PARALLEL SYMBOLIC FACTORIZATION FOR SPARSE LU WITH STATIC PIVOTING

L. GRIGORI[1] J.W. DEMMEL[2] X.S. LI[3]

[1]*INRIA Rennes, Campus Universitaire de Beaulieu, Avenue du General Leclerc Rennes, 35042, France. email: Laura.Grigori@irisa.fr*

[2]*Computer Science Division and Mathematics Department, UC Berkeley, CA 94720-1776, USA. email: demmel@cs.berkeley.edu*

[3]*Lawrence Berkeley National Laboratory, One Cyclotron Road, Mail Stop 5 0F-1650 Berkeley, CA 94720-8139, USA. email: XSLi@lbl.gov*

Abstract. This paper presents the design and implementation of a memory scalable parallel symbolic factorization algorithm for general sparse unsymmetric matrices. Our parallel algorithm uses a graph partitioning approach, applied to the graph of $|A|+|A|^T$, to partition the matrix in such a way that is good for sparsity preservation as well as for parallel factorization. The partitioning yields a so-called separator tree which represents the dependencies among the computations. We use the separator tree to distribute the input matrix over the processors using a block cyclic approach and a subtree to sub-processor mapping. The parallel algorithm performs a bottom up traversal of the separator tree. With a combination of right-looking and left-looking partial factorizations, the algorithm obtains one column structure of L and one row structure of U at each step. The algorithm is implemented in C and MPI. From a performance study on large matrices, we show that the parallel algorithm significantly reduces the memory requirement of the symbolic factorization step, as well as the overall memory requirement of the parallel solver. It also often reduces the runtime of the sequential algorithm, which is already relatively small. In general, the parallel algorithm prevents the symbolic factorization step from being a time or memory bottleneck of the parallel solver.

1. Introduction.

We consider solving a sparse unsymmetric system of linear equations $Ax = b$ using a direct method, which first factorizes the matrix A into the product of a unit lower triangular matrix L and an upper triangular matrix U, then solves $LUx = b$ via two triangular solutions. One distinct property of sparse LU factorization is the notion of fill-in. That is, a zero element in the original matrix A can become nonzero in L and U. The purpose of *symbolic factorization* is to compute the nonzero structure of the factors L and U, which contains the original nonzero elements in A as well as the filled elements. When there is no need to pivot during numerical factorization, the nonzero structure of L and U can be computed without referring to the numerical values of the matrix. This allows us to allocate sufficient memory and to organize computations before numerical factorization. Computationally, symbolic factorization is usually much faster than numerical factorization, and hence there has not been much motivation to parallelize this step. However, as numerical factorization has been made more and more scalable, symbolic factorization can become a serious bottleneck. In particular, a sequential algorithm requires the nonzero structure of the matrix A to be assembled onto one processor, which leads to a serious bottleneck in memory (if not timewise). We have therefore designed and implemented a memory-scalable symbolic factorization algorithm for distributed memory machines. The parallel algorithm and the software are suitable for sparse LU factorization with static pivoting [20].

We developed this parallel algorithm in the context of the SuperLU_DIST [20] solver, which is a widely used sparse direct solver for distributed memory parallel computers. SuperLU_DIST contains several distinct steps in the solution process:

1. Choose a permutation matrix P_1 and diagonal matrices D_1 and D_2 so that the diagonal entries of $P_1D_1AD_2$ are ± 1 and the magnitudes of the off-diagonal entries are bounded by 1. The above equilibration and permutation help ensure accuracy of the computed solution. Currently, this is achieved using the HSL routine MC64 [6]. The purpose of the large-diagonal permutation is to decrease the probability of encountering small pivots during factorization.
2. Reorder the equations and variables by using a heuristic that chooses a per-

mutation matrix P_2 so that the number of fill-in elements in the factors L and U of $P_2 P_1 D_1 A D_2 P_2^T$ is reduced.

3. Perform the symbolic factorization to identify the locations of the nonzero entries of L and U.
4. Compute the numerical values of L and U.
5. Perform triangular solutions.
6. Perform a few steps of an iterative method like iterative refinement if the solution is not accurate enough.

In the current version of SuperLU_DIST, the numerical factorization and triangular solutions are performed in parallel, but the first three steps are done sequentially. Our new parallel symbolic factorization algorithm uses ParMetis [18] (parallel Metis) to find a sparsity-preserving ordering (second step), and performs the subsequent structural analysis in parallel using a distributed input matrix (third step). Therefore, once we have integrated this algorithm in the solver, the only remaining sequential bottleneck will be in the first step, for which parallelization work is underway [21].

There are few results published in the literature on the parallelization of the symbolic factorization. Most of them consider the Cholesky factorization of symmetric positive definite (SPD) matrices. For these matrices, the symbolic factorization can be expressed by a tree structure, called the elimination tree, that represents the transitive reduction of the graph of the Cholesky factor L. The approach used in [23, 24] consists of computing in parallel first the structure of the elimination tree and then the symbolic factorization. While the latter step can be easily parallelized, the parallelization of the former is a difficult task and leads to modest speedup. A parallel symbolic factorization algorithm is also presented in [16] and is part of the PSPASES direct solver for SPD matrices. The algorithm is driven by the two-dimensional distribution of the data (designed for the numerical factorization step) and the elimination tree of A. The authors report that the parallel symbolic factorization takes relatively little time compared to the other steps of the solver.

In the case of unsymmetric matrices, the transitive reduction of the graphs of L and U leads to general directed acyclic graphs. The parallelization of the symbolic factorization of unsymmetric matrices is even more difficult than that of symmetric matrices, since it does not get the benefit of using a tree structure. A parallel algorithm for the factorization of unsymmetric matrices is presented in [2]. It is based on a sequential algorithm that exhibits ideal parallelism, but has two main drawbacks. It needs the input matrix to be replicated on every processor's local memory, and hence is not scalable in memory. The sequential algorithm on which the parallel formulation is based is much slower in practice than more recent and improved sequential symbolic factorization algorithms.

The difficulty in formulating a scalable parallel algorithm lies in the small computation to communication ratio and in the sequential data dependency. Our primary goal in this research is to develop a parallel algorithm that provides memory scalability. At the same time, even if the serial runtime of the symbolic factorization is already relatively small, we still want to achieve reasonable speedup, otherwise it will dominate runtime with larger number of processors. Our purpose is to obtain enough speedup to prevent this step from being a computational bottleneck. For this, our algorithm exploits two types of parallelism. The first type of parallelism relies on a graph partitioning approach, to reduce the fill-in and to permute the matrix to a suitable form for parallel execution. The graph partitioning is applied on the symmetrized matrix $|A| + |A|^T$, where $|A|$ denotes the matrix of absolute values of the entries of matrix A. The second type of parallelism is based on a block cyclic distribution of the matrix, which is a standard technique used in parallel matrix computations. To further decrease the runtime, we exploit the fact that as the elimination proceeds, the remaining submatrices in L and U become progressively full, and at some point we can treat the remaining submatrix as a full matrix.

The choice of using a graph partitioning approach is in accordance with the state-of-the-art in parallelizing the ordering algorithms. The algorithms based on a local greedy strategy, such as minimum-degree ordering algorithm, have proved to be difficult to parallelize. For example, the parallelization of one of its variants was discussed by Chen et al. [1]. The parallel algorithm was designed for shared memory machines, and only limited speedups were obtained. The main sources of inefficiency come from insufficient parallelism and load imbalance. Nested dissection, on the other hand, is an algorithm that uses a top-down divide-and-conquer paradigm. It first computes a vertex separator of the entire graph that divides the matrix into two disconnected parts. The matrix is reordered such that the variables corresponding to the separator are ordered after those in the disconnected parts. This splitting-reordering process is then recursively applied to the two disconnected parts, respectively. The main advantage is that the reordered form of the matrix is suitable for parallel computation. The state-of-the-art nested dissection algorithms use multilevel partitioning [13, 14, 17]. A widely used, highly parallel code is ParMetis [18], which we will use as our ordering front-end preceding the symbolic factorization.

The rest of the paper is organized as follows. Section 2 briefly reviews several sequential symbolic factorization algorithms. Section 3 discusses the parallel symbolic factorization algorithm. Section 4 describes several techniques to identify dense submatrices in the factors L and U and thus reduce the runtime of the symbolic factorization. Section 5 presents the experimental results obtained when applying the algorithm on real world matrices and Section 6 concludes the paper.

2. Unsymmetric symbolic factorization.

In this section we present briefly the various sequential algorithms described in the literature [7, 10, 11, 22], with an emphasis on the algorithm used in our parallel approach. The differences between the existing algorithms mostly lie in three aspects: 1) whether the algorithm uses the structure of A alone or also uses the partly computed structure of L and U; 2) different ways to improve the runtime of the algorithm; and 3) whether the algorithm computes the nonzero structure of the factors by rows, by columns or by both rows and columns. As noted in [10], theoretically none of the algorithms dominates the other. In practice, their complexity is greater than $nnz(L+U)$, but is much smaller than the number of floating-point operations required for numerical factorization. However, the experiments performed on small sized matrices [10] and discussion of aspect 1) showed that the algorithms that use the structure of A alone are slower than the algorithms that use the structure of A and the partly computed structure of L and U. The other results addressing aspect 2) showed that improvements based on approaches like supernodes, symmetric pruning (we will describe these approaches later in this section) significantly reduced the symbolic factorization time.

Let A be a sparse unsymmetric $n \times n$ matrix and let $nnz(A)$ denote the number of its nonzero elements. In what follows, $Str(A)$ refers to the nonzero structure of the matrix A. a_{ij} represents the element of row i and column j of matrix A, while $A(k:r, i:j)$ represents the submatrix consisting of rows k through r and columns i through j of A. $G(A)$ denotes the directed graph associated with matrix A. This graph has n vertices and an edge (i,j) for each nonzero element a_{ij}.

Our parallel algorithm is based on a variant that computes progressively the nonzero structure of L by columns and that of U by rows, using at each step the partial structure previously computed. That is, the structure of the ith column of L is formed by the union of $Str(A(i:n,i))$ with a subset of columns of L before i. This subset of columns is determined from $Str(U(:,i))$. And the structure of the ith row of U is given in a similar way by the union of $Str(A(i,i:n))$ with a subset of the previous rows of U. The following theorem states how the structure of L can be determined by those of A and U.

