Parallel Algorithms for Electromagnetic Moment Method Formulations

David Bruce Davidson

Dissertation accepted for the Degree of
Doctor of Philosophy in Engineering at
the University of Stellenbosch

Promoters:
Prof. J.H.Cloete, University of Stellenbosch
Prof. D.A.McNamara, University of Pretoria

November 1991

Declaration

I the undersigned hereby declare that the work contained in this dissertation is my own original work and has not been previously in its entirety or in part been submitted at any university for a degree.

Signature

Date: 11 November 1991

Summary

This dissertation investigates the moment method solution of electromagnetic radiation and scattering problems using parallel computers. In particular, electromagnetically large problems with arbitrary geometries are considered. Such problems require a large number of unknowns to obtain adequate approximate solutions, and make great computational demands. This dissertation considers in detail the efficient exploitation of the potential offered by parallel computers for solving such problems, and in particular the class of local memory Multiple Instruction, Multiple Data systems.

A brief history of parallel computing is presented. Methods for quantifying the efficiency of parallel algorithms are reviewed. The use of pseudo-code for documenting algorithms is discussed and a pseudo-code notation is defined that is used in later chapters.

A new parallel conjugate gradient algorithm, suitable for the solution of general systems of linear equations with complex values, is presented. A method is described to handle efficiently the Hermitian transpose of the matrix required by the algorithm. Careful attention is paid to the theoretical analysis of the algorithm's parallel properties (in particular, speed-up and efficiency). Pseudo-code is presented for the algorithms. Timing results for a moment method code, running on a transputer array and using this conjugate gradient solver, are presented and compared to the theoretical predictions.

A parallel LU algorithm is described and documented in pseudo-code. A new graphical description of the algorithm is presented that simplifies the identification of the parallelism and the analysis of the algorithm. The use of formal methods for extracting parallelism via the use of invariants is presented and new examples given. The speed-up and efficiency of the algorithm are analyzed theoretically, using new methods that are simpler than those described in the literature. Techniques for optimizing the efficiency of parallel algorithms are introduced, and illustrated with pseudo-code. New parallel forward and backward substitution algorithms using the data distribution required for the parallel LU algorithm are described, and documented with pseudo-code. Results obtained with a Occam 2 moment method code running on a transputer array using these parallel LU solver and substitution algorithms are presented and compared with the theoretical predictions.

PARNEC, a new Occam 2 implementation of the thin-wire core of NEC2, is discussed. The basic theory of NEC2 is reviewed. Problems with early attempts at combining Occam and FORTRAN are reported. Methodologies for re-coding an old code written in an unstructured language in a modern structured language are discussed. Methods of parallelizing the matrix generation are discussed. The accuracy of large moment method formulations is investigated, as is the effect of machine precision on the solutions. The use of the biconjugate gradient method to accelerate convergence is briefly considered and rejected. The increased size of problem that can be handled by PARNEC, running on a transputer array, is demonstrated.

Conclusions are drawn regarding the contributions of this dissertation to the development of efficient parallel electromagnetic moment method algorithms.

Opsomming

Hierdie proefskrif ondersoek die momentmetode oplossing van elektromagnetiese straling- en strooiingprobleme d.m.v. multiverwerkers. In besonder, elektromagneties groot probleme met arbitrêre geometrieë word beskou. Sulke probleme vereis 'n groot aantal onbekendes om 'n voldoende benaderde oplossing te kry, en stel groot berekenings vereistes. Hierdie proefskrif beskou in detail die doeltreffende benutting van die potensiaal wat multiverwerkers vir sulke problem bied, in besonder die klas van lokale geheue Veelvoudige Instruksie, Veelvoudige Data stelsels.

'n Kort geskiedenis van multiverwerkers word gegee. Metodes vir die kwantifisering van die effektiwiteit van multiverwerkers word hersien. Die gebruik van pseudokode vir die dokumentering van algoritmes word bespreek en 'n pseudokode notasie word gedefinieer wat gebruik word in latere hoofstukke.

'n Nuwe parallelle toegevoegde helling-algoritme wat geskik is vir die oplossing van algemene stelsels van lineêre vergelykings word aangebied. 'n Metode word beskryf om op 'n doeltreffende wyse die Hermitiese transponent van die matriks, wat deur die algoritme benodig word, te hanteer. Sorgvuldige aandag word aan die teoretiese analise van die paralleleienskappe van die algoritme gegee (in die besonder, versnelling en doeltreffendheid). Pseudokode word aangebied vir die algoritmes. Resultate vir die looptyd van 'n momentmetode program, wat op 'n transputerskikking loop, word gegee en vergelyk met die teoretiese voorspellings.

'n Parallelle LU algoritme word beskryf en gedokumenteer in pseudokode. 'n Nuwe grafiese beskrywing van die algoritme, wat die identifikasie van parallelisme en die analise van die algoritme vergemaklik, word gegee. Die gebruik van formele metodes vir die onttrekking van parallelisme d.m.v. invariante word getoon en nuwe voorbeelde word gegee. Die versnelling en doeltreffendheid van die algoritme word teoreties geanaliseer, d.m.v. nuwe metodes wat eenvoudiger is as dié wat in die literatuur beskryf word. Tegnieke vir die optimering van die doeltreffendheid van parallelle algoritmes word ingevoer, en geïllustreer met pseudokode. Nuwe parallelle voor- en truwaarts-substitusie algoritmes wat die data verspreiding van die parallelle LU algoritme gebruik word beskryf, en gedokumenteer met pseudokode. Resultate verkry met 'n Occam 2 momentmetode program wat op 'n transputerskikking loop en die parallelle LU en substitusie algoritmes gebruik, word gegee en vergelyk met teoretiese voorspellings.

PARNEC, 'n nuwe Occam 2 implementering van die dun-draad kern van NEC2, word bespreek. Die basiese teorie van NEC2 word opgesom. Verslag word gedoen oor probleme met vroeë pogings om Occam en FORTRAN te kombineer. Metodes om 'n ou program, geskryf in 'n ongestruktureerde taal, in 'n moderne gestruktureerde taal te herskryf word bespreek. Metodes om die matriksopwekking te paralleliseer word bespreek. Die akkuraatheid van groot momentmetode formulerings word ondersoek, asook die effek van masjienpresisie op die oplossings. Die gebruik van die dubbeltoegevoegde helling-metode om konvergensie te versnel word kortliks beskou en verwerp. Die vergrote probleemgrootte, wat met PARNEC op 'n transputerskikking uitgevoer kan word, word gedemonstreer.

Gevolgtrekkings word gemaak rakende die bydraes van hierdie proefskrif tot die ontwikkeling van doeltreffende parallelle elektromagnetiese moment-metode algoritmes.

Abstract

This dissertation investigates the moment method solution of electromagnetic radiation and scattering problems using parallel computers. In particular, electromagnetically large problems with arbitrary geometries are considered. Such problems require a large number of unknowns to obtain adequate approximate solutions. These problems arise due to the requirement for solving radiation and scattering problems in the "gap" region between the moment method and the geometrical theory of diffraction. Existing techniques for extending the frequency coverage of these methods are reviewed, including the exploitation of symmetry, the impedance matrix localization method, hybrid methods and iterative methods. The suitability of these methods for addressing the "gap" problem is discussed. Then the possibility of exploiting the capabilities of parallel computers to address this "gap" problem by providing the necessary computational capabilities for conventional moment method formulations is introduced. It is this topic that this dissertation considers in detail, in particular the efficient exploitation of the potential offered by parallel computers, in particular the class of Multiple Instruction, Multiple Data systems.

A brief history of parallel computing is presented. Methods for quantifying the efficiency of parallel algorithms are reviewed. The use of pseudo-code for documenting algorithms is discussed and a pseudo-code notation is defined that is used in later chapters.

A new parallel conjugate gradient algorithm, suitable for the solution of general systems of linear equations with complex values, is presented. A method is described to handle efficiently the Hermitian transpose of the matrix required by the algorithm. Careful attention is paid to the theoretical analysis of the algorithm's parallel properties (in particular, speed-up and efficiency). Pseudo-code is presented for the algorithms. Timing results for a moment method code, running on a transputer array and using this conjugate gradient solver, are presented and compared to the theoretical predictions.

A parallel LU algorithm is described and documented in pseudo-code. The parallelism in this algorithm is not obvious and a new graphical description of the algorithm is presented that simplifies the identification of the parallelism and the analysis of the algorithm. The use of formal methods for extracting parallelism via the use of invariants is presented; new examples are given of the application of the methods. The use of formal methods for the analysis of the LU algorithm is shown in considerably greater detail than is available in the literature. Pseudo-code for the parallel LU algorithm is given. The speed-up and efficiency of the algorithm are analyzed theoretically, using new methods that are simpler than those described in the literature. Techniques for optimizing the efficiency of parallel algorithms are introduced, and illustrated with pseudo-code. New parallel forward and backward substitution algorithms using the data distribution required for the parallel LU algorithm are described. Pseudo-code for these algorithms is given. Results obtained with an Occam 2 moment method code for straight, thin wires using these parallel LU solver and substitution algorithms are presented and compared with the theoretical predictions.

PARNEC, a new Occam 2 implementation of the thin-wire core of NEC2, is discussed. The basic theory of the code is reviewed. Problems with early attempts combining Occam and FORTRAN are reported. A methodology for re-coding an old code written in a non-structured language in a modern structured language is presented; an example is given of the methodology applied to one of the more complex subroutines of NEC2. Methods of parallelizing the matrix generation are discussed. The stability of moment method solutions for problems requiring a large number of unknowns is investigated by using a physically symmetrical structure and solving the problem both with and without exploiting the symmetry. The effect of machine precision is also investigated and shown to affect the rate of convergence. The use of the biconjugate gradient method to accelerate convergence is considered and rejected, since for the test cases investigated with a large number of unknowns, the biconjugate gradient method is shown either to require more iterations than the conjugate gradient algorithm, or not to converge at all.

Conclusions are drawn regarding the role of the work in: deriving, analysing and implementing an efficient parallel conjugate gradient algorithm; introducing, analysing and implementing an efficient parallel LU algorithm; the development of theoretical models for predicting algorithm performance on local memory MIMD systems; the use of formal methods in parallel algorithm development and analysis; the use of pseudo-code for documenting parallel algorithms; the re-writing of major parts of a powerful general-purpose moment method code, NEC2, to properly exploit parallelism; investigating the accuracy of the moment method applied to electromagnetically large problems; and finally popularizing parallel computing in computational electromagnetics.

The dissertation is based on original research done by the author, except where explicit reference is made to the work of others.

Dedicated to my parents

Contents

List of Figures

List of Tables

Notation

$\nabla\times$	The curl operation
$\nabla\cdot$	The divergence operation
$\times$	The vector cross product of two vectors or the Cartesian product of two sets - the meaning will be clear from the context
$\overline{E}$	The (field) vector E
$O(M^n)$	of the order of M^n. Formally, $\mathcal{N} = O(M) \Rightarrow_{M\to\infty}^{lim} \log\mathcal{N}/\log M = n$
$[A]$	The matrix A
$[A]^T$	The Hermitian (complex conjugate) transpose of matrix A
$[A]^t$	The transpose of matrix A (interchange of row and columns only)
$[A]^*$	The complex conjugate of matrix A
a_{ij}	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; $\|\|[x]\|\| \equiv \sum_{i=1}^{n} \|x_i\|^2$. When necessary, the notation $\|\|[x]\|\|_2$ will be used to distinguish this norm from other norms.
$\lfloor x \rfloor$	The floor function of x, i.e. the integer part of x
$\lceil x \rceil$	The ceiling function of x, i.e. the smallest integer $\geq x$
$\equiv$	is defined as
$\forall$	for all
$\vee$	The Boolean OR operation
$\wedge$	The Boolean AND operation
$\|z\|$	Absolute value of z
$\mathbf{K}$	The set $\mathbf{K}$
$\mathbf{Z}^+$	The set of positive integers
$\|\mathbf{K}\|$	The size of $\mathbf{K}$ (i.e. number of elements)
$\Rightarrow$	implies

Glossary

- *Banked memory*: Memory that is split into separate banks that can be accessed in parallel by the CPU.
- *Benchmarking*: Establishing the real performance of a computer by measuring the execution time taken on a certain typical problem. The performance measured is in general a function of the problem.
- *Concurrent*: A synonym for parallel in the context of parallel processing.
- *Central processing unit (CPU)*: The unit that is responsible for fetching instructions from memory, moving data to and from memory, and carrying out logical or arithmetic functions.
- *Complexity*: The number of elementary arithmetic operations required by some operation. A synonym for "operation count".
- *Deadlock*: Deadlock occurs when two communicating processes are both waiting for an event that can never happen. A typical example is that process A is waiting to send a message to process B, but process B is waiting to send a message to process A.
- *Direct Memory Access (DMA)*: Communication paths to memory.
- *Geometrical Theory of Diffraction (GTD)*: An analytical technique for the prediction of high-frequency diffraction phenomena. The theory is essentially an extension of geometrical optics to include diffracted rays, in addition to direct, reflected and refracted rays, to describe the total field at a point in space.
- *Memory bandwidth*: The rate at which data can be moved from the CPU to memory and vice-versa. Normally measured in MBytes/s.
- *Method of Moments (MoM)* or *Moment Method*: A numerical technique for reducing an operator equation to a set of linear equations. It is most closely related to the Method of Weighted Residuals. A more complete discussion will be found in Chapter 1.
- *Multiple Instruction, Multiple Data (MIMD)*: A class of computer that performs different instructions on different data simultaneously.
- *Multiple Instruction, Single Data (MISD)*: A class of computer with multiple instructions operating on the same datum simultaneously (primarily of theoretical interest).

- *MFLOP/s*: Million floating point operations per second. This is the primary computational specification of interest in numerical analysis.
- *MIP/s*: Million instructions per second. This specification must be used with caution, since it is not necessarily a good indication of the rate of floating point computation.
- *The Numerical Electromagnetics Code – Method of Moments Version 2 (NEC2)*: A public domain code that uses the method of moments to solve electromagnetic radiation and scattering problems. It is discussed in more detail in Chapter 6.
- *paradigm*: An accepted way of looking at something.
- *procedural language*: Also known as an imperative language. This is the type of language generally used in science and engineering. The program instructs the computer to carry out a sequence of operations. Examples are FORTRAN, Pascal and Occam. (Languages such as LISP represent a different type of language, namely declarative languages).
- *Random access memory (RAM)*: The memory available for code and data on a computer.
- *Reduced Instruction Set Computer (RISC)*: A computer with a small instruction set containing only frequently used instructions. The computer is, however, designed to execute these instructions as fast as possible.
- *Scaling*: A scalable algorithm is one whose efficiency is a function of the "grain" of the problem, where the "grain" is a function of the number of matrix elements per processor, in the context of this thesis.
- *Single Instruction, Multiple Data (SIMD)*: A class of computer that performs the same instruction on different data simultaneously.
- *Single Instruction, Single Data (SISD)*: A class of computer that performs one operation on one datum at a time.
- *Virtual Memory*: Memory that is obtained by swapping the contents of real memory (RAM) with a mass storage device, typically a disk. This allows users to access massive amounts of memory that would not be economically viable to provide as real memory. This "paging" (swapping) is transparent to the user, in that he does not have to write any code to initiate or control the paging, which is handled by the operating system. However, the paging does impose some time penalty.

Acknowledgements

The author would like to acknowledge most gratefully the efforts of his promoters, Prof. J.H. Cloete of the University of Stellenbosch, and Prof. D.A. McNamara of the University of Pretoria. Their support and guidance of his research, their very careful review of the first draft of this dissertation and their constructive criticism have been most appreciated. The author would also like to acknowledge the efforts of Prof. Cloete in establishing the antenna and electromagnetic facilities at the University of Stellenbosch.

A number of people contributed to this thesis through advice, discussion, preparation of illustrations and the provision of computational facilities. They are P.J. Bakkes, T.W. Coates, C.F. du Toit, S. Mostert, Prof. J.J. du Plessis, Prof. P.W. van der Walt and D.M. Weber, of the University of Stellenbosch. The author in particular wishes to acknowledge the efforts of Prof. J.J. du Plessis in obtaining the MC^2 computer, the transputer array on which the parallel algorithms described in this thesis were implemented and tested. He also wishes to acknowledge D.M. Weber for the provision and maintainance of the VAX cluster on which many of the validation exercises and much of the debugging was performed, and for introducing the author to LaTeX. Others who contributed to the initial research through frequent discussion were Dr. D.J. Janse van Rensburg of the University of Pretoria, Prof. J.C. Olivier, now at the University of Potchefstroom, Dr I.M.A. Gledhill of the Council for Scientific and Industrial Research (CSIR) in Pretoria, and Dr. D.L. Hawla, previously of the CSIR and now at Hibbit, Karlson and Sorensen, Inc., Rhode Island. The help of Miss R.E. Cloete with the proof reading of the references was much appreciated.

The author would also like to acknowledge the contributions of Prof. J.A.G. Malherbe of the University of Pretoria for initially interesting the author in electromagnetics, and Dr. D.E. Baker, previously of the CSIR and now at EM-Lab, Pretoria, for further stimulating this interest.

The work was done at facilities provided by the Department of Electrical and Electronic Engineering of the University of Stellenbosch. Some of the initial work reported in Chapter 2 was carried out while the author was at the CSIR.

The probing questions of Dr. E.K. Miller of Los Alamos National Laboratories, New Mexico, regarding computational electromagnetics, and in particular his regular *PCs for AP* column in the IEEE Antennas and Propagation Society Magazine (which is about rather more than just personal computers) have been a constant source of ideas for the author. The author would also like to acknowledge the comments and encouragement of the Magazine's editor, Dr. W.R. Stone, of Expersoft Corporation, La Jolla, California.

This thesis was typeset by the author in LaTeX, Dr. L. Lamport's extension of Prof. D. Knuth's typesetting program TeX.

Chapter 1

Introduction

Even if we do discover a complete unified theory, it would not mean that we would be able to predict events in general ... even it we do find a complete set of basic laws, there will still be in the years ahead the intellectually challenging task of developing better approximation methods, so that we can make *useful predictions of the probable outcomes in complicated and realistic situations.*

from "A Brief History of Time", S. Hawking, Bantam 1988, pp.168-9 (the present author's emphasis).

Maxwell's equations provide a complete classical description of electromagnetic wave propagation and interaction with structures. They are invariant under the Lorentz transform and hence automatically incorporate special relativistic effects [Kon86, p.1]. The theory is classical in that quantum effects are not included. The necessary extension to include quantum theory may however be effected by replacing the field vectors by operators; see [HM89, p.xxi] and [Kon86, Section 5.10]. In the modern form, the equations are

$$\nabla \cdot \overline{D} = \rho \tag{1.1}$$

$$\nabla \cdot \overline{B} = 0 \tag{1.2}$$

$$\nabla \times \overline{E} = -\frac{\partial}{\partial t}\overline{B} \tag{1.3}$$

$$\nabla \times \overline{H} = \overline{J} + \frac{\partial}{\partial t}\overline{D} \tag{1.4}$$

where the field vectors have their usual meaning [Kon86, p.2]. Maxwell's original work is still available in the Dover reprint edition [Max54] and it is interesting to compare his notation with that of modern electromagnetics. These equations were first written in the modern form given above by Heaviside and Hertz working independently, and for some years bore the name Hertz-Heaviside equations [Nah88, p111]. Maxwell wrote out his equations in Cartesian component form; he also used quaternionic concepts in his notation[1][Nah88, p.109].

The solution of Maxwell's equations is not simple for general problems. With time-harmonic fields, i.e. frequency domain problems, the problem of the solution of the equations, with appropriate boundary conditions, can be formulated as a boundary value problem. Analytical solutions exist only for a very special class of problems requiring that the boundary(ies) of the problem coincide with a constant coordinate surface. For arbitrary problems, where the geometry does not satisfy this requirement, computer methods have become the standard method of solving the resultant equations. With time-domain problems, the problem can be formulated as an initial value problem, and again, computer methods have become the standard method of solution.

There are two fundamentally different approaches used in the computer solution of Maxwell's equations. These are:

- *The Method of Moments (or Moment Method)* The moment method embraces a very wide array of numerical methods based on a finite series approximation of the unknown field quantity. It will be discussed in more detail shortly. References on this subject are legion; useful introductory treatments are to be found in [Ell81, Chapter 7], [ST81, Chapter 7] and [Bal89, Chapter 12]. More advanced treatments may be found in [Har82, Mit73, Skw81, MP86, Wan91]. Several survey papers on the topic are available; two useful ones are [Ney85] and [Har87]. Hansen's collection of reprints contains many of the seminal papers [Han90]. A short history of the method of moments applied to electromagnetic field computation is given by Harrington [Har90][2]. Miller's tutorial paper [Mil88] on computational electromagnetics concentrates on the moment methods: the theoretical basics of the various moment methods are very succinctly summarized, as well as the computational requirements of each method. The moment method has no inherent theoretical limitations regarding frequency, but the computational requirements of the method limit the electromagnetic size of object that

[1]Quaternions enjoyed substantial support in the nineteenth century amongst mathematicians and physicists, but have since been relegated to mathematical history by the far simpler vector notation. Heaviside devoted part of his career to promoting the use of vector notation [Nah88, Chapter 9].

[2]This paper also explains Harrington's choice of the name "method of moments" — the name originates with the Russian mathematicians Kantorovich and Akilov.

can be solved; more detail is given shortly. The moment methods are thus "low-frequency" methods in the sense that there are upper bounds to the frequencies that can be tackled in practice. However, in the absolute sense, moment methods are frequently used well into the microwave spectral region, depending on the physical dimensions of the structure.

- *Asymptotic methods* The geometrical theory of diffraction is a well known asymptotic method, and is based on asymptotic approximation of the fields. A most useful textbook on the topic is that of McNamara [MPM90]; another good text is that of James [Jam86] and Balanis provides a very useful coverage [Bal89, chapter 13]. Many seminal papers on the topic may be found in Hansen's collection of reprints [Han81]. The GTD, being based on optics, is a high-frequency method.

The moment method may be defined formally as a method to convert a linear operator equation to a matrix equation[3]. The basic procedure may be summarized in its most general form as follows:

Consider the inhomogeneous equation

$$\mathcal{L}f = g \tag{1.5}$$

The unknown function f is expanded in a set of *basis* (or *expansion* or *trial* functions) as

$$f \approx \sum_{j=1}^{M} x_j f_j \tag{1.6}$$

where f_j is the known j-th basis function and x_j is the j-th unknown amplitude of this basis function. Using the linearity of the operator $\mathcal{L}$, equation (1.5) becomes

$$\sum_{j=1}^{M} x_j \mathcal{L} f_j \approx g \tag{1.7}$$

This yields one equation in M unknowns. Re-writing equation (1.7) as

$$\sum_{j=1}^{M} x_j \mathcal{L} f_j - g = R \tag{1.8}$$

[3]Thus defined, the moment method encompasses all the other non-asymptotic computer methods used in computational electromagnetics which use either the Maxwell curl equations or source integrals employing Green's functions [Mil88, p.1283]. Examples of the former are the finite element and finite difference methods; an example of the latter, the boundary element method.

yields the *residual* R; this can then be forced to zero at M points to produce the desired set of M equations — the collocation method used in NEC2 — or more generally, *weighted* with a set of M weighting functions, w_i, and the *weighted residuals* set equal to zero as

$$\sum_{j=1}^{M} < w_i, x_j \mathcal{L} f_j - g >= 0 \quad i = 1, 2, \ldots, M. \tag{1.9}$$

The weighting of the residuals with the weighting functions may be viewed as forming a moment, hence the name of the method. This method is also known in the general engineering literature as the Method of Weighted Residuals [FS66]. The $< \cdot, \cdot >$ symbol defines an inner product with the necessary mathematical properties, see [Kre78, p.129].

Equation (1.9) can be re-written in matrix notation as

$$[A][x] = [b] \tag{1.10}$$

where

$$a_{i,j} =< w_i, \mathcal{L} f_j >$$

and

$$b_i =< w_i, g >$$

and hence the linear operator equation has been converted to a matrix problem, namely the solution of a system of linear equations.

Different choices of weighting function correspond to various well-known methods: a set of Dirac delta functions yields the collocation method; identical weighting and testing functions produce the Galerkin method. If the operator is positive definite, then it may be shown that the Galerkin technique is equivalent to the Rayleigh-Ritz variational procedure, permitting a proof of convergence [Dud85].

The moment method considered in this thesis uses source integrals employing a Green's function[4]; the unknown is the (equivalent) surface current on the surface of the structure[5]. Time-harmonic excitation is assumed, permitting the problem to be formulated in the frequency domain. This is a very efficient formulation for the problem out of which the work reported in this thesis grew, namely the requirement by local industry for the prediction of antenna performance on vehicles composed of highly conducting materials. More details on the specific formulation will be found in Chapter 6.

For the analysis problems considered in this dissertation, the moment methods generate matrices of the form

[4]The specific integral equation is the Electric Field Integral Equation[BP81a, p.3]; this is a Fredholm integral equation of the first kind [DM85, p.3].

[5]Such a formulation may also be called a boundary element formulation [SF90, p.163].

$$[A][x] = [b] \tag{1.11}$$

where the matrix $[A]$ is known (it represents the discretized version of the operator $\mathcal{L}$), the vector [b] is known (it corresponds to the excitation field), and the vector [x] is unknown and must be solved for. The computationally expensive parts of the method of moments[6] are:

- Filling the matrix $[A]$ — an $O(M^2)$ operation, where M is the number of unknowns required to adequately discretize the problem, and is proportional to frequency.
- Solving the system of equations $[A][x] = [b]$ — an $O(M^3)$ operation.

These issues will be discussed in more detail in Chapter 2. The storage requirement is $O(M^2)$.

As has already been briefly mentioned, the moment methods have no inherent theoretical limitation in terms of the electromagnetic size of the problem, unlike the asymptotic methods. However, the moment methods become very expensive computationally as the number of unknowns increases. The asymptotic methods, such as the GTD, require that the problem be sufficiently large in terms of wavelengths; all reflection/diffraction and source points must be of the order of at least one wavelength apart, and a similar restriction pertains to the geometrical dimensions of the structure (for instance the radius of a cylinder). Using conventional MoM formulations and conventional computers, it transpires that there is a substantial "gap" between the capabilities of the MoM and the GTD. This will be illustrated by an example in Chapter 2.

This thesis addresses the question of the computational capability required for the moment method solution of problems which require a large number of unknowns for an accurate solution, but are electromagnetically too small for the asymptotic methods such as the GTD to be applied reliably. In particular, this thesis addresses the design, implementation and testing of efficient parallel algorithms for the solution of MoM formulations on local memory Multiple Instruction, Multiple Data (MIMD) computers.

In Chapter 2, the computational requirements of the MoM are reviewed, and various methods that have been proposed to allow a moment method formulation to handle large numbers of unknowns in a computationally efficient

[6]The basic operations of matrix fill and matrix solve are normally the most time-consuming part of any moment method code, but the specific *orders* given here are for the formulation using source integrals employing Green's functions that is used in this thesis. Other formulations, such as a finite element formulation [SF90, Mor90] have different computational dependencies, see [Mil88, Table VII], and also different storage requirements.

manner are discussed in some detail. The advantages and disadvantages of each are discussed. It is shown that for problems involving arbitrary geometries, none of the methods is entirely satisfactory at present. The chapter is concluded with a discussion of the very substantially increased computational capability that can be obtained if the possibilities offered by parallel computing can be systematically and efficiently exploited; it is this approach that is investigated in detail in this dissertation. Chapter 3 presents a review of parallel computing, with special application to computational electromagnetics. Attention is focused on pipelined and replicated systems. Examples are given of two computers exemplifying these different approaches, the CRAY-1 and a transputer array. Parallel computers using arrays of transputers are discussed in some detail, since such a machine was used to run the parallel programs described in this thesis. However, the algorithms described in this thesis are suitable for any local memory MIMD system, and are not restricted to transputer arrays. Pseudo-code is introduced that is used in later chapters to document the parallel algorithms. Chapter 4 discusses a parallel iterative solver, the conjugate gradient method. A new parallel conjugate gradient algorithm is proposed, theoretically analyzed, implemented and measured timing results compared with results predicted by the theoretical analysis. Chapter 5 describes a parallel LU algorithm and new parallel forward and backward substitution algorithms. Formal methods are introduced and applied to systematically extract potential parallelism, both for simple examples and for the LU method. A new graphical exposition is given that greatly simplifies understanding the algorithm. Theoretically obtained timing results available in the literature are checked and new, simpler derivations presented. The parallel LU algorithm is also implemented, and predicted and measured timing results compared. The accuracy of the LU method for moment method problems with a large number of unknowns is also investigated. Chapter 6 considers the parallelization of a very important public domain code, NEC2 [BP81c]. The problems encountered will be discussed and the methodology used by the author to re-engineer very substantial parts of the code in Occam 2 described. The new code runs very efficiently on a transputer array. The accuracy of the new code was checked carefully using physical symmetry to permit the comparison of the solution of a large system of equations with the solution of a rather smaller, but equivalent, system of equations. Some work on attempting to accelerate the convergence of the conjugate gradient algorithm, by using the biconjugate gradient algorithm, is also described and results presented. Chapter 7 draws some general conclusions about the work, and highlights the contributions.

Chapter 2

The MoM Solution of Large Problems

'...my circuits are now irrevocably committed to calculating the answer to the Ultimate Question of Life, the Universe and Everything ...but the programme will take me a little while to run.'

Fook glanced impatiently at his watch.

'How long?' he said.

'Seven and a half million years,' said Deep Thought.

from "The Hitch Hiker's Guide to the Galaxy", by Douglas Adams, Pan 1979, p.130.

2.1 Introduction

Solving electromagnetically large problems with the method of moments is computationally taxing. In this chapter, the basic computational requirements of the particular moment method used in this thesis are given. Various methods that have been described in the literature, including the use of symmetry, artificially zeroing matrix entries, hybridizing the MoM and GTD, and iterative methods, are considered. The advantages and disadvantages of each are discussed, with special regard to the solution of the *general* problems encountered in antenna engineering out of which this work grew, namely the prediction of the performance of antennas mounted on complex structures. Then the use of parallel computers to handle the generation and solution of the large matrices arising from a conventional MoM formulation is discussed; this is the main subject of this dissertation. This is put into perspective with the other methods.

2.2 Discretization of Electromagnetic Problems for a MoM solution

It has been found experimentally that around ten segments per wavelength are required to provide adequate sampling for a MoM formulation [BP81c, p.3]; if a surface is being modelled this squares to one hundred segments per square wavelength and if a volume is being modelled, it cubes to one thousand segments per cubic wavelength. The time required to solve the system of linear equations (using the LU method) goes up as the third power of the number of segments for a full matrix, so an integral equation formulation has a computational dependency of the power six for surfaces and of power nine for volumes [Mil88, Table VII, p 1286]. If the body is composed of a homogeneous material, the problem dimension can be reduced by one by an equivalent current formulation on the contour / surface for two and three dimensional problems respectively. This is expressed succinctly in the following formula [ibid.]

$$T_\omega \propto (\frac{L}{\Delta L})^{3(D-1)} \; ; \; D \in \{2,3\} \tag{2.1}$$

where T_ω is the solution time, L is the problem size, ΔL is the spatial resolution and D is the problem dimension; the D-1 assumes a homogeneous (or impenetrable) body where the source integrals are usually one dimension less than the problem dimension, as already noted[1].

At the time of writing, a typical "departmental" computer can solve a moment method problem with about 500 to 600 unknowns in an hour[2]. For objects of the size of the space shuttle, to take an example from [Mil90a, p.50], with a surface area of around 540 m^2, the maximum frequency at which a MoM code will be able to solve such a problem in an hour on the above computer is about 30 MHz. The controlling equation in this case is the following

$$f_{max} = c\sqrt{\frac{M_{max}}{Ad^2}} \tag{2.2}$$

[1]This formula is not valid for a one dimensional problem, where the relevant power is three.

[2]These figures are based on a Digital Equipment Corporation VAX 3600, the largest system readily available to the author at the University of Stellenbosch; the machine is rated at 2.9 VAX MIP/s; this performance rating is relative to the VAX 11/780, which is rated at 1 VAX MIP/s. The 3600 and 11/780 are representative of the type of computer readily available to the *average* university department at the time of writing. Note also that these figures assume that one has the machine entirely at one's disposal — such a job will normally be running in a batch queue and almost certainly will be sharing the machine's resources with interactive users and other batch queues, so the *elapsed* time is more likely to be several hours.

where f_{max} is the maximum frequency, c is the velocity of light in m/s, M_{max} is the maximum number of unknowns that can be solved in one hour, A is the area in m^2 and d is the sampling density in samples per wavelength (with $d = 10$ for the preceding calculation). If an antenna is mounted on the fuselage, which is several metres across (say 3 m), the lowest frequency at which the GTD is usable is about 100 MHz, and such calculations may well be of dubious validity. Clearly there is a "gap" in the coverage of existing techniques, which for the space shuttle sized vehicle falls in the VHF communication band — a most inconvenient place to be unable to predict antenna performance.

Inadequately discretizing a structure can generate plausible but dangerously incorrect results. The degradation can be gradual, as shown by the author in work for his Master's degree (reported in [DM87] and in more detail in [Dav86, Chapter 4]), or dramatic, as shown in Figures 2.1 and 2.2. These figures shows the radiation pattern of an antenna mounted on the rear of a truncated cone cylinder at S-band (nominally 2–4 GHz). The problem was modelled using the body of revolution formulation [DM87]. The total length of the generatrix was about 8 to 9 wavelengths. Figure 2.1 shows the radiation pattern obtained using approximately 30 segments — approximately $\frac{1}{3}$ of the number required by the ten segments per wavelength guideline. Figure 2.2 shows the same results, but with 89 segments. The main lobe has moved from approximately broadside to front-firing — where one would expect it to be from the viewpoint of the travelling wave excited on the cone/cylinder. The under-discretized solution is thus completely misleading.

2.3 The Exploitation of Symmetry

The most efficient way of handling larger problems is to exploit whatever symmetry may be available; for the majority of structures of practical engineering importance this is unlikely to be more than left-right symmetry for the basic structure, and the antenna is very likely to be mounted asymmetrically. Methods such as the Numerical Green's Function (NGF) option available in NEC2 [BP81c, p.89-92] permit one to model the basic structure (without the antenna) using symmetry, perform a LU decomposition on the symmetrical structure, save a NGF file (which is essentially the factored matrix), then add an antenna to the structure and use methods to factor the full (and now asymmetrical) matrix that re-use the already factored matrix representing the structure, without major additional computational cost. While simple in concept, it greatly complicates the coding — a very substantial part of NEC2 is devoted to handling the NGF option. In any case, this left-right symmetry decreases the computational load by at most about four; this can

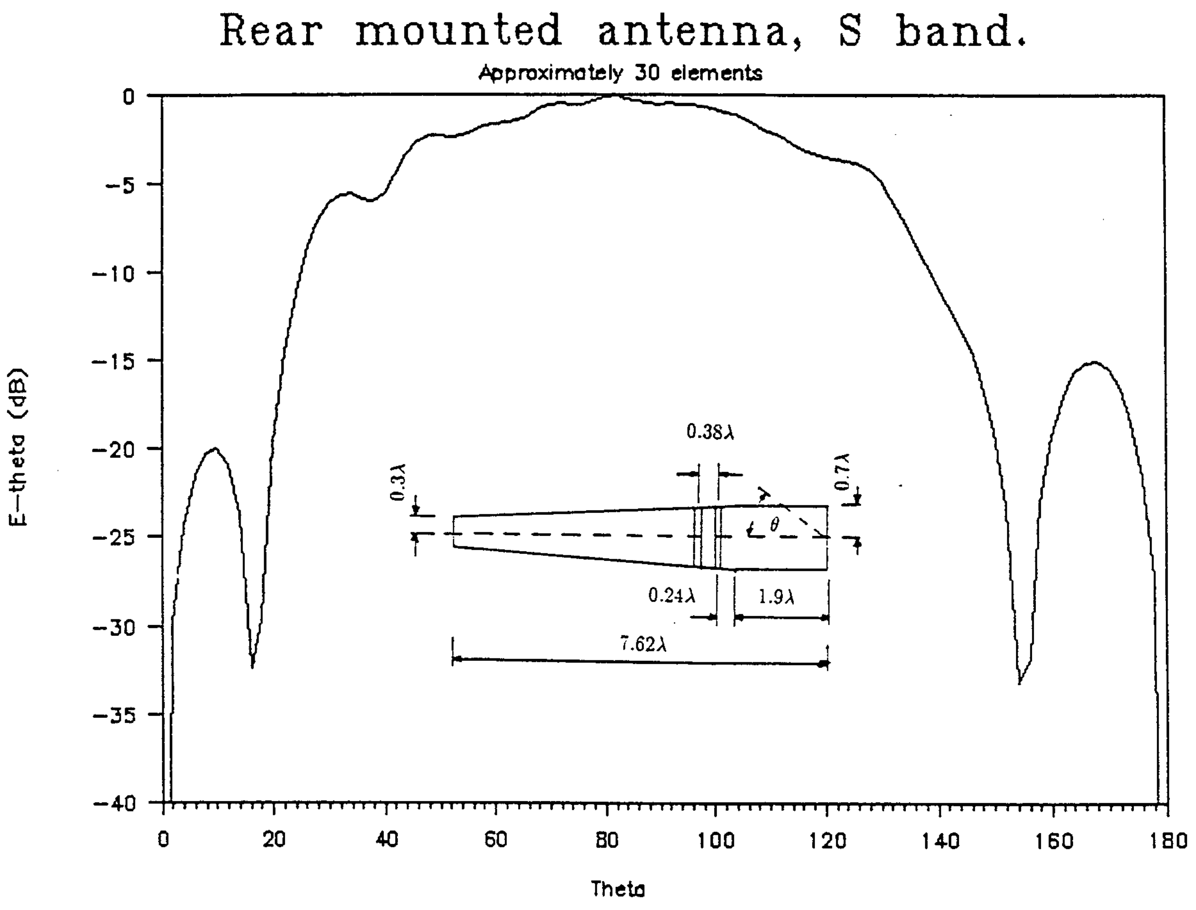


Figure 2.1: Capped cone-cylinder, S band, approximately 30 unknowns, pitch plane radiation pattern. Under-discretized by a factor of three according to the ten segments per wavelength guideline.

