The Parallel Solution of Partial Differential Equations 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 present an implementation of a finite-difference approximation for the solution of partial differential equations on transputer networks. The grid structure associated with the finite-difference approximation is exploited by using geometric partitioning of the data among the processors. This provides a very low degree of communication between the processors.

The resultant system of linear equations is then solved by a variety of Conjugate Gradient methods. Care has been taken to ensure that the basic linear algebra operations are implemented as efficiently as possible for the particular geometric partitioning used.

1 Introduction

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

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

derived from a finite-difference approximation to a partial differential equation (PDE), using parallel implementations of various Conjugate Gradient iterative methods.

Systems like (1) are characterized by being large, structured and sparse. Because of the very low density of non-zero elements any efficient parallel implementation must exploit the structure inherently present in the coefficient matrix. The use of a general-purpose iterative solver in a parallel environment to solve such problems will not be efficient, as noted by the authors in [2].

The solution of these systems using iterative methods involves the repeated application of a relatively small number of linear algebra operations, namely matrix-vector products, saxpys (see [5]), inner-products and vector (or matrix) norms. The efficiency of a parallel implementation of these methods is therefore strongly dependent upon the efficiency of these operations.

When solving (1) by an iterative technique, it is to be expected that the number of iterations required to achieve a solution within a prescribed tolerance will be very small compared to the order of the system. Preconditioned Conjugate-Gradient methods usually provide the solution in relatively few iterations.

* To appear in “Transputing for Numerical and Neural Network Applications”, IOS Press, Amsterdam.

A brief outline of the paper follows. In §2, we give a description of the transputer-based machine and the structure of processes used in the implementations. In §3 we present the linear algebra subroutines developed. Conjugate-Gradient methods, including the standard Conjugate-Gradient (CG) [5], the Conjugate Gradient Squared (CGS) [13] and the Bi-CGSTAB [14] are described in section 4 along with some aspects of the use of polynomial preconditioners. Finally, in section 5 the results obtained from solving a variety of partial differential equations are presented.

2 Parallel environment

The algorithms were implemented on a MEiKO SPARC-based Computing Surface using T800 transputers with 4 MB of external memory. Double-precision arithmetic was used throughout.

The processors were interconnected using a mesh topology; either square or rectangular meshes can be used. Each processor was connected to its neighbours using bidirectional communications.

Since the processors did not have equal workloads (see §3), we separated the computation and communication tasks into a set of processes, running concurrently in each transputer, to allow each processor to run as fast as possible. Each transputer has 8 processes to handle the incoming and outgoing messages, two being used to control each link. Two buffers are provided on each processor to store incoming data for the computation process.

3 Linear Algebra Subroutines

We present in this section the algorithms developed to perform the basic linear algebra operations required. In [2], the authors have presented algorithms to compute some linear algebra operations in the context of an iterative solver for a general system of linear equations. The saxpy and the inner-product algorithms presented in [2] can be used for this particular application, with the difference that the vectors are structured in blocks of rows and columns to match the discretization grid used in the finite-difference approximation. The linear algebra operation which takes most advantage of the inherent parallelism of the finite-difference approximations is the matrix-vector product.

In the description that follows, A is a matrix, u, v and w are vectors and α is a scalar. We used discretization grids with sizes l = 64, 128, 256 and 512 internal points, leading to matrices and vectors of order $N = l^2 = 4096, 16384, 65536$ and 262144. The parallel implementations used square meshes of 4, 9, 16, 25 and 36 processors.

3.1 Matrix-vector product $v = Au$

The matrix-vector product Au can be implemented by computing inner-products between its rows and the vector u. In our case, since A is highly sparse, this does not provide an efficient parallel implementation (see [2]). However, since A is formed using the five-point finite-difference approximation of a PDE (see Figure 1), we can exploit the structure of the approximation and, by using geometric distribution of data, achieve an efficient implementation.

The discretization of the PDE is obtained by specifying a grid size l and the associated grid has $N = l^2$ interior points (note that this is the order of the linear system to be solved).

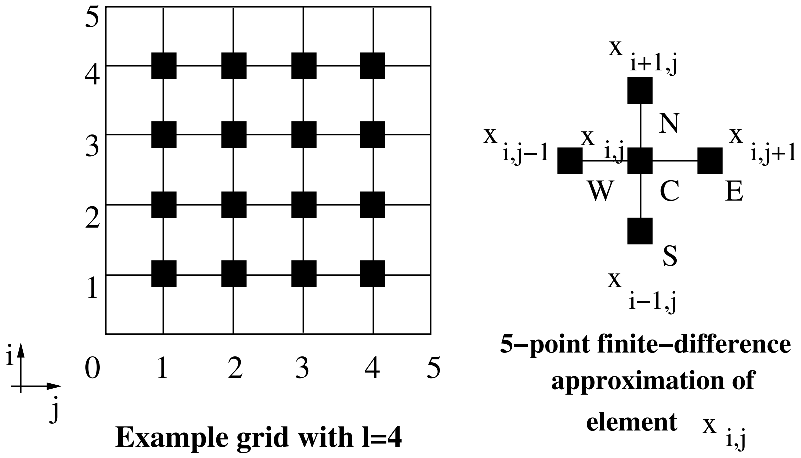

Figure 1: Five-point finite-difference approximation to a PDE.

