Parallel LU Decomposition on a Transputer Network

Rob H. Bisseling and Johannes G. G. van de Vorst
Koninklijke/Shell-Laboratorium, Amsterdam (Shell Research B.V.)
P.O. Box 3003, 1003 AA Amsterdam, the Netherlands

Abstract

A parallel algorithm is derived for LU decomposition with partial pivoting on a local-memory multiprocessor. A general Cartesian data distribution scheme is presented which contains many of the existing distribution schemes as special cases. This scheme is used to prove optimality of load balance for the grid distribution. Experimental results of an implementation of the algorithm in occam-2 on a square mesh of 36 transputers show an efficiency of 88% and a speed of 21.5 Mflop/s for a matrix of size $n = 1000$.

1. Introduction

The local-memory multiprocessor is an important new type of parallel computer. It consists of a number of powerful processors and a communication network. Each processor has its own local memory; communication with other processors is done by message-passing. Examples of this architecture are hypercubes and transputer networks.

The recent availability of parallel computers as a tool in scientific computing has renewed the interest in many long-existing basic algorithms, in particular in the field of linear algebra. Much attention has been paid to the parallelisation of the Gaussian elimination algorithm for the solution of a system of linear equations, and to the parallelisation of the equivalent LU decomposition algorithms.

This paper presents a practical and efficient algorithm that calculates the LU decomposition with partial pivoting of an $n \times n$ matrix A on a two-dimensional mesh of p processors. The efficiency of the algorithm is achieved by balancing the computational work load and by reducing the communication costs. The complexity of the algorithm is $2n^3/3p + n^2/\sqrt{p}$ floating point operations, and $O(n^2/\sqrt{p})$ communications.

A number of parallel LU decomposition algorithms have been proposed for local-memory computers, that are based on row- or column-oriented data distribution schemes {6, 7, 12, 18}. The complexity of these algorithms is approximately $2n^3/3p + O(n^2)$ floating point operations, and at least $O(n^2)$ communications. The efficiency of row- or column-oriented algorithms decreases rapidly with increasing p, due to the growth of the communication cost $O(n^2)$ relative to the computation cost $2n^3/3p$. The range of applicability of these algorithms is therefore limited to $p \ll n$, and to situations where the communication cost is mainly determined by the communication start-up time and not by the length of the messages. The latter

situation exists in some of the currently available hypercubes, which have an extremely high communication start-up penalty.

Good load balancing can be achieved by *cyclic* or *interleaved* assignment of matrix rows (or columns) to processors. In such an assignment row i $(0 \leq i < n)$ is allocated to processor i **mod** p (cf. {6, 7, 12}). Carrying this idea further, it has been proposed {11, 22} to distribute the elements cyclically in *both* coordinates; the resulting *grid distribution* {22}, also called *scattered distribution* {11}, has an even better load balance {22}, and a lower communication complexity as well. The grid distribution has been employed by Fox *et al.* for LU decomposition of banded matrices on a hypercube {11}.

The present paper is based on the work of Ref. {22}. A practical LU decomposition algorithm with explicit partial pivoting is derived and shown correct by use of formal methods, and its performance on a mesh of p transputers is tested.

The remainder of this paper is organised as follows. Section 2 gives a formal derivation of the parallel LU decomposition program. Section 3 introduces the notion of Cartesian clustering, to facilitate the analysis of the load balancing and communication properties of the algorithm. Section 4 presents experimental timing results of an implementation on a square mesh of transputers. Section 5 gives the conclusions.

2. The Parallel LU Decomposition Algorithm

2.1 Introduction

A parallel version of the LU decomposition algorithm with partial pivoting is derived in this section, using invariants {8,14} and the Gries-Owicki theory {19,20}.

The problem of LU decomposition with partial pivoting can be formulated as follows {13}: given a nonsingular $n \times n$ matrix $A = (a_{st}, 0 \leq s, t < n)$, find a permutation $\pi \in S_n$, a unit lower triangular matrix $L = (l_{st}, 0 \leq s, t < n)$, with $l_{st} = 0$ for $s < t$ and $l_{st} = 1$ for $s = t$, and an upper triangular matrix $U = (u_{st}, 0 \leq s, t < n)$, with $u_{st} = 0$ for $s > t$, such that

$$a_{\pi(s),\, t} = (LU)_{st} \quad \text{for all } s,t. \tag{2.1}$$

(Here, and in the sequel, the bounds on s and t are omitted for the sake of brevity.) In addition to this, it is required that

$$|l_{st}| \leq 1 \qquad \text{for } s > t. \tag{2.2}$$

Later on, we shall see that this requirement will lead to the partial pivoting procedure of choosing a maximal pivot element in the pivoting column. Partial pivoting ensures numerical stability.

