SIAM J. SCI. STAT. COMPUT.
Vol. 12, No. 4, pp. 787-799, July 1991
© 1991 Society for Industrial and Applied Mathematics
004

PARALLEL TRIANGULAR SYSTEM SOLVING ON A MESH NETWORK OF TRANSPUTERS*

ROB H. BISSELING† AND JOHANNES G. G. VAN DE VORST†

Abstract. A parallel algorithm is presented for triangular system solving on a distributed-memory MIMD computer with a square mesh topology. The algorithm is based on the square grid (scattered) distribution of matrix elements across the processors. The theoretical time complexity is $n^2/p + O(n)$, for p processors and an $n \times n$ matrix. Experimental timings of an implementation in occam 2 on a square mesh of $p = 36$ transputers confirm the theoretical time model. The scaled speedup achieved for $n = 1200$ is 24 on a 36 transputer mesh. This corresponds to a computing rate of 11.7 Mflop/s.

Key words. parallel algorithms, triangular systems, transputers

AMS(MOS) subject classifications. 65F05, 65W05

1. Introduction. This paper presents a parallel algorithm for the solution of the lower triangular system

$$Lx = b \tag{1.1}$$

on a distributed-memory MIMD computer with a square mesh topology. The matrix $L = (L_{ij}, 0 \leqq i, j < n)$ is a dense $n \times n$ unit lower triangular matrix ($L_{ij} = 0$ for $i < j$, and $L_{ii} = 1$), and $x = (x_i, 0 \leqq i < n)$ and $b = (b_i, 0 \leqq i < n)$ are vectors of length n.

The solution of triangular systems is an important part of scientific computing: triangular solvers are often included in linear system solvers, and these in turn lie at the heart of many scientific codes. In certain situations, the portion of overall computing time consumed by the triangular solver may be large. Developing an efficient parallel triangular solver for a distributed-memory MIMD computer is a particularly difficult task, because it is very important to limit the amount of communication between the processors and to spread the work load evenly among them.

Recently, much effort has been devoted to the development of parallel triangular system solvers for distributed-memory MIMD computers [4], [5], [8]-[12], [15], [17]. These solvers can be divided into two classes, on the basis of their data distribution scheme. The first class of solvers [4], [8], [10], [11], [15] assumes a row- or column-oriented distribution of matrix elements across the processors. For a review of these solvers and an experimental comparison on a hypercube, see Heath and Romine [8].

The second class of solvers [5], [9], [12], [17] assumes the *square grid distribution* [16], defined by assigning matrix element L_{ij} to processor (i **mod** Q, j **mod** Q) of a $Q \times Q$ mesh. The algorithm described in the present paper falls into this class. The square grid distribution was introduced by Fox et al. [5] as *scattered square decomposition*, and by Johnsson [9] as *cyclic storage*. Fox et al. [5], [17] present an algorithm that is intended for banded triangular systems with multiple right-hand sides b. Asymptotically (for large problems) it achieves a full speedup of Q^2 on Q^2 processors, provided the number of right-hand sides is a multiple of Q. On the other hand, for a single right-hand side it only achieves an asymptotic speedup of Q. Our algorithm improves on this by achieving an asymptotic speedup of Q^2, for a single right-hand side. Johnsson [9] discusses the complexity of an algorithm which is similar to our algorithm, except for the fact that his algorithm assumes a hypercube topology, and

* Received by the editors October 24, 1988; accepted for publication (in revised form) May 22, 1990.
† Koninklijke/Shell-Laboratorium, Amsterdam, P.O. Box 3003, 1003 AA Amsterdam, the Netherlands.

makes use of spanning trees to broadcast newly computed components of x. A formal derivation of our algorithm for a completely connected network (the "QWERTY" algorithm) has been published elsewhere [12].

In many situations, the data distribution for the triangular solve cannot be chosen freely, because the data are already distributed as a result of previous calculations, and the cost of redistribution would be higher than the cost of the triangular solve itself. Our motivation in developing the algorithm described in this paper was the desire to solve triangular systems with L and U factors in square grid distribution, since this is the optimal distribution for the LU decomposition [1], [16]. Combining the parallel program for the decomposition $A = LU$ with the parallel programs solving the triangular systems $Lx = b$ and $Uy = x$, all with the square grid distribution, yields an efficient parallel program solving the linear system $Ay = b$.

It turns out that our grid-oriented algorithm for triangular system solving has good load balancing properties and a low communication overhead. A theoretical analysis shows that the computation complexity of the algorithm is $n^2/Q^2 + 5n$ floating point operations, and that the communication complexity is $4n$ neighbour-to-neighbour communications.

2. Outline of the algorithm. The purpose of this section is to establish the notation, state the assumptions, and present an outline of the parallel triangular system solving algorithm. The algorithm consists of the parallel composition of $p = Q^2$ processes (s, t), $0 \leqq s,\ t < Q$, each executing on one processor, denoted by (s, t). To simplify the exposition of the algorithm, it is assumed that $n \ \mathbf{mod}\ Q = 0$. Each processor has a local memory. The processors use a square mesh communication network without wrap-around links to pass messages between them. Two processors, (s, t) and (s', t'), $0 \leqq s,\ s',\ t,\ t' < Q$, are therefore able to communicate if and only if $|s - s'| + |t - t'| = 1$. The communication mechanism may be either synchronous or asynchronous. The only assumption made is that message order is preserved between pairs of communicating processors. To simplify the following program text, it is assumed that communications are safeguarded against the crossing of mesh boundaries. For example, a processor (s, t) with $s = 0$ executes the statement "send y to process $(s-1, t)$" as "skip." Similarly, it executes the statement "receive y from process $(s-1, t)$" as "$y := 0$."

