Parallel Pivots LU Algorithm on the Cray T3E

R. Asenjo
E.L. Zapata

February 1999
Technical Report No: UMA-DAC-99/01

Published in:

4th International Conference of the ACPC (ACPC'99)
University of Salzburg , Austria, pp. 38-47, Feb. 16-18, 1999

University of Malaga
Department of Computer Architecture
C. Tecnologico • PO Box 4114 • E-29080 Malaga • Spain

Parallel Pivots LU Algorithm on the Cray T3E*

Rafael Asenjo and Emilio Zapata

Computer Architecture Department
University of Málaga, Spain,
{asenjo,ezapata}@ac.uma.es,
WWW home page: http://www.ac.uma.es

Abstract. Solving large nonsymmetric sparse linear systems on distributed memory multiprocessors is an active research area. We present a loop-level parallelized generic algorithm which comprises analyse-factorize and solve stages. To further exploit matrix sparsity and parallelism, the analyse step looks for a set of compatible pivots. Sparse techniques are applied until the reduced submatrix reaches a threshold density. At this point, a switch to dense routines takes place in both analyse-factorize and solve stages. The SPMD code follows a sparse cyclic distribution to map the system matrix onto a $P \times Q$ processor mesh. Experimental results show a good behavior of our sequential algorithm compared with a standard generic solver: the MA48 routine. Additionally, a parallel version on the Cray T3E exhibits high performance in terms of speed-up and efficiency.

1 Introduction

The kernel of many computer-assisted scientific applications is to solve large sparse linear systems. We find examples of these kinds of applications in optimization problems, linear programming, simulation, circuit analysis, fluid dynamic computation, and numeric solutions of differential equations in general.

Furthermore, this problem presents a good case study and is a representative computational code for many other irregular problems. More precisely, this problem represents those in which the computational load grows with the execution time (fill-in) and matrix coefficients change its coordinates due to row/column permutations (pivoting).

We say that a matrix is sparse if it is advantageous to exploit its null elements with the development of a sparse version of an algorithm, instead of a dense one. This way, using sparse techniques is justified when sparsity is big enough, and when it remains more or less constant during the process. However, if the matrix suffers from fill-in it will be worthwhile to combine sparse and dense approaches: we should choose the point in the code in which it will be advantageous to switch to a dense code.

* The work described in this paper was supported by the Ministry of Education and Science (CICYT) of Spain under project TIC96-1125-C03, by the European Union under contract BRITE-EURAM III BE95-1564, by the Human Capital and Mobility programme of the European Union under project ERB4050P1921660, and by the Training and Research on Advanced Computing Systems (TRACS) at the Edinburgh Parallel Computing Centre (EPCC)

Therefore, our unsymmetric sparse system solver algorithm follows sparse processing techniques until the reduced submatrix reaches a certain threshold density. At this point, we switch to a parallel dense LU factorization code which applies partial pivoting and uses BLAS as much as possible. Forward and backward substitution are also parallelized, taking into account that there should be sparse and dense versions for both stages. The algorithm is mapped onto a two-dimensional mesh by a sparse cyclic distribution.

Experimental results will be conducted on a Cray T3E with 16 DEC 21164 (Alpha EV-5) processors at 300 MHz with a peak performance of 600 Mflops per processor. We have used `shmem-put` as communication routines on this distributed memory machine. The unsymmetric sparse matrices used to test our algorithm are taken from Harwell-Boeing and University of Florida sparse matrices collections. In addition, we have compared our sequential code with a standard generic sparse solver: the MA48 routine developed by Duff and Reid [17]. We also want to present some comparisons with UMFPACK and SuperLU codes, and test the parallel algorithm with a bigger number of processors, for the final version of this paper.

The next section presents some background and alternatives to solve sparse linear systems, justifying the selection of the alternative presented in this work. Section 3 gives a summarized description of this complex code, avoiding deep implementation details for the sake of shortness. Experimental results, validating the sequential code, comparing with the MA48 routine, and presenting parallel performance, comprise section 4. Finally, the related work and conclusions section close the paper.

2 Sparse systems solving methods

Let us present the linear system of n equations as:

$$Ax = b \tag{1}$$

where A is a non-singular unsymmetric sparse matrix of dimensions $n \times n$ with $\alpha = c \cdot n$ nonzero elements (entries), so that $\alpha \ll n^2$. This way, c represents the average number of entries per row (or column). The density of the matrix is $\rho = \alpha / n^2$, and sparsity can be defined as $\sigma = 1 - \rho$.

One possible classification for the alternatives we find to solve equation 1, is the following:

- Iterative
- Direct
  - Frontal
  - Multifrontal
  - Supernode
  - Generic

Iterative solvers are attractive in many situations due to the lack of dependencies in the sparse matrix-vector multiplication (its kernel), making the parallel

implementation of the code easy. However, there is no one single iterative method robust enough to solve all sparse linear systems accurately and efficiently.

We will focus on those methods based on Gaussian elimination (direct methods), and particularly on the LU factorization of the sparse matrix: $\Pi A \Gamma = LU$, where Π and Γ are permutation matrices, and L and U are lower and upper triangular matrices, respectively. In more detail, if we call π and γ the permutations vectors, the necessary steps to solve the system are:

1. Factorize A so that $A_{\pi_i,\gamma_j} = (LU)_{ij} \ \ \forall\, i,j, \ \ 1 \leq i,j \leq n$. We obtain the L and U matrices and corresponding permutation vectors, π and γ.
2. Permute b following $d_i = b_{\pi_i}, \ 1 \leq i \leq n$, to obtain the vector d.
3. Solve the system $Ly = d$ to obtain y (forward substitution).
4. Solve the system $Uz = y$, to obtain z (backward substitution).
5. Permute z following $x_{\gamma_j} = z_j, \ 1 \leq j \leq n$, resulting in the solution vector x.

As we see in the previous classification, from the algorithmic point of view, there are four approaches to solve LU factorization. Frontal schemes can be regarded as an extension of band or variable-band schemes and will perform well on systems whose bandwidth or profile is small. The multifrontal scheme is an extension of the frontal method. At the beginning, this extension permits efficiency for matrices with symmetric or nearly symmetric pattern. More recent works present multifrontal LU factorization for unsymmetric sparse matrices such as UMFPACK [11]. Another nonsymmetric system solver is based on supernode techniques: the SuperLU [13] code is a left-looking, blocked algorithm which includes symmetric structural reduction for fast symbolic factorization, and supernode-panel updates to achieve better data reuse in cache using BLAS.

Finally, there are the generic approaches (such as MA48 [17] or Y12M [33]), with the following main characteristics:

- The pivot selection stage aims to preserve sparsity and to guarantee numerical stability.
- Do not impose any restriction on the system matrix.
- The sparse data structure is used on the whole code, even on the innermost loop.
- The loop structure in the sparse generic codes is similar to the dense counterparts, but these loops traverse sparse data structure instead of dense ones.

The last two issues led us to focus on these generic methods. In general, we can say that multifrontal and supernode codes convert a sparse problem into a dense subproblems hierarchy. This way they profit from the good behavior of dense codes (regularity, exploiting the cache, etc). On the other hand, it seems to us a more provocative challenge to solve a sparse problem as is, instead of avoiding it by turning it into a collection of dense subproblems. This way we will be able to study dynamic data structures, pivoting and fill-in issues, and solutions to other problems which can be extrapolated to many other irregular algorithms. Moreover, as the computational kernel of the generic codes reside in three nested loops (as in dense LU factorizations), the problem can be parallelized at the

loop level. That means, that we could face this sparse generic problems from the data-parallel compilers point of view in which we are interested [4,5]. As far as we know, the parallel versions for multifrontal or supernode codes only exploit parallelism at the task level [18,26] which is more sensitive to load balance and scalability problems.

