High Performance Computing II
M. Durand and F. El Dabaghi (Editors)
© Elsevier Science Publishers B.V. (North-Holland), 1991

The Parallel Solution of Triangular Systems of Linear Equations

Rudnei Dias da Cunha[a] and Tim Hopkins[b]

[a]Computing Laboratory, University of Kent at Canterbury, U.K.
Centro de Processamento de Dados, Universidade Federal do Rio Grande do Sul, Brasil
Phone: +44-227-764000 Fax: +44-227-762811
E-mail: rdd@ukc.ac.uk

[b]Computing Laboratory, University of Kent at Canterbury, U.K.
E-mail: trh@ukc.ac.uk

Abstract

We present a parallel algorithm for solving triangular systems of linear equations on distributed-memory multiprocessor machines.

The parallelism is achieved by partitioning the rows of the coefficient matrix in segments of a fixed size and distributing these sets in a wrap-around fashion among the processors, connected in a ring. The granularity of the algorithm is controlled by varying the segment size.

The algorithm is compared to other previously published algorithms and is shown to provide a better performance.

1. INTRODUCTION

We consider the solution of a non-singular lower triangular system of N linear equations,

$$Lx = b. \tag{1}$$

The solution of an upper triangular system is similar and will not be discussed further.

The parallel solution of triangular systems has been the subject of discussion of many authors. Heath and Romine [4] present three different families of solvers, namely the "fan-in" and "fan-out", wavefront and cyclic algorithms. Their algorithms were implemented on a hypercube architecture. Eisenstat et al. [2] present a modified version of the cyclic algorithm presented in [4]. Li and Coleman ([5], [6]) present column-oriented algorithms. Finally, Di Zitti et al. [1] present a folding-based technique to map a wavefront algorithm onto a 2×2 square mesh of transputers.

Our algorithm is based on a row-oriented forward-substitution scheme where the matrix L has sets of its rows distributed in a wrap-around fashion among the processors. In the sections that follow, we will describe the mechanics of the algorithm and present some experimental results; finally we compare our algorithm with other previously published algorithms.

2. THE PLTSLES ALGORITHM

2.1. First considerations

Our "Parallel Lower Triangular System of Linear Equations Solver" (PLTSLES) algorithm is based on the forward-substitution scheme to solve system (1)

$$x_i = (b_i - \sum_{j=0}^{i-1} l_{ij} \times x_j)/l_{ii}, \quad i = 1, 2, \ldots, N \tag{2}$$

The forward-substitution scheme is not obviously suitable for parallelization, since each x_i can only be computed after all the x_j, $j < i$ have been computed. However, we can compute the summation terms used in (2) in parts.

The partial computation of sums is not in itself sufficient to guarantee that it will make scheme (2) suitable for parallel computation. Suppose that we have N rows and P processors linked in a ring. If we partition the rows of L as sets of contiguous rows among the processors in the sequence $0, 1, \ldots, P-1$, we can compute the solution of (1) using partial sums to compute (2). We note that this distribution of data will not allow an efficient parallel algorithm. As soon as a given processor has computed all its $x-$variables it will have no data to work on, leading to the case where the parallel algorithm is slower than its serial counterpart due to the communication overheads. The solution to this problem is achieved by using a wrap-around scheme.

2.2. Description of the algorithm

The sets of rows of L are distributed among P processors in a wrap-around fashion. We will refer to these sets as "segments".

A segment is a pair of the form $(a, b), a \leq b$, where a and b denote the first and last rows of the matrix which make up the segment. Each segment has at most δ rows, the last segment having less rows if N is not exactly divided by δ. Let $S = \lfloor N/\delta \rfloor + sign(N \bmod \delta)$ represent the number of segments in which the rows of L are divided, s be a segment number $(0 \leq s \leq S-1)$, and $S-1$ identify the last segment.

A given processor p will receive a segment s if $p \bmod s = 0$ and it will operate on at most $\lfloor S/P \rfloor + sign(S \bmod P)$ segments. We will denote by $\mathcal{R}_p$ the set of segments assigned to each processor. In order to ensure that all processors will receive at least one segment, we require $S \geq P, \forall P$ (condition 1).