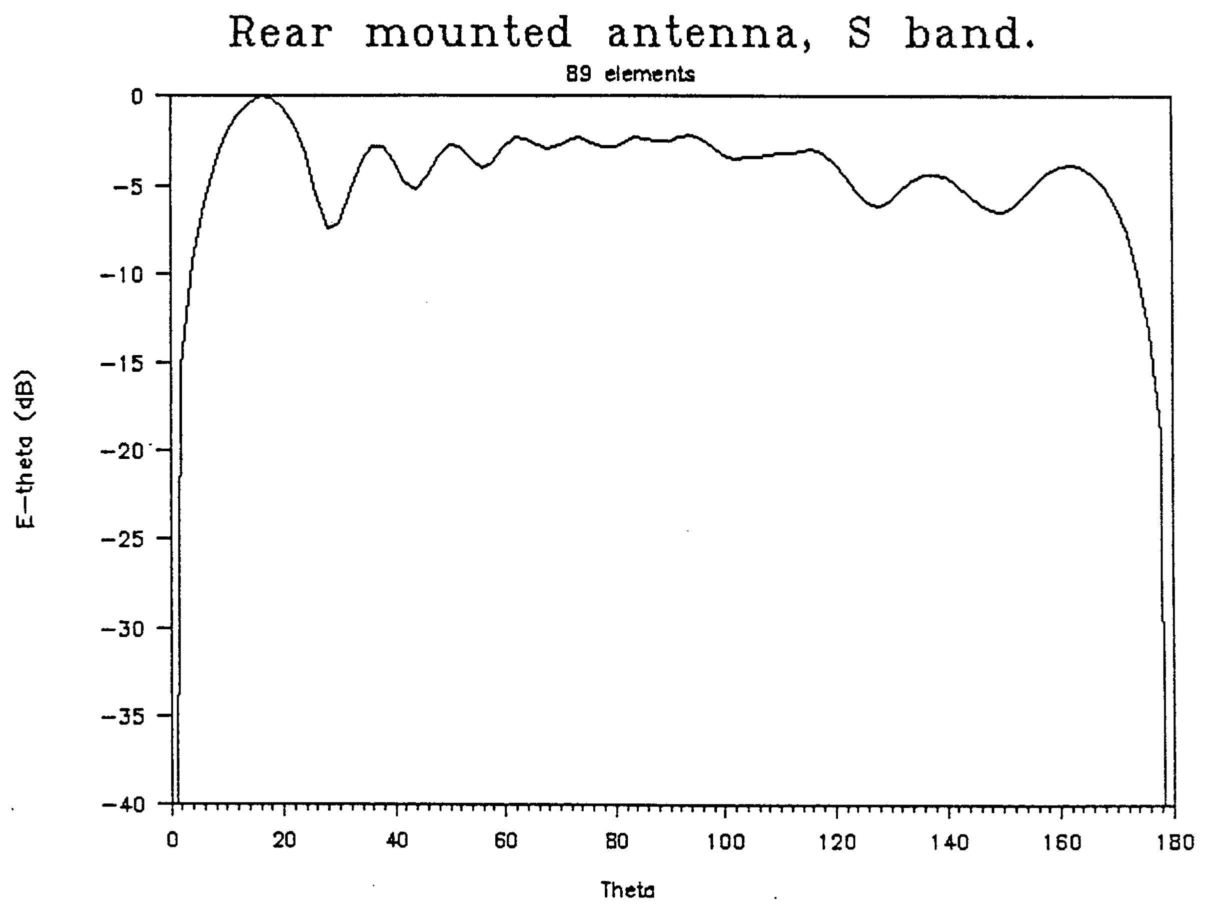


Figure 2.2: Capped cone-cylinder, S band, 89 unknowns, pitch plane radiation pattern. Correctly discretized according to the ten segments per wavelength guideline. Note shift of main lobe compared to previous figure.

be computed as follows:

The general formula for the time to compute a single frequency MoM solution in free space for the MoM formulation used in this thesis is given by the following formula, adapted from [BP81b, p. 165]:

$$T = A\frac{M^2}{S} + B\frac{M^3}{S^2} \tag{2.3}$$

where M is the number of segments (the number of unknowns for NEC2), S is the number of degrees of symmetry, and A and B are parameters dependant on the computer and algorithm used. The first term represents the matrix fill operation and the second term the matrix solve operation. There is also a term of $O(M^2)$ representing the substitution step(s) and a term of $O(M)$ representing the calculation of radiation patterns. These terms are not shown in the above, since for large systems, the two terms shown are the dominant terms unless an extra-ordinary number of different excitations and/or field points are required. The constant A may also be weakly geometry dependant if the numerical solution enforces current continuity between segments, as is the case with NEC2.

Hence halving the number of unknowns using left-right symmetry will result in a saving of time of the order of four [BP81b, p.32].

2.4 The Impedance Matrix Localization Method

Another more radical method is that recently reported by Canning [Can90], the Impedance Matrix Localization (IML) method. The origin of the method is the idea of limiting the interaction distance between elements, only computing matrix elements within this limited interaction distance and zeroing the rest. Such methods been reported by Moore and Pizer [MP86, Chapter 15]; they called the method the sparse matrix approximation. The result is a matrix with a number of zero elements, and if correctly numbered (which is not in general a trivial problem), the resultant matrix is banded, which can be exploited by an LU solver to save both memory and time. Alternately, an iterative solver may be used to avoid the re-numbering problem, also with attendant memory and time savings.

The IML method is an extension of this idea. Canning's ideas grew out of two observations: firstly, one has a very wide choice of basis and testing functions in the MoM, and if one could choose them in some fashion to make nearly all the MoM matrix entries negligible — something which cannot be said of the standard interaction distance limiting methods with any certainty — then the desired result has been achieved. Secondly, this must be possible, since the GTD achieves this in the high-frequency limit.

The IML may be written mathematically as

$$[T] = [A][Z][A]^T \tag{2.4}$$

where $[Z]$ is the interaction (impedance) matrix generated by a conventional MoM formulation, $[A]$ is some transformation matrix and $[T]$ the transformed matrix [Can90, eqn.1]. The idea is to develop a set of basis functions which correspond to a matrix $[A]$ which accomplishes the near-zeroing of most of the elements of $[T]$.

Initial work produced a transformation matrix that was very ill-conditioned, but Canning's later work produced a suitable, well conditioned transformation matrix. Canning reports very promising results [Can90]. The result of Canning's work is to demonstrate that it is possible to reduce the number of significant matrix elements from $O(M^2)$ to order $O(M)$.

This work appeared during the closing stages of the work reported in this thesis; it should be emphasized that the very promising results reported by Canning do not invalidate the basic contents of this thesis. By incorporating Canning's work into the now existing parallel MoM codes, far larger problems could be addressed than is presently possible, since the IML method will still be time-intensive for large problems. Indeed, Canning comments "What is needed today is a method which scales better with problem size, so that it may be efficient on these modern (powerful) computers"[Can90, p.18]. Canning's present method requires iterative solvers, using an incomplete LU decomposition, and this thesis considers both parallel iterative solvers and parallel LU decomposition.

2.5 Hybrid MoM-GTD formulations

Several researchers have investigated the possibility of combining the MoM and GTD. Thiele and Newhouse [TN81] augmented the MoM impedance matrix elements by including GTD contributions. Burnside, Yu and Marhefka [BYM81] computed the field due to MoM currents at a distance from the segments sufficiently great that the field can be matched to a GTD scattered field. This permitted them to extend the GTD to problems where the canonical diffraction terms were unknown.

For complex structures with many possible optical paths for the diffracted/reflected rays, the ray-tracing problem is far from trivial computationally and the codes can have very substantial computational requirements, sufficient to warrant parallelizing the code [Sch90]. Such multiple interactions remain significant in the "gap" region where the structure is electromagnetically relatively small. Hence a hybrid code can be expected to have substantial computational requirements in the "gap" region.

The basic requirements of the GTD in terms of electromagnetic size will also have to carefully monitored in the regions where the MoM is not used. For reliable operation, a typical GTD code such as the Numerical Electromagnetic Code — Basic Scattering Code [BMY73] requires that all plate structures should have edges at least a wavelength long; the major and minor radii and length of all elliptic cylinders should also be at least a wavelength and each antenna element should be at least a wavelength from all edges and the curved surface. These limitations will not disappear with a hybrid formulation; the use of the MoM in the region near the antenna will result in a relaxation of the requirements in this region, but not for the entire structure.

It would be reasonable to view the hybrid methods as attempting to close the "gap" between the conventional MoM and GTD by pushing the minimum usable frequency of the GTD downwards, as opposed to the work on different MoM formulations that attempts to push the maximum usable frequency upwards. The author is not aware of any general purpose code readily available at the time of writing that successfully combines the two methods at a *fundamental* level.

2.6 Iterative Methods

Recent years have seen much debate on the topic of iterative methods as a method for solving large problems [DM88, JvRM88, RP88, Wan90a, Wan90b, Wan91, Sar88]. Two issues may be identified in the debate:

- The first relates to the relationship of the "direct" iterative methods and the moment methods, and the convergence of each to the exact solution;
- the second relates to the question of computational efficiency.

The debate on the first issue was precipitated by Sarkar [Sar86]. Great stress was laid in his article on the difference between the application of the CG method for the solution of the system of linear equations resulting from a MoM formulation, and the "direct" application of the CG method to the underlying operator equation itself, without (explicitly) discretizing the equation, as required by any moment method formulation. The guaranteed convergence property of the CG algorithm was then used to argue that the "direct" method converges to the exact solution as the number of unknowns increases. Similar claims were emphatically repeated in [Sar87b]; Sarkar stated "There is a heaven and hell difference between the application of the conjugate gradient method to solve a matrix equation in the method of moments, as compared to the application of this method directly to solve an operator equation" (sic) [Sar87b].

The author tested the claims in Sarkar's controversial paper [Sar86] by applying the method to the solution of radiation from a body of revolution; details have been published in [DM88]. The result was the observation that the "direct" application to the operator equation and the application to the system of linear equations generated by a MoM collocation (delta weighting functions) formulation using pulse expansion functions led to precisely the same systems of linear equations, and the solutions were numerically identical. The run-time for the "direct" application was rather longer than for the CG-MoM program since the matrix elements were essentially being recomputed at each iteration. Similar findings were published by Janse van Rensburg and McNamara [JvRM88]; their results using the "direct" method for scattering from a straight thin wire were also indistinguishable from a MoM formulation. The reason for this may be found in the theory of functional analysis. In order to represent an operator in discrete form (i.e. as a matrix representation) for computational purposes, it is necessary to restrict its domain (range) to some finite-dimensional subspace of the continuous domain (range) and select basis functions in both these subspaces. The basis functions of this restricted domain of the operator will be the set of expansion functions for the unknown, while the basis functions for the restricted range of the operator will be the weighting functions. These observations also permit the adjoint operator, required by the "direct" CG method, to be constructed very simply using the well-known fact that in any finite dimensional Hilbert space, the adjoint operator is simply the Hermitian transpose [Kre78]. This point is elucidated in [JvRM88].

Ray and Peterson's paper [RP88] is a particularly well written exposition of the first issue, demonstrating clearly the difference between what they identified as *solution convergence* and *algorithm convergence*. Solution convergence addresses the question: as the representation of the function improves (i.e. more unknowns are used in the finite series representation), how well does it approximate the exact continuous solution? Algorithm convergence and error address the question: given an estimate of a solution within a particular representation(i.e. a particular number of unknowns), how close is it to the best possible solution within that representation? By (incorrectly) overlooking the implicit discretization in the "direct" approach, Sarkar used results for algorithm convergence to (incorrectly) claim solution convergence.

Despite the conclusive results of Ray and Peterson's work, as well as the supporting results of several others, the present author included [DM88, JvRM88, Sar88], an exchange indicating a difference of views on this subject was still in progress at the time of writing [Wan90a, Wan90b, Wan91].

Another important claim made for the "direct" method is that it does not break down at frequencies corresponding to internal resonances [NRS87]; this related to the debate on the first issue. This is also a controversial claim since the problem is that the underlying operator is ill-conditioned in this case, not just the discretized MoM formulation. It is shown later in this thesis that the CG method is very robust, permitting the solution of very ill-conditioned matrices that even a double precision LU solver is unable to solve. This thesis does not investigate this claim further, but the author would suggest that it is this *robustness* of the CG algorithm rather than the application of the "direct" method *per se* that permits one to obtain solutions at frequencies corresponding to internal resonances, and that the same results would have been obtained had a conventional MoM formulation with a CG matrix solver been used. Mittra and Klein [MK75, p.140] show that for the case of a square cylinder, at frequencies corresponding to internal resonances, the condition number of the matrix becomes very large — it is, however, finite. Thus the author's explanation of the results observed by the "direct" method adherents at internal resonance frequencies have a reasonable basis. It is also supported by [RP88, Section V].

The debate on this first issue has tended to overshadow the second issue, namely the memory saving and computational efficiency claimed for the method. This issue is the more important one in the context of this dissertation. Both the memory saving and the computational efficiency are actually the result of exploiting Toeplitz symmetry. Very similar time and memory savings have been obtained in conventional MoM codes that correctly exploit the available symmetry; an example will be given shortly.

It has been claimed that the iterative methods permit the modelling of much larger structures than the conventional MoM [Sar86]. This claim is rather misleading, for the following reasons. While it is true that the "direct" CG formulation only requires storage for several vectors of $O(M)$ as opposed to the MoM which requires storage for a matrix of $O(M^2)$, it does that at the cost of essentially re-computing the coefficient matrix at each iteration [Wan91, p.257]. It is only when the Fast Fourier Transform (FFT)[3] can be used to evaluate the matrix elements that the "direct" CG method becomes a viable computational contender, *and this occurs only with a Toeplitz operator* [Sar86]. The finite dimensional matrix representation (using a uniform discretization) of a Toeplitz operator is also obviously Toeplitz. A classic example of Toeplitz symmetry arises with the MoM formulation for radiation or scattering from a uniformly discretized straight, thin wire. It should be noted that for this case, algorithms exist with similar computational require-

[3]Many of the methods that exploit the computational efficiency of the FFT can be traced back to work by Bojarski, reported in [Boj82]

ments to the "direct" CG method (viz $Mlog_2(M)$ per iteration; note that there is an error in the closing paragraph of [Sar86, p.11] where it is stated that the dependence is $O(M)$; the previous page [Sar86, p.10] gives the correct order). An example is given in [PM85b, p.129] where the CG method is used to solve the set of linear equations generated by the MoM, but the matrix-vector products required by the method are implemented using the fact that with a Toeplitz matrix, this operation can be viewed as a discrete convolution and the FFT used to implement the multiplication. It is notable that in work on problems without the convolutional property required to exploit the FFT, the adherents of the "direct" approach frequently do not give run-times for the algorithms [NRS87]. If the FFT is used with the "direct" method interpretation, a MoM interpretation is still possible, but the details are rather more involved [PM85b]. Some other problems with using the FFT were noted by Steyn and Davidson [SD90].

When the necessary symmetry exists, permitting the full exploitation of the FFT, impressive results are possible. Zwamborn has recently described an iterative formulation that permits a full three dimensional model of an inhomogeneous object using around 27 000 cells ($30 \times 30 \times 30$) [Zwa91, p.138]. The Toeplitz symmetry is obtained by embedding the target in a cube, using a contrast function to differentiate the target and the free space "embedding" and using an regular grid. Such methods are, however, difficult if not impossible to apply *accurately* to general problems involving *arbitrarily* orientated surfaces, such as the vehicle mounted antenna problems of engineering interest that initiated the work reported in this thesis.

2.7 Parallel Computing

This chapter has considered a number of methods for the solution of electromagnetically large problems. One final method has yet to be considered — the exploitation of far more powerful computers. On the one hand, this solution is trivial if one waits for conventional computers to increase in speed, as they have been doing consistently for the last four and a half decades. On the other hand, there is the challenge of developing new algorithms to exploit the radical increase in processing power made possible by the development of parallel processing. To develop such algorithms so that they will run *efficiently* on this new class of computers is a very far from trivial problem. It is this problem that is addressed in this thesis.

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. This thesis considers the use of a rather cheaper type of computer, the local memory Multiple Instruction Multiple Data (MIMD) computer. Such systems offer performance rivaling that of the vector supercomputers, but require that the algorithms be very carefully designed to exploit the parallel architecture and obtain something approaching the manufacturer's claimed peak performance. The burden of this thesis is primarily the derivation, analysis, implementation and testing of such algorithms.

The parallel algorithms and timing models that will be developed in this thesis are applicable to any local memory MIMD system, and the theoretical results depend on only two machine dependant parameters, the speed of computation and communication. To validate the theoretical analysis, the algorithms are implemented in Occam 2 and run on a transputer array; the machine parameters mentioned are obtained by benchmarking and the predicted and measured results compared. These specific timing results are inevitably dependant on the particular computer used, but the basic algorithms and theoretical analysis will not date as newer computing technologies replace the transputer technology that was available at the time of writing.

It is also important to ensure that the large systems of equations can indeed be solved accurately, and that the rounding errors imposed by finite digit arithmetic do not degrade the solutions. In general, the matrices generated by moment methods do not permit use of the existing theorems on the growth of rounding errors in LU or iterative solutions of systems of linear equations, and the question of accuracy must be investigated by computational validation.

While the use of more powerful computers with existing algorithms is still ultimately limited by the third power law on the solution of the system of equations, it has been seen from the example given that the "gap" between the MoM and GTD does not require a tremendous increase in the maximum usable frequency of the MoM to bridge; for the space shuttle example discussed, the ratio of the minimum reliable frequency for the GTD to the maximum usable frequency for the MoM is about three, using the criteria discussed in Section 2.2 (viz. for the MoM, one hour of CPU time on a typical system such as the VAX discussed — about 500 to 600 unknowns — and for the GTD, minimum dimension approximately 1λ). An increase in computational speed by a factor of slightly less than 1 000 would be sufficient to bridge this gap[4]. While the maximum number of processors used in this

[4]At the time of writing, the Jet Propulsion Laboratory, California Institute of Technology, were installing an array of 512 Intel i860 processors [Cwi91]. The i860 is a rather faster processor than the T800 transputers used in this thesis.

thesis was 31, the algorithms that are described demonstrate excellent scaling properties, and will run efficiently on much larger numbers of processors, so the basic premise of extending the MoM using large processor arrays is valid and tenable.

2.8 Conclusions

The methods receiving attention in the literature for the solution of electromagnetically large problems have been reviewed. The discretization requirements for reliable approximate solutions have been discussed and an example shown to illustrate the "gap" between the MoM and the GTD for a problem representative of those of practical interest. The possibilities offered by exploiting symmetry have been reviewed; many problems of practical interest do not unfortunately possess the necessary symmetry. The Impedance Matrix Localization Method, an extension of the sparse matrix approximation, has been described. Preliminary results obtained with the method appear promising. Hybrid MoM-GTD formulations have been briefly described and also show promise in closing the MoM-GTD "gap". The use of iterative methods has been reviewed and a debate in the literature regarding the relationship of "direct" iterative methods to iterative solvers applied to moment method codes has been summarized. The computational efficiency of the iterative methods is the result of the exploitation of Toeplitz symmetry; this limits the applicability of the methods to the general problems of interest in this thesis.

Finally, the systematic and efficient exploitation of parallel processing has been identified as a topic of great interest. Other methods discussed such as the Impedance Matrix Localization method will also benefit from the availability of efficient parallel algorithms. The work on extending the MoM (on the one hand via algorithms with lower computational requirements such as the Impedance Matrix Localization method, and on the other through the development of parallel algorithms that will run efficiently on large processor arrays — the subject of this dissertation) should be viewed as complementing work on hybridizing the GTD and MoM.

In the next chapter, a review of parallel computing is presented, concentrating on the types of computers that have shown the greatest utility for computational electromagnetics, namely pipelined vector computers and replicated MIMD arrays. Examples are given of contemporary machines epitomizing these two approaches.

Chapter 3

A Review of Parallel Computing

3.1 Introduction

This chapter provides an overview of parallel computing, with reference to numerical analysis in general and computational electromagnetics in particular. The chapter starts with a brief review of the history of parallelism and the general principles, then goes on to look at two examples of parallel computers that embody the two main types of parallelism encountered, viz. pipelining and replication. Then the classification of parallel computers in general is discussed. The question of quantifying the performance of a parallel algorithm is addressed. Theoretically predicting the approximate performance of an algorithm on particular hardware can save much wasted coding, by allowing the comparison of different algorithms without having to implement them all and then compare by benchmarking. A pseudo-code notation is introduced that is used to document the parallel algorithms in the rest of the thesis.

3.2 General Principles and Historical Background

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

- *Pipe-lining*: overlapping parts of operations in time.

- *Replication*: providing more that one functional unit (eg CPU).

These are discussed in more detail in Sections 3.3 and 3.4 respectively.

The history of modern (i.e. digital, electronic) computers begins towards the end of World War II; the first general purpose electronic digital computer was ENIAC, and rather interestingly contained many parallel aspects — for example, 25 independent computing units [HJ88, p.8]. Over a century previously, the initial designs of the machine generally regarded as the first computer, Babbage's *Analytical Engine*, also contained the concept of parallelism — although this appears to have been dropped in his later designs [HJ88, p.8].

Some form of parallelism has long been a feature of computer designs. By the 1960's most scientific computers were using bit-parallel arithmetic, i.e. processing the bits of a word in parallel. The 1953 IBM 701 was an early computer to use bit-parallel arithmetic [HJ88, p.12]. Another form of parallelism long used is parallelizing the I/O and arithmetic units by providing an I/O channel, to allow useful work to continue during read to / write from (almost inevitably slower) peripheral units. The IBM 704, commissioned in 1955, is an early example of such as system [ibid.]

These forms of parallelism are ideal from the user's viewpoint, since they are entirely transparent to him and all he sees is a machine with faster through-put. The next stage of computer development involved pipelining, that is the overlapping of operations in time. These operations could be instruction processing, where the operation of instruction fetch, decode, address calculation and operand fetch are overlapped on successive operations. An early example was the University of Manchester/Ferranti (later ICL) ATLAS machine (1961) [HJ88, p.14]. Alternatively, pipelining can be applied to arithmetic operations.

Possibly the biggest single advance in computational power was the introduction in 1976 of the CRAY-1[1]. This computer successfully combined a number of important concepts, incorporating pipelining, interleaved memory and an attention to the detail of parallelism that made the machines the fastest computers in the world when first delivered in 1976 [HJ88, p.118]. Due to the historical importance of the CRAY, it is described in more detail in Section 3.3.2.

However, there is a limit to the amount of parallelism that can be extracted via pipelining — it is certainly difficult to see how *general purpose* pipelines can be developed for much more than the basic operations already described. Recent designs have thus increasingly exploited replication to increase processing speed.

[1]Cray's eponymously named computer is one of the few thus distinguished.

When first introduced, replication tended to involve either providing a few, powerful processing elements, or providing a massive number of very simple processing elements. The former was the approach used on the CRAY-XMP — the XMP is essentially one, two or four CRAY-1-like CPU's sharing a common memory [HJ88, p.123][2]. The latter was the approach that ICL followed with their DAP (Distributed Array Processor) [HJ88, p.290]. The first production models[3] contained a two-dimensional array of 4096 1-bit processors.

The introduction by INMOS of the transputer[4] has had a major impact on parallel processing, due to the fast processing and communication speeds, the simple and effective provision made for communication between transputers, and the relatively low cost [HJ88, p.322]. The transputer was probably the first microprocessor to fully exploit the possibilities of VLSI (Very Large Scale Integration) technology for concurrent processing. The technology required to build moderately large transputer arrays (64 processor arrays are not uncommon) is rather simpler than that required for a pipelined supercomputer, making supercomputers based on replicated transputer arrays rather cheaper than a pipelined supercomputer, but with impressive computational potential. The algorithms developed in this thesis were tested on a transputer array, and more detail is provided on transputers in Section 3.4.2. Such arrays fall in between the extremes of the CRAY-XMP and DAP.

In the next section, pipelining and replication are discussed in detail.

3.3 Pipelining

3.3.1 Description of Pipelining

Suppose it is desired to add two floating point vectors $[a]$ and $[b]$ together, to produce the vector $[c]$. In the first clock cycle (or tick), the first elements a_1 and b_1 are fetched, in the second tick they are added, and in the third tick the result is stored. Thus an output is obtained every third tick. Now, by tick 2, the hardware dedicated to fetching operands is idle, and by tick 3, both this unit and the arithmetic unit are idle. Only on tick 4 is new data fetched. This is illustrated in Figure 3.1 for the case of vectors of length 3.

Instead of leaving these units idling, assume that it can be arranged that while the arithmetic unit is adding a_1 and b_1, the fetch unit is fetching a_2 and b_2 on tick 2, and on tick 3 the fetch unit is fetching a_3 and b_3, the arithmetic

[2]The two processor version was announced in 1982 and the four processor version in 1984 [HJ88, p.118].

[3]The first DAP computer was delivered in 1980.

[4]Transputers were first available in 1984.

unit is computing a_2 and b_2 and the output unit is outputting $c_1 = a_1 + b_1$. Thus the operations have been overlapped in time, and now (after an initial hiatus of two ticks) a result is obtained every tick instead of every third. This situation is illustrated in Figure 3.2. Thus, with *the same technology* the computer has been speeded up by almost three. On real computers, the amount of work that can be done in one clock cycle is rather less than in this idealized example, so the pipelines are generally deeper, offering more potential for speed-up.

$$\begin{array}{ll}
tick_1 & \xrightarrow{in} a_1; b_1 \\
tick_2 & c_1 := a_1 + b_1 \\
tick_3 & c_1 \xrightarrow{out} \\
tick_4 & \xrightarrow{in} a_2; b_2 \\
tick_5 & c_2 := a_2 + b_2 \\
tick_6 & c_2 \xrightarrow{out} \\
tick_7 & \xrightarrow{in} a_3; b_3 \\
tick_8 & c_3 := a_3 + b_2 \\
tick_9 & c_3 \xrightarrow{out}
\end{array}$$

Figure 3.1: Sequential execution of $\vec{a} + \vec{b}$.

$$\begin{array}{llll}
tick_1 & \xrightarrow{in} a_1; b_1 & & \\
tick_2 & c_1 := a_1 + b_1 & \xrightarrow{in} a_2; b_2 & \\
tick_3 & c_1 \xrightarrow{out} & c_2 := a_2 + b_2 & \xrightarrow{in} a_3; b_3 \\
tick_4 & & c_2 \xrightarrow{out} & c_3 := a_3 + b_3 \\
tick_5 & & & c_3 \xrightarrow{out}
\end{array}$$

Figure 3.2: Pipelined execution of $\vec{a} + \vec{b}$ with a pipe of depth 3.

In general, the time, t_{pipe}, to operate on a vector of length n is

$$t_{pipe} = [s + l + (n - 1)]\tau \qquad (3.1)$$

where l is the length of the pipe, $s\tau$ is the set-up time, and represents the penalty to be paid with a pipeline — a certain amount of time is required to set it up — and τ represents the interval between ticks. The set-up time is constant irrespective of the length of the vector. It can therefore be expected that the pipeline will not be efficient for short vectors, where $s\tau$ is of the same order as $(n-1)\tau$. The actual values of s and τ are determined either from a detailed knowledge of the specifications of the computer or via benchmarking (measurement).

A useful general form of characterization for parallel processors introduced by Hockney and Jesshope [HJ88, Section 1.3.2] is the $(r_\infty, n_{1/2})$ model. The execution time of an operation on a vector of length n is given by

$$t = r_\infty^{-1}(n + n_{1/2}) \tag{3.2}$$

This discussion has not specified what operation is implied. Each basic arithmetic operation is often allocated a special purpose pipeline, as will be seen in the example of the CRAY-1.

The *maximum or asymptotic performance* r_∞ is the maximum rate of computation in MFLOP/s. For a pipelined computer, this occurs as the length of the vector tends to infinity, hence the name. Comparing equations (3.1) and (3.2), $r_\infty = \tau^{-1}$ for a pipelined computer. The *half-performance length* $n_{1/2}$ is the vector length required to obtain half the maximum performance. For a pipelined computer, $n_{1/2} = s + l - 1$.

3.3.2 An Example — the CRAY-1

This section describes the CRAY-1[5] computer, which at the time of writing was a most important computer, very successfully exploiting pipelining. It can be expected that technological developments will date this *particular* section; however, as has already been stated in the introduction to this chapter, the parallel processing principles used in this thesis are not tied to the specific computer technology of 1991.

The CRAY-1 is a pipelined vector machine. Three floating point, pipelined, functional units are provided, one each for addition, multiplication and reciprocal approximation. The pipelines are 6, 7 and 14 deep respectively. These can produce a result every clock cycle (8.5ns on later models). Eight vector registers, each able to hold 64 64-bit long floating point numbers, are provided, along with three 64-bit data paths for direct memory access, and a set of 32 machine instructions for manipulating and performing arithmetic on these vectors [HJ88, Section 2.2.2].

[5]Cray was a founding member of Control Data Corporation (CDC), and was intimately involved with the design of the CDC 6600, a 1960's machine that combined principles such as pipelining and memory interleaving. In 1972 he left CDC to start his own company, with the aim of producing the fastest computer in the world. In the very short space of four years he succeeded, and in 1976 the first CRAY-1 was delivered.

The CRAY-1 benchmarked by Hockney and Jesshope [HJ88, Section 2.2] had a clock time of 9.5ns, which would imply a maximum through-put of around 105 MFLOP/s *per pipeline.* (Note that there are three such pipelines that can be chained; this concept is discussed later). The maximum rate at which one pipeline can produce results is also the maximum rate at which the system can transfer (on two separate 64 bit DMA links) the two operands from memory and (on another DMA link) the output to memory. However, the vector registers are provided to allow further operations on data without first transferring results to main memory. This helps reduce the memory bandwidth problem. On the subject of memory, the CRAY-1 also uses *banked* (also known as *interleaved*) memory, which is another type of parallelism encountered on virtually all supercomputers. Memory banks are groups of memory, which can send data to the processor in parallel. Thus relatively slow memory can be matched to a fast processor. (This concept was found in the very successful CDC 6600[6] of the 1960's [HJ88, p.16].).

The CRAY-1 also permitted *chaining* of vector instructions. Chaining means that if, for example, two vector operations using different pipelines are to be executed sequentially, the second pipeline starts as soon as a result is produced. Thus the system does not wait for the first pipeline to empty before initiating the second, but rather as soon as data is available. The result is to effectively multiply the length of the pipeline by the number of units that can be chained (assuming that the pipe lengths are approximately the same).

The CRAY-1 achieved its success by using the fastest semiconductor technology available at the time of its design, using pipelining, using vector registers to store temporary results and by carefully matching memory and processing requirements. Note that pipelining on its own is not enough — a designer of a large pipelined system has many other problems to consider, the most important of which are firstly, providing mechanisms to get data to the pipelines from memory and vice versa sufficiently fast to keep them occupied, and secondly, providing enough (sufficiently fast) memory.

The CRAY series supports FORTRAN as its main scientific language. Software support for parallel processing is provided in a special multi-tasking library, invoked by calls to FORTRAN routines in this library. At the time of writing, there was to be little in the way of standards for the parallel languages. This is an issue that the committee specifying FORTRAN 8X were addressing [HJ88, p.407].

The CRAY-1 was chosen here as an example to illustrate pipelining, since it is a most successful working example of such an implementation. The CRAY-1 regularly achieved 130 MFLOP/s on suitable problems, such as

[6] Also designed by Cray.

matrix multiplication. The biggest problem with the system is cost and availability. The systems cost several tens of millions of dollars at the time of writing, and the installations require special refrigeration gear for cooling purposes. (A CRAY-XMP weighs $5\frac{1}{4}$ tons, requires two 25-ton compressors and a 175 kVA generator [HJ88, p.120]). Virtually all such systems have been installed in large computer centres[7] The state of the art in supercomputers at the time of writing was probably the CRAY-2, with a maximum theoretical throughput of 2 GFLOP/s — 430 MFLOP/s has been reported on favourable problems [HJ88, Section 2.2.7]. The CRAY-2 has a clock period of 4.1ns, and uses some novel concepts, such as three-dimensional pluggable modules to keep interconnection distances short, and liquid immersion cooling, whereby all circuit boards and power supplies are totally immersed in a bath of slowly circulating clear inert fluorocarbon liquid. The CRAY-3 is an implementation in gallium arsenide, which is expected to allow a clock period of 1ns.

The CRAY series was chosen as example of a vector supercomputer, but before moving on to replicated designs, mention should be made of the recent trend towards minisupercomputers, such as the Convex range [HJ88, p.49]. These provide many of the capabilities previously reserved for CRAY users at a much cheaper price. The Convex is broadly similar to the CRAY series, in that it is based on a series of functional units working from vector registers. The entry-level system, the C-1, (now known as the C-120) has been benchmarked at 14 MFLOP/s after careful hand optimization. This is about an order of magnitude slower than a CRAY-1, but then it also costs one-tenth the cost of a CRAY-1. The Convex systems are air-cooled and are thus far easier to install and maintain.

3.4 Replication

3.4.1 Description of Replication

Returning to the basics of parallel processing, consider Figure 3.3. Suppose that instead of using a pipeline, additional processors are provided, each one of which took three ticks to do the required work, i.e. each one working at the same speed as the single processor in Figure 3.1, and hence all using the same technology. If enough processors are available (three in the example shown), then all the answers will be available simultaneously after three ticks.

This example can also be used to illustrate the different philosophies possible with replication. Each processor may have its own local memory —

[7]At the time of writing, there were in addition entirely non-technical political problems: the U.S. government was only permitting the export of supercomputers to a relatively small number of countries.

$$\begin{array}{llll}
tick_1 & \overset{in}{\longrightarrow} a_1; b_1 & \overset{in}{\longrightarrow} a_2; b_2 & \overset{in}{\longrightarrow} a_3; b_3 \\
tick_2 & c_1 := a_1 + b_1 & c_2 := a_2 + b_2 & c_3 := a_3 + b_3 \\
tick_3 & c_1 \overset{out}{\longrightarrow} & c_2 \overset{out}{\longrightarrow} & c_3 \overset{out}{\longrightarrow} \\
 & & & \\
processor & 1 & 2 & 3
\end{array}$$

Figure 3.3: Execution of $\vec{a} + \vec{b}$ on an array with three processors.

an example being a transputer array. Alternatively, the memory may be global (shared) — an example being the CRAY-XMP. All the processors may be executing the same instruction simultaneously ("lock-step" in computer parlance), or each processor may be running totally different programs. The former is frequently referred to as SIMD (Single Instruction, Multiple Data) and the latter as MIMD (Multiple Instruction, Multiple Data); this nomenclature is discussed in detail in Section 3.5. Communication is required between the processors; this can either be implemented using global memory (which requires complex hardware and/or software mechanisms to avoid memory contention problems[8]), by explicitly passing messages from processor to processor, or by a combination of these mechanisms. A transputer array implements a message passing system.

The performance degradation due to the start-up time of the pipe-line has its analogy with processing arrays. Three different mechanisms can be identified which degrade the performance of the system, namely

- *Scheduling*: The efficiency with which the available work is divided up among the processors. This is also known as load balancing.
- *Synchronization*: Synchronizing the different processors so that operations take place in the correct order.
- *Communication*: Different processors almost always need to communicate results at some stage of the algorithm; time will be spent in performing this communication that could have otherwise been spent computing.

Which effects are important will depend on the system used. Hockney and Jesshope [HJ88, Section 1.3.6] develop a characterization that incorporates all the above effects following the (r_∞, $n_{1/2}$) model used to describe SIMD performance. Another frequently used one parameter model — used in this

[8]Two or more processors simultaneously trying to access the same address in memory.

thesis — incorporates all the above effects into a parameter termed speed-up, S. Speed-up tells the user how much faster his algorithm will run on N processors than on one, which is really the fundamental issue of importance for the user. It 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.

$$S = \frac{T_s}{T_p} \tag{3.3}$$

S has an upper bound of N.

Another parameter, namely efficiency, ϵ, tells the user how efficiently the N processors are being used. It is simply the speed-up normalized by N [9]

$$\epsilon = \frac{S}{N} \tag{3.4}$$

ϵ is normally bounded[10] from above by 1. For systems where the degradation is primarily due to communication, such as the transputer array to be studied, this can be explicitly represented by [FJL*88, Chapter 3]

$$T_p(N) = \frac{T_s}{N}(1 + f_c) \tag{3.5}$$

where f_c is the fractional communication overhead. S may be written as

$$S = \frac{N}{1 + f_c} \tag{3.6}$$

There has been some debate on what times should be compared in the definition of S, since the optimal serial algorithm is not necessarily the optimal parallel algorithm. This point will not be developed further, since it is of no concern for the parallel algorithms described in this thesis. Note also that

[9]Speed-up and efficiency have become standard terms in the applied parallel processing community, due largely to books such as [FJL*88], published relatively recently. Karp and Flatt have recently introduced another metric, the experimentally determined serial fraction, and claim this to be superior to the speed-up / efficiency characterization [KF90]. The present author has not investigated their work in detail.

[10]From time to time, a paper will appear showing an efficiency exceeding 100%. The author suggests that this is possible, but only under extreme circumstances and due to very subtle design details of the computers. For example, a very short code with very small data requirements could just exceed the 4kB of fast, on-chip RAM available on one transputer — but with two transputers, the code and data (divided by two) now fit into the 4kB of each transputer. Using exclusively on-chip RAM produces an immediate speed-up of around three, so a code with minimal communication requirements is likely to show a speed-up of about six and an efficiency of about 300% when run on two transputers. This is speculation and the author has not actually observed this effect, but it is theoretically possible. It will not of course be repeatable on a general MIMD array.

the use of T_p here is not exactly the same as in [HJ88, 1.3.6], but agrees with [FJL*88] and [Mod88, p36].

3.4.2 A Local Memory MIMD Example — Transputer Arrays

The example given here is an example of a local memory MIMD array, using transputers as processing elements. The specifications given are germane to the transputers readily available at the time of writing; the performance ratings given will of course become dated. However, INMOS, the company who manufactures the transputer, has recently announced a new generation of transputers with greatly improved specifications. This should ensure the longevity of transputer based array processors; the code written for the transputers described here will also run on the new generation of transputers. With MIMD processors, computer languages and extensions to existing languages were very specific to particular computers at the time of writing. Hence the language that the algorithms described in this thesis were written in (Occam 2) is discussed and compared to the other main contender for the transputer, Parallel FORTRAN.

The transputer is a type of processor chip incorporating a CPU, memory and communication links. When introduced in 1984, the extensive exploitation of VLSI (Very Large Scale Integration) technology distinguished the transputer from its competitors. Several variants of transputer are available; at the time of writing, the T400, T414, T425, T800, T801 and T805. This description concentrates on the 800 class (the last three), which is the class of transputer useful for numerical analysis, since a floating-point processor has been added to the chip. (This is on the *same* chip as the CPU, memory and communication links). In particular, the T800 is described, since the transputer array available to the author, the Massively Concurrent Computer (MC^2), used the T800.

The transputer is a 32-bit RISC[11] design. See Figure 3.4 for a schematic of the floating point chip. The 20 MHz T800 transputers used in the MC^2 are specified at 10 MIP/s and 1.5 MFLOP/s. The four "links" provide bidirectional communication either with a host processor or with other transputers. The link speed is 20 Mbit/s. Each link has a DMA channel into the memory system — this does slightly reduce the memory-bandwidth to the CPU[12], but not significantly. All components execute concurrently; each of the four links and the floating point processor can perform useful work while the processor is executing other instructions. The link concurrency is exploited in the

[11]See glossary.

[12]The rate at which the CPU can get data to and from memory.

applications considered in this thesis. 4kB of on-chip RAM is also provided, but this is not significant for the applications considered in this thesis.

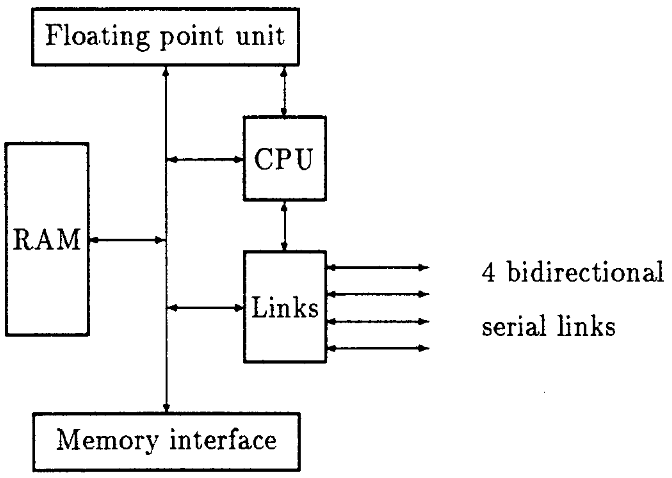


Figure 3.4: Floating point transputer. After [INM89, p.31]

When introduced, the transputer was a very powerful processor in its own right — benchmarking using NEC2 showed that one T800 transputer was slightly faster than a MICROVAX II rated at 0.9 VAX MIP/s[13] [lRBdPC88]. When first introduced, one application of transputers was as an "accelerator" board hosted in a PC or VAX system. Programs were developed and filed using the normal host operating system — on a PC, DOS. The compilers and compiled code then ran on the transputer, permitting a great increase in computational power for especially PC users[14]. This application has decreased in importance as the host systems have increased in speed.