2.1 Loop level parallelism

To exploit loop level parallelism presents some advantages. A loop level parallelized code has the same structure as the sequential code except for these two issues:

- Iteration space for parallel loops is reduced according to the number of processors.
- To solve data dependencies some communication stages may be inserted.

This way the parallel code is a generalization of the sequential one. In these cases, we can try to write this kind of code using some data parallel language (such as HPF-2 [22]) which simplifies the development tool. Moreover, loop level parallelized codes are less sensitive to load balance and scalability problems than task level parallelism (in which tasks may have unbalance computational loads and some processors become idle when there are not enough tasks).

In the right-looking LU factorization we can parallelize the two internal nested loops. Additionally, the sparsity of matrix A give us a degree of freedom to choose the pivots with the aim of achieving more parallelism. This idea comes from the compatible pivots definition presented by Calahan (1973) [6]: two matrix entries a_{ij} and a_{rs} are compatible if a_{is} and a_{rj} are zero. By choosing a set of `m` compatible pivots we will be able to apply a parallel `m`-rank update of the reduced submatrix instead of `m` sequential 1-rank updates.

This way, the factorization process consists of three steps for each outermost loop `k` iteration: to look for a set of `m` compatible pivots in the active matrix at this `k` iteration, $A^{(k)}$; carry out a maximum of `m` row and column permutations (full pivoting) to put selected pivots on the diagonal; and finally perform the `m`-rank update of the reduced submatrix, as shown in figure 1. For the next `k` iteration, the reduced submatrix will be the new active matrix.

2.2 Phases of the sparse linear system solution

Given the complexity presented on the sparse linear system, it is usual to divide the problem into four phases:

1. **Reordering**: Aimed to reduce the computational complexity of the forthcoming stages. For example, if we reorder the original matrix A in a block triangular or block diagonal matrix, we can factorize each irreducible block of the resulting reordered matrix in parallel. We will assume our code will work with already irreducible block matrices.

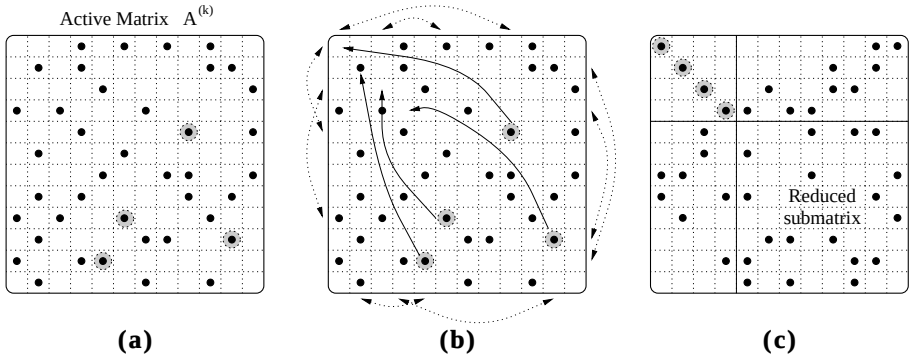


Fig. 1. Steps of the parallel pivots algorithm: (a) Selection of compatible pivots; (b) parallel permutations; (c) `m`-rank update

2. **Analyse**: Aimed to determine the permutation matrices Π and Γ which are going to select which coefficients of matrix A will be placed on the diagonal (i.e., pivots). To fulfill the sparsity preservation constraint we will follow the Markowitz strategy [27]. On the other hand, we will ensure the numerical stability selecting pivots with an absolute value greater than a certain threshold.
3. **Factorize**: The most computationaly expensive stage, in which the factorization $\Pi A \Gamma = LU$ takes place.
4. **Solve**: Comprises forward and backward substitution.

Due to the full pivoting necessities in our algorithm, the analyse and factorize stage are joined into a single analyse-factorize stage. This means that, for each outermost loop iteration `k` of the analyse-factorization process, we first determine the pivot set (analyse), followed by the update process (factorize).

Additionally, analyse-factorize and solve stages will be further divided into sub-stages: due to the fill-in which takes place on the reduced submatrix, it will be worthwhile to switch to a dense code at some `k` iteration (i.e., when the density of the reduced matrix exceeds a threshold input parameter).

This way we will implement a sparse analyse-factorize code followed by a dense one, with a switch routine between them, to change the sparse data structure to a regular two-dimensional dense array. Obviously, the solve stage will be similarly split: sparse forward and dense forward substitution ended by dense backward and sparse backward substitutions. We will use BLAS for the dense subroutines and show in the experimental results that the switch overhead is negligible in comparison to the time saved when avoiding factorizing a quite dense submatrix with a sparse code.

3 Parallel sparse LU Factorization

In this section we present a parallel algorithm for the sparse generic linear system solver, called **SpLU**. The main characteristics of this code are:

- The algorithm is right-looking. The code, written in C, is SPMD and is designed for a distributed memory multiprocessor. There is a portable version thanks to the MPI message passing interface, and a more optimized one for the Cray T3D or T3E using the SHMEM library.
- Data distribution follows the two dimensional sparse cyclic scheme (scatter), mapping the system matrix A onto a $P \times Q$ processor mesh.
- The data structure used to store local matrices is a semi-ordered two-dimensional doubly linked list. Entries are linked in an ordered way by rows and by columns in any order.
- We exploit both inherent parallelism in the updating loops and the parallelism we achieve by selecting `m` compatible pivots. Analyse and factorize stages are joined into a single analyse-factorize one.
- We use a threshold-based heuristic to ensure numerical stability, and the Markowitz criterion (min row in min column) to preserve sparsity. The number of columns in which we search for compatible pivots change according to matrix density in an adaptive way. Explicit full pivoting reduces the unbalancing problems [7, 20].
- When matrix density reaches certain threshold, we switch to a parallel dense LU factorization code.
- The solve stage is also a parallel phase.

In the parallel sparse factorization, the sequential outermost loop, `k`, nests the three mentioned parallel stages: look for a set of `m` compatible pivots, called `PivotSet`; parallel rows and columns permutations; and reduced submatrix update.

Apart from the data structure for matrix A, we will need two additional nonzero count vectors, R and C, of dimension n, to implement the Markowitz strategy. The value $R_i^{(k)}$ represents the number of entries in row `i` of active matrix at iteration `k`, whereas the number of entries in active column `j` for the same iteration is indicated by $C_j^{(k)}$.

Vectors $C^{(k)}$ or $R^{(k)}$ are also used to decide the switch iteration to a dense factorization code. Active submatrix density, `dens`, is calculated at each `k` iteration as `dens`$= (\sum_{i=k}^{n-1} R^{(k)})/(n-k)$. When `dens`$>$`maxdens`, where `maxdens` is an input parameter, and $n-$`k` is big enough to compensate the data structure change, the switch to a dense code takes place.

On the other hand, π and γ will be the permutation vectors needed for the forthcoming solve stage. When $\pi_i^{(k)} =$`r` this means that at iteration `k`, row `r` from the original matrix $A^{(0)}$, is now placed at row `i` in matrix $A^{(k)}$ (and analogously for column permutation vector γ, changing row for column). As program is written in C, the space iteration for all indices mentioned in these paragraphs, `i`, `j` and `k`, are from zero to $n-1$.