The vector $\hat{x}$ denotes the unique solution of (1.1). It satisfies

$$\hat{x}_i = b_i - \sum_{j=0}^{i-1} L_{ij}\hat{x}_j, \qquad 0 \leqq i < n, \tag{2.1}$$

because L is unit lower triangular. The matrix L is distributed across the processes according to the square grid distribution, which assigns element L_{ij}, $0 \leqq i,\ j < n$ to process $(s, t) = (i \ \mathbf{mod}\ Q, j \ \mathbf{mod}\ Q)$. Define an $n \times Q$ matrix $\hat{w}$, whose elements are the partial sums

$$\hat{w}_{it} = \sum_{\substack{j=0 \\ j \ \mathbf{mod}\ Q = t}}^{i-1} L_{ij}\hat{x}_j, \qquad 0 \leqq i < n, \qquad 0 \leqq t < Q. \tag{2.2}$$

Because of the square grid distribution of L, process (s, t) can compute the partial sums $\hat{w}_{it}$ with $0 \leqq i < n$ and $i \ \mathbf{mod}\ Q = s$ locally without any communication of matrix elements L_{ij}, and hence only with the communication of values $\hat{x}_j$.

A variable vector x is manipulated by the program until it contains the solution $\hat{x}$. The distinction between the variable vector x and the constant vector $\hat{x}$ enables us to make assertions about computed or communicated values, such as "$x_i = \hat{x}_i$." It is

convenient to distribute the vectors x and b in the same way as the diagonal of L, i.e., elements x_i and b_i are assigned to process (i **mod** Q, i **mod** Q). An $n \times Q$ variable matrix w is used as working space to compute $\hat{w}$. This matrix is distributed in the obvious way, by assigning w_{it} to process (i **mod** Q, t). The distribution of L, w, b, and x is illustrated in Fig. 1.

The constants $\hat{x}$ and $\hat{w}$ are used only in assertions; they do not occur in the program text itself. The matrix L and the vector b are constants which are known at the start of the program, so that they can be used both in the program text and in assertions.

The following program consists of the parallel composition of Q^2 processes (s, t), each of which executes an initialisation, followed by a loop of n/Q steps, numbered $k = 0, Q, 2Q, \cdots, n-Q$. Each step consists of four phases, lettered (a)-(d). In step k of the program the values $\hat{x}_k, \hat{x}_{k+1}, \cdots, \hat{x}_{k+Q-1}$ are computed. To achieve this, every process (s, t) keeps two process *invariants* [6] true. The main process invariant is the logical expression $P_1[s, t]$,

$$P_1[s, t] \equiv \forall_i[(0 \leqq i < k \wedge i \textbf{ mod } Q = s \wedge s = t) \Rightarrow x_i = \hat{x}_i] \wedge \quad (2.3)$$

$$0 \leqq k \leqq n \wedge k \textbf{ mod } Q = 0.$$

At the start of the computation, i.e., for $k = 0$, and also for $s \neq t$, the invariant is trivially true. On the other hand, the truth of all process invariants $P_1[s, s]$, $0 \leqq s < Q$, for $k = n$ implies the successful termination of the program. An additional invariant $P_2[s, t]$

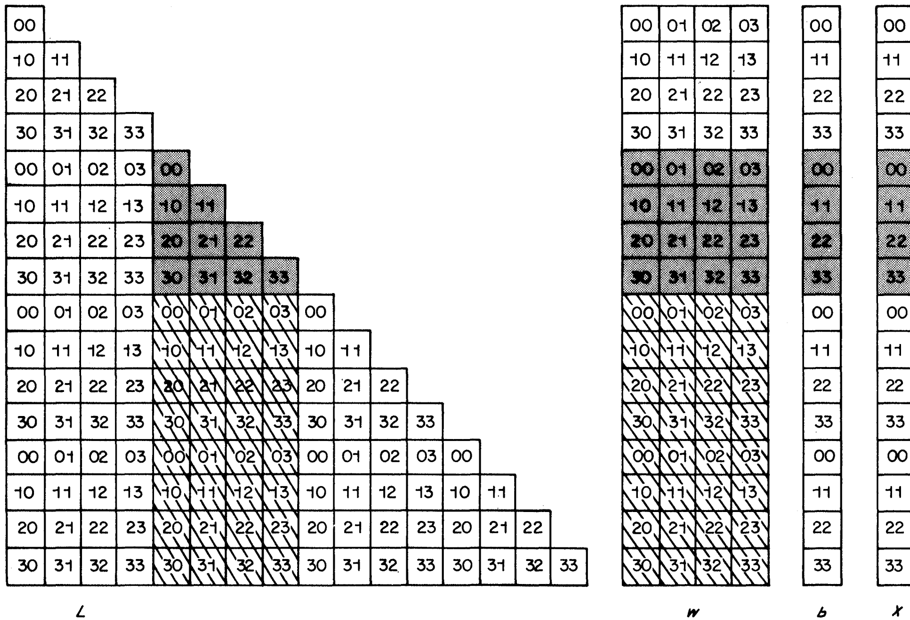

FIG. 1. *Matrix view of the distribution of data elements across the processors. The elements of the lower triangular part of L and the elements of w, b, and x are represented by cells. Each element is assigned to a processor* (s, t), $0 \leqq s, t < Q = 4$. *The order of L is* $n = 16$. *The number of processors is* $p = 16$. *The shaded elements of L, w, and b are used in phases* (a)-(c) *of step* $k = 4$ *to compute the shaded elements of x. These elements in turn, together with the hatched elements of L, are used in phase* (d) *to update the hatched elements of w.*

gives the relevant information about the current status of w,

$$(2.4) \qquad P_2[s, t] \equiv \forall_i \left[(k \leqq i < n \wedge i \ \mathbf{mod}\ Q = s) \Rightarrow w_{it} = \sum_{\substack{j=0 \\ j \ \mathbf{mod}\ Q = t}}^{k-1} L_{ij}\hat{x}_j \right].$$

In step k, variable w_{it} contains the accumulated sum of local values $L_{ij}\hat{x}_j$ for $j < k$. Such sums w_{it} are needed only for $i \geqq k$.