At each interior point, we associate a set of values, namely the coefficients C, N, S, E, W. Using the five-point approximation to the PDE (see [12]) at each interior point (i, j), the matrix-vector product $v = Au$ is computed by

$$v_{i,j} = C_{i,j}u_{i,j} + E_{i,j}u_{i,j+1} + W_{i,j}u_{i,j-1} + N_{i,j}u_{i+1,j} + S_{i,j}u_{i-1,j}\,. \tag{2}$$

Note that we do not need to form A explicitly.

Examining (2) 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. Figure 2 shows the geometric partitioning of a 4×4 grid onto a 2×2 mesh of transputers. This type of partitioning allows the use of very large grids, since all the data is distributed among the processors.

The parallel computation of (2) proceeds as follows. We have l rows and columns in the discretization grid, which we want to partition among a $p \times q$ mesh of processors. Each processor will then carry out the computations associated with a block of $\lfloor l/p \rfloor + sign\,(l \bmod p)$ rows and $\lfloor l/q \rfloor + sign\,(l \bmod q)$ columns of the interior points of the grid.

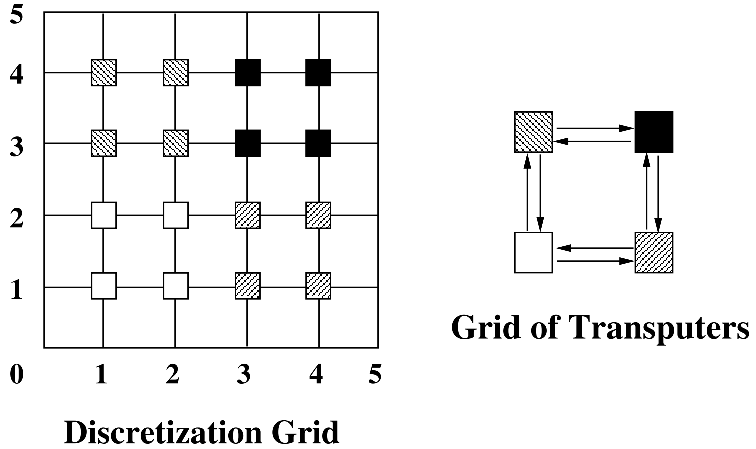

Figure 2: Geometric partitioning of data.

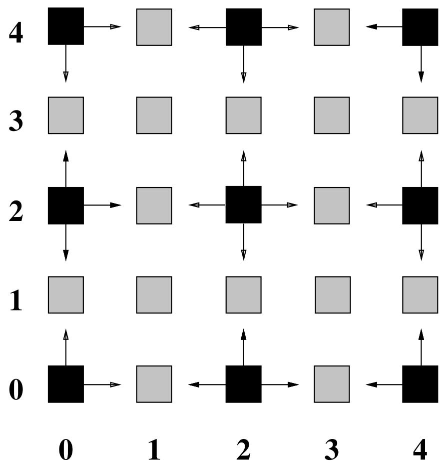

Figure 3: Pattern of communication used in the matrix-vector product.

The actual computation is started by all processors 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 only with 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 3.

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 the processors.

3.2 Loop-unrolling

Loop-unrolling is an effective tool used to reduce the overheads associated with the execution of the loop and load phase of array elements during arithmetic instructions (see [11], [9]).

The matrix-vector product discussed here makes extensive use of arrays. We would thus expect a substantial reduction in the execution time of the operation when using loop-unrolling. Experimental results have shown a reduction of more than 50%.

3.3 Efficiency of the implementations

Figures 4 and 5 show the efficiencies obtained for the matrix-vector product, the saxpy and the inner-product operations.

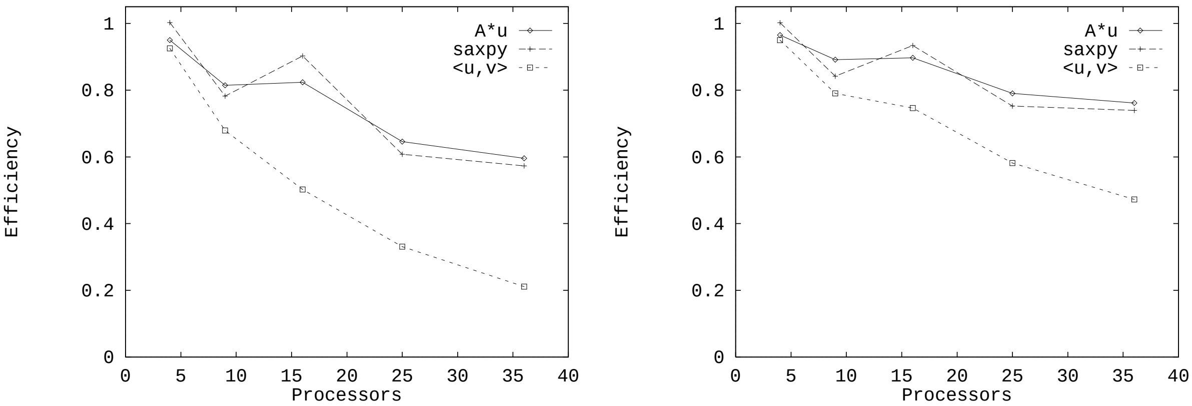

Figure 4: Linear Algebra Subroutines, grid sizes (a) 64×64, (b) 128×128.

