two levels of parallelism in this implementation: the tree level, and the node level. The *tree level parallelism* is the coarse grain type of parallelism inherent among the nodes in the inverse assembly tree. The *node level parallelism* is of a finer grain than the tree parallelism and refers to the parallelism obtained by partitioning the work within a node in the inverse assembly tree. We shall show that taking advantage of the node parallelism can significantly improve the performance of the algorithm. There is, of course, the even finer grain parallelism at the individual operation level - this is implicitly exploited using the machine's vector capability and optimized BLAS primitives. We use a two-dimensional scheme to partition the tree nodes (tree tasks) to obtain the finer grain node tasks. The unprocessed tree and node tasks are maintained in a single queue. From this queue, node tasks are scheduled to processors using the guided self-scheduling scheme

* email: yec@cis.ufl.edu.

† Computer and Information Science and Engineering Department, University of Florida, Gainesville, Florida, USA. (904) 392-1481, email: davis@cis.ufl.edu. Technical reports and matrices are available via the World Wide Web at http://www.cis.ufl.edu/~davis, or by anonymous ftp at ftp.cis.ufl.edu:cis/tech-reports.

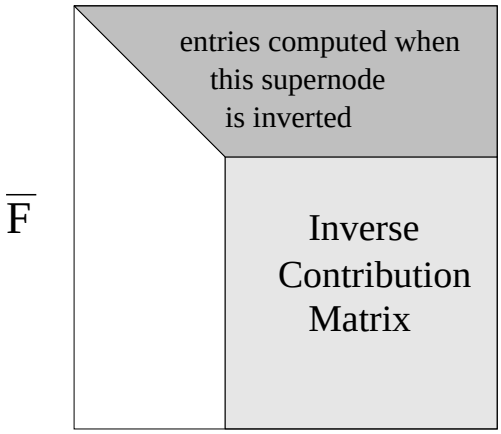

FIG. 2.1. *Entries computed when a supernode is inverted*

discussed in [10].

Section 2 presents a brief summary of the sequential inverse multifrontal algorithm presented in [3]. Various aspects of the block-partitioned algorithm including task partitioning and task dependencies, scheduling, and memory management are discussed in Section 3. Section 4 contains the results and discussion of the numerical experiments. Finally, some concluding remarks are given in Section 5.

2. The inverse multifrontal algorithm. In [3] the sequential computation of Zsparse for numerically symmetric matrices was formulated using an inverse multifrontal approach. We briefly review the basic features of this approach in preparation for the development of the parallel Zsparse algorithm. In the inverse multifrontal approach, every supernodal frontal matrix F_i formed in the LDU factorization of A, is mapped to a corresponding inverse supernodal frontal matrix $\overline{F}_i$. The column patterns of F_i and $\overline{F}_i$ are identical and are represented by $\mathcal{U}_i$. (Since A is symmetric, the frontal and inverse frontal matrices are also symmetric.) Entries in the inverse frontal matrices are entries of the inverse matrix (i.e. $(\overline{F})_{ij} = z_{ij}$). The index set $\mathcal{U}_i$ is partitioned into two disjoint sets: the pivotal index set $\mathcal{U}_i^{'}$, and a non-pivotal set $\mathcal{U}_i^{''}$. The row set is partitioned similarly. The partitioning of the column and row sets result in the partitioning of F_i and $\overline{F}_i$ into four block matrices: the (inverse) pivot block, the off-diagonal (inverse) pivot row and column blocks, and the (inverse) contribution block. Figure 2.1 shows the general structure and partitioning of F_i and $\overline{F}_i$. The inverse entries that are evaluated when an inverse supernode $\overline{F}$ is inverted are the entries in the upper triangular part of the inverse pivot row of $\overline{F}$.

If $\overline{\mathcal{F}}$ is the corresponding set of inverse frontal matrices then the set of inverse entries in Zsparse is given by,

$$\text{Zsparse} = \bigcup_{i \in \overline{\mathcal{F}}} \{ z_{kj} | k \in \mathcal{U}_i^{'}, j \in \mathcal{U}_i, k \leq j \} \tag{2.1}$$

where $z_{kj} \in (Z)_{kj}$ and $Z = A^{-1}$. To compute Zsparse we therefore need to invert all inverse frontal matrices in the set $\mathcal{F}$. Furthermore, in computing Zsparse, these are the only entries that need be computed since no entries from outside of the Zsparse set enter in the computation.

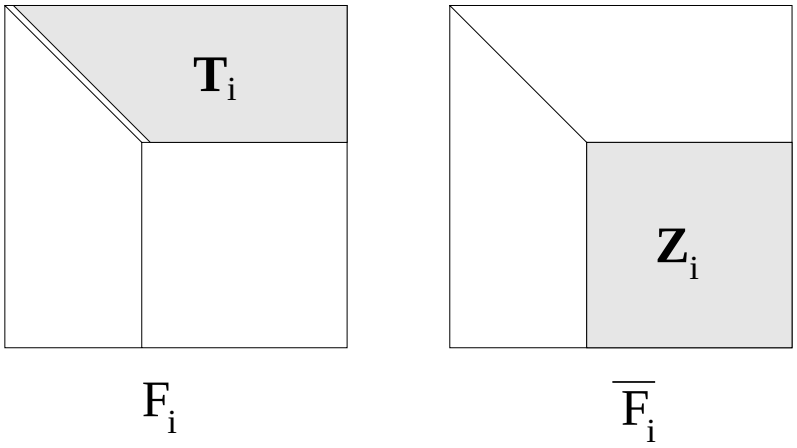

FIG. 2.2. *Relating* F_i, $\overline{F}_i$, $\mathbf{T}_i$, $\mathbf{Z}_i$

The direct u- and z-dependency sets were introduced to formulate the inversion of an inverse frontal matrix in terms of matrix-vector and matrix-matrix type of operations. The direct u-dependency and direct z-dependency of an inverse frontal matrix refer to the set of entries $u \in U$ and $z \in Z$, respectively, needed to invert the matrix. If $\mathbf{T}_i$ is the direct u-dependency set of $\overline{F}_i$ then,

$$\mathbf{T}_i = \{-u_{jk} | j \in \mathcal{U}_i^{'}, k \in \mathcal{U}_i, k > j\}.$$

That is, the numerical values of the entries in $\mathbf{T_i}$ are given by the negative values of the off-diagonal entries in the pivot row block of frontal matrix F_i. The equation for the direct z-dependency set $\mathbf{Z}_i$ is

$$\mathbf{Z}_i = \{z_{jk} | j, k \in \mathcal{U}_i^{''}\}.$$

It is easy to show that $\mathbf{Z_i}$ is equal to the set of entries in the inverse contribution block of $\overline{F}_i$. Figure 2.2 shows the matrices $\mathbf{T_i}$ and $\mathbf{Z_i}$ in relation to the structure of the frontal and inverse frontal matrices.