A reformulation of the problem is obtained by expansion of eqn (2.1) in terms of a matrix $X = (x_{st}, 0 \leq s, t < n)$, with $x_{st} = u_{st}$ for $s \leq t$ and $x_{st} = l_{st}$ for $s > t$, and by use of the triangular properties of L and U. For $s \leq t$,

$$a_{\pi(s),\,t} = \sum_{j=0}^{n-1} l_{sj}u_{jt} = u_{st} + \sum_{j=0}^{s-1} l_{sj}u_{jt} = x_{st} + \sum_{j=0}^{s-1} x_{sj}x_{jt}, \tag{2.3}$$

and for $s > t$,

$$a_{\pi(s),\,t} = \sum_{j=0}^{n-1} l_{sj}u_{jt} = l_{st}u_{tt} + \sum_{j=0}^{t-1} l_{sj}u_{jt} = x_{st}x_{tt} + \sum_{j=0}^{t-1} x_{sj}x_{jt}. \tag{2.4}$$

The new formulation then becomes: given a nonsingular $n \times n$ matrix A, find a permutation π and a matrix X such that

$$x_{st} = a_{\pi(s),\,t} - \sum_{j=0}^{s-1} x_{sj}x_{jt} \qquad \text{for } s \le t, \tag{2.5}$$

$$x_{st}x_{tt} = a_{\pi(s),\,t} - \sum_{j=0}^{t-1} x_{sj}x_{jt} \qquad \text{for } s > t, \tag{2.6}$$

$$|x_{st}| \le 1 \qquad \text{for } s > t. \tag{2.7}$$

For compactness of notation it is convenient to define a partial sum function f. If we fix the matrices A and X, and the permutation π, then f is a function of three arguments:

$$f(s,t,k) = a_{\pi(s),\,t} - \sum_{j=0}^{k-1} x_{sj}x_{jt} \qquad \text{for } 0 \le k \le n. \tag{2.8}$$

2.2 Postcondition and invariants

The method of invariants { 8, 14} solves problems by the use of a formalism of logical expressions. For the present problem this requires the establishment of the *postcondition* R,

$$R \equiv \forall_{s,t} R[s,t], \tag{2.9}$$

where the *local postconditions* $R[s,t]$ are defined by

$$R[s,t] \equiv (x_{st} = f(s,t,s) \wedge s \le t) \vee (x_{st}x_{tt} = f(s,t,t) \wedge |x_{st}| \le 1 \wedge s > t). \tag{2.10}$$

The postcondition suggests, in analogy with Ref. {22}, the use of an *invariant* P,

$$P \equiv \forall_{s,t} P[s,t], \tag{2.11}$$

where the *local invariants* $P[s,t]$ are defined by

$$P[s,t] \equiv (x_{st} = f(s,t,k) \wedge k \le s \wedge s \le t) \vee (x_{st} = f(s,t,s) \wedge k > s \wedge s \le t) \vee$$
$$(x_{st} = f(s,t,k) \wedge k \le t \wedge s > t) \vee (x_{st}x_{tt} = f(s,t,t) \wedge |x_{st}| \le 1 \wedge k > t \wedge s > t) \tag{2.12}$$

An initialisation which establishes P is:

$$k := 0 \ ; \ \pi := \mathrm{id} \ ; \ X := A. \tag{2.13}$$

After that, k will be increased until $k = n$, while simultaneously keeping P valid. This will establish R, since $P \wedge (k = n) \Rightarrow R$. Note that the value of k will depend upon s and t, so $k = k(s,t)$.

2.3 Maintaining the invariant

The invariant P should remain valid after increasing the index k. To achieve this, we shall focus on the local invariants $P[s,t]$, and fix $k = k(s,t)$ in each of them. This reflects the fact that later on we want to perform the operations necessary for each $P[s,t]$ in parallel. Our aim is to determine which operations have to be performed in order to maintain $P[s,t]$ when k is incremented by one.

Before turning our attention to the maintenance of the invariants $P[s,t]$, some terminology is introduced. Let $x_{q\bullet}$ denote the q-th row of X. For fixed k, the pair (s,t) is called *active* if $k < \min(s,t)$; it is called *critical* if $k = t < s$ and it is called *passive* if $k > \min(s,t) \vee (k = s \leq t)$.

For an active pair, a comparison of $f(s,t,k)$ and $f(s,t,k+1)$ in eqn (2.8) shows that the validity of $P[s,t]$ is maintained if the program fragment

$$x_{st} := x_{st} - x_{sk} x_{kt} \ ; \quad k := k + 1 \tag{2.14}$$

is executed, provided $P[s,k]$ and $P[k,t]$ hold.

For a passive pair with $k > \min(s,t)$, incrementing k leaves $P[s,t]$ invariant, since $P[s,t] \equiv R[s,t]$, both for k and $k+1$, and $R[s,t]$ is not dependent upon k. For a passive pair with $k = s \leq t$, incrementing k leaves $P[s,t]$ invariant, since $x_{st} = f(s,t,k) = f(s,t,s)$.

For a critical pair, eqn (2.12) shows that the program fragment

$$x_{st} := x_{st} / x_{kk} \ ; \quad k := k + 1 \tag{2.15}$$

maintains the validity of $P[s,t]$, provided that $P[k,k]$ and the additional initial condition $|x_{kk}| \geq |x_{sk}|$ hold. The initial condition can be established by the execution of the program fragment

$$\pi := \pi \circ (k,r); \qquad \mathrm{swap}\,(x_{k\bullet}, x_{r\bullet}) \tag{2.16}$$

with r such that

$$|x_{rk}| = \max \{ |x_{sk}| : s \geq k \} . \tag{2.17}$$

The proof that this program fragment leaves P invariant is given in Appendix A.