When using a 3×3 mesh of transputers, with $l = 64$, 128, 256 and 512, a steep reduction in performance appears. This is because the operations have their loops unrolled by a factor of 16 (see [2]) and, in the 3×3 case, the size of the blocks allocated to each processor is not a multiple of 16 and hence some data are processed using standard loops. Note that for the 4×4 mesh all the data will be accessed using unrolled loops with an accompanying increase in efficiency.

As expected, the inner-product provides the lowest efficiency level of the three operations considered. The saxpy and matrix-vector product are highly efficient and for all combinations of approximation grids and processor mesh sizes more than 60% of efficiency is achieved.

The flattening out of the efficiency plots close to one as the problem size increases is important; it means that the overheads of distributing the calculations are almost negligible for large approximation grids and, for such grids, larger processor meshes will still be effective.

It is interesting to compare the results given in this section for the Au operation with those obtained with a general-purpose sparse Au, presented by the authors in [2]. Since the number of non-zeros per row is a constant for the five-point approximation, the sparsity of the resulting coefficient matrix increases with the grid size.

For the general sparse implementations, we found that there is a minimal sparsity below which efficiency decreases rapidly. Unfortunately, the sparsity of the systems resulting from

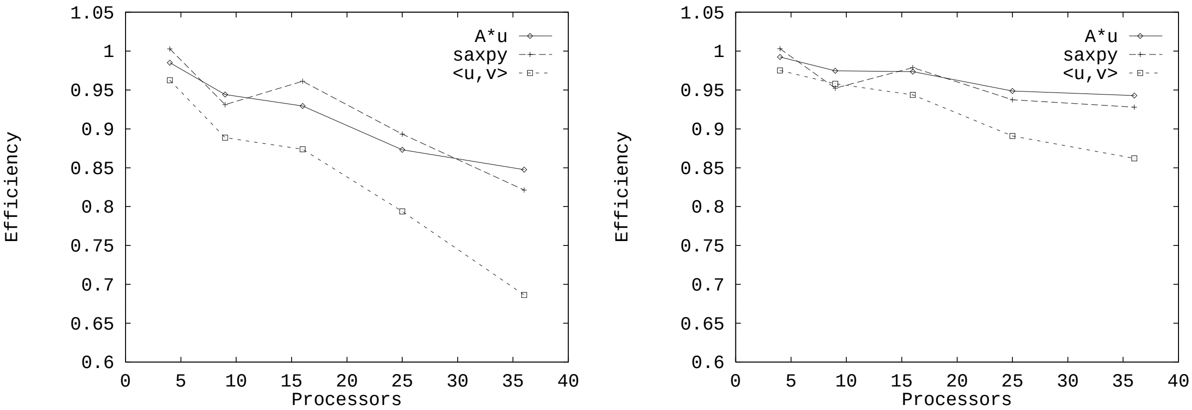

Figure 5: Linear Algebra Subroutines, grid sizes (a) 256×256, (b) 512×512.

the five-point approximation is below this minimum. Such parallel implementations are thus not suited to solving these systems. However, the implementation described here has the opposite behaviour. The efficiencies obtained from our experiments show that as we increase the grid size we increase the efficiencies, thus confirming our idea that specific rather than general purpose code is required for efficiency in this case.

4 Conjugate Gradient methods

Conjugate Gradient methods are widely used in practical applications to solve both symmetric and nonsymmetric systems of linear equations. Such methods tend to be robust (at least for some variants of the method) and convergence requires relatively few iterations compared to other iterative methods. However, as noted in [8], no single variant of CG has been found that will solve all linear systems.

It is common practice to use CG methods together with a preconditioner to improve the rate of convergence. In this paper, we will consider the use of polynomial preconditioners (see [3], [7], [10], [1]) applied to the standard CG method, **PPCG(**m**)**, as described in [2], to the Conjugate Gradient Squared (CGS) and to the Bi-CGSTAB methods. The preconditioning matrix used is described in [2, §5.1].

4.1 CGS

The CGS method, proposed by Sonneveld [13], is a modification of the Bi-CG [4] for the solution of some nonsymmetric systems.

Its main advantage over the Bi-CG is that it is twice as fast in some cases (in terms of the number of iterations required for convergence) and avoids the matrix-vector product involving A^T. The drawback is a rather chaotic convergence behaviour, sometimes leading to an incorrect solution. For a discussion of the convergence properties of CGS and other CG variants, we refer the reader to the paper of Nachtigal et al. [8].

