Parallel Preconditioned Conjugate-Gradients Methods on Transputer Networks*

Rudnei Dias da Cunha
Computing Laboratory, University of Kent at Canterbury, U.K.
Centro de Processamento de Dados, Universidade Federal do Rio Grande do Sul, Brasil

Tim Hopkins
Computing Laboratory, University of Kent at Canterbury, U.K.

Abstract. We show how highly efficient parallel implementations of basic linear algebra routines may be used as building blocks to implement efficient higher level algorithms. We discuss the solution of systems of linear equations using a preconditioned Conjugate-Gradients iterative method on a network of transputers. Results are presented for the solution of both dense and sparse systems; the latter being derived from the finite-difference approximation of partial differential equations.

1 Introduction

The numerical solution of non-singular systems of N linear equations,

$$Ax = b \tag{1}$$

is required in a wide range of practical applications. When N is large, the solution of (1) becomes time-consuming and techniques must be sought to accelerate the solution process. This may be accomplished by

1. using an iterative method with an improved rate of convergence, possibly with the aid of preconditioning,

2. devising a parallel implementation of the method that allows the efficient use of a parallel machine.

In this paper we discuss the solution of (1) using preconditioned Conjugate-Gradients (PCG) method [1]. This method may be defined in terms of a relatively small number of basic linear algebra operations and we obtain our parallel version of the PCG method by using highly efficient parallel implementations of those operations.

A brief overview of the paper is as follows. The programming model used, the basic underlying structure of the parallel processes and the network of transputers are discussed in §2. The parallelisation of the linear algebra operations is given in §4. The formulation of PCG and the results obtained are described in §5. Finally we present our conclusions.

*To appear in *Transputer Communications*

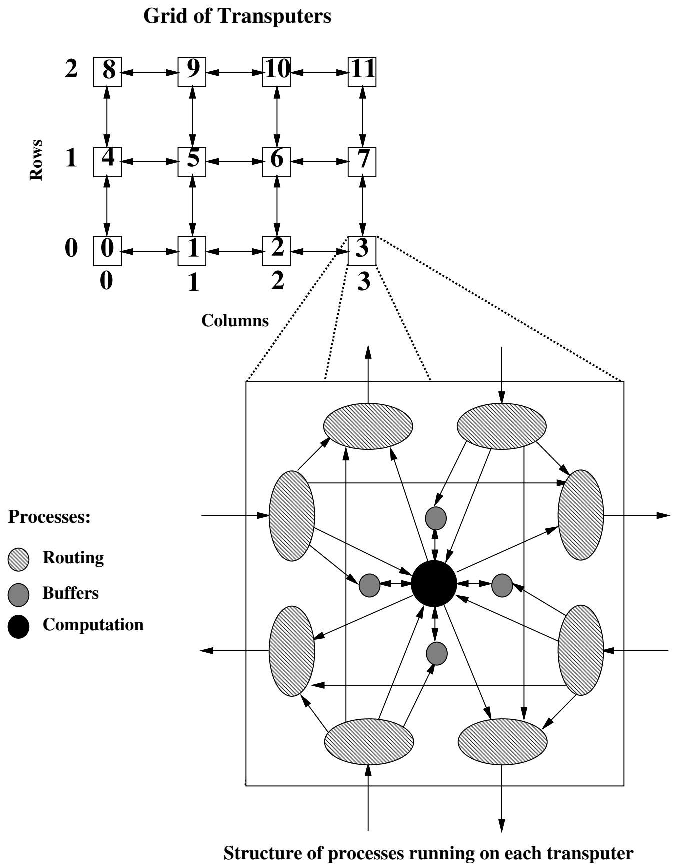

Figure 1: Network of transputers and individual processes structure.

2 Programming model

Our algorithms were implemented using the *Single Program, Multiple Data* model (SPMD). SPMD involves writing a single code that will run on all the processors cooperating on a task. The data are partitioned among the processors which know what portions of the data they will work on.

The iterative methods deal with three data types; scalar values, vectors and matrices. Only the latter two are liable to data partitioning (see §3); scalar values are required to be present in all processors. If a scalar value is derived from some computation over data distributed among the processors, the use of the SPMD model may result in a high communication overhead while this value is sent to all the processors.

For example, such communication is necessary when computing inner-products and vector 2-norms. However, we will see that for some combinations of problem data sizes and the number of processors it is still possible to achieve good performance.

2.1 Structure of processes and processors

The processors are interconnected using a mesh topology; either a square or rectangular grids being used. Each processor is connected to its neighbours using bidirectional communications.