However, it is the use of the transputer as an *element of a processing array* that is of significance to this thesis. From its inception, the transputer was designed as an element of a parallel computer, so many critical issues such as communication were efficiently addressed using special hardware. Developing a transputer based parallel computer required primarily developing the inter-transputer switching network; on simpler machines this was hardwired, whereas more complex machines provided software control over the switches, permitting different interconnection topologies to be implemented for different applications. More detail on issues related to interconnecting

[13]See Section 2.2.

[14]Information regarding the present commercial availability of transputer hardware is available in [Dav90b, p.9]

processors are considered in Section 3.6. Details of the specific switching strategy adopted by the designers of the MC^2 are available in [Vil89].

At the time of writing, the transputer array at the University of Stellenbosch contained 64 T800's, 16 with 1MB of RAM, 29 with 2MB, 14 with 4MB, 4 virtual memory boards and 1 special purpose board. The MC^2 computer is nominally rated at 100 MFLOP/s, giving a maximum throughput in the same order as a CRAY-1[15]. However, to achieve anything approaching this will requires very careful coding to fully exploit the hardware. Such issues will be discussed in detail in Chapters 4, 5 and 6.

Just prior to submission of this thesis, INMOS recently released full preliminary technical information on the next generation of transputer, the T9000 [INM91]. The T9000 is specified at 200 MIP/s or 25 MFLOP/s. It has a 64 bit floating point unit on chip (similar to the T800). The four serial links each provide a bidirectional bandwidth of 20 MByte/s, as opposed to the 20 MBit/s of the T800. The T9000 also incorporates a 16 kB instruction and data cache, which should be far more useful for general applications that the 4 kB on-chip RAM of the T800. Also described is the C104 packet routing switch, a low-latency (less than $1\mu s$ packet latency) 32 by 32 crossbar switch that should greatly simplify the design of large transputer arrays. INMOS plan to release the T9000 and C104 in 1992.

Thus far, the technical details of the transputer have been discussed. Now it is time to consider the languages available for coding algorithms to run on transputer arrays. The algorithms described in this thesis were implemented in Occam 2. For transputer applications, the user is faced with on the one hand, the group of general purpose procedural languages with parallel extensions, namely Parallel FORTRAN, Pascal, Modula 2 and C, and on the other, Occam (which is also a procedural language but is based on the fundamental requirement to express parallelism as naturally as possible).

Occam[16] implements the concept of Communicating Sequential Processes, which was introduced by Hoare at Oxford University in the mid-1970's [Hoa85]. This concept views a computational process as a group of sequential processes which have to communicate with each other at certain times. The

[15]An important question for potential users of a system is the financial cost thereof. While exact figures have not been publicly released, the $MC^2/64$ computer described in this thesis cost about two hundred thousand U.S. dollars. At the time of writing, such a system would cost rather less: since it was delivered, the price of transputers has dropped dramatically, and the memory chips required have also dropped in price. It has also been noted that slightly slower — and rather cheaper — memory could have been used; see [Dav90b, p.17].

[16]Occam takes its name from the minimalist philosophy of the 14th century philosopher William of Occam, from which derives Occam's Razor — "Entia non sunt multiplicanda praeter necessitatem", which translates as "Entities should not be multiplied beyond necessity". This can be loosely paraphrased as "Seek the simplest solution" [Gal90, p.3].

transputer was designed to implement this philosophy.

Programming in Occam requires rather more effort than FORTRAN for some tasks; the I/O routines are not as comprehensive as the FORTRAN standard[17] and Occam does not support complex numbers, requiring the user to write the necessary procedures. These disadvantages are outweighed by the advantages of the very simple and efficient handling of inter-processor communication in Occam, and the excellent integrated development environment, the Transputer Development System (TDS)[18]. Programs are developed using a "folding editor". Briefly, the concept is that sections of code are put into folds — similar to taking a piece of paper and folding part of it away . A fold does not affect program operation at all, but greatly improves readability, since a section of code devoted to input can be put in a fold and then the fold marked "input section". It was used consistently in developing the codes described in this thesis, and greatly aided proper "top-down" design. The post-mortem (i.e. not interactive) debugger was also extensively used for debugging. At the time of writing, a version of FORTRAN supporting debugging had just become available; when the work was initiated, the FORTRAN compiler had no debugging support at all. A detailed description of the TDS will be found in [Dav90b, p.10-11].

The TDS runs under DOS but uses its own filing system. A stand-alone Occam 2 compiler is provided with the Occam Toolset. This also incorporates a post-mortem debugger similar to that of the TDS. An *interactive* debugger is also provided with the Toolset; it is restricted to codes running on one transputer. Unfortunately, when tested by the author, the interactive debugger worked satisfactorily for simple test codes but failed to work at all for real codes, such as code stubs of PARNEC that the author was debugging.

A parallel FORTRAN compiler, Parallel FORTRAN Version 2.0 [Par88], was also available to the author, and initially it had been planned to develop the codes in FORTRAN. However, the author encountered very serious problems with the Parallel FORTRAN compiler; codes would often run once and then fail on a second pass, and even very simple test codes would often not run reliably. The Parallel FORTRAN package did not support any debugging at all. The problems with Parallel FORTRAN were so serious that the approach outlined in [Dav90b, p.11], namely using the Occam Toolset's ability to mix alien languages and Occam, was eventually rejected[19]. It had been hoped to develop an Occam "harness" which would handle the inter-

[17]An example of this is that the task of reading in a set of real numbers, separated by blanks, requires the user to write a special routine to implement the necessary parsing, whereas FORTRAN handles this implicitly — at least, FORTRAN 77 does.

[18]The latest version available at the time of writing was the TDS3.

[19]This is discussed in more detail in Section 6.3.

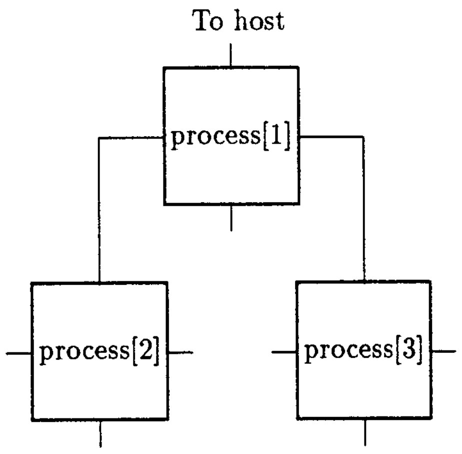
processor communication and interface the Occam matrix solvers with the rest of the FORTRAN code. This would have been a very elegant solution if the FORTRAN compiler had worked reliably. The *concept* remains attractive, since it would be easier to apply to improve the performance of existing codes. This would permit the bulk of the code to be retained in the original language, and only the computationally intensive parts of the program would be parallelized in Occam. As the compilers improve, it may well become a practical method.

This discussion has been presented to motivate the author's choice of Occam as the implementation language for the algorithms described in this thesis. Different parallel processors support different languages, and this discussion relates thus specifically to a transputer based array processor. It should be appreciated that the software technology available to the author for the work reported in this thesis was not of the level that one finds on PC's, VAXes and workstations, either in terms of functionality (for example, interactive debuggers) or reliability. Despite these problems, the linear equation solvers and moment method codes developed by the author are functional and reliable.

Transputer hardware and software has been discussed in this section. Now it will be shown how the previously discussed vector addition example would be parallelized on an array of three transputers; see Figure 3.5. The connections (links) are shown from processor 1 (running `process[1]`) to processors 2 and 3 (running `process[2]` and `process[3]` respectively). Unused links are left unconnected. Processor 1 is acting as the "master" transputer in this example, and is also connected to the "host" (typically a PC or VAX). This example uses pseudo-code to be defined in Section 3.7; note that each processor has a process "mapped" onto it. These processes can be debugged by simulating all three parallel processes on one actual transputer. Once the program has been debugged, the processes are mapped onto physical processors to obtain the speed-up required. This interchangeability of "simulated" and "real" parallelism is due to the fundamental design of the transputer and makes the transputer particularly attractive for developing code to run on very large arrays; the development and debugging can all be done on one transputer and only the final speed-up tests need be performed on the actual array.

3.5 Classification of Parallel Computers

The proliferation of parallel hardware has given rise to a need for a systematic classification of parallel computers. One of the earliest was Flynn's classification [Fly72]; it has retained its utility to the present. He based his



```
process[1]:
begin
  par
    send a2,b2 to process[2]
    send a3,b3 to process[3]
    c1 := a1+b1
  par
    receive c2 from process[2]
    receive c3 from process[3]
end{process[1]}

process[2]
begin
  receive a2,b2 from process[1]
  c2 := a2+b2
  send c2 to process[1]
end{process[2]}

process[3]
begin
  receive a3,b3 from process[1]
  c3 := a3+b3
  send c3 to process[1]
end{process[3]}
```

Figure 3.5: Vector addition parallelized for three transputers

taxonomy on the multiplicity of data and instructions. This leads to four distinct classes of machine:

- *Single Instruction, Single Data (SISD)*: A system with one instruction stream operating on one datum at a time.
- *Single Instruction, Multiple Data (SIMD)*: A system with one instruction stream, where each processing element operates on different data in lockstep with the global instruction stream.
- *Multiple Instruction, Single Data (MISD)*: A system with multiple instruction streams operating on one datum.
- *Multiple Instruction, Multiple Data (MIMD)*: A system where each processing element operates independently with potentially different instructions on different data.

Within the SISD classification, most machines based on the von Neumann architecture[20] reside, such as the ubiquitous PC, most of the VAX range, and most workstations. The ICL DAP array, briefly mentioned previously, is an example of a SIMD system. Arrays such as the transputer array are clearly MIMD systems. MISD systems are primarily of theoretical interest [HJ88, p.57].

Flynn's taxonomy has a very important advantage: it is very simple. It is also descriptive — up to a point. Thus it is very common to encounter the terms SIMD and MIMD in the literature. However, Flynn's taxonomy suffers from the problem of over-simplification for some applications. (The reason for this is probably that the work that his classification was based on dates to the mid-1960's, when parallel computers in general were primarily of theoretical interest). Firstly, pipelining does not fit into the above scheme at all comfortably. While pipelined machines are often grouped under the SIMD classification, this ignores the fact that pipelining derives from an overlapping in time of operations, not a replication of processing elements. Since at any

[20]"von Neumann" refers to the architecture proposed by von Neumann for the computer built at the Princeton Institute for Advanced Study in the later 1940's; a von Neumann architecture refers loosely to any computer not employing concurrency [FJL*88, p.491]. In the parallel processing literature, the term sometimes carries a slightly pejorative sentiment. It is important to remember von Neumann's crucial contribution of the concept of the stored program, which was central to his thinking, whereby the computer program is also data, stored in the computer's memory. This idea is so ubiquitous nowadays it is difficult to conceive the methods used to program the world's first electronic computer, the ENIAC, which was re-programmed by resetting switches and replugging cables. A popular account of von Neumann's pioneering work on computers may be found in Regis's history of the Princeton Institute for Advanced Study [Reg87, Chapter 5].

one time, different parts of an operation are being performed on different data, one could possibly argue that pipelined machines belong in the MIMD class. Another argument is that different operations are being performed on the *same* data in the pipeline, so pipelined machines should be classified as MISD systems. It is at best an unresolved question. In the author's view, pipelining should best be treated as a separate entity, orthogonal to the instruction/data stream classification, since both a SIMD and a MIMD computer can incorporate pipelining in the processing elements[21].

The second point, and the more significant in this thesis, is that the classification ignores the question of memory. Particularly with MIMD systems, the question of whether memory is local (i.e. may only be accessed directly by the processor that it is attached to) or global (i.e. all processors have access to the same memory) can totally change the suitability of a parallel algorithm.

Nonetheless, Flynn's taxonomy, if extended slightly to also give information about pipelining (if present) and memory type, provides a sound basis for further sub-division. It should be viewed as analogous to the classification of animals by family in the familiar biological and zoological taxonomy — further sub-division is necessary for an accurate description. Hockney and Jesshope provide such a further sub-division [HJ88, Section 1.2.5]. As an example, the CRAY-1 is classified as a single instruction stream computer, with pipelined execution units — this is a parallel unicomputer in Hockney and Jesshope's taxonomy. It can be further subdivided into the class of such parallel unicomputers with vector instructions with special purpose pipes.

Hockney and Jesshope also develop an algebraic-style structural (ASN) notation [HJ88, Section 1.2.4], which allows a very detailed description of a computer, down to data path widths, clock speeds, memory interleaves etc. This is very elaborate for most purposes but is useful as a succinct but complete (albeit cryptic) description of a machine. This ASN notation is not used in this thesis.

3.6 Interconnection Topologies

With a local memory processing array, each processor must communicate with other processors at some stage of the algorithm. Such communication is obtained via the interconnect topology. This may be done dynamically, i.e. during program execution, or statically, i.e. set up before the run and left

[21]Intel's i860 chip, suitable for use in a MIMD array, uses pipelined arithmetic units to obtain an impressive peak throughput of a claimed 80 MFLOP/s.

unaltered during execution[22]. Consider a general array, consisting of *nodes* — the node is the processor plus memory[23]. For a general system with P inputs and Q outputs,there are P^Q possible mappings (including $P!$ one-to-one and the rest one-to-many), and these can be implemented using a full crossbar switch with PQ switches[24]. This rapidly becomes very expensive. An N node transputer array contains $4N$ inputs and $4N$ outputs, since each transputer has four bi-directional links. A detailed treatment of switching networks requires permutation theory and can be found in [HJ88, Section 3.3].

Hafner [Haf89] has described some popular interconnection topologies. Probably the most generally encountered with transputer arrays are pipelines, rings, meshes, trees and hypercubes. Some examples are shown in Figures 3.6, 3.7 and 3.8. The "dimension" of a hypercube has the conventional geometrical interpretation for hypercubes up to dimension 3. Graph theory can be used to obtain certain important properties of these topologies; these are summarized in Table 3.1. Diameter means the maximum number of links required to connect any two nodes[25]. Wraparound means that the left-most and right-most columns of processors in the lattice are connected together, as are the upper-most and lower-most rows. See [Mod88] for a review of graph theory.

The topologies used in this thesis were the binary tree and the mesh. The former was used for the parallel conjugate gradient algorithm to be described in Chapter 4, and the latter for the parallel LU algorithm described in Chapter 5. The motivation for each choice is presented in the relevant chapter. Both are of course implementable on a general transputer array permitting any link on any processor to be connected to any link on any other processor.

3.7 A Pseudo-code Notation

For documenting the algorithms to be presented in the rest of this thesis, it is useful to introduce a pseudo-code notation. Some of this notation is very loosely based on Occam 2 [Gal90]. The notation emphasizes *readability*. The notation is an extension of that defined in [FJL*88, Appendix A]; a

[22]Programs written for the transputer require an explicit mapping of "channels" (the software communication abstraction) onto actual physical links at compile time, hence transputer arrays are always statically switched.

[23]The following results are summarized from [HJ88, p.259–61].

[24]Note in this context that the Communicating Sequential Processes (discussed in Section 3.4.2) concept allows only one process to communicate with one other process at a time, i.e. only one-to-one mappings are permitted.

[25]Vertices and edges, respectively, in graph theory.

Topology	*Order*	*Number of processors*	*Diameter*	*Comments*
Binary tree	d	$N = 2^{d+1} - 1$	$2d$	
Hypercube	d	$N = 2^d$	d	
Lattice Mesh	-	N	$2\sqrt{N} - 2$	No wraparound
Lattice Mesh	-	N	$2\lfloor \frac{1}{2}\sqrt{N} \rfloor$	Wraparound
Linear (pipeline)	-	N	$N - 1$	
Cyclic (ring)	-	N	$\lceil N/2 \rceil$	

Table 3.1: Properties of common interconnection topologies

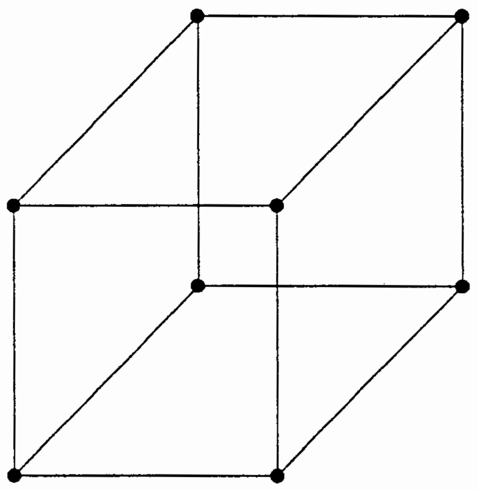


Figure 3.6: Interconnection topologies — hypercube dimension 3

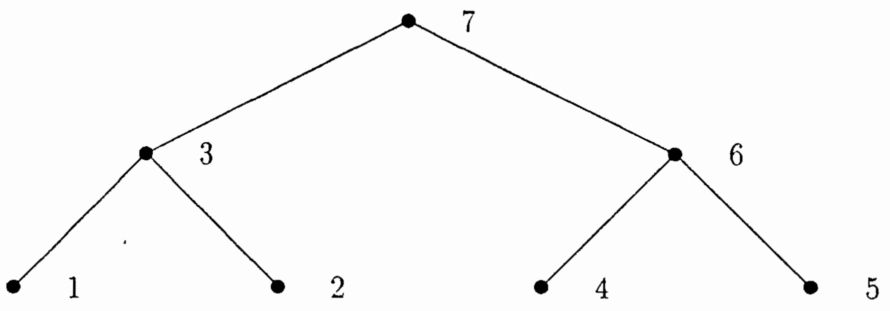


Figure 3.7: Interconnection topologies — binary tree depth 2

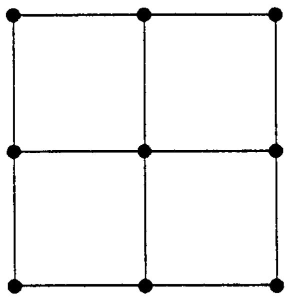


Figure 3.8: Interconnection topologies — mesh (lattice) without wrap-around. $N = 9$.

notation similar to but not as extensive as that used in this thesis will be found in [vdVB89]. Pseudo-code will also be used as part of the software re-engineering procedure to be described in Section 6.4.

Firstly, the assignment statement is defined:

```
a := 1
```

This assigns the value of the right hand side of the equation to the variable on the left hand side.

Then a `begin-end` construct similar to that of Pascal is introduced:

```
begin
  code stub
end
```

Everything within the `begin-end` demarcation is executed sequentially, unless the following `par` construct is encountered:

```
par
  code stub1
  code stub2
  .... other code stubs
end{par}
```

Then `code stub1` and `code stub2` and `....other code stubs` are executed *concurrently*. These code stubs can of course consist of `begin-end` constructs, which can in turn consist of nested `begin-end` and/or `par` constructs. The curled braces { ...} are used to demarcate a comment.

The par construct indicates concurrent execution. This may be obtained by time-sharing of resources such as the CPU[26] or actual parallel hardware, such as multiple processors[27] or hardware support for concurrent execution, for example concurrent computation and communication[28]. The par construct is taken directly from Occam. Occam does not have an explicit begin-end construct; the equivalent is the SEQ construct, which looks as follows:

```
SEQ
  code stub
```

Note that there is no end statement in Occam. This thesis will use the begin-end construct in preference to the SEQ construct.

A conditional loop is required; this is provided by the while construct:

```
while (some Boolean expression)
  code stub
end{while}
```

This loop evaluates some Boolean expression; if FALSE, it behaves as a SKIP construct (to be defined), i.e. does nothing, and the loop terminates. If TRUE, code stub is executed , and then some Boolean expression evaluated again. Obviously some action must ultimately change the value of some Boolean expression, or the loop will never terminate. A repeat for a given number of times construct is useful[29]; this is provided by the repeat for construct:

```
repeat for i = start.index to stop.index
  code stub
end{repeat}
```

The section of code in code stub is executed for i from start.index to stop.index (both inclusive); if start.index exceeds stop.index it behaves as a SKIP construct (to be defined), i.e. does nothing, and the loop terminates.

A conditional construct is required; this is the if-then-else construct:

[26]Used on the transputer to "simulate" parallelism during code development, as discussed in Section 3.4.2. Note that by no means all MIMD processors support such a concept — on such systems the code development and debugging must then be performed on the real array.

[27]In the context of transputers, multiple transputers with some form of interconnection network.

[28]Such support is provided on the transputer; the links and the floating point unit were designed to operate concurrently.

[29]This can actually be implemented using the while construct but it is convenient to define this additional construct.

```
if (some Boolean expression)
  then code stub1
  else code stub2
end{if}
```

If the Boolean expression is TRUE, then code stub1 is executed; if FALSE, then code stub2 is executed instead.

A SKIP process is also defined that does nothing; it is derived from Occam and permits explicit coding of a condition that requires no action. An example of its use is to convert the **if-then-else** to effectively an **if-then** construct by making code stub2 in the **if-then-else** a SKIP.

Communication will be written out in English:

```
begin
  receive z from process[k]
  send z to process[s]
end
```

It is not always known on what channels the data will be available on first, so the alt construct is defined:

```
alt
  receive from process[k]
    code stub1
  receive from process[l]
    code stub2
end
```

If process[k] is ready with data first, then code stub1 is executed; otherwise if process[l] is the first ready, code stub2 is executed.

A procedure is a group of statements; it is generally given a name and a list of arguments may be passed when instantiated.

Constants and variables will not be explicitly typed (i.e. declared); their type will be clear from the context[30].

A process construct is declared:

[30] Occam is a strictly typed language; this means all variables and constants must be explicitly declared. While initially somewhat verbose to someone used to FORTRAN's default typing, the advantages for generating checkable code are quickly appreciated and the FORTRAN default typing is seen as the very dangerous construct that it is after a number of Occam codes have been written. Some versions of FORTRAN can be forced to obey strict typing.

```
process[s]
  begin
    code stub
  end
end{process[s]}
```

The process will in general be numbered or named in some fashion; this is done by assigning a number or name to s. Such a process will run on one processor. (When coded, the process may contain procedures, be part of a larger procedure or be the main procedure itself; generally it is only parts of the whole code that need explicit documentation using pseudo-code and the presence of a larger structure is assumed where necessary).

Finally, a placed par construct is defined (also taken from Occam). The placed par is used to indicate that the different processes are placed on different processors.

```
placed par
  processor1
    some process
  processor2
    some other process
  other processors with associated processes
    ....
```

No go-to *construct is defined for the excellent reasons laid down by the "father" of structured programming, Dijkstra [Dij76].*

The use of this notation will be illustrated by the examples given in Chapters 4, 5 and 6.

3.8 Amdahl's Law

Any work on parallel processing should note *Amdahl's law* [FJL*88, Section 3.6], which states that if an algorithm contains both a serial and a parallel part, the relative time taken by the serial part increases as parallelization reduces that of the parallel part, and a law of diminishing returns holds: further parallelization has increasingly little influence on run-time. While this observation is perfectly true, for many problems the ultimate aim is to increase the problem size that can be handled. Thus as more parallelization is made available, larger problems are tackled and the overall serial/parallel split remains fairly constant. Hence Amdahl's law is not the immutable barrier that it has often been held to be, and it will be shown clearly in Chapters 4, 5 and 6 that for the electromagnetically large problems of interest, it is not an obstacle. Gustafson arrived at very similar conclusions following work on a variety of problems in computational mechanics and fluid dynamics [Gus88].

3.9 Parallel Algorithms

The rest of this thesis concentrates on the development of parallel algorithms. Several excellent books appeared during the course of the author's research that provide a good treatment of the basic ideas[31], but not of the specific algorithms considered in this thesis. For the technological background, Hockney and Jesshope's second edition [HJ88] is invaluable. For the development of algorithms and establishing the efficiency thereof, Fox *et al.* [FJL*88] provides a very sound foundation. Although based on their experiences with a hypercube type machine with 128 nodes at Caltech, the book is sufficiently general that the principles developed are applicable to other types of machines. (Note that many books available with "parallel" or "concurrent" somewhere in the title are frequently aimed at a very specific computer, and the methodologies developed may only be applicable to that machine). Fox's book is particularly suitable for the solution of large-grained problems. Another book by Bertsekas *et. al.* [BT89] is particularly strong on the mathematical analysis of algorithms. Modi's book [Mod88] is also a most useful text, being directed specifically at parallel matrix algorithms, but not the specific algorithms considered in this thesis[32].

Parallel algorithms are also published in a number of specialist journals: these include *Parallel Computing*, *Journal of Parallel and Distributed Computing*, *Concurrency: Practice and Experience* and *IEEE Transactions on Parallel and Distributed Systems.* Other established journals in the field of applied numerical analysis also publish work of interest; examples are *Communications of the ACM*[33], *SIAM*[34] *Journal on Scientific and Statistical Computing*, *SIAM Review*, and *Computer Physics Communications*..

[31]As already mentioned, the author's initial research started before these books were available.

[32]A very recently published book on solving linear systems, only available to the author following examination of his thesis, is that of Dongarra *et al.* [DDSvdV91]. It concentrates on vector and shared memory computers, whereas this thesis concentrates on local memory computers.

[33]Association for Computing Machinery.

[34]The Society for Industrial and Applied Mathematics.

3.10 Conclusions

In this chapter a suitable introduction has been provided for the use of parallel computing to solve computational electromagnetic problems. A brief review of the history of parallelism and the general principles has been provided. The two main types of parallelism encountered, viz. pipelining and replication, have been described and examples given of present day parallel computers that embody those principles. The question of quantifying the performance of a parallel algorithm has been addressed. The classification of parallel computers has been discussed. A pseudo-code notation has been introduced that will be used to document the parallel algorithms in the rest of the thesis. A brief review of the *relevant* literature on parallel algorithms has been presented. Some of the work presented in this chapter has been published as a tutorial paper [Dav90b].

It should be clear from the discussion in this chapter that 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 carefully select, analyse and implement suitable parallel algorithms. It is this problem that occupies Chapters 4, 5 and 6.

Chapter 4

A Parallel CG Algorithm

4.1 Introduction

Iterative methods offer one method of solving the system of linear equations

$$[A][x] = [b] \tag{4.1}$$

where [A] is a non-singular known matrix, [b] a known vector and [x] a required (and unknown) vector. Most MoM codes in the past have used LU decomposition followed by forward and backward substitution. (This method is addressed in more detail in the next chapter). Iterative methods for solving the system of linear equations have only attracted much attention over the last decade, since previously, for the size of problems (number of unknowns) being solved, the LU method was quite sufficient. The iterative methods are attractive for two reasons: firstly, with a method with a monotonically decreasing error such as the CG method, the iterations can be stopped once a specified error criterion has been met, and secondly, iterative methods do not suffer from the accumulation of round-off error that compromises the accuracy of the LU method when the matrix $[A]$ is ill-conditioned.

In this chapter, a parallel conjugate gradient algorithm is proposed, analyzed theoretically, implemented and tested on a transputer array, using the binary tree interconnection topology described in Chapter 3. Measured results for speed-up and efficiency are compared to the theoretical predictions. The parallel algorithm uses as its basic building block two parallel matrix-vector products, so efficient parallel matrix-vector product algorithms are investigated in detail. Important fundamental methods for optimizing the efficiency of a parallel algorithm are discussed. The basic parameters used in the theoretical analysis are established using benchmarking. The problem of developing a *general* configuration description, able to handle any depth of binary tree, is considered and a solution briefly discussed. The problem of terminating a parallel algorithm is considered and a solution given for the CG algorithm. It is shown that the proposed CG algorithms scales essentially with the number of rows per processor[1].

The question of the rate of convergence of the CG algorithm for electromagnetically large, complex problems is deferred to Chapter 6.

4.2 Iterative Methods

Iterative solvers for the solution of equation (4.1) can be subdivided into two categories, namely stationary and gradient methods[2]. With a stationary iterative method, succeeding error vectors can be written as

$$[e_{k+1}] = [M][e_k] \tag{4.2}$$

where k refers to iteration number and $[e_k]$, the error vector, is defined by

$$[e_k] = [x_k] - [x] \tag{4.3}$$

For the Jacobi method, the $[M]$ matrix is given by

$$[M]_{Jacobi} = [L] + [U] \tag{4.4}$$

where

$$[L] = \begin{bmatrix} 0 & 0 & \dots & 0 \\ -a_{2,1} & 0 & \dots & 0 \\ \cdot & \cdot & \cdot & \cdot \\ -a_{M,1} & \dots & -a_{M,M-1} & 0 \end{bmatrix} \tag{4.5}$$

and

$$[U] = \begin{bmatrix} 0 & -a_{1,2} & \dots & -a_{1,M} \\ 0 & 0 & \dots & -a_{2,M} \\ \cdot & \cdot & \cdot & \cdot \\ 0 & \dots & 0 & 0 \end{bmatrix} \tag{4.6}$$

Note that $[L]$ and $[U]$ as defined here are *not* the same as the $[L]$ and $[U]$ resulting from a LU decomposition.

The matrix [M] remains constant for each iteration, hence the name "stationary".

It can be shown for the Jacobi method that a necessary and sufficient condition for convergence to the correct solution $[x]$ is that all the eigenvalues of $[A]$ should lie within a unit circle centered on $1 + j0$ when plotted on the complex plane. This condition is not necessarily satisfied by an arbitrary

[1]Scaling is discussed in more detail in Section 4.4.

[2]This section is based on [Jen85, Chapter 6].

matrix. Other examples of stationary methods are the Gauss-Seidel method [Jen85, p.183] and the various relaxation methods such as successive over-relaxation [Jen85, p.186].

The gradient methods use a different approach. One may write succinctly that

$$[e_{k+1}] = [M_k][e_k] \tag{4.7}$$

where the $[M]$ matrix is now different on each iteration. However, this is not the form in which the methods are usually given.

Define a residual

$$[r] = [b] - [A][\overline{x}] \tag{4.8}$$

where $\overline{x}$ is the approximation to $[x]$. If the matrix $[A]$ is symmetric and positive definite (i.e. $[x]^T[A][x] > 0 \ , \ \forall \ [x] \neq 0$), then it may be shown that its inverse is also symmetric and positive definite. If $[A]$ is not positive definite, it can always be made so by solving the modified system of equations

$$[\,\overline{A}\,][x] = [\overline{b}] \tag{4.9}$$

where $[\,\overline{A}\,] = [A]^T[A]$ and $[\overline{b}] = [A]^T[b]$. The transpose operator (superscript T) must be interpreted in the formal sense as the Hermitian transpose if the elements of $[A]$ are complex. One drawback with this procedure is that if the matrix $[A]$ is ill conditioned, then the matrix $[\,\overline{A}\,]$ is very much more ill-conditioned than $[A]$, and this can adversely affect the rate of convergence for large systems[3]. Nonetheless, the results in Chapter 6 show that the rate of convergence is satisfactory for a typical matrix equation generated by a MoM discretization.

Consider the iteration formula

$$[x_{k+1}] = [x_k] + \alpha_k[d_k] \tag{4.10}$$

where α_k is chosen to minimize the error function

$$h_k^2 = [r]^T[A]^{-1}[r] \tag{4.11}$$

and the direction vector $[d_k]$ is discussed below. The positive definite nature of $[A]^{-1}$ ensures that the function $h_k^2 \geq 0$; if $h_k^2 = 0$, then $[x_{k+1}] = [x]$ and the

[3]The example given in Jennings [Jen85, p.221] shows how this procedure effectively squares the condition number of the matrix used in the example. The condition number of a matrix is a measure of the sensitivity of the solution of the system in equation (4.1) to small variations in $[x]$. It will be defined formally in Chapter 5. The condition number can be shown to be the square root of the ratio of the maximum to minimum eigenvalues. The eigenvalue clustering affects the rate of convergence of the conjugate gradient algorithm in particular [Jen85, p.217]. Thus "spreading" the eigenvalues by squaring the condition number decreases the rate of convergence.

system of equations has been solved. The subscript k refers to the iteration number.

Substituting for $[r]$ from equation (4.8) into equation (4.11), it may be shown that h^2 is quadratic in $[\overline{x}]$. Differentiating the resultant expression with respect to α_k to minimize the error function, it may be shown that

$$\alpha_k = \frac{[d_k]^T[r_k]}{[d_k]^T[A][d_k]} \tag{4.12}$$

The various gradient methods — principally the method of steepest descent and the method of conjugate gradients — differ in the choice of the direction vectors $[d_k]$. In the method of steepest descent, $[d_k]$ is chosen to be the direction of maximum gradient of the function at the point $[x_k]$; this can be shown to be proportional to the residual vector $[r_k]$. The result of this choice is an algorithm that tends to overshoot [Jen85, p.214].

The conjugate gradient method chooses as direction vectors a set of vectors $[p_0]$, $[p_1]$, etc., which are chosen to be a close as possible to the direction of steepest descent corresponding to the points $[x_0]$, $[x_1]$, etc., but with the overriding condition that they be *mutually conjugate* with respect to [A] and hence satisfy the condition

$$[p_i]^T[A][p_j] = 0 \text{ , } \forall\, i \neq j \tag{4.13}$$

The result of the orthogonality property is an algorithm that (in the absence of round-off error) converges after M steps, where M is the dimension of the problem. Thus, strictly speaking, the CG method is not an iterative method. However, it is usually *applied* iteratively — a check is kept on the normalized residual (the residual divided by the norm of $[b]$) and when it is sufficiently small, the iterative improvement stops.

Another way of viewing the CG method is that the solution is given in the space spanned by the basis functions $\{[b], [A][b], [A]^2[b], \ldots, [A]^{M-1}[b]\}$, with the weights emerging during the course of the iterations; a demonstration of this may be found in Sarkar [Sar86][4].

Regarding the convergence of the conjugate gradient method, it may be shown that the residual norm decreases monotonically [PM85b, p.16]. The *rate* of convergence is governed by the eigenvalue spread of the coefficient matrix. If the eigenvalues are clustered in a small number of groups, the convergence is very rapid. Conversely, if there are a large number of small but distinct eigenvalues, the rate of convergence is slow.

[4]This property is used by Sarkar [Sar86] to argue the fundamental difference between the "direct" CG method and the CG-MoM method, but as soon as the problem is discretized — which it must be to solve on a computer — a finite basis is implied and such an argument is invalidated [RP88]. See Section 2.6.

The CG method, extended for the general case of a matrix $[A]$ where it is not known if the matrix is positive definite, is as follows:

$$\begin{aligned}
[u_k] &= [A][p_k] && \text{Step 1} \\
\alpha_k &= \frac{\|[\overline{r}_k]\|^2}{\|[u_k]\|^2} && \text{Step 2} \\
[x_{k+1}] &= [x_k] + \alpha_k[p_k] && \text{Step 3} \\
[r_{k+1}] &= [r_k] - \alpha_k[u_k] && \text{Step 4} \\
[\overline{r}_{k+1}] &= [A]^T[r_{k+1}] && \text{Step 5} \\
\beta_k &= \frac{\|[\overline{r}_{k+1}]\|^2}{\|[\overline{r}_k]\|^2} && \text{Step 6} \\
[p_{k+1}] &= [\overline{r}_{k+1}] + \beta_k[p_k] && \text{Step 7}
\end{aligned} \tag{4.14}$$

with initial values

$$[r_0] = [b] - [A][x_0] \tag{4.15}$$

and

$$[\overline{r}_0] = [p_0] = [A]^T[r_0] \tag{4.16}$$

The initial value of $[x]$, viz. $[x_0]$, is frequently chosen as $[0]$, and this is the choice used by the author. If an approximation for $[x]$ is available — for example, from a geometrical optics solution — then this can be used as an alternate starting value. Rather interestingly, the rate of convergence is frequently a very weak function of $[x_0]$ [PM85a].

The approximate floating point operation (FLOP) count per iteration is shown in Table 4.1, retaining only the largest order term for each operation[5]. On the transputer, the time for a floating point addition or multiplication is identical, so these operations are not listed separately. (This will be shown in Section 4.7). Note that α and β 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. The FLOP counts of Steps 1 and 5 (the matrix-vector products) are of $O(M^2)$ whereas the other steps are of $O(M)$ — it is thus Steps 1 and 5 that will be parallelized.

[5]Because of this, a term -2M 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

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 4.1: FLOP count of conjugate gradient algorithm

4.3 A Parallel Matrix-Vector Product Algorithm

The computationally expensive parts of the CG method have been shown to be the two matrix-vector products, hence the parallelization of a matrix-vector product is investigated in detail as a precursor to the development of a parallel CG algorithm[6]. The parallel algorithms of this section are suitable for any local memory MIMD system.

Several paradigms for parallel processing have been identified[7]. The paradigm used in this thesis is domain decomposition, whereby the data is partitioned over the processing array. An example of this has already been shown in Chapter 3 for the problem of the addition of two vectors. In that case, the decomposition was simple and obvious — one element of each vector per processor. For the linear system solvers of interest in this thesis, there will be rather more unknowns than processors, so some form of *clustering*[8] is required. For the matrix vector product (and also the CG algorithm) the decomposition and clustering is relatively straightforward to describe. The LU algorithm considered in the next chapter has a rather more complex decomposition clustering strategy, and more formal mathematical methods will be introduced to extract the parallelism (i.e. perform the decomposition)

equation (4.39).

[6]Very little has been published on parallel iterative methods for full matrices. What has been published normally concentrates on sparse systems, for example [DDSvdV91, Chapter 7].

[7]Examples are domain decomposition, see [FJL*88, Chapter 1], and the processor farm, algorithmic parallelism and geometric parallelism [HJ88, Section 4.5]. Domain decomposition [FJL*88, Chapter 1] is the same as Hockney and Jesshope's geometric parallelism.

[8]*Clustering* is the grouping of matrix elements on a processor.

and analyze the clustering. The algorithms described in this chapter will be re-considered from this formal viewpoint in Chapter 5.

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 derived shortly. These viewpoints imply the following two possibilities for forming a parallel matrix-vector product:

- *Row-block decomposition*: Splitting up the matrix by row block, distributing these row blocks over the processor array, *broadcasting* the entire 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, distributing these column blocks over the processor array, *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 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 (row-block) matrix/vector product.

3. *Scatter*: This process is the reverse of *gather* in that it scatters a vector over the array so that each of the N processors has a different subvector of length M/N.

4. *Accumulate*: This process accumulates the partial inner products resulting from the column-block decomposition.

The row-block decomposition algorithm can be simply illustrated by considering a 2 by 2 matrix partitioned on two processors.

$$\left[\begin{array}{cc} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right] \tag{4.17}$$

where the upper elements are on processor 1 and the lower on processor 2. If partitioned by column block, it appears as follows:

$$\left[\begin{array}{c|c} A_{11} & A_{12} \\ A_{21} & A_{22} \end{array}\right] \tag{4.18}$$

It is important to note that this decomposition of the matrix by column block is the same as the decomposition of the transposed matrix by row block — this is the method used by the author to develop the parallelized conjugate gradient algorithm of the next section, and avoids having to form the matrix transpose required in the general form of the CG algorithm given in equation (4.14).