Theorem 2.1 (Rose and Tarjan [22]).

$$Str(L(:,i)) = Str(A(i:n,i) \cup \bigcup \{Str(L(i:n,j)) \mid (j,i) \textit{ is an edge of } G(U(1{:}i\text{-}1,:))\}$$

To decrease the runtime of the algorithm, we use two different improvements. The first improvement, called symmetric reduction or symmetric pruning, was proposed by Eisenstat and Liu [7]. It is based on the observation from Theorem 2.1 that only the structures of a subset of the columns $L(:,j)$ need to be considered in computing $Str(L(:,i))$. This subset is determined by the symmetric elements of the factors, that are used to remove some (but not all) redundant edges in the graphs of L and U. For example, if both l_{kj} and u_{jk} are nonzero and $k < i$, we must have $Str(L(k:n,j) \subseteq Str(L(:,k))$. Then, when computing $Str(L(:,i))$, to avoid redundant work, it is sufficient to consider only $Str(L(:,k))$. The symmetric reduction of the structure of $U(j,:)$ (structure of $L(:,j)$) is obtained by removing all nonzeros u_{jt} (nonzeros l_{tj}) for which l_{kj} and u_{jk} are nonzeros and $t > k$.

The second improvement is based on the notion of supernode, used in one of the more recent algorithms [4], and also in SuperLU_DIST [20]. In SuperLU_DIST, the symbolic factorization computes the structure of L and U by columns. A supernode groups together successive columns of L with the same nonzero structure, so they can be treated as a dense submatrix. At the end of each step, the symbolic factorization algorithm decides whether the new column i of L belongs to the same supernode as column $i-1$ of L. If it does, then it is not necessary to store the structure of column i in memory. That is, for each supernode, only the structure of its first column is stored. This leads to an important reduction in the memory needs of the symbolic factorization. Furthermore, supernodes also help reduce the runtime of the symbolic factorization. This is because to compute $Str(L(:,i))$, the previous supernodes are used instead of the columns $1,\ldots,i-1$.

In our variant, a supernode groups together successive columns of L having the same structure and successive rows of U having the same structure. This definition is appropriate for our algorithm that computes the structure of L by columns and that of U by rows. The symmetric pruning for each supernode consists of finding the first symmetric nonzero that does not belong to the diagonal block of the supernode. This is equal to the first symmetric nonzero of the last column of L and last row of U of the supernode.

From a memory point of view, the usage of supernodes leads to a decrease in the memory needs of the symbolic factorization algorithm. From a computation point of view, the supernodes used in our variant do not lead to removing more redundant work in Theorem 2.1 than symmetric pruning alone does. For example, consider the computation of the structure of L and a supernode formed by indices $s, s+1, \ldots t$. Denote by i the first symmetric nonzero of this supernode, where $i > t$. When supernodes are used, the structure of the first column of the supernode $L(:,s)$ is used in the computation of the structure of all the columns $L(:,j)$, with $t < j \leq i$ and $u_{sj} \neq 0$. Consider now that only symmetric pruning is used. For every index k such that $s \leq k < t$, the first symmetric nonzero corresponds to index $k+1$, that is $l_{k+1,k}$ and $u_{k,k+1}$ are nonzero. Hence, all the entries after $k+1$ will be pruned. And the symmetric pruning information corresponding to column t of L and row t of U is the same as the symmetric pruning information corresponding to the supernode. For all the indices j with $t < j \leq i$ and $u_{tj} \neq 0$, the structure of column $L(:,t)$ is included in the structure of all the columns $L(:,j)$. But the structure of column t of L is the same as the structure of column s of L and the structure of row t of U is the same as the structure of row s of U. Hence, computing the structure of $L(:,j)$ involves the same number of operations when symmetric pruning is used or when supernodes and symmetric pruning are used. However, since the symmetric pruning information corresponding only to the first column of L and row of U of a supernode is stored, the usage of supernodes helps decrease the time and the memory necessary for storing the symmetric pruning information.

Algorithm 1 presents this variant in a so-called left-looking approach. When it arrives at the ith step of computation, $Str(L(:,1:i-1))$ and $Str(U(1:i-1,:))$ are known. $Str(L(:,i))$ is formed using column i of the pruned structure of $U(1:i-1,:)$ (denoted as $prU(1:i-1,:)$). $Str(U(i,:))$ is formed using row i of the pruned structure of $L(:,1:i-1)$ (denoted as $prL(:,1:i-1)$). Then the first symmetric nonzero in $Str(L(:,i))$ and $Str(U(i,:))$ is determined. The pruned column i of L is added to prL and the pruned row i of U is added to prU. For the sake of clarity, this algorithm does not use supernodes. In Section 3 we will also use a right-looking approach, in which for example for the structure of L, at the ith step of computation, the structure of column i is computed and then is used to update the partially computed structure of columns $i+1$ to n that need it.

Algorithm 1 Left-Looking symbolic factorization

```
for $i := 1$ to $n$ do
    Compute $Str(L(:,i)) =$
    $Str(A(i:n,i)) \cup \bigcup \{Str(L(i:n,j)) \mid (j,i) \text{ is an edge of } G(prU(1:i-1,:))\}$
    Compute $Str(U(i,:)) =$
    $Str(A(i,i:n)) \cup \bigcup \{Str(U(j,i:n)) \mid (i,j) \text{ is an edge of } G(prL(:,1:i-1))\}$
    $k := n$
    determine $k := \min(j)$ such that $l_{ji}$ and $u_{ij}$ are nonzeros
    add $Str(L(i:k,i))$ to $prL$ and $Str(U(i,i:k))$ to $prU$
end for
```

The time spent in step 1 of Algorithm 1 is bounded by the number of operations to multiply the matrix $L(:,1:i-1)$ by the vector $prU(:,i)$ at each step i, which is written informally as $ops(L \cdot prU)$ [10]. Similarly, the time spent in step 2 is bounded by $ops(prL \cdot U)$. The complexity of the entire algorithm is dominated by these two steps.

We illustrate Algorithm 1 in Figure 2.1 on an example matrix A with nonzero diagonal and of order 8. The matrix at the left shows the results obtained when only symmetric pruning is used, while the matrix at the right shows how Algorithm 1 performs when supernodes are used in adition to symmetric pruning. We suppose that the first 5 columns of L and the first 5 rows of U have been computed and that the algorithm arrives at the computation of the 6th column of L. The dark circles represent nonzero elements of the original matrix A and the empty circles represent fill-in elements of the factors. The elements belonging to prL and prU are doubly circled.

Consider first that only symmetric pruning is used (left of Figure 2.1). For example, l_{32} and u_{23} are nonzeros, then for the index 2, only these two elements belong to prL and prU. The structure of $prU(:,6)$ is formed by the elements u_{16} and u_{56}. Hence, to compute the structure of $L(:,6)$, the algorithm forms the union of structures of $A(6:8,6)$, $L(6:8,1)$ and $L(6:8,5)$.

Consider now that supernodes are used in addition to symmetric pruning (right of Figure 2.1). Nodes 2 and 3 form a supernode, because columns 2 and 3 of L have the same structure and rows 2 and 3 of U have the same structure. Similarly, nodes 4 and 5 form another supernode. Symmetric pruning information corresponding only to the first column of L and row of U of a supernode is added to prL and prU. For example, l_{42} and u_{24} are nonzeros, then only elements l_{42} and u_{24} belong to prL and respectively prU (we store only the indices off the diagonal block). The symmetric nonzeros considered for pruning are l_{42} and u_{24}, because 4 is the first node not in the supernode formed by 2 and 3.

To compute the structure of $L(:,6)$, the algorithm forms the union of structures of $A(6:8,6)$, $L(6:8,1)$ and $L(6:8,4)$. Since indices 4 and 5 belong to the same supernode, the structure of $L(6:8,4)$ is the same as the structure of $L(6:8,5)$. Hence, from a computation point of view, forming the structure of column 6 of L

using supernodes and symmetric pruning is equivalent to the previous approach that uses only symmetric pruning. But the usage of supernodes decreases the memory needs. Only the columns of L and rows of U corresponding to indices $1, 2$ and 4 are stored.

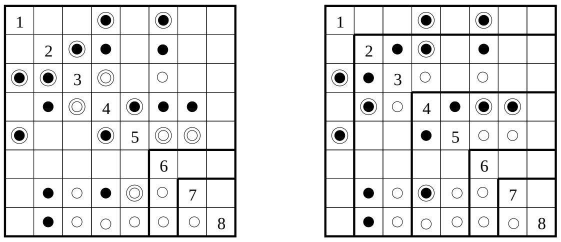


FIG. 2.1. *Example matrix A that illustrates Algorithm 1 using only symmetric pruning (left) and using symmetric pruning and supernodes (right).*

3. Parallel unsymmetric symbolic factorization.

We now describe our parallel unsymmetric symbolic factorization algorithm. Our algorithm uses the partly computed $Str(L)$ and $Str(U)$ as the algorithm progresses and exploits two types of parallelism. The first type comes from the graph partitioning approach, which is used as a fill-reducing ordering step. This approach partitions the matrix in a suitable way for parallel factorization and also provides a tree structure representing the dependencies among the computations. The second type of parallelism depends on a block cyclic distribution of the data.

The partitioning approach uses the graph of the symmetrized matrix $|A| + |A|^T$ to identify a vertex separator that splits the graph into two disconnected subgraphs. If the vertices of the separator are ordered after the vertices in the disconnected subgraphs, a blocked matrix suitable for parallel computation is obtained. The partitioning defines a binary tree structure, called the *separator tree*. Each node of this tree corresponds to a separator. The root of the tree corresponds to the separator from the first level partitioning. Then each branch corresponds to one of the disconnected subgraphs. The partitioning can then be applied recursively on each subgraph. Figure 3.1 illustrates a matrix partitioned in two levels with four disconnected subgraphs. Its corresponding separator tree is depicted in Figure 3.2, where we number the levels of the binary tree starting with 0 at the bottom. This numbering does not reflect the top-down graph partitioning approch. Instead, it reflects the bottom-up traversal of the tree during our symbolic factorization algorithm. In Figure 3.1, the patterned blocks illustrate portions of the matrix that can contain nonzero elements. The meaning of the different patterns used will be described later in this section.

The separator tree plays an important role in sparse factorizations because it shows the parallelism from the partition of the matrix and identifies dependencies among the computations. If $Str(L(:, j))$ is needed in computing $Str(L(:, i))$, then the node owning index j belongs to the subtree rooted at the node owning index i.

In order to perform the symbolic factorization in parallel, the nodes of the separator tree are assigned to processors using the standard subtree-to-subcube algorithm [9, 12]. This algorithm considers that the separator tree is well balanced. That is, each partitioning leads to two disconnected graphs of roughly the same size. In practice, unbalanced trees can be obtained, and a better mapping of nodes to processors can be used. We plan to address this problem in our future work.