The grid has $P = P_r \times P_c$ processors, where P_r is the number of rows of processors and

P_c is the number of columns of processors.

The computation and communication tasks were grouped separately in a set of processes, running concurrently on each transputer, to allow each processor to run as fast as it can. A computation process is responsible for the initiation and control of the activities of each transputer. Each transputer uses 8 processes to handle the incoming and outgoing messages, two being used to control each link. Two buffers per link are required to store incoming data (vector and scalar values) for the computation process.

2.2 *Routing information on the grid*

A message may be either *broadcast* or *specific*. A broadcast message originates on a processor and is relayed through the network until it reaches all other processors. A specific message is one that is directed to a particular target processor.

Broadcast messages originate from a processor called *central* which is situated in the "middle" of the grid. This processor has coordinates $(\lfloor P_r/2 \rfloor, \lfloor P_c/2 \rfloor)$. Messages are broadcast using the *Row-Column Broadcast* algorithm (RCB), which uses the following strategy. Depending on the relative position of the central processor to the boundaries of the grid, it sends the message down 2, 3 or 4 links. These multiple copies are relayed in different ways depending on the direction (horizontal or vertical) from which it has been received. When a message, travelling in the horizontal direction (i.e., in the same row as the central processor), reaches a processor in this row, it is relayed in the vertical direction producing either 1 or 2 more copies, depending on the position of that processor relative to the grid boundaries; the message is also relayed horizontally until it reaches a processor on the boundary of the grid. Messages travelling in the vertical direction are simply relayed up (or down) until they reach a processor on the boundary of the grid. The number of steps required to complete the RCB algorithm (i.e., until all processors have received the broadcast value) is given by $\lfloor P_r/2 \rfloor + \lfloor P_c/2 \rfloor$.

Figure 2 shows the behaviour of the RCB algorithm for a 5×5 grid. The processors become darker as the messages pass through the network.

A specific message is routed through the processors using the *Find-Row-Find-Column* algorithm (FRFC) detailed in [2]. The message is sent from the *originator* processor vertically until it reaches a processor sitting in the same row as the *target* processor. The message is then moved horizontally across the processors in that row until it reaches the target processor, as shown in Figure 3. If either the row or column of the originator and target processors are the same then the message will travel only in a horizontal or vertical direction respectively.

3 Data partitioning

Data partitioning among the processors plays a major role in the design and efficiency of parallel algorithms. There are two main reasons why we partition the data. The first is the need to solve a problem in less time. If a problem has a data size N and it takes T units of time to be completed on a single processor, then P processors working on subproblems of size N/P will ideally solve it in T/P time units. The second reason is the need to solve a problem with a large amount of data which will not fit into the memory of a single processor. Adding more processors makes more memory available and (hopefully) the solution will be obtained efficiently.

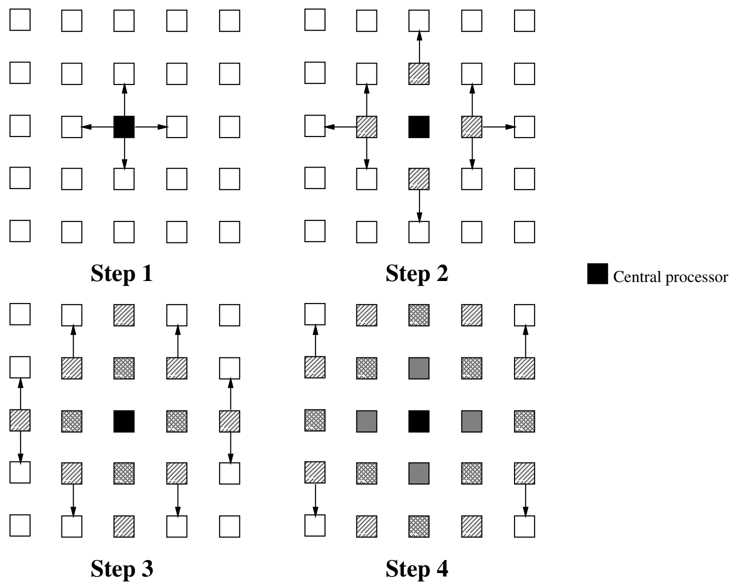

Figure 2: Row-Column Broadcast.