The matrix-vector form of the Takahashi equation for inverting a *simple* (one pivot row/column) inverse frontal matrix $\overline{F}_i$ can be expressed as

$$\boxed{\begin{array}{lcl} Z_i^{''} & = & \mathbf{T}_i * \mathbf{Z}_i \\ z_{ii} & = & D_{ii}^{-1} + \mathbf{T}_i * \mathbf{Z}_{ii} = D_{ii}^{-1} + \mathbf{T}_i * (Z_i^{''})^T \end{array}} \tag{2.2}$$

where $Z_i^{''}$ is the set of off-diagonal entries in the inverse pivot row. Note that the direct z-dependency set of z_{ii} is equal to $Z_i^{''}$.

The assembly tree used in the LDU factorization has its counterpart, the *inverse assembly tree*. The inverse assembly tree specifies the data dependencies among the inverse frontal matrices and is, therefore, used to guide the inversion process. Its structure is identical to that of the corresponding assembly tree, except that the node labels are labels of inverse frontal matrices and the direction of the dependency arrows are reversed, from parent to children instead of the other way around. A top-down traversal of the inverse assembly tree guarantees that the data dependencies among the inverse frontal matrices are satisfied. Furthermore, for any node i in the inverse

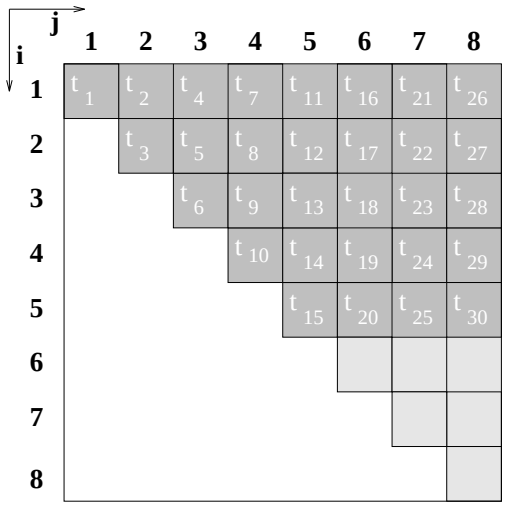

FIG. 3.1. *Partitioned supernode with node tasks* $t_1 \ldots t_{30}$

assembly tree, entries in the inverse contribution matrix of $\overline{F}_i$ can be assembled from the parent node of i in the inverse assembly tree.

3. The block-partitioned inverse multifrontal algorithm. The parallel block-partitioned Zsparse algorithm is based on a 2D (two-dimensional) partitioning of the tree nodes in the inverse assembly tree. Each tree node is divided into a set of node tasks that can be processed concurrently on different processors. This results in a higher level of concurrency than would otherwise be available by processing each tree node on one processor as a single task. Another advantage of the 2D partitioning scheme is that it allows us to control the sizes of the tasks allocated to processors, enabling more efficient load balancing to be done. We first discuss the 2D partitioning scheme, after which the scheduling and memory management policies are outlined. We then present the parallel algorithm and discuss its salient features.

Consider the supernodal inverse frontal matrix shown in Figure 2.1. We wish to evaluate the entries in the upper triangular part of the pivot row block of this node. The 2D partitioned form of this matrix is shown in Figure 3.1, where for simplicity we assume that the partition block size, b, is the same in both the **i** and **j** dimensions. The "staircase" diagonal is in anticipation of the parallel data structure used when inverting the matrix. Using the matrix-vector form of the Takahashi equation, Equation (2.2), we get the following set of equations to compute a block $Z_{\mathbf{ij}}$,

$$\boxed{\begin{array}{lcll} Z_{\mathbf{ij}} & = & D_{\mathbf{ij}}^{-1} + \sum_{k=i+1}^{j} \mathbf{T_{ik}} * Z_{\mathbf{kj}} + \sum_{k=j+1}^{m} \mathbf{T_{ik}} * (\mathbf{Z_{jk}})^T & \mathbf{i} \leq \mathbf{j} \\ (Z_{\mathbf{ij}})_{r*} & = & (Z_{\mathbf{ij}})_{r*} + (\mathbf{T_{ii}})_{[r,r+1:b]} * (Z_{\mathbf{ij}})_{[r+1:b,1,b]} & 1 \leq r < b \\ (Z_{\mathbf{ii}})_{rr} & = & (Z_{\mathbf{ii}})_{rr} + (\mathbf{T_{ii}})_{[r,r+1:b]} * (Z_{\mathbf{ii}})^T_{[r,r+1:b]} & 1 \leq r < b \end{array}} \tag{3.1}$$

where the bold-typed indices are local block indices, $m = m_1 + m_2$, m_1 and m_2 are defined by Equation (3.2), $1 \leq \mathbf{i} \leq m_1$, $1 \leq \mathbf{j} \leq m$, and $\mathbf{T_{ij}} = -U_{\mathbf{ij}}$.

$$\begin{array}{rcl} m_1 & = & \lceil |\mathcal{U}^{'}|/b \rceil \\ m_2 & = & \lceil |\mathcal{U}^{''}|/b \rceil \end{array} \tag{3.2}$$

In addition, we use the notation of Golub and Van Loan [8], where $X_{[s:t,u:v]}$ refers to a matrix X with row indices ranging from s to t, and column indices from u to v. If $s = t$ then $X_{[s,u:v]}$ is the row s vector; if $u = v$, $X_{[s:t,u]}$ is the column u vector; X_{r*} refers to the entries in row r of X.

Each block of entries $Z_{\mathbf{ij}}$ in Equation (3.1) represents a *node* task that can be allocated to a processor. The number of node tasks created per tree node using this partitioning scheme is $m_1(m_1+1)/2 + m_1 m_2$ (t_1 through t_{30} in Figure 3.1). Clearly, the number of these node tasks and, therefore, the degree of concurrency can be increased (decreased) by decreasing (increasing) the value of the block size.

