A Parallel Triangular Solver for a Hypercube Multiprocessor

Guangye Li[1,2]
Thomas F. Coleman[3]

TR 86-787

October 1986

Department of Computer Science
Cornell University
Ithaca, NY 14853

1 Research partially supported by the U.S. Army Research Office through the Mathematical Sciences Institute, Cornell University.
2 Permanent address: Computer Center, Jilin University, People's Republic of China.
3 Research partially supported by the Applied Mathematical Sciences Research Program (KC-04-02) of the Office of Energy Research of the U.S. Department of Energy under grant DE-FG02-86ER25013.A000.

A Parallel Triangular Solver for a Hypercube Multiprocessor

Guangye Li [1,2]
Thomas F. Coleman [3]

Computer Science Department &
Center for Applied Mathematics
Cornell University
Ithaca, New York 14853

October 1986

Abstract: We consider solving triangular systems of linear equations on a hypercube multiprocessor. Specifically, we propose a fast parallel algorithm, applicable when the triangular matrix is distributed around the cube by column in a wrap fashion. Numerical experiments indicate that the new algorithm is very efficient.

A theoretical analysis confirms that the total running time varies linearly, with respect to the matrix order, up to a threshold value of the matrix order, after which the dependence is quadratic. Moreover, we show that total message traffic is essentially the minimum possible.

Finally, we describe an analogous row-oriented algorithm.

Keywords: hypercube multiprocessor, parallel triangular solver, parallel Gaussian Elimination, parallel matrix factorization

AMS Subject Classification: 65F05

---

[1] Research partially supported by the U.S. Army Research Office through the Mathematical Sciences Institute, Cornell University.

[2] Permanent address: Computer Center, Jilin University, People's Republic of China.

[3] Research partially supported by the Applied Mathematical Sciences Research Program (KC-04-02) of the Office of Energy Research of the U.S. Department of Energy under grant DE-FG02-86ER25013.A000.

0. Introduction

Recent work on hypercube algorithms for numerical linear algebra has resulted in efficient parallel methods for the factorization of dense matrices (eg. Geist & Heath[1986], Moler[1986]). However, the discovery of methods for the efficient parallel solution of the resulting triangular systems has lagged behind somewhat. This is especially true in the case where matrices are distributed by column (instead of by row). For example, Geist & Heath [1986] recently demonstrated that the best parallel triangular solver (at that time) was essentially no better than a distributed sequential solver. While there has been some very recent progress on this problem (Romine & Ortega [1986]), we propose to further close this gap with a new and efficient column-oriented parallel triangular solver.

Our main purpose in this paper is to propose and analyze a new algorithm for the parallel solution of a triangular system of linear equations, where the columns of the matrix are distributed over the nodes (processors) of a hypercube multiprocessor in a wrap fashion. We provide experimental results and theoretical evidence to suggest that this new algorithm is an efficient parallel method.

It is worth pointing out that there is a definite need for column-based parallel triangular solvers. Specifically, matrices are often naturally generated by column. For example, consider a finite-difference approach to Newton's equations:

$$F'(x)s = -F(x)$$

where we assume that a subroutine for the evaluation of $F(x)$ is available. Then it is possible to approximate $F'(x)$ with the matrix $A(x)$ where

$$A(x)e_i = \frac{F(x + \delta e_i) - F(x)}{\delta}$$

and e_i is the i^{th} column of the identity matrix. Hence, $A(x)$ is generated by column. Of course a matrix transpose is always possible (McBryan and Van de Velde [1985]) but this is an expense that can be avoided.

We have implemented our triangular solve algorithm on an Intel iPSC hypercube multiprocessor with 16 nodes. However, it should be noted that the full hypercube architecture is not needed by this algorithm: only ring connectivity is used. Therefore, the algorithm can be used on any local memory message-passing multi-processor in which a ring can be embedded provided **send** and **receive** primitives are available. We assume that when control of a node program passes to a **send** statement, the **send** is executed immediately, in time zero, and then control passes on to the next executable statement in this node program. Of course this does not imply the message is received immediately - we discuss this transfer time below. We also assume that when control passes to a **receive** statement in a node program, execution of this node program is suspended until the message is physically received which happens when the appropriate transfer time elapses.

Message passing speed plays an important role in the execution time of algorithms for the solution of triangular systems of linear equations. This contrasts with the factorization stage (Moler [1986]) in which floating point computations clearly dominate. In order to quantify message-passing speed in our analysis, we use a quantity t:

> t is the maximum number of flops that can be executed on a single processor during the time in which a single 'small' message is **sent** by one node and then **received** by a waiting adjacent node.

In this context we define 'small' to be a message of size less than or equal to p double precision words (64 bits each), where p is the number of processors. On the Intel iPSC (release 2.0) with $p \leq 16$, we have estimated t to be approximately 100 flops, where a flop is the operation

$$y \leftarrow ax + y$$

We estimated t in the following way. First, a message of size $p = 16$ was forwarded around a 'ring' of 16 processors for a total of 1000 round-trips. That is, processor i sent the message to node $i+1$; node $i+1$ received the message and then sent it to node $i+2$, ... The total time for these 1000 circuits was recorded as T_1. Second, the time to execute the loop

for $i=1:1000$
    $y \leftarrow ax + y$

was recorded as T_2. Then t was estimated:

$$t \cong \frac{T_1}{16 \times T_2}.$$

Notice that the 'ring' used above is defined by the natural ordering of the nodes. Since the natural ordering does not define a true ring, t is overestimated by this computation: we feel that this difference is relatively insignificant and we ignore it.

Our paper is organized as follows. Section 1 describes the column algorithm and provides numerical results and comparisons. Section 2 provides two analyses of the column algorithm. The first yields an expression for the running time as a function of n,t, and p. This expression is most illuminating and we discuss this result at length. The second analysis establishes that the proposed algorithm induces essentially the minimum message traffic possible. A row-distributed algorithm, similar in spirit to the column method, is discussed in Section 3. Concluding remarks and possible extensions are given in Section 4.

1. The Column Algorithm: Description and Numerical Results

In this section we describe a new algorithm to solve the upper triangular system $Ux = b$ on a hypercube multiprocessor where the n columns of the matrix U are distributed to the p nodes (processors) in a wrap fashion. A wrap mapping is used because it seems a very reasonable choice for the factorization stage (e.g. Geist & Heath[1986]). The order in which the nodes are visited in the wrap assignment is arbitrary; however, the algorithm is most reasonable when the nodes form a ring. We refer to the node containing column j as $P(j)$. Therefore, due to the wrap assignment, $P(j) = P(k)$ if and only if $j = k \ (mod\ p)$.

1.1 The Algorithm

To motivate our new parallel solver, we first consider an ordinary sequential solver followed by a distributed sequential solver. We consider the system $Ux=b$ where U is upper triangular.

A Sequential Solver

The algorithm we present is the well-known 'outer-product' version. The right-hand-side is repeatedly updated by a column of U multiplied by a newly determined component of the solution. The following program, *SEQ*, sequentially computes the solution to $Ux=b$ on a 1-processor system.

