Parallel Matrix Solvers for Moment Method Codes for MIMD Computers

David B. Davidson
Dept. of Electrical and Electronic Engineering
University of Stellenbosch
Stellenbosch 7600
South Africa.
E-mail:**davidson@firga.sun.ac.za**

Abstract

Parallel algorithms are presented that are suitable for the solution of the system of linear equations generated by moment method problems on local memory Multiple Instruction, Multiple Data (MIMD) parallel computers. The two most widely used matrix solution algorithms in moment method codes are described, namely the conjugate gradient (CG) method and LU decomposition. The underlying philosophy of parallelism is briefly reviewed. Suitable parallel algorithms are then described, presented in pseudo-code, their timing behaviour analyzed theoretically, and timing results measured on a particular MIMD computer — a transputer array — are presented and compared to the theoretical timing models. It is concluded that efficient parallel algorithms for both the CG and LU exist and that MIMD computers offer an attractive computational platform for the solution of moment method problems with large numbers of unknowns.

Symbol	*Definition*
t_{comm}	Time to send one complex word between adjacent processors.
t_{calc}	Time for a real floating point + or ×.
β	The ratio t_{comm}/t_{calc}
M	Number of unknowns (dimension of the matrix).
N	Number of processors.
d	Depth of the binary tree.

Table 1: List of symbols used frequently in this paper.

1 Introduction

1.1 Background

It had long been accepted that the applicability of the moment method is limited by available computational capability, in particular memory and speed of computation [1]. For a problem with no special properties such as symmetry as a result of reflection, rotation, or translation, the

Notation	*Definition*
$[A]$	The matrix A.
$[A]^T$	The Hermitian (complex conjugate) transpose of matrix A.
$a_{i,j}$	The ij-th element of matrix A.
$[x]$	The (algebraic) vector x.
x_i	The i-th element of vector $[x]$.
$\|\|[x]\|\|$	The Euclidean norm of the vector $[x]$ of length n; $\|\|[x]\|\| = \sum_{i=1}^{n} \|x_i\|^2$.
$\|x\|$	Absolute value of scalar x.
$\lceil x \rceil$	The ceiling function of x, i.e. the smallest integer $\geq x$.
$\wedge$	The Boolean AND operation.
mod(a)	The modulo(a) operator.
$O(M^n)$	Of the order of M^n.

Table 2: Notation used in this paper.

computer time requirement grows at least as the cube of the number of unknowns, which is at least linearly related to the electromagnetic size (length, surface area or volume, depending on the particular problem and formulation) of the structure being simulated. The matrix equations to be solved are in general complex, non-symmetric and full, although certain formulations — and also physical symmetries, if present — may yield matrices with more structure. For problems which are not small electromagnetically, this presents formidably large systems of linear equations that must be filled and solved. The emergence of *vector supercomputers* has permitted the solution of much larger problems than could previously be handled. These computers, epitomized by the CRAY series, the first of which was installed in 1976, represented a tremendous increase in computational resources for researchers with access to one. However, such systems are extremely expensive, and not readily available outside the U.S.A., Europe and Japan at the time of writing. There are also limits on the computational speed of such systems. This paper considers the use of a different type of computer, the local (also known as distributed) memory Multiple Instruction Multiple Data (MIMD) computer; the algorithms described in this paper were run on an array of INMOS T800 transputers, an example of such an array. Such MIMD systems offer performance potentially rivaling that of the vector supercomputers, but require that the algorithms be very carefully designed to exploit the parallel architecture and thus obtain something approaching the manufacturer's claimed peak performance. [1] This paper concentrates on the derivation, analysis, implementation and testing of such algorithms, for the conjugate gradient (CG) and LU matrix solvers, and is an extension of previous papers by the author [3, 4].

[1]The transputer array used in this paper does not deliver performance on par with conventional supercomputer systems such as the CRAY machines already mentioned. However, in the light of the next generation of massively parallel arrays — with hundreds or thousands of processors compared to the dozens used in this paper, and with each processor running far faster than the transputers used here — the conventional supercomputer "now seems poised for an indefinite but inexorable decline" [2, p.27].

1.2 Parallel Processing

The fundamental principle underlying parallel (or concurrent) processing is that once the limits on speed imposed by a certain computing technology have been reached, the most obvious way of building a faster computer is to perform operations simultaneously. Two fundamental ways of implementing parallelism have emerged, namely pipelining and replication. The former involves overlapping parts of operations in time and is the approach taken by the vector supercomputers; the latter provides more that one functional unit (e.g. CPU), permitting operations to be performed simultaneously, and is the approach taken by such systems as arrays of transputers or i860 processors. The historical background of parallel computers and a more detailed explanation of pipelining and replication may be found in the author's tutorial paper [3], and with minor revisions in [5, Chapter 3].

Several methods have been proposed to characterize parallel computers, but the most widely used are speed-up and efficiency. Speed-up, S, is the ratio of time taken by an equivalent serial algorithm running on one processor, T_s, to the time taken by the parallel algorithm using N processors, T_p. Efficiency, ϵ, is the speed-up normalized by the number of processors. Formally,

$$S = \frac{T_s}{T_p} \tag{1}$$

$$\epsilon = \frac{S}{N} \tag{2}$$

S is usually bounded from above by N and ϵ is hence usually bounded from above by 1 — although under very special circumstances an efficiency exceeding 1 is at least theoretically possible [5, Section 3.4.1]. Speed-up is the fundamental issue of importance for the user — it states how much faster his algorithm will run on N processors than on one. Efficiency is self-evident. The most important requirement for a parallel program — other, obviously, than its correctness — is to obtain the maximum possible speed-up, and thus also efficiency, from the available parallel hardware.

At present a major effort is required by the user to properly exploit parallel processing, in particular for MIMD systems. Automatic vectorizing compilers have simplified the task for pipelined vector computers, and similar tools exist for very small MIMD systems (with 2 or 4 processors), but for large scale MIMD systems the *user* must frequently carefully select, analyse and implement suitable parallel algorithms. On some MIMD systems, some parallelized basic linear algebra algorithms may be available, either from the manufacturer or from software companies, but this was certainly not the case with transputer arrays. Even when such software is already available, the timing models described in this paper should still be useful.

1.3 The Local Memory Message Passing MIMD Computer

The parallel algorithms and timing models considered in this paper have been developed for a particular type of Multiple Instruction, Multiple Data (MIMD) computer, namely arrays of INMOS T800 transputers. The algorithms have been implemented in Occam 2 to validate the theoretical analysis. [2] However, the assumptions made regarding the computer are representative of a substantial *class* of parallel computers, namely local memory message passing MIMD systems, so the algorithms and timing analyses are applicable to other computers in this class.

[2]Occam is a parallel language based on the work of Hoare on Communicating Sequential Processes (CSP); see [3] for more details. The transputer was designed to very efficiently implement the CSP paradigm.

It is important to clearly indicate the properties of this type of computer, so that other researchers with different hardware will be able to establish the suitability of the algorithms, and where modifications to the theoretical analysis will be required, for their computers. The MIMD classification was introduced by Flynn [6] and describes a computer consisting of a number of nodes [7, p.485], each with at least a processing element, which operates independently on its own local instruction stream and data. The further characterization of the machine as *local memory, message passing* derives from the memory allocation and communication methods. On a local memory system, all memory is divided up amongst the available processors, and a processor may only directly access its own memory. Access to the memory on other processors is done by explicit message passing, which is *much* slower than direct memory access. The problem of memory contention that complicates the other main competing approach to memory allocation, namely global memory, is removed with this approach, but the absence of global memory can complicate the algorithm — an example will be given later in this paper. It is further assumed that the computer uses explicit message passing over processor to processor communication channels (links) — as opposed to communication over a common bus, for example — for all communication (including both data and synchronization information). It is assumed that each processor has four such links and these links can operate *concurrently* with high efficiency. This theoretical model describes an array of transputers accurately. More details on transputer arrays may be found in [3, 8, 9].

The algorithms derived in this paper use interconnection topologies requiring at most only four links; the number of links required for both the mesh (four) and the binary tree (three) is not a function of the number of processors. These topologies are illustrated in Figures 1 and 2. [3] Four communication links are required to build a two-dimensional grid, a very useful general purpose topology, so four is a reasonable lower bound on the number of links required. The hypercube topology, [3, Section 6.2] and [7], has attracted much attention, and is possibly the most useful general purpose topology currently in use. The hypercube has the attractive property that for a given number of processors, the diameter (the maximum number of links required to connect any two nodes) is smaller than for many other topologies; see [3, Table I]. However, the number of inter-processor links grows as the dimension d of the hypercube; a hypercube of dimension d has $N = 2^d$ processors. While this is a fairly slow (logarithmic – $\log_2 N$) growth in the dimension, and hence number of links required, as a function of the number of processors, this nonetheless imposes limits for systems with a limited number of links. For example, transputer based hypercubes are limited to 16 processors. Fox et. al. have described a number of algorithms that run on hypercubes [7]. Both the topologies (the binary tree and the two-dimensional mesh) used in this paper may be mapped onto hypercube topologies (see [7, Chapter 19] and [7, Chapter 14] respectively), so the algorithms to be described are also suitable for hypercube MIMD computers. It is possible that fully exploiting the greater connectivity of hypercube machines may yield more efficient algorithms than those presented here.

The theoretical results derived in this paper depend on only two machine dependent parameters, viz. the speed of computation and communication. The link concurrency discussed above was exploited to varying degrees, and is discussed in the relevant analysis. The methods developed in this paper permit one to establish at least approximately, from the manufacturer's specifications and benchmarking, whether particular parallel computer hardware will be suitable

[3]The mesh shown in Figure 2 has column wrap-around, but not row wrap-around. The reason for this is rather subtle: a transputer array has to have one link connected to the "host" — typically a PC — and if row-wrap-around was used as well, no spare link would be available. While it is possible to work around this problem, the coding becomes rather messy. Exploiting full wrap-around would reduce the communication cost slightly, but with the pipelined communication used in this paper, the improvement would not be very significant.

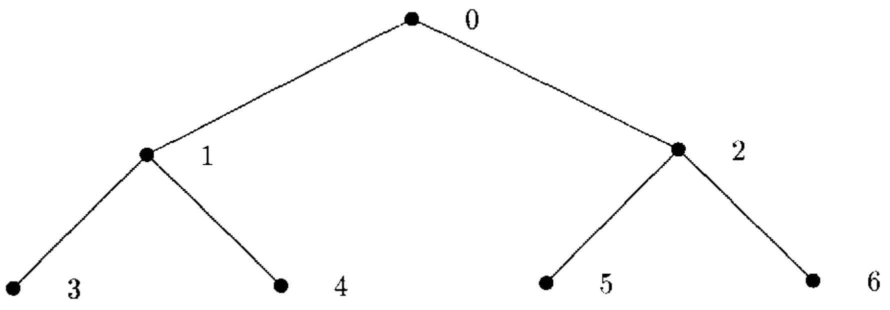

Figure 1: Interconnection topologies - binary tree dimension 2.

for moment method solutions, and is an important step towards *quantifying* the performance of parallel hardware for important algorithms in computational electromagnetics.

Other researchers [8, 10, 9, 11] have also addressed aspects of parallel processing in electromagnetics, all using transputers, and have shown impressive speed-ups and efficiencies. However, these papers have concentrated on measured results, rendering difficult the application of their results to other types of processor arrays, as well as the extrapolation of the efficiencies of their algorithms to larger arrays. Hafner's paper [8] deals with transputer hardware and software in some detail, as well as the parallelization of a Multiple Multi-Pole program using an early parallel FORTRAN compiler. Nitch's work was on the parallelization of the moment method code NEC2 using a mixture of Occam and FORTRAN. Cramb et al. [9] used the processor farm paradigm for what would be classified as a very "coarse grain" decomposition — essentially the same code was run at different scan angles, with communication only between the controller and the worker processor executing the specific set of scan angles. Russel and Rockway [11] used the ParaSoft EXPRESS operating environment, which provides a number of communications routines of the type implemented explicitly in this paper. Their results for four processors were impressive, but they do not address the scaling behaviour of the algorithm for more processors.