3.1. Work partitioning. Processors are allocated work associated with processing tree nodes or the node tasks resulting from the partitioning of the tree nodes. Processing at the tree node level involves *memory allocation* and *inverse assembly*. For a given inverse frontal matrix, memory of size $b^2[(m_1+m_2)(m_1+m_2+1)/2)]$ is allocated as work area (m_1, m_2 defined by Equation (3.2). This assumes the upper triangular staircase diagonal data structure for the inverse frontal matrix shown in Figure 3.1. For the partitioned node in Figure 3.1 the memory required is $36b^2$: $30b^2$ for the node tasks and $6b^2$ for the inverse contribution block. Inverse assembly involves moving data from the parent node to the inverse contribution block area (no floating-point operations are required). The inverse assembly is guided by the inverse assembly tree.

Once memory allocation and inverse assembly are done for a tree node, the node level processing can begin. The tree node is implicitly partitioned as shown in Figure 3.1, where each of the dark-grey shaded squares represent a node task. Associated with each node task t_q is an (**i**,**j**) coordinate pair that correspond to the block indices used in Equation (3.1). For example, the coordinates for t_5 are (**2**,**3**). The work related to the processing of a node task having coordinates (**i**,**j**) involves computing the $b \times b$ block of entries $Z_{\mathbf{ij}}$ using Equation (3.1). The node level partitioning is done implicitly, in the sense that when a node task with coordinates (**i**,**j**) is scheduled for processing, the memory associated with processing that task is assumed to be located at coordinates (**i**,**j**) within the data structure allocated for the tree node to which that task belongs; no separate work area is setup for the node task. In addition, the precedence constraints for the node task are implicitly specified by its (**i**,**j**) coordinates.

3.2. Task dependencies. The *tree task* dependencies are specified by the inverse assembly tree. We use a breadth-first traversal of the inverse assembly tree to obtain tree tasks as the algorithm progresses. Two tree nodes are dependent if there is a parent-child relationship between them. This parent-child precedence relation occurs because the direct z-dependency set of the child node (its inverse contribution block) is a subset of the parent node. Accordingly, the parent node must be inverted prior to inverting any of its children nodes.

For a given tree node let m_1 and m_2 be defined by Equation (3.2). The following data dependencies among the *node tasks* can be readily determined from Equation (3.1). Let S_1 be the set of node tasks located within the inverse pivot block area, and S_2 the remaining set of node tasks (tasks outside of the inverse pivot block). In terms of m_1 and m_2, S_1 is equal to the set of node tasks with coordinates (**i**,**j**), where $1 \le \mathbf{i} \le \mathbf{j} \le m_1$, and S_2 is equal to the set with $1 \le \mathbf{i} \le m_1$ and $m_1 < \mathbf{j} \le m_1 + m_2$. For the example shown in Figure 3.1,

$$S_1 = \{t_i \mid 1 \le i \le 15\} = \{Z_{\mathbf{ij}} \mid 1 \le \mathbf{i} \le \mathbf{j} \le 5\},$$

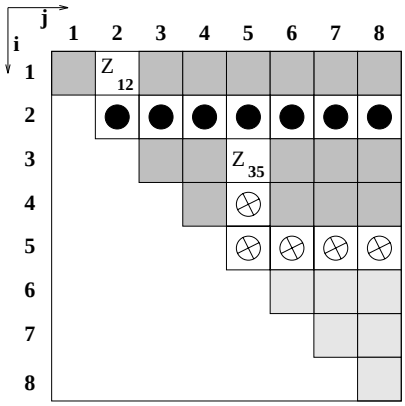

FIG. 3.2. *Node task dependencies for S_1 tasks $Z_{\mathbf{12}}$ and $Z_{\mathbf{35}}$. $Z_{\mathbf{12}}$ depends on tasks in locations with shaded circles; $Z_{\mathbf{35}}$ depends on tasks in positions with $\otimes$'s.*

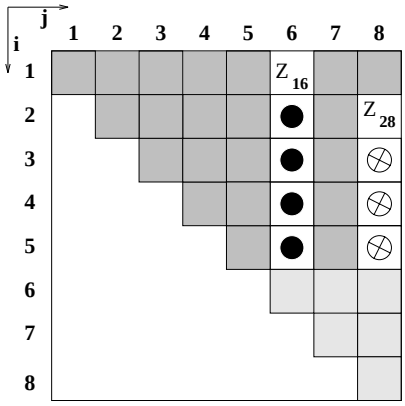

FIG. 3.3. *Node task dependencies for S_2 tasks $Z_{\mathbf{16}}$ and $Z_{\mathbf{28}}$. $Z_{\mathbf{16}}$ depends on tasks in locations with shaded circles; $Z_{\mathbf{28}}$ depends on tasks in positions with $\otimes$'s.*

and

$$S_2 = \{t_i \mid 16 \leq i \leq 30\} = \{Z_{\mathbf{ij}} \mid 1 \leq \mathbf{i} \leq 5, 6 \leq \mathbf{j} \leq 8\}.$$

The following dependency relationships exist among tasks in S_1 and S_2. All node tasks in S_2 are independent of all node tasks in S_1. A task in S_2 depends on a subset of other tasks in S_2. More specifically, task $Z_{\mathbf{ij}} \in S_2$ is dependent on $Z_{\mathbf{pq}} \in S_2$ if $\mathbf{j}=\mathbf{q}$ and $\mathbf{i} < \mathbf{p}$. That is, the two tasks must be in the same column and the dependent task must have a smaller row index. Tasks in S_2 also depend on data in the inverse contribution block; however, since these data are assembled prior to the execution of any node task, we do not consider these dependencies as node task dependencies.

On the other hand, a task in the inverse pivot block, an S_1 task, can depend on both S_2 tasks and other tasks within S_1. The same dependency relationship holds between two S_1 tasks as that between two S_2 tasks just discussed. For $Z_{\mathbf{ij}} \in S_1$ and $Z_{\mathbf{pq}} \in S_2$, $Z_{\mathbf{ij}}$ depends on $Z_{\mathbf{pq}}$ if $\mathbf{j}=\mathbf{p}$. Figures 3.2 and 3.3 illustrate some of these dependencies.