An outline of the algorithm is given by the following program text which records the status change caused by each phase, but which hides the details of the phase itself. For each phase, an annotated program text is given in § 3. For the sake of brevity, only the *changes* of the status are recorded in the comments between the four phases; comments which remain valid are not repeated. For example, in the program text below, $\{P_1[s, t] \wedge P_2[s, t]\}$ is valid prior to phase (a) and it will still be valid prior to phase (b). The program text is:

PROGRAM TEXT OF THE PARALLEL TRIANGULAR SYSTEM SOLVER.

S: **for all** s, t: $0 \leqq s, t < Q$ **par do** process (s, t)
process (s, t): **begin**
  for $r := 0$ **to** $n - Q$ **step** Q **do** $w_{r+s,t} := 0$;
  for $k := 0$ **to** $n - Q$ **step** Q **do**
  begin
    $\{P_1[s, t] \wedge P_2[s, t]$ for $k\}$
    phase (a);
    $$\left\{ w_{\text{right}} = \sum_{t \leqq q < Q} \hat{w}_{k+s,q} \vee s > t \right\}$$
    phase (b);
    $\{P_1[s, t]$ for $k := k + Q\}$
    $\{y = \hat{x}_{k+t} \vee s < t\}$
    phase (c);
    $\{y = \hat{x}_{k+t}\}$
    phase (d)
    $\{P_1[s, t] \wedge P_2[s, t]$ for $k := k + Q\}$
  end
end.

The main loop of the program achieves the aim of the computation: at the start of the first step, $k = 0$, $P_1[s, t]$ is trivially true, and $P_2[s, t]$ is true after the initialisation $w := 0$. At the end of the last step, $k = n - Q$, the assertion $\{P_1[s, t] \wedge P_2[s, t]$ for $k := k + Q\}$ implies $P_1[s, t]$ for $k = n - Q + Q = n$. Since this holds for all s and t, it follows that all components of $\hat{x}$ have been computed. It remains to be shown (see § 3) that the phases of step k establish $P_1[s, t] \wedge P_2[s, t]$ for $k := k + Q$, starting from $P_1[s, t] \wedge P_2[s, t]$ for k.

In phase (a), the processes in the right upper corner of the mesh ($s \leqq t$) cooperate in computing $w_{\text{right}} = \sum_{t \leqq q < Q} \hat{w}_{k+s,q}$ for any of these processes. Phase (a) consists of Q independent fan-ins from right to left. In phase (b), the processes in the left lower corner ($s > t$) cooperate in computing $w_{\text{left}} = \sum_{0 \leqq q \leqq t} \hat{w}_{k+s,q}$ for any of these processes and $w_{\text{left}} = \sum_{0 \leqq q < t} \hat{w}_{k+s,q}$ for the diagonal processes ($s = t$). Each diagonal process uses its w_{left} and w_{right} to compute one new component of $\hat{x}$. This is expressed by the comment $\{P_1[s, t]$ for $k := k + Q\}$. During this phase, the Q newly computed components of $\hat{x}$ are communicated downwards from the diagonal. This is expressed by

the comment $\{y = \hat{x}_{k+t} \vee s < t\}$. Phase (c) takes care of the communication of the newly computed components of $\hat{x}$ in the upward direction from the diagonal. Finally, in phase (d) the working space w is updated. As each process possesses all the necessary values ($\hat{x}_{k+t}$ and part of the L-values), this phase does not require any communications. The use of data elements in these phases is illustrated in Fig. 1.

3. Phases of the algorithm. This section presents and verifies the four phases of one step of the algorithm. Each step has been divided into phases in such a way that no communication between processes in different phases is necessary. Therefore, each phase can be verified as a separate parallel program, which consists of the parallel composition of Q^2 processes. For each phase a *precondition* and a *postcondition* [6] is stated. The precondition of a phase equals the postcondition of the previous phase. To verify a phase it is necessary to check whether its postcondition follows from its precondition and from the execution of its program.

A phase is verified by using assertions in the process text. Similarly to the Gries-Owicki theory [13], [14], an assertion is made about the value of every communicated item. These values are expressed in global constants (such as $\hat{x}$ and $\hat{w}$) which do not depend upon local process variables. The processes are first checked in isolation, as a sequential program, by assuming that assertions about received values are true. The aim of this check is to show that the values computed and sent by any process are indeed the asserted ones. Since the processes are parametrised by s and t, this check involves only a single process text. (There is no need to check Q^2 different processes.) After this check, it should be verified that send and receive assertions of different processes match, and that no deadlock occurs; this verification is omitted for the sake of brevity. The communication pattern of the phases is illustrated in Fig. 2.

The program text of phase (a) is:

ANNOTATED PROGRAM TEXT OF PHASE (a) OF STEP k.

