# The Parallel Solution of Systems of Linear Equations using Iterative 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 present a study of the implementational aspects of iterative methods to solve systems of linear equations on a transputer network. Both dense and sparse systems are considered.

First we discuss the implementation of a set of distributed linear algebra subroutines which are used as building blocks for implementing the iterative methods. We show that the use of loop-unrolling significantly increases the efficiency of these implementations. The effect of the sparsity of the matrices on the performance is analysed.

Finally, serial and parallel implementations of a polynomial preconditioned Conjugate Gradient method are presented.

## 1 Introduction

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

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

Iterative methods for solving system (1) are attractive because their complexity is $O(N^2)$, provided the number of iterations required for convergence is small compared to $N$, whilst direct methods like the LU and Cholesky decompositions are $O(N^3)$, thus precluding their use for very large $N$. Moreover, when considering parallel implementations of such methods, iterative methods are more obviously suited to parallelization through the vectorization of arithmetic operations among replicated functional units (SIMD) or processors (MIMD).

Iterative methods can be built upon a set of basic linear algebra subroutines (BLAS), and the successful parallelization of a given method depends strongly on how efficient these subroutines are. We consider some of the BLAS which are most useful in the implementation of iterative methods, these include inner-products, computation of norms and matrix-vector products.

A brief outline of the paper follows. First, we will present the parallel environment (hardware and software components) used. Then the basic linear algebra subroutines are presented

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

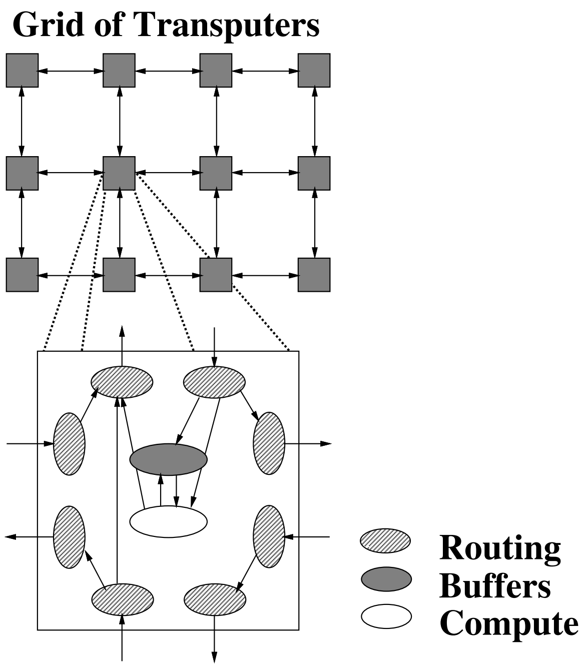



Figure 1: Grid of transputers and structure of processes.

along with the results of some tests designed to ascertain how the efficiency of the implementation of these methods is affected by problem size, sparsity and the number of processors used. Finally, a case study, the implementation of a polynomial preconditioned Conjugate Gradient method, is discussed.

## 2 Parallel environment

### *2.1 Hardware*

The machine used for the implementations was a MEiKO SPARC-based Computing Surface. The processors used were T800 transputers rated at 20 MHz (T800-20). Each transputer has 4MB of DRAM memory.

### *2.2 Software*

The algorithms were implemented in `occam 2` using 64-bit floating-point arithmetic. The topology used to interconnect the transputers is either a rectangular or square grid.

We used the **S**ingle **P**rogram, **M**ultiple **D**ata paradigm in the implementation of the algorithms. The code that ran in each transputer was divided in ten processes. Two processes handled the incoming and outgoing messages in each link and a buffer process was used to decouple communication and computation, the latter being carried out by a separate process. Figure 1 shows the grid of processors and the structure of the processes in each transputer. Some of the connections are not shown for clarity.

We used this structure of processes to overlap communication with computation. We exploited the fact that each link is capable of exchanging data between two processors by assigning two sets of incoming/outgoing- messages handlers to the four links needed for the grid topology. Since these eight processes run at a higher priority and concurrently with the computation process, we can overlap the communication with the computation if the amount of computation is sufficiently large. The data received by the handlers are only passed to the computation process when it is needed.

The communication and routing aspects were encapsulated in the communications-handling processes, hiding these issues from the computation process. This structure allowed the coding of a very clean computation process and reduced the amount of work involved while implementing the algorithms, because the same code (both communications and computation processes) specific to a given method ran in all processors. Also, the reusability of the communications-handling code was enhanced, requiring at most a few minor modifications when implementing other iterative methods.

## 3 Linear Algebra Subroutines

The linear algebra operations that will be discussed are the saxpy, the inner-product and the matrix-vector product.