Sonneveld presents a preconditioned version of CGS using an incomplete LU factorization. We introduce a polynomial preconditioned CGS, **PPCGS(**m**)**, described as

PPCGS(m**)**: $r^{(0)} = b - Ax^{(0)}$, $\tilde{r}^{(0)}$ is an arbitrary vector

$q^{(0)} = p^{(-1)} = 0$, $\rho^{(-1)} = 1$

while $||\, r^{(k)}\, ||_2 / ||\, b\, ||_2 \geq \epsilon$

$$
\begin{aligned}
&M_m z^{(k)} = r^{(k)} \\
&\rho^{(k)} = \tilde{r}^{(0)^T} z^{(k)} \\
&\beta^{(k)} = \rho^{(k)} / \rho^{(k-1)} \\
&u^{(k)} = z^{(k)} + \beta^{(k)} q^{(k)} \\
&p^{(k)} = u^{(k)} + \beta^{(k)} (q^{(k)} + \beta^{(k)} p^{(k-1)}) \\
&M_m z^{(k)} = A p^{(k)} \\
&\alpha^{(k)} = \rho^{(k)} / \tilde{r}^{(0)^T} z^{(k)} \\
&q^{(k+1)} = u^{(k)} - \alpha^{(k)} z^{(k)} \\
&r^{(k+1)} = r^{(k)} - \alpha^{(k)} A (u^{(k)} + q^{(k+1)}) \\
&x^{(k+1)} = x^{(k)} + \alpha^{(k)} (u^{(k)} + q^{(k+1)})
\end{aligned}
$$

Compared with PPCG(m), PPCGS(m) requires one matrix-vector multiplication, one solution of the preconditioning system and five saxpys more per iteration. As we have seen in §3 all these operations may be efficiently implemented in parallel and we would not expect PPCGS(m) to be much less efficient than PPCG(m).

4.2 Bi-CGSTAB

The Bi-CGSTAB method, presented by van der Vorst [14], is another variant of the Bi-CG method which provides smoother convergence behaviour than CGS.

It has been shown by numerical experiments ([14], [15]) that Bi-CGSTAB is usually more efficient than CGS, not only with regard to the number of iterations required for convergence, but also in the quality of the resulting solution.

We now present a polynomial preconditioned version, **PPBi-CGSTAB(m)**, given as

PPBi-CGSTAB(m): $r^{(0)} = b - Ax^{(0)}$, $\tilde{r}^{(0)}$ is an arbitrary vector

$$
\begin{array}{l}
v^{(0)} = p^{(0)}\,,\ \rho^{(0)} = \alpha = \omega^{(0)} = 1 \\
\text{while } \|\, r^{(k)} \,\|_2 / \|\, b \,\|_2 \geq \epsilon \\
\quad \rho^{(k)} = \tilde{r}^{(0)^T} r^{(k-1)} \\
\quad \beta = (\rho^{(k)}/\rho^{(k-1)})(\alpha/\omega^{(k-1)}) \\
\quad p^{(k)} = r^{(k-1)} + \beta(p^{(k-1)} - \omega^{(k-1)} v^{(k)}) \\
\quad M_m y = p^{(k)} \\
\quad v^{(k)} = Ay \\
\quad \alpha = \rho^{(k)}/\tilde{r}^{(0)^T} v^{(k)} \\
\quad s = r^{(k-1)} - \alpha v^{(k)} \\
\quad M_m z = s \\
\quad t = Az \\
\quad M_m \tau = t \\
\quad \omega^{(k)} = t^T z / t^T \tau \\
\quad x^{(k)} = x^{(k-1)} + \alpha y + \omega^{(k)} z \\
\quad r^{(k)} = s - w^{(k)} t
\end{array}
$$

PPBi-CGSTAB(m) requires one solution of the preconditioning system, one saxpy and two inner-products more per iteration compared to PPCGS(m). Due to the presence of more inner-products per iteration, PPBi-CGSTAB(m) will outperform PPCGS(m) only if the solution of the system is obtained in less iterations, a situation that arises for some problems (see [14]). We should note that van der Vorst points out that the use of more inner-products per iteration in vector machines does not cause problems since they are easily vectorizable and provide good performance (see, for instance, [6]). For distributed-memory architectures, however, inner-products are more costly and Bi-CGSTAB will need to provide a solution in substantially less iterations than CGS to be faster.

5 Results

We compare the performances of parallel implementations of PPCG(m), PPCGS(m) and PPBi-CGSTAB(m) in solving test problems obtained by approximating three partial differential equations using the simple five-point finite-difference approximation.

Problem 1 is Laplace's equation subject to Dirichlet boundary conditions,

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 1$$

in the unit square with $u = 1$ on the boundary. This problem was solved using the grid sizes used in §3.

Problem 2 is a linear 2-dimensional partial differential equation,

$$\frac{\partial u}{\partial t} + \sigma_1 \frac{\partial u}{\partial x} + \sigma_2 \frac{\partial u}{\partial y} = 1$$

with $0 \le x, y \le 1$, $0 < t < 100$, $\sigma_1 = 4$, $\sigma_2 = 8$. The spacing along the t-axis was chosen as 100, and the grid sizes used were as in problem 1.

