A Parallel Algorithm for Sparse Symbolic LU Factorization Without Pivoting on Out–of–Core Matrices

Michel Cosnard
LORIA – INRIA Lorraine
Nancy, France
Michel.Cosnard@loria.fr

Laura Grigori
LORIA – Université Henri Poincaré
Nancy, France
Laura.Grigori@loria.fr

ABSTRACT

Finding the nonzero structures of the lower and upper triangular factors of an unsymmetric sparse matrix A is an important problem in the field of sparse matrix computations. Complementing previous research on sequential algorithms, we develop a parallel algorithm by appropriately intertwining a fully concurrent algorithm with an experimentally proved efficient algorithm (both algorithms are previous art). The resulting algorithm, intended to use with out–of–core matrices, leads to sensitive improvements in memory usage.

The algorithm is described in terms of its theoretical complexity, its experimental implementation with the MPI communication library the Origin 2000 multiprocessor, as well as its performance on large size real world matrices.

Categories and Subject Descriptors

D.1 [**Software**]: Programming Techniques; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*; G.1 [**Mathematics of Computing**]: Numerical Analysis; G.1.3 [**Numerical Analysis**]: Numerical Linear Algebra—*Sparse, structured and very large systems (direct and iterative methods)*

General Terms

Algorithms, Performance, Experimentation

Keywords

Sparse Gaussian elimination, symbolic factorization, out of core matrices, parallel algorithms, MPI.

1. INTRODUCTION

Transforming a matrix into its LU form is an important basic operation included in the direct methods of solving an unsymmetric linear system of equations $Ax = b$. In the sparse case intelligent algorithms and efficient implementations can lead to lesser memory requirements and to exposing more available parallelism.

In this paper we develop an unsymmetric symbolic factorization method for a distributed memory machine, by adaptively mixing a fully concurrent algorithm (with low memory requirements but inefficient execution [10]) with a sequential, experimentally proven efficient algorithm (that instead requires a large amount of in–core memory [5]). After an extensive bibliographical research (see below) we came to the conclusion that this new algorithm is among the first initiatives, if not *the* first initiative to parallelize the symbolic factorization of the LU factorization without pivoting.

Today's theoretical approach states that, to solve a sparse linear system by a direct method, four operations are involved. First find a matrix permutation P that preserves the sparsity of A. Next, determine the fill by symbolically factoring A and set up the structures of L and U. Finally, numerically factor A and solve two triangular systems to find the solution vector x.

The direct method for an unsymmetric sparse matrix is materialized by two classes of algorithms. They are differentiated by the solutions presented to the problem of maintaining the numerical stability when the matrix is not diagonally dominant. In earlier solutions, the partial pivoting was used during the LU factorization. More recently, another approach (proposed in [7]) develops techniques that replace the partial pivoting while still maintaining the numerical stability. As of today, at least one scalable and very efficient algorithm was developed for the LU factorization without pivoting [7].

These recent evolutions are leading to an increased importance of the symbolic factorization. For very large matrices, a significant part of the overall memory space is needed by structures used during the symbolic factorization, this being a reason for our algorithm being optimized for out–of–core matrices.

This paper's outline is as follows: in section 2 we briefly review notions related to the unsymmetric symbolic factorization as well as sequential algorithms that implement it. Section 3 presents our algorithm with complexity analysis and implementation details. In section 5 we present the experimental results and section 6 concludes the paper.

2. UNSYMMETRIC SYMBOLIC FACTORIZATION

To develop a parallel algorithm for the symbolic factorization, we first briefly review two algorithms from today's published literature. Since our parallel algorithm (presented in the next section) computes the structures of both L and U by columns, for an easier understanding this section's algorithms also compute the structures of L and U column by column, even if they were originally presented with a different notation.

Let A be a sparse unsymmetric $n \times n$ matrix with m nonzero elements that can be decomposed without pivoting into LU, where L is lower triangular with unit diagonal and U is upper triangular. We define $F = L + U - I$ to be the filled matrix containing both L and U. In what follows, $G(A)$ denotes the directed graph associated with matrix A, a_{ij} represents the element of row i and column j of matrix A, while $A(:, i : j)$ represents all the subscripts of the columns from i to j of A.