If we make $x \equiv b$ before applying the forward-substitution scheme (2), we can rewrite that scheme as

$$x_i = (x_i - \sum_{j=0}^{i-1} l_{ij} \times x_j)/l_{ii}, \quad i = 1, 2, \ldots, N \tag{3}$$

As soon as processor p receives a segment (c, d) of the x-vector, it can start computing the parts of the summation of (3) associated with the variables $x_c, \ldots, x_d$ for all the remaining segments in its set $\mathcal{R}_p$. A processor can only completely update all the variables in a given segment (a, b) after it has received all variables x_j such that $j < a$.

Once a segment has all its variables computed, this segment can be sent to all the other processors, and the processor that has sent it must now compute the parts of the summation of (3) for all its own remaining segments of variables.

We thus have three distinct operations to be done; the first when a processor is waiting for some values and partially updating its remaining segments, the second when a processor is updating its current segment and the third, when it is partially updating the segments left in $\mathcal{R}_p$ after the current segment has been computed. We can now state the computations carried on during each of these operations as

$$x_i = x_i - \sum_{j=c}^{d} l_{ij} \times x_j\,, \tag{4}$$

upon receiving a segment (c, d), for all remaining segments of $\mathcal{R}_p$,

$$\left\{ \begin{array}{l} x_a = x_a/l_{aa} \\ x_i = (x_i - \sum_{j=a}^{b} l_{ij} \times x_j)/l_{ii}; \quad i = a+1, \ldots, b\,, \end{array} \right. \tag{5}$$

where (a, b) means the current segment, and

$$x_i = x_i - \sum_{j=a}^{b} l_{ij} \times x_j \tag{6}$$

for all remaining segments of $\mathcal{R}_p$, after the segment (a, b) has been computed.

We shall now turn our attention to the number of segments that each processor must receive and the number of processors to which a segment must be sent. As soon as a segment is computed, it is sent to $P - 1$ processors; at the end of the algorithm, all processors will store the complete x vector.

For a given segment (a, b), with segment number $s_a = \lfloor a/\delta \rfloor$, we define the number of segments that will be received by the processor hosting this segment, $NSR(s_a)$, as

$$NSR(s_a) = \left\{ \begin{array}{ll} P-1 & \text{if } s_a \geq P-1 \\ s_a & \text{otherwise} \end{array} \right. \tag{7}$$

The P processors are connected in a ring and are labelled in the sequence $0, 1, \ldots, P-1$. Due to the wrap-around distribution, if a segment (a, b) hosted on a processor p is needed by q other processors to update segments $(c, d), (e, f), \ldots$ such that $b < c < e < \cdots$, these processors must lie "ahead" of p, i.e. q is the set $\{p+1, p+2, \ldots\}$. Mapping this set to the ring labelling, we can say that a segment (a, b) will be sent to $P - 1$ processors labelled as $(p + j) \bmod P$ for $\quad j = 1, \ldots, P-1$.

When a processor has computed all its variables, it must receive the variables that have yet to be computed by the other processors. The number of segments that it will receive is given by $NSR(S - 1 - s_l)$ where s_l is the segment number of the last segment stored in this processor, defined in the same way as s_a. We note that the processor that has computed the last segment does not receive any other segments.

The PLTSLES algorithm can thus be stated as

```
PLTSLES (p): for each segment $(a, b)$ in $\mathcal{R}_p$
                 repeat $NSR(s_a)$ times
                   receive segment $(c, d)$
                   apply (4)
                 apply (5)
                 send $(a, b)$ to the next $P - 1$ processors
                 apply (6)
               repeat $NSR(S - 1 - s_l)$ times
                 receive segment $(c, d)$
```

3. EXPERIMENTAL RESULTS

The actual implementation of PLTSLES was on a MEiKO Computing Surface, a transputer-based parallel computer. The algorithm was coded in Occam 2 using 64-bit floating-point arithmetic, running on T800 transputers.

The PLTSLES algorithm was initially tested on systems of order $N = 128, 256, 512$ and 1024. The test system used is defined as

$$\begin{aligned} l_{ij} &= \begin{cases} (i-1) * N + (j-1); & j = 1, 2, \ldots, i-1 \\ 1; & j = i \end{cases} \\ b_i &= \sum_{j=1}^{i} l_{ij} \end{aligned} \tag{8}$$

We tested each system (8) with different δ values, namely $1, 2, 4, 8, 16$ and 32, using rings of processors with $P = 2, 4, 8$ and 16. We will refer to the execution time as t.