Procedure SEQ

For $j = n{:}1$

$b(j) \leftarrow b(j)/U(j,\ j)$

$b(1{:}j-1) \leftarrow b(1{:}j-1)\ -\ U(1{:}j-1,\ j) \times b(j)$

End

A Distributed Sequential Solver

Now let us suppose that the columns of U are distributed amongst the nodes of a hypercube in a wrap fashion as described above. It is not difficult to change *SEQ* into a distributed sequential solver. In particular, the only change is that b must be passed to the processor that computes the next variable. To follow is the node program. We assume that the right-hand side b is initially stored on node $P(n)$. On termination b has been overwitten with the solution on node $P(1)$. Each node executes the following program.

Procedure DSEQ

For $j=n:1$

If $myname = P(j)$

Receive $b(1:n)$ [if $j<n$]

$b(j) \leftarrow b(j)/U(j, j)$

$b(1:j-1) \leftarrow b(1:j-1) - U(1:j-1, j)\times b(j)$

Send $b(1:n)$ to node $P(j-1)$ [if $j>1$]

End

DSEQ is a distributed solver but is not parallel: there is no concurrent computation. However, parallelism can be added by noting that node $P(j)$ can defer most of the b-modification until after **Send**ing some updated information to node $P(j-1)$. In particular, it is only necessary to process p components of b before **Send**ing. The remaining components can be modified afterward and thus concurrently with computations on other nodes.

A Parallel Column Triangular Solver (PCTS)

Mechanically, the algorithm is simple: the $p-1$ array SUM passes around the ring going from $P(j)$ to $P(j-1)$ for $j=n:2$. When SUM arrives at node $P(j)$, $P(j)$ determines $x(j)$, modifies SUM (p flops), and then forwards SUM to node $P(j-1)$. Finally, node $P(j)$ modifies the first $j-p$ elements of array $PSUM$ using column j of U ($j-p$ flops).

The procedure PCTS is executed by every node: the following initial conditions are assumed.

If myname $= P(n)$: $SUM(1:p) = b(n:n-p+1)$
$PSUM(1:n-p) = b(1:n-p)$

If myname $\neq P(n)$: $SUM(1:p) = 0$
$PSUM(1:n-p) = 0$

Also: For brevity, we follow the convention that if an array index is out of bounds, the returned value is assumed to be zero. Each node has at most $m = \left\lceil \frac{n}{p} \right\rceil$ columns of the upper triangular matrix U.

Procedure PCTS ($x[1{:}m]$, $SUM(1{:}p)$, $PSUM(1{:}n-p)$, $U(1{:}n,[1{:}m])$)

For $j=n{:}1$

If myname $= P(j)$

Receive $SUM(1{:}p-1)$ [if $j < n$]

$x(j) \leftarrow (SUM(1) + PSUM(j))/U(j,j)$

$SUM(1{:}p-2) \leftarrow SUM(2{:}p-1) - U(j-1{:}j-(p-2),\ j)\times x(j)$
$\qquad + PSUM(j-1{:}j-(p-2))$

$SUM(p-1) \leftarrow -U(j-(p-1),\ j)\times x(j) + PSUM(j-(p-1))$

Send $SUM(1{:}p-1)$ to node $P(j-1)$ [if $j > 1$]

$PSUM(1{:}j-p) \leftarrow PSUM(1{:}j-p) - U(1{:}j-p,\ j)\times x(j)$

End

We have listed all the arrays used, and their dimensions, in the procedure statement. The square brackets indicate indirect addressing. For example, $x[1{:}m]$ says that there are at most m components of the vector x on this node but they are not necessarily the first m components of the n-vector x. In particular, the components of x are assigned to the nodes in a wrap mapping consistent with the assignment of columns. Rather than introduce indirect indexing in the body of the procedure, we refer to components directly. So for example, $x(j)$ refers to the j^{th} component of the solution x, not the j^{th} component of the array $x[1{:}m]$ on this node. Of course for this reference to be valid, this component must be assigned to this node.

The mechanism behind *PCTS* can be described as a distributed outer product. On each node k the array *PSUM* holds part of the outer product corresponding to processed columns on that node; the travelling array *SUM* funnels the distributed sums together, as needed.

1.2 Lower Solve, Permutations, and Collecting x

A major application of *PCTS* is in the context of Gaussian Elimination in which there are four steps:

1. Factor $PA = LU$
2. $b \leftarrow Pb$
3. $b \leftarrow L^{-1}b$
4. $b \leftarrow U^{-1}b$

We are explicitly addressing step 4 in this paper. Obviously, step 3 can be accomplished by a similar algorithm provided *SUM* and *PSUM* are initialized on node $P(1)$. We will not give details here except to point out that, upon completion of step 3, b is distributed around the cube. Algorithm *PCTS*, as it appears here, apparently needs the right-hand-side on node $P(n)$. In fact, a trivial change to *PCTS* allows you to solve for x with essentially the same algorithm without collecting b onto a single processor.

In the next section, numerous experiments are discussed; some involve the complete solution to $Ax = b$ via steps 1-4 above. The factorization algorithm used is essentially that of Moler[1986] except that full rows of A are interchanged instead of partial rows. Thus, on completion, A is overwritten with LU such that $PA = LU$. The permutation matrix P is stored as a single n-array, distributed over the cube in accordance with the column distribution. Therefore, in order to execute Step 2, we first collect P onto a single processor, node $P(1)$: Every node sends 1 message of length $\left\lceil \frac{n}{p} \right\rceil$ to node $P(1)$. A similar collection step is performed, with respect to the solution, after step 4. Experimentally, these two collection steps contribute very little to the running time of the triangular solution stage.

1.3 Numerical Experiments with PCTS

Our experiments were performed on an Intel hypercube iPSC (release 2.0) with 16-nodes using Fortran ftn286. All computations were performed in double precision. This cube is operated by the Theory Center at Cornell University. This particular system has extra memory boards so that the total available memory per node is approximately 4 megabytes. Hence, the total available memory on a 16-node cube is approximately 64 megabytes: we were able to run experiments on dense matrices up to order approximately $n = 2000$.

The megaflop rate is the number of millions of floating point operations (megaflops) per second. *Note: Megaflop rate does not refer to millions of flops* *per second but rather millions of floating point operations per second. One flop requires two floating point operations.* Experimentally, we determined the maximum megaflop rate per node, $mflp_1^*$, to be .033 megaflops per second. Hence, the maximum megaflop rate for the cube on 16 nodes, $mflp_{16}^*$, is approximately .53 megaflops per second. $mflp_1^*$ was determined by executing *PCTS* solver on one node for large n; then,

$$mflp_1^* = \frac{n^2 \times 10^{-6}}{T}$$

where T is the execution time.

For all our experiments, we used a single test problem where the matrix A is

$$A(i, j) = \frac{1}{n-i-j+1.5}$$

and $b(i) = n-i+1$.

Pivoting is required for stability purposes when solving this system.

One final remark concerning our reported results: we did not, in fact, use a ring ordering for our experiments. The natural ordering of the nodes of the cube was used. Hence we expect a modest improvement in the performance of *PCTS* when implemented on a real ring .

In the Context of LU Solve

Our original motivation for pursuing this research was to develop a column based parallel triangular solver whose running time was insignificant relative to the *LU* factorization time for all reasonably large n. A distributed sequential triangular solver such as *DSEQ* does not have this property.