The aim of the symbolic factorization is to compute the structure of F, containing all nonzero elements of A and new nonzero elements appearing during the numerical factorization. Consider a column j of U for which u_{ij} is nonzero. This column will be updated during the numerical factorization by the column i. We thus need to include the elements of column i of F in column j of F during the symbolic factorization. For illustration we present in figure 1 an example matrix A, its directed graph and the filled matrix F obtained after the symbolic factorization.

2.1 FILL2 Symbolic Factorization

Rose and Tarjan [10] present two algorithms to determine the structures of L and U. In our research we used the second algorithm, noted FILL2, which was proposed for matrices with large fill-in compared to their sparsity, and experimentally proved to be more efficient then the first algorithm, noted FILL1. Algorithm FILL2 computes the fill-in using the following path lemma [10]: if G^+ is the filled graph of A (the graph of F), then (x, y) is an edge of G^+ if and only if there is a path in $G(A)$ from x to y whose intermediate vertices are all smaller than $\min(x, y)$.

The algorithm 1 presents the details of FILL2. To compute the column $F(:, j)$, the algorithm finds all the vertices reachable from j which satisfy the path lemma. To locate these vertices a search from j is performed using an iterative algorithm, starting at 1 and iterating over a threshold. At each iteration, when a vertex k less than the threshold is reached, it is recorded in *front* and it will be examined later to extend the path search for the current value of the threshold. When a vertex i greater then the threshold is reached, it is added to $F(:, j)$. The array *mark* used in the algorithm 1 is initialized with 0. Since each edge is examined once per starting column, Rose and Tarjan show that the time complexity of FILL2 is bounded by $O(nm)$.

ALGORITHM 1. *FILL2*

```
for j := 2 to n do
  mark[j] := j
  for i in A(:,j) do
    mark[i] := j
    add i to F(:,j)
  end for
  for threshold := 1 to j−1 such that mark[threshold] = j do
    add threshold to front
    while front ≠ empty do
      delete vertex k from front
      for i in A(:,k) such that mark[i] ≠ j do
        mark[i] := j
        add i to F(:,j) if i > threshold
        add i to front if i < threshold
      end for
    end while
  end for
end for
```

2.2 Gilbert Peierls Symbolic Factorization

The second algorithm we used during our research was proposed by Gilbert and Peierls in [6]. In this algorithm, the symbolic factorization of column j is obtained as follows: at the end of the $j-1$ iteration, the nonzero structures of the first $j-1$ columns of L and U are known, together with the graph $G(L(:, 1 : j-1))$. For each element of $A(:, j)$, we traverse the graph of $L(:, 1 : j-1)^T$. All the nodes reached during this traversal belong to the structure of column j of F. The algorithm 2 sketches this observations. Let us denote it the GP algorithm.

ALGORITHM 2. *GP symbolic factorization*

```
for j := 1 to n do
  for each i in A(:,j) with a_ij ≠ 0 do
    if i > j and i not in L(:,j) then
      add i to L(:,j)
    end if
    if i < j and i not in U(:,j) then
      add i to U(:,j)
      perform a depth-first traversal from i in
      G(L(:,1 : j − 1)^T)
    end if
  end for
end for
```

ALGORITHM 3. *Depth first search traversal vertex i*

```
for each h with l_hi ≠ 0 do
  if h > j and h not in L(:,j) then
    add h to L(:,j)
  end if
  if h < j and h not in U(:,j) then
    add h to U(:,j)
    perform recursively depth-first traversal from h in
    G(L(:,1 : j − 1)^T)
  end if
end for
```

A complexity analysis of this algorithm can be found in [5]. For each column j, the traversal of $G(L(:, 1 : j-1)^T)$ takes