Consider a real (scalar) value $\alpha$, three real vectors $u$, $v$ and $w$ of length $N$ and real matrix $A$ of size $N \times N$. The grid of processors has $P_r$ rows and $P_c$ columns, the number of processors being $P = P_r \times P_c$. Assume, without loss of generality, that $N$ is an integer multiple of $P$. Each processor stores $s = N/P$ columns of $A$ and segments of $s$ elements of each vector used. The operations are described below.

### *3.1 Saxpy*

The "scalar-alpha-x-plus-y" or saxpy operation is a vector accumulation of the form $w = u + \alpha v$. This operation has the characteristic that its computation is *disjoint element-wise* with respect to the vectors $u$, $v$ and $w$. This means that we can compute a saxpy without any communication between 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 independently in each processor as

$$w_i = u_i + \alpha v_i, \quad i = 1, \ldots, s. \tag{2}$$

Note that the indices used are relative to the $s$ variables stored in each processor and do not refer to the actual variables.

### *3.2 Inner-product*

The inner-product, defined as $\alpha = \sum_{i=1}^{N} u_i v_i$ is an operation that involves accumulation of data, implying a high level of communication between all processors. The grid topology and the process architecture used reduces the time that processors will be idle waiting for the computed inner-product value arrive, but the problem still remains. The use of the SPMD paradigm also implies the global broadcast of that value to all processors.

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

$$\alpha_p = \sum_{i=1}^{s} u_i v_i, \quad p = 1, \ldots, P \tag{3}$$

Phase 2 is the accumulation of the $\alpha_p$ values computed in all processors. This accumulation is performed following the pattern shown in Figures 2-a and 2-b. We define a processor

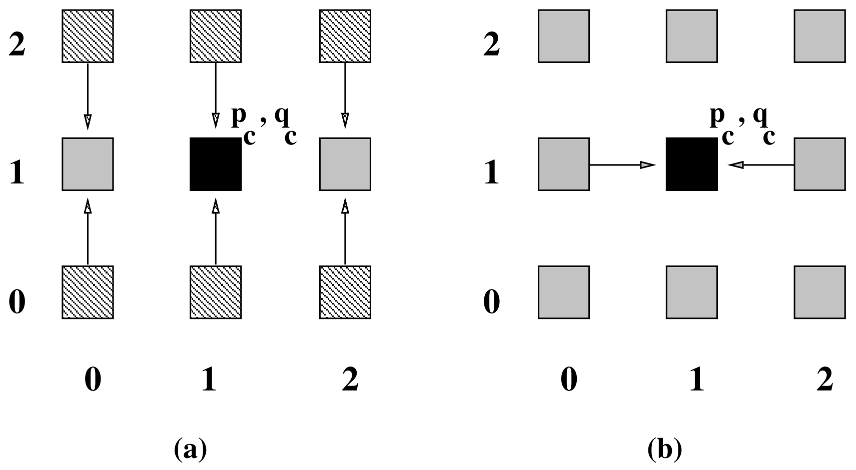



Figure 2: Accumulation phase of the inner-product.

$(p_c, q_c)$ as the processor which will accumulate the partial values $\alpha_p$. This processor sits in the middle of the grid. The accumulation starts with all processors sending their partial values in a north or south direction to the processors that lie in the middle row (Figure 2-a). The partial values are accumulated while passing through each processor. The processors in the middle row then send its column-accumulated partial values to processor $(p_c, q_c)$ as shown in Figure 2-b. At the end of this phase, the inner-product value is present in that processor.

Phase 3 is the broadcast phase where processor $(p_c, q_c)$ sends the inner-product value to its neighbours, which then relay this value to its neighbours, until all processors have the inner-product value stored.

### *3.3 Matrix-vector product*

The matrix-vector product $v = Au$ is computed using a modification of the inner-product algorithm. The main difference is that every processor acts as the accumulating processor once during the execution of the operation, and that no broadcast phase is needed.

The matrix is partitioned by columns among the processors. To save memory, we use the row formulation of the matrix-vector product,

$$v_i = \sum_{j=1}^{N} A_{i,j} u_j \quad i = 1, \ldots, N$$

The computation starts as follows. Each processor computes a vector $w$ of $s$ elements which holds the partial sums of each column with the corresponding element of $u$, for $s$ rows. After this computation is complete, there are $P$ vectors $w$ stored in each processor and $P - 1$ of them are to be sent to a "target" processor. The target processor is the one that holds the current $s$ rows used to compute the $w$'s.

Each processor $p$ sends its $w_p$ 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 *First-Row-First-Column* routing algorithm (see [2]). As it passes through a processor $q$ it is summed to $w_q$, $w_q = w_q + w_p$, and this accumulated vector is again routed to the target processor, possibly being accumulated with other $w$ vectors. 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 $v$. Figure 3 depicts a possible situation for this algorithm.