2.4 The algorithm

Combining the program fragments above, we get the complete program text S:

```
S:    for all s,t : 0 ≤ s,t < n par do process (s,t)
(s,t): var x_st, π, k;
       begin
          x_st := a_st ; π := id ; k := 0;
          { P[s,t] }
          while k < n do
          var r;
          begin
             { P[s,t] }
             find r such that |x_rk| = max{|x_sk| : s ≥ k};
             broadcast r to all processes;
             π := π ∘ (k,r);
             swap (x_k•, x_r•);
             { |x_kk| = max{|x_sk| : s ≥ k} ∧ P[s,t] }
             if k < min(s,t) then
             begin
                par begin
                   receive x_sk from process (s,k);
                   receive x_kt from process (k,t)
                par end;
                x_st := x_st − x_sk x_kt
             end
             else if k = t < s then
             begin
                receive x_kk from process (k,k);
                x_st := x_st / x_kk ;
                send x_sk to all processes (s,q) with q > k
             end
             else if k = s ≤ t then
                send x_kt to all processes (q,t) with q > k
             else if k > min(s,t) then skip;
             k := k + 1
             { P[s,t] }
          end
       end.
```

(Details of the pivot search, broadcast and swap operations are omitted.)

3. *Analysis*

3.1 Cartesian clustering

In many practical situations, full exploitation of parallelism is impossible. This may be due to a limit on the number of processors available, or to the fact that the communication costs overtake the computation costs. As a consequence, compu-

tations that could in principle be performed in parallel have to be grouped in *clusters*. For each cluster there is a corresponding process that maintains the local invariants of that cluster. In terms of matrices, this means that their elements are distributed, and that each process has local variables for the representation of one part of a matrix. There are p processes, running simultaneously on p processors; each processor is responsible for the execution of exactly one process, the operations of which are performed sequentially.

A clustering scheme is efficient for a particular algorithm on a particular network of processors if: (i) it has good load balancing properties, meaning that the computations are spread evenly across the processors, (ii) it is not communication bound.

A number of data clustering schemes have been proposed for linear algebra algorithms on various multiprocessors, such as ring, mesh, torus, and hypercube networks. Clustering of matrix elements can be done in: blocks {4} (i.e., "consecutive storage" {10, 16}); grids {22} (i.e., "cyclic storage" {10, 16} or "scattered square decomposition" {5, 11}); contiguous or scattered rows {6, 15, 21}; dynamically allocated rows {12}; columns {15}; and interleaved blocks {21}. All of these schemes are special cases of the general Cartesian clustering scheme presented below. It will be shown that grid clustering gives an optimal load balance for the present algorithm, and a sufficiently low communication complexity. The choice of clustering has significant influence on the efficiency: e.g. it has been shown {22} that the use of block clustering instead of grid clustering degrades the performance of the algorithm by a factor of three, because of poor load balancing.

In the following analysis, a Cartesian clustering will be seen as a Cartesian product of two partitions of a finite set V. A *partition* of V is a collection $\{V_i : 0 \leq i < N\}$ of mutually disjoint, non-empty subsets $V_i \subset V$, such that $\bigcup_{i=0}^{N-1} V_i = V$. The size of the partition is N, and $|V_i|$ denotes the cardinality of V_i, $0 \leq i < N$.

Let $V = \{s : 0 \leq s < n\}$, where n equals the matrix size. It is our aim to find the best clustering of $V \times V$ for the present algorithm. A *Cartesian clustering* C of $V \times V$ has the form: $C = \{V_i \times W_j : 0 \leq i < M \wedge 0 \leq j < N\}$, where $\{V_i : 0 \leq i < M\}$ and $\{W_j : 0 \leq j < N\}$ are two partitions of V, of size M and N, respectively. The sets $V_i \times W_j$ are called *clusters*. The *size* of C is defined as the ordered pair (M,N). Because each cluster corresponds uniquely to one processor, the total number of clusters equals $p = MN$.

The *grid clustering* G of $V \times V$ of size (M,N) is a special kind of Cartesian clustering:

$$G = \{G_i \times H_j : 0 \leq i < M \wedge 0 \leq j < N\}, \tag{3.1}$$

with

$$G_i = \{s \in V : s \bmod M = i\}, \quad \text{for } 0 \leq i < M, \tag{3.2}$$

$$H_j = \{t \in V : t \bmod N = j\}, \quad \text{for } 0 \leq j < N. \tag{3.3}$$

3.2 Load balancing

Let us consider the load balancing properties of program S using an arbitrary

Cartesian clustering $C = \{V_i \times W_j : 0 \leq i < M \wedge 0 \leq j < N\}$ of the set $V \times V$. A measure *Ncomp*(*C*) for the *computation complexity* of the clustering *C* is given by the expression

$$Ncomp(C) = \sum_{k=0}^{n-1} \max\{N(c,k) : c \in C\}, \tag{3.4}$$

where $N(c,k)$ denotes the number of computations in cluster *c* that have to be performed during phase *k* of the total computation. Phase *k* is defined as the part of the computation between the *k*-th and the $k+1$-th pivot operation, i. e. the part which updates the x_{st}. The number $N(c,k)$ is approximately proportional to the computation time in phase *k* of processor *c*. This approximation is valid because nearly all the computations take the same form (see (2.14)),

$$x_{st} := x_{st} - x_{sk}x_{kt}, \tag{3.5}$$

each computation involving two floating point operations. In the definition of *Ncomp*(*C*) it is assumed that the comparison operations of the pivot search and the division operations of program fragment (2.15) do not contribute significantly to the computation time. Note that the measure does take into account that processes may have to wait for the (intermediate) results from other ones. The communication costs are *not* included in *Ncomp*(*C*); these will be treated in the next subsections.