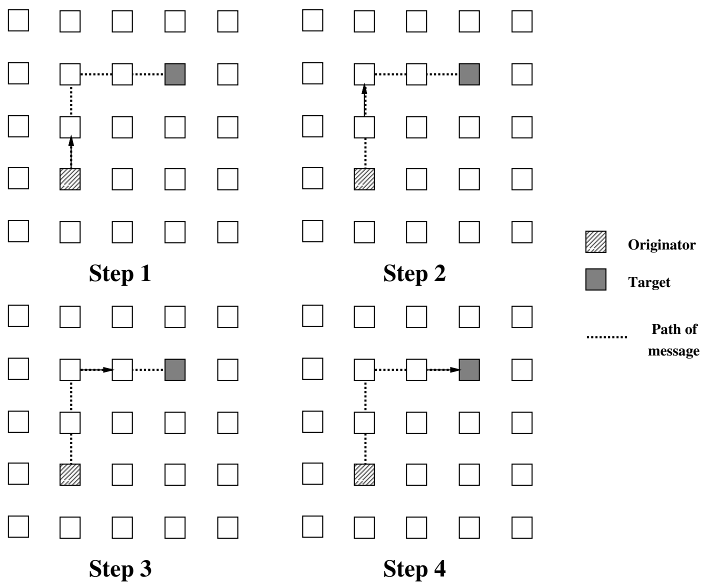

Figure 3: Find-Row-Find-Column Routing.

We use *data partitioning by contiguity*, defined as follows. To partition the data (i.e., vectors and matrices) among the processors, we divide the set of variables $V = \{\, i \,\}_{i=1}^{N}$ into P subsets $\{\, W_p \,\}_{p=1}^{P}$ of $s = N/P$ elements each. We assume without loss of generality that N is an integer multiple of P. We define each subset as $W_p = \{(p-1)s + j\}_{j=1}^{s}$.

Each processor p is responsible for performing the computations over the variables contained in W_p. In the case of vector operations, each processor will hold segments of s variables. The data partitioning for operations involving matrices is discussed in §4.3.

4 Linear algebra operations

4.1 Saxpy

The saxpy $w = u + \alpha v$ operation, where u, v and w are vectors and α is a scalar value, has the characteristic that its computation is *disjoint element-wise* with respect to u, v and w. This means that we can compute a saxpy without any communication between processors; the resulting vector w does not need to be distributed among the processors. Parallelism is exploited in the saxpy by the fact that P processors will compute the same operation with a substantial smaller amount of data. The saxpy is computed as

$$w_i = u_i + \alpha v_i, \quad \forall i \in \{W_p\}_{p=1}^{P} \tag{2}$$

4.2 Inner-product and vector 2-norm

The inner-product $\alpha = u^T v = \sum_{i=1}^{N} u_i v_i$ is an operation that involves accumulation of data, implying a high level of communication between all processors. The mesh topology and the processes architecture used allowed a more efficient use of the processors than, for instance, a ring topology, reducing the time that processors are idle waiting for the computed inner-product value to arrive, but the problem still remains. The use of the SPMD paradigm also implies the global broadcast of the final computed value to all processors.

The inner-product is computed in three distinct phases. Phase 1 is the computation of partial sums of the form

$$\alpha_p = \sum_{\forall i \in \{W_p\}} u_i \times v_i, \quad p = 1, \ldots, P \tag{3}$$

Phase 2 is the accumulation of these partial sums across the grid. This accumulation, which we refer to as *Row-Column Accumulation* (RCA), is implemented using a modification of the RCB routing algorithm. Every processor in the top and bottom rows of the grid sends its partial inner-product value down or up its column to its neighbour. The arriving α value is summed to the α value computed by the receiving processor. This accumulation is repeated as the value travels vertically until it reaches the central column. When the accumulated values have been received and accumulated by the processors in the central row, the processors sitting in the extremes of this row send their accumulated values to their left or right neighbours. The accumulation process is repeated until the accumulated values arrive at the central processor. These accumulated values are then summed together with the central processor's own partial sum thus yielding the required inner-product value. Figure 4 depicts the wavefronts of data for the accumulation phase of the inner-product.

The third phase is the broadcast of the inner-product value to all processors using the RCB algorithm, as required by the SPMD model.

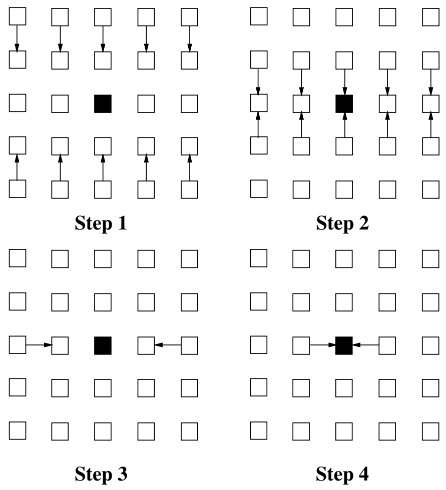

Figure 4: Row-Column Accumulation.

