Performance Complexity of LU Factorization with Efficient Pipelining and Overlap on a Multiprocessor

F. Desprez* and B. Tourancheau †‡

LIP - ENS Lyon, URA 1398 du CNRS, 46, Allée d'Italie, 69364 Lyon Cedex 07, France

The University of Tennessee, Computer Science Department, 107, Ayres Hall, Knoxville, TN 37996-1301, USA

J. J. Dongarra§

The University of Tennessee, Department of Computer Science, 107 Ayres Hall, Knoxville, TN 37996-1301, USA

Oak Ridge National Laboratory, Mathematical Sciences Section, P.O. Box 2008, Bldg. 6012, Oak Ridge, TN 37831-6367, USA

February 8, 1994

Abstract

In this paper, we make efficient use of pipelining on LU decomposition with pivoting and a column-scattered data decomposition to derive precise variations of the computational complexities. We then compare these results with experiments on the Intel iPSC/860 and Paragon machines.

1 Introduction

This paper presents an analytical estimation of the LU factorization algorithm on a distributed-memory, message-passing multiprocessor. We focus on column-scattered data distribution and the columnwise kji, or "right looking," elimination method with pivoting.

Our procedure comprises two parts. First, we study the algorithm with synchronous communication and compare the execution times for a complete network and a ring topology. Second, we introduce the possibility of overlapping the communication by the computation (i.e., asynchronous

*This work was supported by MRE grant No. 974, the CNRS-NSF grant No. 950.22/ 07 and the research program C3.

†On leave from LIP, CNRS URA 1398, ENS Lyon, 46 allée d'Italie, 69364 Lyon Cedex 07, France.

‡This work was supported in part by CNRS-NSF grant 950.223/07, Archipel SA and MRE under grant 974, the National Science Foundation under grant ASC-8715728, the National Science Foundation Science and Technology Center Cooperative Agreement CCR-8809615, the DARPA and ARO under contract DAAL03-91-C-0047, PRC C^3, and DRET.

§This work was supported in part by the National Science Foundation under grant ASC-8715728, the National Science Foundation Science and Technology Center Cooperative Agreement CCR-8809615, the DARPA and ARO under contract DAAL03-91-C-0047

```
for k = 0 to n − 2
   Scale:  execute task S_k
   for all columns j ≥ k + 1
      Update:  execute task U_kj
   endfor
endfor
```

Figure 1: Sequential "right looking" algorithm for Gaussian elimination

communication). Using our earlier analysis, we derive the equivalent of the pipeline ring algorithm and the broadcast algorithm for any complex topology.

In each part, we introduce different cases depending on the target machine parameters that correspond to different critical paths of the execution. The analytical complexities are corroborated by experiments on Intel iPSC/860 and Paragon machines.

2 Gaussian Elimination and LU decomposition

Gaussian elimination can be used in the solution of a system of equations $Ax = b$. This process transforms the matrix A in a triangular form with an accompanying update of the right-hand side, so that the solution of the system $Ax = b$ is straightforward by a triangular solve.

LU factorization uses the same algorithm and converts the matrix A in two matrices L and U, where $A = LU$ and L and U are lower and upper triangular, respectively. Hence, many systems can be solved by two triangular solves, $Ly = b$ and $Ux = y$.

There are many versions of the LU algorithm depending on the way the three internal loops are nested [Rob90, GL89]. We study the kji, or "right looking," form, which is most suitable for parallel implementation (since the matrix can be distributed by columns, the pivoting is done easily inside each processor without communication [PBKP92]).

- Task S_k comprises the pivot search, interchange, and scaling of the elimination column.
- Task U_{kj} comprises the interchange of pivot elements and the updating of the remaining column j.

(Note: the multipliers, L, are saved in the array A in place of the elements that would become zero in task S_k.)

3 Parallel kji Version of the LU Algorithm with Column-scattered Data Distribution

Parallel versions of the LU algorithm are described in [Saa86a, Saa86b, RTV89, Rob90]. Depending on the topology, the difference between the methods is the manner in which the elimination column is sent to all the processors. Two well-known methods for broadcast are the minimum spanning tree broadcast and the pipeline ring (unidirectional broadcast along the ring) algorithms [Saa86a, RTV89, Rob90].

```
Task S_k                          Task U_kj
   pivot(k, ipvt(k))
   interchange(k, ipvt(k))           interchange(k, ipvt(k))
   c = 1/a_kk                        for i = k+1 to n-1
   for i = k+1 to n-1                    a_ij = a_ij - a_ik * a_kj
       a_ik = a_ik * c               endfor
   endfor
```

Figure 2: Algorithms for the tasks S_k and U_{kj} of Gaussian elimination

```
me = my_id()
for k = 0 to n-2
    if (alloc(k) == me)                                 /* I own pivot col.  */
        S_k                                             /* scale pivot col.  */
    endif
    C_k(me)                          /* pipeline broadcast of pivot col.  k */
    for all j >= k and alloc(j) == i
        U_kj(me)                                  /* update internal col.  j */
    endfor
endfor
```