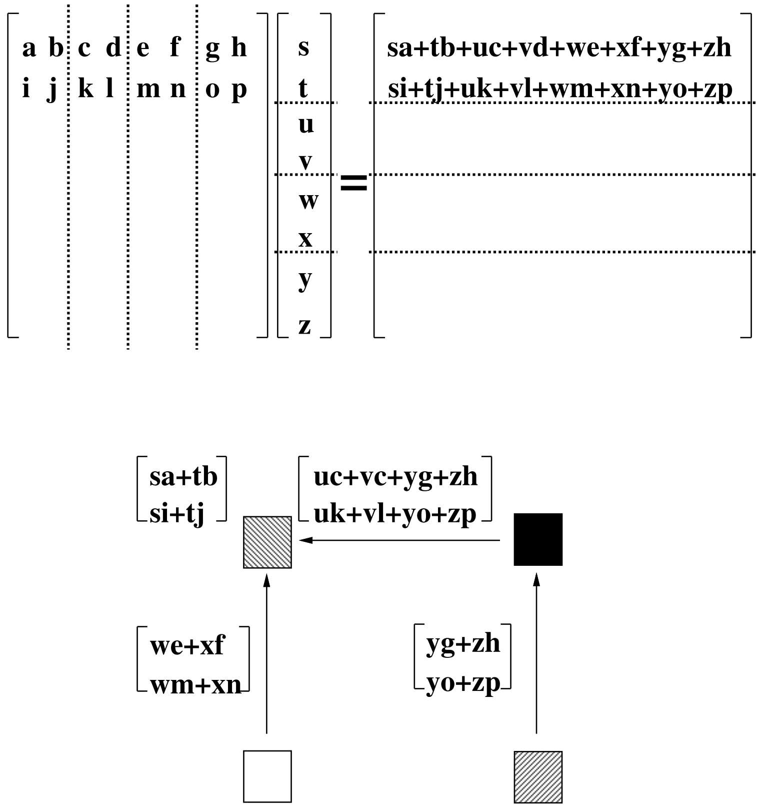



Figure 3: Matrix-vector product using column partitioning.

The column partitioning allows a better distribution of work since the accumulations of data are computed as the vectors $w$ travel through the network. It also allows us to make use of the buffered communications shown in §1, since as soon as a $w$ vector is sent, the processor may start the computations for a different target processor.

### *3.4 Sparse storage*

Large sparse matrices occur frequently in practical applications (see [4]). Special techniques are required if such matrices are to be stored efficiently; and it is especially important that the storage scheme used provides a balance between the ease of accessing the elements and storage requirements.

Many storage techniques are available (see [5]). Some are designed to be efficient with factorization methods (LU and others), some assume a pattern structure (e.g. symmetric or tridiagonal matrices); other techniques are more general.

Our aim was to develop a set of routines which can be used to solve problems in different application areas; this requirement implies the use of a generic sparse storage technique. A modification of the Yale Sparse Matrix Package A-IA-JA format was developed to use with column partitioning.

The storage format uses two integer arrays and a real array to store the pointers to the starting positions of each column, row indices and proper values of the non-zero elements of $A$ respectively. This format allows the representation of any general sparse matrix without taking into account any special pattern that might be present.

### 3.5 *The effect of loop-unrolling*

Loop-unrolling is a technique used to improve the efficiency of loops. The number of times a loop is executed and, hence, the loop overheads are reduced by increasing the amount of useful code in the body of the loop (see [12]).

Loop-unrolling allows a more effective use of the processor through the more efficient scheduling of instructions by the compiler. Some machines with multiple functional units or pipelines, perform better when loop-unrolling is used.

In the case of the transputer, loop-unrolling is used to reduce the overhead of the operand load phase of an arithmetic operation when the operands are vector elements. The number of unrolling levels is thus governed by the number of machine instructions needed to efficiently load the elements of the vectors involved in an arithmetic operation.

According to [10] the efficiency of loop-unrolling on the transputer is maximized at fifteen levels because accessing the vectors in segments of up to sixteen elements allows the most efficient code for the load of operands (in terms of computing the operands' addresses) to be generated by the compiler. Access to vector segments is greatly simplified in `occam 2` by the use of abbreviations (see [8]).