Problem 3 is the partial differential equation (taken from [14])

$$-A\frac{\partial^2 u}{\partial x^2} - A\frac{\partial^2 u}{\partial y^2} + B(x,y)\frac{\partial u}{\partial x} = F$$

where $0 \le x, y \le 1$, $B(x,y) = 2e^{2(x^2+y^2)}$, with Dirichlet conditions on the boundary, $u = 1$ for $y = 0, x = 0, x = 1$ and $u = 0$ for $y = 1$. The values of A are defined as in Figure 6, and $F = 0$ in the domain, except for the subsquare in the centre, where $F = 100$. This problem was solved using a 127×127 grid.

5.1 Efficiency of the implementations

The problems were solved using $x^{(0)} = 0$. For PPCGS(m) and PPBi-CGSTAB(m), the arbitrary vector $\tilde{r}^{(0)}$ was chosen randomly. The systems were symmetrically scaled by the diagonal, $D^{-1/2}AD^{-1/2}$.

Problem 1 was solved using PPCG(m) with $m = 1$ and $\gamma_{1,0} = \gamma_{1,1} = 1$. The tolerance used was $\epsilon = 10^{-10}$.

Table 1 shows the number of iterations, k, required for convergence by the unpreconditioned CG and PPCG(m), the execution time per iteration of the sequential implementations, running on a single transputer and the gain obtained by using PPCG(m), defined as $(1 - (T_{PPCG(m)}/T_{CG})) \times 100$. Since the reduction in the number of iterations is substantial, PPCG(m) is faster than CG, even at the expense of more operations per iteration.

Figure 7 shows the efficiencies obtained by CG and PPCG(m) for problem 1, for the 64×64 and 256×256 grids. Note that there is little to choose as far as the efficiencies obtained by distributing the computation are concerned.

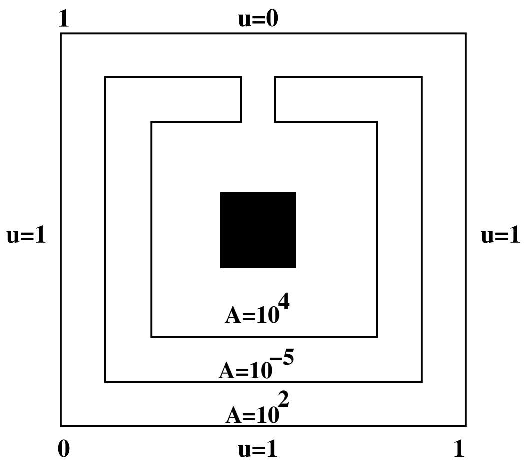

Figure 6: Coefficient values for problem 3.

Grid	CG		PPCG(m)		Gain
	k	Time(s)/k	k	Time(s)/k	
64×64	132	0.17	77	0.28	3.92
128×128	266	0.65	145	1.05	11.94
256×256	533	2.51	272	4.04	17.86
512×512	1076	10.16	536	16.28	20.18

Table 1: CG and PPCG(m) results for problem 1 on a single transputer.

Problem 2 was solved using PPCG(m), PPCGS(m) and PPBi-CGSTAB(m). We used $m = 1$, $\gamma_{1,0} = \gamma_{1,1} = 1$ and $\epsilon = 10^{-10}$. Figure 8-(a) presents the efficiency of PPCG(m), PPCGS(m) and PPBi-CGSTAB(m) for problem 2, using the 256×256 grid. Note that PPBi-CGSTAB(m), is slightly less efficient than PPCG(m), due to the use of more inner-products at each iteration. Table 2 shows that the number of iterations required and the time per iteration obtained by each of the above methods. For this problem PPCGS(m) provides the best performance, even when PPBi-CGSTAB(m) requires less iterations to converge, since the amount of work per iteration is less.

Problem 3 was solved using PPCGS(m) and PPBi-CGSTAB(m), with $m = 1$, $\gamma_{1,0} = \gamma_{1,1} = 1$ and $\epsilon = 10^{-5}$. In this problem, PPBi-CGSTAB(m) is substantially better than PPCGS(m), solving the system in 65 iterations, while PPCGS(m) requires 89. Although requiring a larger amount of work per iteration, PPBi-CGSTAB(m) provides the solution faster than PPCGS(m), due to the smaller number of iterations. Figure 8-(b) shows the efficiencies of both implementations. Note again the slightly less efficient behaviour of PPBi-CGSTAB(m).

6 The three-dimensional case

The distributed algorithm to compute the matrix-vector product using five-point finite-difference can easily be extended to the use of a seven-point finite-difference scheme to solve a PDE in a three-dimensional region. The idea here is to consider the three-dimensional grid as a set of two-dimensional grids as shown in Figure 9.