Figure 3: Parallel pipeline broadcast LU decomposition algorithm

```
Task C_k(me)
  if (alloc(k) == me)
    message := concatenate(ipvt(k), A[k, k+1 : n-1])
  endif
  pipeline_broadcast(me, message, n-k)
```

Figure 4: Algorithm for the task C_k of Gaussian elimination

In the following parallel algorithm, we assume that the data are equally distributed in a wrapped manner along a virtual ring. Thus, with our algorithm, the same number of columns is assigned to each processor; and, as the column distribution is wrapped, a given processor has a scattered collection of columns. This approach ensures a good balance of the computational load between the processors.

For the parallel kji LU decomposition, we selected the following set of computational tasks: S_k is the scaling of the pivot column, $U_{kj}(me)$ is the update of the internal columns (scattered decomposition) of processor me $(0 \leq me < p)$, and $C_k(me)$ is the communication broadcast of the column k on processor me using message passing (note that $C_k(me)$ can be sends, receive and sends, or a receive call, depending of the broadcast strategy and me).

Thus, the LU decomposition of an $n * n$ matrix on p processors is $(n-1)$ tasks S_j, and assuming n and p are even, $(np - \frac{p^2}{2} + \frac{p}{2})$ tasks C_j and $(\frac{n^2}{2} + \frac{n}{2})$ tasks U_{kj}.

4 Models

Our target machines are distributed-memory parallel multicomputers. We assume that the p processors $(0..p-1)$ are identical and that the same program runs (at the same time) on each processor. The communication network topology is assumed to be a fully connected network or a ring; the results for the two configurations will be compared.

In the first part, we assume that the communication protocol is synchronous. In the second part, on the other hand, we assume that the communication protocol is asynchronous; thus, each processor has independent units for communication (which manage direct-memory access) and computation (CPU). One can therefore perform in parallel on the same node at least unidirectional data transfers on each link (half duplex and two-port assumptions) and arithmetic operations.

The time needed to communicate a message of size m between two neighboring processors is modeled as the startup time β_{com} plus the length m times the throughput τ_{com} of the communication channel [SS89]. We assume that the messages are sent in one block.

For arithmetic operations, the arithmetic logical unit of the CPU is assumed to be superscalar. We use a linear model $(\beta + m\tau)$ for its performance on vector of size m. In all our experiments we use 64-bit double precision floating-point arithmetic. This model corresponds roughly to the currently available commercial machines, such as the Intel iPSC/860 [Dun90] and Paragon [EK93, Int93].

In the remainder of this paper we call β_{scal} and τ_{scal} the parameters for the cost of the inversion of the pivot element, the search for the maximum of the vector, the interchange of these two elements, and the scale of the vector by a scalar using the Level 1 BLAS `DSCAL` function [DCDH90].

We define β_{upd} and τ_{upd} as the parameters for the cost of the pivot interchanges and the multiply and add operations on the vector (`DAXPY` function of the Level 1 BLAS).

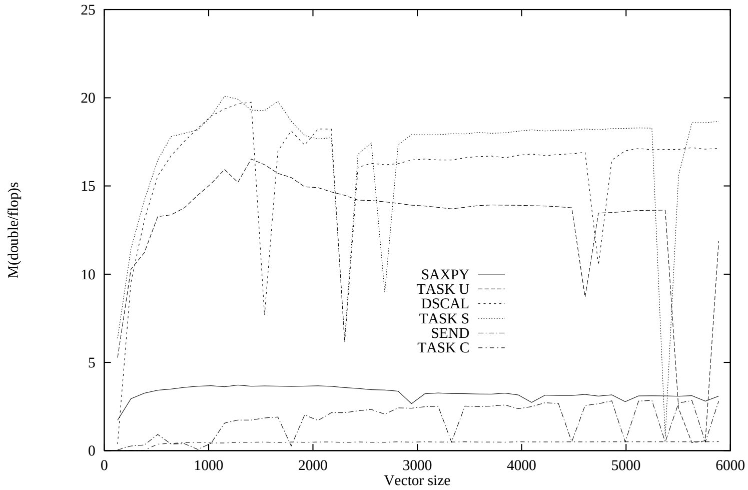

Figure 5: Intel Paragon execution performance as a function of the vector length for the taskU, taskS, and taskC with their corresponding basic kernels (daxpy, dscal, send) in Mflops or in Mwords per second