$\{P_1[s, t] \wedge P_2[s, t]\}$
if $s \leqq t$ **then**
begin

receive w_{right} from process $(s, t+1)$; $\left\{ w_{\text{right}} = \sum_{t<q<Q} \hat{w}_{k+s,q} \right\}$

$\{w_{k+s,t} = \hat{w}_{k+s,t}\}\, w_{\text{right}} := w_{\text{right}} + w_{k+s,t} \left\{ w_{\text{right}} = \sum_{t \leqq q<Q} \hat{w}_{k+s,q} \right\}$

end;
if $s < t$ **then** send w_{right} to process $(s, t-1)$
$\left\{ w_{\text{right}} = \sum_{t \leqq q<Q} \hat{w}_{k+s,q} \vee s > t \right\}$

The verification of this phase is mainly the verification of the assertion $\{w_{k+s,t} = \hat{w}_{k+s,t}\}$ for processes (s, t) with $s \leqq t$. The precondition of phase (a) implies $P_2[s, t]$. Substituting $i = k+s$ in the definition (2.4) of $P_2[s, t]$ and in (2.2) gives

$$w_{k+s,t} = \sum_{\substack{j=0 \\ j \textbf{ mod } Q = t}}^{k-1} L_{k+s,j}\hat{x}_j = \sum_{\substack{j=0 \\ j \textbf{ mod } Q = t}}^{k+s-1} L_{k+s,j}\hat{x}_j = \hat{w}_{k+s,t}, \tag{3.1}$$

since there are no indices j with $k \leqq j < k+s$ and j **mod** $Q = t$, for $s \leqq t$ and k **mod** $Q = 0$.

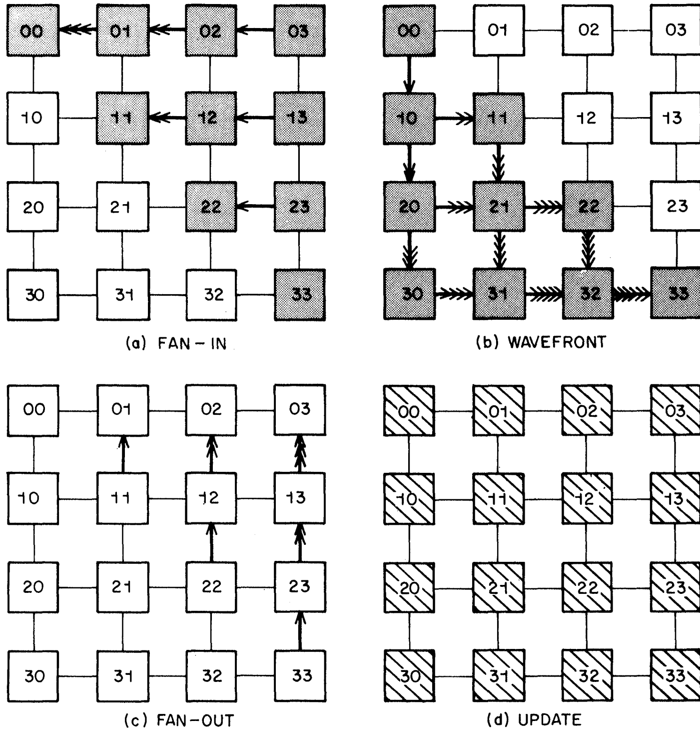

FIG. 2. *Mesh view of the four phases of one step of the triangular system solving algorithm*: (a) *horizontal fan-in of partial sums* $\hat{w}_{it}$; (b) *wavefront computation of* $\hat{x}_i$ *values*; (c) *vertical fan-out of* $\hat{x}_i$ *values*; (d) *update of partial sums* w_{it}. *Each square represents a processor* (s, t), $0 \leqq s,\ t < Q = 4$. *The processors are connected in a square mesh. Communications are shown by arrows. The communications performed first in a phase are represented by a single arrow; those performed next by a double arrow, and so on. Computations are shown by shading or hatching of processors, in correspondence with Fig.* 1.

The program text of phase (b) is:

ANNOTATED PROGRAM TEXT OF PHASE (b) OF STEP k.

$$\left\{P_1[s,t] \wedge P_2[s,t] \wedge \left(w_{\text{right}} = \sum_{t \leqq q < Q} \hat{w}_{k+s,q} \vee s > t\right)\right\}$$

if $s > t$ **then**
begin
  par begin
    receive y from process $(s-1, t)$; $\{y = \hat{x}_{k+t}\}$
    receive w_{left} from process $(s, t-1)$ $\left\{w_{\text{left}} = \sum_{0 \leqq q < t} \hat{w}_{k+s,q}\right\}$
  par end;
  par begin
    $\{y = \hat{x}_{k+t}\}$ send y to process $(s+1, t)$;
    begin
      $\{w_{k+s,t} + L_{k+s,k+t}y = \hat{w}_{k+s,t}\}\, w_{\text{left}} := w_{\text{left}} + w_{k+s,t} + L_{k+s,k+t}y$;
      $\left\{w_{\text{left}} = \sum_{0 \leqq q \leqq t} \hat{w}_{k+s,q}\right\}$ send w_{left} to process $(s, t+1)$
    end
  par end
end;

if $s = t$ **then**
begin
$x_{k+s} := b_{k+s} - w_{\text{right}}$;
receive w_{left} from process $(s, t-1)$; $\left\{ w_{\text{left}} = \sum_{0 \leqq q < t} \hat{w}_{k+s,q} \right\}$
$x_{k+s} := x_{k+s} - w_{\text{left}}$; $\{x_{k+s} = \hat{x}_{k+s}\}$
$y := x_{k+s}$;
$\{y = \hat{x}_{k+t}\}$ send y to process $(s+1, t)$
end
$\{P_1[s, t]$ for $k := k+Q\}$
$\{y = \hat{x}_{k+t} \vee s < t\}$

In the algorithmic notation above, statements between **par begin** and **par end** are performed in parallel, and those between **begin** and **end** are executed in sequence. Nesting of **begin-end** pairs is done with the conventional scope rules. For processes (s, t) with $s > t$ the current value of $w_{k+s,t}$ is not necessarily equal to $\hat{w}_{k+s,t}$ (in contrast to the case $s \leqq t$), but

$$w_{k+s,t} = \sum_{\substack{j=0 \\ j \bmod Q = t}}^{k-1} L_{k+s,j}\hat{x}_j = \sum_{\substack{j=0 \\ j \bmod Q = t}}^{k+s-1} L_{k+s,j}\hat{x}_j - L_{k+s,k+t}\hat{x}_{k+t} = \hat{w}_{k+s,t} - L_{k+s,k+t}\hat{x}_{k+t}. \tag{3.2}$$