To carry out the matrix-vector product using a row-block decomposition, it is clear that every processor needs every element of the vector, thus the *broadcast* paradigm. This approach then forms the inner products

$$\left[\begin{array}{cc} A_{11} & A_{12} \end{array}\right] \left[\begin{array}{c} x_1 \\ x_2 \end{array}\right] \tag{4.19}$$

on processor 1 and

$$\left[\begin{array}{cc} A_{21} & A_{22} \end{array}\right] \left[\begin{array}{c} x_1 \\ x_2 \end{array}\right] \tag{4.20}$$

on processor 2 *in parallel*, i.e. concurrently. After the parallel product of matrix $[A]$ and vector $[x]$ has been formed, the situation is as follows:

$$\left[\begin{array}{c} (A_{11}x_1 + A_{12}x_2) \\ \hline (A_{21}x_1 + A_{22}x_2) \end{array}\right] \tag{4.21}$$

where again the upper element is on processor 1 and the lower on processor 2. To collect these, the *gather* paradigm is required.

The other two paradigms arise from the column-block decomposition; the parallel multiplication requires only certain elements per processor; in this case it will require element 1 on processor 1 and element 2 on processor 2. The column block decomposition forms the partial inner products

$$\left[\begin{array}{c} A_{11} \\ A_{21} \end{array}\right] \left[\begin{array}{c} x_1 \end{array}\right] \tag{4.22}$$

on processor 1 and

$$\begin{bmatrix} A_{12} \\ A_{22} \end{bmatrix} \begin{bmatrix} x_2 \end{bmatrix} \tag{4.23}$$

on processor 2 *in parallel.*

This is the *scatter* paradigm and is simply the reverse of *gather.*

Following the parallel matrix-vector multiplication using column-block decomposition, the final paradigm, *accumulate*, is required. After the multiply, the situation is as follows, with the left row on processor 1 and the right row on processor 2:

$$\left[\begin{array}{c|c} A_{11}x_1 & A_{12}x_2 \\ A_{21}x_1 & A_{22}x_2 \end{array} \right] \tag{4.24}$$

and these rows must be summed at the master node.

Given the restrictions of the transputer hardware — four links — a binary tree is a natural topology for this problem, for the following reasons. It is only necessary to communicate information to and from the 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[9] 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; in Figure 3.7 processor 7 would be the master and the other 6 would be worker processors. It is possible to use a ternary tree, but this does not map very conveniently onto available arrays, where the available number of processors generally follows some power of two.

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 (3.3) and (3.4). Consider first the broadcast process:

Define the time required to send one complex word[10] from one processor to another adjacent processor as t_{comm}. Then the time to send M words is simply Mt_{comm}. In the binary tree, the number of levels that must be traversed is d, so the time to send M complex words from the top to the bottom of the tree is Mdt_{comm}[11]. Here the set-up time of the communication process and the buffering time necessary to write the input to memory from

[9]See Section 3.6.

[10]A complex word consists of the real and imaginary parts; on a transputer, in single precision a complex word is 8 bytes and in double precision, 16 bytes.

[11]This discussion does not consider the use of *pipelining*, which could be used to reduce the communication time of the broadcast operation to around Mt_{comm}. Pipelining is discussed in detail in Section 5.10.

the input link and then write it out to the output link has been ignored. If the processors at each level of the tree are able to output the vectors to the next lower levels in *parallel* [12], the total time for the broadcast operation is then approximately the same as the time for one vector to traverse the tree from top to bottom, i.e.

$$t_{broadcast} = Mdt_{comm} \tag{4.25}$$

Pseudo-code for this is given in Figures 4.1 and 4.2.

```
begin{broadcast section:master}
  par
    send vector to lower left processor
    send vector to lower right processor
  end{par}
end{broadcast section:master}
```

Figure 4.1: Pseudo-code for broadcast: master process

```
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 4.2: Pseudo-code for broadcast: worker process

The timing equation for the gather process may be derived as follows. The information quantum for this case is M/N — the vector sections split over the array. The gather process starts at the bottom of the tree and works

[12]The transputer has such capabilities; see Section 3.4.2.

its way up. Referring to Figure 3.7, the first process, i.e. the communications $1 \rightarrow 3$, $2 \rightarrow 3$, $4 \rightarrow 6$ and $5 \rightarrow 6$ all proceed simultaneously, as do the next level of communications, $3 \rightarrow 7$ and $6 \rightarrow 7$. But note that this level now involves packets of length $3M/N$. In general then, for a tree of depth d, the total communication time will be

$$\frac{M}{N}[1 + 3 + 7 + ... + (2^d - 1)]t_{comm} \tag{4.26}$$

Manipulating the series into a geometric series, and using $N = 2^{(d+1)} - 2$, where N is now interpreted as the number of worker processors (i.e. excluding the processor at the top of the tree)[13], this simplifies to

$$t_{gather} = M[1 - d/N]t_{comm} \tag{4.27}$$

Pseudo-code for the gather operation is given in Figures 4.3 and 4.4. The indices of the various vector stubs are computed from the data distribution; referring to Figure 3.7, the processor numbers relate to the group of rows clustered on the processor, i.e. a vector of length 12 distributed over the binary tree in Figure 3.7 would have rows 1 and 2 on processor 1, rows 3 and 4 on processor 2 e.t.c.[14]. These details are not shown in the pseudo-code.

```
begin{gather section:master}
  par
    receive vector stub1 from lower left processor
    receive vector stub2 from lower right processor
  end{par}
end{gather section:master}
```

Figure 4.3: Pseudo-code for gather: master process

The scatter paradigm is the reverse of gather, except that instead of building up the vectors from the bottom of the tree up, at each stage combining the vector sections from the two lower processors and the processor's own

[13]The top-most processor, processor 7, is used purely to synchronize the processes, and does not perform any of the matrix-vector multiplication work itself. The result is that the maximum speed-up is thus $N - 1$ with N processors. As used in this chapter, N is the number of *worker* processors, thus the total number of processors is $N + 1$ For reasonable numbers of processors — 15, for example — the performance lost by thus not using the top-most processor is negligible, and the coding was simplified.

[14]As noted, processor 7 is performing synchronizing functions, and does not contribute to the matrix-vector product. Thus $N = 6$ in this case.

```
begin{gather section: worker}
  if (not at bottom of tree)
    then
      par
        receive vector stub1 from lower left processor
        receive vector stub2 from lower right processor
      end{par}
    else SKIP
  end{if}
  form new vector stub from vector stub1 & stub2 & own vector stub
  send new vector stub to higher processor
end{gather section: worker}
```

Figure 4.4: Pseudo-code for gather: worker process

local vector section, the vector now "trifurcates"[15] from the top of the tree down. The timing equation is thus exactly the same as t_{gather}. Similarly, the accumulate paradigm is the reverse of the broadcast paradigm. Thus

$$t_{scatter} = M[1 - d/N]t_{comm} \tag{4.28}$$

$$t_{accumulate} = Mdt_{comm} \tag{4.29}$$

As for the row-block case, it was assumed that communications parallelism has been exploited. It was also assumed that M/N is somewhat larger than 1. 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 sub-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 amount of computation involved in a matrix-vector product is M^2 complex multiplications and $M(M-1)$ complex additions, from Table 4.1. On most modern processors, the time required for a floating point addition and a floating point multiplication are approximately the same[16]. 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. Were there no communication, the time for the parallel operation, $T_p(N)$, would be simply

[15]Since the processor at the top of the tree is not used for the parallel products, the vector actually bifurcates at the top level.

[16]As will be shown in Section 4.7 for the transputer.

$8M^2/N$. Adding the communication time for the row-block decomposition gives

$$T_p(N) \approx \frac{8M^2}{N} t_{calc} + t_{broadcast} + t_{gather}$$

where t_{calc} is the time required for a real floating point addition or multiplication. Using equations (3.3), (4.25) and (4.27) yields the speed-up as

$$S \approx \frac{N}{1 + \frac{N}{8M}\beta[d(1-\frac{1}{N})+1]} \tag{4.30}$$

where

$$\beta \equiv \frac{t_{comm}}{t_{calc}}$$

This result was derived for the row-block decomposition case, but because the communication times are the same, the results for the column-block case will be virtually identical. (As noted, there is a small additional computational overhead with the latter decomposition).

Defining n as the number of rows per processor, $n = \frac{M}{N}$, the result can be re-written as

$$S \approx \frac{N}{1 + \frac{\beta}{8n}[d(1-\frac{1}{N})+1]} \tag{4.31}$$

Referring to the previous discussion in Section 3.4, the term $\frac{\beta}{8n}[d(1-\frac{1}{N})+1]$ is clearly identified as f_c. For $d \geq 2$, $N \approx 2^{d+1}$, hence $d \approx \log_2 N - 1$, and the following approximations for S and ϵ

$$S \approx \frac{N}{1 + \frac{\beta}{8n}\log_2 N} \tag{4.32}$$

$$\epsilon \approx \frac{1}{1 + \frac{\beta}{8n}\log_2 N} \tag{4.33}$$

are excellent for trees of depth 2 or more. This equation is very important; it indicates clearly that the matrix-vector product scales essentially with n_r, the number of rows per processor, and rather weakly (logarithmically) with the number of processors. Hence, for a given n_r, the efficiency is almost independent of the number of processors.

Figure 4.5 shows the efficiency for the MC^2 array. A value of 6.6 was used for β, computed from the manufacturers specifications[17][Dav90b, p.14]. The

[17]The results shown in Figure 4.5 were obtained early in this research, and are naïve, in that they assume the *manufacturers* specifications. The value for t_{calc} used here assumed that the transputer was using its fast internal memory. Since there is only 4kB of this,

curve shown in Figure 4.5 is smooth. 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. The former implies time "wasted" while the more lightly loaded processors do nothing; the latter implies "wasted" computation, and is the approach used in the author's Occam 2 implementation. Note that this "wasted" time or computation does not increase the run-time; while either waiting or operating on zero entries, the lightly loaded processors cannot perform any useful work in any case due to the load-balancing problem. This can be incorporated into the preceding analysis by replacing n_r by $\lceil n_r \rceil$. The effect on Figure 4.5 is to replace the smooth curve by a stairstep function.

It is also of interest to determine the point at which $\epsilon = 50\%$, or $f_c = 1$. Then

$$n_{1/2} = \frac{\beta \log_2 N}{8} \tag{4.34}$$

Thus with 62 worker processors, and $\beta = 6.6$, $M_{1/2} \approx 300$, where $M_{1/2} = Nn_{1/2}$. This is a reasonable number of unknowns and indicates that the matrix-vector product is very well suited to the proposed parallelization. Note that d is a rather weak function of N, so it can be stated in general that $M_{1/2}$ is approximately several times N. This is classified as a "large-grained" decomposition — each processor has a substantial number of unknowns — and is a typical property of a MIMD array with powerful processors and local memory.

The actual run-time can be obtained from

$$T_p \approx t_{calc} \frac{8M^2}{N}[1 + \frac{n_{1/2}}{n}] \tag{4.35}$$

Note that this is in a form similar to Hockney and Jesshope's two parameter model. Recognizing that the amount of work s is $8M^2$, defining the half-performance length $s_{1/2} = s\frac{n_{1/2}}{n}$, and recognizing that $r_\infty^{-1} = t_{calc}/N$, equation (4.35) can be re-written as

$$T_p \approx r_\infty^{-1}[s + s_{1/2}] \tag{4.36}$$

large problems will involve off-chip memory. Typically, a T800 slows down by a factor of three when it has to access off-chip memory. This can be incorporated in an obvious fashion into the above analysis - the effect is to decrease β and thus reduce $n_{1/2}$. Note that the actual run-time also increases. Futhermore, the actual value for t_{comm} also differs from the specifications. The full CG analysis of the next section uses results for β obtained from *benchmarking*.

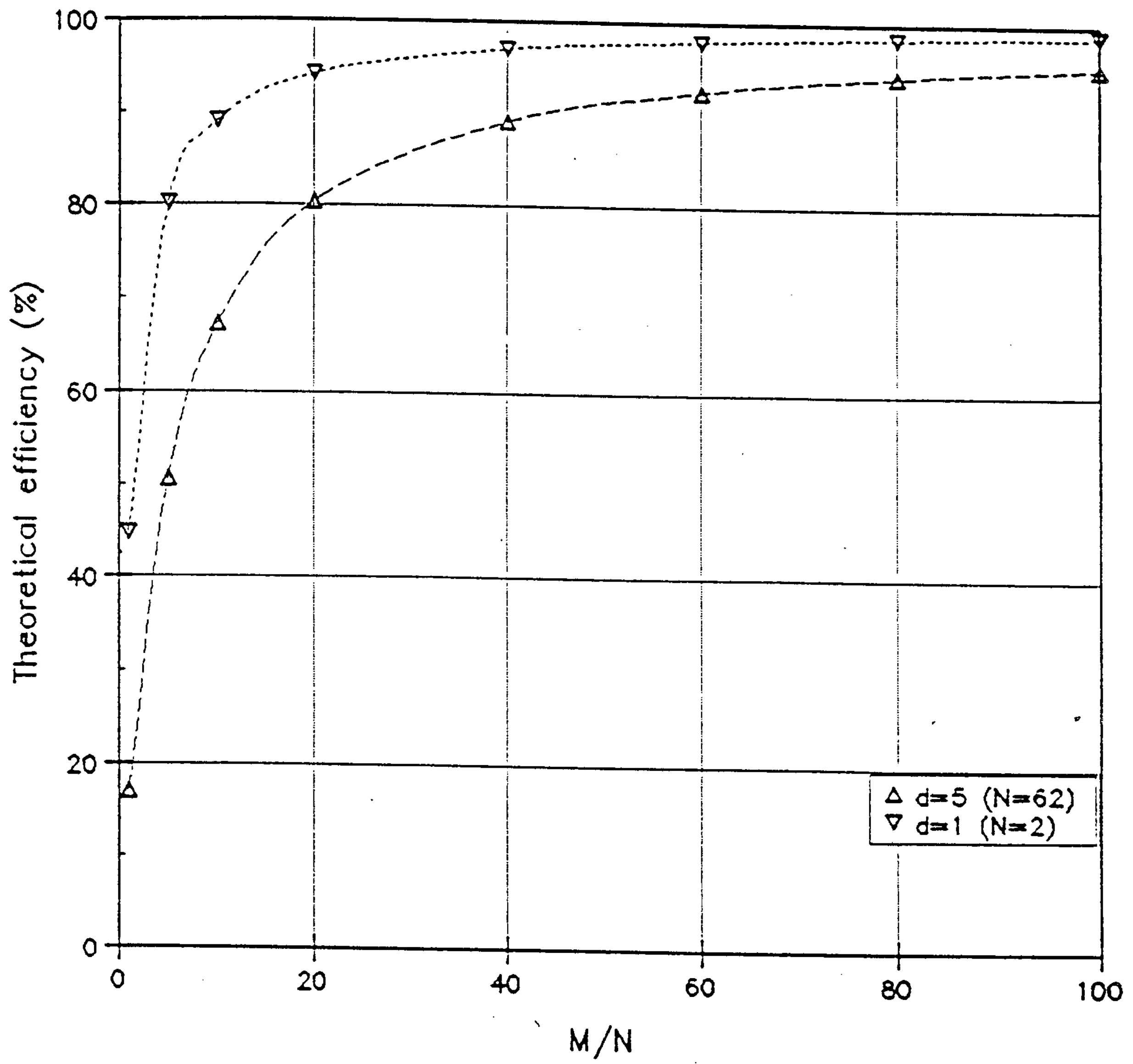


Figure 4.5: Efficiency of parallel matrix-vector product for the MC^2.

4.4 A Parallel Conjugate Gradient Algorithm

The theoretical techniques for the analysis of parallel algorithms developed for the matrix-vector product in Section 4.3 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 by applying the column-block decomposition algorithm to the *row-block* matrix data (with the necessary change of sign of the imaginary part of the matrix entries). This avoids having to either explicitly form the matrix transpose — a very expensive operation on a parallel processor with local memory — or store an additional copy of the Hermitian transpose of the matrix — and thus double the memory requirements of the code. This crucial contribution was the author's [Dav90b], and has not been published elsewhere, to the best of his knowledge.

From Table 4.1, the serial time is:

$$T_s \approx (16M^2 + 44M)t_{calc} \tag{4.37}$$

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:

$$T_p \approx (16M^2/N + 44M + 2dM)t_{calc} + (2M[1 - d/N] + 2Md)t_{comm} \tag{4.38}$$

Forming the quotient of T_s and T_p and simplifying yields

$$\epsilon \approx \frac{1 + \frac{2.75}{M}}{1 + \frac{N}{M}(2.75 + 0.125d + \frac{[1+d(1-\frac{1}{N})]\beta}{8})} \tag{4.39}$$

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.

Comparing to Equation (4.31), many similarities are obvious. Under the assumption $M \gg 1$, the numerators are identical (within a factor N; remember that S and ϵ are being compared). The two additional terms in the denominator of equation (4.39) represent, respectively: the amount of unparallelized computation contained in steps 2, 3, 4, 6 and 7; and the amount of additional computation required in the accumulate process. Note that

```
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 4.6: Pseudo-code for parallel CG algorithm: master process

both the amount of (parallelized) computation *and* the amount of communication approximately double since a matrix-vector product *and* a Hermitian transpose matrix-vector product are required; thus the factor of two in both computation and communication cancels.

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

$$\epsilon = \frac{1}{1 + \frac{N}{M}(2.75 + 0.125d + \frac{\log_2 N\beta}{8})} \tag{4.40}$$

Pseudo-code for the algorithm is given in Figures 4.6 and 4.7 for the master and worker(s) respectively.

The correct termination of the algorithm and the configuration of the workers for an arbitrary depth of binary tree will be now be considered in the next two sections.

4.5 Terminating the Algorithm

Ensuring the correct termination of a parallel algorithm is not, in general, a trivial problem, especially with heterogeneous processes. It is simpler

```
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 4.7: Pseudo-code for parallel CG algorithm: worker process

with the homogeneous worker processes used for the parallel CG code, but nonetheless requires careful coding to terminate all the concurrent processes in the correct sequence.

With a sequential algorithm, the termination is normally fairly simple: the code executes and then terminates (provided it has been properly written and does not contain any infinite loops - *livelock* in Occam parlance). The fundamental principle of terminating a group of parallel processes (the master and workers executing the CG algorithm in this case) is that some explicit termination action is required, initiated by one of the processes. In the code considered, 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 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. While appearing obvious, if not carefully coded it is rather easy to terminate intermediate level workers before they have terminated the lower level workers, leading to a particularly subtle "deadlock" that only manifests itself on the next run of the code. Pseudo-code for the correct termination procedure is given in Figures 4.8 and 4.9.

```
begin{terminate stub: master}
  if termination criteria satisfied
    then
      begin
        flag := terminate
        finished := TRUE {to stop while loop}
      end
    else
      begin
        flag := continue
        finished := FALSE {to continue while loop}
      end
  send flag to two lower workers
  end{if}
end{terminate stub: master}
```

Figure 4.8: Pseudo-code for the termination stub on the master.

Note that the requirement for terminating parallel algorithms properly is a general property of parallel processing, requiring special care on a local memory MIMD array.

4.6 Configuring for an Arbitrary Depth of Binary Tree

The requirement of a configuration description should be reviewed at this stage. The master and worker processes defined thus far are in general written using channels, which are software abstractions of the actual communication channels. The mapping of these channels onto the real hardware links and the explicit placing of processes on real processors is the task of the configurer. Precisely how this is implemented will vary from computer to computer, and also from language to language. The description that follows is specific to Occam 2 on a transputer array; but a similar procedure must be applied on any MIMD computer, and the general procedure followed remains valid.

The problem of generating a configuration description that allows the specification of a binary tree of *arbitrary* (integer) depth is not a simple one. Related to this is the problem of how one actually connects the transputer links in such a way that their switching is guaranteed to be the same as that

```
begin{terminate stub: worker}
  receive flag from higher processor
  if not at bottom of tree
    then send flag to lower workers
  end{if}
  if flag = terminate
    then finished := TRUE
    else finished := FALSE
  end{if}
end{terminate stub: worker}
```

Figure 4.9: Pseudo-code for the termination stub on the worker.

defined by the configuration file.

For a small numbers of processors, the problem can be solved manually, by sketching a graph of the network, assigning links to channels and then explicitly placing each channel and process. But this rapidly becomes unwieldy for large networks and an automatic method becomes necessary. If the binary tree is numbered as shown in Figure 4.10, then a very elegant scheme is possible [Gal90], which is used as the basis for the author's configuration.

However, this scheme is complicated by the following restriction imposed by the MC^2; note that this is a very specific restriction of a particular type of transputer array. Because of the Euler colouring algorithm used [Vil89], the link interconnection is restricted to even-to-even and odd-to-odd link switching. This requires that the even numbered and odd numbered nodes be handled separately. The situation is further complicated by the requirement to provide a "boot-path"; the transputer network loader is only able to load the network over one link, so the highest level nodes (1 and 2 in Figure 4.10) require an extra connection to provide this. Developing a configuration file that handles all this is somewhat complicated. The subtleties (that cause the problems) are lost in pseudo-code, so the necessary Occam configuration code is given in Appendix A.

Once the configuration code is developed, the problem remains of ensuring that the switching of the MC^2 is correct. This switching is software controlled, and the necessary link switching is read from an ASCII file by the *Domain Management System* running on the MC^2 controller.

A utility program, `switchtds`, was written in Turbo Pascal by the author to take the "wiring diagram" that the TDS can generate from a configuration file and automatically generate an MC^2 switch file from it. A check was also performed to ensure that the even-even and odd-odd switching requirements were met. The program also checks that links are actually connected: e.g. if processor 1, link 3 is defined as being connected to processor 2, link 1, then the program checks that processor 2, link 1 is in turn indeed connected to processor 1, link 3. Program `switchtds` is of course applicable to any parallel program running under the TDS program, not just the binary tree configuration[18].

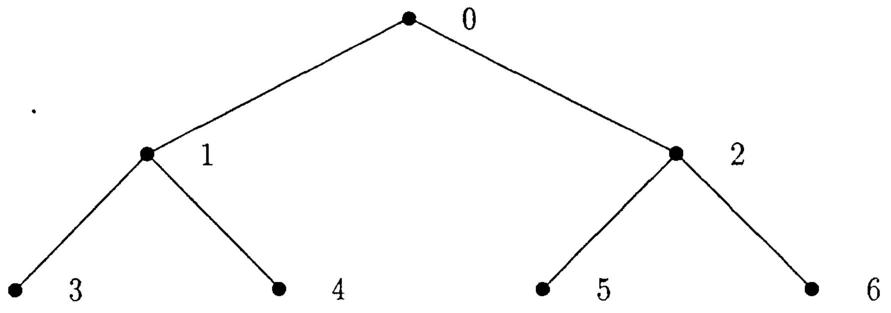


Figure 4.10: Interconnection topologies — binary tree dimension 2, renumbering following [Gal90, p.123]

4.7 Benchmarking

The previous 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; the pseudo-code presented here will be useful for benchmarking any local memory MIMD system; the specific results are for the transputer arrays tested.

The computation benchmark involved the addition of 7 vectors of length 1000. The core of the benchmark is shown in Figure 4.11. It was attempted

[18]Another utility developed by the author is program `Euler`, a variant of `switchtds`, which checks only the even-even and odd-odd properties and link connectedness of a supplied switching file. An invalid file will not load, but the Domain Management System produces only an extremely cryptic error message, and does not indicate where the fault lies, hence the requirement for tools such as `Euler`.

```
begin{flop benchmark}
  repeat for i = 1 to 1000
    begin
      vector2[i] := vector1[i] + vector2[i]
      vector3[i] := vector1[i] - vector2[i]
      vector4[i] := vector3[i] + vector2[i]
      vector5[i] := vector3[i] - vector2[i]
      vector6[i] := vector4[i] + vector3[i]
      vector7[i] := vector4[i] - vector3[i]
      vector8[i] := vector5[i] + vector4[i]
    end
  end{for}
end{flop benchmark}.
```

Figure 4.11: Pseudo-code for FLOP benchmark

to ensure that there were dependencies between the various vectors to prevent any optimization of the code by the compiler — such optimization is potentially very dangerous with benchmarking since one may well not realize that it is occurring. (For example, some compilers may be able to recognize the operation of adding zero to all the elements of a vector as not affecting the vector at all, and not generate any code for these operations.) Seven operations were chosen to minimize the role of loop overhead (the incrementing and testing of i) which could adversely affect the measurement. A vector length of 1000 was chosen to be sufficiently long to obtain a proper average. This exercise was performed for single and double precision and then repeated for multiplication. No division operations were included in tests since the number of divisions required by the algorithms considered in this thesis was insignificant. Results are given in Table 4.2 and 4.3 for the MC^2 and a TM4 board[19] respectively. Both systems use 20 MHz T800 transputers. The approximately 10% improvement on the MC^2 is due to the faster memory used on that system. It is interesting to note that addition and multiplication take *exactly* the same time. It is also notable that double precision (64 bit) arithmetic is about only 1.4 times slower than 32 bit arithmetic — the floating point unit on the T800 is a 64 bit unit. The approximation of 0.5 MFLOP/s used previously [Dav90b] is thus shown to be quite accurate for single precision.

The pseudo-code used for the benchmarking of the communication speed

[19]The TM4 board consists of one T800 transputer with 4 MB of RAM.

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

Table 4.2: Computation benchmarks on the MC^2

Precision	*Operation*	*MFLOP/s*
Single	Addition	0.48
Double	Addition	0.34
Single	Multiplication	0.48
Double	Multiplication	0.34

Table 4.3: Computation benchmarks on the TM4 board

```
process[master.link]
begin
  par
    begin
      send sp.vector to processor[1]
      receive sp.vector from processor[1]
    end
    begin
      send dp.vector to processor[2]
      receive dp.vector from processor[2]
    end
  end{par}
end{process[master.link]}
```

Figure 4.12: Pseudo-code for communication benchmark: master process

```
process[sp.worker.link]
begin
  receive sp.vector from processor[0]
  send sp.vector to processor[0]
end
end{process[sp.worker.link]}
```

Figure 4.13: Pseudo-code for communication benchmark: single precision worker process

```
process[dp.worker.link]
begin
  receive sp.vector from processor[0]
  send sp.vector to processor[0]
end
end{process[dp.worker.link]}
```

Figure 4.14: Pseudo-code for communication benchmark: double precision worker process