3.1 Distribution scheme

Matrix A will be distributed cyclically over a $P \times Q$ processor mesh. The processor will be identified by its coordinates (p, q), with $0 \leq p < P$ and $0 \leq q < Q$.

That is, the matrix entries will be assigned to processors following this equation:

$$A_{ij} \longmapsto PE(i \bmod P,\ j \bmod Q) \quad \forall i,j,\ \ 0 \le i,j < n. \tag{2}$$

This sparse cyclic (also scatter or grid) distribution will lead to even distributions when the probability of a nonzero coefficient is independent of its coordinates. This is true for random pattern matrices and for those which do not present periodicities on entry coordinates[1]. Additionally, scatter distribution will spread clusters of entries on different processors.

Permutation vectors, π and γ will be partially replicated. That is, we will store π_i on processors with coordinates $(i \bmod P,\ *)$, where $*$ represents any integer value between zero and $Q-1$. Similarly, γ_j will be replicated by rows. Vectors R and C will be distributed in the same way as π and γ, respectively.

We will call $\hat{A}$ to the local matrix of dimensions $\hat{m} \times \hat{n}$, where $\hat{m} = \lceil n/P \rceil$, and $\hat{n} = \lceil n/Q \rceil$. Therefore, on processor (p,q), the relationship between A and $\hat{A}$ will be given by the following equation:

$$\hat{A}_{\hat{i}\hat{j}} = A_{\hat{i}P+p,\hat{j}Q+q} \quad \forall \hat{i},\hat{j},\ \ 0 \le \hat{i}P+p, \hat{j}Q+q < n. \tag{3}$$

On the other hand, we should select a proper data structure to store system matrix entries. We will call the combination of these two aspects (data structure + data distribution) the distribution scheme. The distribution scheme comprises all the information we need to localize any matrix entry on the local memories of the multiprocessor system. Maintaining the same data structure of the sequential code in the parallel one, leads to the parallel code being a generalization of the sequential one.

To chose the data structure we should take into account that in our algorithm we implement a full-pivoting stage. This reduces the sets of possible data structures to those with permit access by rows and columns. We have selected a two dimensional doubly linked list (LLRCS), which links entries of the same row in an orderly way, and entries of the same column in any order. We will also use two pointer vectors to point to the first entry on each row, `rows`, and column, `cols`. Each entry on the list stores the coefficient value, local indices and pointers to the previous and next entries in the same row and column. The C data structure declaration follows:

```
struct matrix  {
                 struct entry *rows[m̂], *cols[n̂];
               };
struct entry   {
                 int Row,Col;
                 double Val;
                 struct entry *previ, *prevj, *nexti, *nextj;
               };
```

[1] Only periodicities of non prime period with the number of processors can lead to unbalanced distributions.

Due to the doubly linked structure, we simplify the delete operation of entries, needed in explicit permutations. However, the wastage of memory due to storage of pointers is evident. We show in figure 2 an example of this distribution scheme for a sparse matrix with $n = 8$ and $\alpha = 24$ mapped onto a 2×2 processor mesh.

3.2 Right-looking parallel algorithm

A more detailed description of the algorithm follows:

```
1.    π = γ = identity;
2.    Initialize R and C; Compute dens
3.    k = 0;
4.    while ( (k< n)   &&   (dens<maxdens))
        {
5.          Search for parallel pivots: PivotSet = (i_r, j_r) : 0 ≤ r < m;
6.          Parallel row permutations, π_{i_r} and R_{i_r} : 0 ≤ r <m;
7.          Parallel column permutations, γ_{j_r} and C_{j_r} : 0 ≤ r <m;
8.          Update A^(k);
9.          Update nonzero count vectors R and C;
10.         k = k + m;
        }
11.   Dense submatrix factorization;
12.   Solve (forward and backward substitution)
```

In this code, we implicitly assume that each processor, (p, q), does the computations over its local data. At the end of the in-place algorithm, matrix $A^{(n-1)}$ stores coefficients of matrix L and U, vectors $\pi^{(n-1)}$ and $\gamma^{(n-1)}$ contain its final values, and vector x will be the solution to equation $Ax = b$.

Parallel pivot search. It is well known that for a selected pivot $A_{ij}^{(k)}$, the maximum number of new entries that can be created may be $M_{ij}^{(k)} = (R_i^{(k)} - 1)(C_j^{(k)} - 1)$, where $M_{ij}^{(k)}$ is the Markowitz count at iteration `k` for the mentioned pivot.

Therefore, to preserve sparsity, selected pivots would have a minimum Markowitz count and a maximum absolute value to ensure stability. On the other hand this search is prohibitive, as one needs to visit the whole active matrix. To keep the search for compatible pivots effective and simple, we will only search in the `ncol` columns with the least $C_j^{(k)}$ in each column of the processor mesh. In these columns we will select candidates to be pivots to those with minimum $R_i^{(k)}$ (min row in min column technique) and complying wiht the following equation to ensure numerical stability:

$$|A_{ij}^{(k)}| \geq u \cdot \max_l |A_{lj}^{(k)}| \tag{4}$$

Input parameter `u`: $0 < \mathtt{u} \leq 1$, will prioritize stability when $\mathtt{u} \to 1$, or sparsity when $\mathtt{u} \to 0$.

From the candidate sets of pivots, we will mark incompatible ones, and pick up compatibles to build `PivotSet` with `m` parallel pivots. To control sparsity, an