As an example of how loop-unrolling can be used, we consider the matrix-vector product. The code excerpts below show the routines using both standard and unrolled loops

```
{{{  standard loop
SEQ i=0 FOR N
  VAL AA IS A[i]:
  SEQ
    z:=0.0(REAL64)
    SEQ j=0 FOR N
      z:=z+(AA[j]*u[j])
    v[i]:=z
}}}
```

```
{{{  unrolled loop
SEQ i=0 FOR N
  VAL AA IS A[i]:
  SEQ
    k:=0
    z:=0.0(REAL64)
    SEQ j=0 FOR M
      SEQ
        VAL AAA IS [AA FROM k FOR 16]:
        VAL uu IS [u FROM k FOR 16]:
        SEQ
          z:=z+(AAA[0]*uu[0])
          z:=z+(AAA[1]*uu[1])
          .         .
          .         .
          .         .
          z:=z+(AAA[14]*uu[14])
          z:=z+(AAA[15]*uu[15])
        k:=k+16

    VAL AAA IS [AA FROM k FOR VR]:
    VAL uu IS [u FROM k FOR VR]:
    SEQ j=0 FOR VR
      z:=z+(AAA[j]*uu[j])
    v[i]:=z
}}}
```

Table 1 shows the effect of computing the matrix-vector product for standard and unrolled loops for $N = 250$ and $500$, using a serial implementation on a single transputer. The gain is defined as $(1 - T_{unrolled}/T_{standard}) \times 100$. The MFLOPS rate is also shown here where MFLOPS is given by $(2N^2 - N) * 10^{-6}/T$.

The use of loop-unrolling reduces the execution time by approximately $45\%$ whilst achieving a MFLOPS rate of $70\%$ of the theoretical maximum attainable value (approximately 0.7

| N | Standard Time(s) | Standard MFLOPS | Unrolled Time(s) | Unrolled MFLOPS | Gain(%) |
|---|---|---|---|---|---|
| 250 | 0.3657 | 0.3412 | 0.2529 | 0.4932 | 44.5344 |
| 500 | 1.4582 | 0.3426 | 0.9864 | 0.5075 | 47.8233 |

Table 1: Comparisons of standard and unrolled loops.

MFLOPS) by the T800-20 using 64-bit floating-point arithmetic. Similar results were obtained for the other operations presented here.

## 4 Experimental results of the BLAS

In this section, we present the results obtained for the BLAS described in §3.

We tested the routines for different problem sizes on different numbers of processors. Results are given for a serial implementation of the routines, running on a single transputer, for comparison with the parallel implementation.

The vectors are filled with random values, using a normal distribution with mean $10$ and standard deviation $5$, so as to not bias the measurements towards the FPU of the transputer (which does not execute some arithmetic operations if one of the operands is $0$ or $1$). The same is true for the scalar value $\alpha$.

The code for the operations was generated using the "REDUCED" mode of the compiler, which uses the minimum amount of exception testing (only floating-point checks are made).

Table 2 show the timings obtained with $N = 1000$ and $10000$ for $P = 1$ (serial), $4, 9, 16, 25$ and $36$ processors. The matrix-vector product is not computable on any of the processor grids for $N = 10000$ due to memory restrictions.

### *4.1 Efficiencies*

Figure 4-a shows the effects of loop-unrolling for relatively small values of $N$. The efficiencies increase or decrease, for some combinations of $N$ and $P$, due to the fact that relatively

| N=1000 Operation | Serial Time(s) | $P=4$ Times(s) | $P=9$ Time(s) | $P=16$ Time(s) | $P=25$ Time(s) | $P=36$ Time(s) |
|---|---|---|---|---|---|---|
| $u+\alpha v$ | $0.00390$ | $0.00103$ | $0.00046$ | $0.00026$ | $0.00017$ | $0.00012$ |
| $u^T v$ | $0.00416$ | $0.00154$ | $0.00109$ | $0.00141$ | $0.001216$ | $0.001664$ |
| $Au$ | $--$ | $0.99834$ | $0.47750$ | $0.29280$ | $0.19002$ | $0.13317$ |
| **N=10000 Operation** | **Serial Time(s)** | **$P=4$ Times(s)** | **$P=9$ Time(s)** | **$P=16$ Time(s)** | **$P=25$ Time(s)** | **$P=36$ Time(s)** |
| $u+\alpha v$ | $0.03763$ | $0.00954$ | $0.00424$ | $0.00241$ | $0.00157$ | $0.00110$ |
| $u^T v$ | $0.04038$ | $0.00979$ | $0.00474$ | $0.00339$ | $0.00250$ | $0.00256$ |

Table 2: Timings of the BLAS, $N = 1000$ and $N = 10000$.

