Sparse LU Factorization on the Cray T3D

R. Asenjo
E.L. Zapata

May 1995
Technical Report No: UMA-DAC-95/08

Published in:

Int'l. Conf. on High-Performance Computing and Networking
Milan, Italy, May 3-5, 1995, pp. 690-696
(Springer-Verlag, LNCS 919)

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

SPARSE LU FACTORIZATION ON THE CRAY T3D*

R. Asenjo and E. L. Zapata

Computer Architecture Department. University of Málaga
Plaza El Ejido S/N. 29013 Málaga. SPAIN.
E-mail: {asenjo, ezapata}@atc.ctima.uma.es
Tel: +34 5 213 14 04 Fax: +34 5 213 14 13.

Abstract. The paper describes a parallel algorithm for the LU factorization of sparse matrices on distributed memory machines by using SPMD as programming model and PVM as message passing interface. We address all the difficulties arising in sparse codes, as the *fill-in* or the dynamic movement of data inside the matrix. The cyclic distribution has been used to evenly distribute the elements onto a mesh of processors, whereas two local storage schemes are proposed: A semi-ordered and two-dimensional linked list, which fulfils better the requirements of the algorithm, and a compressed storage by rows, which behaves better in the use of memory. The properties of the code are extensively analyzed and execution times on the CRAY T3D are presented to illustrate the overall efficiency achieved by our methods.

1 Introduction

The solution of linear systems $Ax = b$, where the coefficient matrix A has a sparse sort and huge dimensions, plays a basic role in many fields of the science, engineering and economy.

Throughout this paper, we assume a nonsingular sparse matrix A with dimensions $n \times n$ (in the whole matrix there will only be α nonzero elements, such that $\alpha \ll n^2$). By exploiting the sparsity of A, it is possible to reduce significantly the execution time and the memory required to solve the linear system.

There are several methods for solving sparse linear systems [4], [13]. One of them is based on the LU factorization of A [5]. The output of such a factorization is a couple of matrices, L (lower triangular) and U (upper triangular), with dimensions $n \times n$, as well as the permutation vectors, π y ρ, with dimension n, such that:

$$A_{\pi_i,\rho_j} = (LU)_{ij} \qquad \forall i,j,\ 0 \leq i,j < n. \tag{1}$$

* This work was supported by the Ministry of Education and Science (CICYT) of Spain under project TIC92-0942-C03, by the Human Capital and Mobility programme of the European Union under proyect ERB4050P1921660, and by the Training and Research on Advanced Computing Systems (TRACS) at the Edinburgh Parallel Computing Centre (EPCC)

The permutation vectors, π y ρ, are needed due to the permutation process taking place during the factorization in rows and columns of A, with the aim of preserving the sparsity rate and ensuring the numerical stability.

Sequential algorithms already developed for the sparse LU factorization, like MA28 [3] or Y12M [12], perform several iterations, each involving a pivot search in the reduced matrix, followed by a row and column swapping, and an update of range one in the *reduced matrix* (defined as a submatrix containing $(n-k) \times (n-k)$ elements A_{ij}, such that $k \leq i, j < n$, in the $k-th$ iteration). This pivot must be chosen in such a way that the sparsity rate be preserved and the numerical stability guaranteed.

The more widely heuristic strategy used for finding pivots to preserve the sparsity rate is known as Markowitz's strategy [8]. When choosing a pivot, A_{ij}, it may create $M_{ij} = (R_i - 1)(C_j - 1)$ new nonzero elements in the worst case, where R_i (C_j) denotes the number of nonzero elements in the $i-th$ row ($j-th$ column). The upper bound M_{ij} is known as *Markowitz count* [4, Chap. 7]. The pivot will be chosen such that minimizes the Markowitz count.

In addition, the numerical stability must be guaranteed. In order to accomplish it, we must avoid the selection of pivots having a low absolute value, what leads us to accept candidate pivots, A_{ij}, fulfilling the condition:

$$|A_{ij}| \geq u \cdot \max_{l} |A_{lj}|, \tag{2}$$

where u, $0 \leq u \leq 1$ is a threshold parameter [4, Chap. 7].

This paper is organized as follows. Section 2 describes the data distribution. For a detailed explanation of the parallell algorithm and its implementation see [1]. Section 3 presents the execution times and workload balance on the CRAY T3D [9] for different sizes of the sparse matrix selected from the Harwell-Boeing sparse matrix collection.

2 Parallel Sparse LU Algorithm

The parallel algorithm executes a number of iterations, each involving three different phases: Pivots search, rows and columns permutation, and reduced matrix and R and C vectors update. This factorization was broached by Stappen et al. on a network of transputers [11].