3.3. Task scheduling. We use the guided self-scheduling technique (GSS) [10] to assign *node tasks* to processors. The basic idea behind the GSS scheduling scheme is that a *variable-sized* group of tasks is assigned to a processor each time it becomes available. The number of tasks in the variable-sized group depends on the number of tasks left to be scheduled. If on the ith scheduling pass a processor finds R_i tasks left to be scheduled, then that processor is assigned $\lceil R_i/p \rceil$ tasks, where p is the number of processors assigned to the job. For example, if a tree node is partitioned into 30 node tasks and $p = 4$, the first processor to which tasks are scheduled gets $\lceil 30/4 \rceil = 8$ tasks; the second processor gets $\lceil (30-8)/4 \rceil = 6$ tasks; the third processor $\lceil (30-8-6)/4 \rceil = 4$ tasks; the next assignment is a group of 3 tasks, the other assignments are groups of 3,2,1,1,1, and 1, respectively. As shown in [10], by "guiding" (or adjusting) the amount of work given to a processor on a given scheduling pass, guided self-scheduling can result in both good load balancing and low scheduling overhead. A purely self-scheduling scheme would ordinarily assign a *fixed* number of tasks to every processor on each assignment. For the example just discussed, if this fixed number of tasks to assign is one, the number of scheduling assignments will be 30 compared to 10 used by the GSS scheme. This results in a higher scheduling overhead since some overhead is associated with each assignment as processors compete for access to the scheduling critical section, and set and unset locks. If the assignment size is increased the scheduling overhead will decrease but the overhead resulting from load imbalance can then increase. The GSS scheme has the property that the scheduling is adjusted to ensure a good balance between low scheduling overhead and balanced distribution of work to processors.

The GSS scheme can be parameterized. In the GSS(k) scheme, the k parameter ensuring that a processor is given at least k tasks per schedule. We use the GSS(2) scheme. In the example just discussed, the assignments with the GSS(2) scheme will be groups of 8, 6, 4, 3, 3, 2, 2, and 2 tasks respectively - a reduction in the number of accesses to the scheduling critical section by two as compared to the GSS(1) scheme. The GSS($k > 2$) scheme can reduce the scheduling overhead even further. Choosing k much greater than two, however, can severely reduce concurrency since fewer and fewer groups of tasks are available as k is increased, resulting in greater load imbalance, especially towards the end of the computation. We found GSS(2) a good choice for our application. For example, as the computation approaches the leaf nodes of the inverse assembly tree the nodes get smaller and more numerous, and the number of node tasks per tree node gets smaller (usually approaching about two or three node tasks per tree node, depending on the block size). With GSS(1), a tree node partitioned into two node tasks may be assigned to two different processors, incurring some synchronization overhead. With GSS(2), however, both of these node tasks will be assigned to a single processor, avoiding the synchronization overhead.

In our implementation a single task queue, Q, is maintained. At any one time Q may contain tree nodes (tree tasks) in one of three states (Figure 3.4). First, there tree nodes are placed into Q when their parent nodes have completed processing. No memory is yet allocated for these tree tasks, nor are their inverse contribution blocks assembled; these nodes are said to be in the *waiting* state. Second, there are tree nodes for which memory has been allocated and inverse assembly done, but no node task has yet been scheduled from these nodes; these nodes are in the *ready* state. Node tasks can be scheduled from these *ready* tree nodes at any time. Finally, there are tree nodes from which some node tasks have already been scheduled; these nodes are in the *running* state.

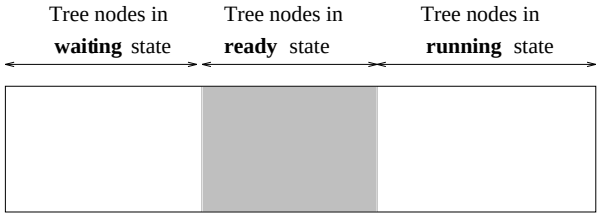

FIG. 3.4. *State of tree tasks in taskpool at any given time*

We adopt a policy that once scheduling has begun from a given tree task, all node tasks must be scheduled from that tree task until all its node tasks are depleted, before scheduling can start from another ready tree task. The rational here is to get a tree task completely processed as quickly as possible because the memory allocated to the tree task remains allocated until it is completely processed (all node tasks completed) and the inverse entries stored. By focusing the computation on a single tree node we reduce the number of tree tasks that are in the ready and running state and, consequently, the contention for memory that can severely limit concurrency. Another aspect of our scheduling is that on a given scheduling pass a processor is only assigned node tasks from a single tree task; that is, no processor is allowed to have scheduled work from two or more tree tasks at any given time. This helps to simplify the code logic associated with node task scheduling and identification.

Whenever a processor completes its assigned tasks, it retrieves a range of task-numbers from the tree node at the head of the task queue. This range of numbers specifies the next group of tasks to be processed by the processor. A given task-number (t) must first be mapped to its task-coordinates ($\mathbf{i}$,$\mathbf{j}$). With the GSS scheme this mapping is done outside of the scheduling critical section, resulting in a small scheduling time per assignment. Even though the mapping is done outside the critical section it is still essential that the cost associated with the computation of the coordinates be low. Thus as simple a mapping function as possible is desirable. We used the mapping function,

$$\boxed{\begin{array}{lcl} \mathbf{j} & = & \lceil (t/m_1) \rceil \\ \mathbf{i} & = & t - m_1 * (\mathbf{j} - 1) \end{array}} \tag{3.3}$$

where m_1 is given by (3.2). The necessity to keep the mapping function as simple as possible results in the creation of some "null" tasks in the lower triangular part of the inverse pivot block (see Figure 3.5). No memory is actually allocated for these node tasks, and the overhead associated with computing the coordinates for them and identifying them is very minimal. The mapping produced by this scheme for the tree node example in Figure 3.1 is shown in Figure 3.5. The null tasks are those in locations marked by the circles. Note that it is important that the combination of task-numbering and task-mapping produce task-coordinates that satisfy the dependency relationships discussed earlier. For example, the task at (**3**,**4**) in Figure 3.5 must have a task-number that is larger (18 in this case) than that at (**2**,**3**) (12 in this case), since $Z_{\mathbf{23}}$ depends on $Z_{\mathbf{34}}$.