3.1. Segment size

Figures 1 and 2 show the curves $\delta \times t$, for $P = 2, 4, 8,$ and 16. Each figure presents the four curves obtained for each different N value.

The minimum execution time is obtained when $\delta = 8$, although the increase in execution time for $\delta > 8$ is very slight.

3.2. Number of processors

Figures 3 and 4 shows the curves $P \times t$, for $N = 128, 256, 512$ and 1024. Each figure presents the six curves obtained for each different δ value. Note the different time scales in the graphs.

The δ-curves become closer to each other as N is increased, from which we can conclude that for large N, the selection of δ is not important (provided that it is not too small), and that $S \geq P$.

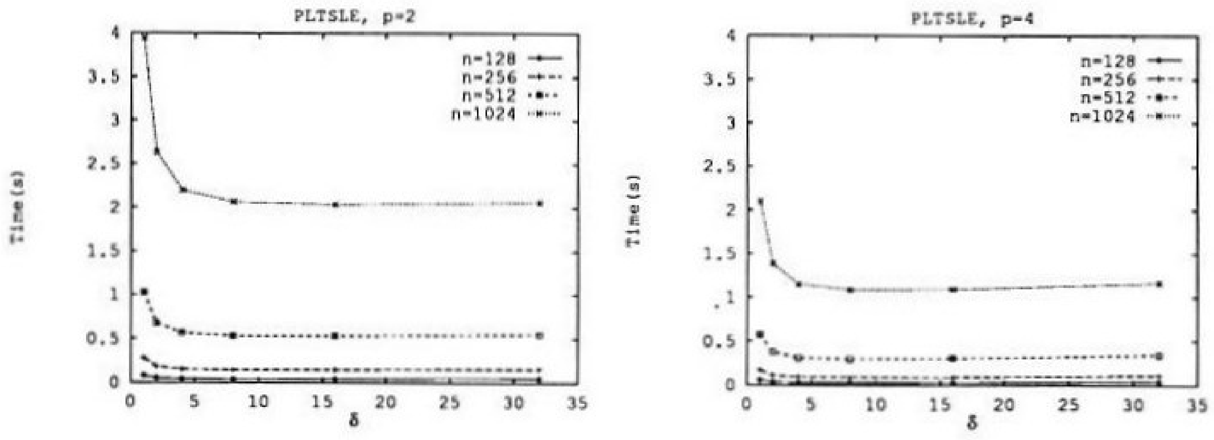


Figure 1: $\delta \times t$, (a) $P = 2$, (b) $P = 4$.

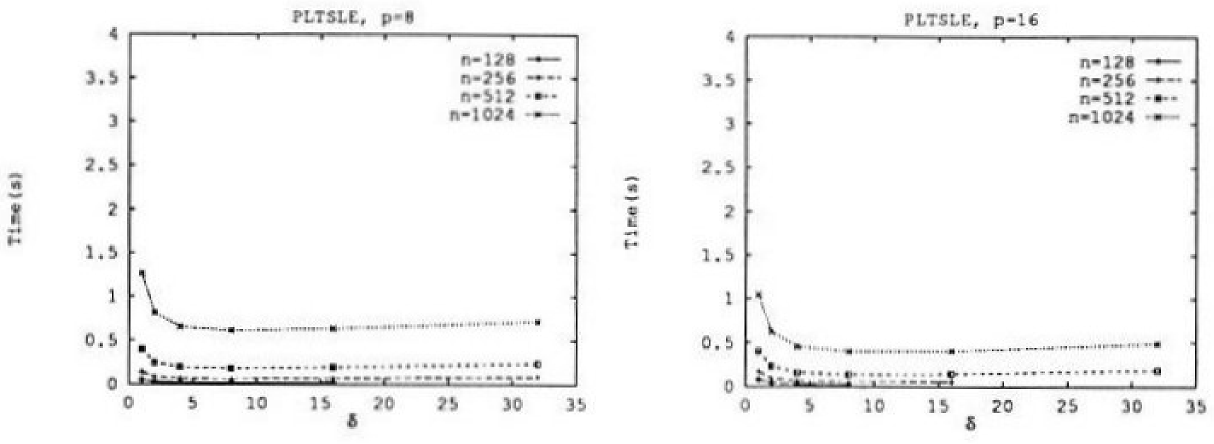


Figure 2: $\delta \times t$, (a) $P = 8$, (b) $P = 16$.