This equation follows from $P_2[s, t]$, because in this case there is exactly one index j, $j = k+t$, such that $k \leqq j < k+s$ and j **mod** $Q = t$. Processes (s, t) with $s = t$ can subtract the values of their local variables w_{left} and w_{right} from b_{k+s}, to obtain $\hat{x}_{k+s}$ and thereby to establish $P_1[s, t]$ for $k := k+Q$. This follows from

$$w_{\text{left}} + w_{\text{right}} = \sum_{0 \leqq q < s} \hat{w}_{k+s,q} + \sum_{s \leqq q < Q} \hat{w}_{k+s,q} = \sum_{0 \leqq q < Q} \hat{w}_{k+s,q} = \sum_{0 \leqq q < Q} \sum_{\substack{j=0 \\ j \bmod Q = q}}^{k+s-1} L_{k+s,j}\hat{x}_j = \sum_{j=0}^{k+s-1} L_{k+s,j}\hat{x}_j = b_{k+s} - \hat{x}_{k+s}. \tag{3.3}$$

The program text of phase (c) is:

ANNOTATED PROGRAM TEXT OF PHASE (c) OF STEP k.

$\{y = \hat{x}_{k+t} \vee s < t\}$
if $s < t$ **then** receive y from process $(s+1, t)$; $\{y = \hat{x}_{k+t}\}$
if $s \leqq t$ **then** send y to process $(s-1, t)$
$\{y = \hat{x}_{k+t}\}$

The program text of phase (d) is:

ANNOTATED PROGRAM TEXT OF PHASE (d) OF STEP k.

$\{P_2[s, t] \wedge y = \hat{x}_{k+t}\}$
for $r := k+Q$ **to** $n-Q$ **step** Q **do** $w_{r+s,t} := w_{r+s,t} + L_{r+s,k+t}y$
$\{P_2[s, t]$ for $k := k+Q\}$

Comparing the invariant $P_2[s, t]$ for k and $k+Q$, (2.4), and substituting $i = r+s$ gives

$$w_{r+s,t}(\text{for } k+Q) = \sum_{\substack{j=0 \\ j \bmod Q = t}}^{k+Q-1} L_{r+s,j}\hat{x}_j = \sum_{\substack{j=0 \\ j \bmod Q = t}}^{k-1} L_{r+s,j}\hat{x}_j + L_{r+s,k+t}\hat{x}_{k+t} = w_{r+s,t}(\text{for } k) + L_{r+s,k+t}\hat{x}_{k+t}, \tag{3.4}$$

since there is exactly one index $j, j=k+t$, with $k \leqq j < k+Q$ and $j \bmod Q = t$. This shows that the loop of phase (d) indeed establishes the invariant $P_2[s, t]$ for incremented k.

4. Theoretical time analysis. The aim of this section is to derive a simple theoretical model that accurately predicts the performance of the parallel triangular system solving algorithm. To simplify the analysis it is assumed that there are synchronisation barriers between the different steps k of the algorithm, and between the different phases (a)-(d) of each step. This means that processes are assumed to wait until all other processes are ready before proceeding to the next step or phase. However, in the actual program the separate steps or phases may overlap to some extent. The predicted time is therefore an upper bound on the actual time. The time unit is the time needed to perform a *flop*, a floating point operation. The time, measured in flop units, needed to transfer one real value to a neighbouring process is assumed to be a constant α, which is the *communication-to-computation ratio.* It is assumed that each process is able to compute and communicate with all the neighbouring processes in parallel. (This assumption reflects the architecture of the transputer.) In the following analysis we neglect the time needed to evaluate conditionals, because these can be removed from the main loop of the algorithm. We also neglect any other overhead, such as indexing and addressing.

Phase (a) consists of Q simultaneous fan-ins (see Fig. 2(a)). The longest fan-in is from process $(0, Q-1)$ to process $(0,0)$, and involves $Q-1$ communications and $Q-1$ additions. The time t_a of this phase is

$$t_a = (Q-1)(\alpha+1). \tag{4.1}$$

Phase (b) starts with the simultaneous subtraction of w_{right} from b_{k+s} by all diagonal processes, in one time unit. This is followed by a wavefront-like flow of data from process $(0,0)$, through all processes (s, t) with $s \geqq t$, to process $(Q-1, Q-1)$ (see Fig. 2(b)). Process $(0,0)$ initiates the data flow at time 1 by sending $\hat{x}_k$ to process $(1,0)$; this communication terminates at time $\alpha+1$. Process $(1,0)$ then sends $\hat{x}_k$ to process $(2,0)$, finishing at time $2\alpha+1$. Parallel to this, process $(1,0)$ also performs two flops to compute w_{left} and then sends the result to process $(1,1)$; this operation is completed at time $2\alpha+3$. Continuing the analysis in the same manner for the other processes (s, t) with $s \geqq t$, it turns out that the last process, $(Q-1, Q-1)$, finishes at time $(Q-1)(2\alpha+4)-1$, for $Q>1$. The time of phase (b) is therefore

$$t_b = (Q-1)(2\alpha+4)-1. \tag{4.2}$$

Note that processes which have to receive values are always ready to do this. However, processes which have to send values often do this after some delay, because they must perform a number of flops first. For horizontal communication (from (s, t) to $(s, t+1)$) the delay is three flops if $t>0$, and two flops if $t=0$. For vertical communication there is only a delay, of one flop, if $s=t>0$. Phase (c) consists of Q simultaneous fan-outs. The time of phase (c) is

$$t_c = (Q-1)\alpha. \tag{4.3}$$

Phase (d) contains the bulk of the computations, and no communications, with

$$t_d = \frac{2(n-k-Q)}{Q}. \tag{4.4}$$

The total time required by the parallel algorithm for $p=Q^2$ processes is

$$
\begin{aligned}
T_{\text{par}} &\leqq \sum_{\substack{k=0 \\ k \bmod Q=0}}^{n-Q} \left\{ (Q-1)(4\alpha+5)-1+\frac{2(n-k-Q)}{Q} \right\} \\
&= \frac{n^2}{p}+(4\alpha+5)n-(4\alpha+7)\frac{n}{\sqrt{p}} \qquad (4.5) \\
&\leqq \frac{n^2}{p}+(4\alpha+5)n.
\end{aligned}
$$