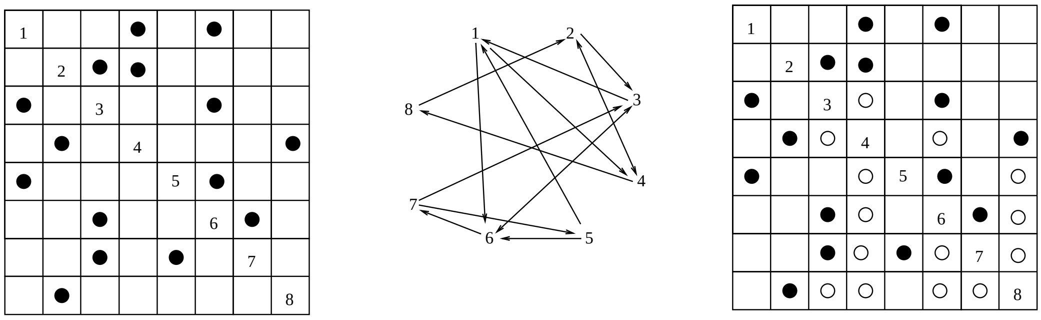

Figure 1: Example matrix A, its directed graph and its filled matrix F.

time on the order of number of operations to multiply the matrix $L(:,1:j-1)$ by the vector $U(:,j)$. The complexity of the entire algorithm was written informally in [5] as *flops(LU)*.

Several optimizations were done on the GP symbolic factorization algorithm. The first optimization, called symmetric reduction, tries to reduce some redundant edges of $G(L)$ and it was introduced in [3] by Eisenstat and Liu. The second optimization, proposed in [2], is to use supernodes (columns of L with an identical structure) during the symbolic factorization. We will explain in detail these optimizations in section 3 where they will be related to our parallel algorithm.

As noted in [5], theoretically none of the two algorithms dominates the other. However, the experiments performed on small sized matrices showed [5] that the GP algorithm outperforms the FILL2 algorithm. The optimizations introduced in the GP algorithm made this gap even bigger. The experimental results from [3] show that the symmetric reduction significantly reduces the symbolic time. In [2] the impact of supernodes is measured and the savings due to the symbolic factorization are outlined.

3. PARALLEL UNSYMMETRIC SYMBOLIC FACTORIZATION

In the previous section, we presented two known algorithms for the unsymmetric symbolic factorization. The first algorithm, FILL2 [10], determines the structure of column j of F using the structures of the first j columns of the original matrix A. Each processor needs the original matrix A in its local memory, no communication between processors and no supplementary memory is needed, inducing this algorithm as a perfect candidate for parallelization.

The GP algorithm has been proven to be very efficient experimentally. To compute column j, a processor p_k will need several of the previously computed columns $1 \ldots j-1$. Some of these columns have to be received from the other processors which computed them. Each processor has to store in local memory its own previously computed columns plus the columns received from the other processors. The overall memory needs roughly equal the memory needed for the symbolic factorization of the entire matrix multiplied by the number of processors. This might be an important impediment to the execution of the LU factorization, since for very large matrices the memory is already known as a potential bottleneck in the symbolic factorization.

The new algorithm was developed starting from an intuitive observation of the relation between the GP algorithm and FILL2. Suppose that during the computation of column j, $u_{ij} \neq 0$. The GP method would include in column j all the elements of $L(:,i)$ while FILL2 would include all the columns of the original matrix A which were normally considered in the computation of column $L(:,i)$ (and were not yet included in the current column j). Remark that FILL2 can be interpreted as a simulation of the computation of $L(:,i)$ using the original matrix A.

In the new algorithm, the key idea is to dynamically switch between GP and FILL2 depending whether a particular processor has (or has not) stored the computed column i of L. Assume that processor p_k computes column j, and during this computation $u_{ij} \neq 0$. If processor p_k *has stored* the computed column i of L (it were the owner or it received it from the owner processor) then include this column in the column j using the GP algorithm. In the other case, when processor p_k did not own i (or *has not stored* the received column i due to memory constraints), then use FILL2 starting at column i of the original matrix A.

Algorithm 4 sketches this observations in the distributed unsymmetric symbolic factorization executed by a processor p_k (SPMD paradigm).

