A New Method for Solving Triangular Systems on Distributed Memory Message-Passing Multiprocessors

Guangye Li[*]
Thomas F. Coleman[†]

TR 87-812
March 1987

Department of Computer Science
Cornell University
Ithaca, New York 14853-7501

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

[†]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 New Method for Solving Triangular Systems on Distributed Memory Message-Passing Multiprocessors

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

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

March 1987

Abstract: Efficient triangular solvers for use on message passing multiprocessors are required, in several contexts, under the assumption that the matrix is distributed by columns (or rows) in a wrap fashion. In this paper we describe a new efficient parallel triangular solver for this problem. This new algorithm is based on the previous method of Li and Coleman [1986] but is considerably more efficient when $\frac{n}{p}$ is relatively modest, where p is the number of processors, and n is the problem dimension.

A useful theoretical analysis is provided as well as extensive numerical results obtained on an Intel iPSC with $p \leq 128$.

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

An important part of many computations is the solution of a triangular system of equations. On a sequential computer, this is a simple task; however, an effective parallel procedure on a message-passing multiprocessor is far from trivial. Indeed, recently there has been a flurry of research activity directed at just this issue (e.g. Heath[1986], Li & Coleman[1986], Moler[1986a,b], Romine & Ortega[1986]). In this paper a new algorithm is described: this procedure is an enhancement of the recent method proposed by Li & Coleman [1986] and represents a significant improvement.

The algorithm proposed by Li & Coleman [1986] is applicable on any message-passing multiprocessor (with no shared memory) on which it is possible to embed a ring. Theoretically the new algorithm proposed in this paper can get by with just ring connectivity as well; however, this new algorithm will usually be more efficient when there is more connectivity than this. The precise architectural design is unimportant provided a ring can be embedded; we have conducted all our experiments on hypercube computers.

We assume that the system to be solved is $Ux = b$ where U is an upper triangular matrix of order n. We also assume that U is distributed to the nodes (processors) by column. (A similar algorithm exists for the row-distributed case; however, since it is quite straightforward given the column algorithm, we will not discuss it further.) This is a natural assumption in many cases. For example, the finite-difference estimation of Jacobian matrices yields a matrix column-by-column. Finally, we assume that the columns of U are assigned to the nodes in a wrap fashion. So, for example, if the node containing column j is $P(j)$ then $P(j) = P(k)$ if and only if $j = k(mod\ p)$, where there are p nodes altogether. A wrap mapping is chosen because it seems a very reasonable choice for the matrix factorization stage (e.g. Geist & Heath [1986], Chamberlain [1986]) which often precedes a triangular solve.

The original algorithm proposed by Li and Coleman [1986] is capable of effectively solving triangular systems in many cases. For example, the numerical results reported by Li & Coleman[1986], Heath [1986], and Moler[1986b] suggest that in many cases it represents the best available algorithm. However, upon close inspection it appears that this algorithm begins to lose ground as $\frac{n}{p}$ decreases (note: we always assume $n \geq p$). For example, Moler [1986b] compared the Li-Coleman algorithm to another parallel solver due to Romine & Ortega [1986]. In these experiments the Li-Coleman method was faster by a factor of 4.5 when $p = 16$ and $n = 710$

whereas this ratio decreased to 1.38 when $p = 128$ and $n = 1890$. Heath's [1986] results are even more dramatic. When $p = 16$ and $n = 900$ Heath reports a ratio of about 2 in favour of the Li-Coleman algorithm. However, when $p = 64$ and $n = 900$ the tables have turned; the ratio is approximately .8. (We note in passing that these results are not really comparable. In both cases Intel iPSC computers were used; however, different languages and implementations, different precisions, and different timing 'rules' make it very difficult to compare.)

The purpose of this paper is to describe a modification of the original Li-Coleman algorithm. This modified version does not degrade in performance as p increases relative to n. Moreover, it actually reduces to the original Li-Coleman algorithm when $\frac{n}{p}$ is relatively large and therefore the modified algorithm maintains a high level of performance under these circumstances as well.

The proposed algorithm can be used on any distributed-memory message-passing multiprocessor 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 (e.g. Moler [1986c]) 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 $\frac{p}{q}$ double precision words (64 bits each), where p is the number of processors and q is a positive integer to be discussed later.

Our paper is organized as follows. In Section 1 the new algorithm will be motivated and described; results of numerical experiments will be given.

An analysis of the new algorithm will be provided in Section 2. In Section 3 we briefly discuss a modified version of the new algorithm - this modifed version achieves a well-balanced (rectangular) work distribution (instead of triangular). Section 4 will contain a summary and concluding remarks.

1. The Algorithm: Motivation, Description, and Numerical Results

1.1 Motivation

The parallel triangular solver *PCTS*, proposed by Li and Coleman[1986], is based on a ring architecture and assumes the columns are assigned to the nodes in a wrap fashion. In particular, if the node that contains column i is $P(i)$ then $P(j) = P(k)$ if and only if $j (mod\ p) = k\ (mod\ p)$. In the embedded ring, node $P(i\ (mod\ p))$ is connected to $P(i+1\ (mod\ p))$, $i=1{:}p$.

Mechanically, algorithm *PCTS* 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 $\neq P(n)$: $SUM(1{:}p) = 0$
$PSUM(1{:}n-p) = 0$

```
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 $j < n$]
            $x(j) \leftarrow (SUM(1) + PSUM(j))/U(j,j)$
            $SUM(1{:}p-2) \leftarrow SUM(2{:}p-1) - U(j-1{:}j-(p-2), j)\times x(j)$
                        $+ PSUM(j-1{:}j-(p-2))$
            $SUM(p-1) \leftarrow -U(j-(p-1), j)\times x(j) + PSUM(j-(p-1))$
            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
```

Note: 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.

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.

Assuming p is fixed, Li and Coleman showed that the running time of *PCTS*, $T(n)$, is a function which is linear up to a threshold value of n after which it is quadratic. In the quadratic range $T(n)$ represents essentially optimal speedup (this is because node $P(n)$ is almost always busy doing useful floating point computations - the other nodes are not quite as busy as $P(n)$ due to variation in column size). However, $T(n)$ reflects less than optimal speed in the linear region and since the threshold value of n can be quite large, it is worthwhile trying to improve *PCTS* in this region.