Computer technology moves so rapidly that any paper published giving absolute run-times and computational benchmarks is out of date almost as it goes to print. A comparison of the computational speed obtained with the algorithms described in this paper running on transputer hardware with what may be expected from a typical workstation *at the time of writing* is given in the conclusions of this paper; it must be emphasized that the main thrust of this paper is to describe suitable parallel algorithms for the broad class of local memory MIMD parallel processors — of which the transputer is an contemporary example — and to develop methods for predicting performance of parallel algorithms at least approximately, rather than promoting transputer technology *per se*.

2 A Parallel Conjugate Gradient Algorithm

2.1 Iterative Algorithms and the Conjugate Gradient Algorithm

Over the past decade, much effort has been expended in the application of iterative methods, and in particular the conjugate gradient and related algorithms, to computational electromagnetics.

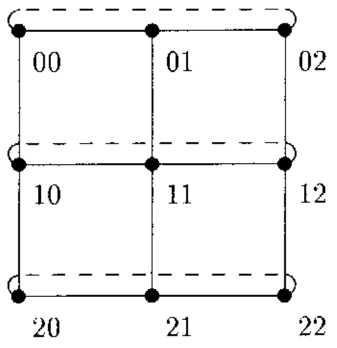

Figure 2: Interconnection topologies - mesh (lattice) with column wrap-around. See text for further discussion of the wrap-around.

Representative references may be found in Wang's recent book [12, p.68]. [4] Golub and O' Leary's paper provides a recent and comprehensive review of the mathematical history of the algorithm, with an annotated bibliography [14]. A compact description of iterative methods in general and the conjugate gradient algorithm in particular may be found in Jennings [15, Chapter 6]. Regarding parallel iterative algorithms, very little appears to have been published on solvers for full matrices, and what has been published has been frequently directed at different architectures, for example the recent book by Dongarra *et al.* [16] on solving linear systems, which concentrates on vector and shared memory computers.

The CG method, extended for the general case of a matrix $[A]$ with complex entries where the matrix is not known to be positive definite, is as follows [15, pp.220-221]:

$$\begin{aligned}
[u_k] &= [A][p_k] && \text{Step 1}\\
\alpha_k &= \tfrac{\|[\bar{r}_k]\|^2}{\|[u_k]\|^2} && \text{Step 2}\\
[x_{k+1}] &= [x_k] + \alpha_k[p_k] && \text{Step 3}\\
[r_{k+1}] &= [r_k] - \alpha_k[u_k] && \text{Step 4}\\
[\bar{r}_{k+1}] &= [A]^T[r_{k+1}] && \text{Step 5}\\
\beta_k &= \tfrac{\|[\bar{r}_{k+1}]\|^2}{\|[\bar{r}_k]\|^2} && \text{Step 6}\\
[p_{k+1}] &= [\bar{r}_{k+1}] + \beta_k[p_k] && \text{Step 7}
\end{aligned} \tag{3}$$

with initial values $[r_0] = [b] - [A][x_0]$ and $[\bar{r}_0] = [p_0] = [A]^T[r_0]$. This algorithm is suitable for application to the matrix set up by the method of moments. Later in this paper, the question of

[4]There has been a long-running debate in the electromagnetics literature on the relationship of the "direct" application of the CG method to the underlying operator equation as opposed to the use of the method as a matrix solver for the matrix set up by the method of moments, see for example [13] and more recently [5, Chapter 2]. This point will not be pursued further in the present paper, which is directed at the solution of the matrix equation set up by the method of moments.

Step	*Complex operations*	*Real FLOP count*
1	$2M^2$	$8M^2$
2	$4M$	$16M$
3	$2M$	$4M$
4	$2M$	$4M$
5	$2M^2$	$8M^2$
6	$4M$	$16M$
7	$2M$	$4M$

Table 3: FLOP count of conjugate gradient algorithm. M is the number of unknowns or equivalently, the dimension of the matrix.

whether the convergence of the CG method justifies its application to a full matrix is discussed.

Note that there are a number of very closely related conjugate gradient algorithms; one recently discussed in the electromagnetics literature is the bi-conjugate gradient (BiCG) method [17]. The author has also implemented the BiCG algorithm; the modifications required to implement it have very little effect on the timing analysis. While the BiCG algorithm sometimes accelerates convergence [18], it can also slow down convergence or stagnate [5, Section 6.8],[17]. Pre-conditioning is also widely used in the Finite Element community to accelerate the convergence of the CG method; unfortunately, previous work by the author on the application of pre-conditioning indicated that it was not suitable for moment method matrices [18]. The reason for this is not clear.

The floating point operation (FLOP) count per iteration is shown in Table 3, retaining only the largest order term for each operation. (Because of this, a term $-2M$, with M the number of unknowns, is missing in the real operations counts in both Steps 1 and 5; this comes from the number of additions, which is actually $M(M-1)$, not M^2. The impact on the analysis is minimal; it is convenient to use the M^2 approximation for the parallel matrix-vector analysis, and this also indicates clearly the difference between the parallelized matrix-vector products and the unparallelized vector operations in subsequent results.) Note that α_k and β_k in Steps 3, 4 and 7 are real, not complex, and this affects the conversion from complex to real FLOPs. One complex addition is equivalent to two real FLOPs and one complex multiplication is equivalent to six real FLOPs; since it is the number of additions and multiplications that dominate the FLOP count, and furthermore the addition and multiplication FLOP counts are almost identical, an average factor of four can be used. (On most modern processors, the time required for a floating point addition and a floating point multiplication are approximately the same: benchmarking the transputer yielded *exactly* the same times for both operations.)

2.2 Parallel Matrix-Vector Products

From Table 3, the computationally expensive parts of the CG method can be seen to be the two matrix-vectors products — Steps 1 and 5, of $O(M^2)$ whereas the other steps are of $O(M)$ — hence efficient parallel matrix-vector product algorithms, taking into account the hardware limitations discussed in the Introduction, are required. (The work of Fox *et al.* discusses parallel matrix-vector products for hypercube architectures [7, Section 21-3.4], and uses a decomposition different from that considered here).

The product of a $M \times M$ matrix by a vector of length M can be considered from two viewpoints. The first is as the forming of M inner products. These inner products can be computed in parallel.

The second approach is as the forming of M^2 products, followed by an accumulation process. The M^2 products can be computed in parallel, and the accumulation process can be parallelized. The computational dependence of both is very similar — detailed expressions will be shown shortly. These viewpoints imply at least the following two possibilities for forming a parallel matrix-vector product:

- *Row-block decomposition*: Splitting up the matrix by row block, *broadcasting* the vector to all processors, performing the inner products in parallel and then *gathering* together the different parts of the vector split up over the processors

  or

- *Column-block decomposition*: Splitting up the matrix by column, *scattering* the vector over the processing array, performing partial inner products in parallel, and then *accumulating* the resultant vector. This is a special case of the M^2 parallel product approach, with all the elements of a column *clustered* (grouped) on a processor, and entire columns clustered in turn.

The four communications paradigms required by the two different decompositions can be formally defined as follows, assuming N processors and a matrix dimension of M:

1. *Broadcast*: This process broadcasts identical copies of the same vector to all the elements of the array.
2. *Gather*: This process builds a vector up from its N disjoint sections of length M/N distributed over the array after the parallel matrix/vector product.
3. *Scatter*: This process is the inverse of *gather* in that it scatters a vector over the array so that each of the N processors has a different vector of length M/N.
4. *Accumulate*: This process accumulates the partial inner products resulting from the column-block decomposition.

A graphical illustration of the operation of the two possible algorithms may be found in [3] and [5, Chapter 4], where the communication algorithms are also described in more detail.

The next stage of the development of a parallel algorithm is the identification of a suitable topology, i.e. interconnection topology. This issue has been addressed in detail in [3], [5, Chapter 3], and also in [8], and the restrictions imposed by the transputer hardware have already been discussed in the Introduction. Considering the type of communications required, the binary tree, an example of which is shown in Figure 1, is a natural topology for this problem, for the following reasons. It is only necessary to communicate information to and from the (controlling) processor at the top of the tree from and to other lower level processors, and not from one side of the tree to the other. Thus for approximately the same number of processors, the *effective* diameter of the binary tree is actually one less than the diameter of the equivalent hypercube. The processor at the top of the tree can either be used purely for co-ordinating the process, or can also share the workload. The algorithm described here follows the former process. It is possible to use a ternary tree — and the enhanced parallel communications will produce a more efficient algorithm — but this does not map conveniently onto available arrays, where the available number of processors generally follows some power of two. Thus the choice of topology is motivated not only by the algorithm, but also by the available hardware, and typical configurations thereof.

```
begin{broadcast section: worker}
  receive vector from higher processor
  if (not at bottom of tree)
    then
       par
         send vector to lower left processor
         send vector to lower right processor
       end{par}
    else SKIP
  end{if}
end{broadcast section: worker}
```

Figure 3: Pseudo-code for broadcast: worker process

Having identified the parallelism in the problem, the next stage of algorithm analysis is the development of timing equations. These will allow the prediction of the speed-up and efficiency defined in equations (1) and (2). The timing equations have been derived in [3] and [5, Chapter 4] and only the results will be presented. Defining the time required to send one complex word, consisting of the real and imaginary parts — 8 bytes in single precision and 16 bytes in double precision on a transputer, and indeed any system implementing IEEE arithmetic — from one processor to another directly connected processor as t_{comm}, it may be shown that the communication requirements of the matrix-vector product algorithms for $M \gg 1$ are as follows [3] and [5, Chapter 4]:

$$t_{broadcast} = M d \, t_{comm} \tag{4}$$

$$t_{gather} = M[1 - d/N] \, t_{comm} \tag{5}$$

$$t_{scatter} = M[1 - d/N] \, t_{comm} \tag{6}$$

$$t_{accumulate} = M d \, t_{comm} \tag{7}$$

where d is the depth of the binary tree. Since the top-most processor is used purely for co-ordinating the process, the number of worker processors is $N = 2^{d+1} - 2$.

It is important to note that in deriving these results, it has been assumed that the communications parallelism available on a transputer has been exploited — this has been discussed in the Introduction. Figure 3 shows an example of this for the broadcast primitive running on the worker processors. The algorithms are most conveniently documented using pseudo-code — flowcharts are very rarely used for parallel algorithms. The pseudo-code used, loosely based on Pascal, is formally defined in [5, Section 3.7]. The meaning of the code should be intuitively obvious to anyone used to high-level, structured languages. The only construct that may be new is the **par** construct; the code stubs within **par** ...**par{end}** are executed in parallel.

Note also that there is a certain amount of computation that occurs after each communication phase with the accumulate paradigm, arising from the addition of two vectors at each level; this should be included in the overall compute time. The additional term is $2Md$ (the factor 2 arising from the conversion from complex to real arithmetic). The use of pipelining, to be discussed later, has not been considered here.

The amount of computation involved in a matrix-vector product is M^2 multiplications and $M(M-1)$ additions. Thus the total amount of computation is approximately $2M^2$ complex flops or $8M^2$ real flops. This is T_s, the time for the serial operation. The time for the parallel operation, $T_p(N)$, is the sum of the computation time for the parallelized matrix-vector product, viz. $8M^2/N$, and the communication time for either the row-block or column-block decomposition. The details of the derivation of the speed-up and efficiency have been given in [3]. Only the result for the following important case will be shown. If n_r is defined as the number of rows per processor, $n_r = \frac{M}{N}$, then for $N \gg 1$, $N \approx 2^{d+1}$, hence $d \approx \log_2 N - 1$, and the following approximate formulae for ϵ is obtained:

$$\epsilon = \frac{1}{1 + \frac{\beta}{8n_r}\log_2 N} \tag{8}$$

where $\beta = t_{comm}/t_{calc}$ is the ratio of communication to computation speed. t_{calc} is the time required for a real floating point addition or multiplication. This formula is very important; it indicates clearly that the matrix-vector product scales essentially with n_r^{-1}, the inverse of the number of rows per processor, and rather weakly (logarithmically) with the number of processors, N. Hence, for a given n_r, the efficiency is almost independent of the number of processors. This prediction is confirmed by the measured results shown in Figure 7.