The columns were distributed among processors using a load balancing algorithm, which will be presented in section 4. The mapping creates groups of dependent columns such that the GP algorithm can be used as much as possible.

In the algorithm 4, we use several new notations: $procOwner$ is an array storing the processor owning each column; $workArray$ is a pointer storing the beginning of the necessary array $L(:,i)$ or $A(:,i)$.

Due to the use of FILL2 algorithm in our mixed approach, we have to traverse the elements of $U(:,j)$ in ascending order, this having the side-effect that column j of U will be stored in ascending order. For the ease of understanding, the algorithm presented uses the threshold to traverse the array called $mark$ to find the elements of the column j in ascending order. However, in our implementation we used a heap to store and access the elements of the column j of U in ascending order.

ALGORITHM 4. *Parallel symbolic factorization*

```
for j := 2 to n such that procOwner[j] = myProc do
  mark[j] := j
  for each i in A(:, j) do
    mark[i] := j
    add i to L(:, j) if i > j
  end for
  for threshold := 1 to j − 1 such that
  mark[threshold] = j  do
    add threshold to front
    add threshold to U(:, j)
    if myProc has stored column threshold then
      workArray = L(:, threshold)
    else
      workArray = A(:, threshold)
    end if
    while front ≠ empty do
      delete vertex i from front
      workArray = A(:, i) if i ≠ threshold
      for each k in workArray such that mark[k] ≠ j
      do
        mark[k] := j
        add k to front if k < threshold
        add k to L(:, j) if k > j
      end for
    end while
  end for
  send column j to all other processors
  try to receive columns from other processors
end for
```

3.1 Complexity Analysis

Suppose that processor p_k computes a column j. If, for all the elements i belonging to $U(:, j)$, the processor p_k stored the column $L(:, i)$ in its local memory, then the mixed algorithm will compute the column j using only the GP method. In that case, the complexity for column j is $flops(L(:, 1 : j-1) \times U(:, j))$. However, if there exists at least an element i such that the processor p_k doesn't have the column i of L, then the FILL2 algorithm is used for this element, and the complexity is bounded by $O(m)$, where m represents the number of elements in the matrix A. Added up over all columns owned by a processor, the complexity of the algorithm with a block cyclic distribution of the matrix is bounded by $O((mn + flops(LU))/p)$, where p is the number of processors.

3.2 Reducing the Original Matrix

Recall that each processor has to store the original matrix A in its local memory, imposing a minimal threshold on memory requirements. We try to lower this threshold by removing unnecessary edges from A. Let a_{ij} be a nonzero element such that $i < j$. If j is involved in the computation of some column by the FILL2 algorithm, then the column i will be also included in that computation. This means that we can eliminate all the edges $a_{kj}, j < k$ if a_{ki} is also nonzero. We could extend this reasoning by using paths in A to eliminate unnecessary edges, but the time for the reduction would become significant compared to the benefits obtained. The reduced matrix is computed in parallel before the symbolic factorization algorithm. Each processor first reduces its own columns and then sends them to the other processors.

For the ease of understanding, we presented a column by column factorization version of this algorithm. But in an efficient implementation, we need to introduce supernodes and symmetric pruning. Next we study if the use of supernodes and symmetric pruning is possible and under what circumstances. We also give details of the respective implementation.

3.3 Pruning the Symbolic Structure

The symmetric reduction was introduced in [3] by Eisenstat and Liu to reduce some redundant edges in the graph of L by using the symmetry of the filled matrix F. The symmetric reduction of $L(:, 1 : j)$ is obtained by removing all nonzeros l_{ki} for which $l_{ji}u_{ij} \neq 0$ for some $k > j$.

In the GP symbolic factorization algorithm presented in section 2 the symmetric pruning can be computed by an efficient method [3], and it significantly reduces the total factorization time. A supplementary array $fsnz$ stores the first symmetric nonzero found for each column i. All the columns $j > fsnz[i]$ being updated by i will access those entries in the structure of $L(:, i)$ that are less than or equal to $fsnz[i]$.