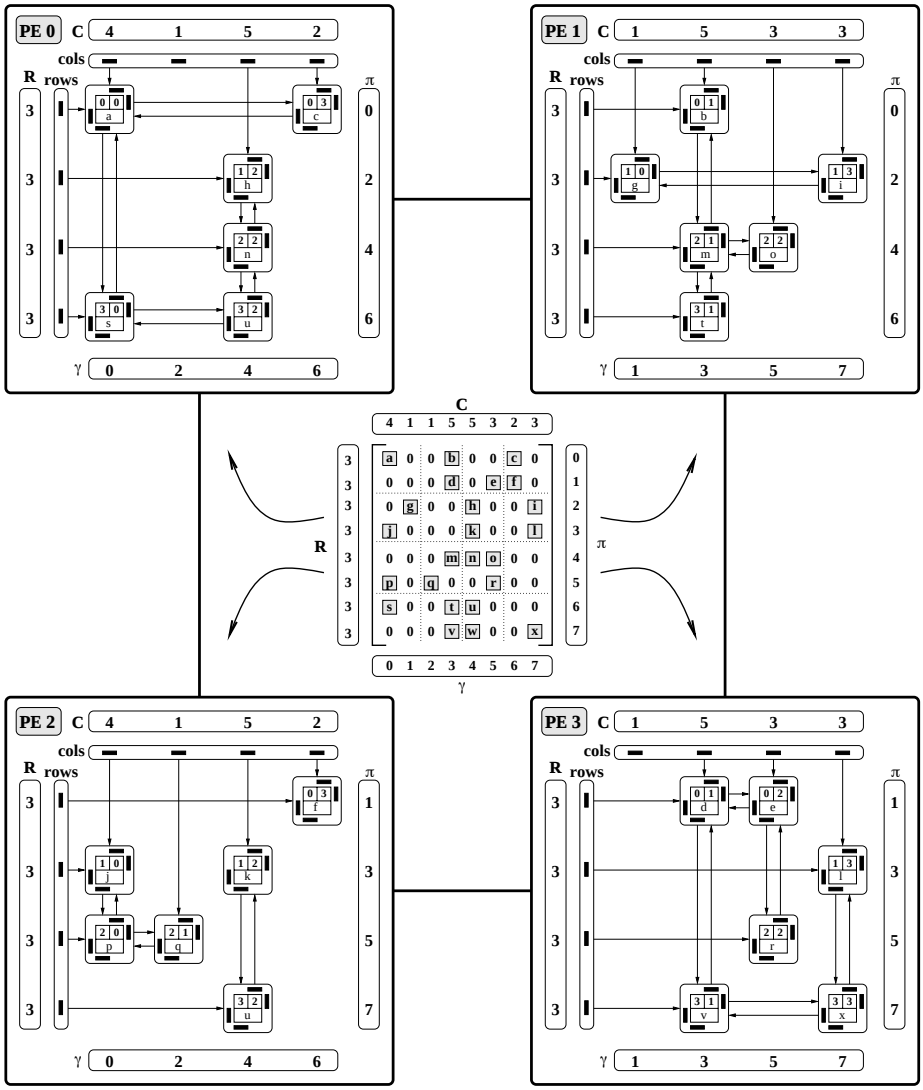


Fig. 2. Distribution scheme LLRCS-Scatter

additional input parameter `a` will be used to reject pivots with an unacceptable Markowitz count. In particular, candidates with $M_{ij} > a \cdot M_{i_0,j_0}$ will be rejected, where M_{i_0,j_0} is the minimum Markowitz count of the candidates.

Another issue is related with the fill-in that takes place during factorization. When active matrix density increases, looking for parallel pivots tends to be unproductive. For this reason, initially, `ncol` contains the number of columns per processor in which the search for candidates is performed, but this `ncol` parameter will change dynamically during factorization, automatically adapting to the density.

Parallel permutations. This stage is in charge of moving selected compatible pivots to the diagonal, building the `m`×`m` diagonal block, as shown in figure 1 (b). Permutation can be done explicitly (i.e., moving row and column entries [29, 7, 1]) or implicitly (using permutation vectors π and γ to indirectly access the matrix [28, 30]). In parallel implementations, it has been observed [7, 20] that explicit pivoting leads to better load balance. Performance increases due to this fact, and compensates communication overheads due to row and column movement.

Parallel update. The reduced submatrix update process comprises two different steps: first, we divide, in parallel, subcolumns under the diagonal block following this equation:

$$A_{ij}^{(k)} = A_{ij}^{(k)} / A_{jj}^{(k)} \quad \forall i,j : (k+m \leq i < n) \wedge (k \leq j < k+m) \wedge (A_{ij}^{(k)} \neq 0) \quad (5)$$

In a second step, we properly update the reduced submatrix with:

$$A_{ij}^{(k)} = A_{ij}^{(k)} - A_{il}^{(k)} A_{lj}^{(k)} \qquad \forall \ i,j,l : \begin{cases} (k+m \leq i,j < n) \\ (k \leq l < k+m) \\ (A_{il}^{(k)} \neq 0) \wedge (A_{lj}^{(k)} \neq 0) \end{cases} \quad (6)$$

To carry out equation 5 in parallel, a previous communication stage (broadcast by mesh columns) is necessary to make pivots visible to processors who are going to divide its local columns. In the same broadcast we will pack the block of rows needed in the forthcoming update (figure 3 (a)).

In a similar way, to complete equation 6, the broadcast (by mesh rows) of a block of previously divided columns is carried out (figure 3 (b)).

Parallel switch and dense factorization. When we reach the threshold submatrix density (`maxdens`) and this submatrix is big enough, we proceed to switch to a dense factorization code. Clearly, we initially need to change the data structure for the final active submatrix $A^{(k)}$, from the linked list to a bidimensional regular array, AD, with dimensions $(nd \times nd)$ where $nd = n - k$. This operation is fully parallel (without communications) and, when finished, the AD matrix remains automatically distributed in a cyclic way.

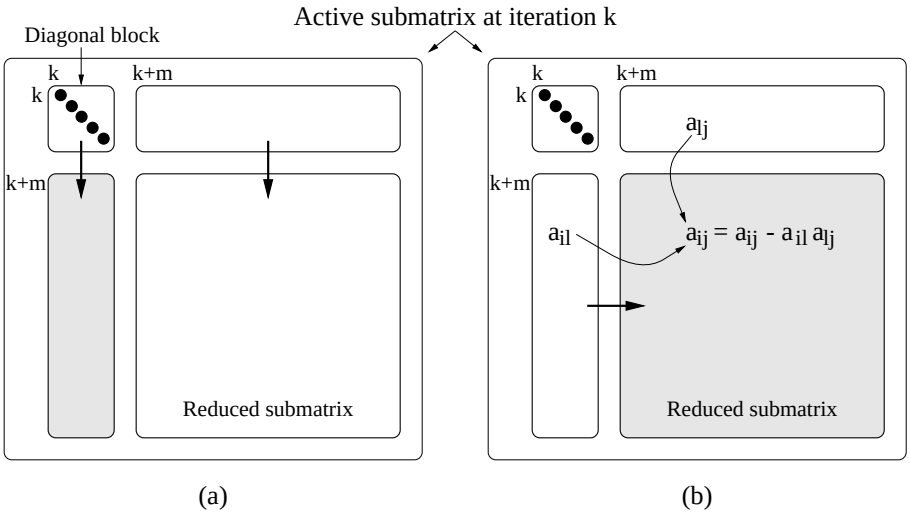


Fig. 3. Update steps. (a) Broadcast by mesh columns and columns division by pivots (grey block). (b) Broadcast by mesh rows and reduced submatrix update (grey submatrix)

Subsequently, we are ready to call to a parallel dense factorization routine. The one we have designed provides for row partial pivoting to ensure numerical stability and uses BLAS-2 as much as possible.

Parallel solve stage. Even when the stages which solve $LUx = b$ is two or three orders of magnitude faster than the factorization stages, it is sometimes worthwhile to parallelize it. For example, optimization problems like simplex or iterative solvers such as ILU preconditioned Conjugate Gradient, needs to solve many forward and/or backward substitutions.

As the analyse-factorize stage switches at a certain iteration to a dense code, the solve stage switches at the same one. However, for the sake of shortness we will only focus on sparse forward substitution, as the dense solve stage is sufficiently well known and sparse backward substitution is similar to the forward one.

The forward substitution algorithm, $Ly = b$, can be outlined by the following equation:

$$y_i = \frac{1}{l_{ii}} \left[b_i - \sum_{j=1}^{i-1} l_{ij} y_j \right] \tag{7}$$

As we can see, there is a flow dependence for index i, as y_i may only be computed when coefficients $y_1, \cdots, y_{i-1}$ are known, if $l_{ij} \neq 0, \quad 1 \leq j < i$. However, if the L matrix is sparse, many of the l_{ij} coefficients are zero, and many unknowns may be computed in parallel. In fact, coefficients $y_i, \cdots, y_{i+m}$ are independents if pivots $A_{i,i}, \cdots, A_{i+m,i+m}$ are compatible and belong to the same diagonal

block, as shown in figure 4. In terms of loop level parallel programming, we can carry out the execution of loops with indices i and j in parallel, while maintaining the bidimensional L matrix distribution.