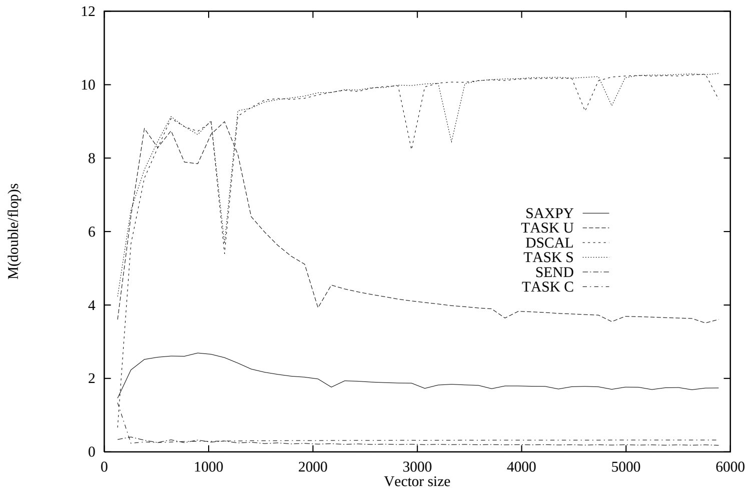

Figure 6: Intel iPSC/860 execution performance as a function of the vector length for the taskU, taskS, and taskC with their corresponding basic kernels (daxpy, dscal, send) in Mflops or in Mwords per second

Machine	$\tau_{upd}(\mu sec/flop)$	$\tau_{scal}(\mu sec/flop)$	$\tau_{com}(\mu sec/double)$
iPSC/860	0.097	0.57	5
Paragon	0.058	0.32	2

Table 1: Values of the machine parameters for the different Tasks.

These parameters of the target machine are determined experimentally with the results shown in Figures 5 and 6. The corresponding asymptotical values of the τ parameters are given in the table of Figure 1. Notice that the pivot search and interchange are costly (taskS is far from the peak performance of the `DSCAL` function) and that the target machine presents its best performance for vector of size around 1 K for the iPSC/860 and 1.5 K for the Paragon (probably because of memory and cache management).

In this paper, we study the critical path of the *LU* decomposition algorithm with column-scattered data distribution. For our path analysis, we use the notation of [LKK83], where a task is an indivisible unit of computational activity and the precedence binary relation between tasks is denoted by $\prec$. This relation is nonreflexive, antisymmetric, and transitive. If we let A and B belong to the set of tasks, then $A \prec B$ means that task A must complete execution before B commences execution.

5 Related Work

Numerous methods have been proposed for *LU* factorization (see [PBKP92] and the related works of [Saa86b, Cap87, CTV87, RT88, CRT89, DO90, Rob90]). For example, [CG87] advocates partial pivoting and load balancing in rowwise methods with a straightforward parallel triangular solve algorithm, but [LC89] shows that the parallel triangular solve algorithm can have the same performance with columnwise storage. In [RTV89], the panel-wrapped column-distribution method is proved to be efficient because it leads to a good load-balancing between computation and communication; and, in practice, this method has given good results with blocked computations [RT88, DO90]. Blockwise distribution is introduced in [CDW92], and [BDL91, DGW92] show that the communication can be reduced by using a block-wrapped data decomposition of the data.

To understand the effectiveness of these various methods and to be able to *predict* performance on a target machine, we must carry out an analytical analysis. In this paper, we focus on the performance of column-decomposition algorithms. We show that a nearly complete overlap of communications can be achieved. Thus, we obtain quasi-optimal performance with a simple column-scattered data distribution. The overlapping strategy can easily be extended to blockwise algorithms.

6 Synchronous Communication

We begin with an analysis of the algorithm with no overlapping. Two configurations are compared: a complete network and a ring topology.

6.1 Analysis of the Pipeline algorithm on the Complete Network (PC algorithme)

With our model when no overlap is allowed, the critical path of LU decomposition is given by the precedence constraints of the sequential execution and is described by the precedence graph of Figure 7, where the edges of the path represent the relation $\prec$ between tasks.

The critical path follows the following schema (where k represents the remaining local columns to update):

$S_0 \prec C_0(0) \prec C_0(1) \prec U_{0,k}(1) \prec S_1 \prec C_1(1) \prec C_1(2) \prec U_{1,k}(2) \prec \ldots \prec S_{n-2} \prec C_{n-2}(p-2) \prec C_{n-2}(p-1) \prec U_{n-2}(p-1)$

Following this critical path, we compute the execution time analytically.

Proposition 1 *The total parallel execution time of the PC algorithm with no overlap of communication and computation is given by Equation 1.*

$$T_{pipe}^{complete} = T_{scal}^{c} + T_{com}^{c} + T_{upd}^{c}, \tag{1}$$

where

$$T_{scal}^{c} = \sum_{i=0}^{n-2} (\beta_{scal} + (n-i-1)\tau_{scal}) = (n-1)\beta_{scal} + \frac{n(n-1)}{2}\tau_{scal} \tag{2}$$

$$T_{com}^{c} = \sum_{i=0}^{n-2} (\beta_{com} + (n-i)\tau_{com}) = (n-1)\beta_{com} + \frac{(n+2)(n-1)}{2}\tau_{com} \tag{3}$$

$$\begin{aligned} T_{upd}^{c} &= \sum_{j=0}^{\frac{n}{p}-1} \left[\sum_{k=1}^{p} \left((\frac{n}{p} - j)(n - (j(p-1)+k))\tau_{upd} + \beta_{upd} \right) - (\beta_{upd} + \frac{n^2}{p}\tau_{upd}) \right] \\ &= (n-1)\beta_{upd} + \left(\frac{n^3}{3p} + \frac{n^2}{4} - \frac{3n^2}{4p} + \frac{n}{4} - \frac{np}{12} \right) \tau_{upd} \end{aligned}$$