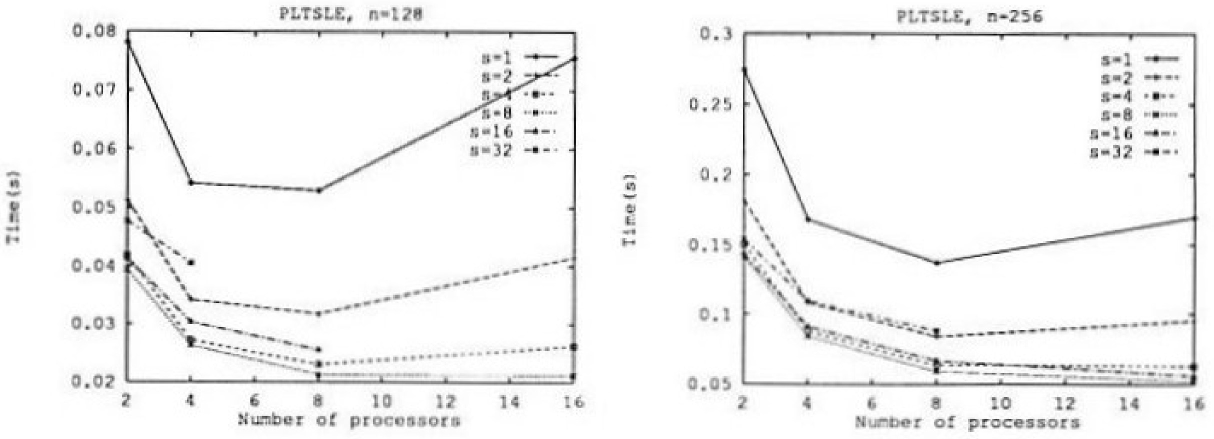
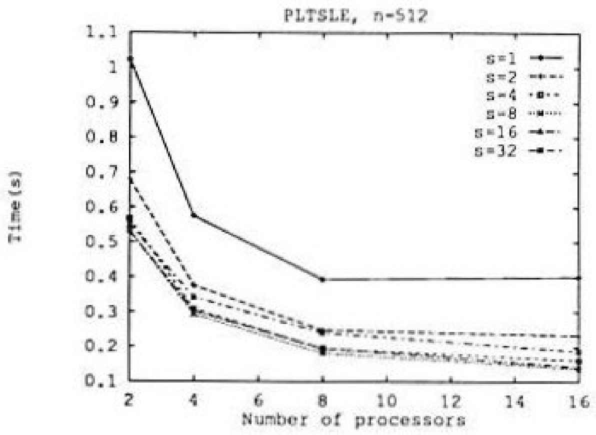


Figure 3: $P \times t$, (a) $N = 128$, (b) $N = 256$.



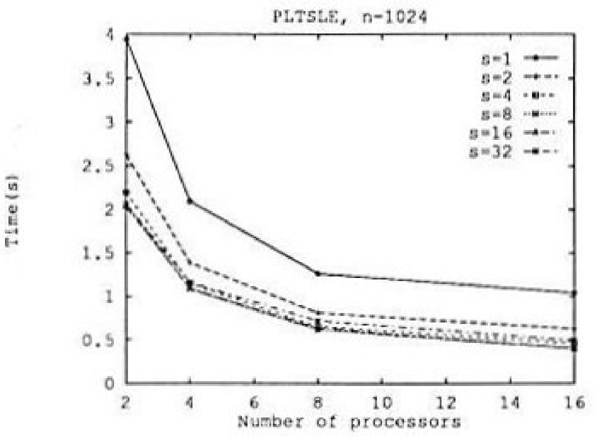
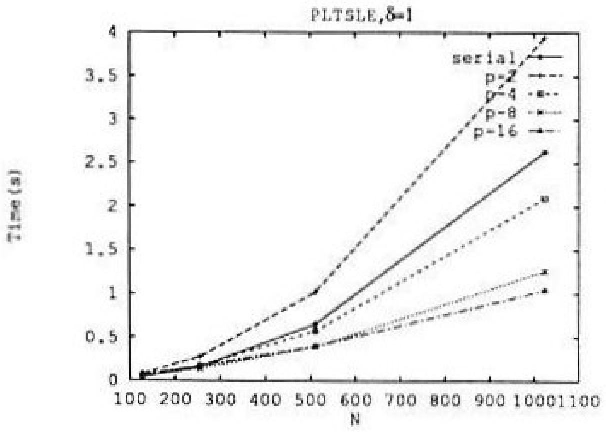


Figure 4: $P \times t$, (a) $N = 512$, (b) $N = 1024$.