Let $U_k = \{s : k < s < n\}$, so that $U_k \times U_k$ equals the set of active pairs. This set is exactly the set of pairs for which a computation of the form (3.5) has to be performed. Therefore, the expression *Ncomp*(*C*) can be written as:

$$\begin{aligned} Ncomp(C) &= \sum_{k=0}^{n-1} \max\{N(c,k) : c \in C\} \\ &= \sum_{k=0}^{n-1} \max\{|(V_i \times W_j) \cap (U_k \times U_k)| : 0 \leq i < M \wedge 0 \leq j < N\} \\ &= \sum_{k=0}^{n-1} \max\{|(V_i \cap U_k)| : 0 \leq i < M\} \cdot \max\{|(W_j \cap U_k)| : 0 \leq j < N\}. \end{aligned} \tag{3.6}$$

The grid clustering is optimal among Cartesian clusterings from the point of view of load balancing: if *C* is an arbitrary Cartesian clustering of $V \times V$, with size (M,N), and *G* is the grid clustering of $V \times V$ of the same size, then

$$Ncomp(C) \geq Ncomp(G). \tag{3.7}$$

PROOF.

$$Ncomp(C) = \sum_{k=0}^{n-1} \max\{|(V_i \cap U_k)| : 0 \leq i < M\} \cdot \max\{|(W_j \cap U_k)| : 0 \leq j < N\}$$

$$\geq \sum_{k=0}^{n-1} \left\lceil \frac{|U_k|}{M} \right\rceil \cdot \left\lceil \frac{|U_k|}{N} \right\rceil = \sum_{k=0}^{n-1} \left\lceil \frac{n-k-1}{M} \right\rceil \cdot \left\lceil \frac{n-k-1}{N} \right\rceil$$

$$= \sum_{k=0}^{n-1} \max\{|(G_i \cap U_k)| : 0 \leq i < M\} \cdot \max\{|(H_j \cap U_k)| : 0 \leq j < N\}$$

$$= Ncomp(G). \tag{3.8}$$

END OF PROOF.

A lower bound on the total computation time for any clustering, of arbitrary size, is obtained by dividing the total number of computations by p,

$$Ncomp(C) \geq \frac{1}{p} \sum_{k=0}^{n-1} (n-k-1)^2 = \frac{n(n-1/2)(n-1)}{3p} = \frac{n^3}{3p} - \frac{n^2}{2p} + O(n). \tag{3.9}$$

An upper bound for the grid clustering of size (M,N) is given by

$$Ncomp(G) \leq \sum_{k=0}^{n-1} \left(\frac{n-k-1}{M} + 1 \right) \cdot \left(\frac{n-k-1}{N} + 1 \right)$$

$$= \frac{n(n-1/2)(n-1)}{3p} + \frac{n(n-1)(M+N)}{2p} + n. \tag{3.10}$$

This upper bound is minimal if the grid clustering is square, $M = N = \sqrt{p}$. From the upper bound on the computation complexity, a simple upper bound on the computation time for square grid clustering can be obtained,

$$T_{\text{comp}}(G) \leq 2t_{\text{flop}} \left(\frac{n^3}{3p} + \frac{n^2}{2\sqrt{p}} + O(n) \right). \tag{3.11}$$

Here t_{flop} is the time needed to perform one floating point computation such as a multiplication or an addition. Two such operations are needed for each computation of the form (3.5). Comparison of this upper bound with the time of the sequential algorithm, $2t_{\text{flop}}n^3/3$, shows that a good load balance is obtained by square grid clustering, for values of p up to the order of n^2. The point of 50% efficiency loss due to load imbalance is reached at the value of $p = 4n^2/9$.

The grid clustering of size (M,N) with $M = 1$ $(N = 1)$ is simply the interleaved column (row) clustering. An exact count shows that its complexity equals $n^3/3p + O(n^2)$, so that the load imbalance is of a higher order than the load imbalance of the square grid clustering.

3.3 Communication on a full network

In this subsection, we shall determine the number of communications that are required by program S, for a Cartesian clustering C. Because we do not yet want to

restrict ourselves to one particular hardware network, it is assumed that a full communication network is available. The total number of communications is counted. The particular hardware of the communication network determines the number of communications that can be done in parallel, and the actual number of communication steps that are needed to route data from source to destination (this might be more than one, if the network is not full).