Thus,

$$\begin{aligned} T_{pipe}^{complete} &= (n-1)(\beta_{upd} + \beta_{scal} + \beta_{com}) + \frac{n(n-1)}{2}(\tau_{scal} + \tau_{com}) \\ &+ \left(\frac{n^3}{3p} + \frac{n^2}{4} - \frac{3n^2}{4p} + \frac{n}{4} - \frac{np}{12} \right) \tau_{upd} \end{aligned} \tag{4}$$

Note that on other topologies, the critical path must follow the *alloc* decomposition of the $C_j(alloc(j))$ and $U_j(alloc(j))$ between the S_j and S_{j+1} tasks.

6.2 Analysis of the Pipeline algorithm on the Ring (PR algorithm)

Figure 7 shows the critical path of the pipeline LU algorithm corresponding to the ring topology on the left. This topology introduces idle times because of the precedence edges between tasks $C_j(i-1)$ (send) and $C_j(i)$ (receive), which are now executed one by one from processor to processor. Depending on the machine parameters, these dependencies introduce various idle periods of time in the critical path. If we examine the "zoom" in Figures 8 and 9, we can distinguish two cases: when $T_{scal(k)} > T_{com(k-2)}$ (case 1) and its inverse (case 2):

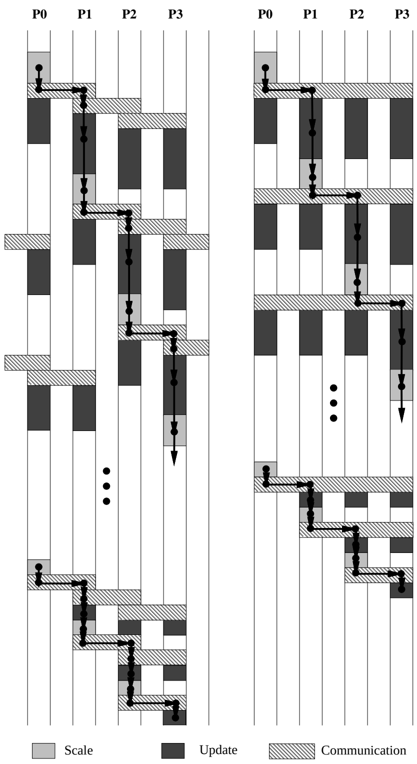

Figure 7: Time diagram of synchronous algorithms (left on the ring, right on the complete network)

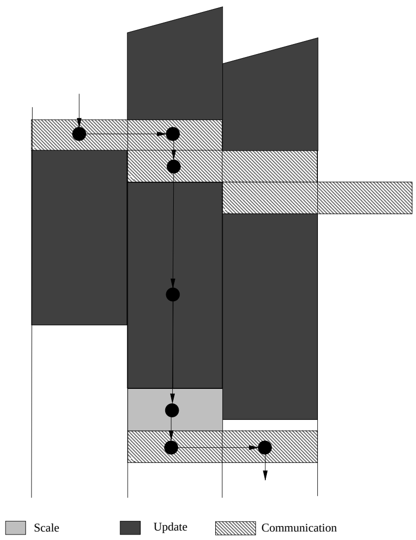

Figure 8: Zoom on the idle times in the critical path (case 1)

At step k, we can determine into which case the algorithm fell:

$$\begin{aligned} T_{scal(k)} &> T_{com(k-2)} \\ \beta_{scal} + (n-k+1)\tau_{scal} &> \beta_{com} + (n-k+2)\tau_{com} \\ k > or &< \frac{\beta_{com} - \beta_{scal} + (n+2)\tau_{com} - (n-1)\tau_{scal}}{\tau_{com} - \tau_{scal}} \end{aligned} \tag{5}$$

depending on whether the sign of $\tau_{com} - \tau_{scal}$ is > 0 or < 0. Generally (and it is true with our target machines), τ_{com} is at least one order bigger than τ_{scal}. Thus the preceding formula 5 can be estimated as follows:

$$k \quad > \quad (n+2) + o(1),$$

which is never true. Hence, on most machines the complexity analysis will correspond to case 2.

The execution time of the PR algorithm is then computed following the critical path and taking into account the idle times (note that we do not find the same total time of [PBKP92] since we take into account the idle times introduced by the pipeline strategy). As the pivot column needs to be transmitted to the next processor (which holds the next column with our scattered data distribution), the T_{com} time is increased by a factor of two. The idle time T_{idle} can be expressed as a function of T_{scal} and T_{com} .