```
process{link benchmark}
placed par
  processor[0]
    process[master.link]
  processor[1]
    process[sp.worker.link]
  processor[2]
    process[dp.worker.link]
end{link benchmark}
```

Figure 4.15: Pseudo-code for communication benchmark: configuration

is given in Figure 4.12, 4.13 and 4.14 for the master and the two workers respectively and the configuration code is given in Figure 4.15. This communication benchmark combines two tests: firstly, it checks the transfer rate for single and double precision, and secondly, it checks that the links can indeed be operated efficiently in parallel. The vector `sp.vector` is a single precision vector of length x, and `dp.vector` is a double precision vector of length $x/2$. Both processes should thus take the same time; for the double precision case, half as many elements, each individually twice as long, will be sent. Results of the test are given in Tables 4.4 and 4.5.

Precision	*MByte/s*
Single	1.32
Double	1.39

Table 4.4: Communication benchmarks on the MC^2

Precision	*MByte/s*
Single	0.87
Double	0.90

Table 4.5: Communication benchmarks on the TX4

The TX4 board[20] has its links set to 10 MBit/s; one byte is 8 bits and the transputer's link protocol adds 3 padding bits per byte, hence we expect a theoretical value of about 0.91 MByte/s, very close to the 0.87 and 0.9 measured. This clearly shows that the links can indeed operate concurrently with an efficiency of very close to 100%.

The link speed of the MC^2 is specified at 20 MBit/s, i.e. 1.82 MByte/s, taking the 3 bits padding per byte into account. The benchmark, which gives very reliable results in the case of the TX4, shows only about 75% of the predicted value for the MC^2. The reason is probably the delays in the electronic link switches that interconnect links on the MC^2. (The TX4 links are hardwired using jumper cables; there is no electronic circuitry between one transputer's links and another on the TX4 board).

[20]The TX4 board has 4 T800 transputers each with 256kB of RAM; it can be plugged into an expansion slot in a PC.

The parameter β can now be computed from the benchmark results for the case of single and double precision[21]. The numerical values given in Table 4.6 are for the MC^2.

Precision	β
Single	3.22
Double	4.37

Table 4.6: $\beta = t_{comm}/t_{calc}$

4.8 Results and Discussion

This section describes results obtained by the author using his Occam 2 implementation of the algorithm described in this chapter. It represents the experimental validation of the timing models developed in this chapter. The implementation was very time-consuming due on the one hand to the inadequate software tools available — for instance the absence of interactive debuggers made debugging a very slow and tedious process — and on the other hand, due to the absence of any paradigms for writing parallel codes. The fundamental parallelizing paradigms used for the code were developed entirely by the author before useful books on the subject such as [FJL*88] were available. The pseudo-code stubs given in this thesis appear rather simple only with the benefit of hindsight.

The algorithm as described was implemented by the author. Initial validation of the code for *correctness* was done using a test set of different linear systems, for which the solutions were checked using MATLAB's (implicit) linear system solvers [MAT89]. The parallelism was simulated on one transputer for this, as discussed in Section 3.4.2; this was far more convenient for debugging than running the code on the MC^2. Further validation was performed when the parallel CG solver was used as the matrix solver in PARNEC; see Section 6.6. This validation used real parallelism on the MC^2.

Following the initial validation for code correctness, the algorithm was run on the MC^2, to validate the timing predictions. Measured efficiencies are shown in Figure 4.16. Theoretically, equation (4.39) predicts that the

[21]An example of the calculation for single precision: the time for a floating point calculation, from Table 4.2, is $t_{calc} = 1.88\mu s$. One single precision complex word is 8 bytes, thus $t_{comm} = 6.06\mu s$. Thus $\beta = 3.22$. For the double precision case, the results for double precision must be used, and one double precision word is 16 bytes.

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 4.17. *Thus the 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.*

Finally, the measured and predicted results for 2 and 30 workers are shown in Figures 4.18 and 4.19 respectively. At the time of writing, the system was missing one or two processors so tests could not be conducted on the full array, but no larger problem could have been solved since the worker processes are all identical and hence the transputer with the smallest memory determines the total memory that the complete parallel program can use[22].

It will be noted that in Figures 4.18 and 4.19, 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. 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, for example; 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.

In the regime of small $\frac{M}{N}$, various effects that were ignored in the analysis, such as latency[23], come into play. However, this regime is of minimal importance for the solution of large, time-consuming problems. (The aim of parallel processing should always be borne in mind, namely to provide computational power for tackling problems that are prohibitively expensive computationally on a single processor). In the region of large $\frac{M}{N}$, the "serial" times are based on extrapolation. The largest problem whose serial time was measured used 428 segments; using double precision this requires close on 4 MB of memory, the maximum RAM available on one processor on the MC^2. The *measured* data for the larger problems are based on extrapolation of the serial times, and as such, this "measured" data should be treated with some caution[24]. Furthermore, the theoretical timing model ignores some of

[22]At the time of writing, for the complete MC^2, this limit was imposed by the transputers with 1 MB. Thus with 62 workers, 62 MB is available — only 1 MB of the 2 MB and 4 MB boards' memory is thus used. There were sufficient 2 MB and 4 MB boards for 30 workers to use 2 MB, thus a total of 60 MB.

[23]Latency is the delay caused when initiating communications. Using Occam 2 on a transputer array, it is only significant for very short vectors.

[24]It should be possible to run larger tests using the virtual memory boards but the time required becomes quite impractical and one would then have to consider the question: is the "serial" time measured using the virtual memory boards a true reflection of the time that the problem would have run had there been enough RAM available on one transputer

the finer detail of the implementation; for instance, Occam does not by default permit separate parts of vectors to be accessed in parallel, so the actual Occam implementation of the gather process differs slightly from the pseudocode given in this chapter. This is discussed in more detail in Section 5.10. In the light of this the offset noted is not surprising, and again it must be emphasized, not important when the goal of the modelling is borne in mind.

The measured data shown was obtained from *PARNEC*, the parallel version of NEC2 to be described in Chapter 6. Double precision was used. The value for β used in the theoretical model was that measured in Section 4.7.

4.9 Conclusions

In this chapter, a parallel conjugate gradient algorithm has been proposed, analyzed theoretically, implemented and tested on a transputer array. Measured results for speed-up and efficiency are compared to the theoretical predictions. The parallel algorithm used as its basic building block two parallel matrix-vector products, so two efficient parallel matrix-vector product algorithms were investigated in detail. The exploitation of the *complementary* role of these two methods permitted the Hermitian transpose matrix-vector product, required in addition to the matrix-vector product by the general form of the CG algorithm, to be solved very elegantly using only one copy of the matrix and without the additional communication overhead that would have been required had the transpose been explicitly formed. The basic parameters used in the theoretical analysis were established using benchmarking. The problem of developing a *general* configuration description, able to handle any depth of binary tree, has been considered and a solution presented. Automatic methods for connecting the links (the "switching") on an electronically switched array have been described; specific details have been provided for the particular transputer array used. The problem of terminating a parallel algorithm has been considered and a solution given for the CG algorithm. Experimental validation of the theoretical models has been performed by measuring efficiencies of an Occam 2 implementation of the algorithm on a transputer arrays.

It has been shown, both theoretically and experimentally, that the proposed CG algorithm scales essentially with the number of rows per processor — the development of *scalable* algorithms is very important for large MIMD systems. A scalable algorithm is one whose efficiency is a function of the "grain" of the problem, where the "grain" is related to the number of unknowns per processor; in the case of the CG algorithm, it is the number of rows per processor. Hence as the problem size grows, more processors can

board, or is the virtual memory management degrading the performance substantially?

be brought to bear on the problem and the efficiency remains approximately constant - and hence the actual speed-up increases *linearly* with the number of processors.

Some of the work reported in this chapter was presented at international symposia as [DC89, Dav90a].

The CG algorithm is one of the standard methods used in computational electromagnetics for solving the system of linear equations resulting from a MoM formulation. Another standard method of solving the system of linear equations generated by the MoM is LU decomposition. Initially, it appears a rather unlikely candidate for efficient parallelization when compared to the CG algorithm considered in this chapter, and introduces a new problem only hinted at (since it was of no significance) in the work on the CG algorithm, namely *load balancing.* Chapter 5 considers the development of a parallel LU algorithm, with an efficiency comparable to that of the parallel CG method for a similar problem size and number of processors.

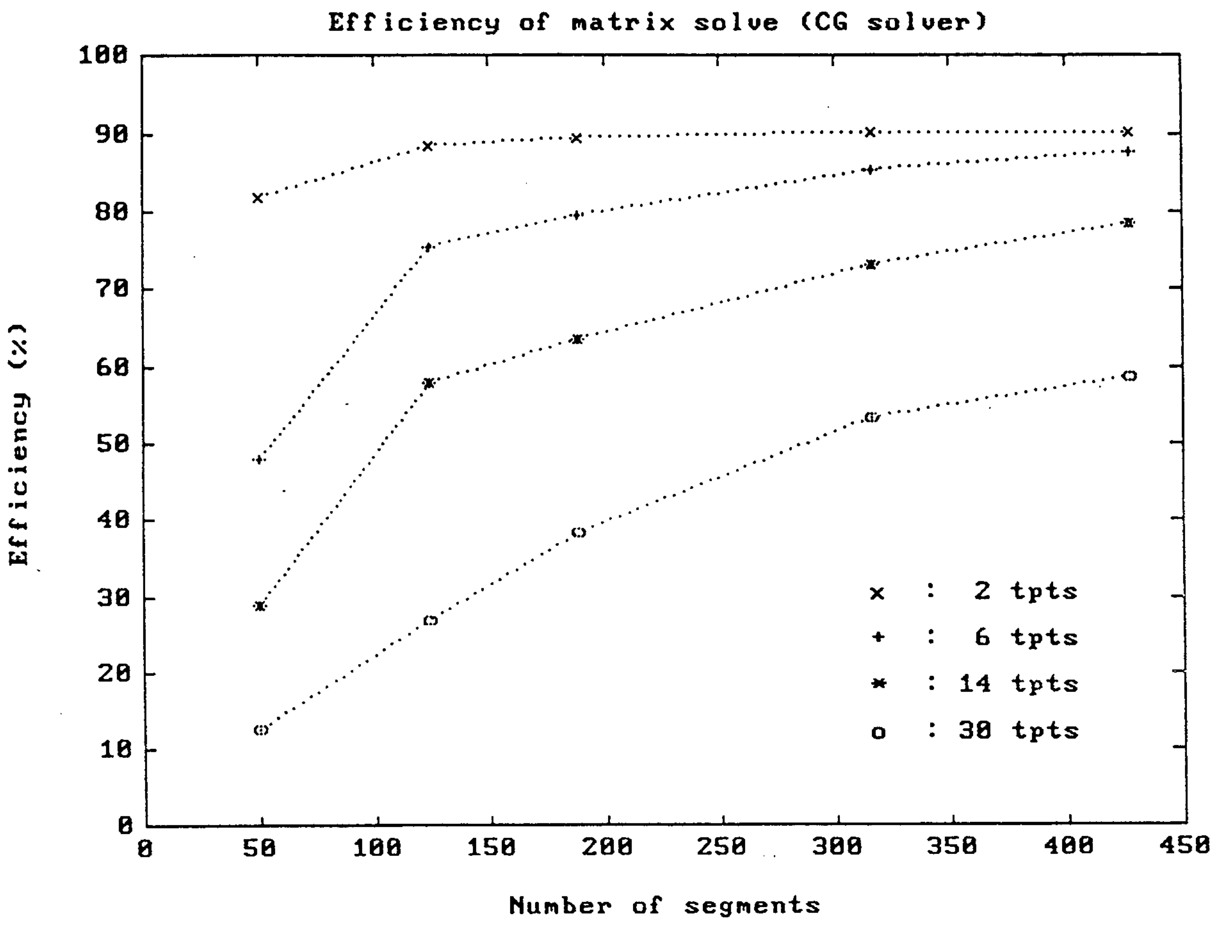


Figure 4.16: Measured efficiency of parallel conjugate algorithm versus unknowns for the MC^2.

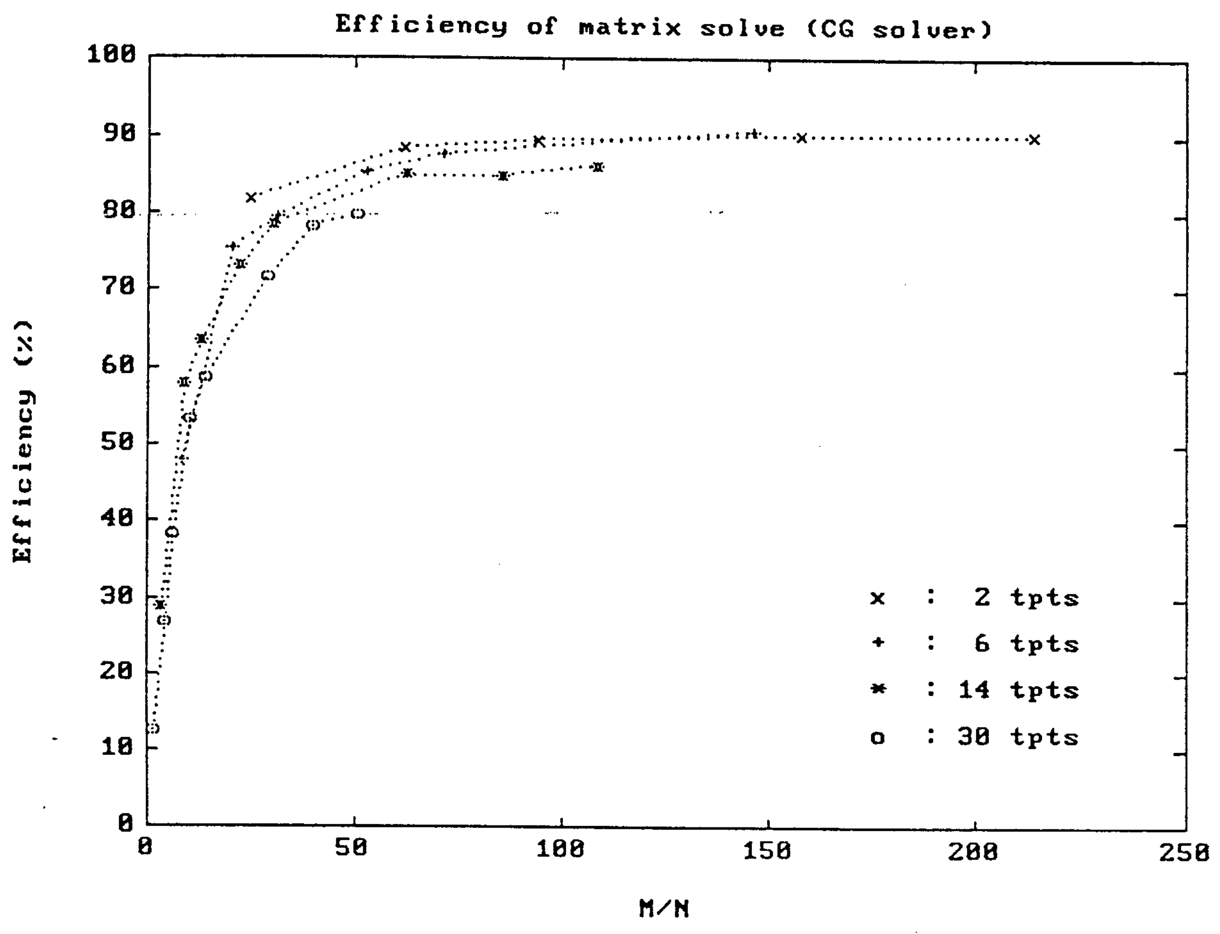


Figure 4.17: Measured efficiency of parallel conjugate algorithm versus rows per processor for the MC^2.

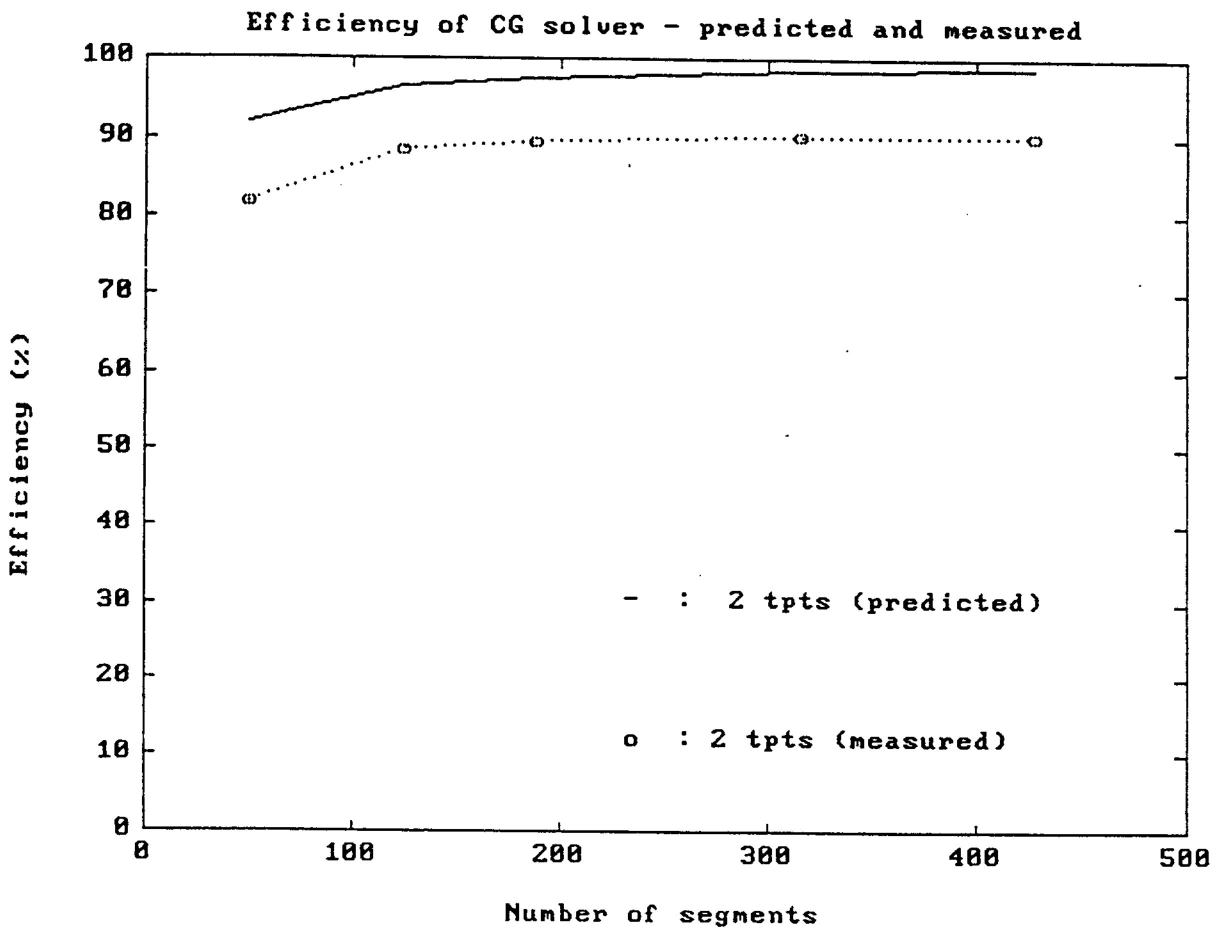


Figure 4.18: Efficiency of parallel conjugate algorithm (2 worker transputers) for the MC^2.

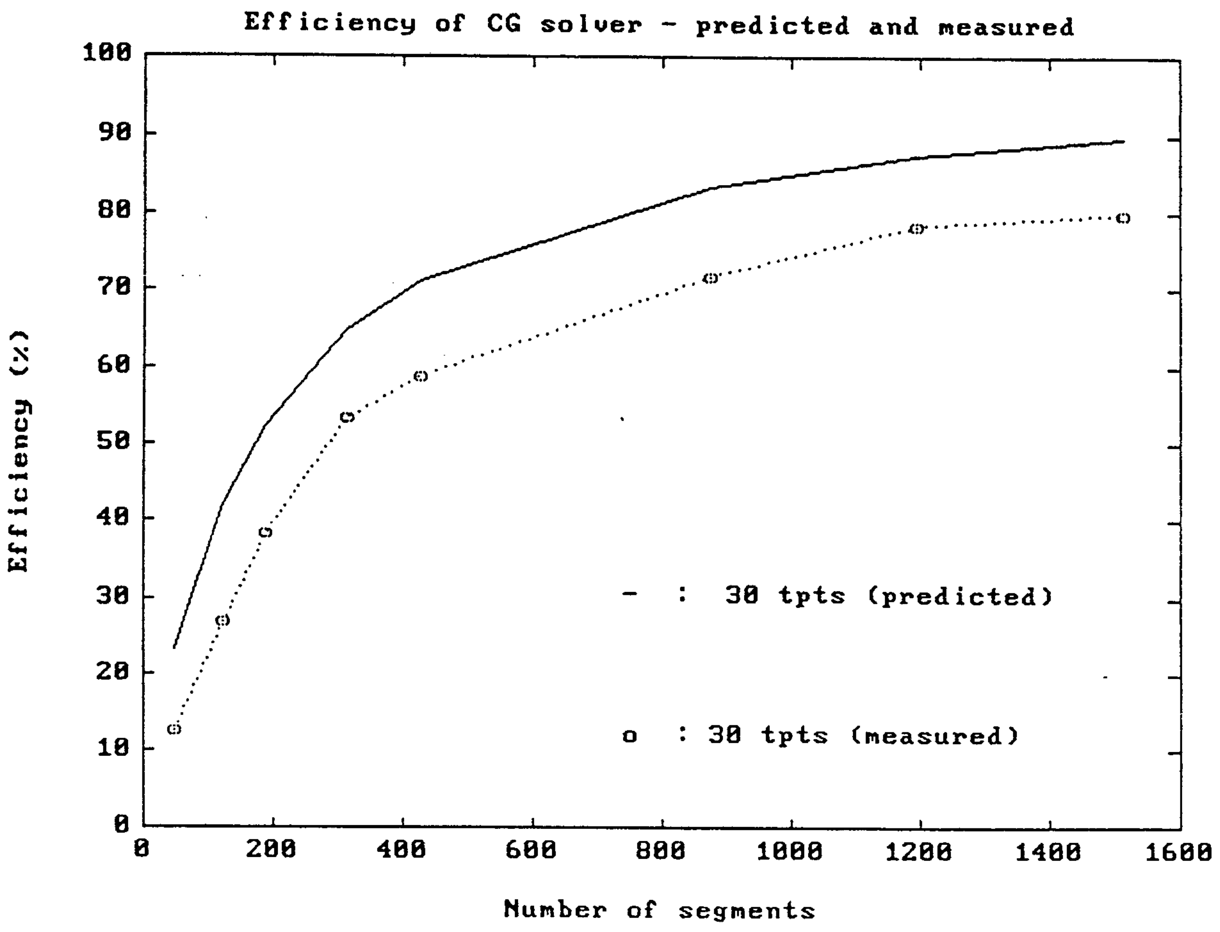


Figure 4.19: Efficiency of parallel conjugate algorithm (30 worker transputers) for the MC^2.

Chapter 5

A Parallel LU Algorithm

5.1 Introduction

The LU method is probably the most widely used algorithm in the solution of square systems of linear equations. Given a system with a moderate number of equations, it is normally the best algorithm to use, provided that the system is not extraordinarily ill-conditioned. For certain special classes of matrix, such as positive definite matrices, it may be shown that the matrix is non-singular [BF85] and furthermore, that the algorithm is stable with respect to the growth of round-off errors. Variants of the LU method exploiting positive-definiteness such as the Choleski method (which forms $[L][L]^T$) combined with methods to exploit bandedness permit the very efficient application of the method to matrices generated by the finite element method [SF90, Chapter 10]. 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.

LU decomposition (an $O(M^3$ operation, where M is the dimension of the problem) followed by forward and backward substitution (each of $O(M^2)$) is always better to use when solving a system of equations than forming the explicit inverse of the matrix and then multiplying the inverse matrix by the right hand side vector; the reason is very simply that forming the explicit inverse amounts to an LU decomposition, followed by M forward and backward substitutions (these M substitutions adding an additional $O(M^3)$ operation). This is clearly more expensive computationally, and the accuracy of the solution can also be degraded by the additional operations. These points are made in almost all books on numerical analysis and the applications thereof, eg. [BF85, p.318] and [SF90, Chapter 10].

In this chapter the basic LU algorithm is reviewed. Then the question of parallelizing the algorithm is discussed, and a parallel algorithm presented using a new, simple, graphical approach. The use of "formal" methods is considered for the extraction of the parallelism. The problem of *load-balancing* is potentially serious with a parallel LU algorithm, and a solution utilizing a row and column interleaved scheme is described. Pseudo-code is given for the parallel algorithm. Then, new parallel forward and backward substitution algorithms are proposed that use the same data decomposition as the parallel LU algorithm. Pseudo-code is given for these algorithms. Timing results obtained using a matrix generated by a simple thin-wire MoM code are presented and discussed.

The ability to solve massive systems of equations made possible by this technique is used to investigate the accuracy of the LU method for large problems, by monitoring the convergence of the input impedance of a dipole computed using a MoM code and also by comparison with the results of the parallel CG solver for the same MoM code. It is established that the LU method only fails when the basic rules for a MoM discretization are *seriously* violated, permitting the conclusion that for large electromagnetic problems discretized according to the established rules, the LU method will be sufficiently accurate.

5.2 The Basic LU Algorithm

Before considering the parallel version of the LU algorithm, the serial form will be briefly reviewed.

The LU algorithm factors a matrix A into the product of an upper ([U]) and lower ([L]) triangular matrix as follows:

$$[L] = \begin{bmatrix} l_{0,0} & 0 & \dots & 0 \\ l_{1,0} & l_{1,1} & \dots & 0 \\ \cdot & \cdot & \cdot & \cdot \\ l_{M-1,0} & \dots & l_{M-1,M-2} & l_{M-1,M-1} \end{bmatrix} \qquad (5.1)$$

and

$$[U] = \begin{bmatrix} u_{0,0} & u_{0,1} & \dots & u_{0,M-1} \\ 0 & u_{1,1} & \dots & u_{1,M-1} \\ \cdot & \cdot & \cdot & \cdot \\ 0 & \dots & 0 & u_{M-1,M-1} \end{bmatrix} \qquad (5.2)$$

The diagonal elements of [L] are most commonly chosen as 1. The algorithm can be found in virtually any book on matrix algebra, for example [BF85, p. 345], and can be derived by noting that $a_{i,j}$ (the ij-th element of A; $i,j \in \{0,1,\dots,M-1\}$; M is the dimension of the matrix[1]) is the product

[1]The matrix entries have been numbered from 0 to M-1 for later convenience: an array a in Occam is numbered a_0, a_1, ...

of the i-th row of [L] with the j-th column of [U]. Exploiting the triangular properties of [L] and [U] then yields the algorithm:

- Step 0, which computes the first row of [U] and the first column of [L], is defined as follows:

$$u_{0,0} = \frac{a_{0,0}}{l_{0,0}} \tag{5.3}$$

$$u_{0,j} = \frac{a_{0,j}}{l_{0,0}} \tag{5.4}$$

$$l_{j,0} = \frac{a_{j,0}}{u_{0,0}} \tag{5.5}$$

- 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:

$$u_{i,i} = \frac{1}{l_{i,i}}[a_{i,i} - \sum_{k=0}^{i-1} l_{i,k}u_{k,i}] \tag{5.6}$$

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

$$u_{i,j} = \frac{1}{l_{i,i}}[a_{i,j} - \sum_{k=0}^{i-1} l_{i,k}u_{k,j}] \tag{5.7}$$

$$l_{j,i} = \frac{1}{u_{i,i}}[a_{j,i} - \sum_{k=0}^{i-1} l_{j,k}u_{k,i}] \tag{5.8}$$

  $l_{i,j}$ and $u_{i,j}$ represent the i,j-th element of the [L] and [U] matrices respectively.

- Step M-1 completes the algorithm by computing the last diagonal element and is given by

$$u_{M-1,M-1} = \frac{1}{l_{M-1,M-1}}[a_{M-1,M-1} - \sum_{k=0}^{M-2} l_{M-1,k}u_{k,M-1}] \tag{5.9}$$

If at any stage $l_{i,i}u_{i,i} = 0$ then the algorithm is terminated with an error message to the effect that factorization is impossible.

It will be noted that the LU decomposition as defined above leaves M degrees of freedom; generally either the diagonal elements of [L] are set to 1 (the most common procedure with a general matrix), or the diagonals of [L] and [U] are set equal[2] (the approach used in Choleski decomposition, where the matrix A is positive definite).

[2]With a complex valued matrix, the diagonal of [L] and the complex conjugate of the diagonal of [U] are set equal.

It is rather interesting that LU decomposition is very closely related to the CG method; both the LU method and the CG method may be derived as special cases of what Sarkar has called the method of conjugate directions (essentially a gradient method) [SA85]. This is perhaps not surprising since the CG algorithm (in the absence of round-off error) reduces the dimension of search by one for each iteration, which is also what the LU method does.

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 and the second step backward substitution. Consider

$$[A][x] = [b] \tag{5.10}$$

with A factored as

$$[A] = [L][U] \tag{5.11}$$

Thus we must solve

$$[L][U][x] = [b] \tag{5.12}$$

Define

$$[U][x] = [z] \tag{5.13}$$

Then $[z]$ can be solved for using forward substitution from

$$[L][z] = [b] \tag{5.14}$$

since $[L]$ is lower triangular; then x can be solved using backward substitution from the preceding equation since [U] is upper triangular.

Using the following useful formulae

$$\sum_{j=1}^{n} j = \frac{1}{2}(n)(n+1) \tag{5.15}$$

$$\sum_{j=1}^{n} j^2 = \frac{1}{6}(n)(n+1)(2n+1) \tag{5.16}$$

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. 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.

5.3 Parallelizing the LU Algorithm — an Introduction

5.3.1 A Brief Review of Previous Work

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 in Chapter 4, 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. For example, Fox *et.al.* [FJL*88, Chapter 20] describe a banded matrix LU decomposition. Brief reviews of parallel LU decomposition may be found in [Hel78, GHN87]; a rather more recent review paper is [GPS90]. Many of the papers published are extremely specific to a particular processor, or are of a very basic theoretical nature — for example, establishing a lower bound on the operation count given as many processors as can be used. They are thus not suitable for the problem of developing parallel algorithms on a MIMD system, with a reasonable — but restricted — number of processors. Recent work by van de Vorst [vdV88, vdVB89] has described a parallel LU algorithm suitable for a MIMD array. Previous work (such as that reviewed in [GPS90, p.99]) is shown to be a special case of van de Vorst's algorithm. The work was published in computer science journals, and is very difficult to read without a grounding in formal methods and the use of post-conditions[3] as a mathematical tool for extracting parallelism. However, the basic ideas can be explained far more simply than is done in van de Vorst's papers. This will first be done using a new graphical approach developed by the author and then van de Vorst's formal approach will be presented in Section 5.5; rather more detail and elucidation will be presented than is available in van de Vorst's papers and rather simpler methods developed by the author for deriving many of his results are given.

5.3.2 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 (5.7) and (5.8) may be performed as soon as data is available, given sufficient processors. As an example, the first summation for each element of row 2 of [U] may begin as

[3]Formal methods and post-conditions will be discussed in Section 5.4.

soon as the relevant element of row 1 of [U] and column 1 of [L] is available. All the summations required for row 2 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 3, 4 etc. may also begin as soon as the results of row 1 and column 1 are available[4]. 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.

The algorithm can be most easily understood graphically. Figures 5.1 to 5.4 show the evolution of the algorithm for a matrix of dimension 4 on a 4 by 4 array of processors, i.e. one processor per element. (This is the upper limit of the parallelism that can be extracted with this algorithm). The $\bullet$ represents elements that are critical i.e. in the last stage of computation. The $\circ$ 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.

This graphical presentation also shows the most serious problem with the algorithm — load balancing. 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]. This load balancing problem may be solved very elegantly using a double-interleaving scheme for data distribution described in [vdV88, vdVB89], whereby both row and columns are scattered modulo$\sqrt{N}$ over a square array of $\sqrt{N}$ by $\sqrt{N}$ transputers, with $\sqrt{N} \ll M$. This is of course the situation of interest, since it is most unlikely that an array of several million transputers will be available to solve problems with a few thousand unknowns. The distribution of a matrix of dimension 9 on a 3 by 3 array using this double interleaved distribution is shown in Figures 5.5 and 5.6.

Note that at each step i, corresponding to one of the Figures 5.1 to 5.4, the algorithm needs two discrete computational steps: firstly, to compute the i-th column of [L], and secondly, to then update the partial sums on the active processors[6]. Hence with M processors the algorithm will take $2Mt_{calc}$

[4]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.

[5]This is derived in detail in Section 5.8.

[6]This assumes the choice of diagonal elements of [L] as in equation (5.29), and the correct initialization as discussed in Section 5.5; in this case, all computations required by

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

Figure 5.1: Step 1 of LU decomposition

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

Figure 5.2: Step 2 of LU decomposition

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

Figure 5.3: Step 3 of LU decomposition

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

Figure 5.4: Step 4 of LU decomposition

to terminate, assuming the times for floating point addition, subtraction, multiplication and division to be similar.

$$\begin{bmatrix} 00 & 01 & 02 \\ 10 & 11 & 12 \\ 20 & 21 & 22 \end{bmatrix}$$

Figure 5.5: 3 by 3 processor array (mesh) numbering

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

Figure 5.6: Scattered grid distribution; 3 by 3 processor array (mesh). The elements in the upper left corner map onto processor 00, those in the upper centre onto 01, those in the left centre onto 10, etc.

5.4 The Use of Formal Methods

van de Vorst used formal methods in the development of his parallel program[vdV88]. The formal methods that he uses are an extension of the work of Owicki and Gries [OG76b, OG76a]. van de Vorst's paper is quite formidable as a result of the methods used, although the results are quite simple once the underlying principles are understood.

Formal methods are techniques used by computer scientists to analyze algorithms, primarily from the viewpoint of correctness. This frequently seems a rather obvious exercise to the numerical analyst, since the algorithms being coded have already been put on a sound foundation using the mathematics of

step i for row i of [U] have been completed.

numerical analysis. However, there are many problems in computer science that arise from problems involving communicating processes — this led to Hoare's work on Communicating Sequential Processes [Hoa85] — and these can be intractable problems indeed, since the times of communication and even the communication paths themselves may be non-deterministic. Hence the development of formal methods. However, how successful they have been in their stated goal of increasing software reliability is a subject for heated debate; Stover's comments on software testing and methods of software validation make very interesting reading[Sto90b, Sto90a].

It will also have been noted thus far in this thesis that the extraction of parallelism has been largely an intuitive process, relying frequently on a geometrical viewpoint to establish the fundamental data indepedencies. The pseudo-code has been used for *documenting*, not *developing*, the parallel algorithms. This section discusses a mathematics to formalize the extraction of parallelism. The necessary mathematics will be introduced by way of several examples, leading up to the full LU problem. The mathematics follows that of van de Vorst [vdV88]; the elucidation is primarily the present author's, as is the discussion of the analysis of the matrix-vector product using formal methods.

5.4.1 An Introductory Example

Consider the problem of forming the sum of two vectors:

$$[c] = [a] + [b] \tag{5.17}$$

This can be written in an equivalent form by defining a *postcondition* R which must be true after the code stub executing this has terminated:

$$R \equiv (\ \forall i : 0 \leq i \leq M-1 : c_i = a_i + b_i) \tag{5.18}$$

This is of course simply the definition of the addition of two vectors.

For the development of a program, we can do this by introducing the concept of an *invariant*. One possible invariant is P_{seq}, defined as follows:

$$P_{seq} = (\ \forall i : 0 \leq i \leq k : c_i = a_i + b_i)\ ;\ 0 \leq k \leq l \quad l \in \mathbf{Z}^+ \tag{5.19}$$

The upper value of l is left unspecified at present. The value of k which satisfies the postcondition will be determined shortly. P_{seq} is computed in an ordered sequence, as k is incremented from 0. To maintain the validity of the invariants at each stage of the process, the code stub

```
c[i] := a[i] + b[i]
k := k+1
```

must be executed. (With an example this simple, this is stating the obvious). Once all the components of P_{seq} have been computed, which occurs when $k = M - 1$, the operation is complete and the *truth of the post condition is established*, to use the necessary formal terms. Symbolically, we can write

$$P_{seq} \wedge (k = M - 1) \Rightarrow R \tag{5.20}$$

(The code will actually terminate with $k = M$, but this does not affect equation (5.20)). The symbol $\wedge$ is used for the logical AND operation; this is standard practice in the computer science literature and is therefore retained in this thesis.

Another possible invariant is the following:

$$P_{par} = (\ \forall i \ \in \ \mathbf{K} : c_i = a_i + b_i); \mathbf{K} = \{0, 1, \ldots, M - 1\} \tag{5.21}$$

What we have now achieved is to remove the sequential index incrementing, and indicate clearly that the *order* in which the components of $[c]$ are computed (using the above code stub) is irrelevant. Furthermore,

$$P_{par} \wedge (|\mathbf{K}| = M) \Rightarrow R \tag{5.22}$$

in other words, the truth of the postcondition is established. $|.|$ is the "size of" operator.

5.4.2 The Matrix-Vector Product Revisited Using Formal Methods

The preceding example is given in [vdV88]. Another, new, example will be considered, namely the matrix vector product $[c] = [A][b]$, which was of course parallelized in the previous chapter. In this case, the necessary postcondition is

$$R \equiv (\ \forall i : 0 \leq i \leq M - 1; c_i = \sum_{j=0}^{M-1} a_{i,j} b_j) \tag{5.23}$$

One possible invariant is the following:

$$P_{par} = (\forall i : i \in \mathbf{K}; c_i = \sum_{j=0}^{M-1} a_{i,j} b_j);\ \mathbf{K} = \{0, 1, \ldots, M - 1\} \tag{5.24}$$

This corresponds to the row-block decomposition with a clustering by rows. The maximum parallelism is M, the number of rows. This formal notation makes clear the independence of the order of row computation and hence a possible parallization technique. The *complexity* (order of operation count) is M^2/N. With M processors, a lower bound on the complexity of the row-block decomposition is M. (The complexity of the serial algorithm is M^2). The truth of the postcondition R may be established as in equation (5.22).

It must be noted that these methods do not necessarily indicate *all* the potential parallelism. Returning to the matrix-vector product problem, another valid invariant is the following:

$$P'_{par} = (\forall i,j : i \in \mathbf{I}, j \in \mathbf{J}; d_{i,j} = a_{i,j}b_j) \; ; \; \mathbf{I},\mathbf{J} = \{0,1,\ldots,M-1\} \tag{5.25}$$

$$P'_{seq} = (\forall i : 0 \le i \le k; c_i = \sum_{j=0}^{M-1} d_{i,j}) \; ; \; 0 \le k \le M-1 \tag{5.26}$$

and

$$[P'_{par} \wedge (|\mathbf{I}| = M) \wedge (|\mathbf{J}| = M)] \wedge [P'_{seq} \wedge (k = M-1)] \Rightarrow R \tag{5.27}$$

The first invariant is the parallel matrix-vector product; the complexity, M^2/N, assuming a $\sqrt{N} \times \sqrt{N}$ lattice, so given sufficient processors the lower bound on the complexity is 1. The second, sequential, invariant is the *accumulate* paradigm described in Chapter 4. The complexity of P'_{seq} is M^2. Similar ideas can now be applied to parallelize P'_{seq}, for example by replacing P'_{seq} with P''_{par}:

$$P''_{par} = (\forall i : i \in \mathbf{K}; c_i = \sum_{j=0}^{M-1} d_{i,j}); \mathbf{K} = \{0,1,\ldots,M-1\} \tag{5.28}$$

This indicates that the summations required for each element of the vector can be performed in parallel. The complexity of this is $M^2/\sqrt{N}$ if the same lattice used for the multiplications is used; given the same maximum number of processors, the lower bound on the complexity is M. But this still does not indicate all the potential parallelism. The summations for each vector can be parallelized by summing columns in parallel; for example on an 8×8 processor array, columns 0 and 1, 2 and 3, 4 and 5 and 6 and 7 can all be summed in parallel in one step, leaving four columns. The same can be repeated at the next step, leaving two columns. In the third and final step, one column is left — which is the required vector. Given M^2 processors, a lower bound on the complexity of this operation may be shown to be $\log_2 M$, assuming M to be a power of 2. Hence a lower bound on the complexity of the column-block decomposition is $1 + \log_2 M$, using M^2 processors. This should be compared with the lower bound on the row-block decomposition of M, using M processors.

It will be noted that these lower bounds entirely ignore the effect of *communication* — which has been shown to be very important in Chapter 4 — so these results are mainly of theoretical interest. Nonetheless, they indicate the utility of extracting parallelism using formal methods.

5.5 A Parallel LU Algorithm Derived Using Formal Methods

The following algorithm[7] assumes the choice of diagonal element

$$l_{s,s} = 1 \ , \forall s : 0 \leq s \leq M-1 \tag{5.29}$$

Firstly we write the post-conditions[8], which are derived by the use of the triangular properties of [L] and [U] and equation (5.29):

$$\forall s \leq t \ : \ a_{s,t} = u_{s,t} + \sum_{j=o}^{s-1} l_{s,j}u_{j,t} \tag{5.30}$$

$$\forall s > t \ : \ a_{s,t} = l_{s,t}u_{t,t} + \sum_{j=o}^{t-1} l_{s,j}u_{j,t} \tag{5.31}$$

To put this a slightly different way, the problem is to find the matrix [X] such that

$$x_{s,t} = a_{s,t} - \sum_{j=0}^{s-1} l_{s,j}u_{j,t} \ ; \ \forall s \leq t \tag{5.32}$$

$$x_{s,t}u_{t,t} = a_{s,t} - \sum_{j=0}^{t-1} l_{s,j}u_{j,t} \ ; \ \forall s > t \tag{5.33}$$

$[X]$ represents [L] and [U]; the lower triangular part is [L], and the diagonal and upper triangular part is [U], or formally, $l_{s,t} = x_{s,t} \ ; \ \forall s > t \ ; u_{s,t} = x_{s,t} \ ; \ \forall s \leq t$. Now a sum function $f(s,t,k)$ is defined:

$$f(s,t,k) = a_{s,t} - \sum_{j=0}^{k-1} l_{s,j}u_{j,t} \ ; \ \forall k : 0 < k \leq M-1 \tag{5.34}$$

[7]This section follows [vdV88] in spirit, but not precisely in detail. van de Vorst's original work has been expanded in places, simplified in others and re-written for clarity in yet others. In addition, later work by van de Vorst [vdVB89] has also been included.

[8]These are valid $\forall s,t > 0$; for $s,t = 0$, the required postcondition is the term before the summation only. This special case is trivial and will not be explicitly indicated in the rest of the analysis; however, the code that implements this must of course implement this case correctly.

and the post-condition is formalized:

$$R \equiv \{\forall s,t\, R[s,t] : 0 \leq s,t \leq M-1\} \tag{5.35}$$

This post-condition is a *set* of local post-conditions, defined as:

$$R[s,t] \equiv (x_{s,t} = f(s,t,s) \wedge s \leq t) \vee (x_{s,t}u_{t,t} = f(s,t,t) \wedge s > t) \tag{5.36}$$

In $f(s,t,v)$ the first two indices (s and t) refer to the indices of the element x — representing either l or u — whereas the last index v refers to the upper summation index. The symbol $\vee$ is used for the logical OR operation.

The following should be noted:

- there are M^2 post-conditions
- a start can be made on establishing the truth of each post-condition $R[s,t]$ as soon as any elements of row s of $[L]$ and the corresponding elements of column t of $[U]$ are available

This second point is formalized by introducing a new variable k, which will play the role of a global "clock"[9] for the processor array. As a first guess, the invariant $x_{s,t} = f(s,t,k) \wedge (0 \leq k \leq \min(s,t))$ is thus obtained, and the following set of *set* of invariants is *proposed*

$$P \equiv \{\forall s,t\, P[s,t] : 0 \leq s,t \leq M-1\} \tag{5.37}$$

$$\begin{aligned} P[s,t] \;\; \equiv \;\; & \{[x'_{s,t} = f(s,t,k) \wedge 0 \leq k \leq \min(s,t)] \\ & \vee [R(s,t) \wedge \min(s,t) < k \leq M]\} \end{aligned} \tag{5.38}$$

Note that this is only one possible set of invariants; others may well be admissible. Unlike the trivial example of Section 5.4.1, the code required to maintain the validity of the invariants as k is increased is not as immediately obvious; the necessary initialization, and the operations required as k is incremented, will now be established.

The algorithm is initialized as follows:

$$\begin{aligned} k \;&:=\; 0 \\ [X] \;&:=\; [A] \end{aligned} \tag{5.39}$$

It is useful to identify the following four different regions:

- *active* if $k < min(s,t)$

[9]On a transputer array, the "clock" is simulated by an explicit synchronizing message.

- *critical* if $k = t \wedge s > t$ (all non-passive l in column k)
- *pseudo-critical* if $k = s \wedge s \leq t$ (all non-passive u in row k)
- *passive*, if $k > min(s, t)$

The graphical interpretation of these is shown in Figure 5.7.

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

$\cdot$	passive elements
$\bullet$	critical elements
$*$	pseudo-critical elements
$\circ$	active elements

Figure 5.7: Graphical interpretation of regions defined in pseudo-code

As the "clock" k is incremented, the invariants must remain valid. For active processes, the following code fragment must be executed:

```
x[s,t] := x[s,t] - l[s,k] u[k,t]
k := k +1
```

For critical processes, the following code fragment must be executed:

```
x[s,t] := x[s,t]/u[k,k]
k := k +1
```

For pseudo-critical and passive processes, incrementing k leaves $P[s,t]$ invariant, hence no action is required.

These computations may each be performed concurrently. This is the crucial fact that the application of the formal methodology extracts; it does this by explicitly indicating that each post-condition may be satisfied *incrementally*, via the variable k, rather than all at once as in the conventional serial formulations.

The maximum degree of parallelism is M^2, since there are that number of post-conditions P[s,t], but the complexity has a lower bound[10] of M, not

[10]The operation count is $2M$, when the *sequencing* of operations within a step are taken in to account, as already discussed in Section 5.3.2.

1, since the computations for row (and column) $k+1$ cannot be finalized until row (and column) k have been computed. Another way of stating this is that the computations are ordered, and van de Vorst goes on to use the "partial" ordering of the post-conditions to examine deadlock, i.e. does P[i,j] need the results of P[k,l] but similarly does P[k,l] need the results of P[i,j] to continue? He shows theoretically that it does not occur; however, once the algorithm is understood as explained in Section 5.3.2 it is obvious that deadlock cannot occur.

5.6 Pseudo-code for the LU algorithm

With the formal development completed, it is useful to recap how the algorithm proceeds. The initialization of equation (5.39) establishes the first row of [U] — actually before the algorithm has started.

- 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 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 $k = M$.

Figure 5.8 shows the communications executed by the algorithm. In the latter 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 $\rightarrow$.

Pseudo-code for the algorithm is given in Figure 5.9. Note that the pseudo-code assumes M^2 processors; if this is not the case, then *clustering* as described in Section 5.3.2 and also Section 5.8 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.

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

$\cdot$	passive elements
$\bullet$	critical elements
$*$	pseudo-critical elements
$\circ$	active elements

Figure 5.8: Communication in parallel LU algorithm.

5.7 Pivoting

One point that has not been considered thus far in the analysis is pivoting. Pivoting is a strategy to optimize numerical stability by ensuring that the largest (in some sense) element is on the diagonal. Two possibilities are listed below:

- *maximal column* pivoting: Select the element in the same column that is below the diagonal and has the largest absolute value, and interchange rows to carry this element onto the diagonal. This algorithm is also known as *partial* pivoting, since only the part of the matrix below the diagonal is considered. The first step requires M comparisons, the second M-1 etc. Using equation (5.15) the number of comparisons can be shown to be $\frac{1}{2}M^2 + O(M)$.

- *maximal* pivoting: At step k, search all the elements $a_{i,j}$ with $i, j > k$, and then interchange rows and columns to carry this element onto the diagonal. To use the notation previously defined, this requires a search of all the *active* elements. Hence the other name for the method, viz. *total* pivoting. This can be shown to require $\frac{1}{3}M^3 + O(M^2) + O(M)$ comparisons [BF85, p.329].

van de Vorst has shown that partial pivoting may be incorporated into a parallel LU algorithm without a major effect on the efficiency of the algorithm; however, the coding becomes even more complicated than that already required.

The author did not incorporate pivoting into the parallel code he developed, because pivoting was thought to be unnecessary with the typical

```
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!}
          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 5.9: Pseudo-code for the parallel LU algorithm; adapted from [vdV88].

systems generated by MoM discretizations, for the reason that such a system, although not strictly diagonally dominant[11], normally has its largest elements on the diagonal, since these represent self-impedance terms and all the other off-diagonal elements represent mutual impedance terms. (This argument assumes a more or less uniform discretization; it will not necessarily be true if very different segment lengths were used on different parts of the structure.) In Section 5.13 it is shown that where a LU decomposition has been shown to fail, partial pivoting was of no help in any case[12].

5.8 Theoretical Timing Equations for the Parallel Algorithm and the Clustering Strategy

The load balancing problem of the algorithm has already been mentioned and a solution proposed in terms of the double-interleave distribution. It was mentioned that the lower bound on efficiency was about 33%; this shall now shown formally.

Consider the extreme case where the number of processors $N = M^2$. As has been discussed in Section 5.3.2, the algorithm then terminates in $2M^2$ steps. The serial time has been shown to be $\frac{2}{3}M^3$; see Section 5.2. Hence the speed-up is $M/3$, and the efficiency 33%.

This is a lower bound on the efficiency as a result of load balancing; in any *real* case of interest $N \ll M^2$ and the actual effect of load balancing will not be quite as serious. Nonetheless, the effect is sufficiently serious to warrant attention. The solution has already been given, namely the double interleaved scheme. More formally, let

$$\mathbf{V} \equiv \{s : 0 \leq s < M\} \tag{5.40}$$

Then we partition $\mathbf{V}$ into $\sqrt{N}$ partitions $\mathbf{V_i}$ and $\mathbf{W_j}$, each of size $\frac{M}{\sqrt{N}}$:

$$\{\mathbf{V_i} : 0 \leq i < \sqrt{N}\} \tag{5.41}$$

[11] A strictly diagonally dominant matrix is defined by the property that $|a_{i,i}| > \sum_{j=0,j\neq i}^{M-1} |a_{i,j}|$. It can be shown that the LU decomposition of such a system does not require any pivoting and the computations are stable with respect to rounding errors [BF85, p.335].

[12] Work published by Cwik [Cwi91], just prior to submission of this thesis, on the parallel MoM codes with LU solvers developed at the Jet Propulsion Laboratory in Pasadena, California, mentioned that their parallel LU solvers perform a substantial amount of pivoting for a typical MoM run; further work is needed before a definitive statement on the necessity or otherwise of pivoting for linear systems generated by the MoM can be made.

and

$$\{\mathbf{W_j} : 0 \leq j < \sqrt{N}\} \tag{5.42}$$

The $\sqrt{N} \times \sqrt{N}$ Cartesian distribution of elements on processors is given by

$$\mathbf{C} \equiv \{\mathbf{V_i} \times \mathbf{W_j} : 0 \leq i, j < \sqrt{N}\}$$

Note that the indices i, j refer to *processor* indices on a square mesh; see Figure 5.5. It is assumed that M is an integer multiple of $\sqrt{N}$ in this analysis. The case where this is not so requires padding; this is considered in Section 5.11.

The previously discussed double-interleaved Cartesian distribution is defined formally as $\mathbf{G}$:

$$\mathbf{G} \equiv \{\mathbf{G_i} \times \mathbf{H_j} : 0 \leq i, j < \sqrt{N}\} \tag{5.43}$$

with

$$\mathbf{G_i} \equiv \{s : s \in \mathbf{V} \wedge s \operatorname{mod}\sqrt{N} = i\} \forall 0 \leq i < \sqrt{N} \tag{5.44}$$

$$\mathbf{H_j} \equiv \{t : t \in \mathbf{V} \wedge t \operatorname{mod}\sqrt{N} = j\} \forall 0 \leq j < \sqrt{N} \tag{5.45}$$

and mod(a) the modulo(a) operator.

For the case illustrated in Figures 5.5 and 5.6:

$$\begin{aligned} \mathbf{G_0} &= \{0, 3, 6\} \\ \mathbf{G_1} &= \{1, 4, 7\} \\ \mathbf{G_2} &= \{2, 5, 8\} \end{aligned} \tag{5.46}$$

and similarly

$$\begin{aligned} \mathbf{H_0} &= \{0, 3, 6\} \\ \mathbf{H_1} &= \{1, 4, 7\} \\ \mathbf{H_2} &= \{2, 5, 8\} \end{aligned} \tag{5.47}$$

The Cartesian product $\mathbf{G_0} \times \mathbf{H_0}$ gives the indices of the 9 elements clustered on processor$_{00}$[13]. The full distribution $\mathbf{G}$ is shown in Figure 5.6.

