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 L. 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 LU 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. Furthermore, this problem presents a good case study and is a representative computational code for many other irregular problems.

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. However, if the matrix suffers from fill-in it will be worthwhile to combine sparse and dense approaches. That way, our parallel nonsymmetric 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 uses BLAS as much as possible.

For the sake of brevity, a sparse LU factorization problem survey and a more detailed description of our algorithm are presented in [2]. In this paper we briefly comment the proposed algorithm, but the main section focus on the experimental results, validation of the sequential code, and presentation of the parallel performance. Finally, the related work and conclusions section close the paper.

* 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)

2 Algorithm outline and input parameters

Summarizing, the main characteristics of the proposed code, called **SpLU**, 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 $(n \times n)$ 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 [3, 13].
- When matrix density reaches certain threshold, we switch to a parallel dense LU factorization code.
- The solve stage is also a parallel phase, with sparse and dense sub-stages for the forward and backward substitution.

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

Apart from the data structure for matrix A, we will need two nonzero count vectors, R and C to implement the Markowitz strategy. The value $R_i^{(k)}$ $(C_j^{(k)})$ represents the number of entries in row `i` (column `j`) of active matrix at iteration `k`. 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 should have a minimum Markowitz count and a maximum absolute value to ensure stability. On the other hand this search is prohibitive, since 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 with the following equation to ensure numerical stability: $|A_{ij}^{(k)}| \geq u \cdot \max_l |A_{lj}^{(k)}|$. This input parameter `u`, $0 < \texttt{u} \leq 1$, will prioritize stability when $\texttt{u} \rightarrow 1$, or sparsity when $\texttt{u} \rightarrow 0$.

To control sparsity, an 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.

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. In addition, 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.

3 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. 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 selected some heterogeneous unsymmetric sparse matrices from Harwell-Boeing [9] and the University of Florida [6] matrix collection. As a message-passing interface, SHMEM routines have been used since 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 $Q = 1$.

3.1 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 1. 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`$= 16$.

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 1. The influence of the `u` parameter on LNS3937

In table 1 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: increasing $\overline{m}$ makes the number of outermost loop iterations decrease, thus reducing both sequential and parallel execution time. Additionally, fill-in is reduced when `u` is diminished, since 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 [8, 7] 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 2 for the same matrix LNS3937, with `u`$= 0.1$ and `ncol`$= 16$.

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

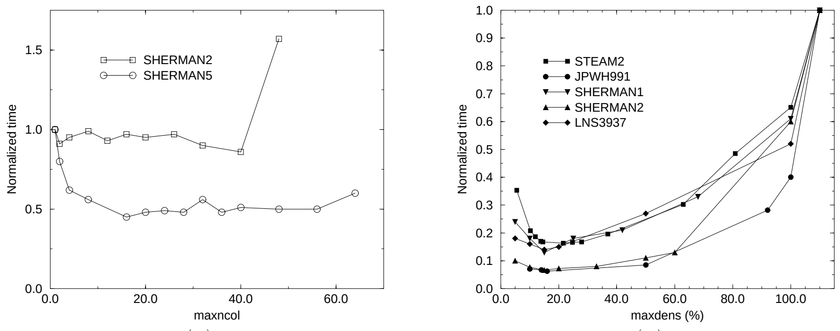
Table 2. The influence of the `a` parameter on LNS3937

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-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 [4,15]).

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 1 (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`.

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.



(a) (b)

Fig. 1. Fixed `ncol` (a) and `maxdens` (b) influence on execution time

Finally, we have also studied execution time dependence with the threshold `maxdens` which decides the switch to a dense code. In figure 1 (b) we show the relation between `maxdens` and execution time for some matrices. Execution times are normalized by the worst case (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`≈15%.

3.2 Comparison with the MA48 routine

Before studying the parallel algorithm performance, it is important to check that the sequential version is good enough. The MA48 routine [10] is one of the generic sparse system solver more widely used. 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 3 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

	Time		Error	Fill-in
Matrix	**SpLU–MA48**	**ratio**	**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 3. SpLU and MA48 comparison

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

3.3 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 3.1 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 4 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 2 shows speed-up and efficiency when factorizing the 9 computationally more expensive matrices for four 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 4. 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 exhibit 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