$T(n)$ is linear when the busiest node, node $P(n)$, has idle time in every cycle of *SUM*. Therefore, it is easy to see that under these circumstances $T(n)$ is dominated by the time it takes *SUM* to complete a cycle:

$$\textit{cycle time} \cong p^2 + tp \tag{1.1}$$

Obviously, if *SUM* could be reduced to size $\frac{p}{q} - 1$, for $q \geq 1$, then the time to complete a cycle would be

$$\textit{cycle time} \cong (\frac{p}{q})p + tp \tag{1.2}$$

provided there are no other compromising ill-effects. In particular, the size of q must be restricted because expression (1.2) is no longer valid if *SUM* arrives back at node $P(n)$ before node $P(n)$ is ready to process *SUM*. Hence, the following principle guides our choice of q: choose q as large as possible subject to the constraint that node $P(n)$ is always (just) ready to process *SUM* when it arrives. Note that this principle suggests to choose $q = 1$ for n sufficiently large - i.e. resort to the original *PCTS* algorithm.

The step from algorithm *PCTS* to the new algorithm *PCTS*$^+$ can best be understood, perhaps, by considering the first cycle of *PCTS* in which the last p components of x are solved for. In particular, notice that the determination of $x(n-1)$ does not involve $x(n) \times U(n-p+1{:}n-2,\ n)$. Similarily, the determination of $x(n-2)$ does not involve $x(n) \times U(n-p+1{:}n-3,\ n)$ nor does it involve $x(n-1) \times U(n-p{:}n-3, n-1)$. Hence it may be possible to ship information **across** the ring (provided the communication links are there) while maintaining essentially the same algorithmic form. So, for example, if $q = 2$ the vector $x(n) \times U(n-p+1{:}n-\frac{p}{2}, n)$ could be shipped 'directly' to node $P(n - \frac{p}{2})$.

Generalizing this notion, in the first cycle node $P(n)$ computes the vector $x(n) \times U(n-p+1{:}n, n)$ in q packets, each of size $\frac{p}{q}$. The first packet (*SUM*) is sent to node $P(n-1)$, the second packet is sent to node $P(n-\frac{p}{q})$, the third packet is sent to node $P(n-2\frac{p}{q})$, and so on. Figure 1.1 shows the communication pattern for $p = 16$ and $q = 4$. For simplicity we also assume that $n = 16$.

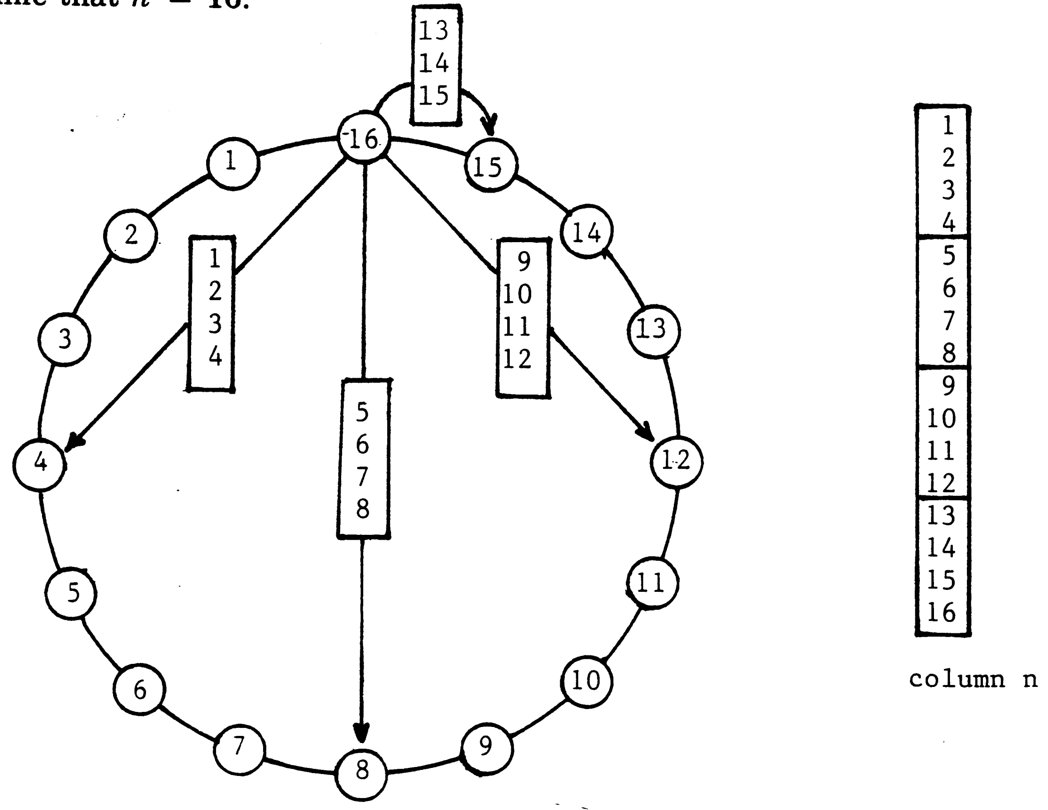

Figure 1.1

Of course a similar communication pattern can be repeated at all nodes. Therefore, in general during a given cycle, a node $P(j)$ will receive q packets, each of size $\frac{p}{q}$, before solving for the next variable and then successively computing and sending off q packets, each of size $\frac{p}{q}$ (actually, one of the packets, SUM, is of size $(\frac{p}{q}-1)$. After this node $P(j)$ will then modify the remaining $j-p$ components. Figure 1.2 illustrates the incoming communication pattern of node P(3), for $p = 16$ and $q = 4$.