The subtree-to-subcube algorithm maps nodes to processors during a top-down traversal of the separator tree. It starts by assigning all the p processors to the root. Then it assigns $p/2$ processors to each of the two subtrees of the root. This process is continued until only one processor is assigned to a subtree. The advantage of this

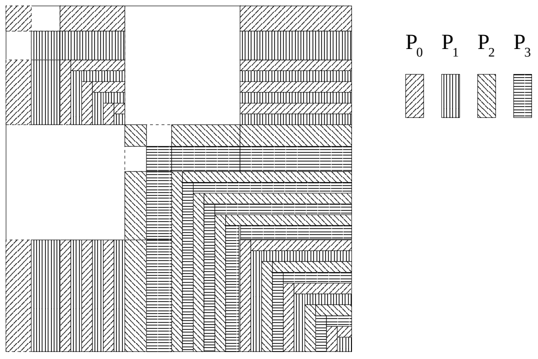


FIG. 3.1. *Example matrix and data distribution for parallel symbolic factorization algorithm executed on 4 processors.*

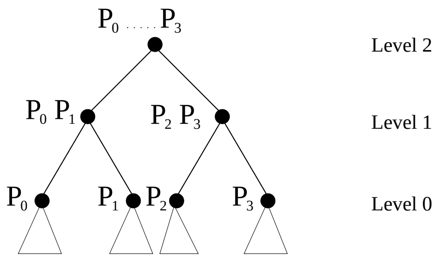


FIG. 3.2. *Separator tree corresponding to the example matrix in Figure 3.1.*

algorithm is that the communication in a subtree is restricted to only the processors assigned to this subtree, thus hopefully ensuring a low communication cost. Each node of the tree is associated with a range of indices. All the processors assigned to a node will cooperate to compute the structure of the columns of L and of the rows of U whose indices are associated with the node. The indices are distributed among those processors using a 1D block cyclic distribution along the diagonal, in a block arrow shape.

Consider again Figures 3.1 and 3.2. Suppose that we have 4 processors. The subtree-to-subcube mapping is illustrated in Figure 3.2 in which we label the processors assigned to each node of the separator tree. The 1D block cyclic data distribution is illustrated in Figure 3.1, where we use different patterns for different processors.

Algorithm 2 describes the main steps of the parallel symbolic factorization algorithm. Each processor performs a bottom-up traversal of the separator tree to participate in factorizing the nodes it owns. It starts with an independent symbolic factorization phase for the indices associated with its leaf node, using a left-looking algorithm similar to Algorithm 1. Then at each higher level of the tree, each processor is involved in computing one node D that it owns at this level. The computation of each D consists of two phases. In the first phase, named *inter-level* factorization, the columns of L and rows of U corresponding to the indices associated with D are updated in a right-looking manner by the columns of L and rows of U belonging to the predecessors of D in the separator tree. In the second phase, named *intra-level* factorization, all the processors participate in completing the symbolic factorization of node D, using a combination of right-looking and left-looking partial symbolic factorization method.

We now use Figure 3.3 to illustrate the execution of Algorithm 2 step by step. Consider level 1 of the tree. The computation of the right node D (doubly circled in the

Algorithm 2 Parallel Symbolic Factorization

```
let $myPE$ be my processor number, $P$ be the total number of processors
compute symbolic factorization of leaf node in a left-looking fashion
for each level from $l := 1$ to $logP$ in the separator tree do
    let $\mu$ and $\nu$ be the first and last indices of node $D$ that $myPE$ owns
    for $i := \mu$ to $\nu$ such that $myPE$ owns index $i$ do
        init $Str(L(:, i)) := Str(A(i : n, i))$
        init $Str(U(i, :)) := Str(A(i, i : n))$
    end for
    Inter-level Symbolic Factorization ($myPE$, $l$) (Algorithm 3)
    Intra-level Symbolic Factorization ($myPE$, $l$) (Algorithm 4)
end for
```

figure) is assigned to processors P_2, P_3. We use the following notation to distinguish the indices and their corresponding columns of L and rows of U associated with each node. The indices corresponding to the right node of level 1 are marked in patterned white, and the indices to be used to update this node and that correspond to the leaf nodes assigned to P_2 and P_3 are in dark gray. All the other indices, not involved in the computation considered here, are displayed in light gray.

At this stage, processor P_2 has already computed the structures of L and U of the indices associated with the left leaf node, and processor P_3 has done so with the right leaf node. During inter-level factorization, processors P_2 and P_3 will determine which columns of L and which rows of U from the leaf nodes are needed for updating node D. This data is first exchanged between P_2 and P_3, and node D is then updated in a right-looking fashion. During intra-level factorization, factorization of node D is completed using a block cyclic algorithm, where factorization of each block is performed in a left-looking fashion, and then the block is used to update the subsequent blocks in a right-looking fashion.

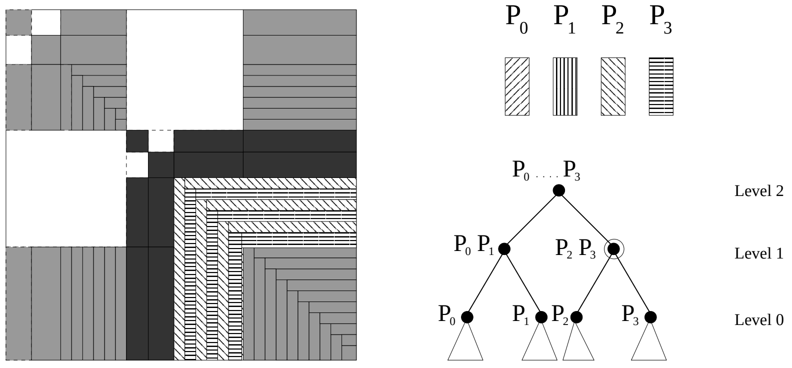


Fig. 3.3. *Example illustrating the parallel symbolic factorization (Algorithm 2) on the example matrix in Figure 3.1 for the doubly circled node assigned to processors P_2, P_3. Its associated indices are marked in patterned white, and the indices to be used to update this node and that correspond to the leaf nodes assigned to P_2 and P_3 are in dark gray. All the other indices are displayed in light gray.*

Algorithm 3 details the inter-level factorization. Consider the computation of node D at level l, which is assigned to processors $P_r \dots P_q$. This phase performs all the updates that the indices of node D must receive from the indices below node D. Only the indices associated with the nodes in the subtree rooted at D can contribute to these updates. Each processor determines the source of the updating data (columns of L and rows of U) that it owns and that belongs to nodes of the subtree rooted at D. Then the data is copied in a send buffer. Because of sparsity, it is very likely that this processor needs to send data to only a small subset of the processors that own node D. Several implementations of this collective communication are possible. In our current implementation, we use a simple scheme that assumes that the receive buffer of each processor is large enough to store all the incoming messages. We have tested this scheme on the matrices in our test set and we have observed that the size of the receive buffer represents a small fraction of the total memory needs. Thus, we expect that this scheme will not affect the memory scalability of our algorithm.

Each processor can determine the destination processors to send the data. After a global exchange among the processors owning node D, each processor also knows from whom the data needs to be received. Afterwards, each processor sends its data to destination processors in a non-blocking manner, and then posts all the necessary receives. Since we need to release the send/receive buffers, the received data have to be consumed right away. This is achieved by using a right-looking approach, as described in Algorithm 5, to update the indices associated with node D. In our example, the indices that contribute to this update are shown in dark gray. Processors P_1 and P_2 exchange necessary parts of columns of L and rows of U corresponding to these indices.

Algorithm 3 Inter-level Symbolic Factorization

```
Input: $myPE$ : my processor number, $l$ : level in the separator tree
let $D$ be the node at level $l$ owned by processor $myPE$
let $P_r \dots P_q$ be the processors assigned to node $D$
let $\mu$ and $\nu$ be the first and last indices of node $D$
copy data (columns of $L$, rows of $U$) to be sent to $P_r \dots P_q$ in $sendBuf$
determine the processors that $myPE$ needs to exchange data with
for each processor $P$ in $P_r \dots P_q$ to which data needs to be sent do
    non-blocking Send $sendBuf$ to $P$
end for
for each processor $P$ in $P_r \dots P_q$ from which data needs to be received do
    Recv data (columns of $L$, rows of $U$) in $recvBuf$
    Right-Looking update of $U(\mu:\nu,:)$ and $L(:,\mu:\nu)$ using $recvBuf$
end for
Wait for all non-blocking Sends to complete
```

Algorithm 4 details the intra-level factorization. The goal is to complete the factorization of node D by performing all the necessary updates between the indices associated with node D. The algorithm is a block variant of right-looking factorization. It provides a general framework to perform asynchronously a right-looking factorization of a sparse matrix. The algorithm proceeds as follows. Each processor $myPE$ iterates over the blocks of node D that it owns. Let $P_r \dots P_q$ be the $q-r+1$ processors that are assigned to node D in 1D block cyclic fashion. Assume processor $myPE$ arrives at the computation of block K.

First, processor $myPE$ has to receive a number of messages from the processors that computed blocks $K-q+r, \dots, K-1$ to update its blocks in a right-looking fashion. To determine the number of messages that have to be received, we use an auxiliary array $noMsgsToRcv$, that is initialized to 0. This array keeps global information on the number of messages that every processor needs to receive. Locally, each processor stores the number of messages that were already received in the variable $myMsgs$.

Processor $myPE$ will receive the array $noMsgsToRcv$ from the owner of block $K-1$ and $noMsgsToRcv[myPE]$ gives the number of messages $myPE$ has to receive at this step of computation. This is implemented as follows. Processor $myPE$ posts a non-blocking receive from the owner of block $K-1$, anticipating the array *noMsgsToRcv*. The condition variable *rcvdArray* is set when this non-blocking receive is completed. While waiting for this message, *myPE* iterates over a loop, in which a non-blocking receive for data is posted at each iteration. When a message is received, if it corresponds to data, the later blocks of node D owned by *myPE* are updated in a right-looking fashion. The variable *myMsgs* is incremented by one. If the message received corresponds to the array *noMsgsToRcv*, then *myPE* knows how many messages must be received. If all the messages were received, the last posted data receive is canceled. If not, *myPE* continues with receiving all the necessary data messages.

Second, processor *myPE* has to complete the factorization of block K itself. This is performed using a left-looking approach, similar to Algorithm 1. The difference is that the algorithm iterates from the first index of block K (μ_K) to the last index of block K (ν_K), instead of iterating from 1 to n as in Algorithm 1, where n is the order of the matrix A.