In reality, the dimension of the problem will not usually be an integral multiple of the number of processors. This can be handled by either loading different processors differently or by padding the matrix and vector with the necessary zeros. This can be incorporated into the preceding analysis by replacing n_r by $\lceil n_r \rceil$. The effect on a plot of the efficiency as a function of M (or n_r) is to replace the smooth curve implied by equation (8) by a stairstep function. This point will be understood as read in the rest of the paper.

The actual run-time of the algorithm can be obtained approximately from $t_{calc}\frac{8M^2}{S}$, indicating the obvious importance of maximizing S for a given N.

2.3 The parallel CG algorithm

The timing analysis for the matrix-vector product of the preceding subsection can now be incorporated into a parallel conjugate gradient algorithm, and S and ϵ predicted. The algorithm exploits the complementary roles of the row- and column-block decomposition; the matrix-vector product is done using the row-block decomposition and the (Hermitian) transpose matrix-vector product is done using the column-block decomposition (with the necessary change of sign of the imaginary part of the matrix entries). This avoids having to either explicitly form the matrix transpose during each operation — a very expensive operation on a parallel processor with *local* memory, since this requires an O(M^2) interchange operation at each iteration — or store an additional copy of the Hermitian transpose of the matrix — and thus double the memory requirements of the code. This important contribution was the author's [3], and has not been published elsewhere, to the best of his knowledge. It is notable that an operation as simple as forming the transpose of a matrix — a trivial interchange of indices on a serial processor — can pose a major problem on a parallel system. Pseudo-code for the algorithm is given in Figures 4 and 5 for the master and workers respectively. Only the matrix-vector products have been parallelized (Steps 1 and 5 in Table 3); the other vector update operations are performed on the master processor at the top of the tree.

From Table 3, the serial time is

$$T_s = (16M^2 + 44M)t_{calc} \tag{9}$$

```
process[master.cg]
begin
  initialization
  while (not finished)
    begin
      broadcast p.k
      gather u.k
      compute alpha.k
      update x.k+1 and r.k+1
      scatter r.k+1
      accumulate r.bar.k+1
      compute beta.k
      update p.k+1
      compute and print normalized residual
      check termination
    end
  end{while}
end{process[master.cg]}
```

Figure 4: Pseudo-code for parallel CG algorithm: master process

```
process[worker.cg]
begin
  initialization
  while (not finished)
    begin
      broadcast p.k
      perform matrix-vector product
      gather u.k
      scatter r.k+1
      perform transpose matrix-vector product
      accumulate r.bar.k+1
      check termination
    end
  end{while}
end{process[worker.cg]}
```

Figure 5: Pseudo-code for parallel CG algorithm: worker process

Precision	*Operation*	*MFLOP/s*
Single	Addition	0.53
Double	Addition	0.38
Single	Multiplication	0.53
Double	Multiplication	0.38

Table 4: Computation benchmarks on the University of Stellenbosch's T800 transputer array.

The parallel time is the sum of the parallelized matrix-vector products, the unparallelized vector operations and the additional computational overhead of the accumulate paradigm, and the communication requirements of the broadcast, gather, scatter and accumulate paradigms:

$$T_p = (16M^2/N + 44M + 2dM)t_{calc} + (2M[1 - d/N] + 2Md)t_{comm} \tag{10}$$

Forming the quotient of T_s and T_p and simplifying yields

$$\epsilon \approx \frac{1 + \frac{2.75}{M}}{1 + \frac{N}{M}(2.75 + 0.125d + \frac{[1+d(1-\frac{1}{N})]\beta}{8})} \tag{11}$$

Note that this result is actually the efficiency of one iteration; since by far the majority of time required by the algorithm is in the iterative cycles, the algorithm as a whole can be characterized by its performance per iteration.

Under the assumption $M, N \gg 1$, this can be simplified to

$$\epsilon \approx \frac{1}{1 + \frac{N}{M}(2.75 + 0.125d + \frac{\log_2 N\beta}{8})} \tag{12}$$

Attention must be paid to correctly terminating parallel algorithms: if not done correctly, certain processes will never terminate, and re-initialization of the array may be required before any other code will load. In the code developed by the author, the termination criteria is that either the normalized residual error must have decreased to less than the user-specified value or that some maximum number of iterations must have been executed (the conventional criteria for an iterative algorithm). The former can only be determined by the master processor. Hence it is necessary for the master process, at the end of each iteration, to monitor the termination criteria. If one (or both) of the termination criteria has been satisfied, then the master must explicitly inform the workers, who then inform the lower level workers and terminate their execution. The configuration program that loads the worker processors and correctly allocates software abstractions (channels in Occam) to hardware (links on a transputer) for an arbitrary depth of binary tree also requires attention; this is dependent on the specific language and configuration meta-language. A suitable configuration for the Occam code developed by the author is given in [5, Appendix A].

This analysis requires two fundamental parameters to characterize the machine: the computation and communication speeds. The most reliable way of obtaining this data is by *benchmarking* - actually measuring the performance of the system under conditions simulating those of the actual code. Two simple benchmarks were developed: the first tested computation speed and the second communication speed. Such benchmarking is necessary for any parallel computer; pseudo-code useful for benchmarking local memory MIMD systems is presented in [5, Section 4.7]. Results are shown in Tables 4 and 5. The parameter β can now be computed from the benchmark results

Precision	*MByte/s*
Single	1.32
Double	1.39

Table 5: Communication benchmarks on the University of Stellenbosch's T800 transputer array.

Precision	β
Single	3.22
Double	4.37

Table 6: $\beta = t_{comm}/t_{calc}$

for the case of single and double precision. The numerical values given in Table 6 are for the transputer array used to generate the results that follow. [5]

2.4 Results and Discussion

This section describes results obtained by the author using his Occam 2 implementation of the algorithm. It was run on a 64 transputer array, developed in South Africa by the Council for Scientific and Industrial Research. The array is known as the Massively Concurrent Computer/64 (MC^2/64, or MC^{2}in this paper). [6] The array has been described in [3, Section 4.1]. At the time of running the timing tests, it was only possible to use half the array, for technical reasons: firstly, the memory was not homogeneously distributed, and secondly, some problems with the inter-cluster switching (from one "cluster" of 16 processors to another) were being experienced. These results represent the experimental validation of the timing models developed. Although the pseudo-code given in Figures 4 and 5 appears simple, much detail — especially in the communication routines — is hidden, and the debugging was very time-consuming and tedious, due to the absence of interactive parallel debuggers. This code was also developed before useful books on the subject such as [7] were available.

Measured efficiencies are shown in Figure 6. Theoretically, equation (11) predicts that the efficiency should be a function mainly of the number of rows per processor, $\frac{M}{N}$, and a weak function of d, the depth of the tree. These predictions are confirmed in Figure 7. *Thus this parallel CG algorithm exhibits a most desirable property - it scales with the number of rows per processor. With a* given *number of rows per processor, the efficiency of the algorithm is a rather weak function of the* number *of processors.* The measured and predicted results for 30 workers are shown in Figure 8. The maximum problem size is limited by the available memory; at the time of writing a maximum of 64MB of usable memory was available.

It will be noted that in Figure 8, the measured and predicted curves agree very well regarding the *shape* of the curve, but there is an offset between the measured and predicted curves. Similar results — not shown in this paper — were obtained for other numbers of processors. It should be

[5]One of the reviewers queried these benchmarks. However, these results agree closely with those reported in [8, Table I] for the FORTRAN benchmarks, when the off-chip RAM is used. Using on-chip RAM yields rather faster results [8], but there is only 4kB of this, so any real application program has to use the off-chip RAM. The computational benchmark was constructed to avoid measuring loop overhead.

[6]The name "Massively" seems rather presumptuous in retrospect, but when initially mooted, the system was indeed massive.

noted that the aim of the modelling is not to be able to predict the performance exactly in the sense that one predicts an antenna's radiation pattern; the aim is simply to indicate trends and determine whether the performance (efficiency) will be satisfactory for the problems of interest. Furthermore, the predictions serve as a check on the correct functioning of the code. Possible causes of the differences are latency (the time to initiate communication), loop overhead and differences between the coding and the model caused by some usage restrictions in Occam. More details may be found in [5, Section 4.8].

The measured data shown was obtained from *PARNEC*, a parallel version of the thin-wire part of the moment method code NEC2 developed by the author in Occam 2 [5, Chapter 6]. Double precision was used.

The efficiency of the parallel CG algorithm that has been described in this paper could be further increased by exploiting communication pipelining, a concept that will be described with reference to the parallel LU algorithm. Some further comments on improving the efficiency of the algorithm will be made in the Conclusions of this paper.

3 A Parallel LU Algorithm

3.1 The Basic LU Algorithm

The LU method is probably the most widely used algorithm for the solution of square systems of linear equations. Given a system with a moderate number of equations, it is usually the best algorithm to use, provided that the system is not extraordinarily ill-conditioned. On serial processors [7], LU decomposition followed by forward and backward substitution is always better to use when solving a system of equations than forming the explicit inverse of the matrix [19, p.347]. Given the fundamental role of the LU algorithm, the development of an *efficient* algorithm suitable for a local memory MIMD array is an essential research topic for parallel computational electromagnetics.

Before considering the parallel version of the LU algorithm, it is necessary to review briefly the serial form of the algorithm. The LU algorithm factors a matrix A into the product of an upper (U) and lower (L) triangular matrix. The diagonal elements of L are most commonly chosen as 1 — although other choices are also useful, for example Choleski decomposition. [8] The algorithm can be found in virtually any book on matrix algebra, for example [19, p. 359]. The algorithm consists of M main steps. Step i, which computes the i-th row of U and the i-th column of L, is repeated for $i = 1, \ldots, M-2$ and is defined as follows:

begin{Step i}

$$u_{i,i} = \frac{1}{l_{i,i}}[a_{i,i} - \sum_{k=0}^{i-1} l_{i,k}u_{k,i}]$$

Repeat for all $j = i+1, \ldots, M-1$:

$$u_{i,j} = \frac{1}{l_{i,i}}[a_{i,j} - \sum_{k=0}^{i-1} l_{i,k}u_{k,j}] \tag{13}$$

$$l_{j,i} = \frac{1}{u_{i,i}}[a_{j,i} - \sum_{k=0}^{i-1} l_{j,k}u_{k,i}] \tag{14}$$

[7]It was brought to my attention by a reviewer that some researchers have concluded that this may not be true on certain parallel systems such as the Connection Machine.

[8]Note that Choleski decomposition is only applicable to symmetric positive definite matrices [15, p.107]. Matrices set up by the moment method do not generally have these properties.