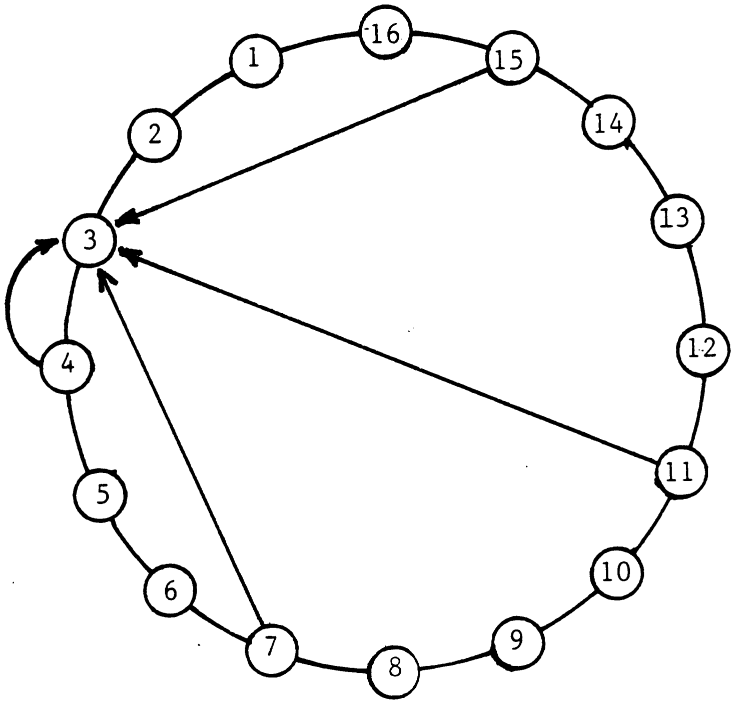

Figure 1.2

1.2 Description

Algorithm $PCTS^+$ is a generalization of $PCTS$; when $q = 1$ the two algorithms are identical. In general, for $q \geq 1$, a $\frac{p}{q} - 1$ array SUM passes around the ring going from $P(j)$ to $P(j-1)$ for $j = n{:}2$. Before SUM arrives, node $P(j)$ receives $q-1$ packets, each of size $\frac{p}{q}$, from $q-1$ predecessor nodes (node $P(j)$ modifies vector $PSUM$ using each packet it receives). When SUM arrives at node $P(j)$, $P(j)$ determines $x(j)$, modifies SUM using $\frac{p}{q}$ flops, and then forwards SUM to node $P(j-1)$. Node $P(j)$ then modifies $p - \frac{p}{q}$ elements of the local array $PSUM$ in packets of size $\frac{p}{q}$; the first packet is sent to node $P(j - \frac{p}{q})$, the second packet is sent to node $P(j - 2\frac{p}{q})$, and so on. Node $P(j)$ then proceeds to modify the remaining $j - p$ elements of array $PSUM$, using the corresponding elements of column j of U. Node $P(j)$ is then ready to begin again, this time with respect to $x(j-p)$.

We assume $\frac{p}{q}$ is an integer and define $\bar{p} = \frac{p}{q}$. The array w identifies $q-1$ of the recipients of the $\bar{p}$-packets. So for example, let k be the lowest numbered column on a node. Then $w(q-1) = P(k-\bar{p}$, $w(q-2) = P(k - 2\bar{p}$) and so forth.

```
Procedure $PCTS^+$   $(x[1:m], SUM(1:p), PSUM(1:n-p), U(1:n,[1:m]),$
                     $BUF(1:\bar{p}), w(1:q-1))$
    For $j=n:1$
        If myname = $P(j)$
            For $i=1:q-1$
                If $j \le n - i \times \bar{p}$
                    Receive $BUF(1:\bar{p})$
                    $PSUM(j-\bar{p}+1:j) \leftarrow PSUM(j-\bar{p}+1:j) + BUF(1:\bar{p})$
            Receive $SUM(1:\bar{p}-1)$   [if $j < n$]
            $x(j) \leftarrow (SUM(1) + PSUM(j))/U(j,j)$
            $SUM(1:\bar{p}-2) \leftarrow SUM(2:\bar{p}-1) - U(j-1:j-(\bar{p}-2), j) \times x(j)$
                        $+ PSUM(j-1:j-(\bar{p}-2))$
            $SUM(\bar{p}-1) \leftarrow -U(j-(\bar{p}-1), j) \times x(j) + PSUM(j-(\bar{p}-1))$
            Send $SUM(1:\bar{p}-1)$ to node $P(j-1)$   [if $j > 1$]
            For $i=q-1:1$
              If $j > (q-i) \times \bar{p}$
                $PSUM(j-(q-i+1) \times \bar{p}+1:j-(q-i) \times \bar{p})$
                    $\leftarrow PSUM(j-(q-i+1) \times \bar{p}+1:j-(q-i) \times \bar{p})$
                      $-U(j-(q-i+1) \times \bar{p}+1:j-(q-i) \times \bar{p},j) \times x(j)$
                Send $PSUM(j-(q-i+1) \times \bar{p}+1:j-(q-i) \times \bar{p})$ to node
                $w(i)$
            $PSUM(1:j-p) \leftarrow PSUM(1:j-p) - U(1:j-p, j) \times x(j)$
End
```

Most architectures will not guarantee a direct link between sender and receiver of the $\bar{p}$-packets. In our implementation on a hypercube computer, we make no attempt to optimize the route - we let the cube operating system take care of it (hopefully, some attempt is made to use idle nodes).

Obviously the choice of q plays an important role. As we mentioned before, a reasonable guiding principle is to choose q as large as possible subject to the constraint that when SUM arrives the receiving node is ready to process SUM. This notion can be formalized (under some simplifying assumptions on message traffic) and an optimal q can be determined analytically. We discuss this in Section 2. Next, we present numerical results using different values of q.

1.3 Numerical Results

Our numerical results were obtained using Intel iPSC hypercube computers. Experiments were performed using RM/Fortran, in double precision, under release 3.0 of the operating system. Experiments with $p > 16$ (i.e. $p = 64$, $p = 128$) were performed at Intel Scientific Computers, Beaverton, Oregon with the help of Cleve Moler. The largest linear system we could solve in this environment was approximately $n = 1700$. The $p = 16$ experiments were performed using the Intel iPSC housed in the Advanced Computing Facility at the Cornell Center for Theory and Simulation in Science and Engineering. This cube is outfitted with extra memory boards allowing systems of approximate order $n = 2000$ to be solved. Our test problems consisted of randomly generated linear systems.