The number of communications $Ncom_1(C)$ needed to find the index r of the pivot row is found as follows: The pivot element $|x_{rk}|$ is defined by eqn (2.17), $|x_{rk}| = \max\{|x_{sk}| : s \geq k\}$, for $0 \leq k < n$. Since the set $\{|x_{sk}| : s \geq k\}$ is distributed across at most M clusters, not more than $M - 1$ communications are needed for the computation of r. Hence

$$Ncom_1(C) \leq n\,(M-1). \tag{3.12}$$

The number of communications $Ncom_2(C)$ needed to broadcast r and to interchange rows r and k, is determined as follows: First, the value of r has to be broadcasted to all processes, so that they can perform the assignment $\pi := \pi \circ (k,r)$. This gives rise to $MN - 1$ communications. The operation swap $(x_{k\bullet}, x_{r\bullet})$ requires either 0 communications (if the rows $x_{k\bullet}$ and $x_{r\bullet}$ are contained in the same cluster), or $2n$ communications (if they are not). Thus, an upper bound for $Ncom_2(C)$ is

$$Ncom_2(C) \leq n\,(MN-1) + 2n^2. \tag{3.13}$$

The number of communications $Ncom_3(C)$ needed to update the local invariants is counted as follows: The set of values needed in phase k is $\{x_{sk} : s > k\} \cup \{x_{kt} : t \geq k\}$. A value x_{sk} is needed in at most N clusters (the clusters that contain row s) and therefore it has to be communicated at most $N - 1$ times. Employing a similar argument for values x_{kt}, an upper bound for $Ncom_3(C)$ is obtained:

$$\begin{aligned} Ncom_3(C) &\leq \sum_{k=0}^{n-1} (N-1)(n-k-1) + \sum_{k=0}^{n-1} (M-1)(n-k) \\ &= \frac{n\,(n+1)(M+N-2)}{2} - n\,(N-1). \end{aligned} \tag{3.14}$$

The total number of communications is

$$Ncom\,(C) = \sum_{i=1}^{3} Ncom_i\,(C) \leq \frac{n^2}{2}(M+N+2) + \frac{n}{2}(2MN+3M-N-4). \tag{3.15}$$

Under the constraint $MN = p$, this bound attains a minimum for $M = N = \sqrt{p}$. In that case we get, approximately,

$$Ncom\,(C) \leq n^2\sqrt{p} + np. \tag{3.16}$$

3.4 Communication on a square mesh

The actual number of communication steps $Ncom$ of the LU decomposition algorithm is dependent on the topology of the hardware network, and on the mapping

of the clusters onto the processors. In this subsection we count the number of communication *steps*, instead of the number of communications, since some communications may be performed simultaneously in one step. The topology of the network is assumed to be a two-dimensional mesh. This is a sufficiently general network, and is embedded in many networks, such as e.g. a torus and a hypercube. (Transputers have four bi-directional links, and can therefore be configured in a two-dimensional mesh.) Because of the reduced bound (3.16) on the total number of communications, the mesh is chosen to be square. The count for grid clustering on a square mesh with the natural cluster-processor mapping is similar to the analysis of the previous subsection.

First,

$$Ncom_1(G) \leq n(\sqrt{p} - 1)/2, \tag{3.17}$$

because r can be obtained by comparing and communicating values of $|x_{sk}|$ in a processor column of length $\sqrt{p}$, finally obtaining the result in the middle processor of the column.

Second,

$$Ncom_2(G) \leq n(\sqrt{p} - 2) + n\left(\sqrt{p} - 1 + \left\lceil \frac{n}{\sqrt{p}} \right\rceil - 1\right)$$

$$\leq \frac{n^2}{\sqrt{p}} + n(2\sqrt{p} - 3). \tag{3.18}$$

A broadcast of r, first horizontally and then in parallel vertically, costs $\sqrt{p} - 2$ steps, giving the first term of (3.18). A row swap can be done in $2\sqrt{p}$ independent vertical pipelines. The start-up time of the pipeline is $\sqrt{p} - 1$, and the additional completion time is $\lceil n/\sqrt{p} \rceil - 1$.

Third, the number of communication steps needed for the updating procedure is

$$Ncom_3(G) \leq \sum_{k=0}^{n-1} \left(\sqrt{p} - 1 + \left\lceil \frac{n-k}{\sqrt{p}} \right\rceil - 1\right) \leq \frac{n^2}{2\sqrt{p}} + n\sqrt{p}. \tag{3.19}$$

There are $\sqrt{p}$ horizontal and $\sqrt{p}$ vertical communication pipelines, which are proceeding simultaneously. The vertical ones involve $n - k$ data elements; they have a start-up time of $\sqrt{p} - 1$ and an additional completion time of $\lceil (n-k)/\sqrt{p} \rceil - 1$. Similar for the horizontal pipelines, which involve one data element less.

The total communication complexity is

$$Ncom(G) \leq \frac{3n^2}{2\sqrt{p}} + \frac{7n}{2}(\sqrt{p} - 1). \tag{3.20}$$

Note that this count is valid for the square grid clustering, but not necessarily for other Cartesian clusterings.

The total communication time on a square mesh is

$$T_{comm}(G) = t_{comm}(\frac{3n^2}{2\sqrt{p}} + O(n\sqrt{p})). \quad (3.21)$$

Here t_{comm} is the time needed to communicate one floating point value to a neighbouring processor. Comparison of eqns (3.21) and (3.11) shows that the point of 50% efficiency loss due to communication overhead is reached at the value of $p = (4t_{flop}/9t_{comm})^2 n^2$.

In conclusion, the communication complexity for the grid clustering is sufficiently low; it is of the same order as the load imbalance of the algorithm; both are of a lower order than the computation complexity.

4. Results

The LU decomposition algorithm has been implemented in the parallel programming language occam-2 {3}, and executed on a square mesh of T800-20 transputers, each with 256 Kbyte of local memory, and communicating with each other at a maximum link speed of 10 Mbit/s. (This maximal rate is the rate for communication of large packets of data.) Timing results were obtained for meshes of size $p = 1, 4, 9, 16, 25,$ and 36 processors, and for matrices of size up to $n = 1200$.