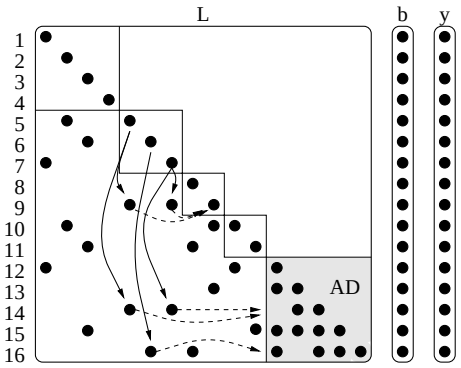


Fig. 4. Data flow in the forward substitution

4 Experimental results

This section aims to analyse the sequential and parallel behavior of our SpLU algorithm, when changing the input matrix characteristics and some of the input parameters. We have selected some heterogeneous unsymmetric sparse matrices from Harwell-Boeing [16] and the University of Florida [12] matrix collection, presented in table 1. The algorithm has been implemented in C. As a message-passing interface, SHMEM routines have been used as they are supported by the CRAY T3E supercomputer. The sequential version of the program is obtained by simplifying the parallel code, removing all redundant or never executed sentences when $P = 1$ and $Q = 1$.

4.1 Fill-in and stability

Two input parameters can be tuned to control stability and fill-in: `u` and `a`.

A study of `u` parameter incidence is presented in table 2. We can see the variation of the average size of diagonal blocks $\overline{m}$, the number of sparse LU iterations, fill-in and factorization errors, for different `u` values. For the sake of brevity, we present these results for the LNS3937 matrix. Other matrices show the same behavior, but the LNS3937 is the worst conditioned and the `u` effect can be better appreciated. In the experiment we fixed `a`$= 4$ and `ncol`$= 16$.

In table 2 we can see that the smaller `u` is, the bigger is the average size of `PivotSet`, allowing us to exploit more parallelism. The same effect can be appreciated in the next row, due to increasing $\overline{m}$, so decreasing the number of

Matrix	Origin	n	α	Density (ρ)
STEAM2	Oil recovery	600	13760	3.82%
JPWH991	Circuit physics modeling	991	6027	0.61%
SHERMAN1	Oil reservoir modeling	1000	3750	0.37%
SHERMAN2	Oil reservoir modeling	1080	23094	1.98%
EX10	2D isothermal seepage flow	2410	54840	0.94%
ORANI678	Economic modeling	2529	90158	1.41%
EX10HS	2D isothermal seepage flow	2548	57308	0.88%
CAVITY10	Driven cavity problem	2597	76367	1.13%
WANG1	Discretized electron continuity	2903	19093	0.22%
WANG2	Discretized electron continuity	2903	19093	0.22%
UTM3060	Uedge test matrix	3060	42211	0.45%
GARON1	2D FEM, Navier-Stokes, CFD	3175	88927	0.88%
EX14	2D isothermal seepage flow	3251	66775	0.63%
SHERMAN5	Oil reservoir modeling	3312	20793	0.19%
LNS3937	Compressible fluid flow	3937	25407	0.16%
LHR04C	Light hydrocarbon recovery	4101	82682	0.49%
CAVITY16	Driven cavity problem	4562	138187	0.66%

Table 1. Test matrices

Values for u	0.9	0.5	0.1	0.05	0.01	0.001
$\overline{m}$	5.84	6.17	6.85	6.90	7.28	8.48
Sparse iterations	493	475	429	420	410	349
Fill-in	283772	250748	241163	222655	216017	216196
Error	2.32E-2	1.05E-2	9.56E-3	2.27E-2	2.57E-2	4.02E-1

Table 2. The influence of the u parameter on LNS3937

outermost loop iteration, thus reducing both sequential and parallel execution time. Additionally, fill-in is reduced when u diminished, as there are more candidates to choose from with a smaller Markowitz count. On the other hand, the factorization error increases when reducing u, which leads to the necessity of choosing a trade-off value. Furthermore, we have observed that the more density is achieved on factors L and U, the bigger is the factorization error, as the number of floating point operations increases. For this reason, for u$= 0.1$ we get the minimum error. These experiments corroborate that the trade-off u≈ 0.1 [15, 14] leads to good results in many situations. In any case, the best selection of u is problem dependent, so we may need to test some u values to find the best one.

The algorithm behavior as a function of the a input parameter is shown in table 3 for the same matrix LNS3937, with u$= 0.1$ and ncol$= 16$.

The greater a is, the bigger will be the average size of the compatible pivots set, and the less the number of sparse iterations. At the same time, if we do not limit the Markowitz count, we can select pivots which will bring about more fill-

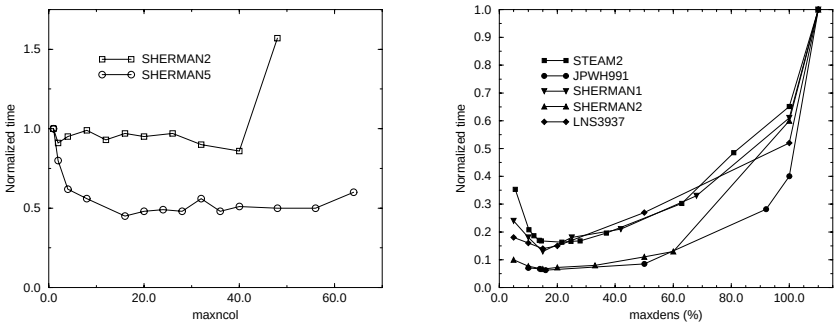
Values for a	10	8	6	4	2	1
$\overline{m}$	8.38	7.91	7.69	6.85	4.27	1.84
Sparse Iterations	361	377	384	429	687	1611
Fill-in	254176	251622	245374	241163	238923	220624

Table 3. The influence of the a parameter on LNS3937

in. To keep a high $\overline{m}$ without provoking an excessive fill-in, the trade-off value for `a` will be around 4 (also selected by other authors [9,25]).

As `ncol` value is dynamically adjusted during program execution, the initial value is not specially significant. In any case, we found an appropriate initial value `ncol`=16.

Searching for a parallel pivots set is worthwhile even in the sequential code, as we can see in figure 5 (a), where we study the execution time versus the value of `maxncol`. In this experiment we have fixed `ncol=maxncol`, cancelling the adaptive function to update `ncol`. For the more sparse matrix in our set (SHERMAN5) and the second most dense one (SHERMAN2), we present in this figure the execution time normalized by the time when `maxncol=1`.



(a) (b)

Fig. 5. Fixed `ncol` (a) and `maxdens` (b) influence on execution time

When factorizing sparse matrices, we see that it is interesting to search for big sets of compatible pivots. Regarding SHERMAN5, the sequential time when fixing `ncol`=16 is over 45% less than the one we get when `ncol`=1. The variable $\overline{m}$ reaches the value 26.7 with `ncol`=36, although execution time is worse due to wastage of time looking for compatible pivots. However, when factorizing more dense matrices, such as SHERMAN2, setting `ncol` to a large value is unproductive. For example, when `ncol`=48, we search for a large set of candidates which later turned out to be incompatible due to high matrix density.

Finally, we have also studied execution time dependence with the threshold `maxdens` which decides the switch to a dense code. In figure 5 (b) we show the relation between `maxdens` and execution time for some matrices. Execution times are normalized by the worst time (when there is no switch to a dense factorization code, identified in the figure by `maxdens`=110). We see that the switch leads to a significant execution time saving. Minimum execution times are obtained when `maxdens`$\approx$15%.