In the graph in Figure 1.3 we plot execution time (y-axis) versus various d values (x-axis) where $q = 2^d$. The test problem is of size $n = 1000$ and $p = 128$.

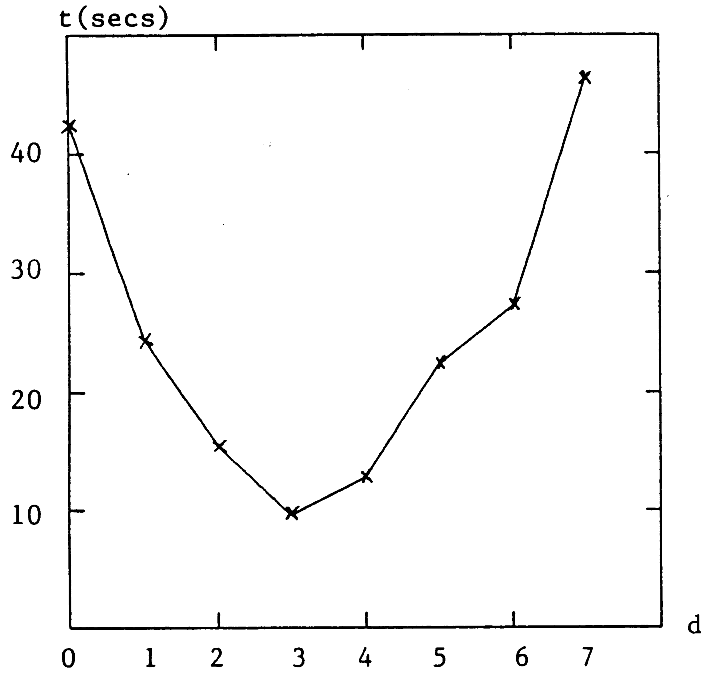

Figure 1.3 ($n=1000$, $p=128$, $q=2^d$)

Obviously the optimal choice for q in this case is $q = 2^3 = 8$. Notice that the difference between $q = 8$ and $q = 1$ is greater than a factor of 4.

As we show in Section 2, the optimal choice of q is dependent on the problem size. Nevertheless, Figure 1.4 illustrates that this dependency is relatively mild in this environment. In particular, $q = 8$ is almost uniformily superior for all $n \leq 1500$ when $p = 128$.

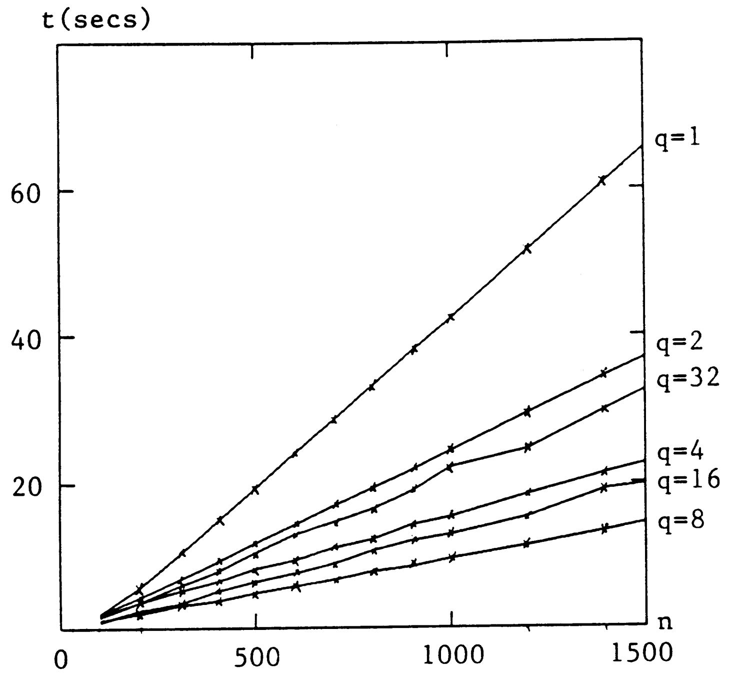

Figure 1.4 ($p=128$)

An important thing to notice is that $q = 1$ and $q = 8$ curves are diverging with n and therefore the significance of q increases with n (up to a point). When $n = 1500$ the $q = 8$ algorithm is approximately a factor 5 better than $q = 1$. Notice that for the $q = 1,2,4,8$ cases, $T(n)$ is clearly linear; for $q = 16,32$ the plot is not quite so true - we explain this phenomenon in Section 2.1.

Interestingly, Theorem 2.1 in Li and Coleman [1986], suggests that if n is below the threshold value (demarking the linear and quadratic regions) for a given p, then increasing the number of nodes p, while keeping n fixed, will cause the execution time to increase. It turns out that this is true in practise as well as theory. However, the introduction of q allows for some mitigation of this effect. Indeed, Theorem 2.3 in the next section indicates that if n is held constant and $\frac{p}{q}$ is fixed then we can expect the execution time to be constant with p. Figure 1.5 illustrates this remark.

In Figure 1.5 n is held constant at $n = 600$ and p is varied. In the first case ($q = 1$) the increase in computation time with p is evident. In the second case, q was varied always maintaining $\frac{p}{q} = 16$. In this case there is virtually no increase in computation time.

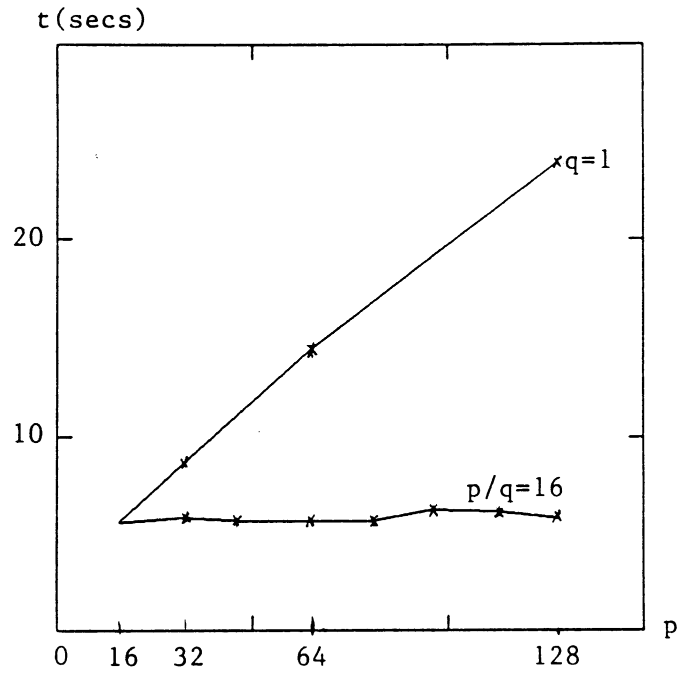

Figure 1.5 ($n = 600$)

2. Analysis

The purpose of this section is to derive an expression $T(n)$ for the running time of algorithm $PCTS^+$, measured in flops. This can easily be done, under a simplifying assumption, and the resulting formula can be used to guide the choice of the parameter q.