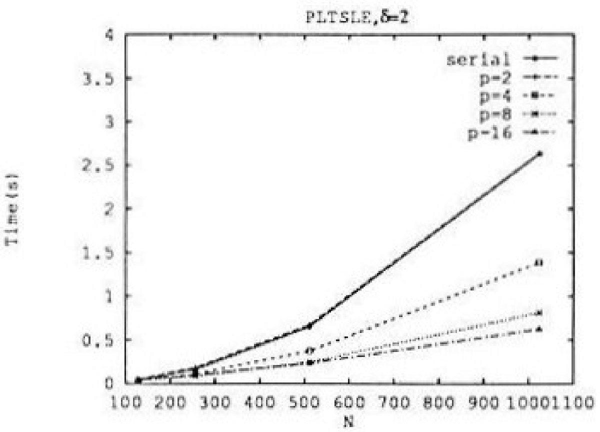


Figure 5: $N \times t$, (a) $\delta = 1$, (b) $\delta = 2$.

3.3. System order

Figures 5-7 shows the curves $N \times t$, for $\delta = 1, 2, 4, 8, 16$ and 32. Each figure shows the five curves obtained for each P value, together with the timings obtained by a serial algorithm which implements the forward-substitution scheme (3) on a single transputer.

We note that when $\delta = 1$, the PLTSLES algorithm is slower on two processors than the serial algorithm. The P-curves flatten as P increases, with respect to N, which means that for large N, a large number of processors will provide a good performance.

3.4. Further experiments

In order to analyse the performance of PLTSLES with larger systems, we solved the system

$$\begin{aligned} l_{ij} &= 1; \quad i = 1, 2, \ldots, N,\ j = 1, 2, \ldots, i \\ b_i &= N^2; \quad i = 1, 2, \ldots, N \end{aligned} \tag{9}$$

As L and b are constants they were not stored, enabling us to run the PLTSLES with larger systems. System (9) was tested for $N = 2^k$, $k = 10, 11, \ldots, 15$.

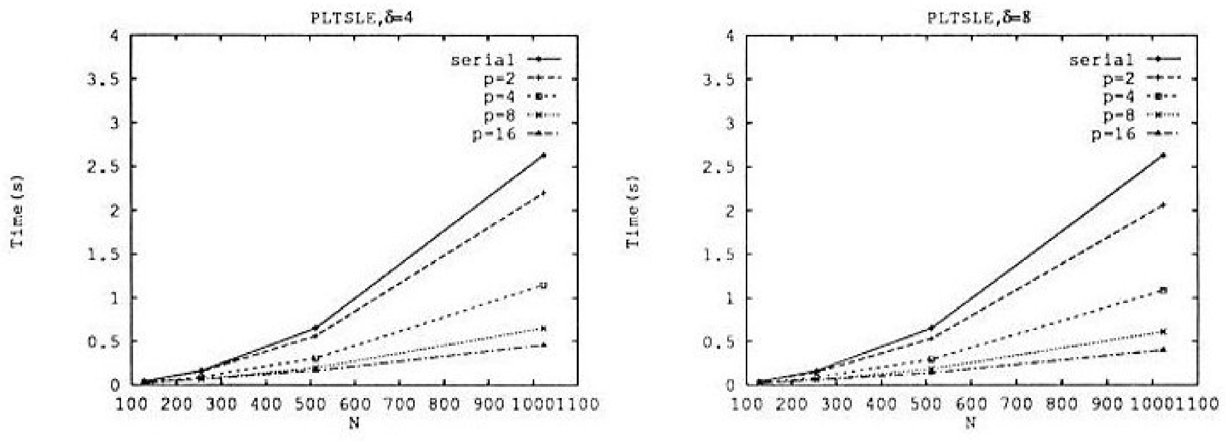


Figure 6: $N \times t$, (a) $\delta = 4$, (b) $\delta = 8$.