Note that the use of symmetric pruning is possible in GP algorithm because in the computation of a column j, it uses $L(:, i)$ and $L(:, fsnz[i])$. We implement this technique in our algorithm and we identify the first symmetric nonzero. All the columns $k > fsnz[i]$ being updated by i will access only the entries less than or equal to $fsnz[i]$ in $L(:, i)$ if the processor p_k has stored also the column $L(:, fsnz[i])$. Otherwise, we cannot use the technique of symmetric pruning, and we have to access all the elements of column $L(:, i)$.

We can remark a major difference from the symmetric pruning in Eisenstat and Liu algorithm. At each iteration j we identify $fsnz[i]$ using $L(:, i)$ or using $A(:, i)$. It means that $fsnz[i]$ at iteration j can have a bigger value than $fsnz[i]$ of Eisenstat and Liu algorithm, and we access more redundant edges that this algorithm.

3.4 Unsymmetric Supernodes

Using the supernode concept, we try to group together successive columns with the same nonzero structure, so they can be treated as a dense matrix. This leads to a speed-up of the symbolic factorization and this scheme was used in the SuperLU solver [2].

At the end of each symbolic factorization step, we decide whether the new column j belongs to the same supernode as column $j-1$, if the column $j-1$ was computed by the same processor. Furthermore, to compute the column j we use the previous supernodes instead of using the columns $1, \dots, j-1$.

In our algorithm we distinguish two cases. If processor p_k has stored the supernode l, the GP algorithm will be used and it will include this supernode and increment threshold with the appropriate value to not access the next columns of the supernode. We can mark as visited all the elements of the columns of A belonging to the supernode. However, in practice we noticed that the gains are good if we will mark only the elements of A of the first column of the supernode as visited.

If processor p_k doesn't have the supernode l, but knows that a column i belongs to the supernode l, then it is enough to consider only the first column of A of this supernode during the computation of FILL2 algorithm. We conclude that supernodes are used in both versions of the mixed algorithm.

3.5 An Implementation Using MPI

We implemented a fully asynchronous algorithm which enables complete overlap between communication and computation. The columns owned by a processor are stored in contiguous memory. This way, we can send several columns together and we eliminate the need for packing and unpacking the messages in a send-receive operation. To decrease the communication latency, at the end of each iteration, we send a group of columns of L if the sum of their sizes is bigger then a minimum chosen value.

To receive columns from the other processors, an asynchronous nonblocking MPI_Irecv is pending. As long as raw memory is available, we try to receive necessary columns from the other processors by checking the status of the pending receive. As soon as a message is received, we post a new pending receive and try to receive new columns. When the processor becomes short of memory, it uses the new columns received just to update the corresponding values necessary for the symmetric pruning and the supernodal factorization.

4. MAPPING ALGORITHM

In this section we describe the algorithm we used to partition the columns between processors. In the mixed approach, we aim at using as much as possible the GP algorithm. For this, we need to group dependent columns on the same processor, while at the same time rightfully balancing their load. A generic mapping algorithm would consider that (1) *elimination tree*[1] of $A + A^T$ can be used to identify all dependencies and (2) the independence property of columns of disjoint subtrees can induce a right mapping of subtrees to processors.

A related problem is to compute the load associated to each column. This can be done by using, again, the elimination tree of $A + A^T$. In fact, as the factors of $A + A^T$ are an upper bound of the factors L and U of A, we can evaluate an upper bound for the load of each column by using the column counts of the factors of $A + A^T$. Efficient algorithms exist to compute the row and column counts of the factor L of a symmetric matrix, without computing the factor itself.

Given all benefits that the elimination tree of $A + A^T$ seem to offer to a mapping algorithm one is easily tempted to question on the associated tradeoffs. It is true that building an elimination tree has its own additional time and memory requirements. However, given that this tree is usually constructed at the beginning of the LU factorization (reorder rows of A in order to reduce the fill-in), there's no actual overhead when re-using it in a mapping step too.