Proposition 2 *The total cost of the PR algorithm with no overlap of communication and computation is given by Equation 6.*

$$T_{pipe}^{ring} = T_{scal}^{r} + T_{com}^{r} + T_{upd}^{r} + T_{idle}^{r}, \tag{6}$$

where

$$T_{scal}^{r} = T_{scal}^{c} \tag{7}$$

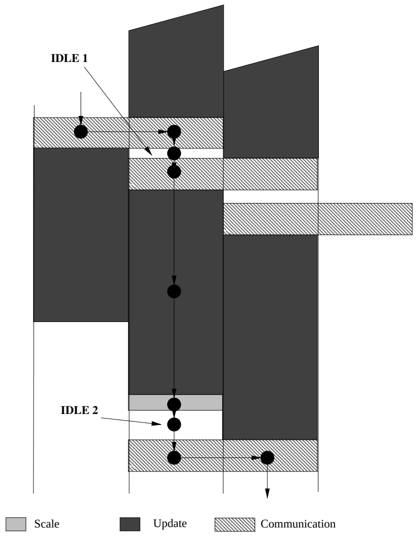

Figure 9: Zoom on the idle times in the critical path (case 2)

$$T^r_{com} = \sum_{i=0}^{n-2} 2(\beta_{com} + (n-i)\tau_{com}) = 2 * T^c_{com}$$

$$T^r_{upd} = T^c_{upd} \tag{8}$$

$$\begin{aligned} T^r_{idle} &= \sum_{i=1}^{n-3} \left((C^{i-1} - S^i)^+ + C^{i-1} - C^i\right) \\ &= \sum_{i=1}^{n-3} ((\beta_{com} + (n-i)\tau_{com} - (\beta_{scal} + (n-i-1)\tau_{scal}))^+ + \tau_{com}) \\ &= \left((n-3) * (\beta_{com} + \beta_{scal}) + \frac{(n+2)(n-3)}{2}\tau_{com} - \frac{n(n-3)}{2}\tau_{scal}\right)^+ \\ &\quad +(n-3)\tau_{com} \end{aligned} \tag{9}$$

where $(x)^+$ is the function which return x if $x > 0$ and 0 otherwise. Thus, we obtain
Case 1 *($T_{scal(k)} > T_{com(k-2)}$) :*

$$T^{ring}_{pipe} = T^{complete}_{pipe} + (n-1)\beta_{com} + \frac{n(n-1)}{2}\tau_{com} \tag{10}$$

Case 2: *($T_{scal(k)} \leq T_{com(k-2)}$) :*

$$
\begin{aligned}
T_{pipe}^{ring} &= T_{pipe}^{complete} + (2n-4)\beta_{com} + (n^2-n-3)\tau_{com} \\
&\quad -(n-3)\beta_{scal} - \frac{n(n-3)}{2}\tau_{scal} \\
&= T_{pipe}^{ring}(case1) + (n-3)\beta_{com} + (n^2/2+3n/2-3)\tau_{com} \\
&\quad -(n-3)\beta_{scal} - \frac{n(n-3)}{2}\tau_{scal} \qquad (11)
\end{aligned}
$$

7 Asynchronous Communications

In this section, we present a strategy for improving the execution times, based on the reduction of the communication time by its partial overlap with computation. This overlap is realized inside a processor; it does not correspond to the overlap between processors steps described in [PBKP92], which is the effect of pipeline algorithms.

We will see that the execution time of the complete network can be obtained with the pipeline algorithm on a ring. Hence, on any complex topology (including a ring, complete network, hypercube, and mesh), the execution time will remain the same.

7.1 Complete Network

Figure 10 shows that the critical path on the complete network is not changed while using asynchronous communications. Although only a slight improvement occurs at the beginning of the task U_{kk} on the pivot processor, the execution time of the critical path is not affected. This is because the task C must wait for the receive to complete before the execution of the U tasks.

Proposition 3 *The total cost of the PC algorithm with asynchronous sends (overlap of communication and computation) is given by Equation 12.*

$$T_{pipe-async}^{complete} = T_{scal}^{c} + T_{com}^{c} + T_{upd}^{c}, \qquad (12)$$

where $T_{scal}^{c}, T_{com}^{c}$, and T_{upd}^{c} are the same as in the synchronous section.

7.2 Ring Topology

When asynchronous communications are available, the processor holding the next pivot column is not delayed by the sending of the current pivot column. Thus, it starts its U tasks as soon as possible.

As the data are distributed equally (see Section 4), the tasks U are nearly of the same duration; thus, the ring communications cannot perturb the critical path execution time (see Figure 11).

As shown in Figure 10, we follow the critical path to determine the total duration of the algorithm execution. It leads to the following results.