Third, processor *myPE* needs to identify all the processors to which block K should be sent. For each such processor P, the value of *noMsgsToRcv*[P] is incremented. And if K is not the last block of node D, *myPE* sends the array *noMsgsToRcv* to the owner of block $K+1$. Since the matrix is sparse, the data needs to be sent to only a small subset of processors. Empirically, we have observed that this number is usually 2 or 3. In particular, in symmetric cases, the data needs to be sent to only the processor owning the next block.

The block cyclic distribution of columns of L and rows of U helps maintain load balance. Note that we need the rows of U to determine what columns of L need to be sent to which processors. Similarly, we need the columns of L to determine to which processors we need to send the rows of U. It is easy to determine this communication pattern because of the arrow shape layout.

4. Exploiting density in the separators.

In this section, we describe one optimization technique that can easily improve the runtime of symbolic factorization. It is well-known that towards the end of the elimination, the factored matrices become progressively full. In addition, we also observed that the submatrices associated with the separators become progressively full as well. A symbolic factorization algorithm that computes $Str(L)$ by columns and $Str(U)$ by rows can easily identify the density in the factors. Moreover, since our parallel algorithm uses a 1D distribution of the data along the diagonal, this detection can be implemented without any communication.

4.1. Separator with dense diagonal block.

Consider Algorithm 2 when it arrives at the intra-level computation of node D consisting of indices $\mu, \ldots \nu$. Let B be the diagonal block submatrix associated with the separator, that is,

$$B = L(\mu : \nu, \mu : \nu) + U(\mu : \nu, \mu : \nu)$$

Once the structures of column μ of L and row μ of U have been computed, the sizes $r = nnz(L(\mu : \nu, \mu))$ and $q = nnz(U(\mu, \mu : \nu))$ are known. The product rq gives a lower bound on the fill in B. If the product rq is $(\nu - \mu)^2$, B would be completely full. Then, there is no need to compute the subsequent rows of L and columns of U corresponding to the indices $\mu + 1$ to ν. Note that even if B is not entirely full, we can use the ratio $rq/(\nu - \mu)^2$ as a parameter to control *relaxation*, that is, padding zeros in B to increase the size of supernodes.

The following theorems show that, given a full B on the diagonal, the off-diagonal block structures for L and U can also be easily determined. In fact, the L part is formed by the dense subrows that end at column ν and the U part is formed by the dense subcolumns that end at row ν, as depicted in Figure 4.1. These structures are often referred to as *skyline* structures.

THEOREM 4.1. *Consider $A = LU$ and assume $u_{i,i+1}$ is nonzero, for $i = \mu$, $\ldots \nu - 1$. If a_{jk} is nonzero for some j with $j \geq \mu$ and $j > k$, then l_{jt} is nonzero for all t such that $j \leq t \leq \nu$.*

Algorithm 4 Intra-level Symbolic Factorization

```
**Input:** $myPE$ : my processor number, $l$ : level in the separator tree
let $D$ be node of level $l$ owned by processor $myPE$
let $P_r \dots P_q$ be the processors assigned to node $D$
let $\mu$ and $\nu$ be the first and last indices of node $D$
let $N_l$ be number of blocks node $D$ is divided into
initialize array $noMsgsToRcv[]$ to 0
initialize variable $myMsgs$ to 0
for each block $K$ from 1 to $N_l$ that $myPE$ owns do
  let $\mu_K$ and $\nu_K$ be the first and last indices of block $K$
  $rcvdArray = FALSE$
  post non-blocking Recv for array $noMsgsToRcv$
  post non-blocking Recv for data in $recvBuf$
  while $myMsgs \neq noMsgsToRcv[myPE]$ $and$ $rcvdArray \neq TRUE$ do
    Wait for a receive to complete
    if message received corresponds to data then
      increment variable $myMsgs$ by one
      Right-Looking update of $L(:, \mu_K : \nu)$ and $U(\mu_K : \nu, :)$ using $recvBuf$
    else
      /* message received corresponds to $noMsgsToRcv[]$*/
      Cancel non blocking receive of data if $myMsgs = noMsgsToRcv[myPE]$
      $rcvdArray = TRUE$
    end if
  end while
  Left-Looking symbolic factorization to compute $L(:, \mu_K : \nu_K)$ and $U(\mu_K : \nu_K, :)$
  if $K$ is not the last block then
    for each processor $P$ in $P_r \dots P_q$ to which $myPE$ needs to send data do
      Increment $noMsgsToRcv[P]$ by one
      Send $L(:, \mu_K : \nu_K)$ and $U(\mu_K : \nu_K, :)$ to $P$
    end for
    Send $noMsgsToRcv[]$ to owner of block $K + 1$
  end if
end for
```

Algorithm 5 Right-Looking update of $L(:, \mu : \nu)$, $U(\mu : \nu, :)$

```
**Input:** $recvBuf$, $L(:, \mu : \nu)$, $U(\mu : \nu, :)$, $myPE$
**Output:** Updated $L(:, \mu : \nu)$, $U(\mu : \nu, :)$
initialize $prL, prU$ to empty
for each index $j$ of $recvBuf$ do
  let $k$ be such that $l_{jk}$ and $u_{kj}$ are nonzeros
  if there is an $i$ st $l_{ij} \neq 0$, $i \leq k$, $\mu \leq i \leq \nu$, $myPE$ owns index $i$ then
    add pruned column $j$ of $L$ to $prL$
  end if
  if there is an $i$ st $u_{ji} \neq 0$, $i \leq k$, $\mu \leq i \leq \nu$, $myPE$ owns index $i$ then
    add pruned row $j$ of $U$ to $prU$
  end if
end for
for $i := \mu$ to $\nu$ such that $myPE$ owns index $i$ do
  Add to $Str(L(:, i))$ the union $\cup \{Str(L(i : n, j)) \mid (j, i)$ is an edge of $G(prU)\}$
  Add to $Str(U(i, :))$ the union $\cup \{Str(U(j, i : n)) \mid (i, j)$ is an edge of $G(prL)\}$
end for
```

Proof. First, l_{jk} is nonzero because a_{jk} is nonzero (here as elsewhere, we do not consider cancellation). Since $u_{i,i+1}$ is nonzero for all i, $\mu \leq i \leq \nu - 1$, then by Theorem 2.1, $Str(L(:,i)) \subseteq Str(L(:,i+1))$, for all i, $\mu \leq i \leq \nu - 1$, Hence l_{jt} is nonzero for all t from j to ν. □

THEOREM 4.2. *Consider $A = LU$ and assume $l_{i,i+1}$ is nonzero, for $i = \mu, \ldots \nu-1$. If a_{jk} is nonzero for some j with $j \geq \mu$ and $j < k$, then u_{jt} is nonzero for all t such that $j \leq t \leq \nu$.*

Proof. Similar to the proof of Theorem 4.1. □

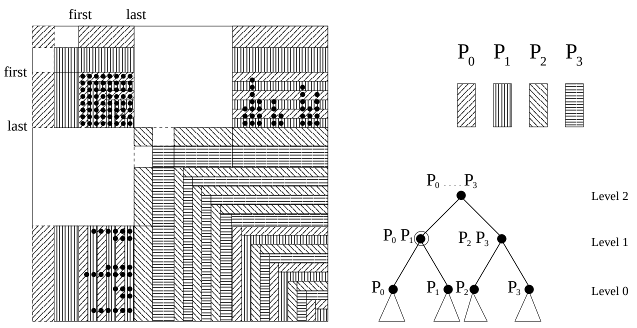


FIG. 4.1. *Example of the skyline structures of L and U when the separator has a dense diagonal block. The left node of level 1 (doubly circled) has the dense diagonal block from μ to ν.*

We explain now how the nonzero structure of a range of indices of L corresponding to node D is computed, using the fact that the diagonal B is full. The computation of the nonzero structure of U is similar. The key is to compute for each column i ($\mu \leq i \leq \nu$) a set $FNZ(i)$ that contains the row indices of the rows of A that have the first nonzero in column i. More formally, FNZ is defined as:

$$FNZ(i) = \{j \mid a_{ji} \neq 0 \ and \ a_{jt} = 0 \ for \ all \ t \in (\mu : i-1)\}$$

The computation of FNZ is achieved as follows. Let processors $P_r \ldots P_q$ be assigned to node D. First, every processor determines locally the column index i of the first nonzero in row j of A owned by this processor. Then a parallel reduction operation is applied among all the processors $P_r \ldots P_q$ to determine the first nonzero column index for each entire row j. This information is used to compute the structure FNZ. Then the structure of column i of L is simply the union of $FNZ(k)$ for $\mu \leq k \leq i$.

Note that the sets $FNZ(\mu) \ldots FNZ(\nu)$ are disjoint. If data is well balanced among processors, the total computational cost is dominated by $O(nnz(L)/P)$ plus the cost of a call to *MPI_Allreduce*. In intra-level symbolic factorization, Algorithm 4 tries to identify dense diagonals after computing the structure of each index. When a dense column and row is found, this computation is performed for the rest of the separator.

4.2. Separators with dense structures belonging to a path of the separator tree. One further optimization is to identify the dense separators on a path of the separator tree. Consider again Algorithm 2 when it arrives at the intra-level computation of node D consisting of variables $\mu, \ldots \nu$. Let B be the submatrix containing the variables associated with all the nodes on the path from node D to the root. Define $r = nnz(L(:,\mu))$ and $q = nnz(U(\mu,:))$. The product rq represents a lower bound on the number of fill in the submatrix B. If r and q are the same and equal the order of B, then we know B is full. That is, all the submatrices associated with the nodes on the path from D to the root are dense. This is illustrated in Figure 4.2. The processor that detects this μ column sends messages to all the other processors involved in the nodes on the path from D to the root. The rest of the algorithm is simply to generate the indices of these dense submatrices.

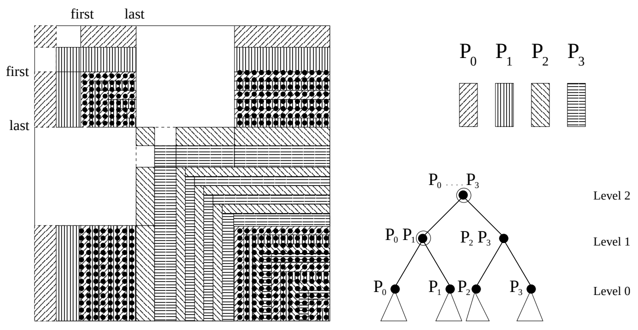


FIG. 4.2. *Example of the dense structures of L and U when there are dense separators on a path to the root. The nodes of the tree corresponding to the dense separators are doubly circled.*

5. Experimental results. In this section, we present experimental results for our parallel symbolic factorization algorithm applied to real world matrices. We tested the algorithm on an IBM SP RS/6000 distributed memory machine at NERSC (National Energy Research Scientific Computing Center). The system contains 2944 compute processors distributed among 184 compute nodes. Each processor is clocked at 375 Mhz and has a peak performance of 1.5 GFlops. Each node of 16 processors has 16 to 64 Gbytes of shared memory.