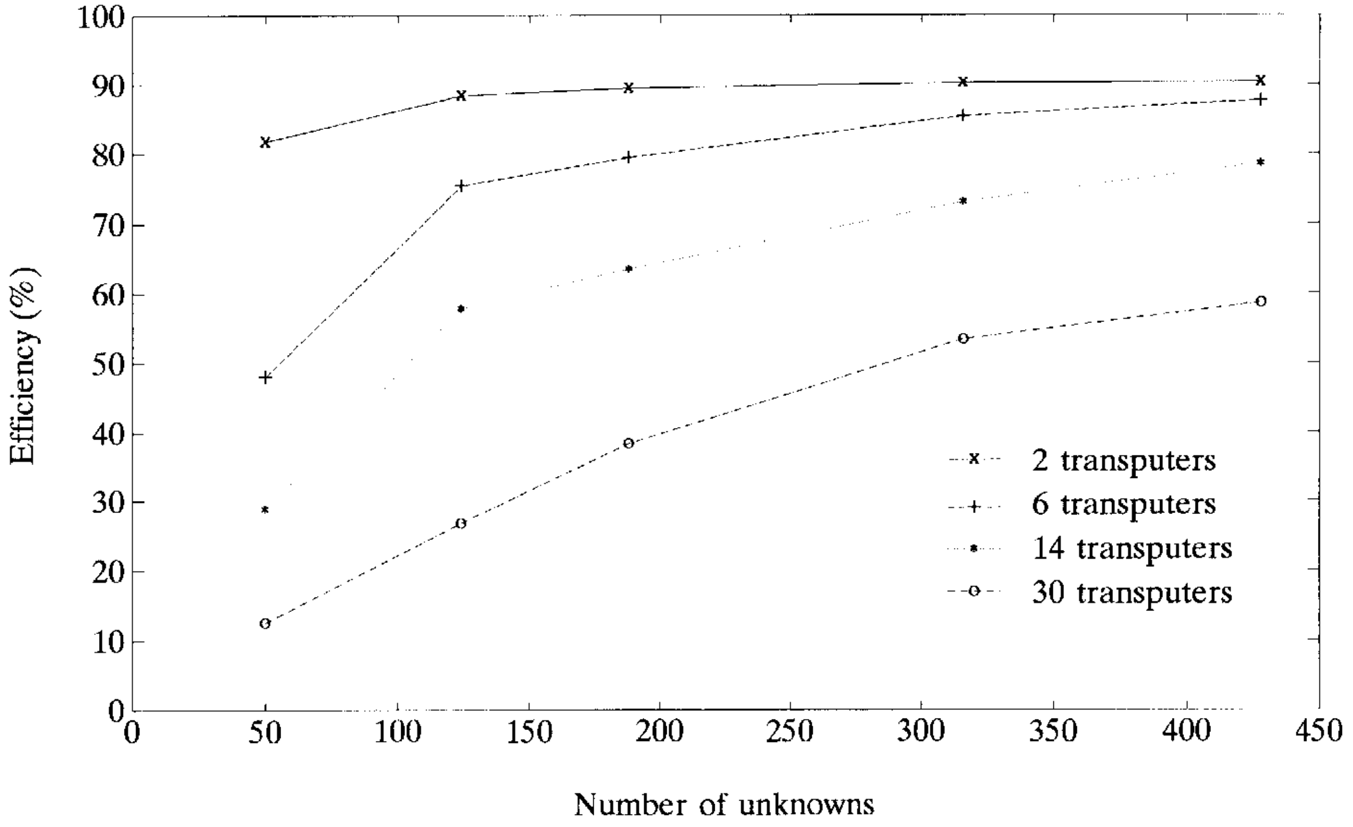

Figure 6: Measured efficiencies of the double precision parallel conjugate algorithm versus unknowns for the MC^2.

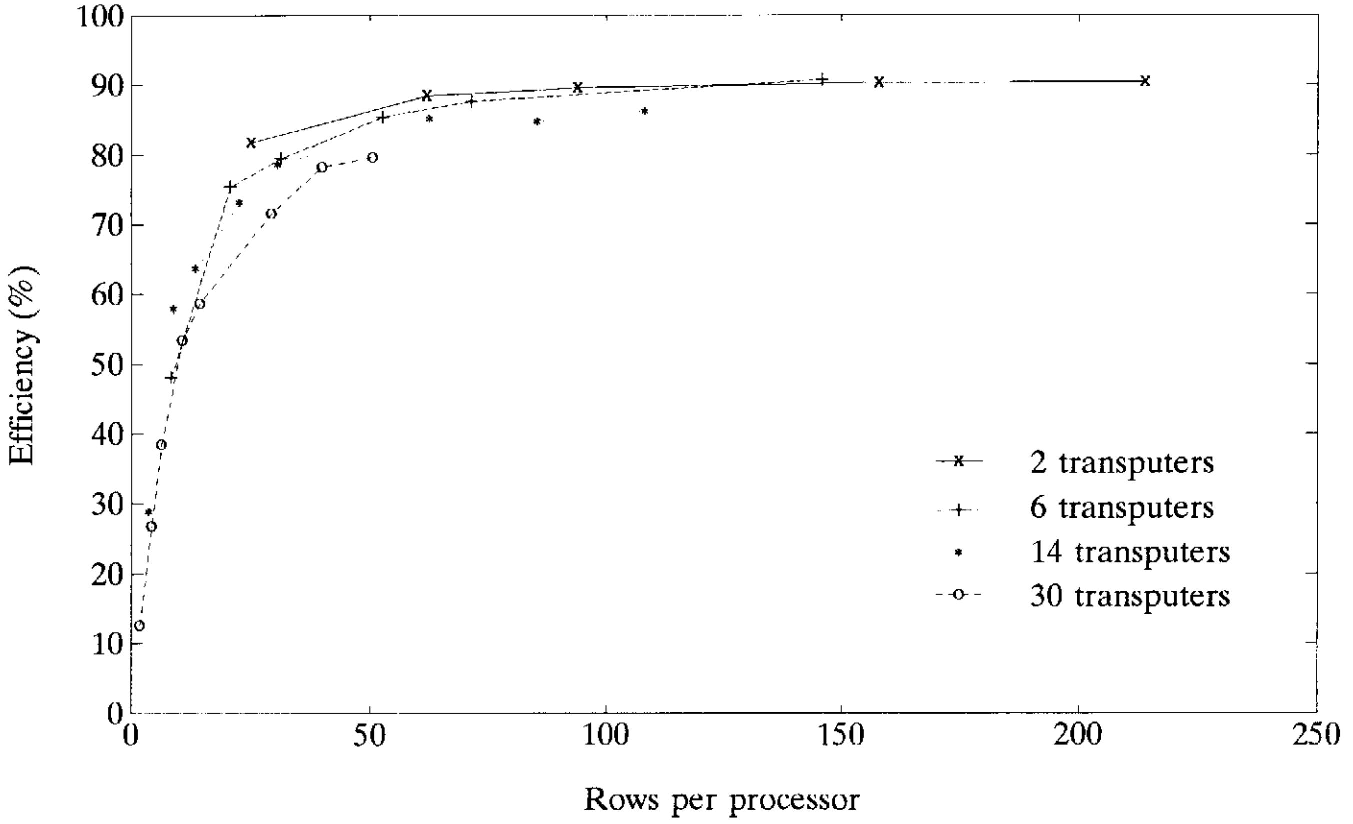

Figure 7: Measured efficiencies of the double precision parallel conjugate algorithm versus rows per processor for the MC^2.

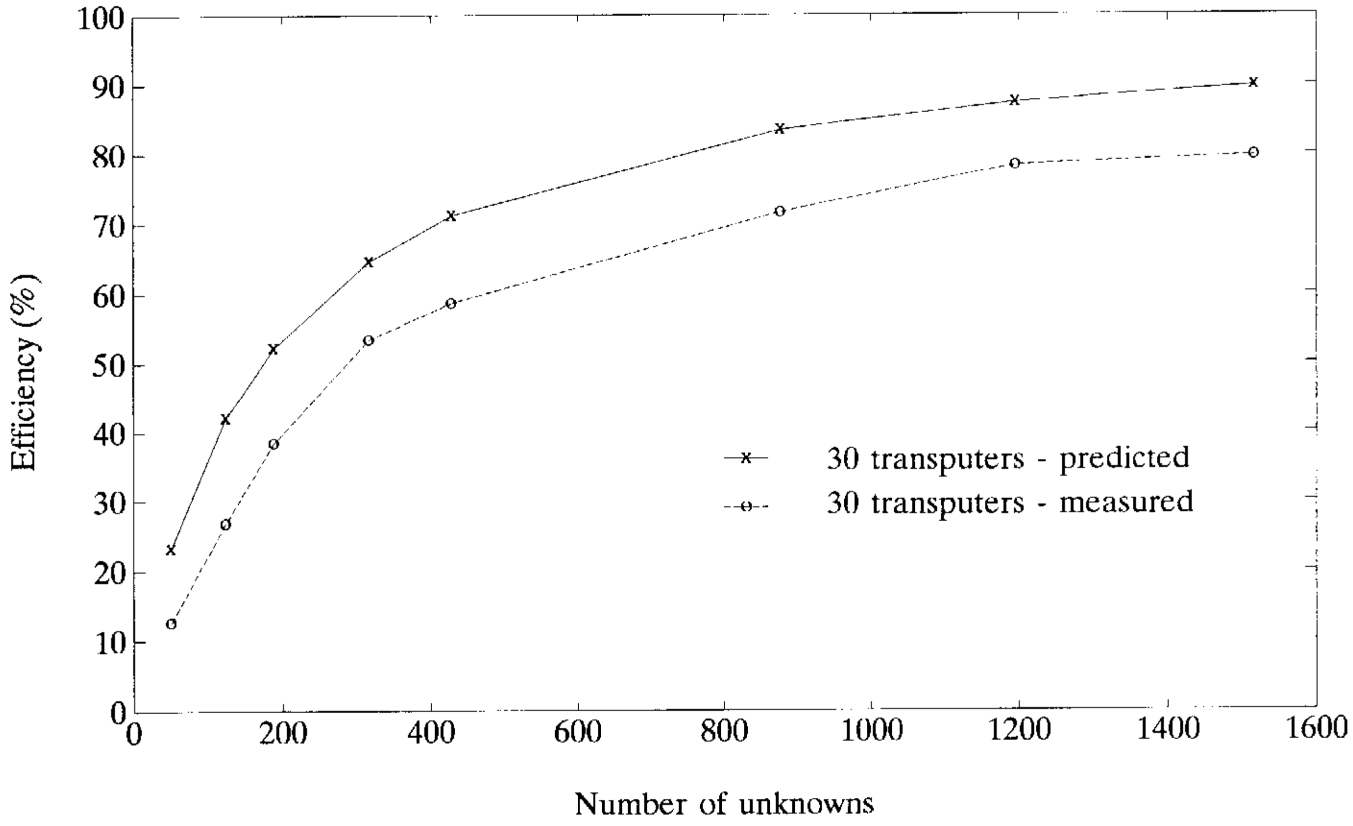

Figure 8: Measured and predicted efficiencies of the double precision parallel conjugate algorithm (30 worker transputers) for the MC^2.

end{Step i}

$a_{i,j}$, $l_{i,j}$ and $u_{i,j}$ represent the i,j-th element of the [A], [L] and [U] matrices respectively; $i,j \epsilon \{0,1,\ldots,M-1\}$; M is the dimension of the matrix. The matrix entries have been numbered from 0 to M-1 for coding convenience: an array a in Occam is numbered a_0, a_1, ...

The algorithm requires special treatment for Step 0 and Step M-1 [19, p.359], and if at any stage $l_{i,i}u_{i,i} = 0$, the algorithm is terminated with an error message to the effect that factorization is impossible. Provided that the matrix is not singular, *pivoting* may be used in such cases — and is advisable whenever the matrix is not known to be well conditioned. Pivoting is a strategy to optimize numerical stability by ensuring that the largest (in some sense) element is on the diagonal. *Maximal column* or *partial* pivoting and *maximal* pivoting [19, p.330–3] are two well-known algorithms; the former involves searching the column below the diagonal, the latter the entire active region, to use the nomenclature of this paper. Bisseling and van de Vorst [20] have shown that partial pivoting may be incorporated into the parallel LU algorithm implemented in this paper without a major effect on the efficiency of the algorithm; the effect on equation (22) is to replace the $\frac{1}{2}$ by $\frac{3}{2}$. However, the coding becomes substantially more complicated than that already required and is not at present incorporated into the author's parallel code.

Following the factorization of $[A]$ into the product of $[L]$ and $[U]$, the unknown left-hand side is solved for in a two-step process; the first step is forward substitution. Consider $[A][x] = [b]$, with A factored as $[A] = [L][U]$. Define $[U][x] = [z]$. Now the system $[L][z] = [b]$ can be solved for using forward substitution, since $[L]$ is lower triangular. Then $[x]$ can be solved using backward substitution from $[U][x] = [z]$ since $[U]$ is upper triangular.