Proposition 4 *The total cost of the PR algorithm with asynchronous sends (overlap of communication and computation) is given by Equation 13.*

$$T_{pipe,asynchrone}^{ring} = T_{scal}^{r} + T_{com}^{r} + T_{upd}^{r}, \qquad (13)$$

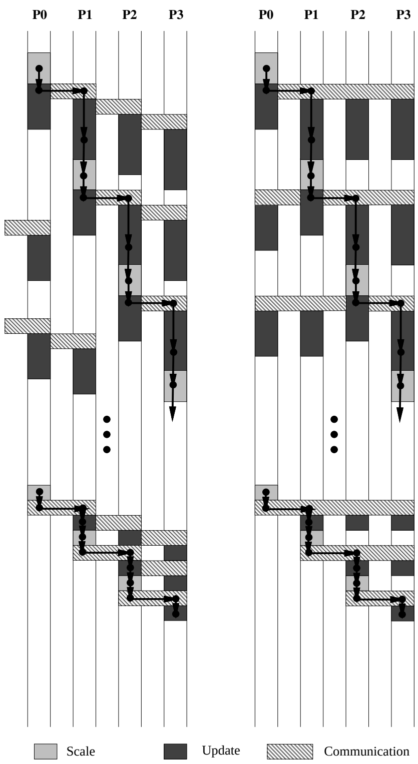

Figure 10: Time diagram of nonblocking algorithms (ring on the left, complete network on the right)

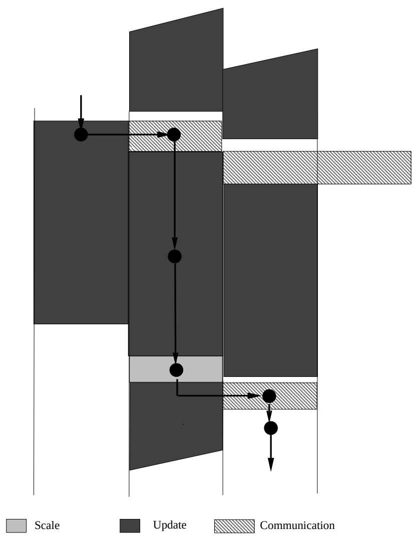

Figure 11: Zoom on the critical path

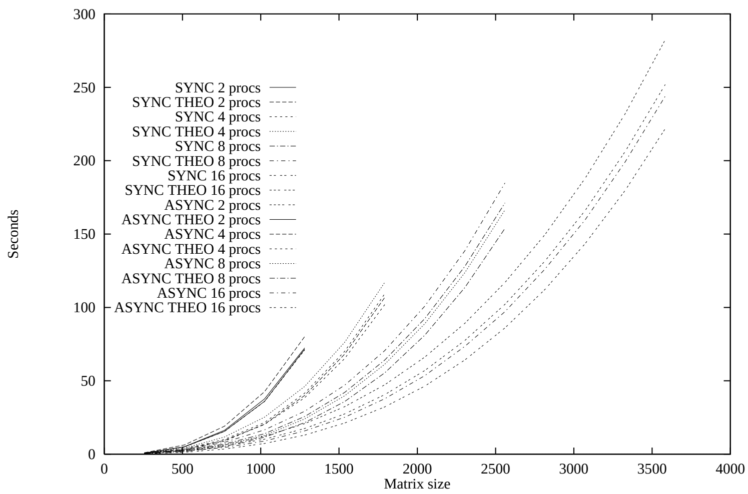

Figure 12: Intel iPSC/860 execution timings and complexity curves as a function of the matrix size

where T^r_{scal} *and* T^r_{upd} *are the same as in the synchronous case and* T^r_{com} *is half of the synchronous one.*

Thus,

$$T^{ring}_{pipe,asynchrone} = T^{complete}_{pipe,asynchrone} \tag{14}$$

We assume that the $T^r_{com} << T^r_{upd}$, which is true since small vector sizes n because $T^r_{com} = O(n^2)$ while $T^r_{upd} = O(n^3)$.

It is remarkable that, from an analytical point of view, as soon as the targeted architecture is able to run asynchronous send communications, the classical pipeline LU decomposition on a ring has the best execution time on any topology.

8 Experiments

The experiments were performed on the Intel machines. On the iPSC/860, we used 2, 4, 8 and 16 processors with a ring embedded in the hypercube. On the Paragon, we used up to 64 nodes and a virtual ring. Obviously, we were not able to conduct experiments on a complete network of processors.

To have real synchronous sends, we used the `sendrecv` function (from the vendor librairy), which makes possible to acknowledgement the reception of the message reception.

As shown by the complexity analysis, the asynchronous methods perform better than the synchronous, and the difference increases slightly with the matrix size (Figures 12 and 13). From the benchmarks described in Figures 6 and 5, we determined the parameters of our complexity analysis (see Table 1) and plotted the corresponding complexity curves with the experiments.