The accumulation phase of the inner-product using the RCA algorithm is completed in the same number of steps as the RCB algorithm (§2.2). An accumulation based on the recursive doubling technique [3, pages 56-61], would require the same number of steps as the RCB requires. This is because of due the need to relay partial values between processors without any accumulation taking place, due to the connectivity of the grid topology.

The vector 2-norm $\alpha = ||\, u \,||_2 = \sqrt{u^T u}$ is computed using the inner-product algorithm described above. Once the inner-product value is received by a processor during the final broadcast phase, it computes the square-root of that value giving the required 2-norm value.

4.3 Matrix-vector product

For the matrix-vector product $v = Au$, we use a *column partitioning* of A. Each processor holds a set W_p (see §3) of s columns each of N elements of A and s elements of u. The s elements of u stored locally have a one-to-one correspondence to the s columns of A (e.g., a processor holding element u_j also holds the j-th column of A). Note that whilst we have A partitioned by columns among the processors, the matrix-vector product is to be computed by *rows*.

The algorithm for computing the matrix-vector product using column partitioning is a generalisation of the inner-product algorithm described in §4.2 (without the need for a final broadcast phase). At a given time during the execution of the algorithm, each one of $P - 1$ processors is computing a vector w of s elements containing partial sums required for the segment of the vector v in the remaining "target" processor. After this computation is complete, each of the P processors stores a vector w. The resulting segment of the matrix-vector product vector which is to be stored in the target processor is obtained by summing together the P vectors w, as described below.

Each processor other than the target processor sends its w vector to one of its neighbouring processors. A processor decides whether to send the vector in either the row or column directions to reach the target processor based on the FRFC algorithm (see §2.2). If a vector passes through further processors in its route to the target processor the w vectors are accumulated. Thus the target processor will receive at most four w vectors which, when summed to its own w vector, yield the desired set of s elements of v. The algorithm may be described as

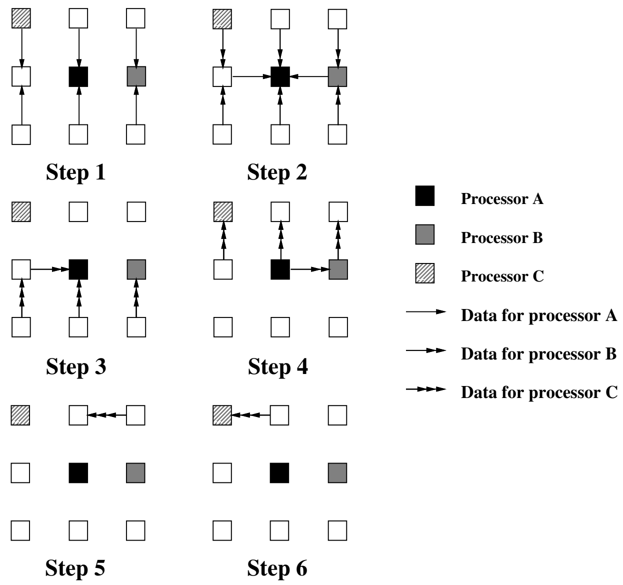

Figure 5: Mechanics of the matrix-vector product using column partitioning.

for $t = 1, \ldots, P$ (t is the target processor)
  for $p = 1, \ldots, P$
    $w_{p,i} = \sum_{\forall j \in W_p} A_{i,j} u_j, \quad \forall i \in W_t$
    if $p \neq t$ then
      sends w_p to t using the FRFC algorithm
  receives k accumulated vectors $w_i, \quad i = 1, \ldots, k, \quad 1 \leq k \leq 4$
  $v_t = w_t + \sum_{i=1}^{k} w_i$

The matrix-vector product using the column partitioning is highly parallel. Since there is no broadcast operation involved, as soon as a processor on the boundary of the grid (either rows or columns) has computed and sent a w_p vector destined to a processor “A”, it can compute and (possibly) send a w_p vector to processor “B”, at which time its neighbouring processors may also have started computing and sending their own w vectors to processor “B”.