In Table 1.1 we present numerical results obtained on a cube of size $p = 16$. The *LU* factorization program used is due to C. Moler (Intel Scientific Computers) but modified slightly: in our version, row interchanges are explicitly performed on the whole row rather than just on the part of the row in U. This has no noticeable effect on the running time of the *LU* factorization but allows for the separation of the tasks 2 and 3 mentioned in Section 1.2.

The triangular solve times listed include the application of the permutation matrix to the right-hand-side as well as the collection of the final solution in node $P(1)$. For comparison we have also listed the running time for algorithm *DSEQ* (a distributed sequential solver). In both cases we used an assembler version of the BLAS (Basic Linear Algebra Subroutines, Lawson et. al [1979]), as implemented by C. Moler.

N	LU	DSEQ	PCTS
100	5.216	2.496	2.080
200	23.248	14.992	4.144
400	125.552	56.992	8.352
600	363.376	112.464	12.416
800	801.424	213.440	16.656
1000	1491.024	286.656	20.592
1200	2509.840	439.296	24.768
1500	4759.842	628.080	30.960
2000	10974.032	1122.624	41.696

LU Factorization Versus
Triangular Solve [seconds]
Table 1.1 ($p = 16$)

As evidenced above, the total running time is clearly dominated by the factorization stage when the new parallel triangular solver is used. For example, when $n = 600$, *DSEQ* takes about 1/3 of the factorization time, whereas *PCTS* takes 1/30 of it.

Figure 1.1 is a plot of *DSEQ* versus *PCTS*: clearly the difference between the two methods is dramatically increasing with n.

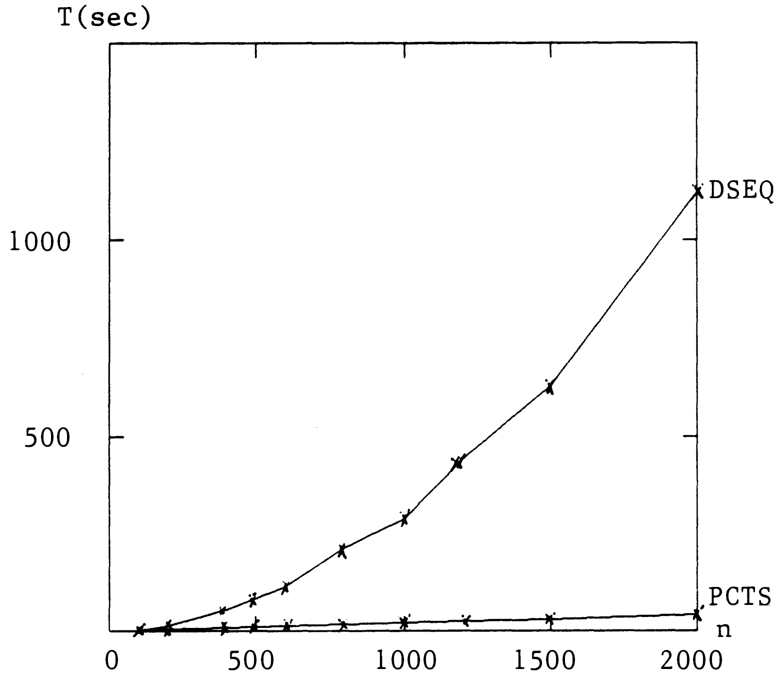

Figure 1.1 ($p = 16$)

Time as a Function of Problem Size n

An important concern is the way in which the running time of *PCTS* varies with n. As we establish in Section 2, the dependence is linear up to a threshold value of n, and then quadratic. We explain this transition in Section 2.

Graphs of the running times of the triangular solve stage, for different n, are shown in Figures 1.2 & 1.3. Note: the times reported represent the sums of the times to apply the permutation matrix, execute both the lower and upper triangular solves, and then collect the solution on node $P(1)$.

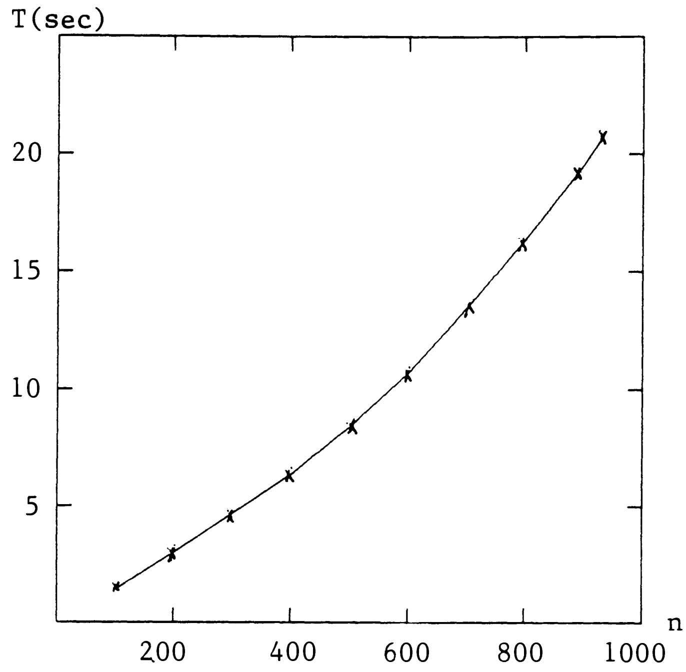

Figure 1.2 ($p = 4$)

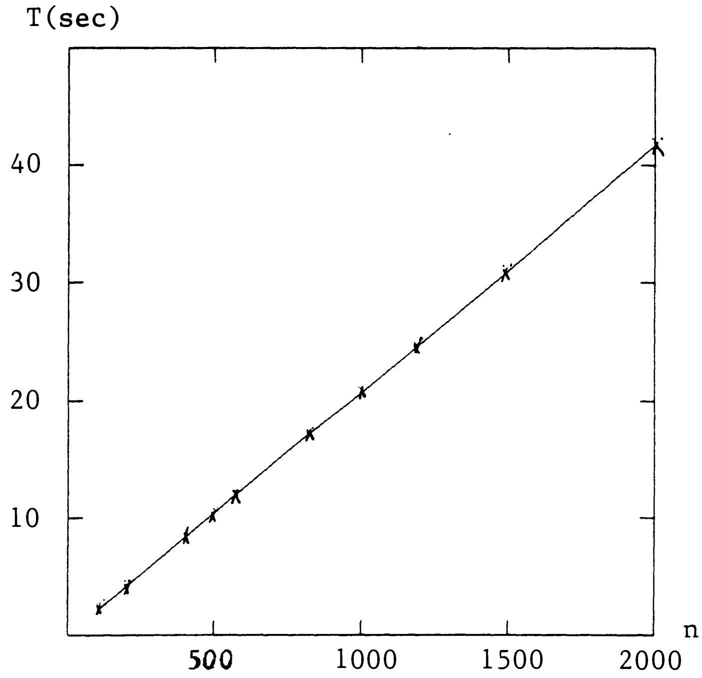

Figure 1.3 ($p = 16$)

In Figure 1.2 the transition from linear to quadratic is clear. The threshold value occurs at approximately $n = 400$ which is close to the theoretical value established in the next section.