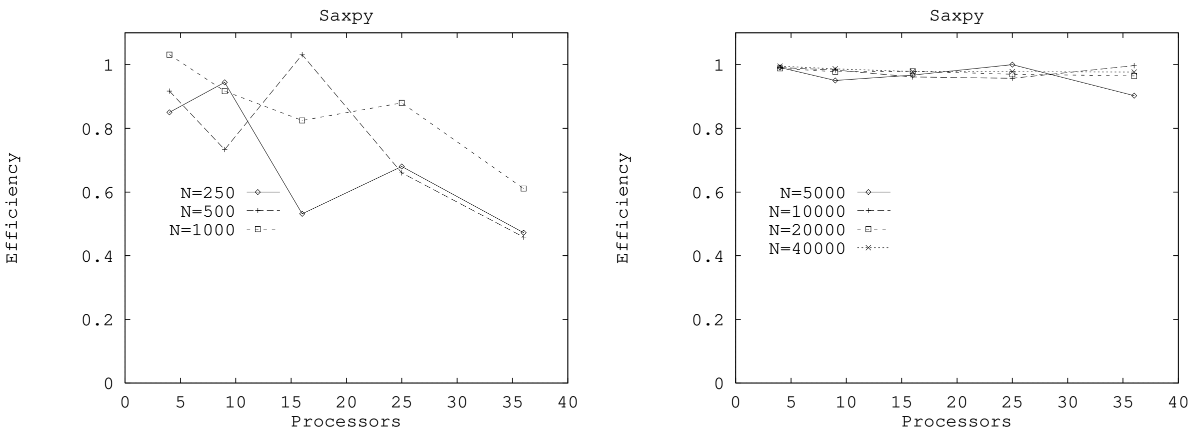



Figure 4: Efficiencies for saxpy, (a) small $N$, (b) large $N$.

few unrolled loops are executed allowing the standard loop to dominate the computation. Figure 4-b shows that for large $N$ the efficiencies are close to $100\%$.

Figure 5-a shows that the inner-product implementation provides acceptable efficiencies as $N$ increases for $P = 36$. The matrix-vector product presents higher efficiencies than those of the inner-product for given $N$ and $P$ values. As will be seen in §6 the combined use of the operations provide a good efficiency, since the more efficient operations (the saxpy and the matrix-vector product) contribute to minimize the effects of the inner-product.

### 4.2 *Sparsity effects*

In this section we analyse the effects of the sparsity of the matrix $A$ on the performance of the matrix-vector product. When a BLAS operation is computed sequentially, one can always assume that taking account of the sparsity of $A$ will reduce the computation time.

This is easily verified if we compare the number of FLOPs performed for an operation in dense and sparse storage modes. The time taken to compute $Au$ for sparse $A$ is proportional to the number of FLOPs required, i.e., $T_{sparse} \propto 2NZ - N$ and $T_{dense} \propto 2N^2 - N$ where $Z$ denotes the number of non-zero elements per row of $A$. Taking the ratio between the two times, we have $T_{sparse} \propto Z/N \times T_{dense}$. Assuming $Z \ll N$, we have $Z/N \ll 1$, implying

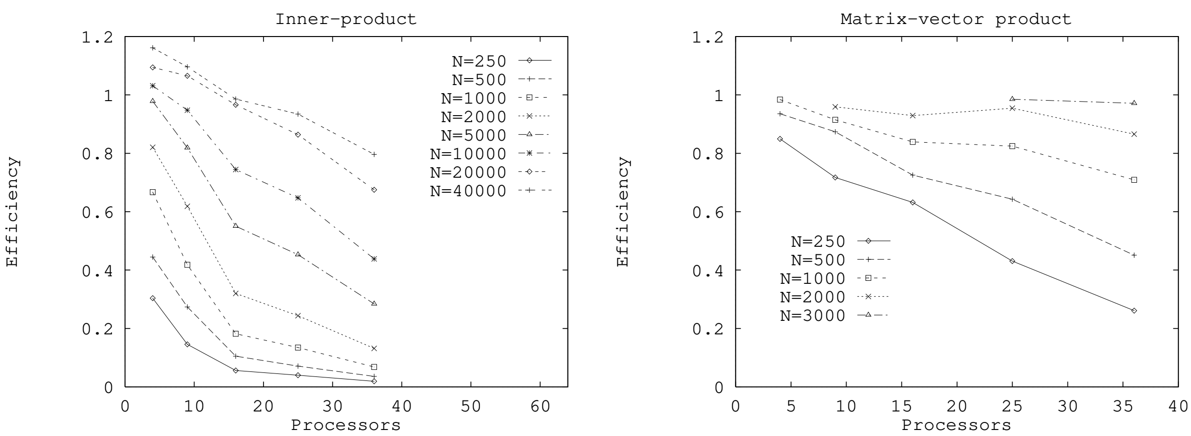



Figure 5: Efficiencies, (a) inner-product, (b) matrix-vector product.