It may be shown that the timing requirements of LU decomposition are approximately $\frac{M^3}{3} + O(M^2) + O(M)$ additions and approximately the same number of multiplications; see [15, p.109]. The constants associated with the lower order terms are small integers, so for all practical purposes, the amount of work required is $\frac{2M^3}{3}$ operations. The factor of 2 comes from the additions *and* multiplications. Similarly, the dominant term in the time for forward substitution is M^2 operations, and the same result also holds for backward substitution.

3.2 Previous Work on Parallel LU Algorithms

This discussion of the serial algorithm now leads to the question of the identification of the parallelism in the algorithm. Compared to the CG algorithm discussed in the preceding section, the parallelism is hardly obvious. Nonetheless, very efficient parallel algorithms can be developed.

Since LU decomposition is such a fundamental algorithm in linear algebra, much work has been done, but very often the work is not applicable to the problem of a full matrix, without any special properties. Brief reviews of parallel LU decomposition may be found in [21, 22]; a rather more recent review paper is [23]. The present paper is based on recent work by van de Vorst and Bisseling [24, 20]; the algorithm used is essentially identical to that of Fox *et al.* [7, Chapter 20], although the very different approaches used to present their algorithms by Bisseling and van de Vorst on the one hand, and Fox *et al.* on the other, make this similarity initially obscure. Van de Vorst's work [24] is particularly difficult to read — cryptic is not an exaggeration for someone unfamiliar with the use of formal methods in computer science — and Fox's work, while far easier to read, is for a banded matrix, hence the difficulties in recognizing the similarity.

3.3 A Parallel LU Algorithm - a Graphical Description

The essence of the parallel algorithm is the following observation. Instead of waiting for Step i to compute $u_{i,j}$ and $l_{j,i}$, as in the serial algorithm described in the previous section, the summations in equations (13) and (14) may be performed as soon as data is available, given sufficient processors ($N = M^2$). As an example, the first summation for each element of row 1 of U may begin as soon as the relevant element of row 0 of U and column 0 of L is available. All the summations required for row 1 may of course be performed in *parallel*, since there is no dependence *within* a row of [U] or a column of [L] (other than on the diagonal element for the final division). Similarly, the first summations for row 2, 3 etc. may also begin as soon as the results of row 0 and column 0 are available. One could of course perform the serial algorithm in exactly the same way, but in the serial case, nothing would be gained, and the algorithm would appear unnecessarily complex. The required summations for row i of U and column i of L are thus computed using a series of *partial sums* performed *in parallel* at each step which *terminates* in Step i. Hence the maximum degree of parallelism in this algorithm is M^2. As will be noted shortly, the algorithm requires at least $2M$ steps to execute. A more detailed explanation may be found in [4], which discusses in a tutorial fashion the mode of thinking required to identify the parallelism inherent in the algorithm.

A parallel algorithm implementing the above is given in pseudo-code in Figure 9. This algorithm assumes the diagonal elements of [L] to be 1. Note that the pseudo-code assumes M^2 processors; if this is not the case, then *clustering* is required. It should be appreciated that efficiently implementing the clustering and communications made the actual Occam code much more complex than the pseudo-code shown. A matrix [X] is used in the pseudo-code; when the algorithm terminates the upper triangular part of [X] is [U], and the lower triangular part of [X] — excluding the diagonal elements, which are 1 by initial choice — is [L]. If the matrix [A] is not needed after factorization, then as the computation of elements of [X] is completed the corresponding elements of [A] may be overwritten.

```
process[s,t] :
begin
  x[s,t] := a[s,t] {initialize matrix}
  k := 0 {initialize global clock}
  while k < n do
    begin
      if k < min(s,t) then
        begin{active}
          par
            receive l[s,k] from process [s,k]
            receive u[k,t] from process [k,t]
          end{par}
          x[s,t] := x[s,t] - l[s,k]*u[k,t]
        end{active}
      else if k = t AND s > t then
        begin{critical}
          receive u[k,k]
          x[s,t] := x[s,t] / u[k,k] {note k=t in this case!}
          send x[s,t] to all  processes[s,q] with q > k
        end{critical}
      else if k = s AND s < or = t then
        begin{pseudo-critical}
          send x[s,t] to all processes[q,t] with q > k
        end{pseudo-critical}
      else if k > min(s,t) then
        SKIP {passive}
      k := k + 1
    end
  end{while}
end. { process[s,t]] }
```

Figure 9: Pseudo-code for the parallel LU algorithm; adapted from [24].

The algorithm can be most easily understood graphically. Figures 10 to 12 show the evolution of the algorithm for a matrix of dimension 3 on a 3 by 3 array of processors, i.e. one processor per element, the upper limit of the parallelism that can be extracted with this algorithm. The • and $*$ represent elements that are critical i.e. in the last stage of computation. (The $*$ represent the row of $[U]$ in the final stage of computation. The choice of the diagonal elements of [L] as 1 means that no computation is required, but the values must still be communicated, hence the distinction and the name pseudo-critical, used in the pseudo-code). The o represents elements that are active, i.e. forming the partial sums. Blank entries represent passive elements, where no work is performed, since the relevant element of L or U has been computed in a previous step. The echelons of completed elements step diagonally downwards in an almost wave-front fashion.

$$\begin{bmatrix} \ast & \ast & \ast \\ \bullet & \circ & \circ \\ \bullet & \circ & \circ \end{bmatrix}$$

Figure 10: Step 0 of LU decomposition

$$\begin{bmatrix} \cdot & \cdot & \cdot \\ \cdot & \ast & \ast \\ \cdot & \bullet & \circ \end{bmatrix}$$

Figure 11: Step 1 of LU decomposition

$$\begin{bmatrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \ast \end{bmatrix}$$

Figure 12: Step 2 of LU decomposition

$\cdot$	passive elements
•	critical elements
$*$	pseudo-critical elements
o	active elements

It is useful to give an example describing how the algorithm given in Figure 9 and illustrated in Figures 10 to 12 proceeds. It is assumed for this discussion that there are M^2 processors, i.e. one processor per matrix element. The initialization of [X] = [A] establishes the first row of [U] — actually before the algorithm has started. (This is because of the choice of diagonal [L] elements).

- On step 0, the first column (column 0) of [L] is computed, and then this column, as well as the first row (row 0) of [U] is sent to all the critical processes so that the partial sums can be

$$\begin{bmatrix} * & * & * \\ \bullet & \searrow & \downarrow \\ \bullet & \rightarrow & \circ \end{bmatrix}$$

Figure 13: Communication in parallel LU algorithm, Step 0. The arrow symbols are defined in the text.

computed. Note that by the end of step $k = 0$, the computations for the second row (row 1) of [U] have been completed.

- On step 1, the second column (column 1) of [L] is computed, and this column, as well as the second column of [U], can be sent to all remaining critical processes so that ongoing partial sums can be computed. By the end of step $k = 1$, the computations for the third row of [U] (row 2) have been completed.
- The algorithm proceeds thus, until $k = M$. (3 in this case).

Note that even given M^2 processors, at each step i, corresponding to one of the Figures 10 to 12, the algorithm given in Figure 9 needs two discrete computational "ticks": firstly, to compute the i-th column of L — in parallel — and secondly, to then update the partial sums on the active processors— also in parallel. Hence with M processors the algorithm will take $2Mt_{calc}$ to terminate, assuming the times for floating point addition, subtraction, multiplication and division to be similar.

Figure 13 shows the communications executed by the algorithm in Step 0. In this figure, the $\downarrow$ indicates communication to all the active elements of the column, and similarly the $\rightarrow$ indicates communication to all the active elements of the row. The $\searrow$ symbol indicates both $\downarrow$ and $\rightarrow$.

Note how fine the grain of parallelism is compared to some other published applications; see for example Cramb *et al.* [9]. They use a processor farm application for antenna array modelling, and decompose their problem by scan angle, producing a parallel system requiring very little data interchange: essentially data initialization, then collection of the finished computations. Such an application is rather easier than those considered in this paper, since far less attention need be given to highly efficient coding.

3.4 Topology, Clustering, Load Balancing and Communications

Given enough processors, the natural topology in the case of LU decomposition is a two-dimensional mesh, reflecting the two-dimensional matrix. The row and column communication shown in Figure 13 can also be implemented very efficiently on such a mesh. However, as with the CG algorithm, the problems of interest are large-grained, where many unknowns must be grouped (or clustered) on each processor. A new problem, not present in the CG algorithm, emerges with the LU algorithm, viz. load balancing. Inspection of Figures 10 to 12 show the problem; the work in each row and column decreases as the algorithm proceeds, resulting in idle processors, producing a lower bound on the efficiency of only approximately 33% [5, Section 5.8]. Hence the topology required for an efficient LU algorithm must not only minimize the communication cost, but also provide a solution to the load balancing problem. The solution to the latter is clearly to interleave rows or columns in some fashion so that the work on each processor remains fairly constant, but this is also clearly linked to the communication cost. Prior to van de Vorst's work, most LU decomposition algorithms clustered the unknowns either by row or by column. However,

$$\left[\begin{array}{ccc|ccc|ccc}
a_{00} & a_{03} & a_{06} & a_{01} & a_{04} & a_{07} & a_{02} & a_{05} & a_{08} \\
a_{30} & a_{33} & a_{36} & a_{31} & a_{34} & a_{37} & a_{32} & a_{35} & a_{38} \\
a_{60} & a_{63} & a_{66} & a_{61} & a_{64} & a_{67} & a_{62} & a_{65} & a_{68} \\
\hline
a_{10} & a_{13} & a_{16} & a_{11} & a_{14} & a_{17} & a_{12} & a_{15} & a_{18} \\
a_{40} & a_{43} & a_{46} & a_{41} & a_{44} & a_{47} & a_{42} & a_{45} & a_{48} \\
a_{70} & a_{73} & a_{76} & a_{71} & a_{74} & a_{77} & a_{72} & a_{75} & a_{78} \\
\hline
a_{20} & a_{23} & a_{26} & a_{21} & a_{24} & a_{27} & a_{22} & a_{25} & a_{28} \\
a_{50} & a_{53} & a_{56} & a_{51} & a_{54} & a_{57} & a_{52} & a_{55} & a_{58} \\
a_{80} & a_{83} & a_{86} & a_{81} & a_{84} & a_{87} & a_{82} & a_{85} & a_{88}
\end{array}\right]$$

Figure 14: Scattered grid distribution; 3 by 3 processor array (mesh). The elements in the upper left corner map onto processor 00 of Figure 2, those in the upper centre onto 01, those in the left centre onto 10 etc.

a better method is to combine these. This double interleaved distribution is also used by Fox *et al.* [7, Section 20-3] for the parallel LU decomposition of a banded LU matrix — the bandedness of the matrix affects the timing analysis, but not the basic algorithm. Fox *et al.* use the term "scattered square decomposition". It would appear that Fox has priority on the double interleaved distribution, but his early work appeared as internal Caltech reports and van de Vorst's and Fox's work appeared in the published literature at much the same time. Van de Vorst's earlier work appears to have been carried out independently of Fox's [24], but Bisseling and van de Vorst later acknowledge the similarity [20].

From the viewpoint of minimizing the communication count, van de Vorst [24] has shown that a square mesh distribution is the optimal $N_1 \times N_2$ grid topology. (Note that a row or column distribution are extreme cases of this general case, in the former case with $N_1 = N$, $N_2 = 1$, and vice versa for the latter). This can be confirmed intuitively: with a column or row distribution, the amount of data to be communicated at each step is $O(M)$ — an entire column (or row) must be communicated — whereas using the grid distribution the amount of data at each step is $O(\frac{M}{\sqrt{N}})$. Furthermore, with the grid distribution, the column and row broadcast pipelines can be run concurrently. This will be explained shortly.