The inherent parallelism of the sparse LU algorithm involves two issues. First of all, in the dense case, we can parallelize the loops traversing all the updating of range one in the reduced matrix. This parallelism is inherited by the sparse algorithm. Secondly, the sparse algorithm allows us to perform parallel computations that must be sequentialized in the dense case. Thus, in the sparse algorithm, it is possible to merge several updates of range one in a single update process of multiple range (m) by modifying the Markowitz strategy in such a way that we search for a pivot set containing m compatible pivots (referred to as $PivotSet$). Two pivots, A_{ij} and A_{kl} are compatible and independent if $A_{il} = A_{kj} = 0$..

2.1 Data Distribution

Matrix A is distributed onto a $P \times Q$ mesh of processors. We will identify each processor by means of its cartesian coordinates (r_0, r_1), with $0 \leq r_0 < P$ and $0 \leq r_1 < Q$. Nonzero elements of A are mapped over processors by using a *scatter distribution* or *cyclic storage*

$$A_{ij} \longmapsto PE(imodP, jmodQ) \quad \forall i, j, \;\; 0 \leq i, j < n. \tag{3}$$

This distribution optimizes the workload balance when the probability of a nonzero element is regardless of its coordinates. Even though such a condition is not fulfilled, when we have clusters grouping nonzero elements (for instance, in the lower right corner of the matrix), those will be spread on several processors, achieving a good workload balance as well.

Permutation vectors π and ρ are partially replicated: We will store π_i on processors with coordinates (i mod P, $*$), and ρ_j on processors with coordinates ($*$, j mod Q). Vectors R and C are distributed in the same way that π and ρ, respectively.

We will name $\hat{A}$ to the local matrix of $\hat{m} \times \hat{n}$, where $\hat{m} = \lceil n/P \rceil$, and $\hat{n} = \lceil n/Q \rceil$. The *hat* notation will be used to distinguish local variables and indices from global ones. Hence, on processor (s, t), the relationship between A and $\hat{A}$ will be given by the following equation:

$$\hat{A}_{\hat{i}\hat{j}} = A_{\hat{i}P+s, \hat{j}Q+t} \quad \forall \hat{i}, \hat{j}, \;\; 0 \leq \hat{i}P + s, \hat{j}Q + t < n. \tag{4}$$

The local data storage follows two strategies. Firstly, it has been used a semi-ordered, two-dimensional doubly linked-list. Such a dynamic data structure, links in a list all the nonzero elements belonging to the same row in a sorted way, and the ones belonging to the same column in a non-sorted way. The set of $\hat{m}$ ($\hat{n}$) pointers pointing to the first elements of local rows (columns) are sorted in an array *rows* (*cols*). Each item stores, not only the value and the local indices, but also pointers accessing to the previous and next element in its row and column. The data structure described above enables data accesses quickly either by rows or by columns, and makes easier the delete operation. On the other hand, each item occupies a significant amount of memory.

We can also choose another structure to reduce the wastage of memory with respect to the previous one. We will name it Block Row Scatter (BRS) [10]. Let us assume the matrix A to be partitioned into a set of submatrices $B(l, k)$ with dimensions $P \times Q$, such that $A_{ij} =^{kl} B_{st}$ where $i = k \cdot P + s, \;\; j = l \cdot Q + t \;\; (0 \leq i, j < n)$. Pairs (i, j), (s, t) and (k, l) represent global, local and block indices, respectively. In order to perform the partition of A giving submatrices $B(k, l)$, it may be necessary to add new rows or columns containing null elements to it. The distribution of the elements of A is performed by mapping each block of size $P \times Q$ onto the mesh of processors, and the data storage format consists of three vectors, D, C y R. The D vector stores the nonzero values of the matrix, as they are traversed in a row-wise fashion. The C vector contains the block column indices of the elements in the D vector. Finally, the R vector stores the

indices in the D vector corresponding to the first non-zero element on each row. By convention, we define one additional element in R with a value equal to the number of elements in D plus one. The memory spent is drastically reduced, although the access to the data by columns may be slowed down.

In Figure 1, we show an example of a sparse matrix with $n = 8$ and $\alpha = 13$ and the distribution process, using BRS (on a 2×2 mesh).

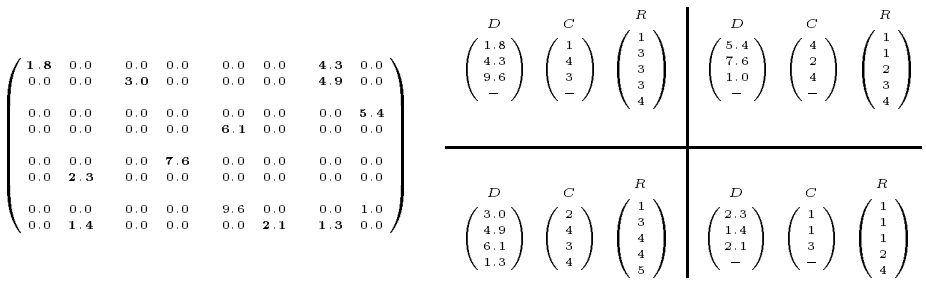

Fig. 1. Doubly linked list and BRS storage ($P = Q = 2$).