As an example consider Figure 5. At a given point in the matrix-vector product computation, the processors are computing w vectors destined to processor A. When these vectors have been accumulated in the row of that processor (step 1), the processors in the top and bottom rows compute and send the w vectors for processor B; while the processors on the left and right columns of the row of processor A send the accumulated r vectors to processor A (step 2). Processor A now stores its set of the resulting v vector (which is the accumulation of the w vectors). In step 3, the processors in the bottom row compute and send the w vectors for processor C while the processor at the left-hand end of the row of processor B sends the accumulated w vectors of that column towards processor B. The next steps are similar to the above. Note that due to the use of buffers (§2.1) it is possible for a processor involved in computing data for one processor to receive data destined for another processor. The data are stored in buffers and used by the computation process as required. Since the exchange of messages in `occam 2` causes the sender and receiver processes to synchronise we use buffers to remove this synchronisation at the numerical computation level, allowing each processor to run at its own speed.

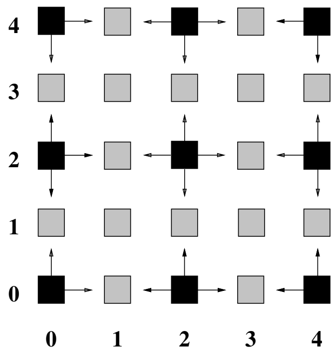

Figure 6: Pattern of communication used in the matrix-vector product.

4.4 Matrix-vector product – finite-difference approximation

When solving (1) where A is derived from a finite-difference approximation of a partial differential equation (PDE), it is important to exploit the sparsity (and regular pattern of the non-zero elements in) the matrix. In our implementation, we exploit the geometry associated with the regular grid of points used to approximate the PDE. A geometric partitioning is used to match the topology and connectivity present in the grid of transputers (§2.1).

The discretisation of the PDE is obtained by specifying a grid size l defining an associated grid of $N = l^2$ interior points (note that this is the order of the linear system to be solved). At each interior point, we associate a set of values, namely the coefficients C, N, S, E and W. Using the five-point approximation to the PDE (see [4]) at each interior point (i, j), the matrix-vector product $v = Au$ is obtained as

$$v_{i,j} = C_{i,j}u_{i,j} + E_{i,j}u_{i,j+1} + W_{i,j}u_{i,j-1} + N_{i,j}u_{i+1,j} + S_{i,j}u_{i-1,j} \, . \tag{4}$$

Note that we do not need to form A explicitly. Examining (4) we note that only neighbouring values are needed to compute $v_{i,j}$. This implies a very low degree of information exchange between the processors which can be effectively exploited with transputers, since the required values of u can be exchanged independently through each link.

The parallel computation of (4) proceeds as follows. We have l rows and columns in the discretisation grid, which we want to partition among a $P_r \times P_c$ mesh of processors. Each processor will then carry out the computations associated with a block of $\lfloor l/P_r \rfloor + sign\,(l \bmod P_r)$ rows and $\lfloor l/P_c \rfloor + sign\,(l \bmod P_c)$ columns of the interior points of the grid.

The actual computation starts with each processor sending the top and bottom rows and the left and right column values of the vector u to its neighbours, depending on the position of the processors in the grid. Processors in the corners of the mesh interact with only two neighbours, those that lie on the boundary of the mesh (but not in the corners) interact with three, and the remaining processors all exchange data with four neighbours, as shown in Figure 6.

As soon as the computation process has sent the boundary rows and columns of the vector u (which is still being transferred to another processor by the appropriate routing process), it can start the computations on the locally stored data, which does not require any values stored in other processors. When these computations are finished, the processor can start computing the values associated with the boundary of the data block once it has received the required data from its neighbours. If the block of local data is large enough, we would expect that, when computation on this block has finished, the communications-handling processes will have already received the data needed for the boundary computations.

The algorithm above can be extended to use other finite-difference approximations, e.g., the nine-point approximation, at the expense of an increase in the information exchange between processors.

4.5 Combined use of the linear algebra operations

In the previous sections we have presented the behaviour of the saxpy, inner-product and matrix-vector product operations when executed separately. When the linear algebra operations are called in sequence, a level of *overlapping between operations* is evident; some processors, having completed their participation in a particular operation, immediately start computing the next operation in the sequence. Each processor is thus allowed to execute at its own speed. This overlap of operations is due to

- messages travelling different distances depending on the positions of the origin and target processors on the grid,
- use of buffers to remove synchronism at the numerical computation level.

For the saxpy, if the processors are synchronised at the beginning of the computation, they will finish it still synchronised, assuming that all processors have the same number of elements assigned to them. The inner-product and matrix-vector product, however, involve different degrees of communication between the processors. It can be noted from Figures 2, 4 and 5 that during the execution of these operations some processors finish their task before others.