One flop reflects the time to execute the operation $y \leftarrow y + ax$ where a,x,y are scalars. We apply this definition a bit loosely, for simplicity. For example when the following statement in $PCTS^+$ is executed,

$$PSUM(j-\bar{p}+1:j) \leftarrow PSUM(j-\bar{p}+1:j) + BUF(1:\bar{p})$$

we count 1 flop per assignment even though there is no multiplication. Similarily, we count just 1 flop each pass through the loop

$$\begin{aligned} SUM(1:\bar{p}-2) \leftarrow SUM(2:\bar{p}-1) \; - \; & U(j-1:j-(\bar{p}-2), j)\times x(j) \\ & + PSUM(j-1:j-(\bar{p}-2)) \end{aligned}$$

even though there is an extra addition.

In the results to follow, we will assume that a cross-ring message of size $\bar{p} \triangleq \frac{p}{q}$ (a $\bar{p}$-packet in $PCTS^+$) takes time at most rt flops, where $r = \log_2(p)$. (Of course the minimum travel time is just t flops by the definition of t.) Furthermore, we assume that message forwarding accrues no overhead cost: a message is forwarded immediately by an intermediate node at zero cost. Finally, for simplicity we assume that $n = mp$ for some integer m.

It turns out that $T(n)$ is a linear function of n provided q and n are both less than threshold values. We derive this expression in 3 stages. First, in Lemma 2.1, we show that if q is sufficiently small, then in the first cycle, SUM is processed ($\bar{p}$ flops) immediately upon receipt by each of the nodes $P(n-1)$, ..., $P(n-(p-1))$. Second, Lemma 2.2 establishes that in the first cycle node $P(n-p) \equiv P(n)$ is also ready to process SUM immediately upon receipt, provided we make the additional assumption that n is less than a threshold value of n, n^*. Finally, in Theorem 2.3 it is shown that SUM is always processed immediately upon receipt by *every* node in *every* cycle; the expression for T is then easily deduced.

The following reasoning leads to an intelligent upper bound on q, q^*. Upon receipt of SUM a node $P(k)$ updates SUM in $\bar{p}$ flops and then forwards

SUM to a neighboring node on the ring. Immediately after this $\bar{p}$ elements of *PSUM* are modified by $P(k)$ and then this $\bar{p}$-packet is forwarded to node $P(k-\bar{p})$. Now, when the cycling vector *SUM* arrives at node $P(k-\bar{p})$ it is important that $P(k-\bar{p})$ is ready to process *SUM*. An obvious necessary condition for this is that the cross-ring $\bar{p}$-packet has been received and processed by $P(k-\bar{p})$ before *SUM* arrives. This consideration immediately leads to the inequality

$$2\bar{p}+rt \le \bar{p}t + (\bar{p}-1)\bar{p}$$

which yields the bound

$$q \le \frac{2p}{(3-t)+\sqrt{(t-3)^2 + 4rt}}. \tag{2.0}$$

Unfortunately (2.0) doesn't quite do the job when the last $\bar{p}$ columns of U, columns $\bar{p},\bar{p}-1, ..., 1$, are processed. The reason is that *SUM* can travel more quickly in this final strech since less processing is required at each node. Specifically, in this last cycle node $P(j)$ processes *SUM* in just j flops instead of the usual $\bar{p}$. Hence this final stretch of $\bar{p}$ nodes leads to the inequality

$$2\bar{p}+rt \le \bar{p}t+\frac{\bar{p}(\bar{p}+1)}{2}-1$$

which implies

$$q \le \frac{2p}{(3-2t)+\sqrt{(2t-3)^2 + 8(rt+1)}}. \tag{2.0'}$$

It is easy to see that (2.0') $\Rightarrow$ (2.0) whenever $q \le \frac{p}{2}$, and since this latter condition is highly reasonable, we define the upper bound for q,

$$q^* = \min\{\frac{p}{2}, \frac{2p}{(3-2t)+\sqrt{(2t-3)^2 + 8(rt+1)}}\}.$$

Lemma 2.1: If $q \le q^*$ then, in the first cycle, SUM travels

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

in time $p\bar{p} + (p-1)t$ flops (i.e. without delay).

Proof: We will show that when *SUM* arrives at node $P(n-k)$, $1 \le k \le p-1$, the receiving node is ready to process SUM. This will be the case if node $P(n-k)$ has completed processing the required $\left\lfloor \frac{k}{\bar{p}} \right\rfloor$ cross-ring $\bar{p}$-packets which it will receive in the first cycle. In particular we will show that the time between 2 $\bar{p}$-packet arrivals is always enough to allow the first to be processed before the second arrives. (This is true even when the second $\bar{p}$-packet is SUM itself.) Since processing the cross-ring $\bar{p}$-packets is the only work that node $P(n-k)$ must do before SUM arrives, our desired result will follow.

Assume that $\bar{p}$-packet i (say) is sent to node $P(n-k)$ at time t_i. Then, it will arrive at $P(n-k)$ by time t_i+rt. But $\bar{p}$-packet $i+1$ is not ready to be processed by node $P(n-k)$ until at least time $t_i+\bar{p}(\bar{p}+t) - 2\bar{p}$. (Note that $\bar{p}$-packet $i+1$ may refer to SUM.) Therefore, if

$$t_i+rt+\bar{p} \le t_i+\bar{p}(\bar{p}+t) - 2\bar{p} \tag{2.1}$$

then it follows that node $P(n-k)$ is ready to process $\bar{p}$-packet $i+1$ upon arrival. But (2.1) is clearly implied by the assumed bound on q and therefore the result follows. $\square$

Node $P(n-p)$ will be ready for SUM in the first cycle provided n is less than a threshold value n^*:

$$n^* = p(t+\bar{p}-1) + \bar{p}$$

Note that n^* depends on q; $n^*(q)$ is monotonically decreasing as q increases.