4.2 Comparison with the MA48 routine

Before studying the parallel algorithm performance, it is important to check that the sequential version is good enough. The generic sparse system solver more widely used is the MA48 routine [17]. The good performance of this routine is mainly due to its left-looking organization which leads to low data traffic with memory and a subsequent good exploitation of the cache. However, this left-looking organization leads to low efficiencies in a loop level parallelized version of the MA48. On the other hand, SpLU exhibits more loop level parallelism due to the *right-looking* and parallel pivots organization, but should be comparable to MA48 performances and execution time if we want to get a competitive generic code.

In table 4 we present a comparison for the more significant characteristics of both algorithms: execution time, factorization error and fill-in. Common parameters are set equally: `u`=0.1 and `maxdens`=15%.

We can see how the execution time ratio (MA48 time divided by SpLU time) is greater than one, for five matrices (ratios in boldface). In these cases SpLU is faster than MA48, reaching a 4.26 factor for the EX10HS matrix. However, for the remaining 12 matrices MA48 is faster than SpLU, although the ratio do not decrease below 0.5, except for WANG1, WANG2 and LHR04C matrices. For the latter, LHR04C, factorization time in SpLU is clearly the worst, but this is in exchange for a numerical error around 20 times better.

For 12 of the 17 matrices the factorization error in SpLU is better than in the MA48 routine. For the remaining 5 matrices, there is never more than an order of magnitude of difference. With regard to fill-in, L and U matrices are sparser on 9 occasions if they are computed by SpLU code.

In spite of the high optimization of the MA48 code, we believe that it can be improved by the SpLU in some cases due to the analyse stage. Even when the MA48 analyse stage is also based on Markowitz and threshold strategies, the fact that this analyse stage takes place before factorizing has its own drawbacks: permutation vectors are selected in advance, but during the factorize stage, the numerical partial pivoting is also allowed and this may undo the analyse decisions to some extent.

SpLU shows a joined analyse-factorize stage where for each iteration a proper set of compatible pivots are selected over the candidates in the active matrix. In many cases this enables a better pivot selection during factorization, yielding better numerical precision. In exchange, the analyse fragment of code is more

	Time		Error	Fill-in
Matrix	**SpLU–MA48 ration**		**SpLU–MA48**	**SpLU–MA48**
STEAM2	1.10–.61	(0.55)	**.29E-13**–.13E-11	**79552**–110466
JPWH991	1.05–.88	(0.84)	**.18E-14**–.82E-13	**89810**–101892
SHERMAN1	.37–.19	(0.51)	.22E-12–.16E-12	43320–43171
SHERMAN2	6.87–16.7	**(2.43)**	**.69E-6**–.15E-5	**325706**–656307
EX10	7.49–24.51	**(3.27)**	**.23E-6**–.31E-6	**283378**–296270
ORANI678	9.23–6.48	(0.70)	.15E-12–.74E-13	**406568**–439280
EX10HS	10.24–43.71	**(4.26)**	**.72E-7**–.20E-6	**321031**–336832
CAVITY10	51.78–25.94	(0.50)	**.16E-9**–.36E-9	1139121–1087769
WANG1	46.99–21.18	(0.45)	.26E-12–.97E-13	1124807–808989
WANG2	49.82–21.02	(0.42)	**.47E-13**–.52E-13	1178085–808989
UTM3060	42.30–26.13	(0.62)	**.16E-8**–.58E-8	**1066896**–1073933
GARON1	69.73–35.07	(0.50)	**.17E-8**–.21E-8	1431657–1257874
EX14	131.75–206.63	**(1.56)**	**.19E+1**–.93E+1	**2293851**–2658661
SHERMAN5	8.69–11.02	**(1.26)**	**.17E-12**–.59E-12	**363186**–519855
LNS3937	34.1–25.8	(0.75)	.95E-2–.13E-2	1078221–1002494
LHR04C	101.43–14.05	(0.13)	**.89E-5**–.16E-3	1988258–870784
CAVITY16	193.46–109.86	(0.56)	.85E-9–.49E-9	2683852–2581086

Table 4. SpLU and MA48 comparison

expensive than the corresponding one in the MA48, due to it searching for a single pivot instead of many which are mutually compatible.

4.3 Parallel performance

In this section we will compare the parallel algorithm execution time over a $P \times Q$ processor mesh with the sequential version, executed over a single Alpha processor.

To make times comparable for both versions, input parameters will be equally fixed. As we saw in subsection 4.1 it seems appropriate to set `u`=0.1 and `a`=4. As for the initial local `ncol`, it will be set to $16/Q$, to make the initial maximum number of compatible pivots independent of the mesh size.

Parallel version exhibits the same fill-in and factorization error as sequential version, as `u`, `a` and `maxdens`, do not affect the parallel version in a different way to the sequential one.

Table 5 presents the speed-up we get when factorizing the 14 biggest matrices in our set. The last three columns in this table show dimension, n, initial density, ρ_0, and the final one, ρ_n. Figure 6 shows speed-up and efficiency when factorizing the 9 computationally more expensive matrices for mesh sizes.

	Speed-up				Density		
Matriz	2	4	8	16	n	ρ_0	ρ_n
SHERMAN2	1.85	3.62	5.98	9.82	1080	1.98%	27.92%
EX10	1.74	2.99	4.25	4.96	2410	0.94%	4.87%
ORANI678	1.77	3.02	4.96	6.67	2529	1.41%	6.35%
EX10HS	1.74	3.65	5.39	5.63	2548	0.88%	4.94%
CAVITY10	1.94	3.72	5.59	8.77	2597	1.13%	16.88%
WANG1	2.16	3.76	7.01	10.44	2903	0.22%	13.94%
WANG2	2.06	4.15	6.60	12.45	2903	0.22%	13.97%
UTM3060	1.88	3.41	5.87	10.07	3060	0.45%	11.39%
GARON1	2.32	3.76	7.18	12.02	3175	0.88%	14.20%
EX14	2.18	4.04	7.22	13.17	3251	0.63%	21.70%
SHERMAN5	1.63	3.00	4.28	5.66	3312	0.19%	3.31%
LNS3937	1.93	3.69	6.33	11.01	3937	0.16%	6.95%
LHR04C	2.02	3.93	7.04	11.79	4101	0.49%	11.82%
CAVITY16	1.99	3.85	7.48	14.11	4562	0.66%	12.89%

Table 5. Speed-up for different mesh sizes

We see that speed-up monotonically increases with the number of processors. When changing from 8 to 16 processors, EX10 and EX10HS exibit a less notable increment of speed-up due to the low computational load presented by these matrices. In these cases, communications dominate local computations and messages comprise a small number of data, so latency prevails over communication bandwidth. We should take into account the high ratio between the power of Alpha 21164-300Mhz and the 500Mbytes/s peak bandwidth and 0.5 to 2 μs latency for the `shmem-put` communication routine.

It is noteworthy that, contrary to dense LU factorization, the computational load depends not only on the matrix dimension but also on the initial or (even more) final density. This way, EX10, EX10HS and SHERMAN5 are the only matrices with ρ_n <5% and with lowest speed-up on 16 processors.