With this grid decomposition required to minimize the communication cost, the load balancing problem may be solved very elegantly using a double-interleaved clustering scheme for data distribution [24, 20], whereby *both* row *and* columns are scattered modulo$\sqrt{N}$ over a square array of $\sqrt{N}$ by $\sqrt{N}$ transputers, with $\sqrt{N} \ll M$. The distribution of a matrix of dimension 9 on a 3 by 3 array using this double interleaved distribution is shown in Figure 14 for the processor mesh shown in Figure 2. The "wave-front" suggested by Figures 10 to 12 now sweeps cyclically through the processor array, each cycle completing $\sqrt{N}$ rows and columns. The algorithm terminates after $M/\sqrt{N}$ cycles. It may be seen by inspection that all the processors remain occupied until the very last cycle of the algorithm. The load-balancing problem is thus alleviated.

In the case where M is not an integral multiple of $\sqrt{N}$, special care is required; the work is divided up as evenly as possible but the processors with one less row and column to work on must be thus explicitly programmed. The method used in the CG code of padding the matrix with rows and columns of zeros is not applicable in this case, since the LU algorithm fails when a zero is encountered on the diagonal.

Formally, the double interleaved distribution is the Cartesian product G of sets $G_i \times H_j$:

$$G \equiv \{G_i \times H_j : 0 \leq i, j < \sqrt{N}\} \qquad (15)$$

with

$$G_i \equiv \{s : s \epsilon V \wedge s \bmod \sqrt{N} = i\} \forall 0 \leq i < \sqrt{N} \tag{16}$$

$$H_j \equiv \{t : t \epsilon V \wedge t \bmod \sqrt{N} = j\} \forall 0 \leq j < \sqrt{N} \tag{17}$$

and

$$\mathbf{V} \equiv \{s : 0 \leq s < M\} \tag{18}$$

The indices i and j refer to processor indices and the indices s and t to matrix element indices. As an example, for the 9 by 9 matrix distributed on the 3 by 3 processor mesh shown in illustrated in Figures 2, $V = \{0, 1, 2 \ldots 8\}$, and G_0, G_1 and G_2 (and H_0, H_1 and H_2) are $\{0, 3, 6\}$, $\{1, 4, 7\}$ and $\{2, 5, 8\}$ respectively. The Cartesian product $G_0 \times H_0$ gives the indices of the 9 elements clustered on processor$_{00}$. The full distribution G is shown in Figure 14.

An upper bound on the load-balancing complexity can be established as follows. The maximum load is carried by processor$_{\sqrt{N-1}\sqrt{N-1}}$ (the processor at the lower right of the processor array). As already discussed, the scattered grid distribution results in a cyclic "sweep" through the processor grid, with $\sqrt{N}$ Steps per cycle and $M/\sqrt{N}$ cycles in total. The amount of work in the last cycle — where there is only one element left to update — is approximately $2(\sqrt{N})$ (the factor 2 comes from the multiplication followed by subtraction, and the $\sqrt{N}$ from the number of Steps in the cycle); on the preceding cycle $2(4\sqrt{N})$; and so on back to the first cycle with $2[M/\sqrt{N}]^2\sqrt{N}$. Summing over all $M/\sqrt{N}$ cycles yields an upper bound of

$$\frac{2}{3}\frac{M^3}{N} + \frac{M^2}{\sqrt{N}} \tag{19}$$

The first term is clearly the parallelized computations; thus the second term is the additional computational overhead caused by the load-balancing term.

The communications use pipelined, concurrent, row and column broadcast. The pipelines are implemented in software; the concept is to overlap the incoming and outgoing vector to further exploit the parallel link operation possible on a transputer. An example is shown in Figure 15 for one of the communication primitives exploiting pipelining. The effect is to speed up the communications by a factor of almost $2\sqrt{N}$, where the factor of 2 derives from the concurrent row and column operation and the $\sqrt{N}$ from the pipelining. Details and more complete pseudo-code may be found in [5, Section 5.10], and also in [4].

An upper bound for the communication count can be derived by considering the *processor* column carrying the heaviest communication load, namely the right-most column. For the first cycle, the amount of data to be communicated is approximately $\frac{M}{\sqrt{N}}$ for each Step in the cycle. For the next cycle, the amount of data is $\frac{M}{\sqrt{N}} - 1$ per Step, and so on. Summing over all the $M/\sqrt{N}$ cycles yields

$$t_{mesh} \leq \{[(\frac{M}{\sqrt{N}})\sqrt{N}] + [(\frac{M}{\sqrt{N}} - 1)\sqrt{N}] + \ldots + [(1)\sqrt{N}]\} t_{comm} \tag{20}$$

There are $\frac{M}{\sqrt{N}}$ square-bracketed terms in total in the above equation (i.e. the number of cycles), which can be re-written as

$$t_{mesh} \leq \{\frac{M^2}{\sqrt{N}} - \sqrt{N} \sum_{k=0}^{\frac{M}{\sqrt{N}}-1} k\} t_{comm} \tag{21}$$

and thus

$$t_{mesh} \leq \frac{1}{2}\frac{M^2}{\sqrt{N}} + O(M) \tag{22}$$

```
procedure broadcast_column_to_right(length)
begin
  {initialize pipeline}
  {note: length of vector passed as argument}
  receive vector[1] from left processor
  repeat for i = 2 to length
    par{run pipeline}
      receive vector[i] from left processor
      send vector[i-1] to right processor
    end{par}
  end{repeat}
  {flush pipeline}
  send vector[length] to right processor
end{procedure broadcast_column_to_right}
```

Figure 15: Pseudo-code for rightwards pipelined column broadcast procedure. This runs in parallel with similar leftwards column broadcast and upwards and downwards row broadcast procedures.

Bisseling and van de Vorst's result [20, equation (3.19)] has an identical dependence on $\frac{M^2}{\sqrt{N}}$, once the necessary change of notation is made.

A theoretical model for the efficiency may now be derived. The serial time, using a conversion factor from complex to real flops of 4 as before, is $(\frac{8}{3}M^3)t_{calc}$; the parallel time is the sum of the computation count, equation (19), and the communication count, equation (22). Summing the last three, using equation (1) and simplifying yields

$$\epsilon = \frac{1}{1 + n^{-1}(\frac{3}{2} + \frac{3\beta}{16\gamma})} \tag{23}$$

where $n = M/\sqrt{N}$ is the *grain* of the problem, i.e. the number of unknowns per processor, and β has the previously defined meaning.

It is instructive to compare this result with that for the CG solver, re-written using the same notation:

$$\epsilon = \frac{1}{1 + n^{-1}\sqrt{N}(2.75 + 0.125d + \frac{\log_2 N\beta}{8})} \tag{24}$$

Note that the terms in n^{-1} in the denominator of the respective equations have similar constant multipliers, but in addition the CG equation has a $\sqrt{N}$ and also a $\log_2 N$ term. Hence it can be expected that for similar n that the LU algorithm is more efficient, a result that is confirmed experimentally. *This indicates that a parallel LU algorithm based on a mesh topology scales better than a parallel CG based on a binary tree* — the mesh and binary tree being considered as the "natural" topologies for the CG and LU algorithms respectively, for the reasons already discussed in this paper. To summarize: the CG algorithm scales with the reciprocal of the number of *rows* per processor, whereas the LU algorithm scales with the reciprocal of the square root of the number of *unknowns* per processor, and the latter is the smaller multiplier. This is a most interesting result, considering how initially unsuitable for parallelism the LU algorithm appeared, and is confirmed by the results in Section 3.6.

```
process[s] :
begin
  z[s] := b[s]; k = 0 {initialize}
  while k < n do
    begin
      if k < s then
        receive z[k] from process [k]
        z[s] := z[s] - L[s,k] z[k]
      else if k = s then
        z[s] := z[s] / L[s,s]
        send z[s] to all processes q with q > k
      else if k > s then
        SKIP
      k := k+1
    end
end. { process[s]] }
```

Figure 16: Forward substitution pseudo-code; solve [L][z]=[b].

3.5 Parallel Forward and Backward Substitution

Following the factorization of $[A]$ into the product of $[L]$ and $[U]$, the unknown left hand side is solved for using the two-step forward and backward substitution processes already discussed. A parallel version of the forward and backward substitution algorithms is also necessary, not because of the computation time, which is $O(M^2)$, but because it is most undesirable to communicate all the elements of the $[L]$ and $[U]$ matrices back to a master processor, since the master must then have enough memory to store the entire matrix and the communication procedure takes time. The former is the more serious problem for a typical MIMD array with local memory; sufficient memory is not available on any one node (processor plus memory) to store the entire matrix. Suitable parallel substitution algorithms have been derived by the author; pseudo-code for forward substitution is given in Figure 16. The modifications for backward substitution are simple; the algorithm may be found in [5, Chapter 6]. Subsequent to publication of the author's own research [25], van de Vorst and Bisseling published an algorithm for parallel forward and backward substitution.

The substitution algorithms operate on only one column of the processing array at a time, and the latest version of the relevant vector ($[z]$ or $[x]$) is passed from column to column as the algorithm proceeds. This is far from the most efficient parallel substitution algorithm possible, since only $\sqrt{N}$ processors are active concurrently, but has the major advantage of using the same scattered grid distribution as the parallel LU algorithm.

3.6 Timing Results

The parallel LU and substitution algorithms described in this section have been implemented by the author in Occam 2 for a transputer array. Details of the implementation are discussed in [5, Section 5.11]. Preliminary results were presented as [25]. Figure 17 shows efficiencies for a number of different processor array sizes as a function of matrix dimension. The timing results are for single precision runs. The matrix was generated using a simple thin-wire moment method scheme using sinusoidal basis functions and collocation, using results from [26, Section 7.5] for the field radiated by a sinusoidal current. This moment method code was also written in Occam 2. The largest problem solved had 1500 unknowns, and used 25 transputers. The LU solver took about

15 minutes to run, which corresponds to a computation speed of 9.6 MFLOP/s, and an efficiency of close on 90%. The matrix was also generated in parallel and the efficiency of the entire code is very similar to that of the LU part, which is of course the most computationally expensive part. The forward and backward substitution algorithms have also been implemented and despite having rather poor efficiency (as expected), the overall impact on the code is negligible due to the $O(M^2)$ computational cost of the substitution algorithm.

Figure 18 shows theoretical predictions, which can be seen to be somewhat optimistic, although the general trend is correctly predicted. Reasons similar to those given in Section 2.4 may be advanced for the differences; note that rather finer grain of communication in the LU algorithm is more difficult to model accurately than the communication in the CG algorithm. Recent work by the author indicated that the pipelines have a subtle problem in that the effect of the *set-up* time — the time to initiate a communication on a link — is *not* negligible when elements are being communicated individually; it is around $6.5\mu s$, approximately equal to t_{comm} in single precision. The effect of this is to double β in this case, and this has been incorporated in Figure 18; however, the theoretical results are still some way off the measured results. To permit comparison of the parallel LU and CG algorithms, measurements for a parallel CG algorithm are also shown in Figure 18 for 14 transputers. (The binary tree and mesh topologies cannot use exactly the same number of processors; a tree of 14 and a mesh of 16 is a fair comparison). The CG results were measured with a single precision version of PARNEC. (Note that the results shown previously for the CG solver are for the double precision version of PARNEC.)

Bisseling and Van de Vorst show similar measured results in [20]; the numerical values for efficiency shown in Figure 17 are not directly comparable with their results, which are presumably for real valued matrices, although the latter is not explicitly stated in their paper. The form of the curves is very similar.

4 Scalar efficiencies of the LU and CG algorithms

Scalar efficiency [9] deals with the actual run-times of the algorithms when run on the same computer — since the efficiencies of the algorithms discussed in this paper are comparable, it is also very important for these parallel algorithms. It has generally been assumed that using an iterative solver reduces the amount of computation from $O(M^3)$ for the LU solver to $n_{iter}O(M^2)$ for the CG solver, where n_{iter} is the number of iterations required for convergence. The motivation for using parallel iterative solvers for full matrix problems was the expectation that the convergence would be sufficiently rapid for the CG algorithm to terminate in a small number iterations, making the run-time considerably less than the corresponding LU factorization. Iterative methods are widely and successfully used in methods resulting in sparse matrices such as the Finite Element method [27, Chapter 10].