3.4. Data structures. Each inverse frontal matrix is stored in the staircase upper triangular matrix form shown in Figure 3.1. Clearly, we only need an upper triangular matrix form, however, we decided to use some additional storage and do some unnecessary floating point operations for the diagonal blocks primarily to

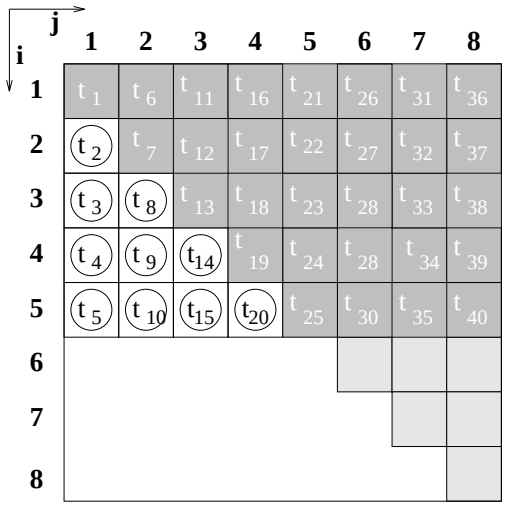

FIG. 3.5. *Partitioning with null tasks (in circles)*

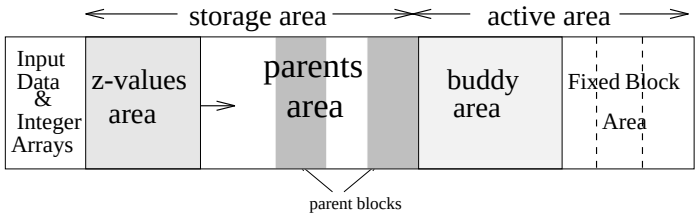

FIG. 3.6. *Memory layout*

simplify the code logic, but also to be able to use the rectangular level 3 BLAS rather than the triangular level 3 BLAS (the triangular BLAS tend to give lower performance). We also maintain a single *taskpool* for the set of tasks in Q mentioned earlier. This taskpool data structure is essentially a 1-dimensional array of size equal to the total number of inverse supernodes (typically much less than the dimension of the matrix).

3.5. Memory partitioning and management. The available memory is partitioned into two main regions, *storage* and *active*. This is essentially the same partitioning scheme used by Amestoy and Duff in [1]. Figure 3.6 shows the general layout of the two areas. The main difficulty is in deciding on the proportion of memory to allocate to the two areas. In a preprocessing step we compute the amount of memory required for the sequential algorithm and use this amount plus 5-10%. For most matrices we tested, the algorithm runs with this amount of memory with less than ten calls to the garbage collection procedure.

The storage area acts as the holding area for inverted parent nodes and permanent store for the computed inverse entries. A parent node is held here until the inverse contribution blocks of all of its children have been assembled. As Figure 3.6 shows, the z-value memory area increases monotonically (indicated by the right arrow) into the parent area. The part of the storage actually used to store parent nodes grows and shrinks as inverted nodes are added or removed. "Holes" can develop between

parent nodes in the storage area as parallel processing progresses, causing internal fragmentation. Garbage collection is done as the need arises to reduce the amount of internal fragmentation.

The primary motivation for the subdivision into storage and active area is that it allows garbage collection and some useful computation (inversion) to proceed concurrently. It may of course happen that a processor completes its work in the active area and needs to save the inverse entries or parent node just when garbage collection is in progress. It must then wait. To avoid significant overhead due to extended periods where processors are idle because garbage collection is in progress, we employed a mechanism for early exit from the garbage collection routine whenever one or more processors are waiting to enter the storage area.

It is possible to overwrite the input factors as the algorithm progresses and so avoid the use of the z-value area altogether. This overwriting of the factors can significantly reduce the memory requirement of the algorithm, and the number of calls to the garbage collection routine; an input parameter is used to control overwriting (the default being to not overwrite).

The active area is used as the computation area during the inversion of an inverse frontal matrix. Recall that a tree node in the taskpool is in the waiting, ready, or running state. The tree nodes in the ready and running states have already been allocated memory from this area. This memory can be accessed concurrently by different processors and synchronization is essential. As noted earlier, the active memory allocated to a tree node remains allocated until all processing on that tree node has completed. For the state of a tree node to change from waiting to ready enough memory needs to be available to accommodate the staircase upper triangular data structure of the node.

As in [1], the active memory is subdivided into a *buddy* area and a *fixed-block* area. The motivation behind this subdivision is to more efficiently use the available memory and, at the same time, lessen memory bottlenecks that would hamper concurrency. Given an amount of memory, M, for the active area, the buddy area can effectively only use $2^{(\lfloor log_2(M) \rfloor)}$ of this memory. The remainder, $M - 2^{(\lfloor log_2(M) \rfloor)}$, is used for the fixed-block area. We found that choosing the fixed-block size to be one-quarter the maximum inverse frontal matrix size gives good performance. Note that it is necessary for the size of the active area to be at least as large as the amount of memory required by the largest inverse frontal matrix.

The buddy area is managed using a buddy memory manager [9], while a very simple link-list manager is used for the fixed-block area. Memory can be allocated and/or deallocated concurrently from the buddy and fixed-block areas.

3.6. The parallel algorithm. An outline of the parallel block-partitioned Zsparse algorithm is given in Figure 3.7. The code for this algorithm is run on each processor (the Single Program Multiple Data model). The algorithm consists of six main steps. In step 1a, processor 0 prepares the first root node for scheduling and enters it into the taskpool in the *ready* state. This step simply involves allocating memory from the active memory area for the inverse frontal data structure; no inverse assembly is necessary for a root node. In the meantime, the other processors wait (step 1c) until processor 0 gives the signal in step 1b that a *ready* tree node is now in the taskpool.