Therefore, better speed-ups on 16 processors are reached for matrices with high `n` and high ρ_n. The best speed-up is presented for CAVITY16. For some of the bigger matrices, such as WANG1, WANG2 and EX14, parallel factorization exhibits super-lineal speed-up even for four processors.

Regarding the solve stage, we did not reach speed-up when using more than 4 processors. The reason is the low computational load presented by these matrices for this step. This can be seen in table 6, where the time expended on the solve stage never exceeded 0.3 seconds for any of the 17 matrices (the more expensive is CAVITY16 expending 0.25 seconds on this stage). Additionally, our solve stage comprises a sparse and a dense part, and the last one can only exploit unidimensional parallelism. Therefore, redistribution for the dense submatrix, AD, implies some overhead.

In any case, the parallel solve stage is worthwhile as it permits us to solve matrices which do not fit on an single processor. Moreover, the parallel solve

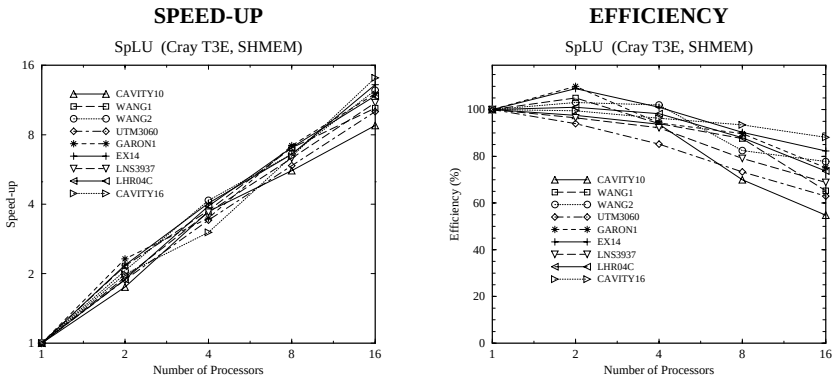


Fig. 6. SpLU speed-up and efficiency

Matrix	JPWH991	SHERMAN1	SHERMAN2	EX10
Time	1.72E-2	2.77E-2	2.90E-2	6.22E-2
Matrix	ORANI678	EX10HS	CAVITY10	WANG1
Time	4.43E-2	6.22E-2	1.09E-2	9.91E-2
Matrix	WANG2	UTM3060	GARON1	EX14
Time	1.03E-1	1.04E-1	1.40E-1	1.79E-1
Matrix	SHERMAN5	LNS3937	LHR04C	CAVITY16
Time	5.37E-2	1.12E-1	1.51E-1	2.54E-1

Table 6. Solve sequential execution time

stage avoids collecting the whole matrix in one processor. As future work we will try to reduce this communication overhead by using a block cyclic distribution.

5 Related work

The computational numeric solution of large sparse systems is a very active research area. However, due to its complexity, there are many open problems. We will try to sumarize recent works, organizing them according to the level of parallelism.

Regarding task level parallelism, Gallivan, Marsolf and Wijshoff (1996) [19] carry out a matrix reordering to a bordered block triangular form. In the same direction, Zlatev et al. (1995) [32] have developed the tool PARASPAR, to solve the linear system on shared memory multiprocessors, by reordering the system matrix. This tool allows for rectangular diagonal blocks, exploiting some advantages, but the lack of singularity or bad conditioning for diagonal blocks is not guaranteed. Therefore, to ensure better stability some techniques were implemented in LORA-P[5] code, which is in charge of the reordering stage for the Y12M3 program. This code achieves speed-ups between 3.0 and 4.7 for some

of the biggest Harwell-Boeing matrices on 8 processors of the shared memory Alliant FX/80 [31].

Apart from reordering, another source of task parallelism is multifrontal or supernode methods. As these methods were traditionally applied to symmetric matrices, parallel cholesky multifrontal codes were quickly developed. For shared memory machines, the works from Johnson and Davis (1992) [23] and Amestoy and Duff (1993) [3] are relevant. A cholesky data parallel version is presented by Conroy, Kratzer and Lucas (1994) [8]. But probably, Gupta, Karypis and Kumar (1995) [21] where the authors who better performance achieved in the sparse cholesky. To reach these results over a distributed memory platform, special care of the elimination tree balance and a regular distribution for dense matrices was necessary. This way, authors exploit loop level parallelism when task level one starts to be exhausted when approaching the root of the tree.

A significant more difficult problem appears when matrices are not symmetric. Here, the supernode tree is the tool to exploit task level parallelism. A parallel version of the SuperLU is presented by Li et al. [26], achieving on 8 processors shared memory machines, and for 21 unsymmetric sparse matrices, the following average speed-up: 3.59 in the SGI Power Challenge, 4.01 in the DEC AlphaServer 8400, 3.85 in the Cray C90 and 4.29 in the Cray J90. Better results can be achieved on distributed memory machines as recently shown by Fu, Jiao and Yang (1998) [18]. However, they have parallelized the factorize stage only, which can be executed in parallel thanks to fill-in overestimation carried out on the analyse stage. As the authors state, the analyse state needs some improvement to avoid the cases in which static symbolic factorization leads to excessive overestimation.

On the other hand, to exploit loop level parallelism it is highly recommended to also profit from matrix sparsity, by looking for a set of parallel pivots. For shared memory machines the work from the following authors is significant: Alaghband (1995) [1,2], Davis and Yew (1990) [9, 10] and Zlatev et al. (1995) [32]. Alaghband uses an $n \times n$ table to represent compatibility between diagonal elements. The author only presents results for two matrices of $n = 144$ and $n = 505$, both with the same density $\rho \approx 2\%$, achieving 33% and 53% efficiency on the 8 processors in the *Sequent Symmetry*.

Davis's D2 algorithm performs a parallel search for candidates to be pivot. Compatible candidates are added to the parallel pivot set by critical sections. Experiments on the Alliant FX/8 reports an average efficiency of less than 50% in 8 processors.

Zlatev's Y12M2 is a parallel pivot version of a previous Y12M routine. For 27 Harwell-Boeing matrices, Y12M2 achieves speed-ups between 2.3 and 5.0 on the FX/80 with 8 processors. We conclude that actual system solver implementations for shared memory multiprocessors exceeds 50% efficiency in 8 processors with great difficulty.

Better results are reached for distributed memory machines as presented by Stappen, Bisseling and van der Vorst (1993) [29] for a square Transputer mesh,

and by Koster and Bisseling (1994) [24, 25] also for a transputer mesh. In these codes, they do not present a switch to a dense code stage nor parallel solve stage.

6 Conclusions

This work presents a complete tool, SpLU, to solve large nonsymmetric linear systems on distributed memory multiprocessors. SpLU code comprises analyse-factorize and solve stages. Both of them were split into sparse and dense steps to avoid applying sparse techniques when fill-in turns the problem into a dense one. The algorithm follows a generic approach exploiting loop-level parallelism and takes advantage of matrix sparsity due to parallel pivoting selection. We have compared sequential SpLU with another generic sequential nonsymmetric sparse solver: the high optimized MA48 routine. Our SpLU code leads in many cases to fewer numerical errors and fill-ins than MA48 does. On the other hand, MA48 is usually slightly faster than SpLU. However, we found the loop-level parallelization of MA48 costly due to its left-looking approach, and we have not seen any published parallel version for this routine. Therefore, as SpLU exhibits a high degree of parallelism, speed-up computed as MA48 sequential time divided by parallel SpLU execution time is still competitive. Additionally, SpLU may also be faster than MA48 for some matrices.