Unfortunately, for arbitrary moment method problems, the number of iterations appears to be a quite substantial fraction of the number of segments, for structures discretized according to some nominal segment length rule, for example $\lambda/10$. For problems that are over-discretized, the number of iterations appears to be a function of the problem, and increases only weakly with the discretization, once the structure is satisfactorily discretized. See for example [18]. The reason for this is probably that the extra eigenvalues introduced by the over-discretization are not very significant; see [28]. Unfortunately, it is problems that are just satisfactorily discretized that are frequently of the greatest interest to electromagnetic modellers.

Hence, for the important case of structures just satisfactorily discretized, the computational

[9] The term was suggested by a reviewer, and describes the issue very succinctly.

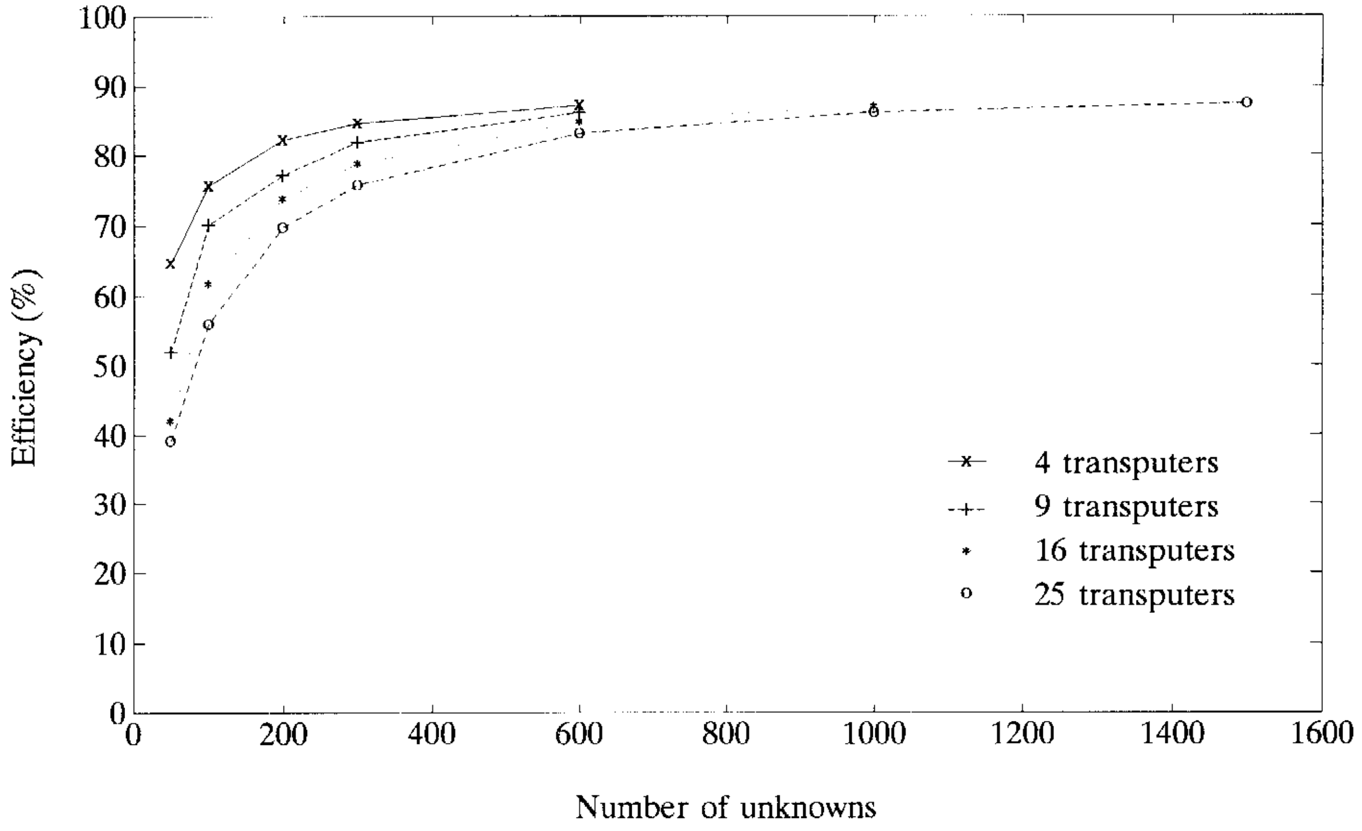

Figure 17: Measured efficiencies of the single precision parallel LU solver.

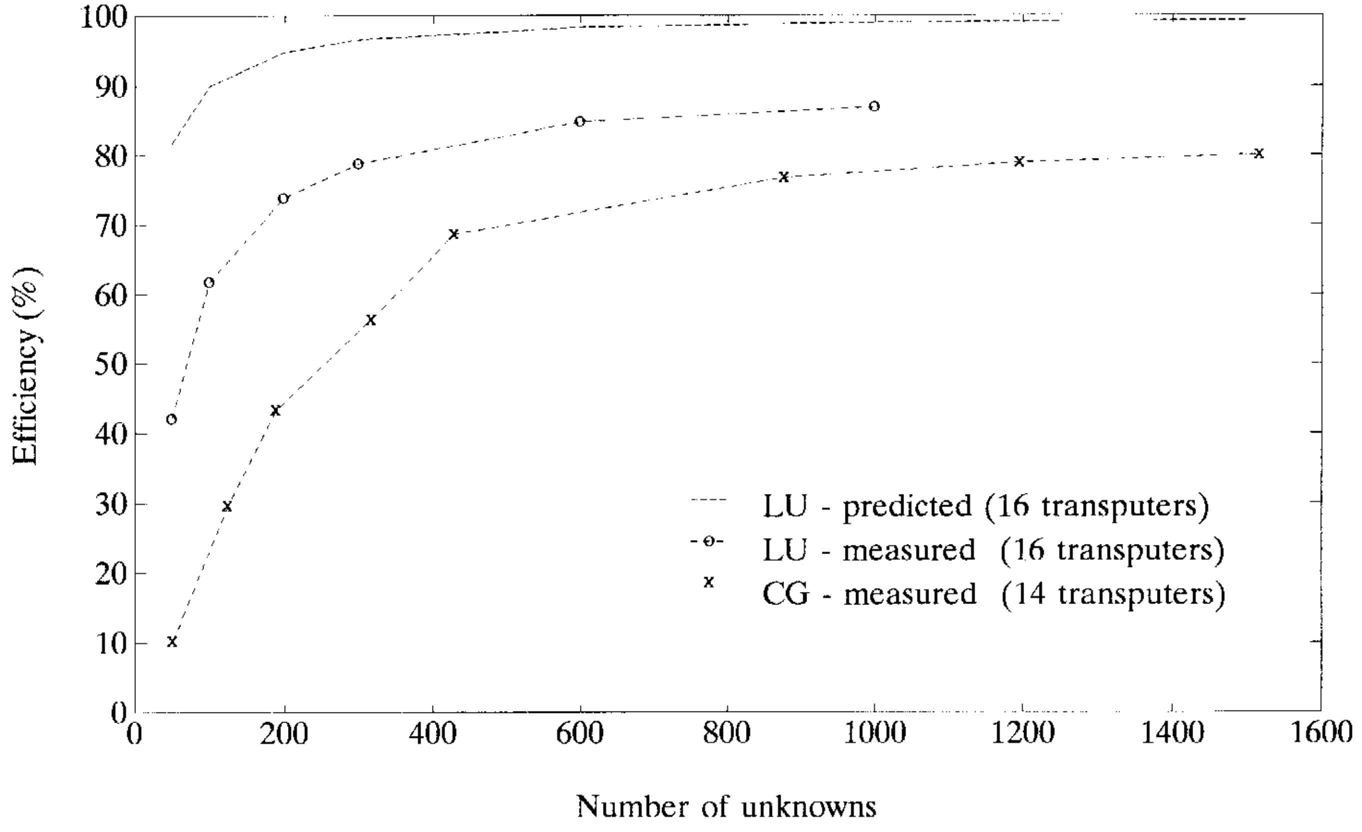

Figure 18: Comparative efficiencies for the single precision LU and CG solvers for similar numbers of transputers.

dependence of the CG algorithm would also appear to be essentially $O(M^3)$. Since the efficiencies of both parallel methods are comparable, the serial break-even point can be used, namely where the number of iterations is 1/6 of the matrix dimension. However, even in the largest case investigated to date by the author, using about 2 000 segments, this fraction was closer to 1/4, and was even larger for smaller problems; see Table 7. Peterson and Mittra reported similar results several years ago, for smaller systems with at most a few hundred unknowns [29]. The present author used a normalized error criterion of 10^{-2}, giving an error of around 1%. So, unless one is satisfied with a larger error, the LU method would have been slightly, to considerably, faster for all the problems investigated. The rate of convergence is highly dependent on the problem; for some other work recently performed on relatively large systems (1 000 to 2 000 unknowns), the rate of convergence was much poorer than that mentioned above.

With a multiple right-hand side problem, such as a typical radar cross section problem, the superiority of the LU method has long been acknowledged. The work of Smith *et al.* [30] on using the CG method to solve multiple right-hand sides, by re-using some of the data generated for previous right-hand sides, showed that although significant time savings compared to the standard CG method were possible, for many right-hand sides the LU method remained the better approach. However, a new technique recently proposed by Kastner and Herscovici [31] shows promising results for a multiple right-hand side CG formulation.

5 Parallelizing the matrix fill

A number of researchers have reported that the time required for matrix fill, although an $O(M^2)$ operation, can still dominate moment method codes for large numbers of unknowns [11]. Certainly with a patch code, especially if using the Galerkin formulation, this could be a serious problem. With thin-wire collocation codes such as NEC2, the matrix solve fairly rapidly dominates the matrix fill; an example is shown in Table 7. This is for a CG solver; the number of iterations is also listed. The "break-even" number of iterations where the CG run-time equals that of LU decomposition is $M/6$. Using the NEC2 formulation and the CG solver row-block decomposition, the problem of parallel matrix generation was easily solved — one simply decomposes by match point, in precisely the same fashion that the NEC2 out-of-core solver functions. Work is at present in progress on incorporating the LU solver into the parallel NEC2 code.

6 Conclusions

6.1 General

This paper has presented parallel algorithms for the two algorithms most frequently used in computational electromagnetics for the solution of systems of linear equations. The basic algorithms have been reviewed; parallel algorithms have been presented — both informally and formally in pseudo-code, analyzed, and results obtained with an implementation of the algorithms on a specific parallel computer reported. The experimental results for the CG and LU solvers have been compared both to the theoretical predictions and with each other for similar numbers of processors, demonstrating both theoretically and practically that the parallel LU algorithm presented is more efficient than the parallel CG algorithm shown. The *scalar* efficiency of the LU algorithm is also better since the run-time of the CG method is highly dependent on the rate of convergence of the CG algorithm, and it has been found that the rate of convergence of the CG algorithm for practical problems is not sufficient for the CG algorithm to out-perform the LU algorithm.

Number of segments M	t_{solve}/t_{fill}	Number of iterations
50	1.0	14
124	2.2	75
188	2.7	134
316	2.4	131
428	7.2	372
876	9.1	405
1196	10.4	409
1516	11.9	414
1996	21.1	543

Table 7: Ratio of the matrix fill to solve times for a particular simulation, viz. a cone-cylinder with four monopoles [5, Section 6.7]. 30 worker transputers were used. All data except for the last entry are for double precision: the 1996 segment data was generated using single precision.

6.2 Scaling behaviour and grain size

A very important result has been demonstrated, both theoretically and experimentally, namely the *scaling* properties of the algorithms; larger problems can be solved in an approximately constant time by increasing the number of processors. The scaling property of the parallel LU algorithm considered in this paper has been shown to better than that of the parallel CG algorithm discussed, although both have quite satisfactory scaling properties. It might be thought that it is self-evident that as the grain size (the number of unknowns per processor, n^2 as used in this paper), increases, so the efficiency will increase — however, this is only a property of an algorithm where the computation cost as a function of the grain size increases faster than the communication cost, and is by no means a general property of all parallel algorithms.