[1]Many sparse matrix algorithms use elimination trees to model the parallel factorization. Etrees were first used in the symmetric case, and a detailed description of the role and use of elimination trees can be found in [8]. In the case of an unsymmetric matrix, the elimination tree of $A + A^T$ gives all the dependencies which can appear during the LU factorization without pivoting.

Several mapping algorithms were proposed in the literature, in the context of the Cholesky factorization. We used the proportional mapping algorithm proposed by Pothen and Sun in [9]. This algorithm tries to achieve low communications and takes into account the workload distribution of the tree. It starts by assigning all processors to the root. The columns of the root are assigned in a wrap-map manner to these processors. Then it considers the subtrees of the root in descending workload order. It assigns to each such subtree a number of processors proportional with the workload of the subtree. The method continues until only one processor remains assigned to each subtree.

Due to rounding to integral number of processors, some details in implementing the algorithm are important in obtaining an effective mapping, and these details are explained in [9]. The advantage of this algorithm is that the message exchanges in a subtree are limited only to those processors which are assigned to this subtree, thus ensuring a lower communication cost.

We consider the symbolic factorization of each column as a task. To be able to use the proportional mapping algorithm in our mixed approach, we did minor modifications to the algorithm proposed in [9]. For example, at each step of the proportional mapping, the root of the subtree contains only one column. The computation of this column is assigned to only one processor, the least loaded in the set.

5. NUMERICAL RESULTS

In this section, we show experimental results obtained when applying the proposed factorization method on real-world matrices. We choose to test our factorization method on a Silicon Graphics cache-coherent distributed shared memory architecture, the Origin 2000. The configuration contains 64 R10000 processors clocked at 195Mhz. The total physical memory size is 24Gbytes. The cache hierarchy has two-levels: L1 is a separated 32kbytes data and instruction on-chip cache while L2 is a 4Mbytes on-board unified cache. The interconnection network is a 4-ary hypercube, achieving a peak bandwidth between nodes of 140Mbytes/sec. We used the MIPSPro C compiler v7 with the third optimization level enabled (`-O3`).

We used several large size matrices from a variety of application domains. These matrices[2] and their characteristics are presented in table 1. We employed a fill-reducing heuristic: the approximate minimum degree column ordering by Davis et al. [1].

From device simulation matrices we used wang3. The matrices raefsky3, raefsky4, af23560 and venkat01 are used in fluid flow modeling. The bbmat and ex11 matrices are used also in fluid flow modeling. The third column represents the matrix order, and the fourth column contains the number of nonzeros $|A|$ before symbolic factorization. With A_r denoting the matrix obtained after the reduction, column five lists the number of entries in A_r. The sixth column lists the total number of factor entries obtained in $L + U$.

[2]Matrices were obtained from the Harwell-Boeing Collection and from the ftp site maintained by Tim Davis of the University of Florida, ftp.cise.ufl.edu/pub/faculty/davis/matrices.

Table 1: Benchmark matrices

Mat	Name	Order	$\|A\| \times 10^3$	$\|A_r\| \times 10^3$	$\|L+U\| \times 10^3$
1	raefsky3	21200	1488	113	8132
2	raefsky4	19779	1328	127	13023
3	af23560	23650	484	326	11748
4	wang3	26064	177	177	10920
5	venkat01	62424	1717	294	11845
6	bbmat	38744	1771	436	34819
7	ex11	16614	1096	115	11431

By comparing the fourth and fifth columns one can notice the important improvements in memory usage due to the reduction of A, A_r being used by the mixed algorithm. This reduction goes up to 92%. Note that for matrix wang3, we didn't obtain any reduction. However, this matrix has a large fill-in and the memory needs for storing the factors L and U largely dominate the memory needs for storing the original matrix A.