Lemma 2.2 : If $q \le q^*$ and $n \le n^*(q)$ then, in the first cycle, node $P(n)$ is ready to process SUM upon arrival (at time $p(t+\bar{p})$).

Proof: By Lemma 2.1, SUM arrives at node $P(n)$ at time $p(\bar{p} + t)$. We must now argue that node $P(n)$ has enough time to process column n of U, as well as the $(q-1)$ cross-ring $\bar{p}$-packets that arrive in staggered fashion, before SUM arrives. But, by algorithm $PCTS^+$, node $P(n)$ processes column n first before doing the $(q-1)$ $\bar{p}$-packet updates. Since it is clear that the last $\bar{p}$-packet, arriving from node $P(n-p)$, can be processed by $P(n)$ before SUM arrives (provided $P(n)$ isn't otherwise busy) we can, without loss of generality, count backwards. That is, assume that $P(n)$ processes this last $\bar{p}$-packet during time $[p(\bar{p}+t)-(\bar{p}-1),\ p(\bar{p}+t)]$. Similarily, we can assume that $P(n)$ processes the second last $\bar{p}$-packet during time $[p(\bar{p}+t)-(2\bar{p}-1),\ p(\bar{p}+t)]$ (By definition of q^*, it is clear that $P(n)$ has processed this packet by time

$p(\bar{p}+t)-(2\bar{p}-1))$. Continuing in this fashion, assume, without loss of generality, that the interval $[p(\bar{p}+t)-((q-1)\bar{p}-1),\ p(\bar{p}+t)]$ is used to process all $(q-1)$ cross-ring $\bar{p}$-packets. Hence we must only show that $P(n)$ has finished processing column n by time $p(\bar{p}+t)-(q-1)\bar{p}+1$. But node $P(n)$ is finished processing column n of U at time n and with $n \le n^*(q)$ we obtain

$$n \le p(\bar{p}+t)-((q-1)\bar{p}-1) \qquad \Box$$

Theorem 2.3 : If $q \le q^*$ and $n \le n^*(q)$ then the total running time of PCTS$^+$ satisfies

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

Proof: By Lemmas 2.1 and 2.2, *SUM* is processed immediately upon receipt by nodes

$$P(n-1),\ P(n-2),\ \ldots,\ P(n-p) = P(n)$$

in the first cycle. But if node $P(n-p)$ is free when *SUM* arrives in the first cycle, then it is clear that every node will be free when *SUM* arrives in subsequent cycles (except possibly when the last $\bar{p}$ columns, column $\bar{p}$, ..., column 1, are being processed). The reason is simple: such nodes are in the same situation as node $P(n)$ is in the first cycle. In particular, each such node must process its current column of U and then the $(q-1)$ arriving $\bar{p}$-packets before *SUM* returns. But since the size of the columns of U is diminishing, and since $P(n)$ had enough time in the first cycle (by Lemma 2.2), it follows that each node will have sufficient time to do this.

However, it is necessary to consider the last $\bar{p}$ columns (i.e. columns $\bar{p}, \bar{p}-1, \ldots, 1$) separately because in this stretch the nodes $P(\bar{p}), \ldots, P(1)$ do less work in processing *SUM*. In particular, in this final stretch *SUM* requires i flops at node $P(i)$, $1 \le i \le \bar{p}$. Hence it is conceivable that *SUM* arrives at node $P(j)$, say, with $1 \le j \le \bar{p}$, before $P(j)$ is ready. But it is easy to see that this cannot be the case: the bound on q, q^*, ensures that each node $P(j)$ has enough time to process the $\bar{p}$-packet from $P(j+p)$ before *SUM* arrives. Hence it follows that *SUM* is processed immediately upon arrival at all nodes during every cycle.

We are now ready to compute T and thus prove the theorem. The total running time T is just the time for *SUM* to cycle around the ring m times. Since every node is always ready to process *SUM* upon arrival, it is straightforward to compute this cycling time. With respect to each column

j, $\bar{p}<j\leq n$, we will associate the time $\bar{p}+t$, which represents the time required to process *SUM* plus the time for *SUM* to travel to the next node on the ring. Similarily, for $2\leq j\leq \bar{p}$ column j is charged $j+t$ flops; column 1 is charged just 1 flop. Therefore, the total time T is just the sum of the charges:

$$T = (n-\bar{p})(\bar{p}+t) + (\bar{p}-1)t + \sum_{i=1}^{\bar{p}} j$$

which yields the result. $\square$

2.1 Remarks on Theorem 2.3

The main practical importance of Theorem 2.3 is that it can be used to guide the choice of q. Specifically, T decreases as q increases (for fixed p,n,t) and therefore the optimal value of q, q_*, is attained by increasing q until either $n = n^*(q)$ or $q = q^*$, whichever comes first. Therefore q_* is given by

$$q_* = \max\{1, \min\{q^*, [\frac{p(p+1)}{n-p(t-1)}]^+\}\}$$

Let us now consider how this theoretical estimate matches our computational experience reported in Section 1. We have estimated $t \cong 40$ on the Intel iPSC (release 3.0). Using this estimate and $p = 128$ we obtain $q^* = 19$. Hence if $n = 1000$ then $q_* = q^* = 19$. Considering the results given in Figures 1.3 and 1.4, we see that this estimate is a bit high: i.e. the numerical results clearly indicate that $q = 8$ is superior to $q = 16$. Indeed, this overestimation is typical and can be explained, we think, by considering our assumptions. Specifically, in order to make the analysis tractable, we have assumed that the forwarding of cross-ring messages occurs at no cost: i.e. there is no interruption in the computational work performed by the forwarding nodes. However, as q increases this assumption becomes increasingly questionable since, in truth, forwarding nodes will be interrupted. Hence it is best to regard the theoretical value of q_* as an upper bound on q. Indeed, it is the violation of this assumption as q increases that accounts for the deviation from linearity exhibited by the $q = 16,32$ curves in Figure 1.4.