The dependence of the efficiency on the grain has some implications for massively parallel systems that should be considered explicitly. If 50% efficiency is considered acceptable, then a grain size of several hundred is required for acceptable efficiency for the LU algorithm; or put slightly differently, a sub-matrix per processor of dimension twenty or so. An important theoretical result in this paper is that for a given efficiency, this grain size remains constant for the LU algorithm and is only weakly dependent on the number of processors for the CG algorithm (the dependence is approximately $\sqrt{N}$); actual timing results confirm this (Figure 7). Note that since the efficiency is a function of the β, the communication to computation ratio, this break-even point will also be a function of this ratio. For the transputer technology used, this ratio produced very acceptable efficiencies on problems of practical interest: it is, of course, a function of the processor technology, and the user must accept it as a given for a particular processor. For arrays with hundreds of processors, where the algorithms remain relatively coarse-grained, the results in this paper can be extrapolated to show that what are really the classic serial algorithms (albeit in parallel form) can still give very acceptable efficiencies. It should be stated, however, that these results may not apply to truly massively parallel systems, with perhaps tens or hundreds of thousands of processors. The fundamental philosophical issue is that of global interaction (viz. integral equation methods) versus local interaction (viz. differential equation methods) and it is likely that the latter methods with their highly local interaction requirements may be far better candidates for massively parallel computers.

6.3 Workstations or transputers?

This section compares T800 based arrays with workstations available at the time of writing (1992) and will inevitably date. As was clearly indicated in the Introduction, the aim of this paper was not to blindly promote transputer arrays; a sober analysis of competing computer technologies is necessary. The present transputer technology (the T800) dates to around the mid-nineteen eighties, and at the time of writing, a contemporary high performance RISC workstation would probably be a better investment; the maximum through-put of a 64 transputer array is about 33 MFLOP/s (in single precision, with a 100% efficiency of the parallel algorithm) as soon as one uses the off-chip RAM, as is typically the case. The author has only been able to use about half this array (25 processors), and obtained 9.6 MFLOP/s. The author has benchmarked the HP720 RISC workstation at close to 20 MFLOP/s on LU decomposition (also in single precision).

Note that this is in not a very fair comparison, since it involves the comparison of computing technologies separated by five to six years; the balance will change back dramatically in favour of transputer arrays when the T9000 is shipped from Inmos [4, 11], so the time invested in developing the parallel algorithms described here is time well invested for future arrays; as serial processors (and the related processors such as super-scalar architectures) increase in speed, so do the individual elements of processor arrays. Cwik and Patterson have reported the accurate solution of what can only be described as *massive* moment method problems with 30 000 unknowns on a 512 node i860 array [32].

6.4 Issues still to be addressed

An issue that has not been addressed in this paper is the stability and accuracy of electromagnetically large problems discretized using the moment method. The stability of the LU method, applied to computational electromagnetic problems, has been studied by the author using a thin-wire problem and results obtained indicate that in all except the most exceptional circumstances, involving serious violation of the basic "thin-wire" assumption, the solutions obtained using the LU solver are stable. The availability of a parallel version of NEC2 has permitted the investigation of the accuracy of the moment method for large problems. This was done by using a physically symmetrical problem; first solved with, and then without, exploiting the symmetry. Using symmetry reduces the number of unknowns by the degree of symmetry, thus requiring the solution of a much smaller system of equations. This method has been used to demonstrate the accuracy of NEC2 for problems with up to 2 000 unknowns [5, Chapter 6]. Some preliminary details are to be published in [4].

The CG algorithm was the first major parallel code developed by the author and is not optimal in a number of respects: if pipelining, as exploited in the LU algorithm, were to be exploited in the CG algorithm for the broadcast and gather operations, improvements should be anticipated — this has not been implemented, however. Further, the unparallelized vector operations (addition, subtraction etc.), responsible for the 2.75 factor, could probably also be reduced by parallelizing the vector operations. These amount to "fine-tuning" the existing binary tree algorithm. (One should also bear in mind that given a processor with four communication links, a ternary tree would be more efficient than a binary tree — as mentioned in [3]). However, in the light of the predicted and measured performance of the parallel LU algorithm, an interesting question that arises is whether implementing the parallel CG algorithm on a mesh would result in communication performance similar to that of the parallel LU algorithm. This is a topic for future research. At present, the whole question is possibly more of academic than practical interest, since the existing parallel CG code, while admittedly not optimal, is still very efficient for the problems of interest on presently available arrays. However, rather larger MIMD arrays involving possibly thousands of processors may require parallel CG algorithms with better scaling properties.

6.5 General Conclusions

While the use of more powerful computers with existing algorithms is still ultimately limited by the third power law (see Section 4), for many problems a relatively modest increase in problem size closes the gap between moment method analyses and asymptotic analyses such as the Geometric Theory of Diffraction. The importance of the algorithms discussed in this paper is the good scaling properties that permit the efficient exploitation of large — but possibly not massive – processor arrays for large problems.

Acknowledgements

The work published here is based on the sections of the author's Ph.D. dissertation, awarded by the University of Stellenbosch. The author would like to acknowledge the contributions of his promoters (Prof. J.H. Cloete, University of Stellenbosch, and Prof. D.A.McNamara, University of Pretoria) to this work. Prof. J.J. du Plessis was instrumental in obtaining the transputer array, and his efforts in this regard are much appreciated by the parallel processing community at this University. The author also would like to thank Prof. Cloete and Messrs. P.Steyn and D.B. Le Roux for their careful proofreading of this paper, and the six anonymous reviewers for raising a number of important points which led to substantial revision of this paper.

References

[1] E. K. Miller, "A selective survey of computational electromagnetics," *IEEE Trans. Antennas Propagat.*, vol. 36, pp. 1281–1305, September 1988.

[2] G. Zorpette (editor), "Special issue: supercomputers," *IEEE Spectrum*, vol. 29, pp. 26–76, September 1992.

[3] D. B. Davidson, "A parallel processing tutorial," *IEEE Antennas Propagat. Magazine*, vol. 32, pp. 6–19, April 1990.

[4] D. B. Davidson, "Parallel processing revisited: a second tutorial," *IEEE Antennas Propagat. Magazine*, October 1992. To appear.

[5] D. B. Davidson, *Parallel Algorithms for Electromagnetic Moment Method Formulations.* PhD thesis, Dept. Electrical & Electronic Engineering, University of Stellenbosch, 1991.

[6] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, vol. C-21, pp. 948–60, September 1972.

[7] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker, *Solving Problems on Concurrent Processors, Vol I: General Techniques and Regular Problems.* Englewood Cliffs, New Jersey: Prentice-Hall, 1988.

[8] C. Hafner, "Parallel computation of electromagnetic fields on transputers," *IEEE Antennas Propagat. Society Newsletter*, vol. 31, pp. 6–12, October 1989.

[9] I. Cramb, D. H. Schaubert, R. Beton, J. Kingdon, and C. Upstill, "Antenna array modelling by parallel processor farms," *Applied Computational Electromagnetics Society Journal*, vol. 6, pp. 143-151, Winter 1991.

[10] D. C. Nitch and A. P. C. Fourie, "Adapting the Numerical Electromagnetics Code to run in parallel on a network of transputers," *Applied Computational Electromagnetics Society Journal*, vol. 5, pp. 76–86, Winter 1990.

[11] L. C. Russel and J. W. Rockway, "Application of parallel processing to a surface patch/wire junction EFIE code," *Applied Computational Electromagnetics Society Journal*, vol. 7, pp. 48–66, Summer 1992.

[12] J. J. H. Wang, *Generalized Moment Methods in Electromagnetics.* New York: John Wiley and Sons, 1991.

[13] S. L. Ray and A. F. Peterson, "Error and convergence in numerical implementations of the conjugate gradient method," *IEEE Trans. Antennas Propagat.*, vol. 36, pp. 1824–1827, December 1988.

[14] G. H. Golub and D. P. O'Leary, "Some history of the conjugate gradient and Lanczos algorithms: 1948–1976," *SIAM Review*, vol. 31, pp. 50-102, March 1989.

[15] A. Jennings, *Matrix Computation for Engineers and Scientists.* Chichester: John Wiley and Sons, 1985.

[16] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst, *Solving Linear Systems on Vector and Shared Memory Computers.* Philadelphia: SIAM, 1991.

[17] C. F. Smith, A. F. Peterson, and R. Mittra, "The biconjugate gradient method for electromagnetic scattering," *IEEE Trans. Antennas Propagat.*, vol. 38, pp. 938–940, June 1990.

[18] D. B. Davidson and D. A. McNamara, "Comparisons of the application of various conjugate-gradient algorithms to electromagnetic radiation from conducting bodies of revolution," *Microwave and Optical Technology Letters*, vol. 1, pp. 243–246, September 1988.

[19] R. L. Burden and J. D. Faires, *Numerical Analysis.* Boston: Prindle, Weber and Schmidt, 4th ed., 1989.

[20] R. H. Bisseling and J. G. G. van de Vorst, "Parallel LU decomposition on a transputer network," in *Parallel Computing 1988: Shell Conference Proceedings, (Lecture Notes in Computer Science series)*, (G. A. van Zee and J. G. G. van de Vorst, eds.), (Amsterdam), pp. 61-77, Springer-Verlag, 1989.

[21] D. Heller, "A survey of parallel linear algorithms in numerical linear algebra," *SIAM Review*, vol. 20, pp. 740–777, October 1978.

[22] G. A. Geist, M. T. Heath, and E. Ng, "Parallel algorithms for matrix computations," in *The Characteristics of Parallel Algorithms*, (L. H. Jamieson, D. B. Gannon, and R. J. Douglass, eds.), Cambridge,MA: MIT Press, 1987.

[23] K. A. Gallivan, R. J. Plemmons, and A. H. Sameh, "Parallel algorithms for dense linear algebra computations," *SIAM Review*, vol. 32, pp. 54 135, March 1990.

[24] J. G. G. van de Vorst, "The formal development of a parallel program performing LU-decomposition," *Acta Informatica*, vol. 26, pp. 1–17, 1988.

[25] D. B. Davidson, "Parallel LU decomposition on a transputer array," in *Symposium digest, Vol III, of the 1991 IEEE AP-S International Symposium*, pp. 1500–1503, June 1991. Held in London, Ontario, Canada.

[26] W. L. Stutzman and G. A. Thiele, *Antenna Theory and Design.* New York: John Wiley and Sons, 1981.

[27] P. P. Silvester and R. L. Ferrari, *Finite Elements for Electrical Engineers.* Cambridge: Cambridge University Press, 2nd ed., 1990.

[28] A. F. Peterson, C. F. Smith, and R. Mittra, "Eigenvalues of the moment-method matrix and their effect on the convergence of the conjugate gradient algorithm," *IEEE Trans. Antennas Propagat.*, vol. 36, pp. 1177–1179, August 1988.

[29] A. F. Peterson and R. Mittra, "Convergence of the conjugate gradient method when applied to matrix equations representing electromagnetic scattering problems," *IEEE Trans. Antennas Propagat.*, vol. 34, pp. 1444–1454, December 1986.

[30] C. F. Smith, A. F. Peterson, and R. Mittra, "A conjugate gradient algorithm for the treatment of multiple incident electromagnetic fields," *IEEE Trans. Antennas Propagat.*, vol. 37, pp. 1490–1493, November 1989.

[31] R. Kastner and N. Herscovici, "A concise conjugate gradient computation of plate problems with many excitations," *IEEE Trans. Antennas Propagat.*, vol. 38, pp. 1239–1243, August 1990.

[32] T. Cwik and J. Patterson., "The solution and numerical accuracy of large MoM problems," in *Symposium digest of the 1992 URSI Radio Science Meeting*, p. 335, July 1992. Held in Chicago, Illinois, USA.