All processors compete to enter the critical section to get a set of tasks via the guided self-scheduling scheme, step 2. Once a tree task in the ready or running state exists in the taskpool when a processor enters the GSS critical section, it will

```
      if (mytid = 0) then
1a:      make first root (treenode) ready and put it into taskpool
1b:      post start event
      else
1c:      wait until start event posted
      endif
      while (computing Zsparse)
         begin critical section
2:          use GSS(2) to get node tasks numbered t1 through t1 − s + 1
         end critical section
         if (node tasks assigned) then
            for t = t1 to t1 − s + 1
3a:            map t to (i,j) using Equation (3.3)
               if (i ≤ j) then
3b:               compute entries in block Zij using Equation (3.1)
                  if (i=1 and j=1) then
4a:                     put children of F̄ into taskpool
4b:                     if (number of children of F̄ > 0) save F̄
4c:                     save inverse entries and deallocate active memory
                  endif
               endif
            endfor
         else
            begin critical section
               if (waiting treenode in taskpool) then
5a:               get treenode
               else
5b:               get one of the remaining root treenode
               endif
            end critical section
            if (treenode found) then
6:             make treenode ready and put it into taskpool
            endif
         endif
      endwhile
```

FIG. 3.7. *The parallel Zsparse algorithm*

be assigned a group of node tasks. If no ready or running tree task is available in the taskpool, one or more processors will fail to get any work and will attempt to either make a *waiting* tree node in the taskpool ready (steps 5a and 6), or make one of the remaining root nodes ready and enter it into the taskpool (steps 5b and 6). Transforming a non-root node from the waiting state to the ready state involves both allocation of active memory and inverse assembly. If not enough active memory is available or garbage collection is in progress the processor must wait.

When a processor does get a set of node tasks, it processes them sequentially, in descending order of task numbers to ensure that the node task precedence relations are maintained. Processing of a node task begins with the mapping of the task number to task coordinates, step 3a. As noted earlier, we include a set of null tasks to simplify

the mapping function. These null tasks are easily identified ($\mathbf{i} > \mathbf{j}$). For the non-null tasks, processing continues with the evaluation of the inverse entries within the node task, step 3b. When the final task in the current tree node $\overline{F}$ is processed, its children tree nodes (obtained from the inverse assembly tree) are entered into the taskpool in the *waiting* state, step 4a. The upper triangular part of $\overline{F}$ is then put into the storage area, step 4b, for later use in the assembly of the inverse contribution blocks of its children. The inverse entries are then saved and the active memory associated with the inverse matrix deallocated, step 4c.

A number of enhancements can be made to this basic algorithm to optimize memory usage and CPU time. For example, instead of saving both the inverse entries and the inverted parent node in steps 4b and 4c, we can postpone saving the inverse entries until after all children of the inverted parent node have been processed and the inverted parent node *storage* memory is about to be deallocated. A further enhancement is to leave the inverted parent node in the fixed-block part of the active area if there is an excess of memory in this area. Care is needed here to avoid deadlock. Both of these enhancements result in reduced data movement (and memory traffic), and fewer calls to the garbage collection routine.

3.7. Computational cost. Let O_f be the number of multiply-add floating point operations involved in the inversion of an inverse frontal matrix $\overline{F}_f$. O_f can be easily computed from Equation (3.1):

$$\begin{aligned} O_f &= \sum_{j=1}^{m^f} \sum_{i=1}^{j} [\sum_{k=i+1}^{m^f} b^3] + (b^2(b-1)/2)(m_1^f(m_1^f-1)/2 + m_1^f * m_2^f) + b(b-1)m_1^f \\ &= b^3 m^f[(m^f)^2 - 1]/3 + [b(b-1)/4][b(m_1^f * m_2^f + (m_1^f)^2) + m_1^f(2-b)] \end{aligned}$$

where $m^f = m_1^f + m_2^f$ and m_1^f, m_2^f are defined by Equation (3.2). The total number of multiply-add floating point operations involved in computing Zsparse is then $\sum_{f=1}^{s} O_f$, where s is the total number of inverse frontal matrices. This theoretical operation count gives a lower bound on the number of floating point operations actually performed. For example, this count does not include the floating point operations done in computing m_1, m_2, or in computing the coordinates of a node task. We found the number of these additional floating point operations to be quite small compared to the theoretical count. We use the theoretical value in reporting megaflop rates in Section 4. The total number of required operations is identical to the number of operations required to factorize the matrix, as discussed in [3]. The theoretical operation count includes the unnecessary operations done in partially computing entries in the lower triangular part of the diagonal blocks. The ratio of the number of these unnecessary operations to the number of required operations per node is approximately b^2/m_a^3, where b is the block size and $m_a = (m_1 + m_2)/2$ is a measure of the size of the node. Once b is kept small relative to the average node size the required computation will overwhelmingly dominate the total operation count.

4. Numerical results and discussion. We implemented the parallel Zsparse algorithm on a CRAY-C98 at CRAY Research, Inc., a shared-memory, parallel-vector machine with 8 processors. It has a CPU clock cycle time of 4.167 ns, 2 vector pipelines, 512 million words (64 bits per word) of shared-memory partitioned into 512 banks, and 4 memory ports (2 read, 1 write, 1 I/O). The machine used the UNICOS 9.0bv operating system, the CF77 6.0.4.1 fortran compiler, the CRAY FPP 6.0 preprocessor, and the CRAY FMP 6.0.4.1 intermediate processor. Each processor has a theoretical peak rating of 1 billion floating point operations per second (1 Gflop). Hardware and software features are available that support task control,

TABLE 4.1
Statistics on matrices

matrix	n	$\frac{\lvert L\rvert}{\times 10^3}$	discipline	comments
PLAT1919	1919	83	oceanography	Atlantic and Indian oceans
BCSSTK25	15439	2426	structural eng.	76-story skyscraper
FINAN512	74752	9565	economics	portfolio optimization

TABLE 4.2
PLAT1919*:* $b = 16$

nproc	time	speedup	efficiency	pfrac	Mflops
1	0.18	1.00	1.00	-	27
2	0.10	1.90	0.95	0.95	51
3	0.06	2.80	0.93	0.96	75
4	0.05	3.66	0.91	0.97	98
5	0.04	4.53	0.91	0.97	122
6	0.03	5.23	0.87	0.97	141
7	0.03	5.73	0.82	0.96	154
8	0.03	6.26	0.78	0.96	168

synchronization of tasks, and critical sections. We used the macrotasking primitive TSKSTART to start the subroutine corresponding to algorithm in Figure 3.7 on separate processors. The overhead for the TSKSTART library call is fairly significant (about 250,000 clock cycles for the first call), but we used this call only once at the beginning of computation. The TSKWAIT library call was used to do a join when all supernodes were processed. A TSKWAIT call only carries an overhead of about 1400 clock cycles. We used the LOCKON and LOCKOFF primitives to control access to the critical sections (steps 2 and 5 of the algorithm). Each call to LOCKON or LOCKOFF costs about 4000 clock cycles. The total overhead due to the calls to LOCKON and LOCKOFF can be significant; the GSS scheme, however, helped in reducing this cost.