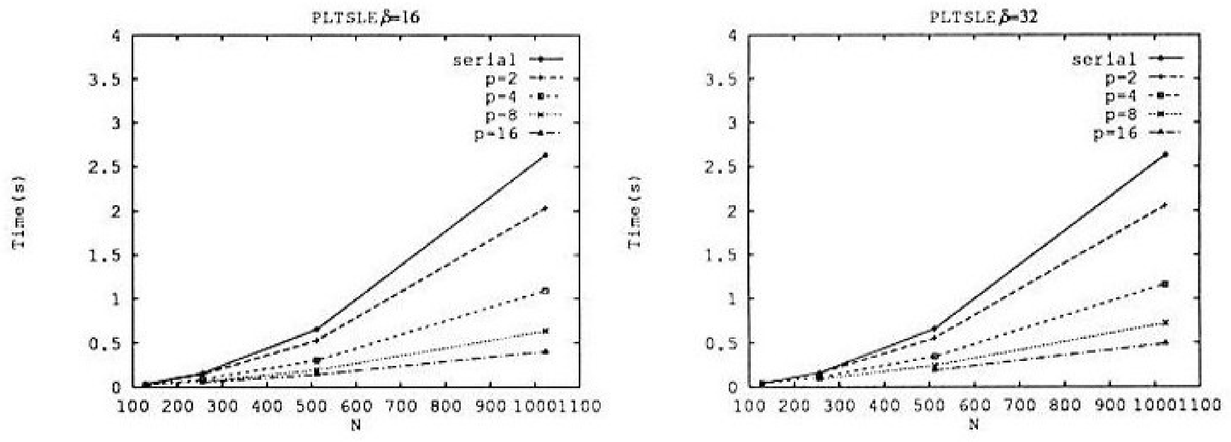


Figure 7: $N \times t$, (a) $\delta = 16$, (b) $\delta = 32$.

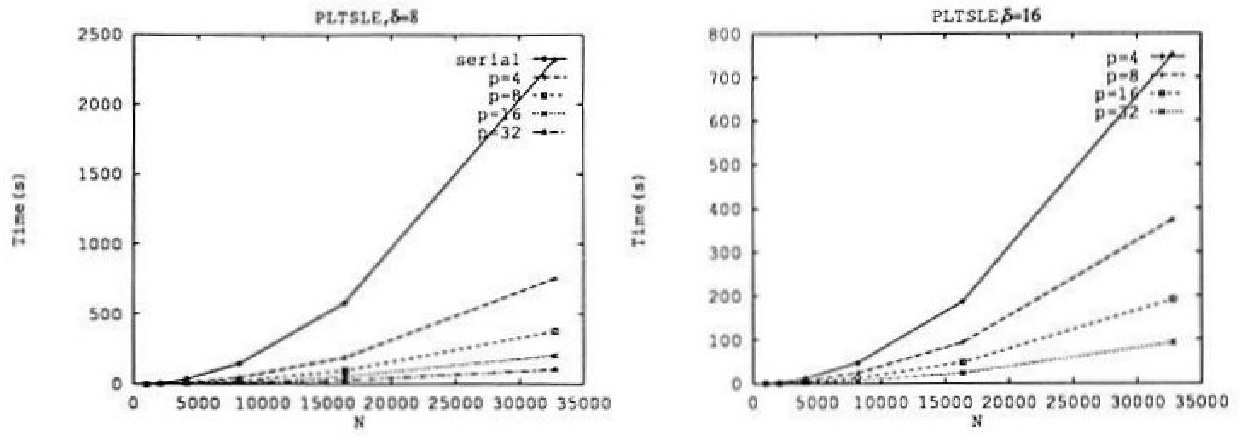


Figure 8: $N \times t$, (a) $\delta = 8$, (b) $\delta = 16$.

Figure 8 show the graphs of $N \times t$, for $\delta = 8$ and 16. The first graph shows the curves obtained by a serial algorithm and $P = 8, 16$ and 32; the second graph does not show the curve for the serial algorithm for readability. We note that PLTSLES performs better when solving a large system. The optimum δ value depends on the values of N and P. When P and N are small, the execution time varies slightly with respect to δ; when P and N are large, we notice a higher reduction in the execution time. As N gets larger, the selected value of δ makes no difference to the execution time. The efficiency achieved by PLTSLES for large N is in excess of 80%.

4. COMPARISON WITH OTHER IMPLEMENTATIONS

We compared our PLTSLES algorithm with the row-wavefront algorithm proposed by Heath and Romine (hereafter referred to as "RW-H&R") [4], implemented on the NCUBE and Intel iPSC hypercube computers, the "PRTS" algorithm developed by Li and Coleman [5], implemented on the iPSC, and the optimized-wavefront algorithm due to Di Zitti et al. (hereafter referred to as "OW-DiZitti") [1], which was implemented on a mesh of transputers.