Figure 1.3 shows that T varies linearly for n up to its maximum possible value when $p=16$. In this case the threshold value of n, demarking the transition from linear to quadratic performance, is beyond the storage capacity of the 16-node cube.

In Figures 1.4 and 1.5 we chart the megaflop rates obtained for different values of n for both the 4-node and 16-node cubes. The megaflop rate is computed as follows. For each run the system clock records the running time, T, for the triangular solution stage (ie. apply P, two triangular solves, collect solution on node $P(1)$). Then, the megaflop rate on a p-node cube is :

$$mflp_p = \frac{2n^2 \times 10^{-6}}{T}$$

We use $2n^2$ because this is the total number of floating point operations to solve the lower and upper systems. The maximum megaflop rate, $mflp_p^*$, was determined as described previously.

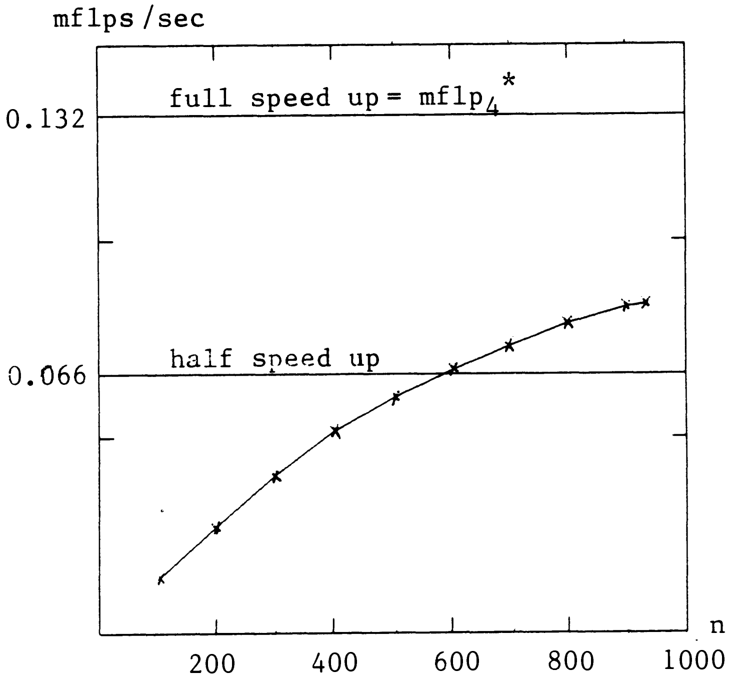

Figure 1.4 (p = 4)

Notice that, in Figure 1.4, the linear/quadratic transistion is evident. In Figure 1.5, the curve remains linear because the quadratic stage demands a matrix of size exceeding the storage capacity.

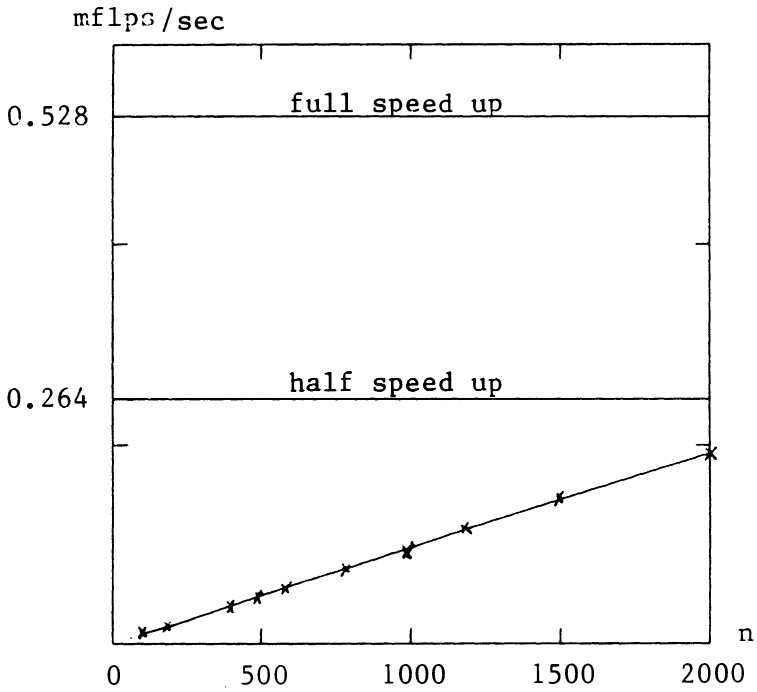

Figure 1.5 ($p = 16$)

Comparisons

In this section we experimentally compare our new triangular solver, *PCTS*, to an alternative parallel column solver, *CPIP*, suggested by Romine & Ortega [1986] and implemented by C. Moler. (We used C. Moler's implementation in our experiments. We believe it represents a careful and efficient implementation of the algorithm described by Romine & Ortega [1986]. It is written in the same style as *PCTS*: e.g. the 'BLAS' were used wherever possible. We should note that the assembler versions of the BLAS were not used in these particular experiments. The reason is that the assembler codes available to us did not allow for step increments ('INCX') different from unity: hence, *CPIP* could not benefit from an assembler implementation. Therefore, to be fair, we used only Fortran code in both cases.)

This algorithm complements *PCTS* in the sense that while *PCTS* is a distributed outer-product method, *CPIP* computes a distributed inner-product.

Procedure CPIP

For $i = n{:}1$

$t \leftarrow 0$

For each $k > i$ such that $P(k) = myname$

$t \leftarrow t + x(k) \times U(i,k)$

Participate in fan-in: $\sum t \rightarrow t$ on node $P(i)$

If myname $= P(i)$

$$x(i) \leftarrow \frac{b(i) - t}{U(i,i)}$$

End

We use the notation '$\sum t \rightarrow t$ on node $P(i)$ ' to indicate that the distributed partial sums are added together, using a tree structure embedded in the hypercube, with the final sum determined on node $P(i)$. Moler [1986] describes this process in more detail.

Results of our experiments comparing our implementation of *PCTS* with Moler's implementation of *CPIP* follow in Table 1.2.

N	CPIP	PCTS
100	3.904	2.192
200	7.744	4.432
400	16.480	8.880
600	25.424	13.360
800	35.328	17.776
1000	45.568	22.432
1200	56.208	26.576
1500	73.744	33.840
1765	90.688	40.624

Comparison of new method (*PCTS*) versus inner product method (*CPIP*) [seconds]

Table 1.2 ($p = 16$)

Notice that the difference between *PCTS* and *CPIP* increases with n. Indeed, this trend is clearly illustrated in Figure 1.6.