3 Experimental Results on the CRAY T3D

The algorithm has been implemented by using the C language and the PVM message-passing interface [7] as provided for the CRAY T3D supercomputer. The portability of the C language and the PVM interface let the target code be compiled with minor modifications over other platforms. We have performed all the experiments on the CRAY T3D, using a maximum of 64 DEC-Alpha processors (150 MHz) connected by a tridimensional torus topology.

On this machine, the communications under PVM have the following features: bandwidth reaches 40-60 Mbps and latency 50-70 μs, both using communication functions as *pvm_send* and *pvm_recv*; however, by using *pvm_fastsend* and *pvm_fastrecv* (non-standard PVM functions), latencies are about 15-20μs when the message length is reduced (256 bytes by default).

Typical optimizations to minimize the communication latency time have been implemented by packing all the messages as many as possible. Further, messages are broadcasted from the center to the edges and vice versa, what decreases the number of messages and duplicates the speed of the broadcast and reduce operations.

As an example, we accumulate all the incompatible pivots in the processor $(P/2, Q/2)$, where we create $PivotSet$ and, subsequently, we broadcast it with a complexity of $\mathcal{O}((P+Q)/2)$. Note that Stappen et al. create the PivotSet by considering a pipeline in the first row of the mesh, in which all the pivots are passing one by one through the mesh, from processor $(0, t)$; $0 \leq t < Q$ to processor $(0, Q-1)$, creating in this last processor the $PivotSet$. This alternative increases the total number of messages and the overhead associated to the latencies.

3.1 Harwell-Boeing Sparse Matrix Collection

With the aim of testing the performance achieved by our algorithm, we have chosen a set of five sparse real matrices and non-symmetric [2] with CCS (Compressed Column Storage) format from the Harwell-Boeing sparse matrix collection [6]. These matrices come from several realistic applications and possess size enough to make them worth to be implemented on a multiprocessor machine like the CRAY T3D. In table 1 we show the features for these matrices, as the dimension, n, and the number of nonzero elements or *entries*, α. Moreover, we will note α', the number of *entries* once the factorization is over (number of *entries* in the $L\backslash U$ matrix). It is also interesting to show the number of *entries* by row, initially (α/n), and finally (α'/n).

Matrix	Origin	n	α	α'	α/n	α'/n
STEAM2	Oil reservoir simulation	600	13760	52099	22.93	86.83
JPWH 991	Electronic circuit simulation	991	6027	63789	6.08	64.37
SHERMAN1	Oil reservoir simulation	1000	3750	22680	3.75	22.68
SHERMAN2	Oil reservoir simulation	1080	23094	177417	21.38	164.28
LNS 3937	Compressible fluid flow	3937	25407	437099	6.45	111.02

Table 1. Test set of sparse matrices

3.2 Execution Times and Speed-up

Throughout this section we compare the execution times of the parallel program, executed on a mesh of $P \times Q$ processors, and the sequential version, executed on a single Alpha processor. The parallel program is an implementation of the algorithm described in section 2. The data structure used is a semi-ordered two-dimensional doubly linked-list. We will make the features of the algorithm regardless of the number of processors so that the execution times over different meshes are comparable. Thus, we will establish the input parameters beforehand. The number of columns in which each processor will search for candidate pivots, $ncol$, will be determined to $16/Q$ ($Q = 1 \Rightarrow ncol = 16$, $Q = 2 \Rightarrow ncol = 8$, ... $Q = 16 \Rightarrow ncol = 1$). All the candidates must fulfil equation 2 with $u = 0.1$ as Duff, Erisman and Reid [4, Chap. 7] suggest. Candidates with $M_{ij} > a \cdot M_{i_0,j_0}$ will be rejected. We will set $a = 4$ as Davis and Yew [2] and Stappen et al. [11] did in their experiments.

The sequential program is an optimized version of the parallel program, made by simplifying the parallel program, where we remove all the parallel overhead and exploit $P = 1$ and $Q = 1$ as much as possible.

Table 2 presents the execution times for the 5 matrices and different sizes of the mesh. We will name T_p to the parallel time, and T_{seq} to the sequential one.

As we can see, times are monotonically decreasing as the number of processors is increased. With SHERMAN1 there is one exception going from 4×4 to

[2] In case of symmetric matrices, the Cholesky factorization algorithm is more efficient

8×8 processors. The reason is that SHERMAN1 is the most sparse matrix in table 1, not only previously to the factorization, but also after it (α/n and α'/n minimums). This produces a low relation between the number of local operations and communications. Furthermore, the messages sent have few data; so, the latency time predominates over the transmission time.