4.1. PLTSLES and the RW-H&R algorithm

The RW-H&R algorithm is very similar to the PLTSLES. Both algorithms share the same basic idea to obtain parallelism; that by breaking down the solution into parts we may overlap the work on these parts. The data is distributed among the processors in a wrap-around fashion in both algorithms. The granularity in both is controlled by the same parameter, the size of the segment containing the components of the x vector, although this is employed in different ways. Another similarity is that the processors are interconnected in a ring for both implementations.

The two main differences between PLTSLES and RW-H&R are

1. On PLTSLES, the components of the x vector are sent to other processors in segments of a fixed size, except perhaps the last segment; on RW-H&R, the segments grow in size to a maximum stipulated size;

2. On PLTSLES, the segments travel a distance of at most $P-1$ processors; on RW-H&R, the segments are kept travelling around the ring throughout the execution of the algorithm.

We compared the PLTSLES to the RW-H&R for a system of order $N = 500$, with $P = 16$, for different segment sizes. The data for the RW-H&R were taken from [4]. Figure 9 shows the two curves obtained for this problem.

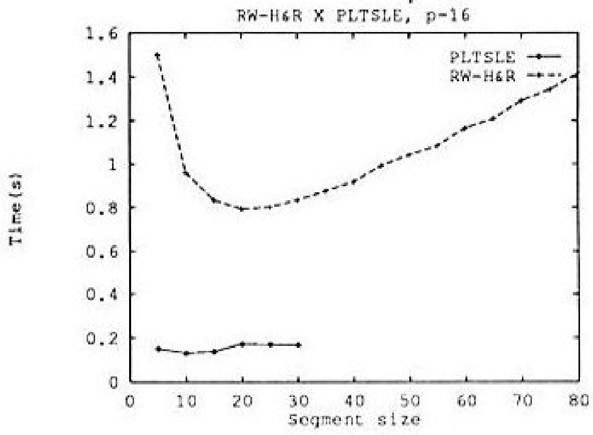


Figure 9: RW-H&R $\times$ PLTSLES, $N = 500$, $P = 16$.

We note that due to the difference in the way the segment size is used in both algorithms, the maximum segment size that could be used with PLTSLES for this specific problem is 30.

Disregarding the time in itself, due to the different target machines used, we note the different format of the curves. The PLTSLES has an almost constant behaviour, while the minimum execution time is obtained when $\delta = 10$. The curve for RW-H&R has a logarithmic-like format, and we can see that, when the segment size exceeds the optimum (in this problem, 20), the performance degrades somewhat rapidly, which does not happen with PLTSLES.

Although we do not have performance data for the RW-H&R for larger N, we would expect that it would be roughly similar to that exhibited for the problem above. From the experimental data obtained using PLTSLES, as shown in figures 1-8, we would expect it to perform better than the RW-H&R.

4.2. PLTSLES and the PRTS algorithm

The PRTS algorithm is a row-oriented version of the PCTS algorithm proposed by Li and Coleman [5]. The main idea behind the PCTS algorithm is to distribute an outer product (see [3]) – a vector containing the updated x components is passed around the ring, and on the i-th processor, the component $x(i)$ is computed. As on PLTSLES, the processors are interconnected in a ring.

We compared our PLTSLES algorithm with $P = 16$ and $\delta = 10$, to the PRTS for $N = 200, 400, 600, 800$ and 1000. The data for the PRTS were taken from [5]. Figure 10 shows the two curves obtained for each algorithm.

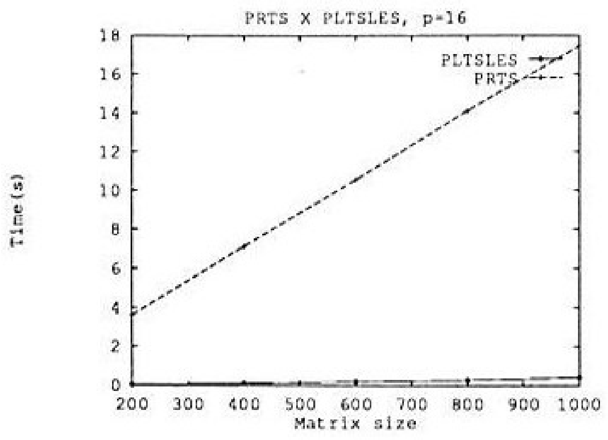


Figure 10: PRTS × PLTSLES, $\delta = 10$, $P = 16$.