They nicely corroborated the timing curves on the iPSC/860 machine (see Figure 12). On one hand, the asynchronous theoretical timings are slightly better than the experiment and we think this is due to the overhead of the program management and the setup of the asynchronous calls. On the other hand, the synchronous theoretical timings are slightly worse than the experiment,

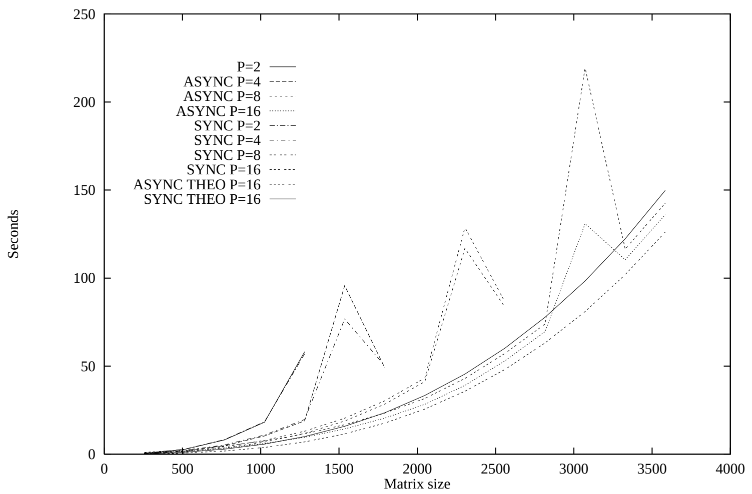

Figure 13: Intel Paragon execution timings and complexity curves as a function of the matrix size

probably because of an overlap between communication and idle time resulting of the successive communication calls (an example of such an effect for spanning binomial tree braodcast in hypercube is studied in [DFT93]).

The results were less accurate on the Paragon machine, because we were using version 1.1 of the multitasking operating system. For small machine sizes (up to 8 processors), the theoretical curves are somewhat pessimistic, but they are very close to the experimental ones and perfect for the 16-node experiments (see Figure 13). For bigger machine sizes, the operating system plays a more important role while managing the communications; thus, the theoretical predictions are far more optimistic.

We believe, however, that such analysis can predict the behavior of the algorithm on stable machines for any number of processors, giving a timing interval in which the experimental data will fit.

With the assembly-coded Level 1 BLAS routines, the total performance is in the range of our kernel tests (see Section 4). The maximum performance obtained while decomposing a 2K matrix with 8 processors is 64 Mflops on the iPSC and 136 Mflops on the Paragon (representing, respectively, 8 and 17 Mflops per processor). Figure 14 shows the timing results for our algorithm with the ring topology in the asynchronous case for the Paragon machine with up to 64 nodes.

Note that because we are not using a blocked version of the *LU* decomposition, we not could use the assembly-coded Level 3 BLAS routines and reach the performance of Choi et al. [CDW92].

9 Conclusion and Future Work

We have described an approach for analyzing the complexity of the parallel *LU* decomposition with a scattered-column data distribution in both synchronous and asynchronous communication protocols cases. We have presented a complexity analysis for each algorithm. In addition, we have described experiments to determine the values of the target machine parameters. We thus have a tool that can be used for predicting the execution time.

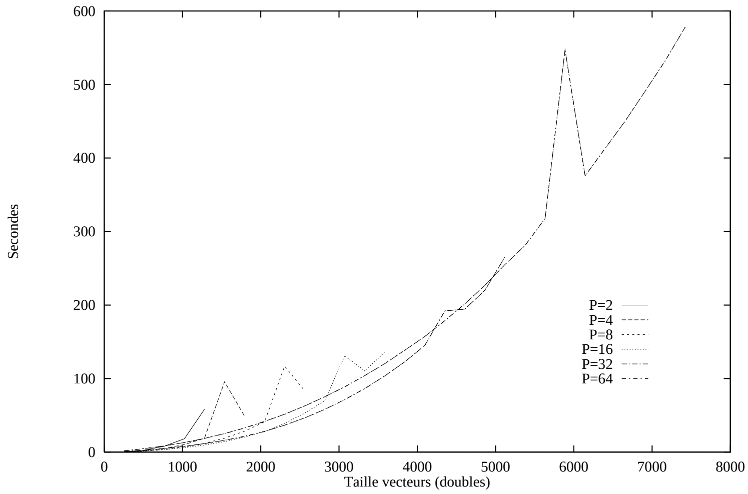

Figure 14: Intel Paragon best performance as a function of the matrix size

We have showed that in the case of synchronous communications, idle times have to be taken into account in the ring topology complexity analysis. Furthermore, these idle periods are increasing with the communication/computation elementary times ratio. On the other hand, in the case of asynchronous communications, we have showed that no idle time is introduced by the topology restriction to the ring.

Our focus has been on complete networks and on ring topologies. The analysis can be extended, however, to grid topologies with scattered block data decomposition.