It may be seen by inspection that with this double-interleaved scheme, only on the final $\sqrt{N}$ steps is any processor left with no work to do and the impact is thus minimal for any reasonably large grained problem. This may be confirmed by the following analysis, which establishes an upper bound on the overall computation count and from this, the load-balancing count. The

[13]On processor$_{00}$, a_{11} is thus a_{33} of the original matrix. This shows the difference between the local and global indices, and is another complication that must be taken care of in the code.

maximum load is carried by processor$_{\sqrt{N-1}\sqrt{N-1}}$ (the processor at the lower right of the processor array). 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); 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

$$\frac{2}{3}\frac{M^3}{N}+\frac{M^2}{\sqrt{N}} \tag{5.48}$$

The first term is clearly the parallelized computations; thus the second term is the additional computational overhead caused by load-balancing. This is confirmed by van de Vorst's analysis of load balancing for a general rectangular mesh. For the special case of a square mesh as used by the present author, he obtains the same upper bound on the load-balancing operation count; this is the second term in equation (3.11) [vdVB89][14].

For the communication count, the following should be noted. At step k, the algorithm requires the row broadcast of all the elements of column k of $[L]$ that have just gone critical, and a column broadcast of all the elements of row k of $[U]$ that have just gone pseudo-critical. These can be broadcast using two concurrent *pipelines*[15] for efficiency. An upper bound for the communication can be derived as follows:

Consider the *processor* column carrying the heaviest communication load. By inspection, it is the most right-most column. For the first $\sqrt{N}$ steps, the amount of data to be communicated is $\frac{M}{\sqrt{N}}$. For the next $\sqrt{N}$ steps — the algorithm has completed one cycle through the processor array and has now returned to the first processor[16] — the amount of data is $\frac{M}{\sqrt{N}}-1$. The upper bound on the communication count [17] is thus

$$t_{mesh} \leq \{[(\frac{M}{\sqrt{N}})\sqrt{N}]+[(\frac{M}{\sqrt{N}}-1)\sqrt{N}]+\ldots+[(1)\sqrt{N}]\}t_{comm} \tag{5.49}$$

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

$$t_{mesh} \leq \{\frac{M^2}{\sqrt{N}}-\sqrt{N}\sum_{k=0}^{\frac{M}{\sqrt{N}}-1} k\}t_{comm} \tag{5.50}$$

[14]The author checked most of the results in van de Vorst's work; only one error was noted. In [vdV88, Section 6.5] the first term in $E(grids)$ should be $\frac{n^2}{3}$, not $n^{\frac{2}{3}}$.

[15]Pipelines are discussed in detail in Section 5.10.

[16]Remember that the distribution is modulo$\sqrt{N}$.

[17]The pipelines are modelled simply by the time required to output the data; it is assumed that the vector lengths to be output are much greater than the diameter of the mesh, so that complex pipeline models of the type discussed in Section 3.3 are not required. The set-up time for a pipeline on a transputer array using Occam 2 is negligible.

which can be simply manipulated using equation (5.15) to give

$$t_{mesh} \leq \frac{1}{2}\frac{M^2}{\sqrt{N}} + O(M) \tag{5.51}$$

This result is also given in [vdVB89, equation 3.19]. (If pivoting is included, the result is very similar but with a constant of $\frac{3}{2}$ for the dominant $\frac{M^2}{\sqrt{N}}$ term [vdVB89, equation 3.20]). van de Vorst's full communication analysis was checked by the author, with special regard for the highest order terms. Some minor differences were observed in the lower order terms.

It can be easily seen from the preceding why a mesh distribution is better than either a column or row distribution. With either of these, 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, the column and row broadcast pipelines run concurrently with the grid distribution. It may be shown formally that for a local memory MIMD array, as regards communication, a square grid is the optimal grid distribution of the general class of rectangular grid distributions for this parallel LU algorithm [vdV88, vdVB89].

A theoretical model for the efficiency will now be derived. The serial time, using a conversion factor from complex to real flops[18] of 4 , is $(\frac{8}{3}M^3)t_{calc}$; the parallel time is the sum of the parallelized computations, viz. $(\frac{8}{3}M^3/N)t_{calc}$, the load-balancing term (from van de Vorst's analysis, as discussed above) is $(4M^2/\sqrt{N})t_{calc}$ and the communication term is $(\frac{1}{2}M^2/\sqrt{N})t_{comm}$. Summing the last three, using equation (3.3) and simplifying yields

$$\epsilon \approx \frac{1}{1 + \frac{\sqrt{N}}{M}(\frac{3}{2} + \frac{3\beta}{16\gamma})} \tag{5.52}$$

The symbol β has the meaning defined in Chapter 4, viz. $\frac{t_{comm}}{t_{calc}}$.

It is interesting to compare this result with that for the CG solver, equation (4.40), repeated here for convenience:

$$\epsilon \approx \frac{1}{1 + \frac{N}{M}(2.75 + 0.125d + \frac{\log_2 N\beta}{8})} \tag{5.53}$$

It is notable that the dominant terms in the denominator have a $\frac{N}{M}$ multiplier in the CG case, the reciprocal of the number of *rows of the matrix* per processor, whereas in the LU case, the multiplier is $\frac{\sqrt{N}}{M}$, the reciprocal of the square root of M^2/N, this last term being the number of *matrix elements*

[18]The number of additions and multiplications is almost identical, and the former has a conversion factor of 2, the latter 6.

per processor. The latter is the smaller multiplier — *this indicates that the LU algorithm scales better than the CG algorithm*[19]. This is an impressive result, considering how initially unsuitable for parallelism the LU algorithm appeared, and is confirmed by the results in Section 5.12. It is also interesting to note that for typical values of β (between 3 and 4 approximately for the MC^2, as shown in Chapter 4), the load imbalance term is of the same order of importance as the communication term.

5.9 Parallel Forward and Backward Substitution

Following the factorization of $[A]$ into the product of $[L]$ and $[U]$, the unknown LHS is solved for in a two-step process; see Section 5.2, equations (5.12) to (5.14).

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[20] and the pseudo-code is given in Figures 5.10 and 5.11.

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.

5.10 Coding for Maximum Efficiency

The aim of developing a parallel algorithm is obviously to obtain the maximum reasonably possible speed-up, and in this section, some important gen-

[19]This remains true even if pipelining was fully exploited in the CG algorithm; the effect of this is to replace the $\log_2 N$ term by 1

[20]van de Vorst and Bisseling [vdVB89] mention the existence of parallel substitution algorithms developed by them, but these had not been published in readily accessible journals at the time that the author was carrying out his own research.

```
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 5.10: Forward substitution pseudo-code; solve [L][z]=[b]

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

Figure 5.11: Backward substitution pseudo-code; solve [U][x]=[z]

```
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 5.12: Pseudo-code for rightwards pipelined column broadcast procedure: worker

eral techniques that were used for the parallel LU algorithm are described in detail.

Firstly, it has already been shown that the links can operate in parallel. This has been exploited in the CG algorithm, see Figures 4.1 and 4.2 for the broadcast process. But this technique can be refined even further by using a *pipeline*, where outputting of the vector is overlapped with the inputting thereof, so that the input and output links are overlapped in time. Note that a pipeline has thus been implemented on a fundamentally *replicated* system. Such methods are quite advanced and were not used in the CG algorithm, which was one of the first parallel programs written by the author, but were implemented in the LU algorithm.

Pseudo-code for the pipelined rightwards broadcast of columns procedure is given in Figure 5.12. A similar procedure is assumed for the leftwards broadcast, since the direction of communication varies during a "sweep". The `alt` construct was used to detect which input channel was active. This is shown in Figure 5.13. A similar procedure is needed for the row broadcast. The row and column broadcast procedures must run concurrently for optimum efficiency, as shown in Figure 5.14. For coding efficiency reasons, the Occam code differs slightly from the pseudo-code, although the principles are identical.

On the general subject of coding for maximum efficiency, it should be

```
procedure broadcast_column
begin
  alt
    receive length from left processor
      call procedure broadcast_column_to_right(length)
   receive length from right processor
      call procedure broadcast_column_to_left(length)
end{procedure broadcast_column}
```

Figure 5.13: Pseudo-code for pipelined column broadcast procedure: worker

```
procedure broadcast
begin
  par
    call broadcast_column
    call broadcast_row
  end{par}
end{procedure broadcast}
```

Figure 5.14: Pseudo-code for broadcast procedure: worker

```
begin{gather}
  par
    receive one part of vector
    receive a different part of vector
  end{par}
end{gather}
```

Figure 5.15: Pseudo-code for a valid Occam construct that is not accepted by the compiler

noted that there are some valid constructs that the Occam compiler does not (by default) permit one to use. The construct shown in Figure 5.15, although valid, is rejected by the compiler[21].

The reason for this is that Occam does not permit parallel access to an array — even though the elements of the array can be established as disjoint at compile time[22]. The author worked around the problem by establishing two separate arrays, and then building up the total array once the two halves have been received. Alternately, one can disable the compiler's *usage checking* — a dangerous idea since the usage checker picks up many potential faults. Sacrificing a very small loss in speed for comprehensive error checking is, to the author, quite acceptable.

5.11 Some Coding Details

Pipe-lining, as expounded in the preceding section, was exploited in the coding of this algorithm; the vertical (column) and horizontal (row) pipes were run concurrently for maximum speed-up. The section of the code corresponding to the active processes was coded as efficiently as possible, since this is a time-critical part of the algorithm. *Padding*, required to ensure an integral number of unknowns per processor, is not as simple as in the CG case, where additional rows and columns of zeros were introduced[23]. A check was made

[21]This construct was not required in the previous pipelining examples shown, but was required elsewhere in the CG and LU codes.

[22]INMOS admits that this is a bug

[23]Padding has a slight effect on run-time; as noted previously on page 58, if measured with sufficient resolution, it will be found that the graph of efficiency is actually a stairstep, not a smooth, function. The run-time increases from the value predicted for the actual value of $M/\sqrt{N}$ to that predicted for $\lceil M/\sqrt{N} \rceil$. For example, a problem with 10 unknowns on a 3×3 processor array, i.e. $M/\sqrt{N} = 3.33$, will run at the same speed as a problem with $M/\sqrt{N} = 4$, i.e. 12 unknowns. For large M, the effect is insignificant for the values

on the local index to check whether the corresponding global index exceeded the actual matrix dimension, and if so, the process was terminated. This code used implicit termination, where each processor monitors the global index k, as opposed to the explicit termination used in the CG algorithms, where the master process monitored the termination criteria and explicitly informed the worker processes when to terminate. The clustering — the double interleaved distribution — and concurrent horizontal and vertical pipelined communications resulted in a complex (but efficient) code; the parallel LU code is about 2500 lines of Occam. By comparison, the serial LU code is about 100 lines. The reason the code is this complex is to obtain efficiency, and indicates the time and effort required to develop efficient parallel code at present.

For the initial code development stages, the matrix was read in from disk and then scattered over the processor array. When testing of the parallel performance started, the code was extended to include a thin-wire MoM formulation (which is described in detail in the Section 5.12), and the indices were set up to generate the matrix in parallel and with the correct double-interleaved distribution.

Testing the correctness of this code proved rather more difficult than for the CG code. The reason is that the CG code has fewer potential failure mechanisms, since the data distribution is far simpler. The combination of processors on the edge of the mesh requiring special handling, the double interleaved distribution and the requirement for padding proved challenging; correcting one bug tended to introduce a new and different one, or reveal another bug that was masked by the previous bug. The only debugger available was a post-mortem dump debugger; finding bugs frequently required running the code with a STOP (conditional on the processor index and/or array index) intentionally inserted to cause an error; the system could then be examined using the post-mortem dump debugger. If the guess regarding where to halt the code was incorrect, it required changing the position of the STOP and/or the conditions, re-compiling, running and then examining the results, and repeating until the fault was identified. This process is lengthy and tedious, and *the development of* **interactive** *debuggers able to examine* **concurrent** *programs is imperative for the large scale adoption of parallelism by the scientific and engineering community.*

Returning to the problem of validation, a standard test set was built up to check a standard sequence of problems of different dimensions on simulated meshes of different sizes. This permitted rapid checking of any modifications for unexpected side-effects.

As for the CG algorithm, writing general purpose configuration code was

of N available for this research.

not trivial. The actual parallel codes were run using the Occam Toolset[24], described in Section 3.4.2 It was while tests were being run that a technical problem was discovered with the MC^2; although what little documentation exists gives the impression that any valid switching satisfying the even-even / odd-odd requirements discussed in Section 4.6 can be switched, this is actually only true *on one cluster of 16 processors.* The reason is the restricted number of inter-*cluster* links. The problem is a hardware peculiarity of the particular transputer hardware available and has no effect on the basic discussion in this chapter[25].

5.12 Timing Results

The algorithms described in this chapter have been implemented by the author in Occam 2 for a transputer array. Figure 5.16 shows efficiencies for a number of different 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 [ST81, Section 7.5] for the field radiated by a sinusoidal current. This MoM code was also written in Occam 2. The largest problem solved was a 1500 unknown problem, using 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 5.17 shows theoretical predictions, which can be seen to be rather optimistic, although the general trend is correctly predicted. This is due to the rather fine grain of communication, which is difficult to model accurately, as well as the effect of loop overhead etc. To permit comparison of the parallel LU and CG algorithms, measurements for a parallel CG algorithm are also shown in Figure 5.17 for 14 transputers[26]. The CG results were measured

[24]It was initially thought that the TDS would not run on the MC^2. The Toolset has a slightly different configuration strategy to the TDS, but the details are very subtle and the effects on the basic principles unimportant. The configuration code for the parallel LU algorithm as run under the Toolset on the MC^2 is given in Appendix B.

[25]With the new C104 packet-routing switch mentioned in Section 3.4.2, the problem should not exist with the next generation of T9000 based transputer arrays using the C104 packet-routing switch.

[26]The binary tree and mesh topologies cannot use exactly the same number of processors;

with a single precision version of PARNEC. (Note that the results shown in Chapter 4 are for the double precision version of PARNEC.)

van de Vorst shows similar measured results in [vdVB89]; the numerical values for efficiency shown in Figure 5.16 are not directly comparable with van de Vorst's results, since his results are presumably for real valued matrices, although the latter is not explicitly stated in his paper. The form of the curves is nonetheless very similar, and the numerical values fairly close.

5.13 Accuracy Studies

The availability of a MoM code able to handle massive data structures permits the exploration of the accuracy of the code, by investigating the convergence of the input impedance of a thin wire. The code used the "frill" generator model [ST81, Section 7.7]. Results are shown in Table 5.1 for the convergence of the input impedance of a centre-fed thin wire dipole for a fairly thin wire[27]. The wire radius a and equivalent coaxial radius b of the "frill" generator were chosen to give $Z_O = 50\Omega$ in an air-filled cable. The results were obtained using a double precision code. For the 1499 segment case, the input impedance was checked using a version of the same MoM code, but using the CG matrix solver described in Chapter 4. The input impedance computed with the CG solver was $2 \times (43.89 + j25.25)\Omega$[28]; the difference between the LU and CG results was thus about 0.4%. The specified residual norm tolerance on the CG solver was 1%, so the difference between the impedances is within the specified accuracy of the CG solution.

With single precision versions of the these codes, the difference between the impedance computed with the CG and LU solvers was about 7%. The single and double precision versions (of both the CG and LU solvers) gave results differing by about 20% for the reactance, with a normalized residual of 1% specified for the CG solver. Clearly the precision with which the matrix entries are *generated* affects the solution for such large problems, and indicates that in the 1499 segment case, double precision was required. This is a function of the problem; results shown in Section 6.7 for more realistic problems using similar numbers of unknowns indicate that single precision is acceptable.

The results display several interesting phenomena. Firstly, the results

a tree of 14 and a mesh of 16 is a fair comparison.

[27]Due to a coding error, the results shown in these tables must be multiplied by 2 to obtain the actual dipole impedance. This will be indicated by an explicit factor of 2 in the discussions in this section. The results for conductance and susceptance shown in Figures 5.18 and 5.19 have been corrected.

[28]See previous footnote.

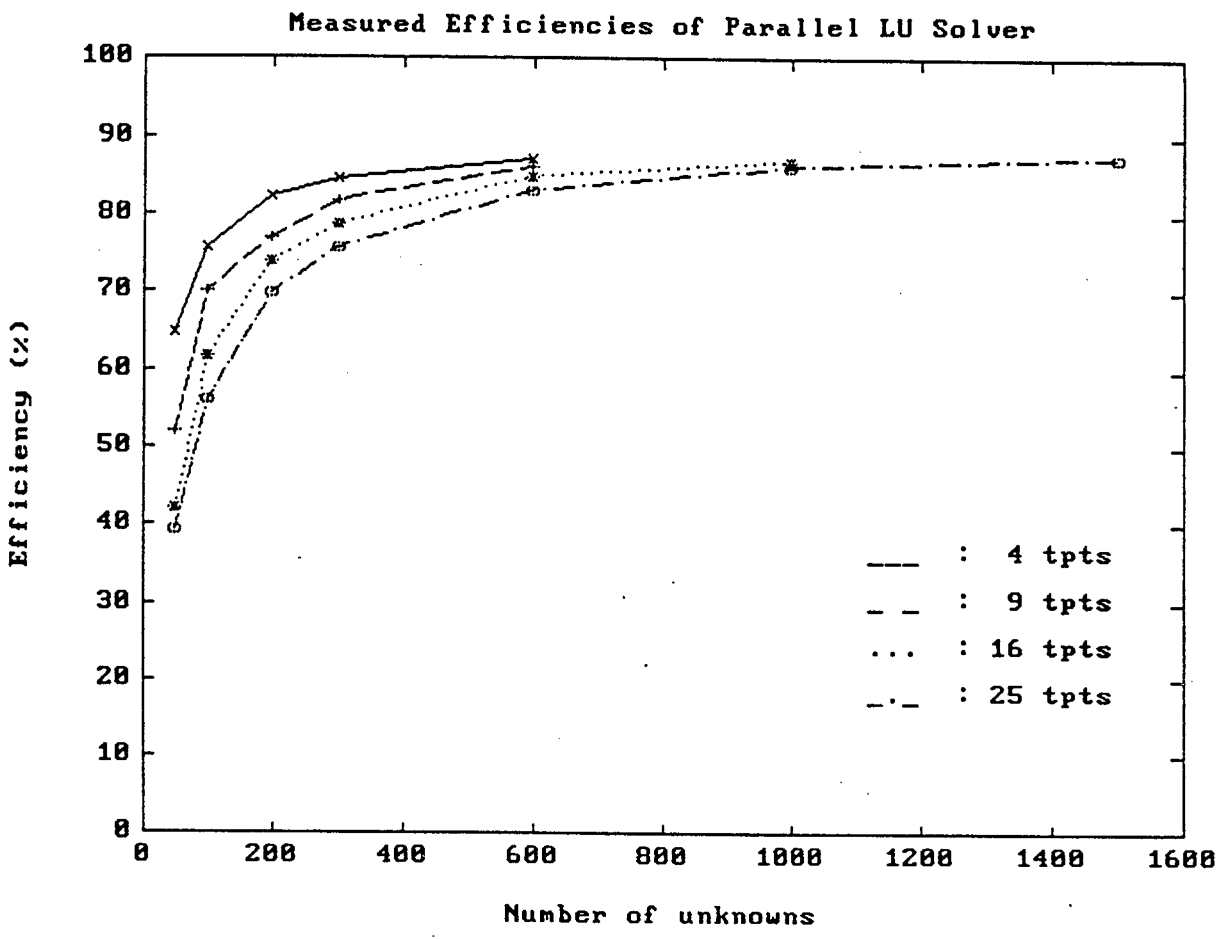


Figure 5.16: Measured efficiencies of the single precision parallel LU solver.

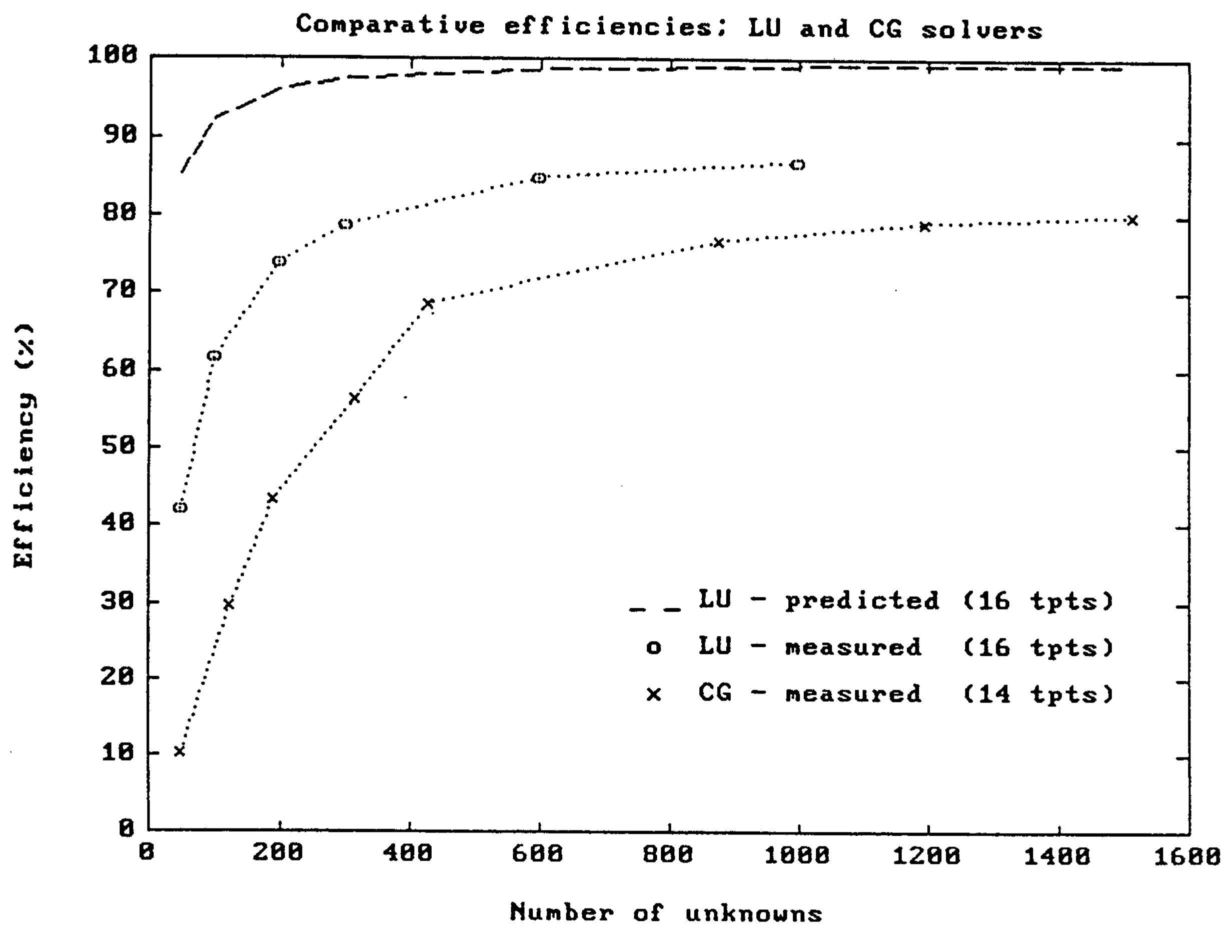


Figure 5.17: Comparative efficiencies for the single precision LU and CG solvers for similar numbers of transputers.

obtained for the input impedance using the customary guideline of 10 segments per wavelength (see Chapter 2) were worthless; a much higher order of discretization is required. This is due to the very thin wire used; as the wire was made thicker, so did a smaller number of unknowns generate more useful data. Secondly, the impedance appears[29] to genuinely converge, in contrast to results frequently reported using other models (such as delta-fed gaps and impressed E-field models). This is shown clearly in Figures 5.18 and 5.19 for the conductance and susceptance of the input admittance[30] plotted against the number of unknowns; the latter parameter is plotted logarithmically[31]. By way of comparison, the second order King-Middleton solution gives approximately $2 \times (43.1 + \jmath 20.4)\Omega$; this result was obtained by interpolating linearly from [Ell81, Table 7.5, p.315]. Finally, the LU code generates correct results with large matrices — for this particular geometry.

The experiment was repeated using a somewhat thicker wire. Much more rapid convergence was noted — but also a very interesting breakdown of the algorithm at between 499 and 549 segments. The negative input impedance predicted by the code for 549 segments is physically impossible for a passive system; this is an example of the use of basic physics to check the operation of a numerical code as proposed by Miller [Mil88]. Results are shown in Table 5.2. Results computed for more than 549 segments are not shown since they are worthless. The results were checked using a double precision CG solver and 1499 segments; the program terminated after 2998 iterations with a normalized residual of about 0.2% and gave the input impedance as $2 \times (58.02 + \jmath 28.6)\Omega$. (For this case, the CG code was set to terminate when the normalized residual had decreased below 0.01% or when the number of iterations exceeded twice M). This result agrees very well with the 400 segment case, so it can be concluded that the CG solution is valid. Hence we can conclude that the double precision LU solver is not reliable beyond about 400 segments for the particular case under consideration. Using single precision, the problem manifests itself with fewer unknowns, as expected:

[29]Following examination of this thesis, it was been pointed out[Mil91a] that for Table 5.2, the thin wire requirement $s > L/2a$, where s is the segment length, is violated for $M > 25$. Similarly for Table 5.1, the thin wire requirement is violated beyond $M > 250$. Hence, although the results show convergence, one should be cautious in drawing the conclusion that the results have converged to the *correct* impedance value, especially in the case of Table 5.2. It has also been commented [Mil91a] that for the thinner wire, convergence should be more rapid since the current is better approximated as a sinusoid. The results shown here, intended primarily to validate the LU code for large problems, would benefit from further investigation of the electromagnetic aspects. Such research should also take note of Janse van Rensburg's work [JvR90].

[30]Admittance is the inverse of impedance; conductance and susceptance are respectively the real and imaginary parts of the admittance.

[31]This plotting procedure was suggested to the author by Miller [Mil91b].

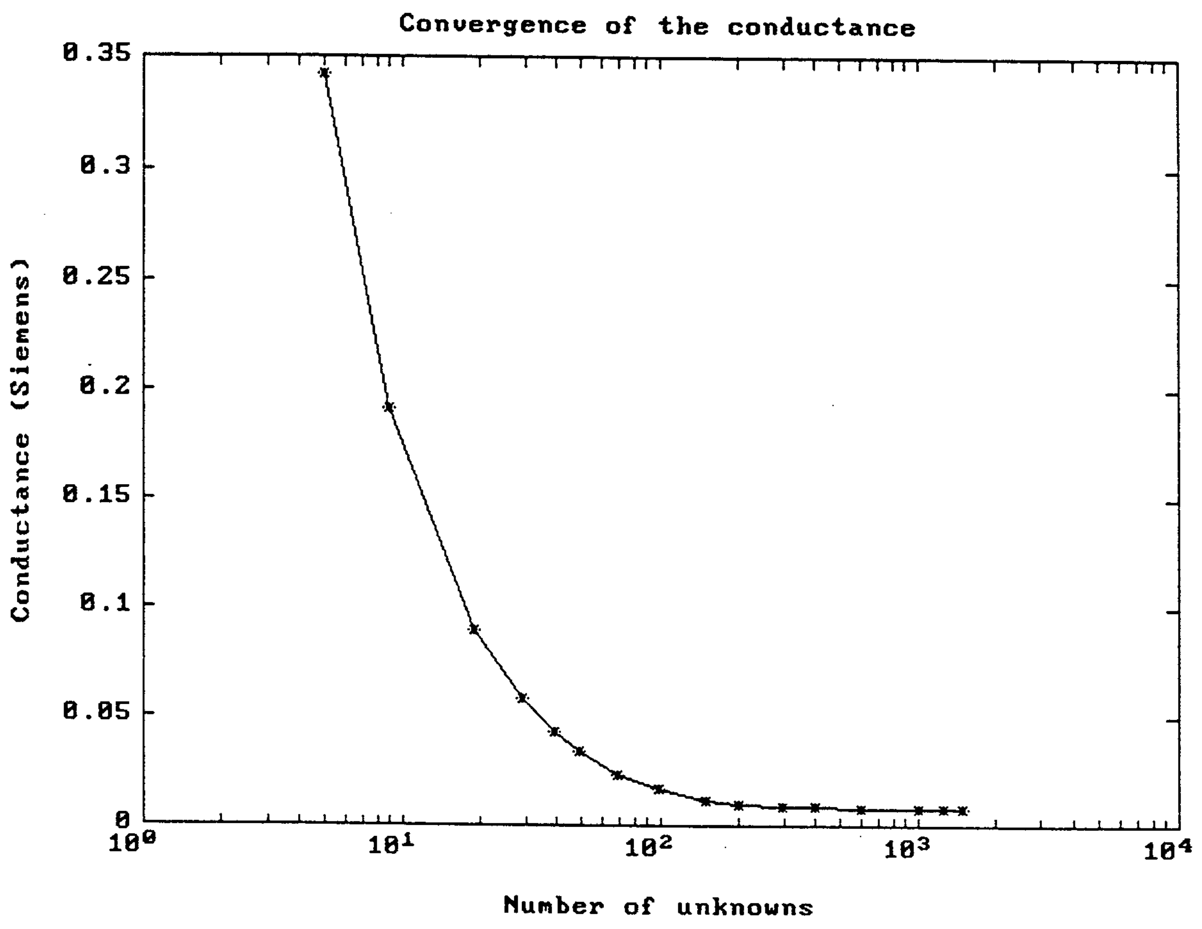


Figure 5.18: Convergence of the conductance of a dipole, $\frac{L}{\lambda} = 0.5$, $\frac{a}{\lambda} = 0.001$,$\frac{b}{\lambda} = 0.0023$

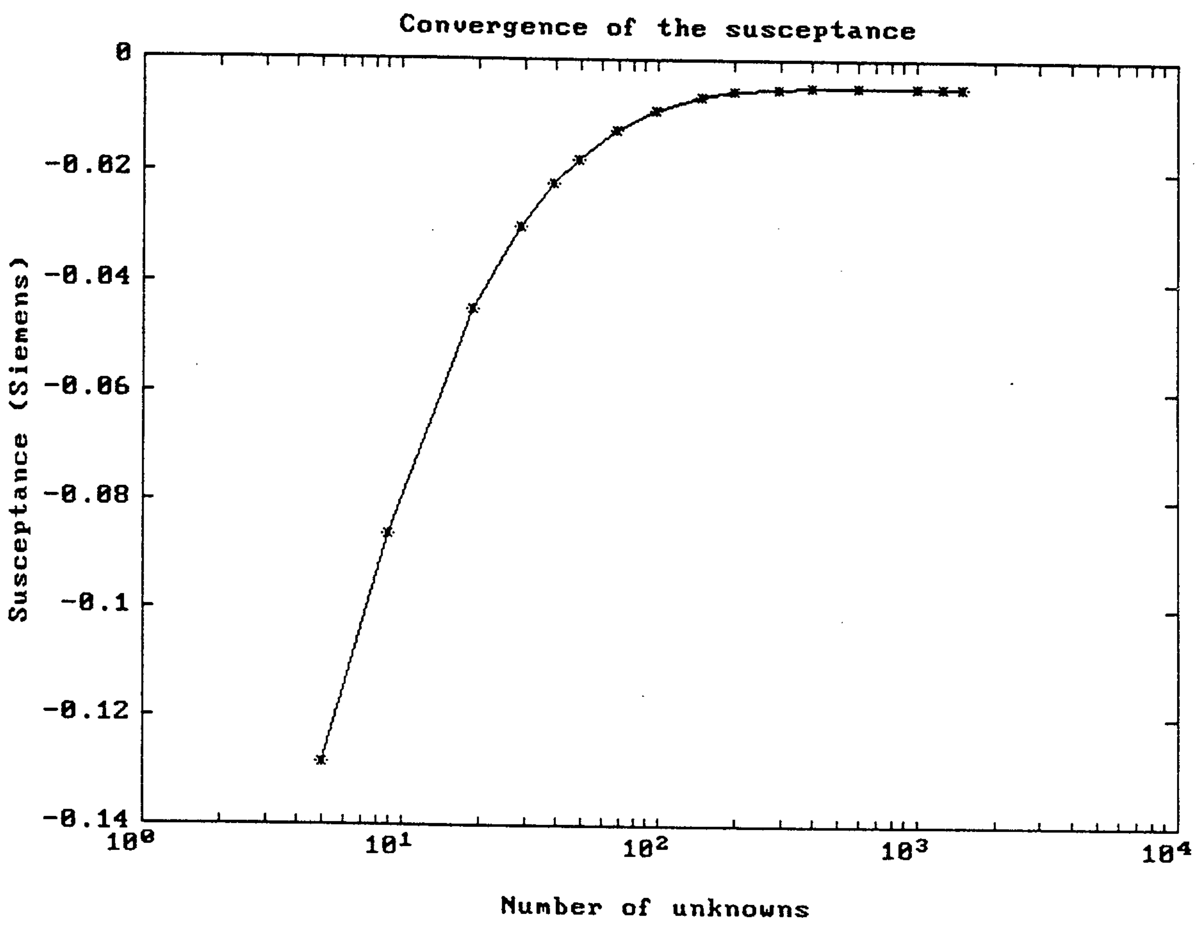


Figure 5.19: Convergence of the susceptance of a dipole, $\frac{L}{\lambda} = 0.5$, $\frac{a}{\lambda} = 0.001$,$\frac{b}{\lambda} = 0.0023$

Number of Segments	Re $Z_{in}/2$ (Ω)	Im $Z_{in}/2$ (Ω)
5	1.28	0.48
9	2.17	0.98
19	4.45	2.24
29	6.79	3.52
39	9.19	4.82
49	11.58	6.12
69	16.33	8.69
99	22.97	12.29
149	31.65	17.07
199	37.08	20.16
299	41.67	22.96
399	42.92	23.90
599	43.41	24.43
999	43.57	24.84
1249	43.62	24.99
1499	43.67	25.12

Table 5.1: Convergence of the input impedance of a dipole, $\frac{L}{\lambda} = 0.5$, $\frac{a}{\lambda} = 0.001$,$\frac{b}{\lambda} = 0.0023$. Double precision LU code.

the problem was encountered with between 250 and 300 unknowns for the same wire shown in Table 5.1 when single precision was used.

Number of Segments	Re $Z_{in}/2$ (Ω)	Im $Z_{in}/2$ (Ω)
5	13.4	4.24
9	23.28	8.23
19	41.09	15.27
29	48.27	18.71
39	50.96	20.46
49	52.23	21.53
69	53.58	22.98
99	54.77	24.16
199	56.90	26.44
249	57.57	27.14
299	58.11	27.71
349	58.57	28.18
399	58.97	28.59
499	69.29	19.23
549	-5.05	-12.83

Table 5.2: Convergence of the input impedance of a dipole, $\frac{L}{\lambda} = 0.5$, $\frac{a}{\lambda} = 0.01, \frac{b}{\lambda} = 0.023$

This is hardly surprising; it is well known in computational electromagnetics that one cannot use very short segments with a thin-wire kernel, certainly not when the length of the segment starts becoming a fraction of the wire radius. Elliott [Ell81, Appendix E] presents a detailed analysis of the thin-wire kernel, and concludes that the approximation is valid if the computations include an integration that extends over a length of the dipole of at least plus and minus several wire diameters; compare this with a length to diameter ratio of about 0.06 at the point at which the problems were noted (for the double precision case).

The underlying theory aside, what happens numerically to the coefficient matrix in this case is that the entries just off the diagonal become so close to the diagonal that the division required in the LU algorithm cannot be computed with the necessary accuracy. It should be noted that no pivoting strategy can fix this problem; pivoting aims to get the largest (in some sense) element on the diagonal. It cannot fix the problem of the elements adjacent to the diagonal having almost the same value as the diagonal element. This was confirmed by the following: the point where the single precision LU method failed used sufficiently few unknowns to run on one transputer, and the serial algorithm *did* implement partial pivoting. The pivot elements were all the diagonals, i.e. no pivoting occurred. Even the more costly total pivoting strategy will also be of no avail here.

However, it is also clear that to generate a problem using a MoM formulation that the LU method solves incorrectly required breaking virtually every guideline for discretizing structures. Hence, the LU solver should encounter no problem with the matrices encountered in normal MoM analyses.

The condition number of a matrix gives a formal indication of the ill-posedness of a system of equations. The condition number of the square matrix $[A]$ is defined as [MK75, p.131]

$$\text{cond}([A]) \equiv ||[A]|| \cdot ||[A]^{-1}|| \tag{5.54}$$

where $||\,[A]\,||$ is the norm of the matrix [A]. Using the Euclidean norm[32], the condition number is [Mit73, p.131]

$$\text{cond}([\text{A}]) = \sqrt{\lambda_{\text{max}}/\lambda_{\text{min}}} \tag{5.55}$$

where λ_{max} and λ_{min} are the maximum and minimum (in magnitude) eigenvalues of $[A]^T[A]$. Jones also discusses condition numbers; see [Jon87, p.75].

The application of the condition number is as follows; suppose that the system of equations $[A][x] = [b]$ is perturbed as follows

$$[A][x+\delta x] = [b+\delta b] \tag{5.56}$$

Then

$$\frac{||\,[\delta x]\,||}{||\,[x+\delta x]\,||} \leq cond([A])\frac{||\,[\delta b]\,||}{||\,[b+\delta b]\,||} \tag{5.57}$$

Similarly, if $[A]$ is perturbed, then

$$\frac{||\,[\delta x]\,||}{||\,[x+\delta x]\,||} \leq cond([A])\frac{||\,[\delta A]\,||}{||\,[A+\delta A]\,||} \tag{5.58}$$

Hence if the condition number is large, then small variations in $[A]$ or $[b]$ result in large variations in the solution $[x]$. Computing the matrix condition number is expensive computationally, and the following inequality is often used to obtain a rough estimate of the condition number [Mit73, p.133]:

$$\frac{1}{M}\text{cond}_\infty([A]) \leq \text{cond}_2([A]) \leq M cond_\infty([A]) \tag{5.59}$$

[32]Also known as the 2-norm. For a matrix, $||A||_2 \equiv \max_{||x||_2=1} ||[A][x]||_2$. Thus the matrix norm is defined in terms of a vector norm.

where the ∞ subscript refers to the infinity norm, which is defined as

$$\| [A] \|_\infty = \max_{0 \le i \le M-1} \sum_{j=0}^{M-1} |a_{ij}| \tag{5.60}$$

This is the maximum row sum of $[A]$.

It would be useful to build in a matrix condition number estimator in the parallel solver. This has not been done for the parallel code, and should be addressed by future research. However, such facilities are available in MATLAB[33]. Results are shown in Table 5.3 for the 2-norm condition number of the coefficient matrix of a 0.01λ wire. *The program was unable to compute the condition number for the 399 segment case*, terminating with an error message while trying to compute the condition number, a clear indication of the ill-posedness of the problem. It is thus clear that the matrix is so ill-posed for the case of 399 (and more) segments that no reliable solution can be obtained. A common rule of thumb is that $\log_{10}(\text{cond}[A])$ gives the number of digits of accuracy lost due to round-off error [MAT89, p.3-41], and this is also shown in Table 5.3.