In table 2 it is also shown the speed-up, $S_p = T_{seq}/T_p$ where $p = P \times Q$ processors, and the efficiency, $E_p = S_p/p$, of the parallel algorithm by using these matrices and different sizes of the mesh. As we see, the algorithm scales rather good. The low efficiencies achieved on 64 processors can be explained if we take into account the small size of the matrices and the low number of floating point operations. In addition, there is a high ratio between the power of the Alpha processor and the bandwidth of the interconnection network using PVM. The best efficiencies are achieved with the SHERMAN2 matrix. This is

Matrix	Time (sec.)				Speed-up			Efficiency(%)		
	seq	2×2	4×4	8×8	2×2	4×4	8×8	2×2	4×4	8×8
STEAM2	9.89	3.57	1.90	1.47	2.77	5.19	6.71	69.28	32.45	10.49
JPWH 991	16.22	6.74	3.16	2.14	2.40	5.12	7.54	60.16	32.03	11.79
SHERMAN1	3.02	1.93	1.35	1.39	1.56	2.23	2.17	39.08	13.95	3.40
SHERMAN2	81.08	22.81	8.12	4.67	3.55	9.98	17.33	88.87	62.38	27.09
LNS 3937	204.55	81.89	28.58	18.07	2.49	7.15	11.31	62.44	44.73	17.68

Table 2. Time, speed-up and efficiency on different sizes of the mesh

the most dense matrix in table 1, not only before the factorization (density of 1.97%), but also after it (density of 15.21%). It is also seen that the efficiency depends more on the ratio between the number of entries and the number of rows (α/n) than on the number of entries itself.

4 Future Work

In subsequent experiments we will be able to measure execution times for matrices with higher number of rows and entries per row, whereby the program must reach even better efficiencies.

Moreover, we will prove the BRS strategy to provide good efficiencies in addition to less memory requirements. As we discussed in subsection 2.1, following this storage strategy, the access to the data by columns it is slower than in the linked list structure. Because of this, we implement the pivot search by rows and permute the two inner loops in order to divide rows by the diagonal.

The back-substitution and forward-substitution phases will also be implemented soon, since they are required by a wide number of real applications.

Acknowledgments

We gratefully thank to K.I.M. Mc Kinnon from the Mathematics and Statistics Dept. of the University of Edinburgh the interest showed in this work, as well as the staffs of TRACS support and the Edinburgh Parallel Computing Centre for giving us access to the CRAY T3D machine and initiate us on its handling.

References

1. R. Asenjo and E.L. Zapata. Sparse LU factorization on the Cray T3D. Technical report, Computer Architecture Dept. Univ. of Málaga, September 1994. Anonymous ftp: ftp.atc.ctima.uma.es (SpLUCray.ps).
2. 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.
3. I. S. Duff and J. K. Reid. Some design features of a sparse matrix code. *ACM Trans. Math. Software*, 5:18–35, 1979.
4. I. S. Duff A. M. Erisman and J. K. Reid. *Direct Methods for Sparse Matrices.* Oxford University Press, Oxford, U.K., 1986.
5. G. H. Golub and C. F. Van Loan. *Matrix Computation.* The Johns Hopkins University Press, Baltimore, MD, 2 edition, 1989.
6. I. S. Duff R. G. Grimes and J. G. Lewis. Sparse matrix test problems. *ACM Trans. Math. Software*, 15:1–14, 1989.
7. A. Geist A. Beguelin J. Dongarra W. Jiang R. Manchek and V. Sunderam. *PVM 3 User's Guide and Refernce Manual.* Engineering Physic and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, May 1993.
8. H. M. Markowitz. The elimination form of the inverse and its application to linear programming. *Management Sci.*, 3:255–269, 1957.
9. S. Booth J. Fisher N. MacDonald P. Maccallum E. Minty and A. Simpson. *Parallel Programming on the Cray T3D.* Edinburgh Parallel Computing Centre, University of Edinburgh, U.K., September 1994.
10. R. Asenjo L. F. Romero M. Ujaldón and E. L. Zapata. Sparse block and cyclic data distributions for matrix computations. In *High Performance Computing: Technology and Application.* Grandinetti et al. (Eds.) Elsevier Science, (to appear).
11. 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.
12. Z. Zlatev J. Wasniewski and K. Schaumburg. Y12M—Solution of Large and Sparse Systems of Linear Algebraic Equations. In *Lecture Notes in Computer Science*, number 121, pages 61–77, Berlin, 1981. Springer-Verlang.
13. Z. Zlatev. *Computational Methods for General Sparse Matrices, Mathematics and Its Applications.* 65. Kluwer Academin Publisher, Dordrecht, the Netherlands, 1991.