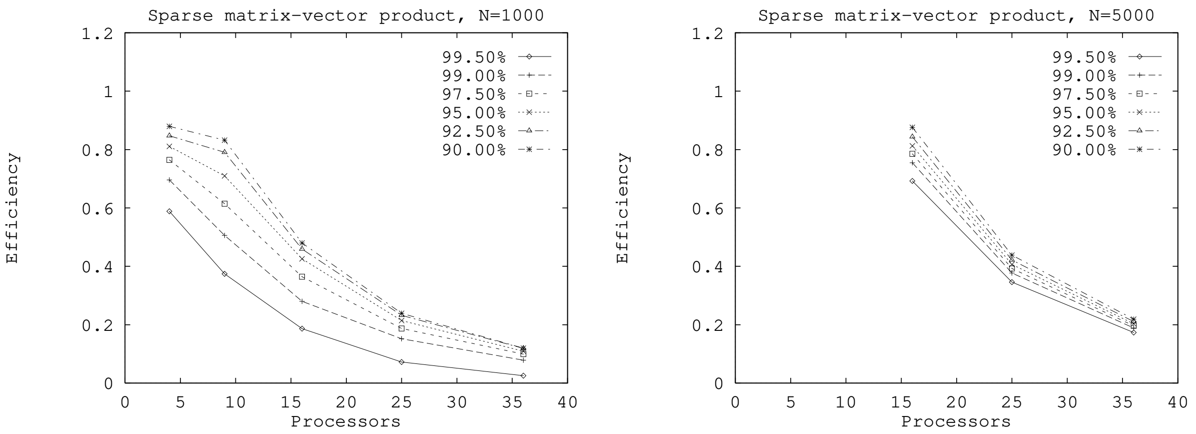



Figure 6: Efficiency for sparse $Au$, (a) $N = 1000$, (b) $N = 5000$.

$T_{sparse} \ll T_{dense}$. It should be noted that the sparse storage scheme implies an overhead in accessing the data.

This reduction in computing time may not occur when the sparse form of the matrix-vector product is distributed among the processors because the efficiency of the implementation is directly related to the ratio between the computation and communication workloads. If the amount of computation is very small (as we expect, since $Z \ll N$) then more time will be spent on communication resulting in a reduction in efficiency.

We measured the effects of the sparsity of $A$ on the performance of the matrix-vector multiply. Figure 6 shows the efficiencies obtained by $Au$ operations for $N = 1000$ and $5000$, on $P = 1$ (serial), $4, 9, 16, 25$ and $36$ processors. Each graph shows the curves of the different sparsities considered, $99.5\%$, $99\%$, $97.5\%$, $95\%$, $92.5\%$ and $90\%$. The sparsity is given by $(1 - Z/N) \times 100$.

As the sparsity increases, the efficiency is reduced, depending on the number of processors used. Up to $99.5\%$ sparsity we can use $16$ processors with an acceptable performance.

## 5 Case study – A Polynomial Preconditioned Conjugate Gradient implementation

The Conjugate Gradient (CG) method is one of the most widely used techniques for the solution of systems of linear equations. Since its revival by Reid [9], who first considered it as an iterative method, several variations have been proposed (see [1]).

CG is usually used with a preconditioner which both improves the rate of convergence and prevents divergence of the basic iterative method. It is well known that the convergence of CG is dependent on the distribution of the eigenvalues of the coefficient matrix. If the eigenvalues are in the interval $[a, 1]$, convergence will be improved if there is a clustering of the eigenvalues around $1$. Preconditioners are thus used to ensure that the preconditioned coefficient matrix has its eigenvalues with the above characteristics.

Preconditioners based on incomplete Cholesky and LU factorizations are among the most widely used (see [6]). However, these factorizations are sometimes numerically unstable and, due to the inherently sequential nature of the forward-backward substitutions needed, are not well suited to parallel implementations.

Another kind of preconditioner that has received attention in the past years is the polynomial preconditioner (see [6]). This preconditioner is attractive in a parallel environment as it may be implemented using the BLAS operations described in §3.

### *5.1 Polynomial preconditioning*

The use of a preconditioner with the Conjugate Gradient methods involves the repeated solution of systems of equations of the form $Mv = r$, where $M$ is the preconditioning matrix. To introduce the polynomial preconditioner, consider the splitting $A = P - Q = P(I - P^{-1}Q)$. Following Dubois [3], the inverse of $A$ is

$$A^{-1} = (I - P^{-1}Q)^{-1}P^{-1} = \left(\sum_{i=0}^{\infty}(I - P^{-1}A)^i\right)P^{-1} \tag{4}$$

The inverse of the polynomial preconditioner $M_m$ is taken as a truncated form of (4) with $m + 1$ terms, i.e.,

$$M_m^{-1} = \left(\sum_{i=0}^{m}\gamma_{m,i}(I - P^{-1}A)^i\right)P^{-1} \tag{5}$$

where the $\gamma_{m,i}$ are the coefficients of the polynomial of degree $m$ in $I - P^{-1}A$ and usually $P^{-1}$ is chosen to be $(diag(A))^{-1}$.