As an example consider the execution of an inner-product, a matrix-vector product, a saxpy and a 2-norm, in this order, on a 2×2 grid of processors; this sequence of operations is similar to those required in many iterative methods, including the Conjugate-Gradients method (described in §5).

Figure 7 shows the steps involved in this sequence. The processors are assumed to be synchronised before the computation of the inner-product begins. During steps 1, 2 and 3 all processors are involved in the inner-product computation. As the processors receive the inner-product value they start computing the matrix-vector product (steps 4 to 10); in step 10 two processors are computing the saxpy and two are computing the matrix-vector product.

In step 11 the processors that have computed the saxpy begin to compute the 2-norm; the partial sums are sent to the processors in the top row of the grid and are stored in buffers, so that those processors may continue to compute the saxpy. Note the difference between the computation of the 2-norm as shown in steps 11 to 14 and that of the inner-product, essentially because the processors are not synchronised when the 2-norm computation is initiated.

If we count the number of steps required to compute each operation individually, and assumed no overlap of operations, then this sequence would require 15 steps to be completed, instead of 14. The gains in the number of steps increase when larger grids are used and when more operations are chained together.

4.6 Efficiency results

We consider efficiency as the ratio between the execution times of sequential and parallel implementations of an algorithm. The sequential implementation is executed on a single

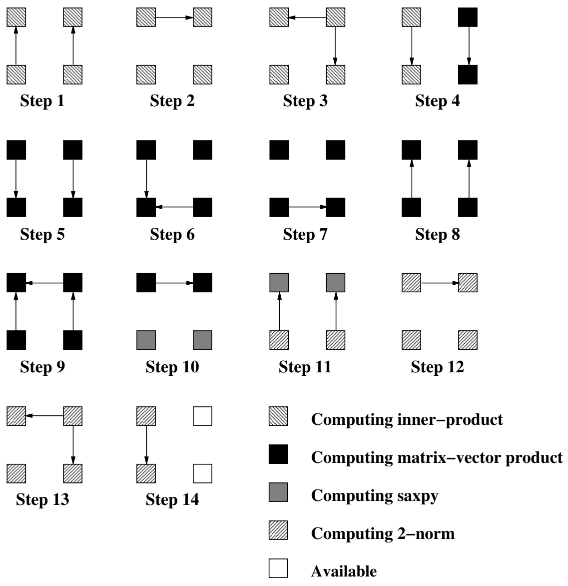

Figure 7: Example of combined use of linear algebra operations: Generic case.

transputer.

Figures 8 and 9 show the efficiencies obtained for the saxpy, inner-product, and matrix-vector products (dense and finite-difference approximation versions) implementations discussed above. Double-precision floating-point arithmetic is used throughout.

Note that the saxpy offers efficiencies close to 100% for any vector sizes. For the inner-product, increasing the vector size provides higher efficiencies even on a large grid of transputers. The matrix-vector product for both versions offer efficiencies as high as 90% for large enough matrices.

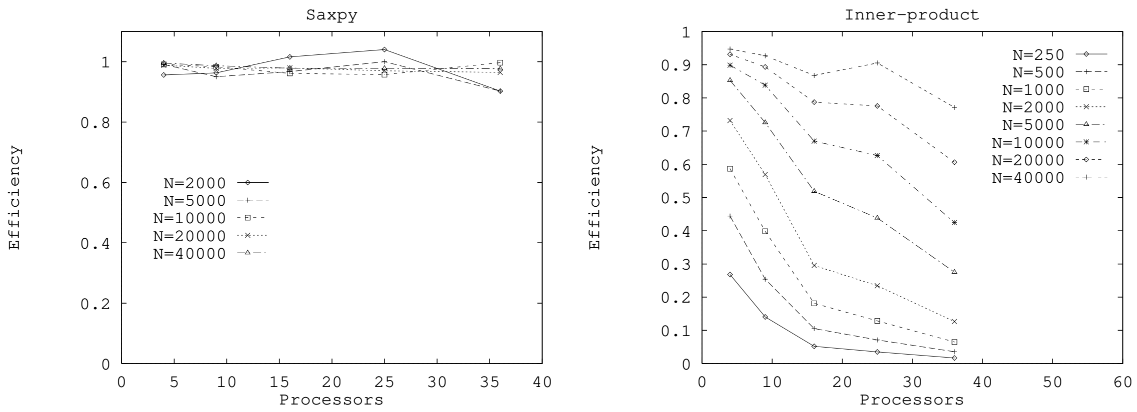

Figure 8: Efficiencies of saxpy and inner-product operations.