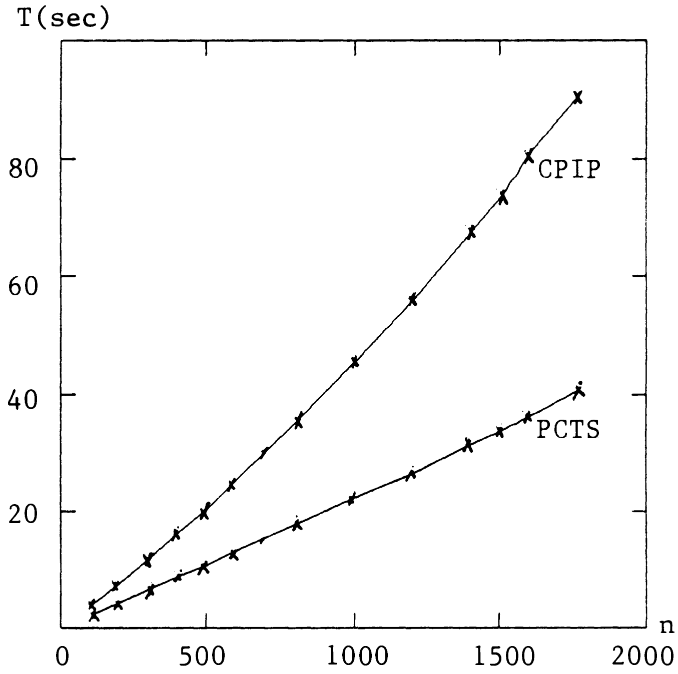

Figure 1.6 (p = 16)

2. Analysis of Column Method

In this section we analyze algorithm *PCTS*. First, we derive an expression for the time complexity T -measured in flops - as a function of n, t, and p. (Recall that t is a measure of the time (in flop units) to send a message of size p from one node to an adjacent node.) Expression T accounts for total computation time, including communication costs. Our theory is completely consistent with, and explains, the emperical evidence reported in Section 1. A second analysis considers only the message passing aspect of *PCTS*: we show that *PCTS* achieves minimum message traffic, in several senses, over all algorithms for solving a distributed triangular system.

2.1 Complexity Analysis

The major result in this section is an expression for the total time T, expressed in number of flops.

For simplicity, let us assume that $n = mp$ for some integer $m > 2$ where p is the number of nodes (processors) and n is the matrix dimension. Further, we assume that the columns are dealt out to the nodes in a wrap fashion; the nodes are 'configured' to form a ring. Let $P(i)$ be the node housing column i. Hence $P(i+1)$ is adjacent to $P(i)$ for $i = 1{:}n-1$ and $P(1)$ is adjacent to $P(n)$.

For convenience, we have taken some freedom with the definition of the term 'flop'. In general, 1 flop represents the operation $y \leftarrow ax + y$ which is 2 floating point operations. However, algorithm *PCTS* does not cleanly break up into flops: for example, the update of $SUM(i)$ for $1 \le i \le p-2$, requires slightly more than 1 flop. We have decided to ignore these deviations in our analysis: we decree that every assignment statement in *PCTS* costs 1 flop. This assumption allows for a much cleaner presentation and obscures no important information. (Alternatively, we could count floating point operations - every multiply, add, subtract, and divide counts as 1. We have decided not to present this analysis because it is very clumsy and no more insightful than the 'flop' approach.)

Theorem 2.1. If $n \le p(t+p)$ *then*

$$T = (t+p)n - \frac{p(p-1)}{2} - t$$

else

$$T = \tfrac{1}{2}\{\frac{n^2}{p} + n + p(t+p)^2 - pt - p^2 + p\} - t.$$

Before proving this result, we first analyze the work profile of node $P(n)$. It is important to realize that node $P(n)$ is the only node which can cause the cycling $p-1$ vector SUM to be delayed.

Theorem 2.2. Every node, except possibly node $P(n)$, processes SUM immediately after the message containing SUM arrives.

Proof: This is obviously true in the first cycle. Without loss of generality, assume it is not true in the second cycle: Let node $P(n-p-j)$, $j \neq 0$, be the first node (except, possibly, for node $P(n)$) at which SUM is delayed. Hence node $P(n-p-j)$ is still busy processing column $n-j$ when SUM arrives in the second cycle. Let T_{n-j} be the time it takes SUM to traverse the cycle

$$P(n-j) \rightarrow P(n-j-1) \rightarrow \cdots \rightarrow P(n-p) \rightarrow \cdots \rightarrow P(n-p-j) = P(n-j)$$

where we do not count the processing times of node $P(n-j) = P(n-p-j)$. Since SUM cannot be processed immediately by $P(n-j)$ at the end of this cycle, and node $P(n-j)$ must perform $n-j-p$ flops,

$$T_{n-j} < n-j-p. \tag{2.1}$$

Define T_n to be the time it takes SUM to traverse the first cycle

$$P(n) \rightarrow P(n-1) \rightarrow \cdots \rightarrow P(n-p) = P(n)$$

where we do not count any processing time in node $P(n)$. Therefore,

$$T_{n-j} = T_n + W \tag{2.2}$$

where we assume that SUM must wait for time $W \geq 0$ before node $P(n)$ is able to process column $n-p$. Hence, since node $P(n)$ must perform $n-p$ flops before it is ready to process column $n-p$,

$$T_n + W \geq n-p \tag{2.3}$$

Obviously, (2.1)−(2.3) and $j \neq 0$ yield a contradiction. $\square$

The next result says that the total processing time T can be determined by essentially just considering the processing time of node $P(n)$. Let $T(P(n))$ be the total elapsed time of node $P(n)$: i.e. the time at which processor $P(n)$ finishes processing column p.

Lemma 2.3. $T = T(P(n)) + t(p-1) + \dfrac{p(p-1)}{2}$ •

Proof: After column p is completed at node $P(n)$, vector SUM is then passed around the ring one last time: at each node $P(p-i)$, $p-i$ flops are performed. Since this last phase is totally sequential, the result follows. □

We can now prove *Theorem 2.1*.

Proof: The activity of node $P(n)$ alternates between stages of floating point operations, $F_1, F_2, \ldots, F_m$ and possible idle or waiting periods, $W_1, \ldots, W_{m-1}$. Hence, the activity sequence is

$$F_1, W_1, F_2, W_2, \ldots, F_m. \tag{2.4}$$

The number of flops in stage F_i is $n-(i-1)p$. By Theorem 2.2 the vector SUM is processed immediately upon arrival at each node on the cycle (except possibly node $P(n)$). Now, since each node performs p flops before forwarding SUM, and since SUM is transferred to the subsequent node in time t, it follows that W_i is bounded above by $tp + (p-1)p$. However, after forwarding the p-vector SUM, node $P(n)$ must complete F_i: $n-ip$ flops remain. Therefore, the waiting time is

$$W_i = \max\{tp + p(p-1) - (n-ip),\ 0\} \tag{2.5}$$

which we will denote as $(tp+p(p-1) - (n-ip))^+$. Therefore, the total processing time for node $P(n)$ is

$$T(P(n)) = \sum_{i=1}^{m}(n - (i-1)p) + \sum_{i=1}^{m-1}(tp+p(p-1) - (n-ip))^+. \tag{2.6}$$

But

$$\sum_{i=1}^{m}(n - (i-1)p) = \frac{n^2}{2p} + \frac{n}{2} \tag{2.7}$$

and the second term in (2.6) can be simplified if we consider 2 separate cases. In particular if $n \le p(t+p)$ then this term is just $\sum_{i=1}^{m-1}(tp+p(p-1)-(n-ip))$ and with (2.7) yields

$$T(p(n)) = (t+p)n - \frac{p(p-1)}{2} - t\ . \tag{2.8}$$

On the other hand, if $n > p(t+p)$ then

$$\sum_{i=1}^{m-1} (tp+p(p-1)-(n-ip))^{+} = \sum_{i=m-(t+p)+1}^{m-1} (tp+p(p-1)-(n-ip))$$

which simplifies to $\frac{p}{2}\{(t+p)^2-3(t+p)+2\}$. Therefore, in this case,

$$T(p(n)) = \frac{n^2}{2p} + \frac{n}{2} + \frac{p}{2}\{(t+p)^2-3(t+p)+2\} \tag{2.9}$$

By (2.8), (2.9), and Lemma 2.3, the result follows. □

Theorem 2.1 provides considerable insight into algorithm PCTS and its expected behaviour. We discuss this next.