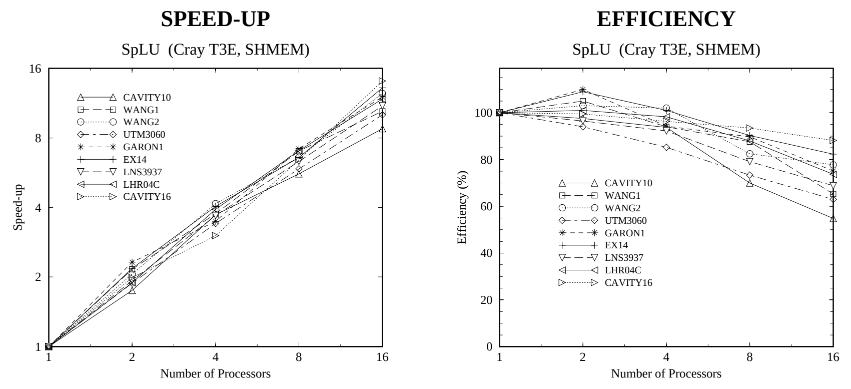


Fig. 2. SpLU speed-up and efficiency

for this step. 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 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 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.

4 Related work

In this section we summarize recent works, organizing them according to the level of parallelism. Regarding task level parallelism, Gallivan, Marsolf, and Wijshoff (1996) [12] carry out a matrix reordering to a bordered block triangular form. In the same direction, Zlatev et al. (1995) [19] have developed the tool PARASPAR, to solve the linear system on shared memory multiprocessors. By using a better reordering stage, LORA-P^5 code, the Y12M3 program achieves speed-ups between 3.0 and 4.7 on 8 processors of the Alliant FX/80 [18].

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. Gupta, Karypis, and Kumar (1995) [14] were the authors who reported probably the best performance for the sparse cholesky.

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. [16], achieving on 8 processors shared memory machines, and for 21 unsymmetric sparse matrices, an average speed-up of less than 4. Better results can be achieved on distributed memory machines as recently shown by Fu, Jiao, and Yang (1998) [11]. 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.

Regarding the loop level parallelism and parallel pivots approach, there are some algorithms for shared memory machines: Alaghband (1995) [1], Davis and Yew (1990) [5], and Zlatev et al. (1995) [19]. The experimental results of these previous works lead us to conclude that actual system solver implementations for shared memory multiprocessors hardly exceed 50% efficiency in 8 processors.

Better results are reached for distributed memory machines as presented by Stappen, Bisseling, and van der Vorst (1993) [17] for a square Transputer mesh, and by Koster and Bisseling (1994) [15] also for a transputer mesh. In these codes, they do not present a switch to a dense code stage nor parallel solve stage.

5 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-in than MA48 does. On the other hand, MA48 is usually slightly faster than the sequential SpLU, but to the best of our knowledge MA48 can not be parallelized efficiently. Therefore, since SpLU exhibits a high degree of parallelism, speed-up computed as MA48 sequential time divided by parallel SpLU execution time is still competitive.

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 overhead 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.

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 sparse matrix solution and performance. *Parallel Computing*, 21(9):1407–1430, 1995.
2. R. Asenjo and E.L. Zapata. Parallel pivots lu algorithm on the Cray T3E. Technical Report UMA-DAC-99/01, Dept. of Computer Architecture, University of Mlaga, Spain, http://www.ac.uma.es/, 1998.
3. E. Chu and A. George. Gaussian elimination with partial pivoting and load balancing on a multiprocessor. *Parallel Comput.*, 5:65–74, 1987.
4. 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.
5. 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.
6. Tim Davis. Sparse matrix collection. At URL http://www.cise.ufl.edu/ davis/.
7. 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.
8. I.S. Duff, A.M. Erisman, and J.K. Reid. *Direct Methods for Sparse Matrices.* Oxford University Press, Oxford, U.K., 1986.
9. 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.
10. 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.
11. 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.
12. K. Gallivan, B. Marsolf, and H.A.G. Wijshoff. Solving large nonsymmetric sparse linear systems using MCSPARSE. *Parallel Computing*, 22(10):1291–1333, 1996.
13. G. A. Geist and C. H. Romine. LU factorization algorithm on distributed-memory multiprocessor architecture. *SIAM J. Sci. Statist. Comput.*, 9:639–649, 1989.
14. 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.
15. 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.
16. X. Li. *Sparse Gaussian Elimination on High Performance Computers.* PhD thesis, CS, UC Berkeley, 1996.
17. 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.
18. 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.
19. 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.