Number of Segments	Re $Z_{in}/2$ (Ω)	Im $Z_{in}/2$ (Ω)	cond($[A]$)	$\log_{10}$cond($[A]$)
9	23.28	8.23	83.19	1.92
19	41.09	15.27	108.4	2.03
29	48.27	18.71	103.5	2.01
49	52.23	21.53	106.0	2.02
99	54.77	24.16	114.0	2.06
199	56.90	26.44	3031	3.48
249	57.57	27.14	50 330	4.70
299	58.11	27.71	887 070	5.95
349	58.57	28.18	1.63×10^7	7.21
399	58.97	28.59	Error	Error

Table 5.3: Condition number of the coefficient matrix of a MoM formulation for a dipole, $\frac{L}{\lambda} = 0.5$, $\frac{a}{\lambda} = 0.01$,$\frac{b}{\lambda} = 0.023$

The topic of the accurate computation of the input impedance of a thin

[33]MATLAB 386 was available to the author, and illustrated graphically how much had been achieved with the parallel algorithms on the transputer arrays. The 386 PC used produced a maximum benchmarked speed of about 0.2 MFLOP/s on LU decomposition; compare this with the almost 10 MFLOP/s achieved on 25 transputers for the same operation.

wire is a very complex one [JvRM89, JvRM90, JvR90]; these experiments were performed primarily with the aim of checking the accuracy of the LU code by monitoring the convergence behavior and comparing the results computed using an LU and a CG solver for the same system of equations. However, these results also indicate the type of work possible with the parallel software tools now available for MoM codes.

The ability of the CG solver to produce excellent results in situations where an LU solver is useless was alluded to in Chapter 2; the results shown here demonstrate this property. They also show how the CG method could indeed generate a solution for a problem where internal resonance phenomena are encountered.

5.14 Conclusion

In this chapter, the basic LU algorithm has been reviewed. Then the question of parallelizing the algorithm was discussed. A simple graphical exposition of a parallel algorithm was then introduced. The use of formal methods was considered for the extraction of the parallelism; several examples were given, the matrix-vector product of Chapter 4 was re-visited and the parallel LU algorithm was analyzed using formal methods. The problem of *load-balancing* was shown to be potentially serious with a parallel LU algorithm, and a solution utilizing a row-and-column interleaved scheme was described. Pseudo-code was presented for the parallel algorithm. Parallel forward and backward substitution algorithms were proposed that use the same data decomposition as the parallel LU algorithm. Pseudo-code was given for these algorithms. Timing results obtained using a matrix generated by a simple thin-wire MoM code were presented and discussed. Some of the results presented in this chapter have been presented at an international symposium [Dav91c].

The basic algorithm described in this chapter comes from recently published work in the computer science literature [vdV88, vdVB89]: the contribution of this work on the LU algorithm is:

- confirming van de Vorst's theoretical results [vdV88, vdVB89],
- extracting the essence of the method,
- presenting new and far simpler methods for deriving the results than those of van de Vorst [vdV88, vdVB89],
- presenting a new and much easier to understand graphical exposition of the algorithm,

- implementing the algorithm to provide a working parallel LU code for MoM codes, and also providing independent experimental confirmation of van de Vorst's measured results [vdVB89].

In addition, the sections on parallel forward and backward substitution are new and of course the integration of the LU solver into a MoM code is new. The timing results for complex valued matrices values are also new, as is the comparison with the CG solver. The investigation of the accuracy of the LU solver using a problem of engineering interest in computational electromagnetics is also new; a systematic study of the convergence properties of the thin-wire collocation formulation has also been performed, using the parallel LU solver to permit the inclusion of large numbers of unknowns in the study; such a study does not appear to have been undertaken previously. It was established that the LU method only fails when the basic rules for a MoM discretization are *seriously* violated, permitting the conclusion that for large electromagnetic problems discretized according to the established rules, the LU method is accurate. It was also confirmed that the condition number does indeed provide a reliable indication of an unstable solution.

The main thrust of this thesis is *developing the tools* for the *efficient* exploitation of parallel processing for the solution of MoM problems, which this chapter has done for the LU solver — the results given in the closing section of this chapter indicate the type of work now possible with these tools, and future research topics.

Chapter 6

PARNEC — A Parallel Version of NEC2

Cry "Havoc", and let slip the dogs of war.

from "Julius Caesar", Act III, Scene I, by William Shakespeare

6.1 Introduction

This chapter reports one of the most intricate parts of this research: namely, the application of parallelism to a large, existing code, written in an old, unstructured language. The code in question was the Numerical Electromagnetics Code — Method of Moments Version 2, normally abbreviated to NEC2. NEC2 uses the MoM to solve radiation and scattering problems involving perfect or very good conductors. Both a thin-wire Electric Field Integral Equation (EFIE) formulation and a patch Magnetic Field Integral Equation (MFIE) formulation are available; the latter requires that the region where patches are used be a smooth, closed structure. It is a very powerful code — in addition to free space problems, it can handle ground planes, and, when used in conjunction with SOMNEC, even lossy grounds.

The difficulties were not of a fundamental, theoretical nature relating to parallelism — these problems have been solved in Chapters 4 and 5. The problems were rather related to software re-engineering, and indicate fundamental problems to be solved in that field.

A brief review of the theory underlying NEC2 is given. Methods to parallelize the matrix generation, which is not as simple as for the demonstration program used in Chapter 5 for the LU method, are discussed. An attempt to use an Occam "harness" to provide a link between various FORTRAN processes is discussed; the poor results obtained with this approach led to the necessity for re-coding NEC2 in Occam. Fundamental problems posed by such a re-write are highlighted. A systematic (but not presently automatic) methodology for untangling "spaghetti"[1] code is presented; its use is illustrated by application to one of the more formidable NEC2 subroutines.

Methods used to validate the code are described — initially for small structures, and then for large structures. With the former, the problem is simply to ensure that the Occam re-code is a valid re-implementation of the code. With the latter, the problem is more fundamental: is the code accurate for electromagnetically very large structures, or does machine precision impose some limit on the size of structure that can be modelled? This is investigated using a symmetrical structure, solved both using, and then without using, symmetry. The effect of single as opposed to double precision is also investigated; it is shown to primarily affect only the convergence rate of the CG algorithm, not the final answers. The biconjugate gradient method is also briefly investigated; it is shown that the method's initial promise for small systems fails on large systems, where the biconjugate gradient method fails to converge. The biconjugate gradient method is also shown to be far more sensitive to precision than the CG method.

6.2 A Review of the Theoretical Basis of NEC2

6.2.1 The Electric Field Integral Equation

The thin-wire part of NEC2 is based on the Electric Field Integral Equation (EFIE). The full derivation of the EFIE is lengthy; the derivation is similar to that used in the Stratton-Chu formulation [Ell81, Section 1.7]. Poggio and Miller present a full derivation in [PM73].

The form of the EFIE used in NEC2[2] follows from an integral representation for the electric field of a volume current density $\overline{J}$,

$$\overline{E}(\overline{r}) = -\frac{j\eta}{4\pi k}\int_V \overline{J}(\overline{r}')\cdot\overline{\overline{G}}(\overline{r},\overline{r}')dV' \tag{6.1}$$

where

$$\overline{\overline{G}}(\overline{r},\overline{r}') = (k^2\overline{\overline{I}} + \nabla\nabla)g(\overline{r},\overline{r}')$$

$$g(\overline{r},\overline{r}') = \frac{e^{-jk|\overline{r}-\overline{r}'|}}{|\overline{r}-\overline{r}'|}$$

[1] "Spaghetti" code is the generic name used in the literature for code written using unstructured languages [LWA91]; the name originates from the appearance of a flow-chart of the code.

[2] This section is based on [BP81a, pp.3-5].

$$k = \omega\sqrt{\mu_0\epsilon_0}$$
$$\eta = \sqrt{\mu_0/\epsilon_0}$$

and the time convention is $e^{j\omega t}$. The prime and unprimed coordinates refer to source and field points respectively. $\overline{E}(\overline{r})$ is the electric field at field point $\overline{r}$, and $\overline{J}(\overline{r}')$ is the volume current at source point $\overline{r}'$. $\overline{\overline{I}}$ is the identity dyad $(\hat{x}\hat{x} + \hat{y}\hat{y} + \hat{z}\hat{z})$. $\overline{\overline{G}}(\overline{r}, \overline{r}')$ is the free space dyadic Green's function. Tai's monograph provides a comprehensive exposition of dyadic Green's functions in electromagnetics [Tai71], and summaries of the topic may be found in [Kon86, Wan91].

If the body is a perfect conductor, the volume integral reduces to a surface integral, and equation (6.1) becomes

$$\overline{E}(\overline{r}) = -\frac{j\eta}{4\pi k}\int_S \overline{J}(\overline{r}') \cdot \overline{\overline{G}}(\overline{r}, \overline{r}')dS' \tag{6.2}$$

with $\overline{J}(\overline{r}')$ the surface current density.

The observation point $\overline{r}$ is restricted to be off the surface s so that $\overline{r} \neq \overline{r}'$. If $\overline{r}$ approaches S as a limit, the integral must be understood as a principal value integral.

Using the boundary condition for the tangential electric field at a conductor — it must be zero — an integral equation for the current induced on S by an incident field $\overline{E}^I$ can be obtained as:

$$\hat{n}(\overline{r}) \times [\overline{E}^S(\overline{r}) + \overline{E}^I(\overline{r})] = 0 \tag{6.3}$$

where $\overline{E}^S(\overline{r})$ is the field of the induced current, given by equation (6.2), and $\hat{n}(\overline{r})$ is the unit normal vector of the surface at $\overline{r}$.

When the surface is a thin wire, several simplifying approximations can be made. These are:

- Transverse currents can be neglected relative to axial currents on the wire.
- The circumferential variation in the axial current can be neglected.
- The current can be represented by a filament on the wire axis.
- The boundary condition on the electric field need be enforced in the axial direction only.

These "thin-wire" approximations are valid as long as the wire radius is much less than a wavelength and also much less than the wire length and segment length; the last requirement has already been discussed in Section 5.13.

The first three assumptions permit the surface current density $\overline{J}_s(\overline{r'})$ on a wire of radius a to be replaced by a filamentary current I where

$$I(s')\hat{s'} = 2\pi a \overline{J}_s(\overline{r'})$$

where s' is the distance parameter along the wire axis at $\overline{r'}$ and $\hat{s'}$ is the unit vector tangent to the wire axis at $\overline{r'}$.

Using this thin-wire approximation and enforcing the boundary condition in the axial direction only reduces the EFIE to the following scalar integral equation:

$$-\hat{s}\cdot\overline{E}^I(\overline{r}) = -\frac{j\eta}{4\pi k}\int_L I(s')(k^2\hat{s}\cdot\hat{s}' - \frac{\partial^2}{\partial s\,\partial s'})g(\overline{r},\overline{r}')ds' \qquad (6.4)$$

6.2.2 Numerical Solution

In equation (6.4) $\overline{E}^I(\overline{r})$ is assumed to be known; for instance it could be a plane wave or the field impressed between the terminals by a source; I is the unknown axially directed current which must be solved for. NEC2 solves for the unknown current using the Method of Moments (MoM), as described in Chapter 1.

6.2.3 The Current Expansion Functions

NEC2 uses a relatively complex set of current basis functions: each expansion function consists of a sine, cosine and constant term. Each basis function is centered on a segment, but also "spills over" onto *all* connected segments. Each expansion function i, centered on segment i, is then forced to satisfy the following conditions

- The current must go to zero at the outer edges of all the connected segments with zero derivative.
- The current must be continuous at the junction of two segments.
- The current must satisfy a condition derived by Wu and King [WK76] related to charge continuity[3].

It is important to note that the conditions on current continuity and charge continuity are satisfied by *each individual* basis function; since the total current on the wire is a linear combination of all the basis functions

[3]From the continuity equation for the time harmonic case $\nabla\cdot\overline{J} = -j\omega\rho$, a condition on charge continuity is equivalent to a condition on the spatial current derivative.

on the segment, if *each* basis function satisfies current and charge continuity, then the total current will also. The second and third points itemized above are thus *sufficient*, but not *necessary* conditions; it is at least theoretically possible to enforce the condition of current and charge continuity on only the total current. This is a subtle point, requiring a rather intense study of [BP81a, p.11–16] to fully appreciate it.

Using these conditions, it can be shown that there is only one independent unknown coefficient per basis function — it is chosen arbitrarily to be that associated with the constant term. The coefficients of the cosine and sine terms are related to that of the constant term by expressions involving only the geometry of the segments (length, radius and connectivity). They are given by equations (43–63) in [BP81a]. The author checked these results; there is a factor of $1/k$ missing consistently in equations (43–51), but this is compensated for by a missing factor k in equations (52–53), so the results may be used as given. (The same error will also be found in (54–57) and (58–61).) Note also that Q in NEC2 is not the Wu-King $q(z)$; it is rather the constant in Wu and King [WK76, equation (25)].

6.3 The Occam Harness

It is a generally accepted fact for large, numerically intensive programs that a relatively small part of the total code is responsible for a disproportionately large amount of the run time. It is often referred to as the 90-10 or 80-20 rule, where the first number refers to the percentage of run-time consumed by the time-intensive code and the second to the percentage of the total code comprising this time-intensive code. With NEC2, the most time-consuming part is the LU factorization of around 100 lines. The total code is several thousand lines. It would thus be a very satisfactory solution if it were possible to leave the majority of the code in the original language (FORTRAN in the case of NEC2) and develop parallel code for only the relatively small, time-intensive code. The Occam Toolset allows one to do this: a module written in an "alien"[4] language can be called by an Occam code. Communication between the Occam "harness" and the FORTRAN code can then take place over channels. For a parallel program, Occam worker processes with embedded FORTRAN processes would be distributed over the processing array, with the inter-processor communication handled by the Occam harness. More details on the harness concept may be found in [AHZ90].

The Occam/FORTRAN interface code supplied was tested, and the author was able to get simple test cases working — but not reliably. The problems have already been described in Section 3.4.2, and were due to the

[4]On the transputer, any language other than Occam.

unreliability of the FORTRAN compiler and the total absence of debugging support in the FORTRAN environment. The author initiated development of an Occam harness to parallelize the matrix generation. This work was extended by Malan, but the parallel code was very inefficient and did not work reliably [Mal90].

It must be emphasized that the fault here is the inadequate FORTRAN compiler; the basic philosophy is sound and sensible, but was rendered unworkable by the very poor FORTRAN compiler. Other researchers have reported some success along these lines, for example Schuilenburg with a large GTD code [Sch90] and Nitch with NEC2 [NF90b]. Nitch parallelized NEC2 using the harness methodology outlined above and obtained impressive speed-ups, but comments that the effort of manipulating the FORTRAN source code led him to start re-coding NEC2 in C^{++} for his future work [NF90a].

6.4 The Occam Re-write: Philosophy and Methodology

NEC2 is an exceptionally well documented code, and the three part documentation (theory description, code description and user's guide [BP81a, BP81b, BP81c]) could well be prescribed for courses in software documentation. This made possible the re-write; without this documentation the project would not have been feasible.

NEC2 is written in one of earlier versions of FORTRAN, probably FORTRAN 66, although the version distributed with the initial NEEDS[5] release also contained a number of VAX-specific FORTRAN extensions. FORTRAN 66 was released before the advent of structured programming, which dates back to the around 1970 and the work of Dijkstra and Wirth — the latter being responsible for Pascal. As such, the authors of NEC2 had to use a number of unstructured constructs which made the re-write a very far from trivial issue. It must be commented again that, considering the languages available when NEC2 was developed, the development and documentation was done with admirable care. The problems highlighted here were inherently unavoidable in FORTRAN 66.

There are two fundamental problems with FORTRAN from the viewpoint of structured coding: the infamous `go-to` and related constructs, and the `common block`. The `go-to` construct greatly complicates the goal of re-using software [LWA91], and old FORTRAN codes contain numerous examples of the "spaghetti" code that can be produced using several suitably interwoven

[5]Distributed by the Applied Computational Electromagnetics Society.

`go-to`'s. Dijkstra identified the `go-to` as a highly undesirable construct, and using structured programming its use can be entirely avoided, using constructs such as those defined in Section 3.7. The `common block` is a specifically FORTRAN construct, and is a special case of the almost always undesirable global variable. It would appear very useful to be able to define an area in memory common to several routines, and hence create a global variable. The problem is that the use of the `common block` makes it very difficult to discover exactly what side effects a sub-routine will have, since not only variables passed as arguments will be affected by a sub-routine containing `common blocks`. FORTRAN 77 is a tremendous improvement from the viewpoint of structured coding, but as already mentioned, NEC2 is essentially written in FORTRAN 66.

Occam 2, the target language, is a modern, structured, strictly typed language and does not support the `go to` statement at all. Common blocks could be simulated using global variables which are in scope for all procedures, but this is not the modern, structured, approach, which calls for different modules, isolated from one another completely except for the parameters passed (either by value or reference) in the argument list. Extensive use was made of *separate compilation* units. A separately compiled unit[6] is a self-contained section of code, communicating with other units only via the argument list (and in Occam of course, via channels as well.) All subroutines and functions in the FORTRAN code were converted to separately compiled units in the Occam re-write.

The strict typing[7] of Occam permits an Occam compiler to perform extensive checking at compile time, which detects a large number of faults before the program even has an opportunity to run incorrectly with them. An example will suffice to motivate this: a very difficult problem to detect with FORTRAN is the incorrect agreement of parameters in a subroutine call (i.e. the number and/or type of parameters in the defining SUBROUTINE statement does not agree with the number and/or type in the calling statement). The fault is particularly insidious if it is the precision that is incorrect, and such faults can be very difficult to detect. Occam checks agreement of both type and number of parameters across the calling and defining statements. Another insidious FORTRAN bug is the accidental mis-spelling of a variable; the compiler's default typing automatically assumes that this is a new variable. Most dialects of FORTRAN 77 now include an IMPLICIT NONE statement that can be used to detect the accidental mis-spelling faults, but

[6]A separately compiled unit is the same as a TurboPascal unit.

[7]Strict typing means that all variables and constants must have their type explicitly declared before they are referenced. FORTRAN is not, by default, a strictly typed language; the FORTRAN default is that all variables beginning with the letters I–N are integers, and the rest real.

a facility to check parameters lists across calls in FORTRAN is not generally provided.

The one undisputed advantage that FORTRAN does have is its `complex` data type; the author had to split the real and imaginary parts and compute each separately.

The most serious problems during the re-write were encountered with several rather tightly coded routines that used a large number of `go-to`'s. At first sight, the functioning of the routines was so disguised by these `go-to`'s as to be utterly unintelligible. One of the most difficult routines to convert was SBF, which contains no less that 22 `go-to`'s and computed `go-to`'s in about 130 lines of FORTRAN. Another major problem was the use of `common blocks` to communicate data to subroutines, as discussed earlier.

The fundamental problems can be summarized as follows:

- Ensuring that the necessary data is passed to the procedures.

- Recovering the underlying logic of the procedures.

The former is the easier problem. To solve it, extensive use was made of the cross-referenced listing that most good compilers can generate — the author used a VAX system for this. These listings will typically include a list of variables, a warning regarding use in `common blocks` and `equivalencing`[8], lines accessed and how accessed (i.e. read, write). This data was used to determine if a particular `common block` parameter was used in a routine, since by no means all such parameters were actually required in all the routines that contained a `common block` referencing a particular parameter.

The latter problem was rather more difficult. When re-engineering a code, the aim is as far as possible to avoid having to understand the *precise* operation of the original code, to capitalize on the original intellectual investment in the code. This was not always possible with the procedures with substantial "spaghetti" code components. The procedure adopted for these cases was the following: the original logic was recovered using a flow-chart, and then recoded using structured constructs in pseudo-code: primarily the `if-then-else` and `while` constructs. Once functionally equivalent pseudo-code had been derived, the pseudo-code was simplified and then implemented in Occam.

As an example of this approach, the re-coding of subroutine TRIO is considered. TRIO evaluates all the parts of current expansion functions on a single segment due to each of the segments connected to the given segment. It is only 41 lines of FORTRAN, but contains no less than 11

[8]Another powerful and potentially dangerous FORTRAN command, originally provided to conserve scarce memory but frequently the source of very subtle bugs.

`go-to`'s, `if-go-to` combinations and `if (number) i,j,k` constructs (another unstructured construct), indicating how even a short code stub can pose major problems. The original FORTRAN is shown in Figure 6.1.

The flow chart for the program is drawn up. This is shown in Figure 6.2. Now the flow chart is examined for *fundamental* operation. The first action is to decouple parts of the flow chart. Label 8 corresponds to some action followed by termination, so it can be seen that first part of the problem has been reduced to the structure (one hesitates to call it a loop) between the first test on `jcox (<,=,> 0)` and the second such test on either the main path or the branch on the right hand side of the flow chart labelled Branch A. Hence, this is a "loop" that must execute until some termination criteria has been met. This leads one to consider a `while` structure. Eventually, the pseudo-code given in Figure 6.3 results.

To read the flow chart and pseudo-code, it is necessary to know the following about how NEC2 keeps track of connectivity data. The `jcox` test determines whether a segments has any connections: Array `icon1` contains connectivity data for end 1 of segment i.

- if 0, there are no connected segments
- if k , it is connected to end 2 of segment k
- if -k , it is connected to end 1 of segment k

Similar definitions hold for `icon2`, containing connectivity data for end 2 of segment i, except that the role of k and -k are interchanged. An example: if segments are connected 1-2-3-1, then the arrays icon1 and icon2 will be the following: `icon1` = [312] and `icon2` = [231]. With multiple wire connections, segment connectivity data is stored cyclically, only the next lowest segment's number being stored, and hence a "daisy chain" must be traversed to determine whether a segment is possibly connected to several others.

Further improvements are possible. Inspection of the pseudo-code shown in Figure 6.3 reveals that the code is actually executing the same instructions for each end; it would be rather clearer if this was shown explicitly rather than incorporated in the first `if` construct inside the `while` and also in the `jcox <> j` code stub . The code given in Figure 6.4 shows this; note how the readability of the code is increasing. The actual Occam implementation of TRIO is essentially the code given in Figure 6.3 but the technique of simplifying the pseudo-code further once pseudo-code functionally equivalent to the flow-chart had been obtained was applied to several other routines, for instance routines SBF and TBF, which contain similar structures. Note that for simplicity, global variables are used *within* a procedure such as TRIO; the total procedure length is small enough to avoid problems. It is the concept of global variables for the whole code or major parts thereof that is dangerous software engineering practice.

The method had to rely on a certain amount of trial and error; it is extraordinarily difficult to recover the *underlying* logic from "spaghetti" code. The methods outlined for TRIO did not have to be applied to all the NEC2 procedures; some could be converted relatively easily without a full flow-chart and pseudo-code analysis, or only using this analysis in places. However, the following procedures all required a full flow-chart and pseudo-code approach similar to that shown for TRIO: TBF, SBF, INTX and ISEGNO.

At the time of writing, only a limited number of the capabilities provided by the full NEC2 were available, but the basic thin-wire formulation had been completely implemented. The code is around 5000 lines in length, and consists of around 20 major procedures corresponding to the original FORTRAN subroutines and functions. Following good software engineering practice, a code manual and user manual similar to the original NEC2 documentation [BP81b, BP81c] have been produced [Dav91b, Dav91a].

PARNEC has been written so that changing the maximum array dimensions or the number of worker transputers is trivial; these parameters are declared in two library files. (This should be compared to trying to change the dimensions in the original FORTRAN code — a painstaking and far from trivial job as anyone who has done it, such as the author, will vouch.)

The re-write required a detailed knowledge of FORTRAN, and also — in places — of the underlying application. Developing a program able to convert the FORTRAN Figure 6.1 into the readable, functional and structured pseudo-code of Figure 6.4 should be a viewed by computer scientists as a great challenge[9], and would be of great utility for engineers and scientists wishing to port old codes to new languages, to efficiently exploit new architectures, or merely to improve the maintainability of the code.

6.5 Parallelizing the Matrix Fill

It has been mentioned that a test MoM code using a simple straight thin wire was implemented, and that parallelizing the matrix fill was a trivial theoretical problem. This demonstration code used a sinusoidal basis function, collocation, and an analytical solution for the field radiated by an sinusoidal current [ST81, Chapter 7]. Matrix element a_{ij} is the field of basis function j at test point i, and is given by the sum of three terms involving the distance

[9]A rather easier problem would be the development of a code that will automatically convert a program written in Pascal to one in Occam, since these languages share very similar structures and are both strictly typed . This is also a rather easier exercise to perform manually.

```
      SUBROUTINE TRIO (J)
C     COMPUTE THE COMPONENTS OF ALL BASIS FUNCTIONS ON SEGMENT J
      COMMON /DATA/ LD,N1,N2,N,NP,M1,M2,M,MP,X(300),Y(300),Z(300),
     1SI(300),BI(300),ALP(300),BET(300),ICON1(300),ICON2(300),
     2ITAG(300),ICONX(300),WLAM,IPSYM
      COMMON /SEGJ/ AX(30),BX(30),CX(30),JCO(30),JSNO,ISCON(50),NSCON,
     1IPCON(10),NPCON
      DATA JMAX/30/
      JSNO=0
      JCOX=ICON1(J)
      IF (JCOX.GT.10000) GO TO 7
      JEND=-1
      IEND=-1
      IF (JCOX) 1,7,2
1     JCOX=-JCOX
      GO TO 3
2     JEND=-JEND
3     IF (JCOX.EQ.J) GO TO 6
      JSNO=JSNO+1
      IF (JSNO.GE.JMAX) GO TO 9
      CALL SBF (JCOX,J,AX(JSNO),BX(JSNO),CX(JSNO))
      JCO(JSNO)=JCOX
      IF (JEND.EQ.1) GO TO 4
      JCOX=ICON1(JCOX)
      GO TO 5
4     JCOX=ICON2(JCOX)
5     IF (JCOX) 1,9,2
6     IF (IEND.EQ.1) GO TO 8
7     JCOX=ICON2(J)
      IF (JCOX.GT.10000) GO TO 8
      JEND=1
      IEND=1
      IF (JCOX) 1,8,2
8     JSNO=JSNO+1
      CALL SBF (J,J,AX(JSNO),BX(JSNO),CX(JSNO))
      JCO(JSNO)=J
      RETURN
9     WRITE(6,10)  J
      STOP
C
10    FORMAT (44H TRIO - SEGMENT CONNECTION ERROR FOR SEGMENT,I5)
      END
```

Figure 6.1: FORTRAN source for subroutine TRIO.

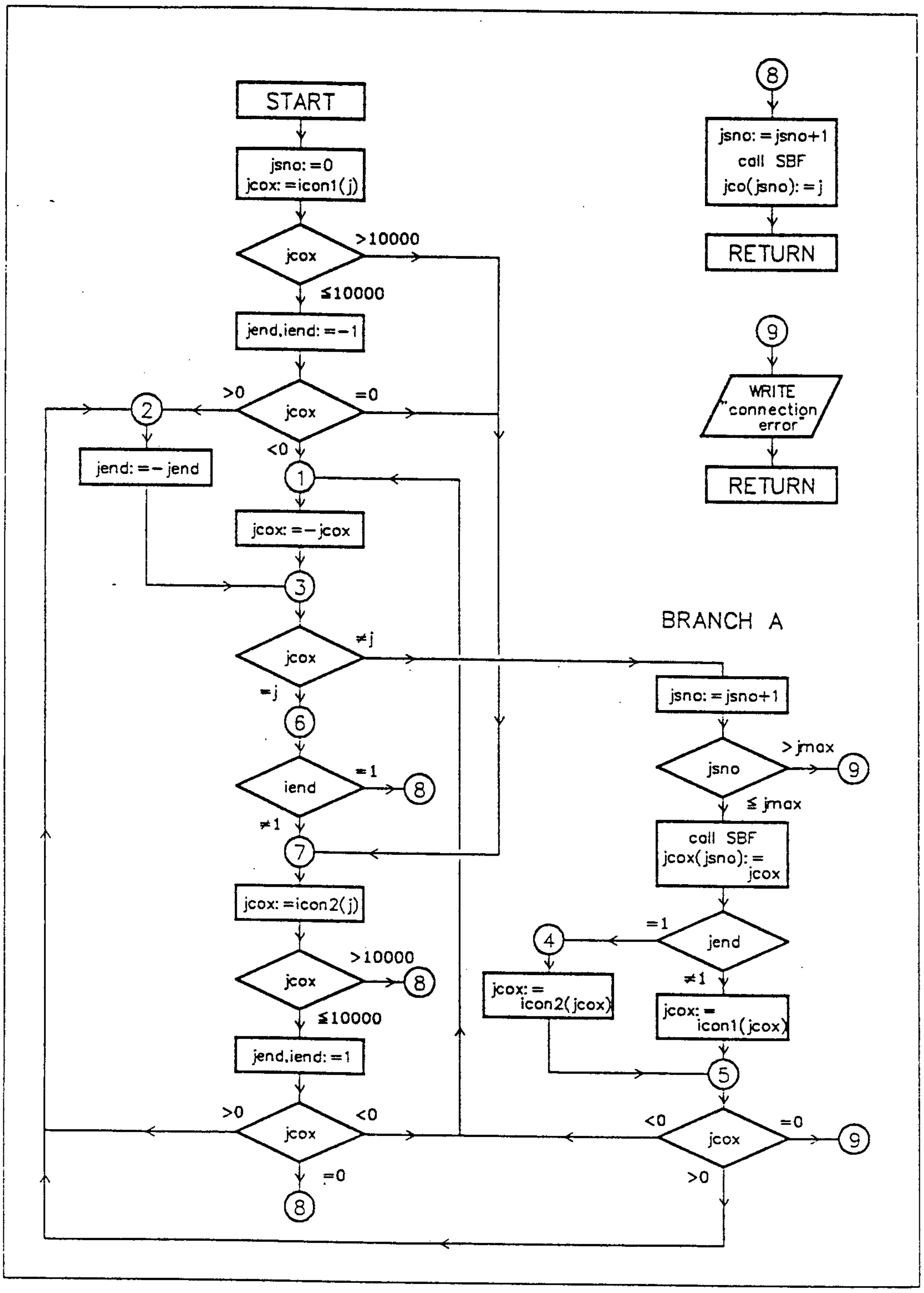


Figure 6.2: Flow chart for subroutine TRIO.

```
process(trio)
begin{trio}
  variable declarations
  {initialize}
  jsno := 0
  jcox := icon1[j]
  patch.error, seg.conn.error, finished := FALSE
  iend, jend := -1
  while (NOT patch.error) AND (NOT seg.conn.error) AND (NOT finished)
    begin
      if{test for connected end}
        jcox = 0 then
          begin{jcox = 0}
            jcox := icon2[j]
            {patch test; not shown}
            iend, jend :=1
            if {It is necessary to re-check jcox since it has now changed}
              jcox < 0 then jcox := -jcox
              jcox > 0 then jend := -jend
              jcox = 0 then finished := TRUE
            end{if}
          end{jcox=0}
        jcox < 0 then jcox := -jcox
        jcox > 0 then jend := -jend
      if
        NOT finished then
          begin{NOT finished}
          if
            jcox = j then
              if
                iend = 1 then finished := TRUE
                iend = -1 then
                  begin{iend = -1}
                    jcox := icon2[j]
                    {patch test; not shown}
                    jend, iend := 1
                    if
                      jcox = 0 then finished := TRUE
                      jcox <> 0 then SKIP -- and repeat the while
                  end{iend = -1}
            jcox <> j then
              begin{jcox <> j}
                jsno := jsno + 1
                if
                  jsno > jmax then seg.conn.error := TRUE
                  jsno <= jmax then
                    begin{jsno <= jmax}
                      call process sbf
                      jco[jsno] := jcox
                      if
                        jend = 1 then jcox := icon2[jcox]
                        jend = -1 then jcox := icon1[jcox]
                      if
                        jcox = 0 then seg.conn.error := TRUE
                        jcox <> 0 then SKIP
              end{jcox <> j}
          end{NOT finished}
        finished then SKIP {occurs if second end is unconnected}
    end{while}
    end_code stub{corresponding to branch 8}
  end{trio}
```

Figure 6.3: Pseudo-code for subroutine TRIO; first pass.

```
process(trio)
begin{trio}
  variable declarations
  procedure end{declare procedure}
    begin{procedure end}
      while (NOT patch.error) AND (NOT seg.conn.error) AND (NOT finished)
        begin
          if{test for connected end}
            jcox = 0 then finished := TRUE
            jcox < 0 then jcox := -jcox
            jcox > 0 then jend := -jend
          if
            NOT finished then
              begin{NOT finished}
                if
                  jcox = j then finished := TRUE
                  jcox <> j then
                    begin{jcox <> j}
                      jsno := jsno + 1
                      if
                        jsno > jmax then seg.conn.error := TRUE
                        jsno <= jmax then
                          begin{jsno <= jmax}
                            call process sbf
                            jco[jsno] := jcox
                            if
                              jend = 1 then jcox := icon2[jcox]
                              jend = (-1) then jcox := icon1[jcox]
                            if
                              jcox = 0 then   seg.conn.error := TRUE
                              jcox <> 0 then SKIP
                          end{jsno <= jmax}
                    end{jcox <> j}
              end{NOT finished}
            finished then SKIP
          end{if}
        end
      end{while}
  end{procedure end}

  {actual start of main code}
  jsno := 0

  {initialize for end 1}
  jcox := icon1[j]
  patch.error, seg.conn.error, finished := FALSE
  iend, jend := -1
  call procedure end {for end 1}

  {initialize for end 2}
  jcox := icon2[j]
  patch.error, seg.conn.error, finished := FALSE
  iend, jend := 1
  call procedure end {for end 2}

  end_code stub{corresponding to branch 8}
end{trio}
```

Figure 6.4: Pseudo-code for subroutine TRIO; improved structure.

from the two ends and the middle of basis function j to the test point. It is of absolutely no concern in what sequence the calculations are performed, so the i, j index set for each processor can be chosen to suit whatever *clustering* is desired; row block clustering for the CG algorithm and double interleaved clustering for the LU algorithm.

However, matters are not so simple with NEC2. The reason is that the explicit enforcement of current and charge continuity results in segments with possibly very different numbers all contributing to the current on segment j. NEC2 adds the necessary contributions to the relevant matrix entry as the segment index increments, which is of course trivial on a serial processor with the matrix in core[10]. Thus, one cannot decompose the matrix fill of NEC2 by segment, since it is very possible that a segment on one processor would be connected electrically to a segment on another processor, and this will require complex coding to handle.

A very simple way of decomposing the problem is to rather do it by field point. This is the way that NEC2's out-of-core storage works: that problem is of course very similar to the parallel decomposition problem, since in the former case one does not want to continually modify elements stored on disk and in the latter case, the same holds for elements stored on different processors. It is also (serendipitously) the row-block decomposition chosen for the CG algorithm — although modifying the CG algorithm to a column block decomposition is an almost trivial operation, one would simply have to interchange the order in which the parallel matrix-vector multiplication paradigms were used.

For use with the parallel LU solver, a row-block clustering to double interleaved clustering mapping scheme will be required. This will require communication of $O(M^2)$ and will thus have minimal effect on the overall code efficiency.

6.6 Software Validation

With any large and complex piece of software, such as the re-engineered PARNEC, the question of validating the code is crucial. One must be particularly careful regarding MoM formulations, since some of the "observables" such as the radiated far fields, gain and radar cross section be may stationary functionals of the surface current. Hence the fields are degraded by the second variation of the current, to use the terminology of the variational calculus. An example — a 10% error in current will reduce to a 1% error

[10]In finite element parlance, this is an *assembly by elements*, as opposed to an *assembly by nodes*; see [NdV78, p.48]

in the radiated far fields [Har61, p.335]. The variational aspects of the moment method have been discussed in detail by Richmond [Ric91]. Thus a validation procedure using the far fields is a fundamentally unsound check.

A far better check is on the currents themselves or the input impedance[11]. The latter is a particularly suitable one-parameter check on the code, and was used in the later stages of the code development, when substantial parts of the code had already been validated. However, for the initial debugging stages, the matrix elements had to be inspected directly.

A representative test set, involving dipoles (with very simple connectivity) and boxes (to represent a far more connected wire structure) was built up, using initially a fairly small number of unknowns. The absence of a (working) *interactive* Occam debugger has already been noted in Chapter 3. The solution adopted was the insertion of write statements at various critical parts of the code (such as preceding and following a procedure call). To permit more selective debugging, these statements were coded so that they could be selected using a library **debug** file. This avoided having to edit the source code if one wanted to select or de-select certain outputs. These intermediate results were then compared to results in the corresponding place in the original FORTRAN code, using the excellent interactive VMS debugger available on the VAX cluster.

During validation of the code, it was noted that in many cases there were minor differences in the interaction matrix elements. Typically these were in the fourth or fifth significant figure. This initially appeared strange, since although the codes were written in two different languages, they were executing precisely the same logic. The reason is that the Occam compiler on the transputer uses standard IEEE arithmetic, whereas the VAX uses its own (and different) standard. This primarily affects the representation of floating point numbers and the precision of operation. Thus, even logically identical code will not produce the same results if extensive floating point operations are involved. This shows up clearly in INTX, the Romberg integration routine used in NEC2.

It is rather tempting to glibly attribute differences in the results to the afore-mentioned differences in floating point precision, whereas the differences may actually be due to an underlying logical error. This was comprehensively investigated by converting to double precision (64 bits per real number) operation. This also required a double precision version of NEC2, which due to the already noted unstructured character of FORTRAN 66 is also a non-trivial job. The methodology adopted to produce a reliable double precision version is described in Appendix C. The finished double precision

[11] Richmond [Ric91] shows that the impedance can also be stationary, if a Galerkin formulation is used. NEC2 uses collocation.

Mean	Standard Deviation
2.579×10^{-7}	4.026×10^{-7}

Table 6.1: Mean and standard deviation of the normalized differences between the interaction matrices generated by PARNEC and NEC2.

code was checked by comparing results computed using the double and single precision FORTRAN codes for example 1 in the NEC2 user manual [BP81c, p.98] and against a box-like structure with about 130 segments; only very minor differences were noted, in line with the difference in precision. The conversion took an entire man-day. By comparison, the Occam re-code took half-an-hour, thanks to the strict typing in Occam. Using the TDS editor's search-and-replace facility, all occurrences of `REAL32` were changed to `REAL64` and the single precision library calls were replaced by their double precision equivalents. Unlike FORTRAN, Occam does not have generic function names. If the incorrect precision function is used, the compiler flags an error, so the conversion is very safe.

With the double precision versions available, the validation could continue. Differences in the matrix elements in the fourth and fifth significant places were still noted. Then the Romberg error criterion (`RX` in INTX) was reduced from 10^{-4} to 10^{-8}, and the number of significant digits of agreements immediately improved dramatically. A Romberg error criterion of 10^{-4} should indeed only generate about 4 digits of precision.

Once all these steps outlined had been taken, the validation showed agreement to working precision for small numbers of unknowns. However, this may not adequately test all the possible paths the code may take; for instance NEC2 uses an approximate method to perform the numerical integration for segments separated by more than that specified by parameter `RKH`. But with large structures, examining the interaction matrices manually is a potentially error-prone operation. Hence the process was automated, and a Pascal program `zmat` was written to read in the NEC2 and PARNEC interaction matrices and compare them. The program computes the standard deviation and mean of the differences on a element by element basis, and also finds the largest absolute and relative differences. It was then applied to several test structures; a typical result for a 50 segment problem, a cube with a monopole, with the Romberg error criterion set to 10^{-8}, is shown in Table 6.1. The mean error is close to the Romberg error criterion.

It is interesting to apply the software metric concept to PARNEC. One useful one is due to Halstead [Sto90b]; it gives the expected number of errors in a program after compilation (i.e. the program syntax is correct but the logic has not been tested) as

$$\frac{(N_1 + N_2)log_2(n_1 + n_2)}{3000} \tag{6.5}$$

with

N_1 : the total number of operators;

N_2 : the total number of operands;

n_1 : the number of *distinct* operators;

n_2 : the number of *distinct* operands.

PARNEC Version 1.1 ran to around 5000 lines of code; a substantial part of this was in-code documentation, so the executable instructions probably comprised about 3000 lines. The average number of operators per line is probably about 2, giving $N_1 \approx 6000$. The total number of operands is probably fairly similar, so $N_2 \approx 6000$. The number of distinct operators is around 20 and the number of distinct operands around 50. Thus Halstead's metric gives an expected number of errors of 25. The actual number of errors detected during validation was about 6. Each procedure was also checked by hand and compared to the original FORTRAN subroutine *before* validation, which explains the smaller number of errors.