To simulate an out–of–core symbolic factorization, we first evaluated the memory needs for the symbolic factorization of each matrix. Then, we imposed upper limits to the memory allocation. In table 2, we report the time performance of the mixed algorithm on one processor. M signifies the size of the memory needed for the factorization of each matrix, and M is successively divided by 4, 8, 16, 32 and 64. The allocated buffer is managed as a circular buffer. It works as follows: suppose at the computation of column j there is no more memory available, columns $i, i+1, \ldots, j-1$ being already stored by the respective processor in its local memory. To free the memory, the processor will send columns $i, i+1, \ldots, k,\ k < j$ to another processor. From that point on, if the computation of a column will involve one of the columns $i, i+1, \ldots, k$, the mixed approach will use the FILL2 algorithm.

We can remark the factorization time depends directly on the amount of reduction in memory size: the more we reduce the memory, the longer the factorization time is. However, the step increase in time length will become insignificant when the reduction in memory will be more important (this corresponds to the moment when the mixed approach starts using almost entirely the FILL2.)

The second column in table 2 reports the results obtained without any reduction in memory. In this case, the mixed approach uses only the GP algorithm, and this shows the effectiveness of the GP approach compared to the mixed approach.

In tables 3 and 4 we evaluate the time performance of the mixed approach on the 195 MHz Origin 2000. The time spent in reducing the original matrix A is included in the factorization time. In the experiments performed on p processors, the memory requirement is divided by p, and each processor has enough memory to store only the columns it has to factor.

In the first version of our algorithm we used a block–cyclic distribution of the columns among processors. Table 3 re-

Table 2: Time performance (in seconds) of the mixed approach on one processor of 195 MHz Origin 2000

Mat	P=1 M	P=1 M/4	P=1 M/8	P=1 M/16	P=1 M/32	P=1 M/64
1	1.	6.31	6.7	7.3	8.9	9.2
2	1.6	8.4	9.8	11.3	14.1	15.2
3	2.2	12.7	14.5	14.9	17.6	18.9
4	1.8	15.2	20.9	36.3	38.9	39.9
5	1.8	12.	13.8	15.4	16.2	17.1
6	6.5	21.7	39.9	43.	46.7	47.6
7	1.5	6.5	7.2	9.4	10.7	11.4

Table 3: Time performance (in seconds) of the mixed approach on 195 MHz Origin 2000 with a block cyclic distribution of the columns

Mat	P=4	P=8	P=16	P=32	P=64
1	6.67	4.9	4.1	2.4	2.1
2	6.3	5.6	5.	4.2	3.07
3	16.9	11.6	7.7	4.7	3.8
4	15.4	9.5	6.	3.6	3.9
5	17.	15.3	12.9	10.02	7.3
6	28.4	23.6	18.5	12.3	8.2
7	6.3	5.1	4.2	3.63	2.8

flects our experiments. We obtained better results by using a column block of 100. For all the matrices, the factorization time continuously decreases when increasing the number of processors up to 64. However, the execution times obtained on a small number of processors are not satisfactory, and show the necessity of a mapping algorithm that helps grouping dependent columns on a same processor.

In table 4 we report the results obtained by the second version of our algorithm, which uses the proportional mapping algorithm presented in section 4. We can remark a major improvement on $4, 8$ and 16 processors. The mapping scheme on a small number of processors is effective, and thus the GP algorithm is principally used in the mixed approach. However, on a larger number of processors, the mapping scheme is less effective, and the results are similar to a block cyclic distribution of the matrix. Further research is needed to develop mapping algorithms for a larger number of processors.

Let us detail some of the implementation aspects that helped

Table 4: Time performance (in seconds) of the mixed approach on 195 MHz Origin 2000 using the proportional mapping algorithm

Mat	P=4	P=8	P=16	P=32
1	1.57	1.53	1.41	1.53
2	1.68	2.09	2.31	2.73
3	1.92	3.85	3.7	4.05
4	1.56	3.9	3.25	6.2
5	2.3	2.2	2.3	1.7
6	6.94	6.98	7.72	10.6
7	1.91	3.1	2.85	2.6

improving the second version of our algorithm. We use the available memory to store our own columns, and also the columns received from the other processors. Let $i, i+1, \ldots, k$ be several received columns stored contiguously starting at address X. When we arrive at this address with the storage of our own columns, we will mark as "unstored" the columns $i, i+1, \ldots, k$, and we will store here our own columns, overwriting whatever previous content.