We used the EVPOST, EVWAIT, EVTEST, and EVCLEAR primitives to enforce the precedence constraints among the node tasks; a node task would wait until one of the node tasks on which it depends posted a completion event. Rather than having to sequentially cycle through the set of tasks on which it depends, doing an EVTEST call on each one to find one that has posted a completion event, we thought that this could have been more efficiently implemented if there was a *wait any* type of event wait primitive that allows a processor to be awakened by any of a pre-identified set of events. Fortunately, the EVTEST library call is relatively cheap, about 200 clock cycles.

We present the results for runs done on of three matrices shown in Table 4 [7]: PLAT1919, BCSSTK25 and FINAN512. The factors and assembly tree information were obtained using a modified version of a modified form of UMFPACK with strict diagonal pivoting [6, 5], although in principal any supernodal or multifrontal factorization algorithm for symmetric matrices could generate the supernodal factors.

Tables 4.2 through 4.10 give the results for these runs. The tables list the number of processors, the wall clock time in seconds, the speedup as compared to the run time with one processor, the efficiency (speedup divided by the number of processors), the

TABLE 4.3
PLAT1919: $b = 32$

nproc	time	speedup	efficiency	pfrac	Mflops
1	0.16	1.00	1.00	-	31
2	0.08	1.90	0.95	0.95	59
3	0.06	2.83	0.94	0.97	87
4	0.04	3.65	0.91	0.97	113
5	0.04	4.12	0.82	0.95	127
6	0.03	4.76	0.79	0.95	147
7	0.03	5.40	0.77	0.95	167
8	0.03	6.02	0.75	0.95	186

TABLE 4.4
PLAT1919: *no blocking*

nproc	time	speedup	efficiency	pfrac	Mflops
1	0.15	1.00	1.00	-	32
2	0.08	1.93	0.96	0.96	61
3	0.05	2.85	0.95	0.97	91
4	0.04	3.58	0.89	0.96	114
5	0.04	4.32	0.86	0.96	138
6	0.03	4.99	0.83	0.96	159
7	0.03	5.60	0.80	0.96	179
8	0.03	6.10	0.76	0.96	195

TABLE 4.5
BCSSTK25: $b = 16$

nproc	time	speedup	efficiency	pfrac	Mflops
1	8.30	1.00	1.00	-	108
2	4.30	1.93	0.96	0.96	208
3	2.90	2.86	0.95	0.98	309
4	2.23	3.73	0.93	0.98	402
5	1.84	4.52	0.90	0.97	488
6	1.58	5.24	0.87	0.97	566
7	1.39	5.96	0.85	0.97	643
8	1.28	6.50	0.81	0.97	702

TABLE 4.6
BCSSTK25: $b = 32$

nproc	time	speedup	efficiency	pfrac	Mflops
1	5.60	1.00	1.00	-	160
2	2.91	1.93	0.96	0.96	308
3	2.05	2.73	0.91	0.95	436
4	1.58	3.54	0.88	0.96	565
5	1.29	4.36	0.87	0.96	696
6	1.12	4.99	0.83	0.96	798
7	1.00	5.60	0.80	0.96	896
8	0.93	6.03	0.75	0.95	965

TABLE 4.7
BCSSTK25: *no blocking*

nproc	time	speedup	efficiency	pfrac	Mflops
1	4.90	1.00	1.00	-	160
2	2.52	1.94	0.97	0.97	310
3	1.72	2.86	0.95	0.97	456
4	1.36	3.62	0.90	0.96	578
5	1.11	4.41	0.88	0.97	705
6	0.96	5.11	0.85	0.96	816
7	0.86	5.67	0.81	0.96	907
8	0.78	6.32	0.79	0.96	1011

TABLE 4.8
FINAN512: $b = 16$

nproc	time	speedup	efficiency	pfrac	Mflops
1	98.30	1.00	1.00	-	137
2	51.57	1.91	0.95	0.95	261
3	35.62	2.76	0.92	0.96	378
4	27.58	3.56	0.89	0.96	488
5	22.55	4.36	0.87	0.96	597
6	18.96	5.18	0.86	0.97	710
7	16.88	5.82	0.83	0.97	797
8	15.30	6.42	0.80	0.96	880

TABLE 4.9
FINAN512: $b = 32$

nproc	time	speedup	efficiency	pfrac	Mflops
1	52.20	1.00	1.00	-	254
2	27.68	1.89	0.94	0.94	479
3	20.14	2.59	0.86	0.92	658
4	16.48	3.17	0.79	0.91	804
5	13.72	3.80	0.76	0.92	966
6	11.55	4.52	0.75	0.93	1147
7	10.30	5.07	0.72	0.94	1287
8	9.43	5.54	0.69	0.94	1406

TABLE 4.10
FINAN512: *no blocking*

nproc	time	speedup	efficiency	pfrac	Mflops
1	63.20	1.00	1.00	-	238
2	34.20	1.85	0.92	0.92	439
3	24.73	2.56	0.85	0.91	608
4	19.43	3.25	0.81	0.92	773
5	16.63	3.80	0.76	0.92	904
6	14.79	4.27	0.71	0.92	1016
7	13.83	4.57	0.65	0.91	1087
8	12.72	4.97	0.62	0.91	1182