This is a simple upper bound which is sharp for $p \gg 1$. The time of the best sequential algorithm is

$$T_{\text{seq}} = n^2 - n. \qquad (4.6)$$

An efficiency of 50 percent or more is achieved if $T_{\text{par}} \leqq 2T_{\text{seq}}/p$, i.e., if $p \leqq p_{1/2} \approx n/(4\alpha+5)$.

The algorithm can be generalised to solve triangular systems with multiple right-hand sides. Exploitation of pipelining increases the efficiency in this case. The algorithm for n_b right-hand sides has n/Q steps k, each of which consists of n_b times phase (a), followed by n_b times phase (b), and so on. The complexity of n_b consecutive executions of phase (a) is $(n_b-1)\alpha$ higher than the complexity of one execution, assuming that flops are overlapped with subsequent communications. The same holds for phases (b) and (c). The complexity of n_b consecutive executions of phase (d) is simply n_b times the complexity of one execution. A simple upper bound can be obtained as above, giving

$$T_{\text{par}}(n_b) \leqq \frac{n_b n^2}{p}+(4\alpha+5)n+3(n_b-1)\alpha\frac{n}{\sqrt{p}}. \qquad (4.7)$$

In the multiple right-hand side case, the lower-order terms are roughly a factor of $\min(n_b, \sqrt{p})$ less important (relative to the first term) than in the single right-hand side case. This means that efficiency loss is reduced by the same factor.

5. Experimental results. The algorithm has been implemented in the parallel programming language occam 2 [3] and executes on a square mesh of INMOS T800-20 transputers. Timing results were obtained for meshes of $p=1$, 4, 9, 16, 25, and 36 processors, and for matrices of order up to $n=1200$. Each transputer possesses a local memory of 256 Kbyte, which allows the storage of a 200×200 matrix per processor. The maximum speed of each communication link is 20 Mbit/s. The time we measured for the communication of a data packet is $t_{\text{comm}}(l)=2.3+2.2l$ μs, where l is the length of the message in 32-bit words. This means that the communication of a single word to a neighbouring processor takes $t_{\text{comm}}(1)=4.5$ μs. All computations were done in single precision (32 bits). All times were measured by an internal timer calibrated with a wall clock. All results were obtained for single right-hand side systems.

Table 1 shows the time $T_p(n)$ of triangular system solving for an $n \times n$ matrix on a square mesh of p transputers. Table 1 shows that the time $T_p(n)$ is a monotonously decreasing function of p, for a fixed matrix order n. Asymptotically the function reaches a lower bound (~1.8 ms for $n=50$). The simple theoretical model (4.5) explains this behaviour: the term n^2/p decreases with p, and the term $(4\alpha+5)n$ stays constant. This implies that *all* $p \leqq n^2$ processors available can be used to speed up the computations. There is no need to estimate a possible optimal number of processors.

TABLE 1
Timings (in ms) of the solution of an $n \times n$ unit lower triangular system on a network of p transputers.

n	$p=1$ (seq)	$p=1$ (par)	$p=4$	$p=9$	$p=16$	$p=25$	$p=36$
50	5.3	5.8	2.3	1.9	1.9	1.8	1.8
100	20.6	21.6	7.1	5.2	4.3	4.0	3.9
200	80.9	83.8	24.5	14.8	11.3	9.8	9.3
300			52.1	28.8	20.9	17.4	15.4
400			89.6	47.7	32.9	26.3	23.1
600				97.9	64.5	49.3	41.2
800					106	78.7	64.4
1000						115	91.4
1200							123

Figure 3 is a graph of the timings $T_p(n)$, with each curve representing the timings for various matrix orders n and for a fixed number of processors p. The form of the curves is parabolic. The curves with large p have a visible linear component which dominates at small values of n, but which is overtaken by the quadratic component at larger values. This agrees with the qualitative behaviour predicted by the theoretical model.

A quantitative test on the validity of the theoretical model was done as follows. The simple model (4.5) predicts an overall time of

$$T_p(n) \approx t_{\text{flop}} \frac{n^2}{p} + \beta n, \tag{5.1}$$

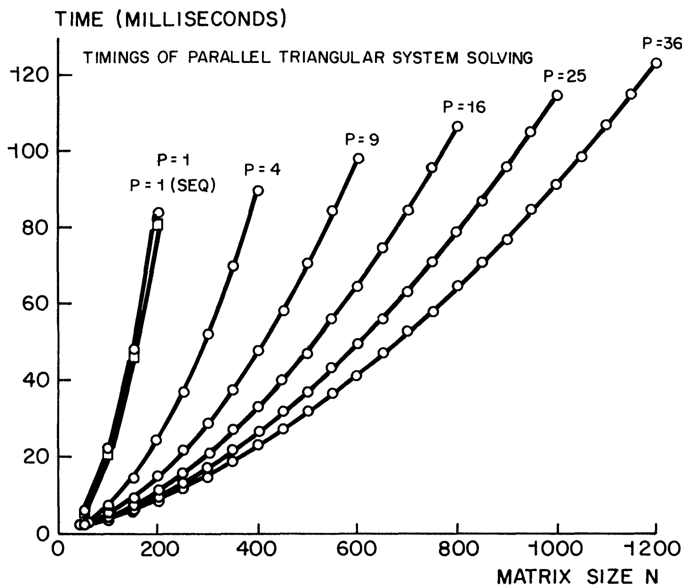

FIG. 3. *Timings of triangular system solving on a square mesh of* T800-20 *transputers. Each curve represents the solution times $T_p(n)$ for unit lower triangular matrices of various orders n and for a fixed number of processors p. The timings of the sequential program are marked by a square; those of the parallel program by a circle.*