If we consider the solution of the diagonally scaled system $\bar{A}\bar{x} = \bar{b}$, then the matrix $P^{-1} = (diag(\bar{A}))^{-1} = I$ and equation (5) can be written as

$$M_m^{-1} = \sum_{i=0}^{m}\gamma_{m,i}G^i \tag{6}$$

where $G = I - \bar{A}$, which is the expression used by some authors (for example, [7] and [11]).

The matrix $M_m^{-1}$ is not formed explicitly as it can be implemented easily using matrix-vector products and saxpys. Thus the efficiency of a parallel implementation will be dependent upon the efficiency of the implementation of these two operations.

Using a Horner decomposition of the polynomial (5), the vector $v$ in $M_m v = r$ may be computed by

**PP**($m$): $v = P^{-1}r$
$\quad p = \gamma_{m,m}v$
$\quad$ for $i$=1 to $m$
$\quad\quad p = \gamma_{m,m-i}v + p - P^{-1}Ap$

Equation (6) may be computed in a similar way and is more economic in terms of the number of arithmetic operations and vectors involved.

### *5.2 Description of the algorithm*

We consider here a polynomial preconditioned version of the standard Conjugate Gradient method ([6]). Given a tolerance $\epsilon$, we use as a stopping criterion the test $|| r^{(k)} ||_2 / || b ||_2 < \epsilon$, where $r^{(k)}$ is the residual vector after the $k$-th iteration (a maximum number of iterations is set to prevent non-termination of the algorithm). The main steps of the **PPCG**($m$) algorithm are outlined as follows

**PPCG**($m$): $r^{(0)} = b - Ax^{(0)}$

$$\begin{aligned}
&\eta^{(-1)} = 1\\
&p^{(-1)} = 0\\
&\text{while } \|\, r^{(k)} \|_2 / \|\, b \,\|_2 \geq \epsilon\\
&\quad M_m z^{(k)} = r^{(k)}\\
&\quad \eta^{(k)} = r^{(k)^T} z^{(k)}\\
&\quad \beta^{(k)} = \eta^{(k)} / \eta^{(k-1)}\\
&\quad p^{(k)} = z^{(k)} + \beta^{(k)} p^{(k-1)}\\
&\quad w^{(k)} = A p^{(k)}\\
&\quad \alpha^{(k)} = \eta^{(k)} / p^{(k)^T} w^{(k)}\\
&\quad x^{(k+1)} = x^{(k)} + \alpha^{(k)} p^{(k)}\\
&\quad r^{(k+1)} = r^{(k)} - \alpha^{(k)} w^{(k)}
\end{aligned}$$

## 6 Experimental results

As test problems, we used two generic systems. The first, a nonsymmetric, dense system, is defined by

$$\begin{aligned}
a_{ii} &= 1\,, \quad i = 1, 2, \ldots, N\\
a_{ij} &= 2\,, \quad j > i\,, \quad j = 1, 2, \ldots, N\\
a_{ij} &= -2\,, \quad j < i\,, \quad j = 1, 2, \ldots, N\\
b_i &= 10\,, \quad i = 1, 2, \ldots, N
\end{aligned} \tag{7}$$

Test cases $I$ and $II$ solve system (7) for $N = 500$ and $1000$, respectively.

The second system, a sparse, symmetric, positive-definite, band matrix is defined by

$$\begin{aligned}
a_{ii} &= 10\,, \quad i = 1, 2, \ldots, N\\
a_{ij} &= 1\,, \quad i = 1, 2, \ldots, N\,, \quad j = i+1, i+2, \ldots, i+W\\
a_{ji} &= 1\,, \quad i = 1, 2, \ldots, N\,, \quad j = i-1, i-2, \ldots, i-W\\
b_i &= 10\,, \quad i = 1, 2, \ldots, N
\end{aligned} \tag{8}$$

where $W$ is the half-bandwith (number of diagonals above and below the main diagonal).

Test case $III$ is system (8) defined by $N = 2000$ and $W = 50$ being $95\%$ sparse. Test case $IV$ has $N = 2000$ and $W = 100$ with a sparsity of $90\%$. These systems were solved using the symmetric scaling, $D^{-1/2} A D^{-1/2}$.

We present the timings obtained by running PPCG on the four test problems. The parallel implementation was run on square grids using $4$, $9$, $16$, $25$ and $36$ transputers. The results of running the serial code are also given for comparison except for tests $II$ and $IV$ which could not be solved on a single transputer due to memory size constraints. We required a tolerance of $10^{-10}$ for convergence, and the polynomial preconditioner was used with $m = 1$. Table 3 presents the timings for the test cases.