T as a Function of n

Let p and t be fixed. Then we see that for $n \leq p(t+p)$, $T(n)$ is a linear function of n; otherwise, $T(n)$ varies quadratically with n. This degree change is easy to explain and is clearly consistent with our numerical experiments. The linear behaviour occurs when the computation time is bound by the time it takes the $(p-1)$-array SUM to travel around the ring (with p flops at each node). In this case, whenever SUM arrives at node $P(n)$, it is processed immediately without delay (i.e. the processor is idle when the message arrives). On the other hand, the quadratic behaviour is exhibited when SUM cannot always be processed immediately when it arrives at node $P(n)$: In this case the overall processing time is computation bound.

Figure 2.1 was generated using the expressions in Theorem 2.1 (i.e. we used a plotting routine to generate Figure 2.1). The linear portion is labelled f_l and the quadratic, f_q. The computation time T follows f_l until it reaches (and is tangent to) f_q, after which T follows f_q. Note that T is always above the curve depicting full speed-up, $\frac{n^2}{2p}$. Of course, from Theorem 2.1,

$$\frac{\frac{1}{2}n^2}{T(n)} \rightarrow p \text{ as } n \rightarrow \infty. \qquad (2.10)$$

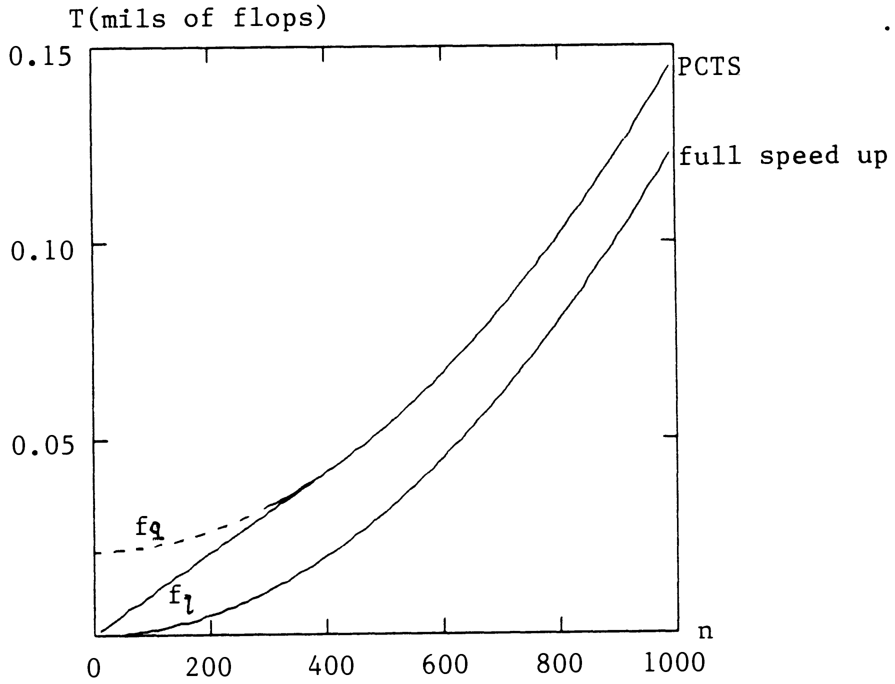

Figure 2.1 $(p = 4, t = 100)$

Theorem 2.1 indicates that the point of intersection occurs when $n = (t+p)p$: this is corroborated by numerical experiments. For example, if $t = 100$ and $p = 4$ then the curves meet at approximately $n = 400$; for $p = 16$ the intersection point is roughly $n = 2000$.

Of course full speed-up is never actually achieved; however, (2.10) indicates that a speed-up of $\frac{p}{2}$ can be obtained for some value of n. Define $n_{\frac{1}{2}}$ to be that value. Hence,

$$\frac{\frac{1}{2}n_{\frac{1}{2}}^2}{T(n_{\frac{1}{2}})} = \frac{p}{2}$$

This measure reflects the rate at which full speed-up is approached, as n increases. In particular, $n_{\frac{1}{2}}$ can help one determine whether or not a substantial speed-up will occur in practise, for reasonable n. Moler [1986] also uses a similar definition of $n_{\frac{1}{2}}$.

Interestingly, a rough estimate of $n_{\frac{1}{2}}$ yields that half maximum speed-up occurs just when f_l intersects f_q. To see this consider $f_l \cong (t+p)n$. But if

$$\frac{\frac{1}{2}n^2}{(t+p)n} = \frac{p}{2}$$

then $n = p(t+p)$. On the other hand,

$$f_q \cong \tfrac{1}{2}\{\frac{n^2}{p} + n + p(t+p)^2\}$$

and using this approximation, half maximum speed-up yields

$$n^2 - pn - p^2(t+p)^2 = 0$$

An approximate root to this quadratic is $n = p(t+p)$.

T as a function of t

As t decreases the threshold value of n (marking the transition from f_l to f_q) decreases. Offsetting this effect, both f_l and f_q decrease in value: a net reduction in T ensues. As expected, T is computation-bound for a wider range of n as the communication time t decreases.

T as a function of p

It is natural to expect T to decrease as the number of processors, p, increases. In fact, this is not entirely true. In particular, suppose n and t are fixed and let p_0 be such that $n < p_0(t + p_0)$. It is now easy to see that $T(p)$ increases as p increases. This incline continues until $n = p(t+p)$ at which point $T(p)$ is then defined by the second function,

$$T(p) = \tfrac{1}{2}\{\frac{n^2}{p} + n + p(t+p)^2 - pt - p^2 + p\} - t$$

Now, as p continues to increase T will generally decrease (provided n is not too small). Hence, the optimal value of p, for fixed p and t, and n sufficiently large, is implicitly defined by $n = p(t+p)$.

2.2 Message Traffic Analysis

Algorithm *PCTS* exhibits low communication costs; the experimental success of this algorithm is due, in part, to this property. In this section we show that message traffic is minimum, in several senses.

First, a few definitions are in order. Define a *Unit of Traffic* (*TU*) to be the transfer of a one word of information from one node to an adjacent node. The *Traffic Volume* (TV) is just the sum of the traffic units.

In Theorem 2.6 we prove that algorithm *PCTS* exhibits essentially the minimum possible TV over a wide class of algorithms to solve $Ux = b$. We establish three preliminary results first. *All results in this section are under the assumption that the columns of U are distributed over the nodes of a hypercube in wrap fashion and $n \geq p$.*

First we consider the traffic required to determine component $x(i)$.