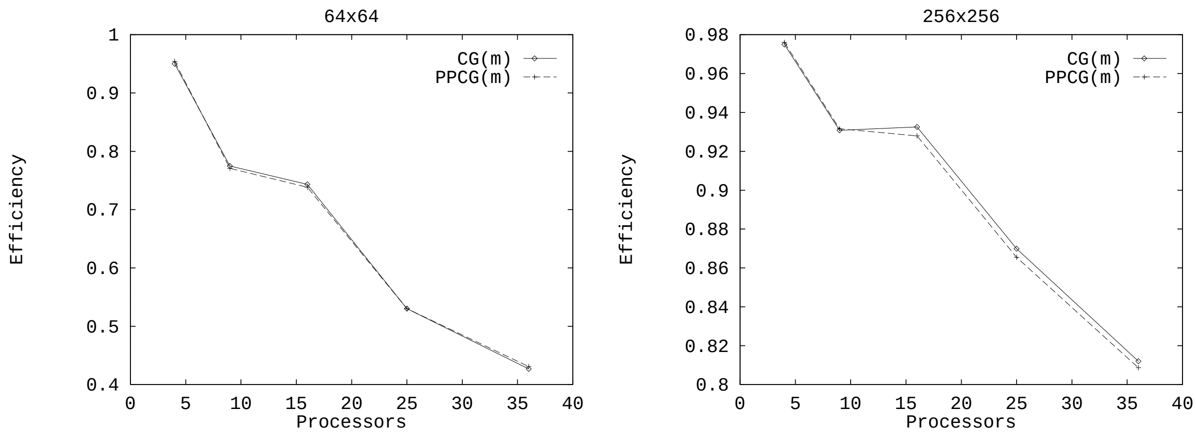

Figure 7: Efficiencies of (a) CG and (b) PPCG(m), for problem 1.

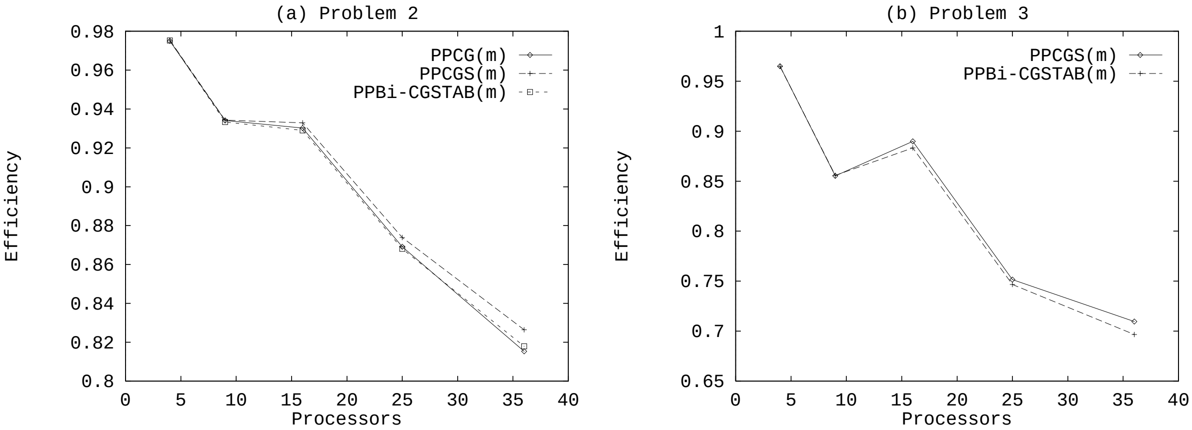

Figure 8: Efficiencies for problems 2 and 3.

Ideally, we would like to have a "transputer" with six links, so that a network of such processors could match the geometry and connectivity of the PDE discretization grid. However, we can use a two-dimensional grid of transputers and partition the data as vertical blocks among the processors, as shown in Figure 10.

The algorithms for the three basic linear algebra operations presented in §3 may be extended to the three-dimensional case. The saxpy is computed using the algorithm for the two-dimensional case, for every plane of the block of discretization points stored in a processor. For the inner-product, a partial inner-product value is computed locally within a processor for the whole block of data. The partial values computed by each processor are then accumulated according to the algorithm in [2]. This modification of the algorithm substantially reduces the amount of communication since the accumulation phase is done only once.

The matrix-vector product using the seven-point finite-difference is expressed as

$$\begin{aligned} v_{i,j,k} &= C_{i,j,k}u_{i,j,k} + E_{i,j,k}u_{i,j+1,k} + W_{i,j,k}u_{i,j-1,k} + \\ & \quad N_{i,j,k}u_{i+1,j,k} + S_{i,j,k}u_{i-1,j,k} + U_{i,j,k}u_{i,j,k+1} + D_{i,j,k}u_{i,j,k-1} \end{aligned} \tag{3}$$

The matrix-vector product is computed using the algorithm in two-dimensions for every $i \times j$ plane in the k axis. The elements of the vector v are then updated with the coefficients U and D according to (3). This update involves no communication between the processors due to the partitioning imposed. Figure 11 shows the efficiencies for the matrix-vector product for grid sizes $64 \times 64 \times 64$ and $80 \times 80 \times 80$. Note that the effects of loop-unrolling are

	PPCG(m)		PPCGS(m)		PPBi-CGSTAB(m)	
Grid	k	Time(s)/k	k	Time(s)/k	k	Time(s)/k
64×64	130	0.29	88	0.53	76	0.64
128×128	139	1.09	84	2.02	83	2.39
256×256	137	4.22	80	7.82	83	9.13
512×512	135	17.09	77	31.63	77	37.32

Table 2: PPCG(m), PPCGS(m) and PPBi-CGSTAB(m) results for problem 2.