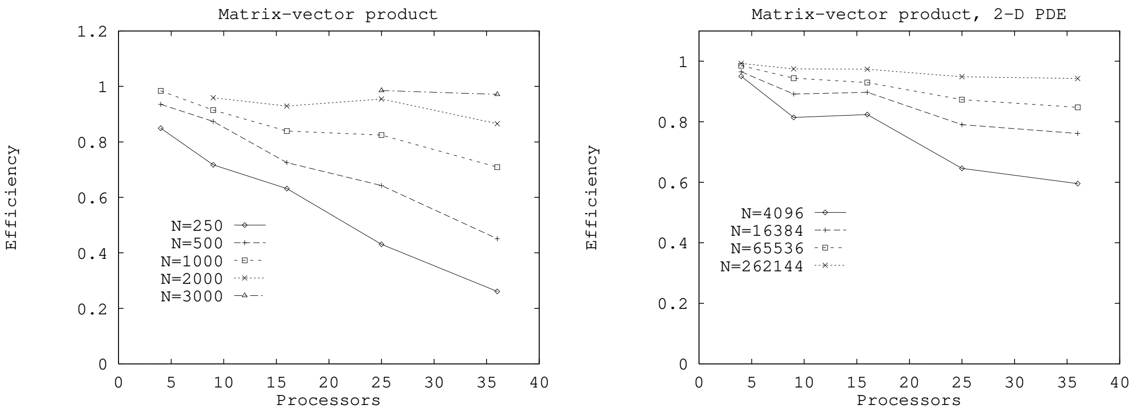

Figure 9: Efficiencies of matrix-vector product operations.

The code for each operation makes extensive use of loop-unrolling techniques which substantially reduces the execution time of code on a transputer (see [5], [6] and [7] for details). The peaks observed in Figures 8 and 9 are caused by the different number of unrolled loops used, depending on the values of N and P, and the number of unrolling levels used (16).

5 Preconditioned Conjugate-Gradients

We now consider a preconditioned version of the Conjugate-Gradients method ([1]). Left-preconditioning is assumed, i.e., the system actually solved is $M^{-1}Ax = M^{-1}b$ where M is a preconditioning matrix that approximates A^{-1}. The main steps of the **PCG(M)** algorithm are outlined as follows

$$
\begin{array}{ll}
\textbf{PCG}(M)\text{:} & Mr^{(0)} = b - Ax^{(0)} \\
& p^{(0)} = r^{(0)} \\
& \rho^{(0)} = {r^{(0)}}^T r^{(0)} \\
& \text{for } k = 1, 2, \ldots \\
& \quad Mw^{(k-1)} = Ap^{(k-1)} \\
& \quad \alpha^{(k-1)} = \rho^{(k-1)} / {p^{(k-1)}}^T w^{(k-1)} \\
& \quad x^{(k)} = x^{(k-1)} + \alpha^{(k-1)} p^{(k-1)} \\
& \quad r^{(k)} = r^{(k-1)} - \alpha^{(k-1)} w^{(k-1)} \\
& \quad \rho^{(k)} = {r^{(k)}}^T r^{(k)} \\
& \quad \beta^{(k)} = \rho^{(k)} / \rho^{(k-1)} \\
& \quad p^{(k)} = r^{(k)} + \beta^{(k)} p^{(k-1)}
\end{array}
$$

The preconditioning used in our implementations is the polynomial preconditioning (see [8], [9], [10] and [11]), which can be implemented very efficiently in a parallel architecture since it is expressed as a sequence of saxpys and matrix-vector products (§4).

In Table 1 we show the execution time (in seconds) per iteration for two implementations of **PCG**(M), using the routines to compute the linear algebra operations discussed in the previous sections. Note that due to the amount of memory available in the transputers we used (4Mbytes) some problem sizes cannot be executed for small grids of processors. In this case, the sequential time presented is an approximation to the expected execution time.

5.1 Efficiency results

Figure 10 shows the efficiencies obtained by an implementation of PCG(M) on a grid of transputers, using the routines to compute the linear algebra operations discussed in the previous sections.

Note that as N increases, larger grids of transputers can be used with good efficiencies. It is also important to note that the overall efficiency attained by the implementations is not determined by the linear algebra operation with the lowest efficiency for an specific value of N; in fact the efficiency of PCG is very close to that of the higher-efficiency operations, the matrix-vector product and the saxpy.