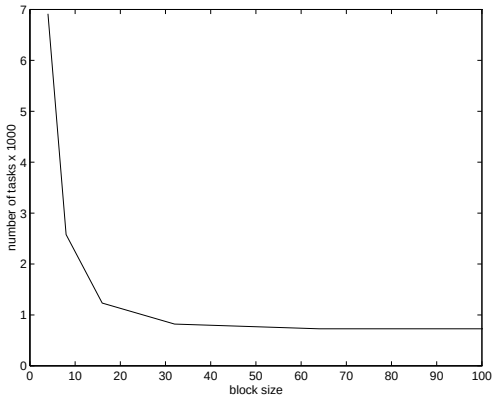

FIG. 4.1. PLAT1919*: variation of number of tasks as* b *changes*

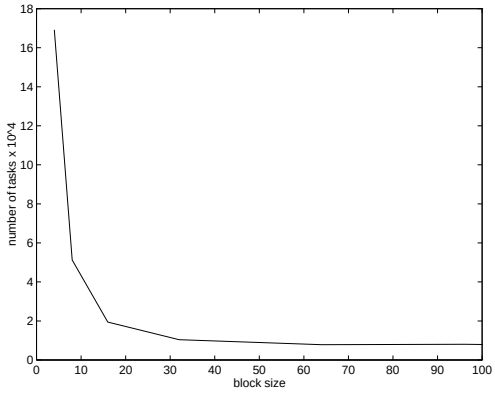

FIG. 4.2. BCSSTK25*: variation of number of tasks as* b *changes*

fraction of time spent in parallel sections of the code (from Amdahl's law, equal to $(p - p/s)/(p - 1)$, where s the speedup when using p processors), and the megaflop rate (theoretical number of floating point operations performed divided by the wall clock time). Each table catalogues the results for runs done with one value of the the blocking parameter b (no blocking simply means that b was set to a large value). Clearly, the blocking factor controls the degree of parallelism obtainable within a given tree node. The higher the value to which b is set the smaller the number of node-level tasks, and therefore the lower the level of concurrency. Figures 4.1, 4.2, and 4.3 show the rapid decrease in the number of node tasks created as the size of the blocking factor is increased. For the PLAT1919 matrix (Figure 4.1) setting the blocking factor much greater than 16 severely restricts the amount of node-level parallelism available. For the BCSSTK25 and FINAN512 matrices (Figures 4.2 and 4.3) $b = 32$ is about the maximum value that the blocking factor can be set to before node-level concurrency plays a secondary role to the tree-level concurrency. Although smaller values of b favor higher levels of concurrency and less overhead due to load balancing problems, the level 2 and level 3 BLAS perform more efficiently for larger values of b. For optimal performance b must be set to a value that balances these two effects. The results for all three matrices show the general decreasing of the wall clock times and increasing

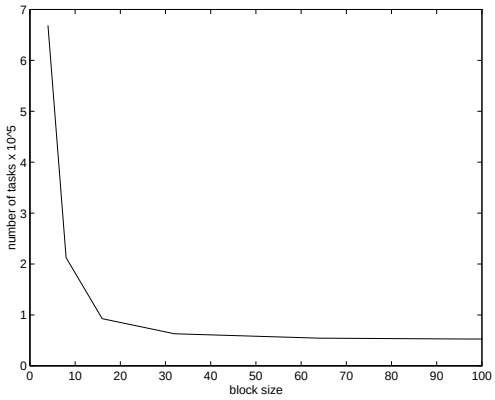

FIG. 4.3. FINAN512: *variation of number of tasks as b changes*

megaflop rates (for given number of processors) as the block size is increased. This is due to the effect of the block size on the efficiency of the BLAS primitives. The speedup and efficiency, however, are higher for the smaller sized blocks. This can be attributed to the loading balancing effect of the block size: the smaller the blocks the easier it is to balance the work done by the processors. Finally, we see that our code is running in parallel mode at least 91% of the time for all three matrices.

5. Conclusions. The numerical results clearly show that the inverse multifrontal Zsparse algorithm can be successfully parallelized on a shared-memory platform. The 2D partitioning scheme results in a very scalable algorithm in which load balancing can be effectively done. The use of the guided self-scheduling scheme was instrumental in keeping the scheduling overhead low and in maintaining balanced distribution of work among the processors. Some care is required, however, in the choice of blocking factor to ensure a high level of concurrency and good performance of the BLAS primitives. We recommend a block size of 16 for small to medium-sized problems and a block size of 32 for the larger problems.

6. Acknowledgments. Support for this project was provided by the National Science Foundation (DMS-9223088 and DMS-9504974), and by CRAY Research, Inc., through the allocation of supercomputing resources.

REFERENCES

[1] P. R. Amestoy and I. S. Duff. Memory management issues in sparse multifrontal methods on multiprocessors. *The International Journal of Supercomputer Applications*, 7(1):64–82, Spring 1993.

[2] Y. E. Campbell. *Multifrontal algorithms for sparse inverse subsets and incomplete LU factorization*. PhD thesis, Computer and Information Science and Engineering Department, Univ. of Florida, Gainesville, FL, November 1995. Also CISE Technical Report TR-95-025.

[3] Y. E. Campbell and T. A. Davis. Computing the sparse inverse subset: an inverse multifrontal approach. Technical Report TR-95-021, University of Florida, Gainesville, FL, 1995.

[4] Y. E. Campbell and T. A. Davis. On computing an arbitrary subset of entries of the inverse of a matrix. Technical Report TR-95-022, Computer and Information Science and Engineering Department, Univ. of Florida, 1995.

[5] T. A. Davis and I. S. Duff. A combined unifrontal/multifrontal method for unsymmetric sparse matrices. Technical Report TR-95-020, Computer and Information Science and Engineering Department, Univ. of Florida, 1995.

[6] T. A. Davis and I. S. Duff. An unsymmetric-pattern multifrontal method for sparse LU factorization. *SIAM J. Matrix Analysis and Application*, (to appear). Also CISE Technical Report TR-94-038.

[7] I. S. Duff, R. G. Grimes, and J. G. Lewis. Users' guide for the Harwell-Boeing sparse matrix collection (release 1). Technical Report RAL-92-086, Rutherford Appleton Laboratory, Didcot, Oxon, England, Dec. 1992.

[8] G. H. Golub and C. F. van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, MD and London, UK, second edition, 1990.

[9] T. Johnson and T. A. Davis. Parallel buddy memory management. *Parallel Processing Letters*, 2(4):391–398, 1992.

[10] C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, C-36(12):1425–1439, December 1987.

[11] K. Takahashi, J. Fagan, and M. Chin. Formation of a sparse bus impedance matrix and its application to short circuit study. *8th PICA Conference Proc., Minneapolis, Minn*, pages 177–179, June, 4-6 1973.

Note: all University of Florida technical reports in this list of references are available in postscript form via anonymous ftp to `ftp.cis.ufl.edu` in the directory `cis/tech-reports`, or via the World Wide Web at `http://www.cis.ufl.edu/~davis`.