Work distribution was done by grid clustering; communication by independent pipelines, as described in subsection 3.4. Performance was maximised by using on-chip memory (4 kByte) as much as possible {1}. Computations were done in single precision (32 bits).

The test matrices were constructed in such a way that a fair amount of pivoting was necessary: half of the n steps of the algorithm included a swap of two rows. These rows were contained in processor rows at an average distance of $1/2\sqrt{p}$ in the processor network, i.e. at a distance of half the mesh size.

The timing measurements were obtained by using an internal timer calibrated with a wallclock. Timings of the parallel algorithm were compared with timings of the sequential algorithm programmed in occam-2 and run on a single transputer.

Table 1 shows the time $T_p(n)$ of the LU decomposition of a matrix of size $n \times n$, on a network of p transputers. The size of the problem that can be solved grows with the number of processors available. Since our transputer network contains 256 kByte of memory per processor, a single transputer can maximally store a matrix of size 200×200.

The results of Table 1 show that large matrices can be decomposed fast: e. g. a 1000×1000 matrix was decomposed in 31 seconds. The overhead paid for parallelism is small, as can be seen by comparing the results of the parallel algorithm with $p = 1$ with the results of the sequential algorithm. Already for a small matrix size, $n = 200$, a speed-up of a factor 15 compared to the sequential algorithm was obtained, using 36 processors. (The intended range of applications consists mainly of matrices of size larger than this.)

Table 1.

n	$p=1$ (seq)	$p=1$ (par)	$p=4$	$p=9$	$p=16$	$p=25$	$p=36$
50	0.16	0.17	0.08	0.06	0.05	0.05	0.04
100	1.06	1.09	0.39	0.25	0.18	0.15	0.13
200	7.8	7.9	2.4	1.3	0.86	0.65	0.52
300			7.3	3.8	2.4	1.7	1.3
400			16.5	8.2	5.0	3.6	2.7
600				25.6	15.2	10.5	7.7
800					34.2	23.1	16.7
1000						43.1	31.0
1200							51.7

Timings of LU decomposition with partial pivoting on a transputer network. $T_p(n)$ = the time (in s) on p processors for an $n \times n$ matrix.

Table 2.

n	$p=1$ (seq)	$p=1$ (par)	$p=4$	$p=9$	$p=16$	$p=25$	$p=36$
50	0.52	0.49	1.1	1.3	1.7	1.8	1.9
100	0.63	0.61	1.7	2.7	3.7	4.4	5.0
200	0.68	0.68	2.3	4.1	6.2	8.2	10.2
300			2.5	4.8	7.6	10.5	13.5
400			2.6	5.2	8.5	11.9	15.9
600				5.6	9.5	13.8	18.7
800					10.0	14.8	20.5
1000						15.5	21.5
1200							22.2

Megaflop rates of LU decomposition with partial pivoting on a transputer network. $R_p(n)$ = the speed (in Mflop/s) on p processors for an $n \times n$ matrix.

Table 2 presents the computational speed $R_p(n)$ of the algorithm in million floating point operations per second (Mflop/s); it was computed from Table 1. Comparison of the maximal speeds of $p = 1$ (sequential) and $p = 36$ shows a speed-up of a factor 32, for matrices of size $n \geq 1000$. This means that a close-to-maximum speed can be sustained even for large numbers of processors, as long as the problem size n stays large enough.

Figure 1 shows the processor utilisation during parallel LU decomposition. Each curve shows the efficiency $E_p(n)$,

$$E_p(n) = \frac{R_p(n)}{pR_{\text{seq,max}}} \tag{4.1}$$

for various matrices of size n on a fixed mesh of p processors. Here $R_{\text{seq,max}}$ is the highest speed obtainable by the sequential algorithm on a single processor; in the present case this is 0.68 Mflop/s.

The results of Figure 1 show that over 90% efficiency can be achieved for all mesh sizes p, as long as the problem is large enough. The point of 50% efficiency is reached for matrix size 250 on a 36 processor mesh, and earlier for smaller meshes.

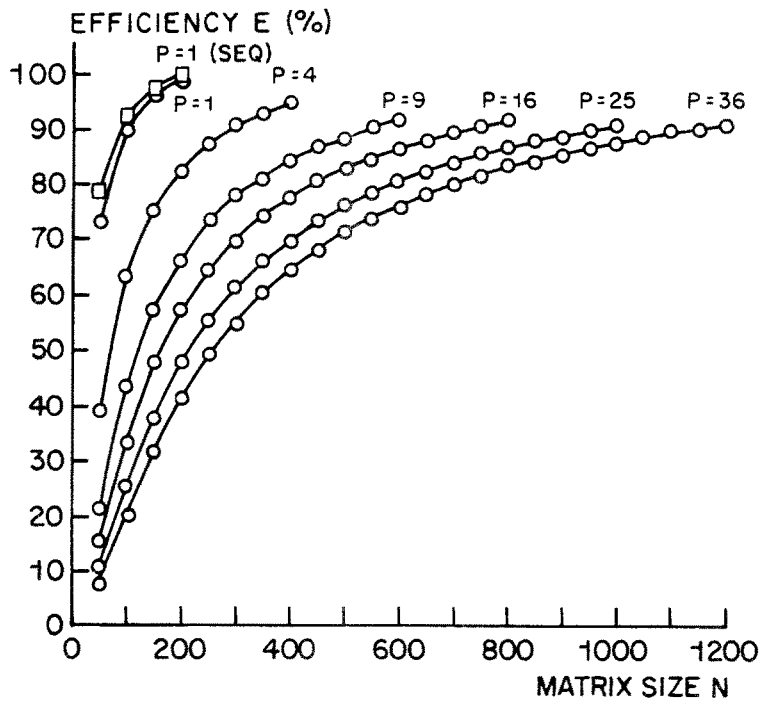

Figure 1.
Efficiency of LU decomposition with partial pivoting on a transputer network.
$E_p(n)$ = the efficiency (in %) on p processors for an $n \times n$ matrix.