Again, as when comparing the PLTSLES to the RW-H&R, we disregard the execution times, due to the different target machines. We note that both curves are linear, however the PRTS has a much larger slope showing that the execution time increases faster with N than the PLTSLES implementation.

4.3. PLTSLES and the OW-DiZitti algorithm

The OW-DiZitti is a wavefront-based algorithm which was mapped on to a 2-D mesh of processors through the use of folding techniques. It was implemented on a 2×2 mesh of T414 transputers.

We compared our PLTSLES algorithm with $P = 4$ and $\delta = 10$, to the OW-DiZitti for system sizes of $N = 50, 100, 150, 200$ and 250. Figure 11 shows the execution times obtained by both algorithms. The data for the OW-DiZitti were taken from [1].

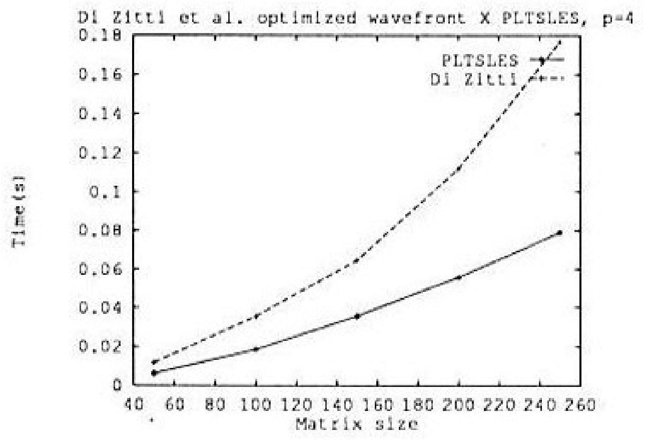


Figure 11: OW-Di Zitti × PLTSLES, $\delta = 10$, $P = 4$.

We note that the curve presented by the OW-DiZitti is exponential-like, while the curve presented by PLTSLES, in this range of system sizes, is almost linear. We would thus expect that for large N, as shown in section 3.3, the PLTSLES algorithm would perform much better than the OW-DiZitti.

5. CONCLUSION

We have presented a parallel algorithm to solve lower triangular systems of linear equations. The parallelism is obtained by dividing the solution in segments and then overlapping the work carried on these. The granularity of the problem is controlled by the size of a segment.

We have shown that the PLTSLES algorithm performs better than other previously published algorithms, especially when the problem size is large. It presents a high level of scalability with respect to the number of processors employed.

The PLTSLES algorithm is being used in parallel implementations of some iterative methods to solve systems of linear equations, currently being developed at the Computing Laboratory, UKC. Preliminary results already taken with these implementations show that we can expect to achieve efficiencies in excess of 80% when solving large linear systems.

6. ACKNOWLEDGEMENTS

We would like to thank Professor Peter H. Welch, from the Computing Laboratory, UKC, for the worthwhile discussions carried on while developing this research.

The first author acknowledges the financial support given by the Brazilian National Council for the Scientific and Technological Development (CNPq) under grant 204062/89.6.

7. REFERENCES

1 E. Di Zitti, G. M. Bisio, F. Curatelli, and G.C. Parodi. High efficiency solution of triangular systems equations on a 2-D array of transputers. *Microprocessing and Microprogramming*, 25:259–264, 1989.

2 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 Journal of Scientific and Statistical Computing*, 9:589–600, 1988.

3 G.H. Golub and C.F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, 1989.

4 M.T. Heath and C.H. Romine. Parallel solution of triangular systems on distributed-memory multiprocessors. Technical report, ORNL/TM-10384, Oak Ridge National Laboratory, March 1987.

5 G. Li and T.F. Coleman. A parallel triangular solver for a distributed-memory multiprocessor. *SIAM Journal of Scientific and Statistical Computing*, 9:485–502, 1988.

6 G. Li and T.F. Coleman. A new method for solving triangular systems on distributed-memory message-passing multiprocessors. *SIAM Journal of Scientific and Statistical Computing*, 10:382–396, 1989.