Our analytical analysis has been corroborated by experiments on the Intel iPSC/860 and Paragon machines. Our experimental results also indicate that our method is efficient: we achieved 0.42 Gflops on the iPSC860 and 0.47 Gflops on the Paragon for a 7K matrix on 64 processors, compared with 1.34 Gflops of the ScaLAPACK implementation of the *LU* decomposition in the LINPACK benchmark of the iPSC860 [DGW92].

References

[BDL91] R.H. Bisseling, L. Daniel, and J.C. Loyens. Towards Peak Parallel LINPACK Performance on 400 Transputers. *Supercomputer*, VIII-5(45):20–27, September 1991.

[Cap87] P.R. Capello. Gaussian Elimination on a Hypercube Automaton. *Journal of Parallel and Distributed Computing*, 4:288–308, 1987.

[CDW92] J. Choi, J.J. Dongarra, and D.W. Walker. The Design of Scalable Software Libraries for Distributed Memory Concurrent Computers. In J.J. Dongarra and B. Tourancheau, editors, *Environments and Tools For Parallel Scientific Computing*. Elsevier, 1992.

[CG87] E. Chu and A. George. Gaussian Elimination with Partial Pivoting and Load Balancing on a Multiprocessor. *Parallel Computing*, 5:65–74, 1987.

[CRT89] M. Cosnard, Y. Robert, and B. Tourancheau. Evaluating speedups on distributed memory architectures. *Parallel Computing*, 10:247–253, 1989.

[CTV87] M. Cosnard, B. Tourancheau, and G. Villard. Gaussian elimination on message passing architecture. In *International Conference on Supercomputing*, Patras, Grece, June 1987. Springer-Verlag.

[DCDH90] J.J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Transaction on Mathematical Software*, 16(1):1–17, 1990.

[DFT93] F. Desprez, P. Fraigniaud, and B. Tourancheau. Successive broadcasts on hypercube. Technical Report CS-93-210, University of Tennessee, CS dept., August 1993.

[DGW92] J.J. Dongarra, R. Van De Geijn, and D.W. Walker. A Look at Dense Linear Algebra Libraries. Technical Report ORNL/TM-12126, Oak Ridge National Laboratory, July 1992.

[DO90] J.J. Dongarra and S. Ostrouchov. LAPACK Working Note: LAPACK Block Factorization Algorithms on the Intel iPSC/860. Technical Report 24, Department of Computer Science - University of Tennessee, 1990.

[Dun90] T.H. Dunigan. Performance of the Intel iPSC/860 Hypercube. Technical Report TM-11491, Oak Ridge National Laboratory, September 1990.

[EK93] R. Esser and R; Knetcht. Intel Paragon XP/S - Architecture and Software Environment. Technical Report KFA-ZAM-IB-9305, Zentralinstitut fur Angewandte Mathematik - Forschungszentrum Julich, April 1993.

[GL89] G.H. Golub and C.F. Van Loan. *Matrix Computation*. The John Hopkins University Press, 1989. Second edition.

[Int93] Intel Corporation - SSD, Beaverton, OR. *PARAGON OSF/1 - User's guide*, April 1993.

[LC89] G. Li and T.F. Coleman. A New Method for Solving Triangular Systems on Distributed Memory Message Passing Multiprocessors. *SIAM Journal on Science and Statistical Computing*, 10:382–396, 1989.

[LKK83] R.E. Lord, J.S. Kowalik, and S.P. Kumar. Solving Linear Algebraic Equations on a MIMD Computer. *Journal of the ACM*, 30(1):103–117, January 1983.

[PBKP92] B.V. Purushotham, A. Basu, P.S. Kumar, and L.M. Patnaik. Performance Estimation of LU Factorisation on Message Passing Multiprocessors. *Parallel Processing Letters*, 2(1):51–60, 1992.

[Rob90] Y. Robert. *The Impact of Vector and Parallel Architectures on the Gaussian Elimination Algorithm.* Manchester University Press - Manchester - UK, 1990.

[RT88] Y. Robert and B. Tourancheau. Block gaussian elimination on a hypercube vector multiprocessor. *Revista de Mathematicas Aplicadas*, 10:77–89, 1988. Universidad de Chile.

[RTV89] Y. Robert, B. Tourancheau, and G. Villard. Data allocation strategies for the gauss and jordan algorithms on a ring of processors. *Information Processing Letters*, 31:21–29, 1989.

[Saa86a] Y. Saad. Communication Complexity of the Gaussian Elimination Algorithm on Multiprocessors. *Linear Algebra and Applications*, 77:315–340, 1986.

[Saa86b] Y. Saad. Gaussian Elimination on Hypercubes. In M. Cosnard, Y. Robert, P. Quinton, and M. Tchuente, editors, *Parallel Algorithms and Architectures.* North-Holland, 1986.

[SS89] Y. Saad and M.H. Schultz. Data Communication in Parallel Architectures. *Journal of Parallel and Distributed Computing*, 6:115–135, 1989.