where t_{flop} is the time needed to perform one flop, and β is, in the first approximation, a constant which does not depend upon p and n. The experimental timing data were used in a least-squares fit to determine $t_{\text{flop}} \approx 1.89$ μs and $\beta \approx 38.0$ μs. These empirical constants were inserted into (5.1), and the resulting times were compared with the experimental times. In most cases the difference between model and experiment was less than 5 percent. The notable exceptions are the cases where both p and n are small, so that $O(n/\sqrt{p})$ terms are significant compared to $O(n)$ terms and also compared to $O(n^2/p)$ terms, and hence cannot be neglected as in the derivation of (4.5). The maximum relative error (for $n = 50$ and $p = 4$) was 32 percent. A least-squares fit to a more refined model which included three terms, n^2/p, n, and $n/\sqrt{p}$, gave an error of typically less than 1 percent, and maximally 9 percent.

Figure 4 shows the efficiency of parallel triangular system solving. Each curve shows the efficiency $E_p(n)$ for various matrix orders n and for a fixed number of processors p. Our measure of efficiency is

$$E_p(n) = \frac{R_p(n)}{pR_{\text{seq, max}}}, \tag{5.2}$$

where $R_p(n) = n^2/T_p(n)$ is the rate in flop/s at which the computation proceeds and $R_{\text{seq, max}}$ is the maximum speed that can be obtained for the problem of triangular system solving by a sequential algorithm on a single transputer with 256 Kbyte memory. This rate $R_{\text{seq, max}} \approx 0.49$ Mflop/s is obtained for $n = 200$. The efficiency of the sequential algorithm is defined as $E_{\text{seq}}(n) = R_{\text{seq}}(n)/R_{\text{seq, max}}$. Note that the sequential efficiency can be less than unity. The maximum rate achieved on $p = 36$ transputers (for $n = 1200$) is 11.7 Mflop/s. The results in Fig. 4 show that for $p \leqq 36$ an efficiency of at least 65 percent can be achieved, if n is large enough. The performance of the algorithm decreases steadily with increasing p, reflecting the fact that the constant communication

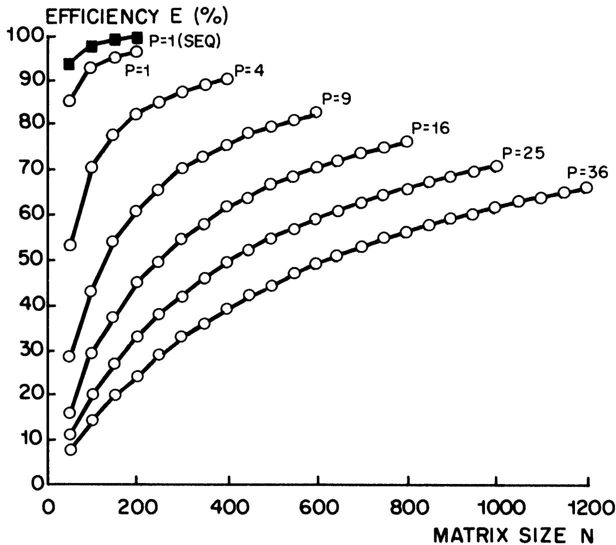

FIG. 4. *Efficiency of triangular system solving on a square mesh of* T800-20 *transputers. Each curve represents the efficiencies $E_p(n)$ for unit lower triangular matrices of various orders n and for a fixed number of processors p. The efficiencies of the sequential program are marked by a square; those of the parallel program by a circle. The efficiency $E_p(n)$ is defined as the ratio between the solution rate in flop/s of the parallel algorithm on p transputers, and p times the maximum sequential rate that can be obtained in triangular system solving on a single transputer with* 256 Kbyte *memory.*

time becomes more important relative to the decreasing computing time. The figure shows that 36 transputers achieve an efficiency of 50 percent for $n \geqq n_{1/2} \approx 650$, which is close to the value $n_{1/2} = \beta p / t_{\text{flop}} \approx 724$ obtained from (5.1).

An alternative performance measure for parallel algorithms on distributed-memory computers is the *scaled speedup* [7], defined as the ratio between the parallel and the sequential computing rate, for a fixed problem size *per processor.* In our case the scaled speedup is

$$S_p(n) = \frac{R_p(n)}{R_{\text{seq}}(n/\sqrt{p})}. \tag{5.3}$$

The scaled speedup achieved for $n = 1200$ and $p = 36$ is about 24.

6. Conclusions. We have presented an efficient algorithm for parallel triangular system solving on a square mesh of processors. The algorithm is simple to use, since no knowledge of the optimal number of processors is required: using all processors available will solve the problem in the shortest time. The time complexity of the algorithm is $n^2/p + O(n)$, so that full efficiency is achieved asymptotically (for $n \to \infty$). Even for small n an efficiency of more than 50 percent can be achieved, provided the time of a single communication and the time of a single floating point operation are of the same order of magnitude. This condition is met by our transputer network, which has a communication-to-computation ratio $\alpha \approx t_{\text{comm}}(1)/t_{\text{flop}} \approx 4.5/1.89 \approx 2.4$, and for instance by the Caltech/JPL Mark II hypercube, which has a ratio $\alpha = 4.7$ for 64-bit words [5, eq. (20.14)]. (Note that any measured value of α is problem-dependent, because different floating point operations have different computing times, and also because floating point operations may have varying amounts of indexing and addressing operations associated with them.)