The experimental timings were used in a least-squares fit to the theoretical performance model, expressed by eqns (3.11) and (3.21). The resulting empirical timing formula is

$$T_p(n) \simeq 1.31\frac{2n^3}{3p} + 40.2\frac{n^2}{\sqrt{p}} \ \mu\text{s}. \tag{4.2}$$

The second term of eqn (4.2) contains communication overhead (eqn (3.21)), load imbalance (eqn (3.11)), comparisons to find the maximum pivot element in a column, and divisions by the pivot elements. All of these take a time of the order $n^2/\sqrt{p}$. The communication overhead is dependent on the actual amount of pivot swaps executed.

In solving a system of linear equations $Ax = b$, the decomposition $A = LU$ is usually followed by the solution of the triangular systems $Ly = b$ and $Ux = y$. An efficient parallel algorithm has been developed for triangular system solving which is based on grid clustering; it is therefore compatible with the preceding grid-based LU decomposition. A detailed description of the algorithm and its analysis can be found elsewhere {2, 17}. The complexity of this algorithm is $n^2/p + O(n)$ floating point operations, and $O(n)$ communications. The experimental timing formula is

$$T_p(n) \simeq 1.89\frac{n^2}{p} + 38n \ \mu s. \tag{4.3}$$

Timings obtained were e.g. 0.0039 s for solution of a triangular matrix of size $n = 100$, and 0.091 s for $n = 1000$, on the same mesh of 36 T800-20 transputers as above.

5. Conclusion

The concept of Cartesian clustering is central to parallel linear algebra: most of the clustering schemes known to us are Cartesian. The availability of such a general scheme gives a powerful tool for the analysis of parallel algorithms, as demonstrated in the optimality proof of grid clustering for LU decomposition.

Interleaving is the key to obtaining good load balance in LU decomposition. For example, interleaving rows by assigning them to processors 0, 1, and 2 by the pattern 012012... will keep all processors busy almost until the end of the computation. Without interleaving computation time is unnecessarily long, e.g. in block (submatrix) clustering $T_{comp}(B) \simeq 2n^3/p$, whereas with interleaved row clustering this is only $T_{comp}(R) \simeq 2n^3/3p + n^2$, thereby gaining a factor 3 and attaining maximum speedup, at least asymptotically (for $n \to \infty$). The *load imbalance term* n^2 is for many purposes too large; it is independent of p and soon becomes the computational bottleneck when increasing p. It can be further reduced by double interleaving: interleaving both rows and columns gives the grid clustering with $T_{comp}(G) \simeq 2n^3/3p + n^2/\sqrt{p}$.

The communication time of the algorithm depends upon the hardware topology and on the speed of the communication links. A ring topology is not sufficiently rich for LU decomposition, since the communication time is of the order $O(n^2)$. A mesh is ideally suited, since the communication time is only of the order $O(n^2/\sqrt{p})$, which is the order of the load imbalance. The even richer topology of the hypercube cannot be fully exploited, since losses due to load imbalance will be the limiting factor of performance.

In the sequential case, the time of linear system solving is almost completely determined by the time of LU decomposition, except for very small matrices, since the time complexity of triangular system solution is of a lower order than the com-

plexity of LU decomposition. The same holds for the parallel case, provided grid clustering is used, compare eqns (4.2) and (4.3).

The LINPACK benchmark {9} is a measure of computational speed in linear system solving; it is the time needed to perform one LU decomposition and two triangular system solutions. The $n = 100$ benchmark for a 36 transputer system is 0.138 s, corresponding to an actual performance rate of 4.8 Mflop/s. The $n = 1000$ benchmark is 31.2 s, corresponding to a rate of 21.3 Mflop/s. All benchmarks are for single precision (32 bits) computations.

More important than a single benchmark of an implementation of an algorithm is the scaling behaviour with an increase in the number of processors. Formula (4.2) shows that the LU decomposition implementation on a transputer mesh scales well. The formula can be used to predict the performance of larger transputer networks: For example, to decompose a 1000 × 1000 matrix at a speed of 100 Mflop/s a mesh of 225 transputers would be needed; the efficiency of the computation would be about 60%.

Acknowledgements

We would like to thank our colleagues Ico van den Born, Daniël Loyens, and Theo Verheggen for their useful remarks on the initial version of this manuscript.

References

1. P. Atkin, "Performance Maximisation," INMOS Technical Note 17, Bristol (1987).

2. R. H. Bisseling and J. G. G. van de Vorst, "Parallel Triangular System Solving on a Mesh Network of Transputers," submitted for publication (1988).