If $p = 16$ then $q^* = 4.6$ and $q_* = q^* = 4.6$ provided $n \le 900$. This theoretical estimate is a bit high once again, since, experimentally, we have found that the best choice for q with n in this range is $q = 2$. If $p = 64$ then $q_* = q^* = 11$ for $n \le 3000$. However, experimentally we obtain the results reported in Figure 2.1.

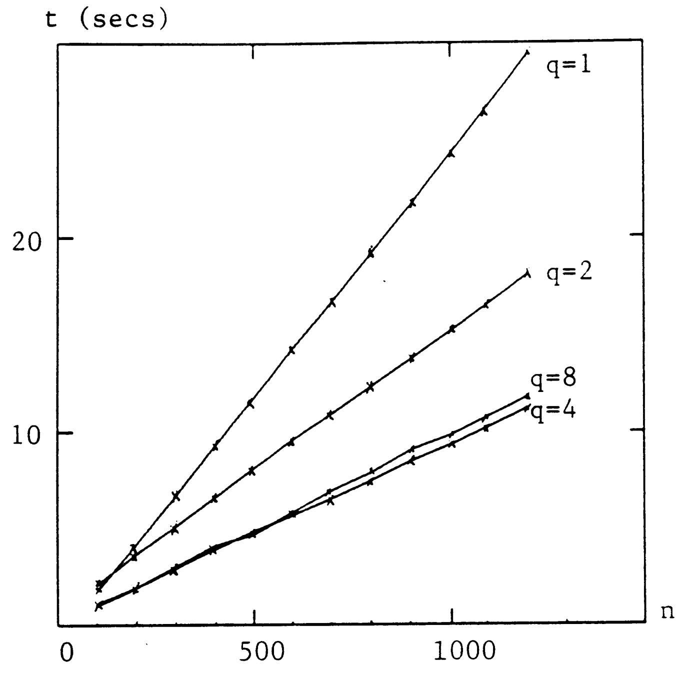

Figure 2.1 ($p = 64$)

Clearly the experimental best choice for q is $q = 4$ with $q = 8$ a very close second.

It is interesting to note that the definition of the upper bound q^* does not depend on on n. Hence q_* will be invariant with respect to n for some range of values of n. For example, considering the $p = 128$ example mentioned above, we will have $q_* = 19$ for all $n \le 5861$ (on the Intel iPSC without extra memory, this figure greatly exceeds the storage capacity). This is of considerable practical significance because it suggests that using a fixed value of q may be a reasonable thing to do despite the apparent dependence of q_* on n. Hence, in practice, it may not be necessary to compute q_* for every problem in which n changes - rather, a good value of q can be chosen given p and t. (Of course, as we mentioned in the preceding paragraph, this choice should probably be somewhat less than the theoretical value q_*).

Theorem 2.3 is very much related to Theorem 2.1 in Li & Coleman [1986]. Indeed, if $q = 1$ then Theorem 2.3 reduces to precisely the first half of the latter result. Theorem 2.3 says nothing about the case when $n > n^*(q)$. The reason is that Theorem 2.1 (Li & Coleman [1986]) covers the interesting ground. In particular, if $n > n^*(q=1)$ then this theorem says that T is a quadratic function of n:

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

The case where $n \leq n^*(q=1)$ but $n > n^*(q>1)$ is not really of interest: the practical choice is $q = 1$ in this case.

3. A Rectangular Triangular Solver

In this section we consider an improvement to algorithm $PCTS^+$; however, since our numerical experiments show that the new algorithm, $RPCTS^+$, offers only a modest gain in efficiency, we will be quite brief in our presentation. Nevertheless, there is no question that $RPCTS^+$ is never worse and sometimes better than $PCTS^+$; the improvement may be significant in some environments.

The basic idea behind $RPCTS^+$ is based on the observation that columns of U diminish in size from right to left: therefore, the workload endured by $PCTS^+$ is not evenly balanced. Moreover, it is quite possible to postpone some of the work on the larger columns until later and thereby achieve a rectangular work distribution (instead of triangular). In particular, in every cycle of $RPCTS^+$ each node processes a 'column' of U of about size $\frac{n}{2}$.

The rectangular triangular solver follows. The parameter h is usually chosen to be approximately equal to $\frac{n}{2}$. Indeed this is the choice that leads to the rectangular work distribution and is always our choice in our reported experiments.

```
Procedure $RPCTS^+$ ($x[1:m]$, $SUM(1:p)$, $PSUM(1:n-p)$, $U(1:n,[1:m])$,
                    $BUF(1:\bar{p})$, $w(1:q-1)$)
    $l = m$
    For $j = n:1$
        If myname $= P(j)$
            For $i = 1:q-1$
                If $j \leq n - i \times \bar{p}$
                    Receive $BUF(1:\bar{p})$
                    $PSUM(j-\bar{p}+1:j) \leftarrow PSUM(j-\bar{p}+1:j) + BUF(1:\bar{p})$
            Receive $SUM(1:p-1)$    [if $j < n$]
            $x(j) \leftarrow (SUM(1) + PSUM(j))/U(j,j)$
            $SUM(1:\bar{p}-2) \leftarrow SUM(2:\bar{p}-1) - U(j-1:j-(\bar{p}-2), j) \times x(j)$
                        $+ PSUM(j-1:j-(\bar{p}-2))$
            $SUM(\bar{p}-1) \leftarrow -U(j-(\bar{p}-1), j) \times x(j) + PSUM(j-(\bar{p}-1))$
            Send $SUM(1:\bar{p}-1)$ to node $P(j-1)$    [if $j > 1$]
            For $i = q-1:1$
              If $j > (q-i) \times \bar{p}$
                $PSUM(j-(q-i+1)\times\bar{p}+1:j-(q-i)\times\bar{p})$
                    $\leftarrow PSUM(j-(q-i+1)\times\bar{p}+1:j-(q-i)\times\bar{p})$
                      $-U(j-(q-i+1)\times\bar{p}+1:j-(q-i)\times\bar{p}, j) \times x(j)$
              Send $PSUM(j-(q-i+1)\times\bar{p}+1:j-(q-i)\times\bar{p})$ to node $w(i)$
        $PSUM(j-h+1:j-p) \leftarrow PSUM(j-h+1:j-p)$
                        $- U(j-h+1:j-p, j) \times x(j)$
        If $j \leq n-h+p$
            For $i = m:l$
                $PSUM(j-2p+1:j-p) \leftarrow PSUM(j-2p+1:j-p)$
                                 $-U(j-2p+1:j-p, i) \times x(i)$
            $l = l-1$
 End
```