For several matrices, the size of the pruned graph needed by the GP algorithm is only a small part of the size of L. We take advantage of this property by reducing the size needed to store the columns received from the other processors. In fact, if the size of the pruned column is significantly less than the size of the entire column, we keep in memory only the pruned column.

In a further quest for performance, we increased the memory size on each processor such that a processor could store in a separate buffer several columns received from the other processors. For the first version of our algorithm, the improvements obtained were not significant. One reason for this is given by the block-cyclic distribution of the columns, especially when the computation of column j involves columns in its locality. Those columns are processed by their respective owner processor, and they will be received after the computation of column j.

Even if the improvements obtained with the second version of our algorithm (which uses the proportional mapping algorithm) were significant enough, more complicated techniques should be used to take advantage of the supplementary memory available. In the current implementation, we continuously store the first columns received until all the memory available is filled up. More complex techniques could be used, for example storing only the columns that are close to the column to be factored; this implies a more intricate memory management, or a costly synchronization between processors; such mechanisms will be considered in our future work.

In conclusion, contrary to the GP algorithm, the mixed algorithm is designed to allow manipulating matrices larger than what fits in the core memory. When we applied the former to out of core matrices its performance showed to be inferior to the latter. For example, the execution time was 2.3 times longer for matrix bbmat with an 8x memory reduction.

We can remark that the execution times of the algorithm does not continuously decrease with the number of processors. This is explained by the fact that in our experiments, the smaller the number of processors, the larger the memory allocated is. This leads to a more prevalent utilization of the GP algorithm, a larger number of supernodes and a smaller pruned graph. With these reasons the execution time becomes equivalent to, or sometimes even smaller than, the execution times on a larger number of processors.

6. CONCLUSIONS AND FUTURE WORK

In this paper we presented a parallel algorithm for the unsymmetric symbolic factorization. The algorithm is designed by intertwining two sequential algorithms. Roughly put, it combines the non-dependence characteristics of the first algorithm with the efficiency of the second. Its use is intended for the symbolic factorization for out–of–core matrices that have a large fill-in compared to their sparsity. The lower bound of the required memory size on each processor is the reduced original matrix A and the memory required to store the owned columns.

To conclude, let us present the characteristics that distinguish our algorithm. We use as much as possible the GP algorithm. For very large matrices for which the memory size is an impediment, we store only the original reduced matrix A and the columns to be factored by a processor.

Our future work includes the development of memory management techniques helping receive and store only adequate columns from the other processors.

7. REFERENCES

[1] T. A. Davis, J. R. Gilbert, S. I. Larimore, and E. G. Ng. A column approximate minimum degree ordering algorithm. Technical Report TR-00-005, Univ. of Florida, 2000.

[2] 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. Matrix Anal. Appl.*, 20(3):720–755, 1999.

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

[4] J. R. Gilbert. An efficient parallel sparse partial pivoting algorithm. Technical Report 88/45052-1, Christian Michelsen Institute, 1988.

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

[6] J. R. Gilbert and T. Peierls. Sparse partial pivoting in time proportional to arithmetic operations. *SIAM J. Scientific and Statistical Computing*, 9:862–874, 1988.

[7] X. S. Li and J. W. Demmel. A Scalable Sparse Direct Solver Using Static Pivoting. *9th SIAM Conference on Parallel Processing and Scientific Computing*, 1999.

[8] J. W. H. Liu. The Role of Elimination Trees in Sparse Factorization. *SIAM J. Matrix Annal. Appl.*, 11(1):134–172, 1990.

[9] A. Pothen and C. Sun. A Mapping Algorithm for Parallel Sparse Cholesky Factorization. *SIAM J. on Scientific Computing*, pages 1253–1257, 1993.

[10] D. J. Rose and R. E. Tarjan. Algorithmic aspects of vertex elimination on directed graphs. *SIAM J. Appl. Math.*, 34(1):176–197, 1978.