3. A. Burns, "Programming in occam-2," Addison-Wesley (1988).

4. P. R. Capello, "Gaussian Elimination on a Hypercube Automaton," *J. Parallel Distrib. Comput.* **4** (1987) 288-308.

5. H. Y. Chang, S. Utku, M. Salama, and D. Rapp, "A Parallel Householder Tridiagonalization Stratagem using Scattered Square Decomposition," *Parallel Comput.* **6** (1988) 297-311.

6. E. Chu and A. George, "Gaussian Elimination with Partial Pivoting and Load Balancing on a Multiprocessor," *Parallel Comput.* **5** (1987) 65-74.

7. G. J. Davis, "Column LU Factorization with Pivoting On a Message-passing Multiprocessor," *SIAM J. Alg. Discr. Meth.* **7** (1986) 538-550.

8. E. W. Dijkstra, "A Discipline of Programming," Prentice-Hall (1976).

9. J. J. Dongarra, "The LINPACK Benchmark: An Explanation," *Proc. 1st Int. Conf. on Supercomputing 1987*, Lecture Notes in Computer Science, Vol. 297, Springer (1988).

10. J. J. Dongarra and L. Johnsson, "Solving Banded Systems on a Parallel Processor," *Parallel Comput.* **5** (1987) 219-246.

11. G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker, "Solving Problems on Concurrent Processors," Vol. 1, Prentice-Hall, Englewood Cliffs, NJ (1988).

12. G. A. Geist and C. H. Romine, "LU Factorization Algorithms on Distributed-Memory Multiprocessor Architectures," *SIAM J. Sci. Statist. Comput.* **9** (1988) 639-649.

13. G. H. Golub and C. F. van Loan, "Matrix Computations," Johns Hopkins University Press, Baltimore (1983).

14. D. Gries, "The Science of Programming," Springer (1981).

15. I. C. F. Ipsen, Y. Saad, and M. H. Schultz, "Complexity of Dense-Linear-System Solution on a Multiprocessor Ring," *Linear Algebra Appl.* **77** (1986) 205-239.

16. S. L. Johnsson, "Communication Efficient Basic Linear Algebra Computations on Hypercube Architectures," *J. Parallel Distrib. Comput.* **4** (1987) 133-172.

17. L. D. J. C. Loyens and R. H. Bisseling, "The Formal Construction of a Parallel Triangular System Solver," *Proc. Int. Conf. on Mathematics of Program Construction 1989*, Lecture Notes in Computer Science, Springer (1989).

18. J. M. Ortega and C. H. Romine, "The *ijk* Forms of Factorization Methods II. Parallel Systems," *Parallel Comput.* **7** (1988) 149-162.

19. S. Owicki and D. Gries, "Verifying Properties of Parallel Programs: An Axiomatic Approach," *Comm. ACM* **19** (1976) 279-285.

20. S. Owicki and D. Gries, "An Axiomatic Proof Technique for Parallel Programs I," *Acta Inform.* **6** (1976) 319-340.

21. Y. Saad and M. H. Schultz, "Parallel Direct Methods for Solving Banded Linear Systems," *Linear Algebra Appl.* **88/89** (1987) 623-650.

22. J. G. G. van de Vorst, "The Formal Development of a Parallel Program Performing LU-Decomposition," *Acta Inform.* **26** (1988) 1-17.

Appendix A. Invariance of P under row interchange

LEMMA. Let k, q, r, and t be fixed; $0 \leq k, q, r, t < n$; $k \leq q, r$. Assume that $P[q,t]$ and $P[r,t]$ are valid, and that $k = k(q,t) = k(r,t)$. Let S_1 be the program fragment:

$$S_1: \qquad \pi := \pi \circ (q,r); \qquad \text{swap}(x_{q\bullet}, x_{r\bullet}). \tag{A.1}$$

Then $P[q,t]$ and $P[r,t]$ will still be valid after the execution of S_1.

PROOF OF LEMMA. Assume that $k, q, r,$ and t are fixed; $0 \leq k, q, r, t < n$; $k \leq q, r$; $k = k(q,t) = k(r,t)$. Assume also that $P[q,t]$ and $P[r,t]$ are valid. For $s = q$ or $s = r$,

$$P[s,t] \equiv \begin{cases} x_{st} = f(s,t,k) & \text{if } k \leq t, \\ x_{st}x_{tt} = f(s,t,t) \wedge |x_{st}| \leq 1 & \text{if } k > t. \end{cases} \tag{A.2}$$

The validity of $P[s,t]$ after the execution of S_1 will only be shown for $k \leq t$. The case $k > t$ can be treated similarly. In the following, primed symbols represent variables, functions, and assertions before execution of S_1, and non-primed those after execution. Without loss of generality it may be assumed that $s = q$.

$$x_{qt} = x'_{rt} = \text{(because of } P'[r,t] \wedge k \leq t\text{)}$$

$$= f'(r,t,k) = a_{\pi'(r),t} - \sum_{j=0}^{k-1} x'_{rj}x'_{jt} = a_{\pi' o(q,r)(q),t} - \sum_{j=0}^{k-1} x'_{rj}x'_{jt}$$

$$= a_{\pi(q),t} - \sum_{j=0}^{k-1} x_{qj}x_{jt} = f(q,t,k). \tag{A.3}$$

This proves that $P[q,t]$ is true. *END OF PROOF.*