Dense case						
	P					
N	1	4	9	16	25	36
600	1.13	0.32	0.16	0.12	0.10	0.09
1200	4.53	1.24	0.59	0.38	0.26	0.22
1800	10.19	–	1.29	0.76	0.54	0.41
2400	18.12	–	–	1.32	0.88	0.67

Finite-difference case						
	P					
N	1	4	9	16	25	36
4096	0.28	0.07	0.04	0.02	0.02	0.01
16384	1.04	0.27	0.14	0.08	0.06	0.04
65536	4.03	1.04	0.48	0.27	0.19	0.14
262144	16.23	–	–	1.03	0.69	0.48

Table 1: Time(s)/iteration of **PCG**(M).

6 Conclusion

We have shown that an iterative method like the preconditioned Conjugate-Gradients may be successfully parallelised by using highly efficient parallel implementations of the linear algebra operations involved. We have used the same approach to parallelise other iterative methods with similar degrees of efficiency (see [6] and [12]).

Acknowledgements

We would like to thank the referees for their comments which helped improve this paper.

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

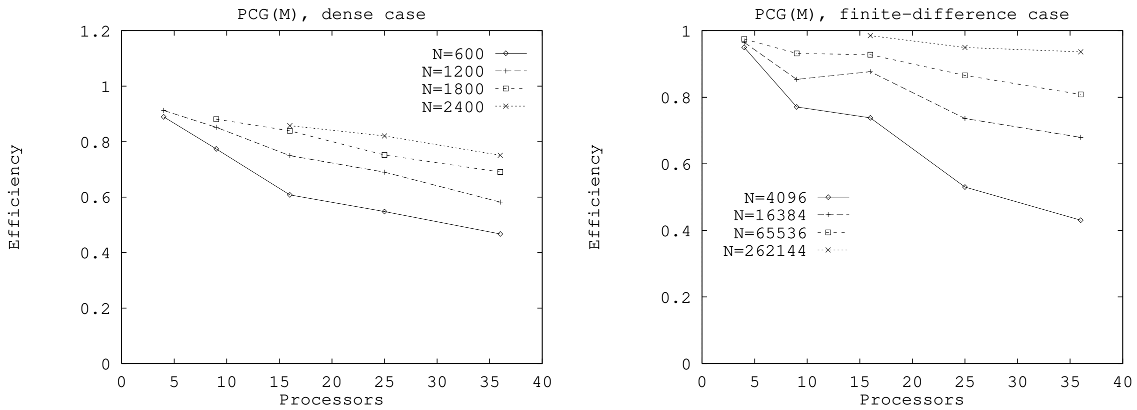

Figure 10: Efficiencies of PCG(M).

References

[1] G.H. Golub and C.F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, 2nd edition, 1989.

[2] U. de Carlini and U. Villano. *Transputers and parallel architectures – message-passing distributed systems*. Ellis Horwood, Chichester, 1991.

[3] J.J. Modi. *Parallel Algorithms and Matrix Computation*. Oxford University Press, Oxford, 1988.

[4] G.D. Smith. *Numerical Solution of Partial Differential Equations: Finite Difference Methods*. Oxford University Press, Oxford, 3rd edition, 1985.

[5] C.F. Schofield. *Optimising FORTRAN programs*. Ellis Horwood, Chichester, 1989.

[6] R.D. da Cunha and T.R. Hopkins. *The Parallel Solution of Systems of Linear Equations using Iterative Methods on Transputer Networks*, pages 1–13. Transputing for Numerical and Neural Network Applications. IOS Press, Amsterdam, 1992. Also as Internal Report No. 16-92, Computing Laboratory, University of Kent at Canterbury, U.K.

[7] P. Atkin. Performance maximisation. INMOS Technical Note 17.

[8] Y. Saad. Practical use of polynomial preconditionings for the Conjugate Gradient method. *SIAM Journal of Scientific and Statistical Computing*, 6:865–881, 1985.

[9] S.C. Eisenstat. Efficient implementation of a class of preconditioned Conjugate Gradient methods. *SIAM Journal of Scientific and Statistical Computing*, 2:1–4, 1981.

[10] L. Adams. m-Step preconditioned Conjugate Gradient methods. *SIAM Journal of Scientific and Statistical Computing*, 6:452–463, 1985.

[11] O.G. Johnson, C.A. Micchelli, and G. Paul. Polynomial preconditioners for Conjugate Gradient calculations. *SIAM Journal of Numerical Analysis*, 20:362–376, 1983.

[12] R.D. da Cunha and T.R. Hopkins. *The Parallel Solution of Partial Differential Equations on Transputer Networks*, pages 96–109. Transputing for Numerical and Neural Network Applications. IOS Press, Amsterdam, 1992. Also as Internal Report No. 17-92, Computing Laboratory, University of Kent at Canterbury, U.K.