Figure 7 shows the efficiencies of PPCG. Test case $I$ shows an acceptable performance for up to $16$ processors. As shown in Table 3, for $P = 36$ the execution time exceeds that for $P = 25$. Test case $II$ provides approximately $70\%$ of efficiency for $P = 25$. Comparing both problems, we expect that increasing $N$ will provide higher efficiencies on a large number of

| Test | Storage | No. of iterations | Processors | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | 1 | $2 \times 2$ | $3 \times 3$ | $4 \times 4$ | $5 \times 5$ | $6 \times 6$ |
| | | | Time(s) | Time(s) | Time(s) | Time(s) | Time(s) | Time(s) |
| $I$ | Dense | 446 | 792.4831 | 201.8778 | 102.2939 | 77.4727 | 63.7057 | 69.3276 |
| $II$ | Dense | 821 | – | 1503.6106 | 726.1686 | 463.9160 | 337.3492 | 297.7701 |
| $III$ | Sparse | 532 | 868.9976 | 255.5875 | 134.1046 | 100.5784 | 108.6247 | 114.9468 |
| $IV$ | Sparse | 1037 | – | 493.7486 | 254.0929 | 178.0733 | 141.8772 | 146.2959 |

Table 3: Timings for PPCG.

processors. Note in Table 3 that for test case $II$ the execution time for $P = 36$ is less than that of $P = 25$.

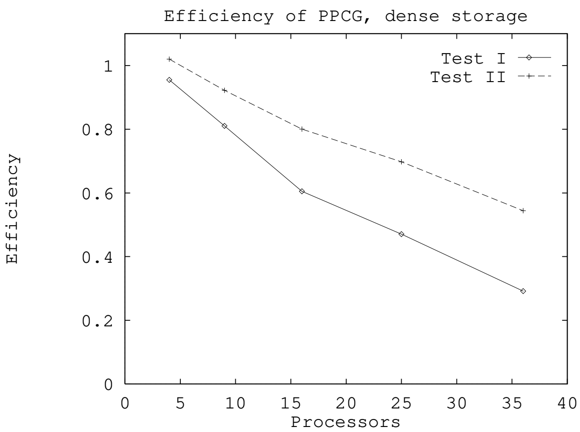
Test cases $III$ and $IV$ show that using the sparse storage we obtain less efficiency with PPCG, mostly due to the efficiencies in the sparse matrix-vector product for these problem sizes. However, increasing the sparsity contributes to an increase in the overall efficiency of the implementation.

## 7 Conclusion

We have presented some results concerning the implementation of distributed linear algebra operations on a network of transputers. The efficiency of the implementations is considerably improved by the use of loop-unrolling.

The sparsity of the coefficient matrix $A$ affects substantially the performance of the sparse matrix-vector product; high sparsity reduces the number of transputers that can be effectively used.

The BLAS discussed were used to implement serial and parallel versions of a Polynomial Preconditioned Conjugate Gradient method. The combined use of the BLAS operations means that the most efficient operations contribute to minimize the effects of the less effective ones.



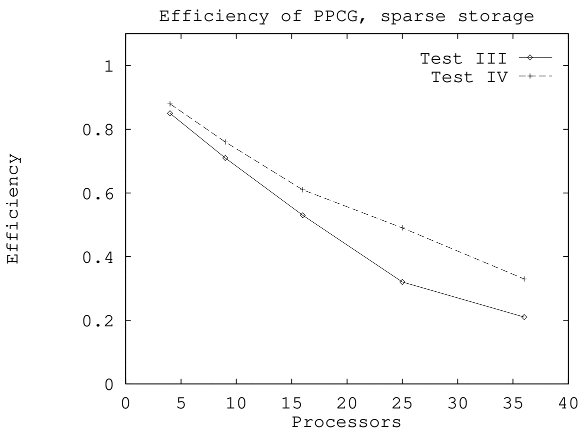




Figure 7: Efficiencies of PPCG, dense and sparse storages.

## 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] S.F. Ashby, T.A. Manteuffel, and P.E. Saylor. A taxonomy for Conjugate Gradient methods. *SIAM Journal of Numerical Analysis*, 27:1542–1568, 1990.

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

[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] I.S. Duff. *Sparse Matrices and Their Uses*. Academic Press, London, 1981.

[5] A. George and J.W. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, Englewood Cliffs, 1981.

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

[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] INMOS Limited. *Occam 2 Reference Manual*. Prentice-Hall, New York, 1988.

[9] J.K. Reid. The use of Conjugate Gradients for systems of linear equations possessing “Property A”. *SIAM Journal of Numerical Analysis*, 9:325–332, 1972.

[10] 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.

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

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