It is important to realize that $RPCTS^+$ can potentially improve on $PCTS^+$ only when $n > n^*(q)$; otherwise, the algorithms have exactly the same running time for all feasible values of q. However, it is possible to extend the linear region beyond $n^*(q)$ with $RPCTS^+$. The reason for this is simply that the definition of the linear region is driven by the size of the largest column processed - the 'column' sizes used by algorithm $RPCTS^+$ are all roughly $\frac{n}{2}$ which contrasts with a maximum column size of n in the case of $PCTS^+$. The following theorem formalizes this notion: notice that the bound on n is roughly twice $n^*(q)$; the expression for T is identical to that for $PCTS^+$. (We omit the proof - it is very similar to the proof of Theorem 2.3.)

Theorem 3.1: If $q \le q^*$ *and* $n \le 2p(t+\bar{p}-2)$ *then*

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

3.1 Numerical Results

In Figure 3.1 we compare $RPCTS^+$ with $PCTS^+$ for $p=16$ and $q=1$. Notice that the graphs are essentially indistinguishable up until $n \cong 1000 \cong n^*(q=1)$. (Theoretically this breakpoint occurs at n = 900.) At this point $RPCTS^+$ continues in a linear fashion while $PCTS^+$ begins its quadratic phase.

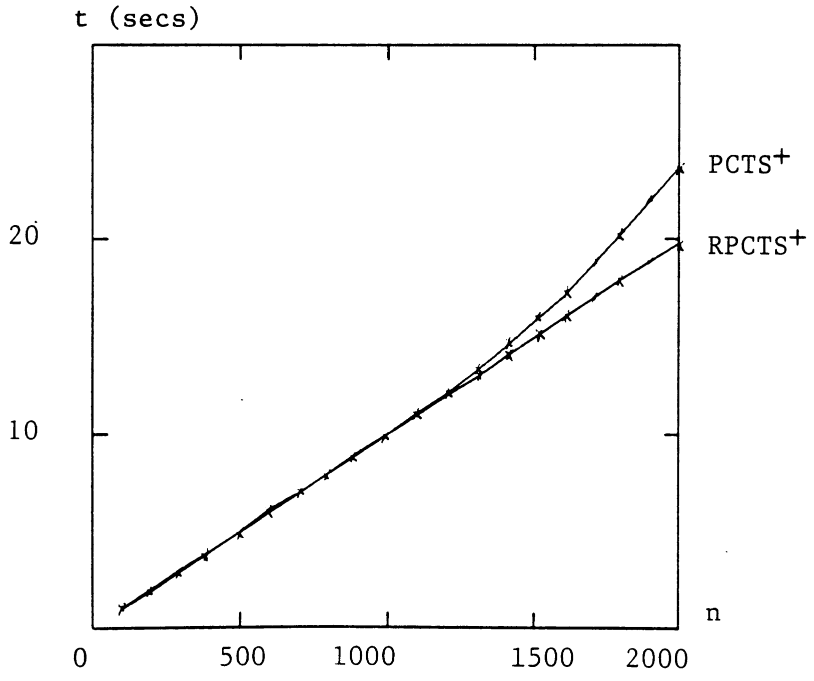

Figure 3.1 ($p=16$, $q=1$)

4. Summary and Conclusions

We have presented and analyzed a generalization of the Li-Coleman[1986] algorithm for solving triangular systems of equations on a multiprocessor. We have assumed that the columns of the matrix are distributed to the nodes in a wrap fashion. We note in passing that a similar algorithm can be constructed for the row-distributed case. This new solver is effective even when $\frac{n}{p}$ is modest whereas the original solver degrades in performance as $\frac{n}{p}$ decreases. The new solver is applicable on a distributed-memory multiprocessor that allows for a ring embedding; however, it is most reasonable when there is additional inter-processor connectivity, beyond that of a ring. The exact nature of this connectivity is unimportant though our experiments have been restricted to a hypercube multiprocessor.

Under a slightly unrealistic assumption on cross-ring traffic (i.e. nodes forward messages at no cost) the proposed method is analytically tractable. This analysis reveals that up to threshold values of n and parameter q, the running time is a linear function of n. More importantly, the analysis yields a theoretically optimal choice for the parameter q which, in practice, serves as a very useful upper bound on q.

Finally, we note that when $\frac{n}{p}$ is sufficiently large then the new algorithm reduces to the original Li-Coleman[1986] algorithm which is quite effective in such circumstances.

Acknowledgements:

We are especially grateful to Richard Chamberlain who suggested to us that the original Li-Coleman algorithm could be improved in a manner similar to the procedure described here. We thank Intel Scientific Computers of Beaverton, Oregon, and Cleve Moler in particular, for allowing us to visit and conduct many of our numerical experiments there. Finally, the balance of our numerical experimentation was performed with the assistance of the Advanced Computing Facility at the Cornell Center for Theory and Simulation in Science and Engineering, which is supported by the National Science Foundation and New York State.

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

M.T. Heath [1986], *Private Communication.*

G. Li & T. Coleman [1986], *A Parallel Triangular Solver for a Hypercube Multiprocessor,* Technical Report TR 86-787, Dept. of Computer Science, Cornell University, Ithaca, NY.

C. Moler[1986a], *Private Communication.*

C. Moler[1986b], *Numerical Comparisons of Triangular Solvers on the Intel iPSC,* presented at the Second Conference on Hypercube Multiprocessors, Knoxville, Tennessee, Sept. 29-Oct.1.

C. Moler [1986c], *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.