We use several medium to large size matrices that are representative of a variety of application domains. The matrices and their characteristics are summarized in Table 5.1, which includes the matrix order, the number of nonzeros, the structural symmetry, and the application domain. We show the average number of nonzeros per column of original matrix A, as a measure of sparsity. The matrices are available from University of Florida Sparse Matrix collection [3] (BBMAT, STOMACH, PRE2, TORSO1, TWOTONE, MARK3JAC140SC, G7JAC200SC, LHR71C) and the accelerator design project at the Stanford Linear Accelerator Center [19] (DDS.QUAD.K-SM, DDS15.K-SM). To analyze the memory scalability of our algorithm, we also use matrices obtained from the 11-point discretization of the Laplacian operator on three dimensional grids (REG3D).

The goal of the experiments is two-fold. Firstly, we want to study the performance of the new parallel symbolic factorization algorithm itself. Secondly, we want to examine the performance improvement of the entire solution process of SuperLU_DIST gained by parallelizing the symbolic factorization step. Our performance criteria include both runtime and memory usage. As stated in the introduction, our goal is to prevent the symbolic factorization step from being a bottleneck (especially in terms of memory) in the entire solution process. We show here that this goal is indeed achieved.

For the second study, we present detailed performance results for the first six matrices in our test set. We compare the performance of two configurations of SuperLU_DIST, one is referred to as SLU_SFseq in which the serial symbolic factorization algorithm is used as before, and the other is referred to as SLU_SFpar in which the new parallel symbolic factorization algorithm is used. We keep the other phases the same as much as possible, including the scaling and the pre-pivoting using the sequential routine MC64 [6] from the HSL collection [15], the fill-reducing ordering using PARMETIS applied to the structure of $|A|+|A|^T$, the numerical LU factorization, and the triangular solutions.

Because of the different data structures used in symbolic factorization, the redistribution of the matrix from the storage format used in symbolic factorization

Matrix	Order	$nnz(A)$	Structural Sym.	$nnz(A)/n$	Application Domain
REG3D	729000	7905960	1.0	10.8	3D Laplacian cubic grid size 90
DDS.QUAD.K-SM	380698	15844364	1.0	41.6	accelerator design
DDS15.K-SM	834575	13100653	1.0	15.7	accelerator design
STOMACH	213360	3021648	.85	14.2	bioengineering
BBMAT	38744	1771722	.53	45.7	fluid flow
PRE2	659033	5834044	.33	8.8	circuit simulation
TORSO1	116158	8516500	.42	73.3	bioengineering
TWOTONE	120750	1206265	.24	10.0	circuit simulation
MARK3JAC140SC	64089	376395	.07	5.9	economic modeling
G7JAC200SC	59310	717620	.03	12.1	economic modeling
LHR71C	70304	1528092	.00	21.7	chemical engineering

TABLE 5.1
Benchmark matrices.

to the storage format used in numeric factorization is different. We will report separately the timings for symbolic factorization ("SFseq" for SLU_SFseq and "SFpar" for SLU_SFpar) and the timings for re-distribution ("RDseq" for SLU_SFseq and "RDpar" for SLU_SFpar). Another difference between the two configurations lies in the supernode definitions. SLU_SFseq groups in a supernode the columns of L with the same structure, whereas SLU_SFpar groups in a supernode the indices corresponding to the columns of L with the same structure and to the rows of U with the same structure. Thus, the definition in SLU_SFpar is more restrictive, and tends to give more supernodes of smaller sizes. This difference may lead to different runtimes of numerical factorization and triangular solutions, but empirically we observed that the differences are very small, and thus we report the average time obtained by these steps in the two solvers.

The matrix A is distributed among processors in a block row format, which is consistent with the input format required by PARMETIS. In SLU_SFseq, the distributed matrix is gathered such that every processor has a copy of it. Then MC64 is called on one processor to determine a row permutation, which in turn is broadcast to every other processor. The structure of $|A| + |A|^T$ is computed in parallel using the distributed input matrix, and then PARMETIS is called. The symbolic factorization step is performed redundantly on each processor. The distributed structure of L and U is then set up among the processors with a 2D block-cyclic distribution over a 2D processor grid, which will be used in numerical factorization.

In SLU_SFpar, the matrix is gathered on one processor only to perform the first step (calling MC64) sequentially. All the other steps involve the distributed input matrix. (Although this means that the solver as tested is not yet memory scalable, the work in progress that parallelizes the first step [21] should solve this problem.) The structure of $|A| + |A|^T$ is computed in parallel, and then PARMETIS is called. The data is then distributed in the arrow shaped format among the processors, and parallel symbolic factorization is performed. Afterwards, the data is re-distributed from this arrow shaped format to the 2D block-cyclic distribution over a 2D processor grid, which will be used in numerical factorization.

Notice that SuperLU_DIST can be used on any number of processors. But the partitioning performed by PARMETIS is only applicable to a number of processors that is a power of two. Therefore, in our code, we have another data re-distribution phase when the processors to be used is not a power of two. That is, we choose the largest power-of-two that is at most the total number of processors, and distribute the data to this subset of processors for parallel ordering and symbolic factorization. After that, we re-populate the matrix to the full set of processors.

To perform the same numeric factorization in both configurations, we turned off supernode relaxation in SuperLU_DIST. This is because the relaxation is determined using the elimination tree that we do not have in SLU_SFpar when the input matrix is distributed. Ideally, each processor can locally compute a partial elimination tree associated with each leaf node of the separator tree, and perform relaxation locally. But we have not implemented this scheme. In [4] it was shown that supernode relaxation leads to improvements of 5% to 15% in the running time of sequential SuperLU. We expect that similar improvements can be obtained in the running time of the distributed numeric factorization.

The experimental results for the first six matrices in our test set are presented in the runtime plots of Figure 5.1, and in the memory usage plots of Figure 5.2.

For the runtime performance, we display in each plot of Figure 5.1 the total time of the solver when using sequential or parallel symbolic factorization (SLU_SFseq and SLU_SFpar), the runtime of numerical factorization (Factor), parallel reordering (ParMetis), sequential and parallel symbolic factorization (SFseq and SFpar) and re-distribution (RDseq and RDpar).

For memory usage evaluation, we report in the plots in Figure 5.2 the maximum memory requirement per processor throughout the entire solution process (except MC64; see below). The peak is likely to occur in the re-distribution routine from the symbolic factorization data structure to the numeric factorization data structure, or in the numeric factorization routine. The re-distribution routine needs as input the nonzero structures of L and U. Memory is allocated to store these structures along with the space for the numerical values of L and U. Several auxiliary arrays are used in this routine when placing the numerical values of A into their corresponding positions of the factors L and U. Those arrays and the input nonzero structures of L and U are deallocated on return. The memory need in the numerical factorization routine is given by the 2D storage of structural and numerical information of the factors plus several supplementary arrays needed for communication.

We ignore for the moment the memory needs of MC64, which is dominated by the size of the matrix A, because in the future this step will be replaced by a parallel algorithm [21]. We report only the memory needs of the symbolic factorization step and of the numeric factorization step. The memory needs of the other steps of the algorithm are dominated by the size of the distributed matrix A, or the size of the distributed factor L, or the size of the distributed factor U, which are not the bottlenecks in the solver. Therefore, we do not report memory statistics for any of the other steps.

Our parallel symbolic factorization algorithm uses two kinds of data, replicated data and distributed data. The replicated data are two arrays of size n, where n is the order of the matrix. The first replicated array is used as a marker during the computation of $Str(L)$ and $Str(U)$. The second replicated array is a mapping array that shows for each variable its owner and its local index. That is, every variable has a global index which is known by all the processors and a local index which is known and used only by its owner. To set up the communication in the algorithms, each processor needs to determine the owners of certain variables. This local-global index translation is achieved by this mapping array. The distributed data are those whose size per processor is expected to decrease when increasing the number of processors. These are the arrays used to store the $Str(L)$ and $Str(U)$, the arrays to store the pruned structures of L and U, and the buffers needed for communication.

In summary, the data reported are as follows: maximum memory used by the solver using sequential and parallel symbolic factorization (SLU_SFseq MAX and SLU_SFpar MAX); memory used during sequential and parallel symbolic factorization (SFseq and SFpar MAX); maximum memory used during numerical factorization (Factor MAX).

Notice that the ordering results of ParMetis differ using different number of processors. (They are usually worse with increasing number of processors.) Consequently, the runtime and the memory usage corresponding to the sequential symbolic factorization changes with different number of processors. With the exception of reg3D, for the runs of all the other matrices we used 16 processors per node.