The messages of our algorithm are short: each message contains only one real number, in the case of a single right-hand side. Furthermore, the messages are not pipelined, since only one out of $Q-1$ communication links is active at any one time, in each horizontal or vertical chain of processors. Therefore, messages cannot be combined to amortise communication startup time. Unfortunately, architectures such as the present Intel and Ncube hypercubes have high communication startup times, and hence a high communication-to-computation ratio α. For example, the Intel iPSC/2 has a ratio $\alpha = 59$ [2, Table 4]. Our algorithm is inefficient on such machines. In the case of multiple right-hand sides the situation is better, because messages belonging to different right-hand sides can be combined into larger messages, and these in turn can be pipelined.

The present study was performed on a square mesh, since this network has sufficient connectivity for parallel linear algebra algorithms such as LU decomposition and triangular system solving, and since it is trivially embedded in many other topologies such as a square torus, a hypercube of even dimension, and a fully connected network. On networks with a richer topology the algorithm can be executed without modification, or with adjustments to exploit the additional connectivity. The gains that can be obtained from richer connectivity are limited: in the best case, for a fully connected network, the communication complexity is reduced by a factor of two, from $4\alpha n$ to $2\alpha n$ [12]. This is also the communication complexity for a hypercube of even dimension [9].

An interesting extension of the present work is the generalisation of the triangular system solving algorithm for a square $Q \times Q$ mesh to an algorithm for a rectangular $M \times N$ mesh, using the corresponding rectangular grid distribution [1]. Under mild

constraints ($M \geqq N$ and $M \textbf{ mod } N = 0$) on the mesh dimensions it is easy to derive a generalised algorithm which has n/M steps, each with phases similar to the phases of the original algorithm. The generalised algorithm has the same complexity $n^2/p + O(n)$ as the original algorithm. The particular choice $M = p$ and $N = 1$ leads to a new so-called immediate-update row-wrapped $(ji - r)$ algorithm [15], which differs from the $ji - r$ algorithm of [15], for instance, because its communication complexity is independent of the number of processors.

In conclusion, a grid-based parallel algorithm has been presented for triangular system solving that can be used in combination with grid-based LU decomposition algorithms. Experiments on a square mesh of transputers have shown that the algorithm can be implemented efficiently on a distributed-memory MIMD computer with a low communication-to-computation ratio.

Acknowledgments. We are grateful to Daniël Loyens for many valuable suggestions on the exposition of the algorithm. We also thank Netty van Gasteren for proposing a more concise form of invariant P_2.

REFERENCES

[1] R. H. BISSELING AND J. G. G. VAN DE VORST, *Parallel* LU *decomposition on a transputer network*, Lecture Notes in Computer Science 384, Springer-Verlag, Berlin, 1989, pp. 61–77.

[2] L. BOMANS AND D. ROOSE, *Benchmarking the* iPSC/2 *hypercube multiprocessor*, Concurrency: Practice and Experience, 1 (1989), pp. 3–18.

[3] A. BURNS, *Programming in occam* 2, Addison-Wesley, Wokingham, U.K., 1988.

[4] S. C. EISENSTAT, M. T. HEATH, C. S. HENKEL, AND C. H. ROMINE, *Modified cyclic algorithms for solving triangular systems on distributed-memory multiprocessors*, SIAM J. Sci. Statist. Comput., 9 (1988), pp. 589–600.

[5] 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.

[6] D. GRIES, *The Science of Programming*, Springer-Verlag, New York, 1981.

[7] J. L. GUSTAFSON, G. R. MONTRY, AND R. E. BENNER, *Development of parallel methods for a* 1024-*processor hypercube*, SIAM J. Sci. Statist. Comput., 9 (1988), pp. 609–638.

[8] M. T. HEATH AND C. H. ROMINE, *Parallel solution of triangular systems on distributed-memory multiprocessors*, SIAM J. Sci. Statist. Comput., 9 (1988), pp. 558–588.

[9] S. L. JOHNSSON, *Communication efficient basic linear algebra computations on hypercube architectures*, J. Parallel Distrib. Comput., 4 (1987), pp. 133–172.

[10] G. LI AND T. F. COLEMAN, *A parallel triangular solver for a distributed-memory multiprocessor*, SIAM J. Sci. Statist. Comput., 9 (1988), pp. 485–502.

[11] ———, *A new method for solving triangular systems on distributed-memory message-passing multiprocessors*, SIAM J. Sci. Statist. Comput., 10 (1989), pp. 382–396.

[12] L. D. J. C. LOYENS AND R. H. BISSELING, *The formal construction of a parallel triangular system solver*, Lecture Notes in Computer Science 375, Springer-Verlag, Berlin, 1989, pp. 325–334.

[13] S. OWICKI AND D. GRIES, *Verifying properties of parallel programs: An axiomatic approach*, Comm. ACM, 19 (1976), pp. 279–285.

[14] ———, *An axiomatic proof technique for parallel programs* I, Acta Inform., 6 (1976), pp. 319–340.

[15] C. H. ROMINE AND J. M. ORTEGA, *Parallel solution of triangular systems of equations*, Parallel Comput., 6 (1988), pp. 109–114.

[16] J. G. G. VAN DE VORST, *The formal development of a parallel program performing* LU-*decomposition*, Acta Inform., 26 (1988), pp. 1–17.

[17] D. W. WALKER, T. ALDCROFT, A. CISNEROS, G. C. FOX, AND W. FURMANSKI, LU *decomposition of banded matrices and the solution of linear systems on hypercubes*, in Proc. Third Conference on Hypercube Concurrent Computers and Applications, Association for Computing Machinery, New York, 1988, pp. 1635–1655.