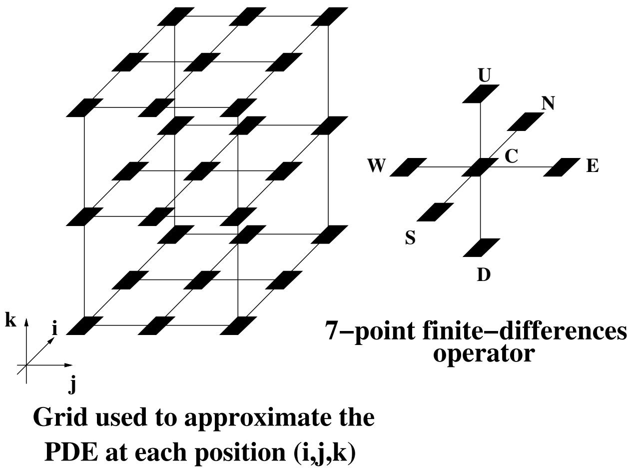

Figure 9: Seven-point finite-difference approximation to a PDE.

noticeable particularly for the grid size $80 \times 80 \times 80$, where the efficiency increases when P is increased from 16 to 25 processors, since in the latter no standard loop is needed, for the same number of unrolled loops in both cases. Also, as the number of planes increase, there is a reduction in the efficiency, since more communication between the processors is needed.

As an example of the use of these operations, Figures 12-a and 12-b show the timings and efficiencies obtained by PPCG(1) when solving Laplace's equation in three-dimensions. The discretization grid sizes used were $64 \times 64 \times 8$, $64 \times 64 \times 16$, $64 \times 64 \times 32$ and $64 \times 64 \times 64$. The corresponding systems have order $N = 32768$, 65536, 131072 and 262144. These systems were solved using symmetric scaling and a required tolerance of 10^{-10}. A solution was achieved after 50, 68, 86 and 96 iterations respectively. Some of the systems can not be solved with small processor meshes due to memory constraints. The efficiencies obtained are lower than those obtained using PPCG(1) for solving systems of similar size in the two-dimensional case. This is due mainly to the increase in the level of communication between the processors when computing the matrix-vector products.

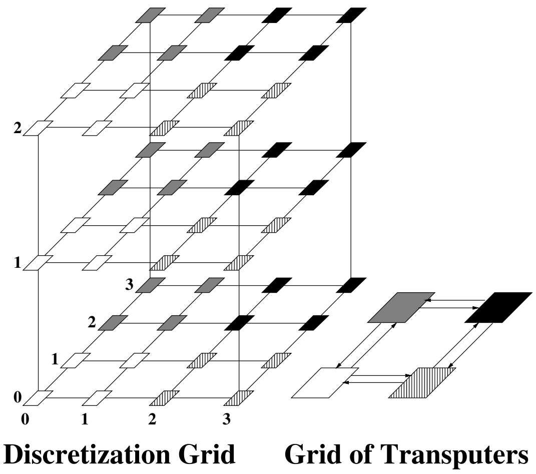

Figure 10: Geometric partitioning of data.

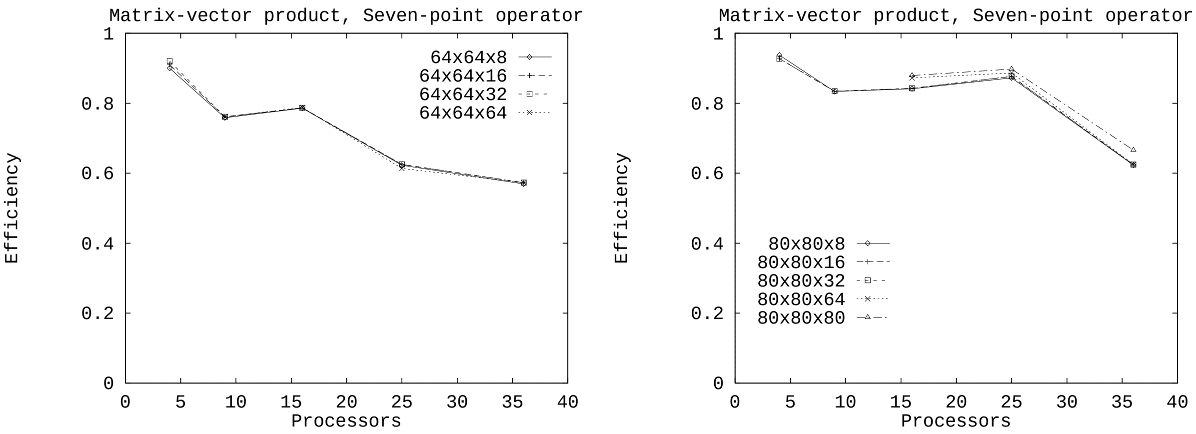

Figure 11: Matrix-vector product, grid sizes (a) $64 \times 64 \times 64$, (b) $80 \times 80 \times 80$.