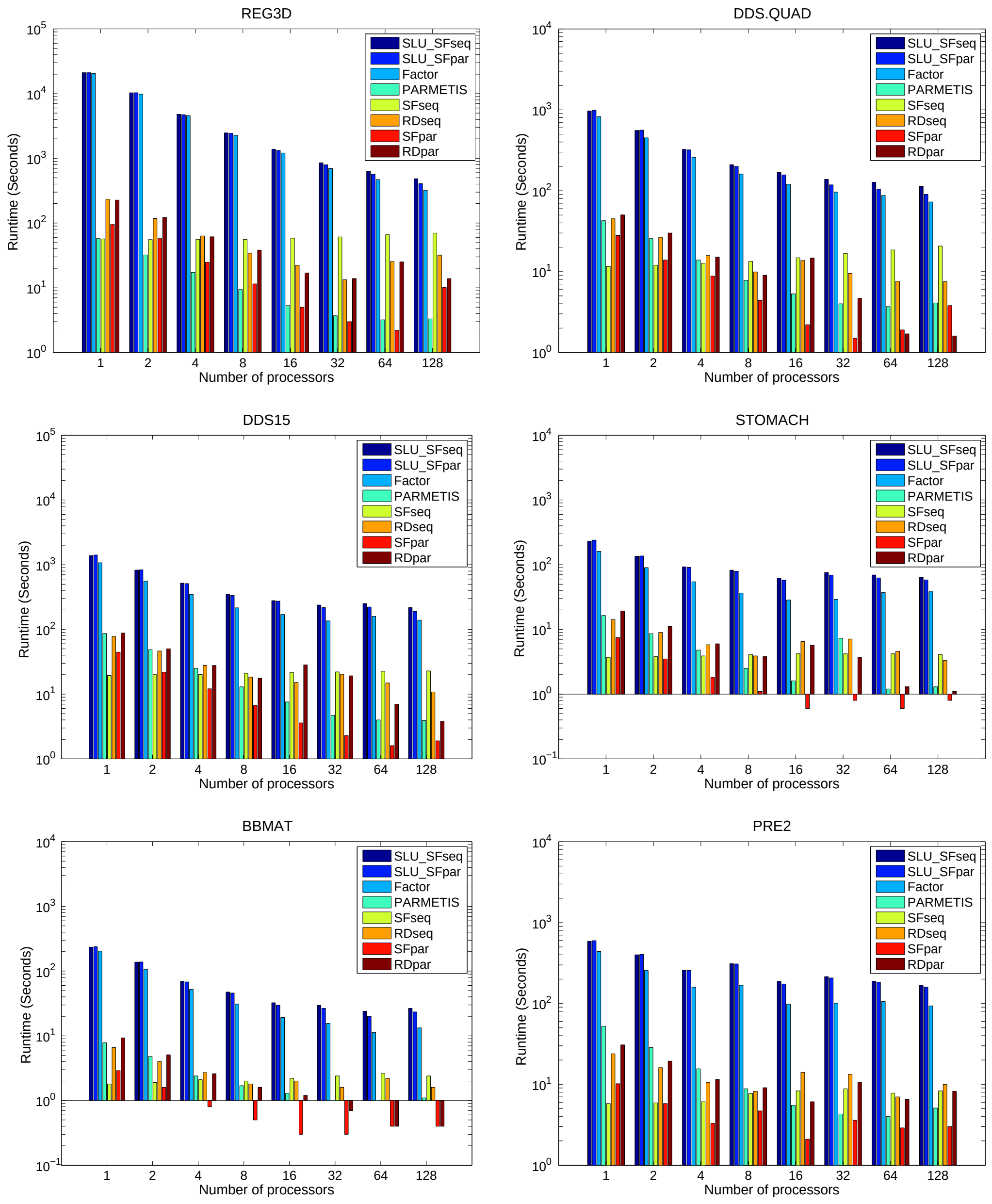


Fig. 5.1. *Runtimes (in logarithmic scale) for parallel reordering (*ParMetis*), sequential and parallel symbolic factorization and re-distribution (SFseq and SFpar, RDseq and RDpar), numeric factorization (Factor) and the entire solvers (SLU_SFseq and SLU_SFpar) for matrices in our test set.*

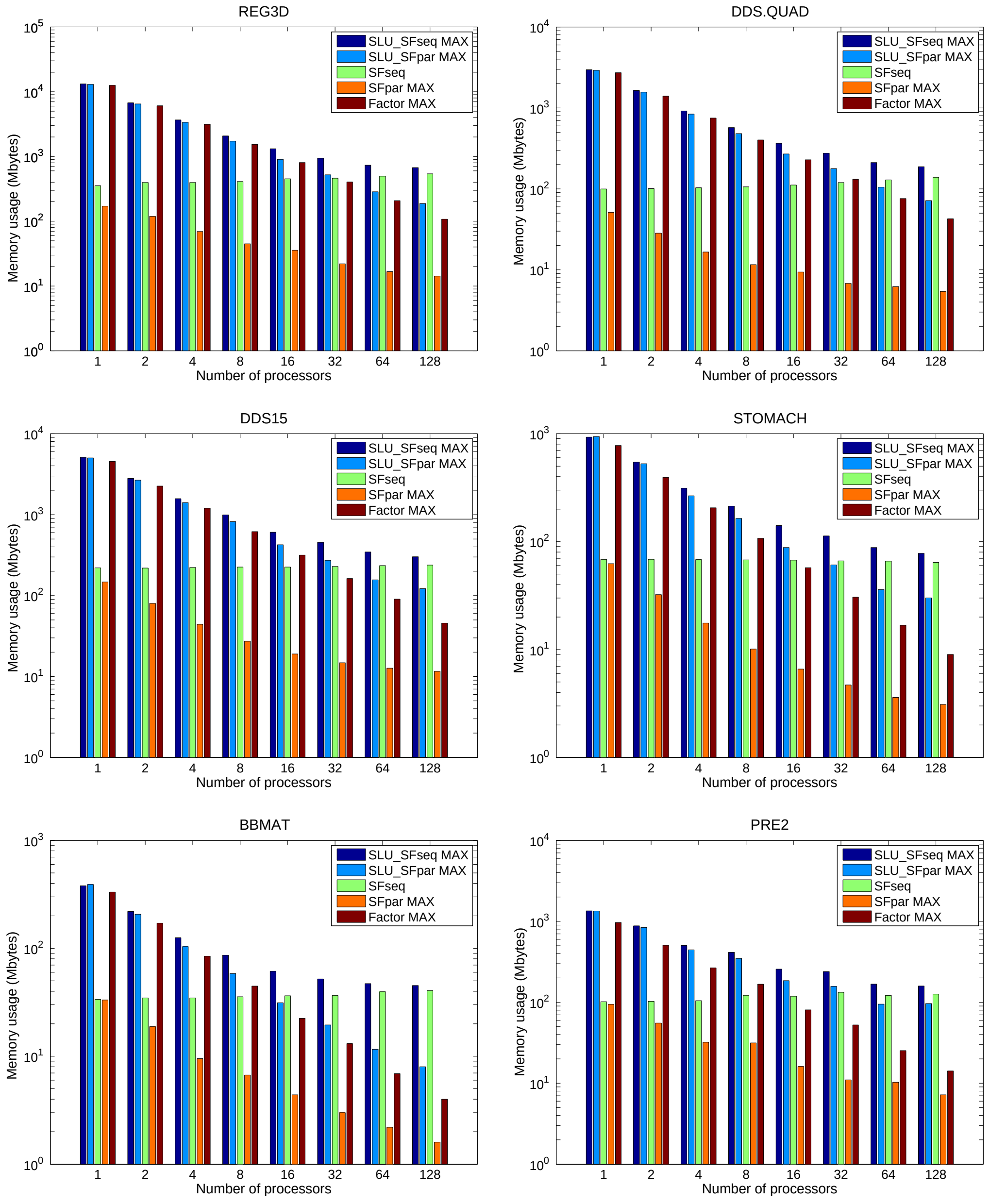


FIG. 5.2. *Memory usage (in logarithmic scale) for sequential and parallel symbolic factorization (SFseq and SFpar MAX), numeric factorization (Factor) and the entire solvers (SLU_SFseq and SLU_SFpar) for matrices in our test set.*

Since the algorithm behaviors are very different for the matrices that are structurally symmetric than for those that are very unsymmetric, we first report the results separately for these two classes of problems.

5.1. Performance with structurally-symmetric matrices. In this section we discuss the results with the structurally symmetric matrices. In fact, our symmetric test matrices are also positive definite, so there is no need to call MC64 (that can destroy the symmetry of the input matrix). The code was compiled using 64-bit addressing mode so that each process can use more than 2 GBytes of memory. In the parallel symbolic factorization algorithm, the block size used in the 1D block cyclic layout is 50.

In our parallel algorithm, the symmetrically pruned graph determines the communication pattern. For symmetric matrices, we obtain the benefit of symmetric reduction to the maximum extent. That is, the pruned graph contains only the first off-diagonal nonzero element in the first off-diagonal block, and is in fact the same as the supernodal elimination tree. Then, every variable is used in the computation of at most one other variable. The communication volume is therefore bounded by the number of nonzeros in $Str(L)$ and $Str(U)$. Thus the algorithm needs a low communication volume in both inter-level and intra-level factorizations. On the other hand, this also implies that the algorithm has a limited parallelism in the intra-level factorization, where a right-looking block cyclic approach is used.

We first examine the results for the model problem REG3D. We use a cubic grid of size $NX = NY = NZ = 90$ with an 11-point discretization. We see in the plot REG3D of Figure 5.1 that the runtime of SFpar continues to decrease with increasing number of processors up to 64, achieving very good speedup—30.0 on 64 processors. The re-distribution routine implemented in SLU_SFpar is usually a little faster than that in SLU_SFseq, with differences larger on more processors. Overall, the entire SLU_SFpar solver is about 16% faster than SLU_SFseq on 128 processors.

The plot REG3D in Figure 5.2 reports the memory usage. On 32 processors, the memory need of SFseq becomes larger than that of the per-processor need in numeric factorization. On 128 processors, the memory need of SFseq is 5 times more than that of the per-processor need in numeric factorization. Thus, sequential symbolic factorization is indeed the bottleneck for memory scalability of SuperLU_DIST. Our parallel symbolic factorization algorithm achieves a good memory scalability. For example, on 128 processors, the memory need of SFpar is 38-fold smaller than that of SFseq. The reduction in the symbolic factorization phase also leads to significant reduction in the overall memory usage. On 128 processors, the maximum per-processor memory needs of SLU_SFpar is only about 28% of that required by SLU_SFseq. Therefore, SLU_SFpar can solve a much larger problem.

We notice that on eight and sixteen processors, the entire memory needs of SLU_SFseq are larger than 16 GBytes, which is the available memory on most of the compute nodes of IBM SP RS/6000 at NERSC. In order to fit the problem in memory, we had to use more compute nodes than necessary. For example, SLU_SFseq on sixteen processors was run on two compute nodes, with only eight processors per node used. In addition, the runtime can slow down severely when the memory used per node is too large (causing disk swapping). In our experiments, this happended when SLU_SFseq was run on 32 processors and when SLU_SFpar was run on 16 processors. Hence, to obtain the best runtimes, we have used for these two runs twice more nodes than necessary.

We now examine the results for the irregular problems from the accelerator design. The plots DDS.QUAD.K-SM and DDS15.K-SM in Figure 5.1 show the runtimes of SLU_SFpar and SLU_SFseq for the two matrices. Again, SFpar exhibits good performance, with a maximum speedup of 14.1 on 64 processors for the matrix DDS15.K-SM. The time of SFpar is always smaller than the time spent in ParMetis. On small number of processors, the times spent in re-distribution are comparable for RDseq and

RDpar. But with increasing number of processors, the time spent in RDpar can be up to four times smaller than the time spent in RDseq. Overall, the entire SLU_SFpar solver is about 20% faster than SLU_SFseq on 128 processors for DDS.QUAD.K-SM.

The plots DDS.QUAD.K-SM and DDS15.K-SM in Figure 5.1 compare the memory usage of SLU_SFpar and SLU_SFseq and various steps in the solvers. We see again that with increasing number of processors, SFseq becomes the memory bottleneck of the SLU_SFseq solver. The parallel algorithm reduces the memory usage by more than 20-fold in symbolic factorization. Overall, the maximum per-processor memory need by the entire SLU_SFpar solver is only about 38-40% of that by SLU_SFseq.