Lemma 2.4: The determination of $x(i)$ requires at least $\min\{p-1, n-i\}$ *TU's.*

Proof: The i^{th} component of x is

$$x(i) = \frac{[b(i) - U(i,i+1{:}n)^T x(i+1{:}n)]}{U(i,i)}$$

But $U(i,i+1{:}n)$ is distributed over $min(p-1, n-i)$ nodes; therefore, $x(i)$ induces at least that many TU's. □

The next result gives the minimum possible TV over all algorithms for $Ux = b$.

Lemma 2.5: The determination of x requires at least $n(p-1) - \frac{p(p-1)}{2}$ *TU's.*

Proof: By Lemma 2.4 the determination of $x(i)$ requires $\min(p-1, n-i)$ TU's. But, because each $x(i)$ correspond to different rows, they must be independent units; hence

$$TV = (n-p)(p-1) + \sum_{i=1}^{p-1} i$$

which simplifies to the stated result. □

We are now ready to show that the message traffic produced by the algorithm *PCTS* is essentially minimum. Let TV_* be the minimum message traffic volume over all algorithms to solve $Ux = b$ (under the wrap assignment assuption).

Theorem 2.6: Let TV be the message traffic volume for PCTS. Then,

$$TV = TV_* + O(p^2)$$

Proof: The message volume is clearly $(p-1)(n-1)$ units which simplifies to $TV_* + \frac{(p-2)(p-1)}{2}$. □

Therefore, for fixed p, PCTS is just a constant away from exhibiting minimum message traffic volume. Note that this gap is somewhat arbitrary and can easily be overcome by a slight modification to PCTS. In particular, the 'send statement' can be changed to read:

Send $SUM(1{:}min(p-1,j-1))$ to node $P(j-1)$ [if $j > 1$]

and the algorithm is still valid but now exhibits minimum message traffic volume.

The previous three results are concerned with traffic volume; the next considers instead the total number of messages. We define an algorithm for $Ux = b$ *consistent* if the final determination of $x(i)$ is computed on node $P(i)$.

Lemma 2.7: At least $n-1$ messages are required by a consistent algorithm for $Ux = b$ (again, a wrap mapping is assumed).

Proof: The variable $x(i)$ cannot be computed until $U(i,i+1)\times x(i+1)$ is known for $(i < n)$. But $x(i+1)$ is finally determined on $P(i+1)$ whereas $x(i)$ is finally determined on $P(i)$; therefore, at least 1 message must pass between these 2 nodes before $x(i+1)$ is finally determined. □

Theorem 2.8: Algorithm PCTS passes the fewest possible total messages over all consistent algorithms. □

Since PCTS produces the fewest number of messages as well as the minimum traffic volume (essentially) the next result follows immediately.

Corollary 2.9: Over all minimum message passing consistent algorithms, PCTS minimizes the maximum message length. □

Remark: It is important to realize that minimizing TV or the total number of messages is not a guarantee of best possible communication. The integration of a given message with other possible activities must be considered. For example, in a typical global sum fan-in procedure (Moler[1986]), the traffic volume is p as is the number of messages, but the total time required is proportional to $\log p$. Alternatively, consider *PCTS* when n is large. In this case concurrent floating point computations completely mask message traffic.

3. The Row Algorithm

The column-based algorithm *PCTS* has an analogous row-based procedure. In particular, a $p-1$ array *XSUB* is passed around the ring, going from $P(i)$ to $P(i-1)$ for $i=n:1$. Upon arrival at $P(i)$, $x(i)$ is determined (p flops), *XSUB* is shifted and forwarded to node $P(i-1)$, and finally, m elements of the right-hand-side are modified. Richard Chamberlain [1986] was the first to suggest this type of row-oriented algorithm.

In the procedure listed below, we follow the convention that if an index is out of range then the resulting expression is zero; all variables are initialized to be zero.

Procedure PRTS $(XSUB[1{:}p], b[1{:}m], U([1{:}m], 1{:}n)$

for $i=n:1$

If myname $= P(i)$

Receive $XSUB(1{:}p-1)$ [if $i<n$]

$b(i) \leftarrow b(i) - U(i,\ i+1{:}i+p-1) \times XSUB(1{:}p-1)$

$b(i) \leftarrow b(i)/U(i,i)$

$XSUB(2{:}p-1) \leftarrow XSUB(1{:}p-2)$

$XSUB(1) \leftarrow b(i)$

Send $XSUB(1{:}p-1)$ to node $P(i-1)$ [if $i>1$]

for $k=i{:}i+p-1$

for each l such that $P(l) = myname,\ l< i$

$b(l) \leftarrow b(l) - U(l,k) \times XSUB(k-i+1)$

End

Again, we have listed all arrays used and their dimensions in the procedure statement where the square brackets indicate indirect addressing. For example, $b[1{:}m]$ says that there are at most m components of the vector b on this node but they are not necessarily the first m components of the n-vector b. In particular, the components of the n-vector b are assigned to the nodes in a wrap mapping consistent with the assignment of columns. Also: rather than introduce indirect indexing in the body of the procedure, we refer to components directly. So, for example, $b(i)$ refers to the i^{th} component of the n-vector b, not the i^{th} component of the array $b[1{:}m]$ on this node. Of course for this reference to be valid, this component must be assigned to this node.

Due to the similarity in structure between algorithms *PCTS* and *PRTS* and since we have fully analyzed the former, we have not included an analysis of the latter.

We have implemented and experimented with *PRTS*. The final do-loop in *PRTS* was implemented as shown: i.e. as a sequence of 'daxpy's. Note that it is also possible to implement this update as a sequence of inner products ('ddot's).

The numerical results follow in Table 3.1. We compare *PRTS* to the row analogue of *CPIP*, *RPIP*, which is structurally very similar to the column method except the 'fan-in' operation is replaced with a 'fan-out'.

RPIP assumes that the rows of the matrix are distributed, in wrap fashion, around the cube. This algorithm was suggested by Geist & Heath [1985] and is found below. In our experiments we used our own implementation of this method: we believe that this program represents a very careful and efficient implementation of the algorithm described in Geist & Heath [1985]. It is coded in the same style as *PRTS*: i.e. the Fortran 'BLAS' were used wherever possible.

Procedure RPIP

for i=n:1

  If myname $= P(i)$

    $x(i) \leftarrow b(i)/U(i,i)$

    Initiate broadcast of $x(i)$

  else

    Participate in broadcast of $x(i)$

  For each $k < i$ such that $P(k) = myname$

    $b(k) \leftarrow b(k) - x(i) \times U(k,i)$

End

The performance of *PRTS* is as expected, similar to the column algorithm *PCTS* and considerably better than the inner product method *RPIP*.

N	RPIP	PRTS
100	4.592	1.936
200	9.120	3.616
400	18.848	7.120
600	29.200	10.544
800	39.336	14.112
1000	51.056	17.456
1200	62.896	21.136
1500	82.032	28.000
1800	102.016	35.552

The new row algorithm (RCTS) versus
inner product row algorithm [seconds]

Table 3.1 ($p = 16$)

In Figure 1.7 we plot *RPIP* versus *PRTS*: clearly the 2 curves drift apart as n increases.