As far as we know, there is no published work for the whole parallel nonsymmetric sparse system solver on current distributed memory machines, including sparse analyse-factorize stage, switch to dense LU factorization stage, and forward and backward substitution.

On the other hand, SpLU could be improved mainly in two areas. The first is further reducing communication overheads by using a block cyclic distribution instead of a cyclic one. The second one is directed at reducing data movements and to make entries insertion easier using an unordered linked list both by rows and columns. This two points joined with better care of cache exploiting would result in higher performances. We also want to present some comparisons with UMFPACK and SuperLU codes, and test the parallel algorithm with a larger number of processors, for the final version of this paper.

Acknowledgements

We gratefully thank Iain Duff and all the members in the parallel algorithm team at CERFACS, Toulouse (France), for their kind help and collaboration. We also thank the CIEMAT (Centro de Investigaciones Energéticas, Medioambientales y Tecnológicas), Spain, for giving us access to the Cray T3E multiprocessor.

References

1. G. Alaghband. Parallel pivoting combined with parallel reduction and fill-in control. *Parallel Comput.*, 11:201–221, 1989.

2. G. Alaghband. Parallel sparse matrix solution and performance. *Parallel Computing*, 21(9):1407–1430, 1995.
3. P.R. Amestoy and I.S. Duff. Memory management issues in sparse multifrontal methods on multiprocessors. *Int. J. Supercomputer Applics.*, 7:64–82, 1993.
4. R. Asenjo, R. Doallo, J.P. Tourino, O.Plata, and E.L. Zapata. Parallel sparse computations involving pivoting and fill-in. Submitted to *IEEE Transaction on Parallel and Distributed Systems*.
5. R. Asenjo, G.P. Trabado, M. Ujaldón, and E.L. Zapata. Compilation issues for irregular problems. In *Workshop on Parallel Programming Environments for High-Performance Computing*, pages 187–199, L'Alpe d'Huez, France, April 1996.
6. D.A. Calahan. Parallel solution of sparse simultaneous linear squations. In *11th Annual Allerton Conference on Circuits and System Theory*, pages 729–735. University of Illinois, 1973.
7. E. Chu and A. George. Gaussian elimination with partial pivoting and load balancing on a multiprocessor. *Parallel Comput.*, 5:65–74, 1987.
8. J.M. Conroy, S.G. Kratzer, and R.F. Lucas. Data-parallel sparse matrix factorization. In J.G. Lewis, editor, *Proceedings 5th SIAM Conference on Linear Algebra*, pages 377–381, Philadelphia, 1994. SIAM Press.
9. T. A. Davis. *A parallel algorithm for sparse unsymmetric LU factorization*. PhD thesis, Center for Supercomputing Research and Development, Univ. of Illinois, Urbana, IL, September 1989.
10. T. A. Davis and P. C. Yew. A nondeterministic parallel algorithm for general unsymmetric sparse LU factorization. *SIAM J. Matrix Anal. Appl.*, 11:383–402, 1990.
11. T.A. Davis and Iain S. Duff. An unsymmetric-pattern multifrontal method for sparse LU factorization. Technical Report RAL-TR-93-036, Rutherford Appleton Laboratory, Chilton Didcot Oxon OX11 0QX, 1993. To appear in SIAM J. Matrix Analysis and Applications.
12. Tim Davis. Sparse matrix collection. At URL http://www.cise.ufl.edu/ davis/.
13. J.W. Demmel, S.C. Eisenstat, J.R. Gilbert, X.S. Li, and J.W.H. Liu. A supernodal approach to sparse partial pivoting. Technical Report UCB//CSD-95-883, Computer Science Division, U.C. Berkeley, Berkeley, California, July 1995.
14. J.J. Dongarra, I.S. Duff, D.C. Sorensen, and H.A. van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers*. Society for Industrial and Applied Mathematics, 1991.
15. I.S. Duff, A.M. Erisman, and J.K. Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, Oxford, U.K., 1986.
16. I.S. Duff, R.G. Grimes, and J.G. Lewis. User's guide for the Harwell-Boeing sparse matrix collection (Release I). Technical report, CERFACS, Toulouse, France, 1992.
17. I.S. Duff and J.K. Reid. The design of MA48: A code for the direct solution of sparse unsymmetric linear systems of equations. *ACM Trans. Math. Softw.*, 22(2):187–226, June 1996.
18. C. Fu, X. Jiao, and T. Yang. Efficient sparse lu factorization with partial pivoting on distributed memory architectures. *IEEE Transaction on Parallel and Distributed Systems*, 9(2):109–125, February 1998.
19. K. Gallivan, B. Marsolf, and H.A.G. Wijshoff. Solving large nonsymmetric sparse linear systems using MCSPARSE. *Parallel Computing*, 22(10):1291–1333, 1996.
20. G. A. Geist and C. H. Romine. LU factorization algorithm on distributed-memory multiprocessor architecture. *SIAM J. Sci. Statist. Comput.*, 9:639–649, 1989.

21. 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. Available at URL: http://www.cs.umn.edu/~kumar.
22. High Performance Fortran Forum. *High Performance Language Specification, Ver. 2.0*, 1996.
23. T. Johnson and T.A. Davis. Parallel buddy memory management. *Parallel Processing Letters*, 2(4):391–398, 1992.
24. J. Koster. *Parallel solution of sparse systems of linear equations on a mesh network of transputers.* Institute for Continuing Education, Eindhoven University of Technology, Eindoven, The Netherlands, July 1993. Final Report.
25. J. Koster and R.H. Bisseling. An improved algorithm for parallel sparse LU decomposition on a distributed memory multiprocessor. In J.G. Lewis, editor, *Fifth SIAM Conference on Applied Linear Algebra*, pages 397–401, 1994.
26. X. Li. *Sparse Gaussian Elimination on High Performance Computers.* PhD thesis, CS, UC Berkeley, 1996.
27. H. M. Markowitz. The elimination form of the inverse and its application to linear programming. *Management Sci.*, 3:255–269, 1957.
28. A. Skjellum. *Concurrent dynamic simulation: Multicomputer algorithm research applied to ordinary differential-algebraic process systems in chemical engineering.* PhD thesis, California Institute of Technology, Pasadena, CA, USA, May. 1990.
29. A. F. van der Stappen, R. H. Bisseling, and J. G. G. van de Vorst. Parallel sparse LU decomposition on a mesh network of transputers. *SIAM J. Matrix Anal. Appl.*, 14(3):853–879, July 1993.
30. E.F. van der Velde. Experiments with multicomputer LU-decomposition. *Concurrency: Practice and Experience*, 2:1–26, 1990.
31. A.C.N. van Duin, P.C. Hansen, T. Ostromsky, H.A.G. Wijshoff, and Z. Zlatev. Improving the numerical stability and the performance of a parallel sparse solver. *Computers Math. Applic.*, 30:81–96, 1995.
32. Z. Zlatev, J. Waśniewski, P.C. Hansen, and T. Ostromsky. PARASPAR: a package for the solution of large linear algebraic equations on parallel computers with shared memory. Technical Report 95-10, Tech. Univ. Denmark, Lyngby, 1995.
33. Z. Zlatev, J. Waśniewski, and K. Schaumburg. Y12M—Solution of Large and Sparse Systems of Linear Algebraic Equations. In *Lecture Notes in Computer Science 121*, pages 61–77, Berlin, 1981. Springer-Verlang.