5.2. Performance of structurally-unsymmetric matrices. The plots BBMAT, STOMACH and PRE2 in Figures 5.1 and 5.2 show the results with the unsymmetric matrices. Since these matrices are smaller than the symmetric ones, each processor does not need more than 2 GBytes of memory, and the code could succeed when compiled with 32-bit addressing mode, In the parallel symbolic factorization algorithm, the block size used in the 1D block cyclic layout is 50.

When comparing the runtimes of the two solvers SLU_SFpar and SLU_SFseq, we see that the time spent in numeric factorization dominates the time spent in any other step, which is true even on 128 processors. The runtime of SFseq is modest for all the matrices. This is probably because the sizes of those unsymmetric matrices are still quite moderate. The parallel algorithm further reduces this time. In particular, the time of SFpar is always smaller than the time spent in ParMetis. The times spent for re-distribution (RDseq and RDpar) are comparable on smaller number of processors (less than 16). But on larger number of processors, RDpar can be much faster than RDseq (up to 5-fold for BBMAT).

When comparing the memory usage of the sequential symbolic factorization of SLU_SFseq and the parallel symbolic factorization of SLU_SFpar, we observe that the memory usage is significantly reduced for all the matrices. It continues to decrease when increasing the number of processors up to 128. The parallel algorithm reduces the memory usage by up to 25-fold in symbolic factorization. Overall, the maximum per-processor memory usage of the entire SLU_SFpar ranges between 18% (BBMAT) and 60% (PRE2) of that of the SLU_SFseq solver.

5.3. Summary. Figure 5.3 summarizes the runtime and the memory usage of our parallel symbolic factorization algorithm for the first six matrices in our test set. In addition, the performance of our algorithm applied to matrices with higher unsymmetry is shown in Figure 5.4.

For runtime scaling, we show the speedup obtained by our parallel symbolic factorization algorithm with respect to the sequential symbolic factorization algorithm. We see that for symmetric matrices, the runtime continues to decrease with increasing number of processors up to 32 for DDS.QUAD.K-SM, and up to 64 for REG3D and DDS15.K-SM. A maximum speedup of 30.0 on 64 processors is obtained for REG3D. We found that the technique for identifying dense separators plays an important role for symmetric matrices, it significantly reduces the number of operations in symbolic factorization. The matrix REG3D represents the best case in terms of exploiting the dense separators, but we observe a big performance drop on 128 processors. After further investigation, we found that the technique of identifying dense separators was much less effective on 128 processors.

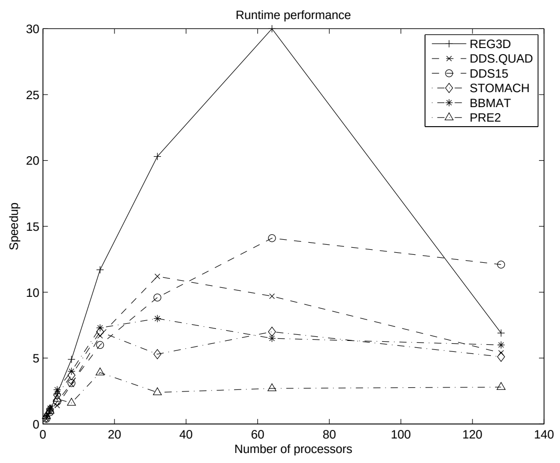
For five of our unsymmetric matrices, the time of the parallel symbolic factorization decreases when increasing the number of processors up to 16. The best speedups for SFpar are for matrices BBMAT (8-fold on 32 processors) and LHR71C (9-fold on 32 processors). These three matrices are relatively denser than the other matrices, and the technique of exploiting dense separators is more effective. For matrix PRE2, which is more unsymmetric and has very little dense structure, the algorithm achieves a speedup of almost 4 on 16 processors. For the other three matrices in our test set (TORSO1, TWOTONE and G7JAC200SC), SFpar achieves only a speedup of two, even with increasing number of processors. However, for these matrices the time of SFpar is generally smaller than the time of ParMetis. For torso1, the time spent in the graph partitioning phase can be ten times larger than the time spent in SFpar.

For memory scaling, we display in Figures 5.3 and 5.4 the decrease in the memory usage of the parallel symbolic factorization algorithm SFpar with respect to the memory needs of the sequential symbolic factorization algorithm SFseq. The memory usage is significantly reduced for all the matrices, with the exception of torso1. In the former case, the reduction of memory is observed all the way up to 128 processors. The largest reduction is obtained with reg3D for which on 128 processors, the memory need of SFpar is 38-fold smaller than that of SFseq. For the unsymmetric matrices, the largest reductions occur with matrices bbmat (up to 25-fold) and lhr71c (up to 21-fold). Note that on a large number of processors, for the unsymmetric matrices, the memory used to store the replicated data may become more than what is needed to store the distributed data. For example, for stomach the replicated data represents 77% of the memory needs on 128 processors. Still, these matrices help show that the parallel algorithm exhibits good memory scalability.

For torso1, the parallel symbolic factorization uses only 4.6 times less memory than the sequential symbolic factorization. However, in this case ParMetis leads to very unbalanced partitioning, as quantified by differences in the sizes of the separators (in terms of the number of their indices and the number of their nonzero elements of the input matrix) at each level of the separator tree. For example, we have found that on 64 processors, the largest ratio of the maximum number of indices per separator relative to the level's average number of indices per separator is obtained at the second level of the separator tree and it is 9.6. On 128 processors this ratio is 9.9 and it is also obtained at the second level of the separator tree. At the lower level of the separator tree, the ratio of the maximum number of nonzero elements in a partition relative to the average number of nonzero elements at this level is equal to 24.9 on 64 processors and 43.3 on 128 processors. As we mentioned already earlier, our data distribution algorithm is not the best choice for unbalanced separator trees and can lead to disparity in the per-processor memory usage in such cases and to low overall memory reduction rates. Indeed, in the case of torso1, we find that the maximum per-processor memory usage is 7.6 times larger than the average memory usage on 64 processor and is 8.5 times larger on 128 processors, consistent with the above hypothesis. We plan to address this problem in our future work.

5.4. Scalability analysis. Recall that our main goal of this research is to develop a parallel symbolic factorization algorithm that is memory scalable. We now quantify the memory scalability of SFpar using the matrices arising from the 11-point discretization of the Laplacian operator on three-dimensional cubic grids. The notion of iso-efficiency is useful when studying the scalability of a parallel algorithm. A parallel algorithm is considered highly scalable if it keeps a constant efficiency as the number of processors increases while the problem size per processor is kept constant, where the problem size could be the size of the dataset or the amount of work. For a memory scalability study, the suitable quantity would be the dataset size, which is the size of the L and U factors in this case. The memory efficiency on P processors is computed as the ratio of the memory usage of the sequential algorithm (SFseq) over the product of the maximum per-processor memory usage of the parallel algorithm (SFpar) and P. That is, Mem. eff. $= \frac{mem(SFseq)}{max.mem(SFpar) \times P}$. Therefore, we choose the dimensions of the cubic grids in such a way that the nonzeros of L and U are maintained roughly constant per processor while increasing the number of processors. Theoretically, when nested dissection ordering is used, for each matrix of order n, the number of nonzeros is on the order of $O(n^{4/3})$ while the amount of work is on the order of $O(n^2)$ [8].

Note that since SFseq and SFpar use very different data organizations (SFseq is column oriented, whereas SFpar is both column and row oriented), the total memory



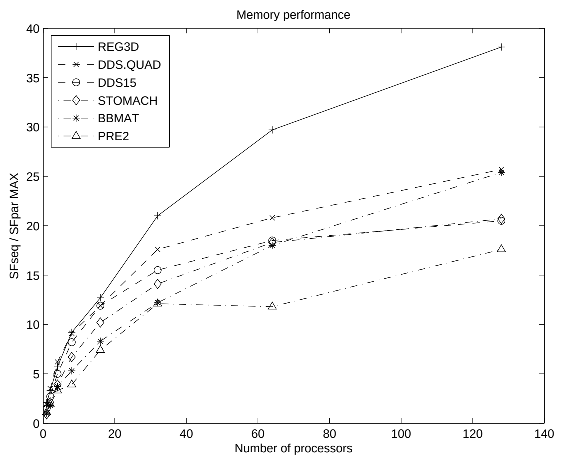
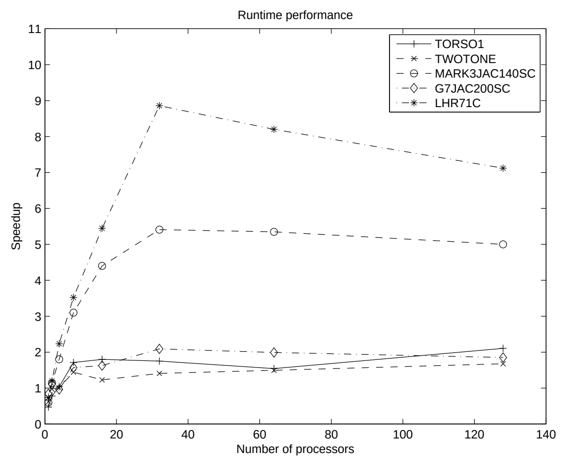


FIG. 5.3. *Speedup and memory saving of the parallel symbolic factorization algorithm for the first six matrices in our test set.*

requirements are very different. In fact, when run on one processor, SFpar usually uses less memory than SFseq. Therefore, for ease of understanding, we will use a simulated single processor run of SFpar as the sequential code baseline. Since the ParMetis ordering gives different numbers of fill-ins using different numbers of processors, we do not use the actual run of SFpar on one processor. Instead, we compute a lower bound of the memory need of Algorithm 1 based on the amount of fill in L and U after ParMetis ordering using P processors. This algorithm stores the structure of L by columns and the structure of U by rows. It also assumes that the input matrix A is stored by columns below the diagonal and by rows above the diagonal. The memory needs are as follows. Two arrays of size n and two arrays of size equal to the supernodal structure of L and U store the structure of the factors. Two arrays of size n and two arrays, each of size twice the number of supernodes, store the pruned graphs of L and U. Two arrays, of size n, are used to identify supernodes. One array, of size n, is used as a marker during the computation of the structure of L and U. Two arrays of size n and two arrays of size equal to the number of nonzeros of the original matrix A are used to store the structure of the input matrix.