7 Conclusion

We have presented some parallel partial differential equations solvers, based on five-point and seven-point finite-difference approximations, using some variants of the Conjugate Gradient family of iterative methods.

Parallelism is exploited by a geometric partitioning of the grid used to approximate the PDE and assigning the resulting blocks of data to each processor of the transputer network. The pattern of computation associated with the five-point approximation can be effectively exploited using a mesh of transputers, due to their ability to communicate independently through each link. A similar approach has been shown to be effective for a seven-point operator in the three-dimensional case.

Efficient linear algebra operations can be implemented by breaking down the computation into a number of simple subproblems and using an adequate architecture of processes. Since implementations of Conjugate Gradient methods make extensive use of such operations the successful use of these methods in a distributed-memory parallel architecture depends heavily on the efficiency of these basic underlying linear algebra operations.

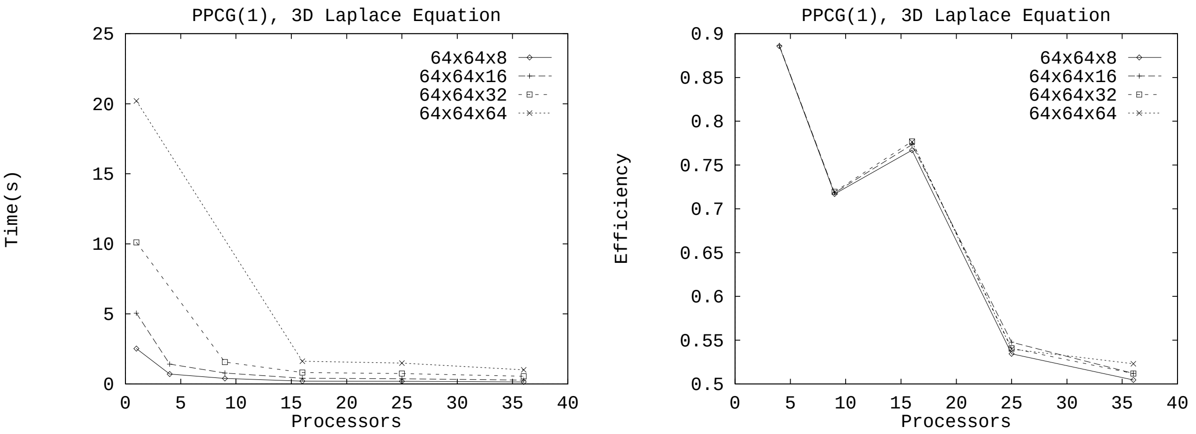

Figure 12: Timings and efficiency of PPCG(1) for Laplace's equation in three-dimensions.

8 Acknowledgements

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.

References

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

[2] R.D. da Cunha and T.R. Hopkins. The parallel solution of systems of linear equations using iterative methods on transputer networks. Report No. 16/92, Computing Laboratory, University of Kent at Canterbury, June 1992. Also to appear in "Transputing for Numerical and Neural Network Applications", IOS Press, Amsterdam.

[3] P.F. Dubois, A. Greenbaum, and G.H Rodrigue. Approximating the inverse of a matrix for use in iterative algorithms on vector processors. *Computing*, 22:257–268, 1979.

[4] R. Fletcher. *Conjugate Gradient Methods for Indefinite Systems*, volume 506 of *Lecture Notes in Mathematics*, pages 73–89. Springer-Verlag, Heidelberg, 1976.

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

[6] W.J. Harrod. *Parallel programming with the BLAS*, pages 253–276. The Characteristics of Parallel Algorithms. MIT Press, Cambridge, Massachusetts, 1987.

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

[8] N.M. Nachtigal, S.C. Reddy, and L.N. Trefethen. How fast are nonsymmetric matrix iterations? Numerical analysis report, 90-2, Department of Mathematics, Massachusetts Institute of Technology, March 1990.

[9] H.W. Roebbers and P.H. Welch. Advanced occam 2 and transputer engineering. Course notes (parts 1 and 2), Control Laboratory, University of Twente and Computing Laboratory, University of Kent at Canterbury, March 1991.

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

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

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

[13] P. Sonneveld. CGS, a fast Lanczos-type solver for nonsymmetric linear systems. *SIAM Journal of Scientific and Statistical Computing*, 10:36–52, 1989.

[14] H.A. van der Vorst. Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM Journal of Scientific and Statistical Computing*, 13:631–644, 1992. Also as Internal report, Mathematical Institute, University of Utrecht.

[15] H. Watanabe and S. Doi. A comparison of Bi-CGSTAB and CGS methods for convection-diffusion equations. In T. Nodera, editor, *Parallel Processing for Matrix Computation*, number 7 in Advances in Numerical Methods for Large Sparse Sets of Linear Systems, Keio, 1991. Keio University.