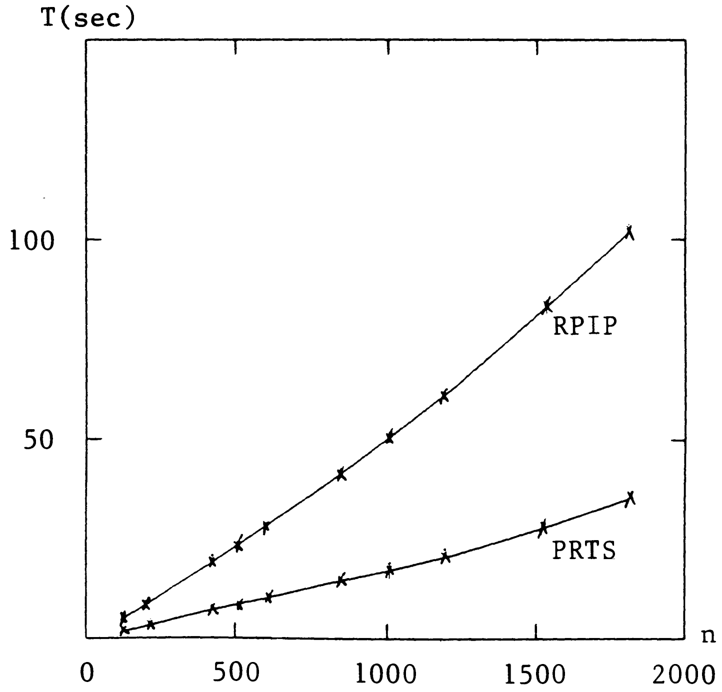

Figure 1.7 ($p = 16$)

4. Concluding Remarks

We have presented a new parallel algorithm, *PCTS*, for the solution of triangular systems of linear equations, applicable when the columns of the matrix are distributed amongst the nodes of a hypercube computer, in a wrap fashion. The running time is clearly dominated by matrix factorization time for all reasonable n. Moreover, our numerical results suggest that algorithm *PCTS* may be significantly faster than the recently proposed fan-in method of Romine and Ortega[1986], which uses a more sophisticated message passing technique (ie. fan-in).

Perhaps there is a global message here. In particular, a distributed triangular system necessitates relatively high communication costs - at least np traffic units (approximately) and $n-1$ total messages - compared to total floating point computation requirements which are of order n^2. In such a situation it appears to be advantageous to use simpler communication methods (e.g. a ring versus a global fan-in) - at least this seems to be true for a modest number of processors.

Indeed, in a recent private communication, M. Heath (Oak Ridge National Laboratory) reported another example in support of this thesis: apparently it is possible to solve distributed triangular systems using an inner-product **ring** algorithm. Heath's experiments indicate that this method may be competitive with *PCTS* in some cases. Once again simple communication methodology appears to win.

On the other hand, the ring mechanism exhibits some advantage even when n is large. In such a case, the computations are essentially pipelined, thus masking communication costs entirely (except at the ends). This contrasts with the fan-in type algorithm in which communication is never masked by computation.

In any event, it now appears that triangular systems can be efficiently solved on hypercube computers either when matrices are distributed by rows or by columns. The choice between row or column distribution should definitely not be based on the resulting efficiency of the triangular solution step.

For fixed p and t, as $n \to \infty$, the difference between the running times of *PCTS* and *CPIP* goes to infinity: ie.

$$T(CPIP) - T(PCTS) \to +\infty.$$

This is easy to see because $T(CPIP)$ is approximately $\frac{n^2}{2p} + tn \cdot logp$ flops whereas $T(PCTS)$ is approximately $\frac{n^2}{2p} + \frac{n}{2}$ flops, for large n. On the other hand, this difference becomes relatively less important as $n \to \infty$. The total computational cost is growing with n^2, but the difference $T(CPIP) - T(PCTS)$ grows only linearly with n.

If p is varied, the situation becomes less clear though recent experiments by C. Moler indicate that the gap between $T(CPIP)$ and $T(PCTS)$ narrows as p increases and n grows more modestly. This seems quite reasonable to us; however, as we mention in the next section, we believe *PCTS* can be modified to increase efficiency when p is large but n is modest. We will experiment with this scheme when we have a 'large' cube available.

The effect of variation in t is difficult to predict. It is worth noting that t is a relative quantity, balancing processor speed with communication time. In the future, we can expect t to fluctuate, as advances are made in message-passing technology and processor speed.

Future Work

There are at least two possible ways to improve *PCTS* is some circumstances.

First, for very large n, and fixed p and t, *PCTS* is not well-balanced. In particular, at the beginning of a run with large n, *PCTS* is computation bound because the vectors are long; later in the computation, vectors are short and therefore the processors are idle much of the time. However, it is possible to re-arrange the work in *PCTS* to achieve an evenly-balanced work profile. For example, work on the upper half of the last $\frac{n}{2}$ columns can be postponed until columns in the first half are being processed. In such a fashion a rectangular work profile can be achieved.

Second, it is possible to exploit the fact that for fixed t and n there is an optimal p: i.e. it is not always advantageous to have p as large as possible. With the help of Richard Chamberlain (Chr. Michelsen Institute), we have discovered that it is possible to divide the 'great ring' on p nodes into several small pieces - each of 'optimal size' - and then revise *PCTS* accordingly. This technique should lead to significant gains in efficiency when p is large and n is not significantly larger than p.

Acknowledgements

We are grateful to a number of people who read a previous draft of this report. As a result of their comments the current version is much improved. In particular we thank Chris Bischof, John Gilbert, Doug Moore, and Earl

Zmijewski. We are also grateful to Cleve Moler and David Scott of Intel Scientific Computers for supplying us with various subroutines and help when needed. We also thank Mike Heath of Oak Ridge National Laboratory for alerting us to the interesting paper by Romine and Ortega, and for being generally available for consultation and discussion.

Finally, we express our appreciation to Cornell University's Theory Center - and the Advanced Computing Facility in particular - for the use of the Intel iPSC hypercube computer.

References

R.M. Chamberlain [1986], *An Algorithm for LU Factorization with Partial Pivoting on the Hypercube*, Technical Report CCS 86/11, Chr. Michelsen Institute, Bergen, Norway.

G.A. Geist and M.T. Heath [1985], *Parallel Cholesky Factorization on a Hypercube Multiprocessor*, Technical Report ORNL-6211, Oak Ridge National Laboratory, Oak Ridge, Tennessee.

G.A. Geist and M.T. Heath [1986], *Matrix Factorization on a Hypercube Multiprocessor*, in Hypercube Multiprocessors 1986, M. Heath ed., SIAM Publications, Philadelphia, PA.

C. Lawson, R.Hanson, D. Kincaid, and F. Krogh [1979], *Basic Linear Algebra Subprograms for Fortran Usage*, ACM Transactions on Mathematical Software, 5, 308-371.

O.A. McBryan and E.F. Van de Velde [1985], *Hypercube Algorithms and Implementations*, Technical Report, Courant Institute of Mathematical Sciences, New York University, New York, N.Y.

C. Moler [1986], *Matrix Computation on Distributed Memory Multiprocessors*, Technical Report, Intel Scientific Computers.

C.H. Romine and J.M. Ortega, *Parallel Solution of Triangular Systems of Equations*, Technical Report RM-86-05, Department of Applied Mathematics, University of Virginia.