Table 5.2 reports the size of the grids used, the number of operations and the size



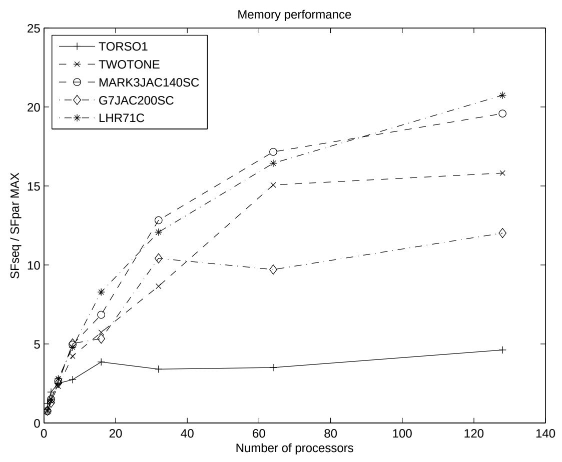


FIG. 5.4. *Speedup and memory saving of the parallel symbolic factorization algorithm for the last five matrices in our test set.*

of the factors per process, the time of SFpar and the memory used in the algorithm. The last two columns give the computed memory lower bound of a sequential symbolic factorization and the memory efficiency. We see that the algorithm achieves very respectable efficiencies ranging from 89% on 2 processors to 25% on 128 processors. The decrease in efficiency with increasing number of processors is mainly due to the two duplicated arrays of size n, which increase with increasing problem size. For example, the size of the two arrays on 64 processors is 3.2 MBytes, and on 128 processors is 4.9 MBytes. This represents almost half of the per-processor memory need. But since now the memory demand in the symbolic factorization phase is not a bottleneck any more, there is no point in reducing this memory usage at the expense of the extra runtime.

The parallel runtime is almost constant up to 64 processors. But we observe a significant increase on 128 processors. With further study, we found that this is due to the same reasons as were previously observed with matrix REG3D. That is, the technique of identifying dense separators was less effective on 128 processors than on smaller number of processors.

P	Grid size	flops/P (10^9)	$nnz(L+U)$/P (10^6)	Time	Mem. SFpar MAX	Mem. SFpar AVG	Mem. SFseq	Mem. Eff. (%)
1	29	5.5	9.5	.8	5.3	5.3	5.2	98
2	34	8.6	10.2	.9	5.0	4.9	9.4	94
4	39	12.2	10.0	.8	5.3	4.8	17.6	83
8	46	16.2	9.9	.8	6.0	4.8	34.9	73
16	53	21.2	9.7	.7	5.9	5.0	63.0	67
32	62	29.6	9.8	.8	6.8	5.5	111.1	51
64	74	46.0	10.7	1.1	8.8	7.2	220.2	39
128	85	55.8	10.0	9.8	12.1	9.2	390.2	25

TABLE 5.2

Time and memory usage of the parallel symbolic factorization algorithm when the average number of nonzeros of L and U per processor is kept constant.

6. Conclusions.

We have presented a parallel symbolic factorization algorithm for sparse Gaussian elimination with static pivoting. The algorithm is suitable when the equations and variables are preordered by a nested dissection algorithm, from which a binary separator tree is available, and our symbolic factorization algorithm exploits parallelism exposed by this separator tree.

The performance of the parallel algorithm depends on the partitioning and reordering (e.g., by PARMETIS), the structural symmetry and the density of the matrix. The more symmetric is the matrix, the smaller is the symmetrically pruned graph, which leads to a smaller amount of communication. The denser the matrix, the denser the separators, which the parallel algorithm exploits for faster structural analysis. For the symmetric test cases, the parallel symbolic factorization algorithm SFpar has achieved a 14.1-fold speedup on 64 processors when compared with the sequential symbolic factorization algorithm SFseq (DDS15.K-SM, Figure 5.3), and the entire solver SLU_SFpar is up to 20% faster than the solver SLU_SFseq (DDS.QUAD.K-SM, Figure 5.1). For the unsymmetric cases, the parallel algorithm SFpar has achieved a 9-fold speedup on 32 processors (LHR71C, Figure 5.4), and the entire solver SLU_SFpar is up to 16.7% faster than the solver SLU_SFseq on 64 processors (BBMAT, Figure 5.1).

On one processor, SFpar can be slower than SFseq. But on a larger number of processors, SFpar can be much faster than SFseq.

In the re-distribution step, both versions achieve good speedup. During this step, the communication patterns for numerical factorization and triangular solutions are determined. In SLU_SFseq, since every processor has a copy of the reduced structures of L and U, there is no communication when computing the communication pattern. In SLU_SFpar, with the distributed reduced structures of L and U, there are fewer accesses to the data on each processor, but there are more all-to-all communications to determine the communication pattern. This explains why on smaller numbers of processors, RDseq is faster, while on large number of processors, RDpar is faster (up to four times faster for several matrices).

The SFpar algorithm exhibits very good memory scalability. For many matrices, on one processor the memory usage of SFpar is less than that of SFseq. For all the matrices, with the exception of one (TORSO1), the maximum per-processor memory usage of SFpar continues to decrease with increasing number of processors up to 128. For the unsymmetric test cases, on 128 processors, the SFpar maximum per-processor memory reduction is up to 25-fold compared with SFseq, and the entire SLU_SFpar solver has up to 5-fold reduction in maximum per-processor memory need compared with the entire SLU_SFseq solver (BBMAT, Figure 5.2).

For matrix TORSO1, PARMETIS leads to an unbalanced partitioning and a corresponding unbalanced separator tree. Our data distribution algorithm is not well suited for unbalanced separator trees, and SFpar leads to a modest reduction of the memory requirement of the symbolic factorization. The unbalance in the partitioning can be related to the fact that the graph partitioning is applied on the symmetrized input matrix $|A| + |A|^T$. Two possible solution can be envisaged, which could remedy this problem. First, one could try to use a partitioning algorithm that accounts for the unsymmetry of the input matrix. Second, one could use a better mapping of the data to processors, that takes into account that the separator tree is unbalanced. We will leave these options to be addressed in the future.

Nothwistanding that, the experimental results show that SFpar greatly increases the problem size the parallel SuperLU solver can handle, and the symbolic factorization effectively removes a memory bottleneck in many, if not most of, common cases.

Acknowledgments. The authors thank the anonymous reviewers for their careful reading and the suggestions to improve the paper.

REFERENCES

[1] T.-Y. Chen, J. R. Gilbert, and S. Toledo. Toward an efficient column minimum degree code for symmetric multiprocessors. *Proceedings of the 9th SIAM Conference on Parallel Processing for Scientific Computing*, 1999.

[2] M. Cosnard and L. Grigori. A parallel algorithm for sparse symbolic LU factorization without pivoting on out–of–core matrices. In *Proceedings of the 15th International Conference on Supercomputing*, pages 146–153. ACM Press, 2001.

[3] T. Davis. University of Florida Sparse Matrix Collection. NA Digest, vol. 92, no. 42, October 16, 1994, NA Digest, vol. 96, no. 28, July 23, 1996, and NA Digest, vol. 97, no. 23, June 7, 1997. http://www.cise.ufl.edu/research/sparse/matrices.

[4] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu. A Supernodal Approach to Sparse Partial Pivoting. *SIAM J. Mat. Anal. Appl.*, 20(3):720–755, 1999.

[5] I. S. Duff, R. G. Grimes, and J. G. Lewis. User's Guide for the Harwell-Boeing Sparse Matrix Collection (Release i). Technical Report TR/PA/92/86, CERFACS, 1992.

[6] I. S. Duff and J. Koster. On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM J. Mat. Anal. and Appl.*, 22(4):973–996, 2001.

[7] S. C. Eisenstat and J. W. H. Liu. Exploiting Structural Symmetry in Unsymmetric Sparse Symbolic Factorization. *SIAM J. Mat. Anal. Appl.*, 13(1):202–211, 1992.

[8] A. George and J. W. H. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice Hall, Englewood Cliffs, NJ, 1981.

[9] A. George, J. W. H. Liu, and E. Ng. Communication results for parallel sparse cholesky factorization on a hypercube. *Parallel Computing*, 10:287–298, 1989.

[10] J. R. Gilbert and J. W. H. Liu. Elimination Structures for Unsymmetric Sparse LU Factors. *SIAM J. Mat. Anal. Appl.*, 14(2):334–352, 1993.

[11] A. Gupta. Improved symbolic and numerical factorization algorithms for unsymmetric sparse matrices. *SIAM J. Mat. Anal. Appl.*, 24(2):529–552, 2002.

[12] A. Gupta, G. Karypis, and V. Kumar. Highly scalable parallel algorithms for sparse matrix factorization. *IEEE Transactions on Parallel and Distributed Systems*, 8(5), 1995.

[13] B. Hendrickson and R. Leland. An Improved Spectral Graph Partitioning Algorithm for Mapping Parallel Computations. *SIAM J. Sci. Stat. Comput.*, 16(2):452–469, 1995.

[14] B. Hendrickson and R. Leland. A Multilevel Algorithm for Partitioning Graphs. *Proceedings of SuperComputing*, 1995.

[15] HSL. A collection of Fortran codes for large scale scientific computation, 2004. http://www.cse.clrc.ac.uk/nag/hsl/.

[16] M. Joshi, G. Karypis, V. Kumar, A. Gupta, and F. Gustavson. PSPASES: An efficient and scalable parallel sparse direct solver. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 1999.

[17] G. Karypis and V. Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *Journal of Parallel and Distributed Computing*, 48(1):71–95, 1998.

[18] G. Karypis, K. Schloegel, and V. Kumar. PARMETIS*: Parallel Graph Partitioning and Sparse Matrix Ordering Library – Version 3.1.* University of Minnesota, August 2003. http://www-users.cs.umn.edu/∼karypis/metis/parmetis/.

[19] K. Ko, N. Folwell, L. Ge, A. Guetz, V. Ivanov, L. Lee, Z. Li, I. Malik, W. Mi, C. Ng, and M. Wolf. Electromagnetic systems simulation - from simulation to fabrication. SciDAC report, Stanford Linear Accelerator Center, Menlo Park, CA, 2003.

[20] X. S. Li and J. W. Demmel. SuperLU_DIST: A Scalable Distributed-memory Sparse Direct Solver for Unsymmetric linear systems. *ACM Transactions on Mathematical Software*, 29(2), 2003.

[21] J. Riedy and J. W. Demmel. Parallel bipartite matching for sparse matrix computation, 2005. In preparation.

[22] D. J. Rose and R. E. Tarjan. Algorithmic aspects of vertex elimination on directed graphs. *SIAM J. Appl. Math.*, 34(1):176–196, 1978.

[23] A. van Heukelum. Symbolic sparse cholesky factorization using elimination trees. Master's thesis, Dept of Mathematics, Utrecht University, 1999.

[24] E. Zmijewski and J. R. Gilbert. A parallel algorithm for sparse symbolic cholesky factorization on a multiprocessor. *Parallel Computing*, 7(2):199–210, 1988.