6.7 Some Results for a Large Problem

Once the validation of PARNEC had been satisfactorily completed, attention could be turned to the problem of whether the code was able to solve large problems accurately, or whether machine precision would limit the useful range of the MoM, as has been suggested previously [Han81, p.384]. Care is needed in designing an experiment to investigate this. Obtaining really good agreement between measured and numerically predicted data is a problem in computational electromagnetics, especially on such a sensitive parameter as input impedance. The reasons are legion; perhaps two of the more common are inaccurate numerical models and inaccurate measurements. The former could be caused by approximations in the underlying theory, or by inadequate representation of the structure (i.e. a piecewise linear approximation of a curved surface). The applied E-field model used in NEC2 and PARNEC falls into the first class — it is known to only approximate the actual impedance. Hence numerically modelling an electromagnetically large structure with PARNEC and comparing the results to measurements will not be a very good test of the inherent accuracy of the MoM, given the problems known to exist with the source model.

A very fast computer with a very high working precision and very large memory would be the ideal benchmark — but it is of course the absence of such a machine that led to the research reported in this thesis. But there is a way in which electromagnetically large structures can be simulated *without* generating a correspondingly large system of equations, and that is by exploiting *symmetry*[12].

The problem investigated was a cone-cylinder [lR89], shown in Figure 6.5. The full structure consists of a cylinder of length 397 mm and diameter 53 mm, capped by a cone of length 107mm at one end and a closed off at the other. 4 monopoles, each 1 mm in diameter and 25 mm long are mounted on the cylinder 28 mm from the cone-cylinder interface. The monopoles make an angle of 30^o with the cylinder. The monopoles are mounted symmetrically, i.e. 90^o apart. The frequency of operation is 3 GHz. The monopoles are exactly a quarter of a wavelength long at this frequency.

The structure is ideal for the required validation purpose. It is electromagnetically long — about 4 wavelengths. Using a 10 mm nominal segment length requires about 1500 segments to model the structure. The wire radius for the wire-grid used to simulate the surface satisfied the "same surface area" rule (or more accurately the "twice surface area rule") [Lud87]. The structure exhibits four-way symmetry.

The numerical experiment was the following: the double precision NEC2 version was applied to the problem using the symmetry, which reduced the number of unknowns to just under 400. The full problem was then solved using the double-precision version of PARNEC[13] and the results are given in Table 6.2. The Romberg error criterion was set to 10^{-8}, and the normalized residual error tolerance for the CG solver was set to 10^{-4} (0.01%). The results agree to four significant digits, the expected accuracy of the CG solver. Results are also shown for the same experiment repeated with the single precision version of PARNEC. The other parameters were as for the double precision run. Agreement to 3 decimal places is noted; this is not quite the precision set by the normalized CG error criterion but is due to the matrix fill procedure, involving subtracting very similar numbers. *These results are very significant; they show that a problem with about 1500 unknowns can*

[12]The author thought of this independently, but subsequent discussions revealed that this method for checking the accuracy of matrix generators and solvers for large MoM systems has been used before [Mil90b].

[13]The results shown in this chapter were generally computed using 14 or 30 worker transputers; the number of transputers — 1 for the serial code, and 2, 6, 14 or 30 worker transputers for the parallel code — had a very small effect on the impedance values, but always less than the normalized CG error tolerance.

PARNEC Single Precision		PARNEC Double Precision		NEC2 Double Precision	
(Symmetry not used)				(Symmetry used)	
Re	Im	Re	Im	Re	Im
23.289	-12.410	23.301	-12.435	23.300	-12.432

Table 6.2: Input impedances for the 1516 segment model (Ω)

be solved to a very high degree of accuracy - certainly far more than the underlying accuracy of the modelling process. For this size of problem, these results also show that the effect of precision on accuracy is insignificant when using a CG solver, unless more than three significant digits of accuracy are required.

A number of smaller versions of the cone-cylinder were also created. All used the same discretization (a nominal segment length of 10 mm); the difference was the amount of the structure modelled. It is these models that were used to generate the timing data for the parallel CG algorithm shown in Chapter 4. A summary of these models is given in Table 6.3. The length of cylinder is the length from the cone-cylinder junction. The input impedances computed are also tabulated; these were computed using the double precision PARNEC, Romberg error criterion 10^{-8}, and normalized residual error tolerance 10^{-2} (except for the 1996 problem that used single precision). The impedances all checked to within the precision expected by the CG solver with the results for the double precision NEC2, and in many cases were rather better than the approximately 1% accuracy expected from the 10^{-2} normalized residual tolerance specified. The stringent normalized residual tolerance used for Table 6.2 was relaxed for these runs since the 10^{-4} criterion resulted in very long execution times. The limitations of the applied E-field model should also be borne in mind.

These results do *not* show convergence as a function of improving discretization of the same structure, but rather the convergence as a function of the same discretization of an improving model of the structure[14]. The effect of the lower part of the cylinder on input impedance is clearly minimal. Note that the radiation patterns, however, are strongly affected [lR89].

The efficiencies of the parallel matrix generation algorithm were also investigated. No theoretical model was derived for this, since the operation is

[14]The last model, with 1996 segments, is actually of a rather longer cylinder; it was included to check the operation of the code for large problems using single precision. The 1516 segment model is the most accurate model of the actual geometry.

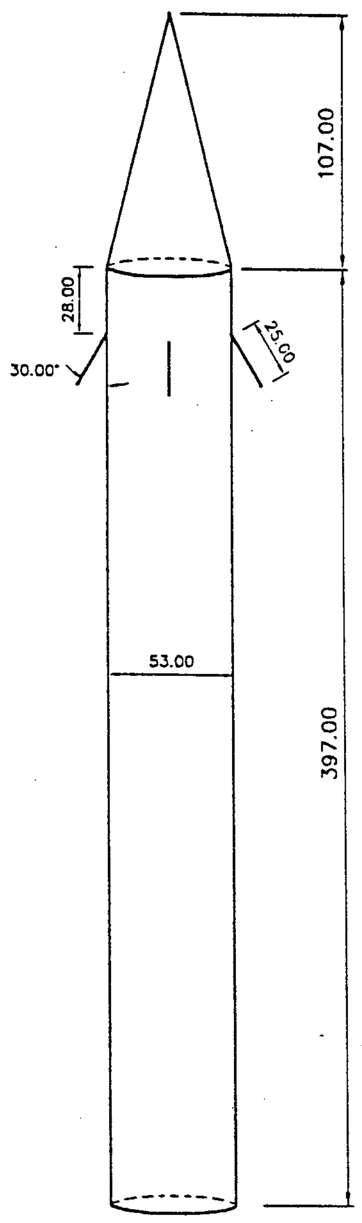


Figure 6.5: Cone-cylinder.

Number of segments	Description	PARNEC (Symmetry not used) Input impedance (Ω)		NEC2 (Symmetry used) Input impedance (Ω	
		Re	Im	Re	Im
124	30 mm long cylinder only; open at both ends	105.1	-40.75	105.1	-40.78
188	40 mm long cylinder only; open at both ends	43.46	-45.30	43.47	-45.30
316	80 mm long cylinder only; open at both ends	24.59	-1.793	24.58	-1.700
428	60 mm long cylinder; cone; cylinder capped.	16.74	-14.79	16.78	-14.75
876	200 mm long cylinder; cone; cylinder capped.	23.32	-12.84	23.34	-12.78
1196	300 mm long cylinder; cone; cylinder capped.	23.25	-12.58	23.34	-12.52
1516	400 mm long cylinder; cone; cylinder capped.	23.21	-12.50	23.30	-12.43
1996	550 mm long cylinder; cone; cylinder capped.	23.19	-12.42	23.28	-12.38

Table 6.3: Different cone-cylinder models

rather simple — a one-off broadcast of the relevant geometrical data to all the processors — and it is difficult to develop a timing model of the matrix fill. The reason for this is the searches required to set up the connection data; the particular numbering scheme used has a very strong effect on this. Results are shown in Figure 6.6. Results are not shown for problems larger that 428 segments since serial timing data was not available, and efforts to extrapolate the available serial data for the larger problems led to efficiencies exceeding 100%; the reason for this failure is the already noted connectivity searches. It is also irrelevant for large problems; Table 6.4 shows the ratio of time required for the matrix fill to the matrix solve for a normalized residual error of 10^{-2}, and a Romberg error criterion of 10^{-4}. All data except for the last entry are for double precision; the 1996 segment data used single precision.

Number of segments	t_{solve}/t_{fill}
50	1.0
124	2.2
188	2.7
316	2.4
428	7.2
876	9.1
1196	10.4
1516	11.9
1996	21.1

Table 6.4: Ratio of the matrix fill to solve times: 30 workers

Results for the efficiencies of the CG solver have already been given in Chapter 4. Efficiencies for the whole code are dominated fairly rapidly by the CG solver, as shown in Table 6.4, so the efficiency of the whole code is not plotted.

One final point that was also investigated was the effect of precision on rate of convergence. This data is presented in Table 6.5. The single precision data is for a Romberg error criterion of 10^{-4}, the double precision for 10^{-8}. The difference in the Romberg error criterion alone produces a maximum difference of one or two iterations (from data not shown), so the differences in Table 6.5 are due to the different precision. The N/A stands for "not available"; the problem required too much memory for the available resources. It is clear that the time advantage gained using single precision is negated by the slower convergence. The only reason to use single precision is the memory saving; double precision requires twice as much memory as single precision for a given number of segments.

Number of segments	Number of iterations	
	Single Precision	Double Precision
50	23	14
124	82	75
188	155	134
316	152	131
428	519	372
876	519	405
1196	521	409
1516	529	414
1996	543	N/A

Table 6.5: Number of iterations required to obtain convergence; normalized residual of 10^{-2} (1%) specified.

The very slow increase in number of iterations required as the number of segments is increased beyond 428 is very noticeable. Almost doubling the number of segments from 876 to 1516 requires only an additional four iterations. It was also noticed in the earlier work on the body of revolution problem, discussed in Chapter 2. It that case, the problem geometry was fixed, but the number of segments was increased [DM88]. Peterson *et.al.* have recently provided an explanation of this [PSM88]; the CG method is minimizing a norm involving only those eigenvalues corresponding to eigenvectors needed to represent the initial residual. The 428 segment problem is a very reasonable approximation of the whole problem; clearly the eigenvalues associated with this problem are also a good representation of those for the whole problem.

Figures 6.7 and 6.8 show the normalized residual error squared as a function of iteration number. It is notable how the method hits stagnant "plateaus" requiring many iterations to break out of. Then the error drops quite rapidly again, until another plateau is reached. This behavior has been noted by other researchers but the author is not aware of an explanation at present.

One final point that may now be commented on: is it better to use a CG or a LU solver? The answer is clearly that it depends on the problem. Since the efficiencies of both methods are comparable, the serial break-even point can be used, namely where the number of iterations is 1/6 of the matrix dimension. Even in the largest case investigated, this fraction was closer to 1/4, and was even larger for smaller problems. This was using 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. 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.* [SPM89] 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 [KH90] shows promising results for a multiple right-hand side CG formulation.

6.8 The Biconjugate Gradient Method

The closing comments in the previous section lead one to the following question: can one not improve the convergence rate of the CG method? Several possibilities have been touted for this: one that has been very successful in Finite Element analysis is pre-conditioning [Sun88]. Another is the biconjugate gradient method, details of which are available in [Sar87a] and [SPM90]. Details on both pre-conditioning and the biconjugate gradient method may also be found in [DDSvdV91, chapter 7][15].

The iterative part of the biconjugate gradient algorithm is given as follows:

[15]The parallel implementations of these discussed in this reference are for vector and shared memory computers and sparse linear systems, not local memory computers and full matrices as considered in this thesis.

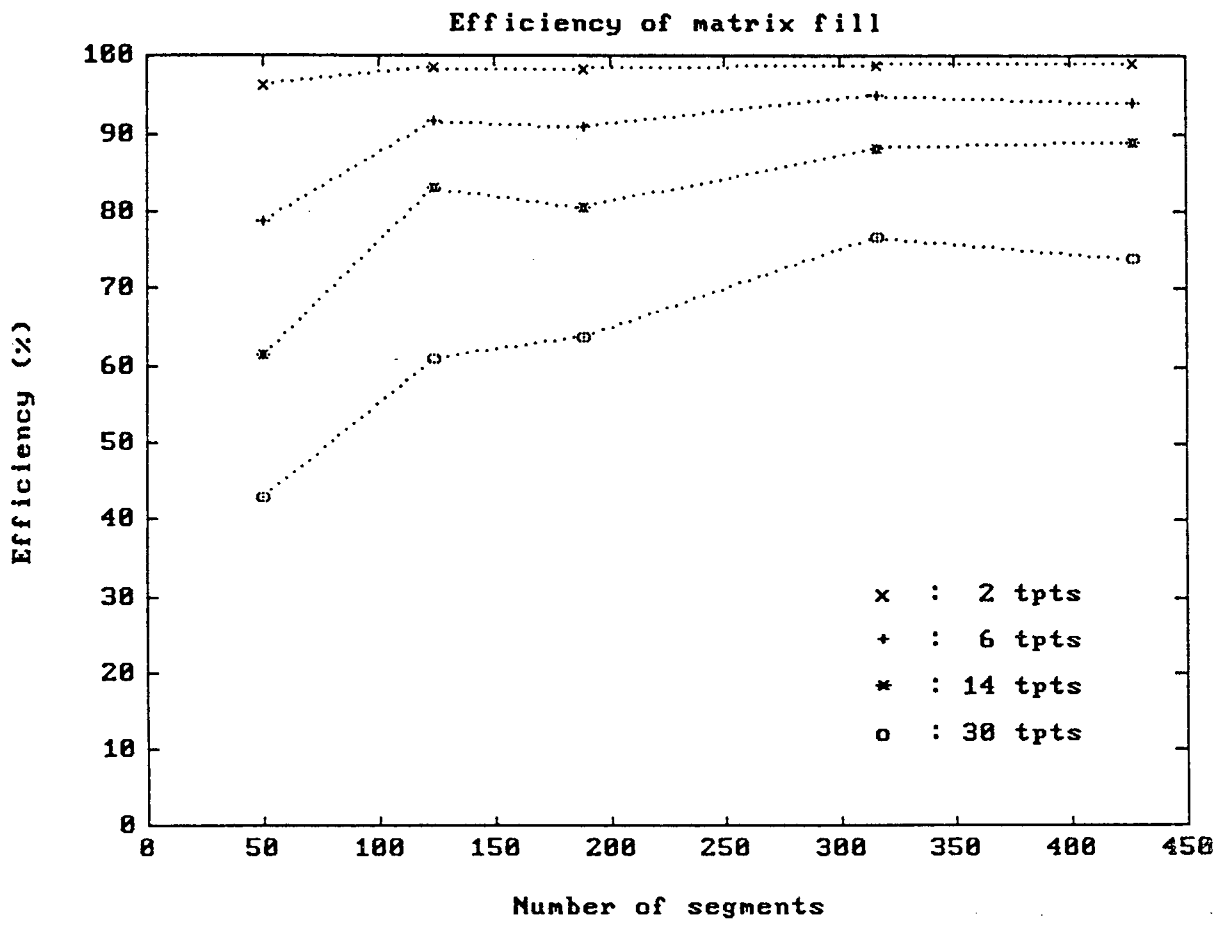


Figure 6.6: Efficiencies for the parallel matrix fill.

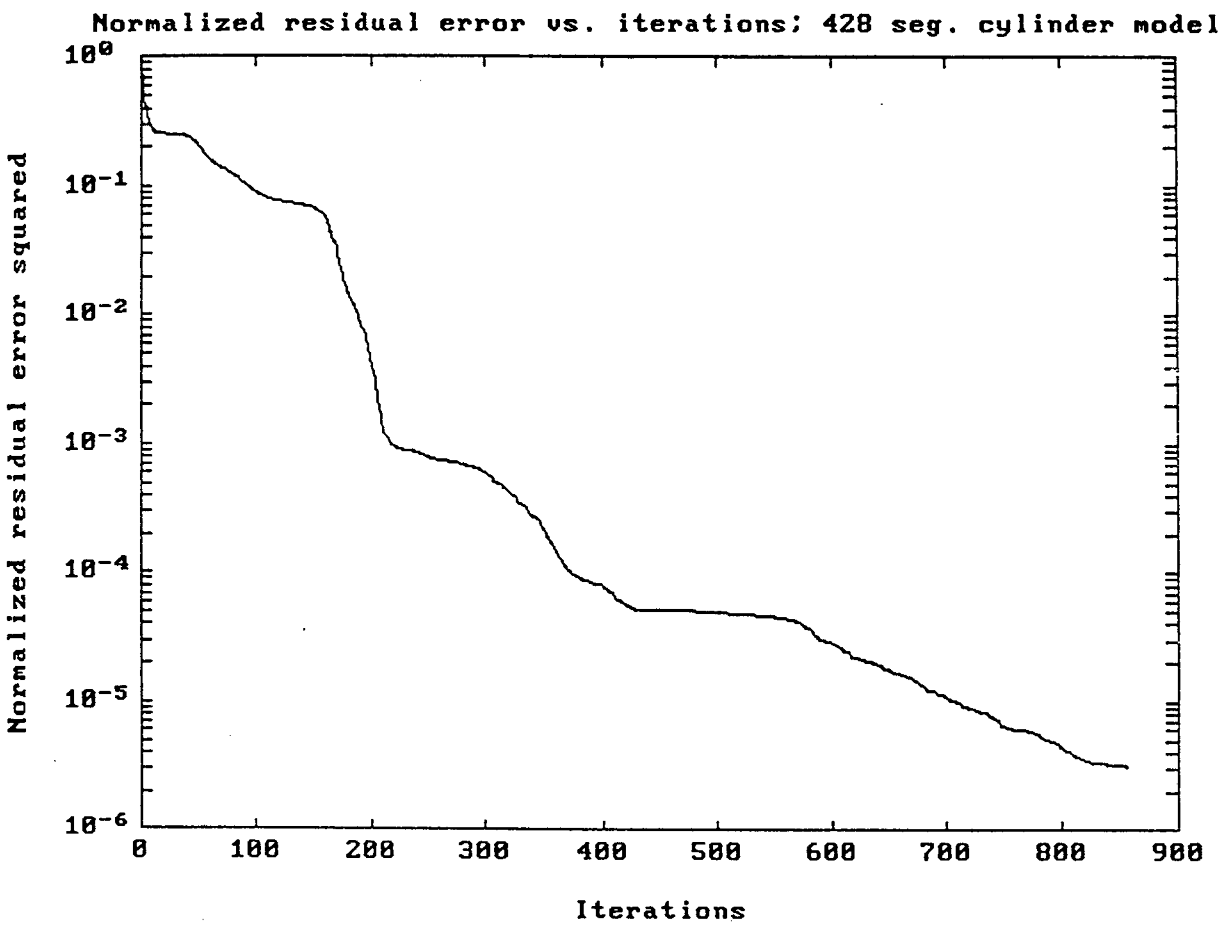


Figure 6.7: Convergence versus number of iterations: 428 unknowns.

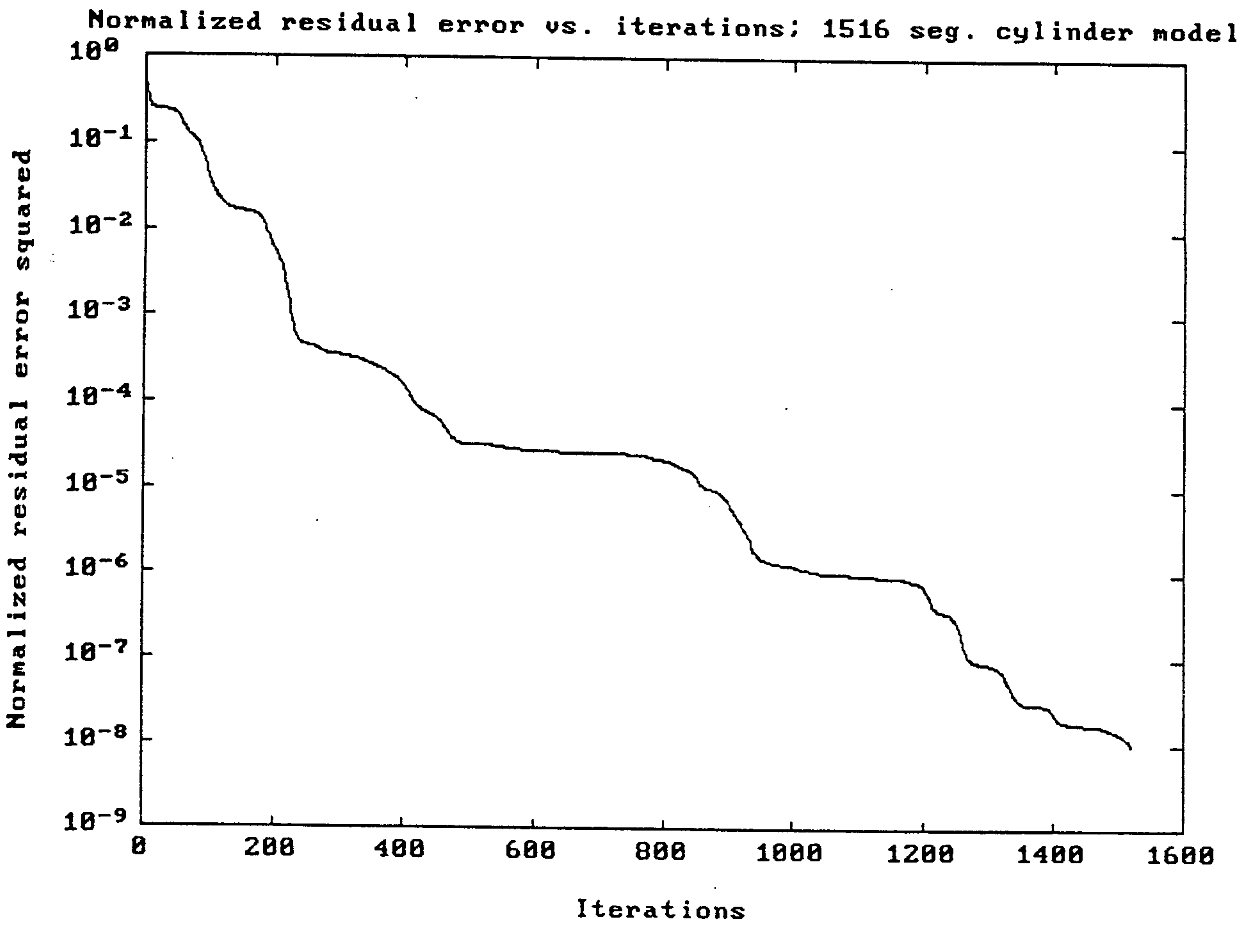


Figure 6.8: Convergence versus number of iterations: 1516 unknowns.

$$
\begin{aligned}
[u_k] &= [A][p_k] \\
\alpha_k &= \frac{[r_k]^t[q_k]^*}{[u_k]^t[w_k]^*} \\
[x_{k+1}] &= [x_k] + \alpha_k[p_k] \\
[r_{k+1}] &= [r_k] - \alpha_k[u_k] \\
[q_{k+1}] &= [q_k] - \alpha_k^* A^T[w_k] \\
[c_k] &= \frac{[r_{k+1}]^t[q_{k+1}]^*}{[r_k^]t[q_k]^*} \\
[p_{k+1}] &= [r_{k+1}] + c_k[p_k] \\
[w_k] &= [q_{k+1}] + c_k^*[w_{k+1}]
\end{aligned}
\qquad (6.6)
$$

Initial conditions are chosen as follows

$$
\begin{aligned}
[r_k] &= [b] \\
[p_k] &= [b] \\
[q_k] &= [b^*] \\
[w_k] &= [q_k]
\end{aligned}
\qquad (6.7)
$$

This choice is known as Jacob's choice [SPM90]. Note that the expressions for α_k and c_k (the latter corresponding to β_k in [SPM90]) are the complex conjugates of those in [SPM90]; the reason is the choice of inner product as

$$< [x], [y] >= [x]^t[y]^*$$

in the algorithm given above, whereas Smith used

$$< [x], [y] >= [x]^T[y]$$

so these inner products are complex conjugates of one another, and $\frac{a^*}{b^*} = (\frac{a}{b})^*$.

Both pre-conditioning and the biconjugate gradient method were investigated by the author, and results for the body of revolution formulation discussed in Chapter 2 may be found in [DM88]. Pre-conditioning was shown to be not at all attractive, but the results reported in [DM88] implied that the biconjugate gradient method converged approximately twice as fast as the CG method for moderately large problems. Unfortunately, this did not survive a more detailed investigation for really large problems using PARNEC. Results are summarized in Table 6.6[16]and it is clear that for large problems,

[16]The results shown in Table 6.6 were all computed using a Romberg error criterion of 10^{-4}, except for the double precision CG case, which used 10^{-8}. As noted in Section 6.7, the differences in impedances computed, and iterations required, using the different Romberg criteria were negligible.

Number of	Number of iterations: CG		Number of iterations: BiCG	
segments	Single	Double	Single	Double
50	23	14	15	15
124	82	75	55	54
188	155	134	102	83
316	152	131	137	109
428	519	372	Unconverged	405
876	519	405	1049	594
1196	521	409	1203	1020
1516	529	414	1455	753
1996	541	N/A	Unconverged	N/A

Table 6.6: Number of iterations required to obtain convergence as a function of precision; normalized residual of 10^{-2} specified.

the biconjugate gradient method converges rather more slowly than the CG method[17]. Note also the very strong influence of precision on the convergence of the biconjugate gradient method.

This stagnation is confirmed by Smith *et. al.* [SPM90]; they also discuss a modification to accelerate the convergence of the biconjugate gradient method, which was not implemented by the author.

6.9 Conclusions

In this chapter, a brief review of the underlying theory of NEC2 has been given. An attempt to use an Occam "harness" to provide a link between various FORTRAN processes was discussed; the poor results obtained with this approach led to the necessity for re-coding NEC2 in Occam. Fundamental problems posed by such a re-write were highlighted. A systematic (but not presently automatic) methodology for untangling "spaghetti" code was presented, using flowcharts and pseudo-code; its use was illustrated by application to one of the more formidable NEC2 sub-routines. Methods to parallelize the matrix generation, which was not as simple as for the demonstration program used in Chapter 5 for the LU method, were discussed.

Methods used to validate the code were described — initially for small structures, and then for large structures. With the former, the problem

[17]Sometimes iterative algorithms converge slowly due to coding errors. The Occam code was checked by writing a MATLAB file to implement the algorithm. Results were identical to within working precision on the computers.

was simply to ensure that the Occam re-code was a valid re-implementation of the code. With the latter, the problem was more fundamental: is the code accurate for electromagnetically very large structures, or does machine precision impose some limit on the size of structure that can be modelled? This was investigated using a symmetrical structure, solved both using, and without using, symmetry. The effect of single as opposed to double precision was also investigated; it was shown to affect only the convergence rate of the CG algorithm, not the final results. The biconjugate gradient method was also briefly investigated; it was shown that the initial promise shown by the method for small systems did not hold for large systems, where the method failed to converge.

It is important to emphasize some of the specific details of what has been achieved. The DEC VAX 3600[18] available at the University of Stellenbosch at the time of writing this dissertation took three-quarters of an hour to solve a problem with approximately 500 unknowns — a 2000 segment problem with four-way symmetry. The time-saving scales as approximately the square of the number of degrees of symmetry, as shown in Chapter 2. Hence, had the machine enough memory (which it did not) this would have taken about 12 hours to compute. Using 30 worker transputers, this problem was solved in an hour. (This is equivalent to a sustained rate of around 10 MFLOP/s.) This computation could have been done in slightly less time had the parallel LU solver been used — it had not been incorporated into PARNEC at the time of writing. Looked at from another viewpoint — the one-hour measure — the largest problem that could be handled by this specific VAX was about 600 segments. This has now been trebled to 2000 segments, increasing the maximum frequency that can be used by almost 2.

The time invested in re-writing NEC2 in Occam is time well spent, even if some future parallel system that one wishes to port the code to does not support Occam. The close similarity of Occam to other modern, structured languages, and the tremendous effort expended in modularizing the code and re-implementing the logic using structured constructs makes a further re-write a much simpler proposition, and the possibility of automating the process is much better.

In concluding this chapter, when the T9000, described in Section 3.4.2, becomes available, further substantial gains should be possible. Using 30 T9000's with enough memory, PARNEC, the code described in this chapter, should be able to handle about 6000 segment problems in one hour.

[18]See Chapter 2 for the specifications of this computer.

Chapter 7

General Conclusions

When the research documented in this dissertation was initiated, it was thought that parallel computing should have significance for computational electromagnetics. *The fundamental contribution of this research has been the successful, quantitative, demonstration that this is indeed so.* This has been done through the derivation, analysis, and implementation of efficient parallel algorithms for electromagnetic moment method formulations, and the quantification of the effectiveness, both theoretically and from measured timing data, of two fundamental algorithms required by the method of moments. A large part of a very important moment method code, NEC2, has been efficiently parallelized, and electromagnetically large problems have been both rapidly and accurately solved using the parallel NEC2.

Specific contributions of this work are the following:

- *The parallel conjugate gradient (CG) algorithm:*
  - The first published parallel CG algorithm suitable for general, full matrices with complex entries for local memory MIMD systems, in the computational electromagnetics and related literature [Dav90b].
  - A detailed theoretical analysis, supported by measured results, of the timing properties — speed-up and efficiency — of the parallel CG algorithm.
  - Publication of pseudo-code for the parallel CG algorithm.
  - A comprehensive investigation into the convergence of the CG method; confirmation of the convergence behavior presented in the literature; and the conclusion that machine precision has some effect on the number of iterations - typically 10% to 25% more iterations were required with single than with double precision.

  - An investigation into the biconjugate gradient method and the confirmation that the method is not suitable for application to large problems without some modifications as recently proposed in the literature [SPM90].

- *The parallel LU algorithm:*
  - The identification of a LU algorithm suitable for a local memory MIMD system.
  - A new, simple graphical exposition of the fundamental parallel operation of the algorithm.
  - New, simplified methods for the analysis of the parallel LU algorithm.
  - A detailed check of previously published theoretical timing results of the parallel LU algorithm.
  - New parallel forward and backward substitution algorithms using the same data distribution as the parallel LU algorithm.
  - A new, detailed investigation of the accuracy of the LU algorithm using the parallel code for large systems — 1 500 complex unknowns — by monitoring the convergence of the input impedance of a thin dipole, and also by comparing the results obtained with the parallel LU and CG codes, and the resulting conclusion that, provided the basic discretization rules of computational electromagnetics are satisfied, the LU method is accurate for large problems.

- *General properties of the parallel CG and LU matrix solvers:*
  - The demonstration of the scaling properties of the two solvers; the algorithms described will run efficiently on much larger processor arrays than those that the codes were tested on.
  - The development and description of methods for increasing efficiency, such as pipelined, concurrent communications.
  - The integration of the parallel matrix solvers with moment method codes.

- *Parallelizing NEC2:*
  - The development of PARNEC, a re-engineered version of the thin-wire parts of the important general-purpose moment method code, NEC2, in Occam 2.

  - The development and description of a systematic approach, using flowchart analysis and pseudo-code, for re-engineering a code written in an unstructured language such as FORTRAN 66 in a modern, structured language such as Occam 2.
  - A careful investigation of the accuracy of PARNEC for large problems — 1 500 to 2 000 complex valued unknowns — and the demonstration of the accuracy of the code by comparison with results computed using a smaller but equivalent problem that exploited symmetry, and the demonstration that single precision is adequate for three digits of accuracy in the solution of the matrix equation for problems of the above size.

- *General contributions to parallel computing in computational electromagnetics:*
  - The development of theoretical models for predicting algorithm performance suitable for local memory MIMD systems — of which a transputer array is but one example — and the identification of the ratio of two fundamental machine parameters (rate of computation and communication) as a critical parameter in determining the efficiency.
  - The definition of a more extensive pseudo-code than has been published in the literature for documenting parallel algorithms in an easily understood format.
  - The demonstration of the utility of the formal methods to extract parallelism: both for the LU method, with a detailed elucidation of a formal approach published in the literature, and a new application to the parallel matrix-vector problem.
  - Making parallel computing more understandable and accessible in the computational electromagnetics context [Dav90b].

This work has raised many points which would benefit from further investigation. Those issues requiring further research in primarily the computational electromagnetics field are the following:

- *Extensions of the existing work*
  - Implementing pipe-lining in the parallel CG solver.
  - Implementing pivoting in the parallel LU solver and investigating typical pivoting patterns for moment method problems.
  - Extending PARNEC to include more facilities than at present.

  - Further validation of PARNEC for different structures.
  - Incorporation of the parallel LU solver into PARNEC.
  - Further investigation of the convergence behaviour of the impedance of thin wires, discussed in Section 5.13.

- *Work of a more fundamental nature*
  - Exploitation of the Impedance Matrix Localization method.
  - Accelerating the convergence of the CG method.

Fundamental issues have also been highlighted that require attention from primarily computer scientists. The *automatic* re-engineering of old codes has been touched on in Chapter 6, and is a problem that will become increasingly pressing as fundamental software re-engineering of old codes is required for efficient execution on new architectures. The requirement for interactive parallel debuggers has also been clearly stated.

Computer technology does not stand still. The particular transputer hardware on which the algorithms described in this dissertation were implemented is already being outdated by developments in micro-electronics at the time of submission. However, the fundamental algorithms and the analysis methods described are valid for the broad class of local memory MIMD computers and, since both serial processors and parallel processors rely on the same underlying micro-electronic technology, the importance of parallel processing will not diminish in the future — as serial processors become more powerful, so will parallel processors. Thus the research represented by this dissertation, and the scalable, efficient, parallel moment method codes, will retain their relevance, permitting yet larger moment method problems to be solved on future parallel computers as computer technology improves.

Bibliography

[AHZ90] R. J. Allan, L. Heck, and S. Zurek. Parallel FORTRAN in scientific computing: a new Occam harness called Fortnet. *Computer Physics Communications*, 59(2):325–344, June 1990.

[Bal89] C. A. Balanis. *Advanced Engineering Electromagnetics.* John Wiley and Sons, New York, 1989.

[BF85] R. L. Burden and J. D. Faires. *Numerical Analysis.* Prindle, Weber and Schmidt, Boston, 3rd edition, 1985.

[BMY73] W. D. Burnside, R. J. Marhefka, and C. L. Yu. Roll-plane analysis of on-aircraft antennas. *IEEE Trans. Antennas Propagat.*, AP-21(6):780–786, November 1973.

[Boj82] N. N. Bojarski. The k-space formulation of the scattering problem in the time domain. *J. Acoust. Soc. Am.*, 72(2):570–584, August 1982.

[BP81a] G. J. Burke and A. J. Poggio. Numerical Electromagnetics Code (NEC) - Method of Moments; Part I: Program Description - Theory. January 1981.

[BP81b] G. J. Burke and A. J. Poggio. Numerical Electromagnetics Code (NEC) - Method of Moments; Part II: Program Description - Code. January 1981.

[BP81c] G. J. Burke and A. J. Poggio. Numerical Electromagnetics Code (NEC) - Method of Moments; Part III: User Guide. January 1981.

[BT89] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and distributed computation: Numerical Methods.* Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[BYM81] W. D. Burnside, C. L. Yu, and R. J. Marhefka. A technique to combine the Geometrical Theory of Diffraction and the moment method. In R.C. Hansen, editor, *Geometric Theory of Diffraction*, pages 393–399, IEEE Press, New York, 1981.

[Can90] F. X. Canning. The impedance matrix localization (IML) method for moment-method calculations. *IEEE Antennas Propagat. Magazine*, 32(5):18–30, October 1990.

[Cwi91] T. Cwik. The solution and numerical accuracy of large MoM problems. In *Symposium digest of the 1991 URSI Radio Science Meeting*, page 96, June 1991. Held in London, Ontario, Canada.

[Dav86] D. B. Davidson. *Predicting electromagnetic radiation from and coupling between antennas mounted on a body of revolution using the method of moments technique.* Master's thesis, Dept. Electronic Engineering, University of Pretoria, December 1986.

[Dav90a] D. B. Davidson. Parallel matrix solvers for moment method codes using transputer arrays. In *Symposium digest of the 1990 URSI Radio Science Meeting*, page 70, May 1990. Held in Dallas, Texas, USA.

[Dav90b] D. B. Davidson. A parallel processing tutorial. *IEEE Antennas Propagat. Magazine*, 32(2):6–19, April 1990.

[Dav91a] D. B. Davidson. Antenna Modelling Code Project PARNEC Version 1.0 Part I: User's Guide. February 1991. Institute for Electronics, University of Stellenbosch, 15 pages.

[Dav91b] D. B. Davidson. Antenna Modelling Code Project PARNEC Version 1.0 Part II: Code Manual. February 1991. Institute for Electronics, University of Stellenbosch, 105 pages.

[Dav91c] D. B. Davidson. Parallel LU decomposition on a transputer array. In *Symposium digest, Vol III, of the 1991 IEEE AP-S International Symposium*, pages 1500–1503, June 1991. Held in London, Ontario, Canada.

[DC89] D. B. Davidson and J. H. Cloete. NEC2 in a MIMD computing environment. In *Symposium digest of the 1989 URSI Radio Science Meeting*, page 131, June 1989. Held in San Jose, California, USA.

[DDSvdV91] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers.* SIAM, Philadelphia, 1991.

[Dij76] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[DM85] L. M. Delves and J. L. Mohamed. *Computational Methods for Integral Equations.* Cambridge University Press, Cambridge, 1985.

[DM87] D. B. Davidson and D. A. McNamara. Predicting radiation patterns from aperture antennas on structures using the method of moments body of revolution technique. *The Transactions of the South African Institute of Electrical Engineers*, 78(2):25–30, December 1987.

[DM88] 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*, 1(7):243–246, September 1988.

[Dud85] D. G. Dudley. Error minimization and convergence in numerical methods. *Electromagnetics*, 5(2-3):89–98, 1985.

[Ell81] R. S. Elliott. *Antenna Theory and Design.* Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[FJL*88] 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.* Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[Fly72] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–60, September 1972.

[FS66] B. A. Finlayson and L. E. Scriven. The method of weighted residuals - a review. *Applied Mechanics Reviews*, 19(9):735–48, September 1966.

[Gal90] J. Galletly. *Occam 2.* Pitman, London, 1990.

[GHN87] G. A. Geist, M. T. Heath, and E. Ng. Parallel algorithms for matrix computations. In L. H. Jamieson, D. B. Gannon, and R. J. Douglass, editors, *The Characteristics of Parallel Algorithms*, MIT Press, Cambridge,MA, 1987.

[GPS90] K. A. Gallivan, R. J. Plemmons, and A. H. Sameh. Parallel algorithms for dense linear algebra computations. *SIAM Review*, 32(1):54–135, March 1990.

[Gus88] J. L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, May 1988.

[Haf89] C. Hafner. Parallel computation of electromagnetic fields on transputers. *IEEE Antennas Propagat. Society Newsletter*, 31(5):6–12, October 1989.

[Han81] R. C. Hansen, editor. *Geometric Theory of Diffraction.* IEEE Press, New York, 1981.

[Han90] R. C. Hansen, editor. *Moment Methods in Antennas and Scattering.* Artech House, Boston, 1990.

[Har61] R. F. Harrington. *Time-Harmonic Electromagnetic Fields.* McGraw-Hill, New York, 1961.

[Har82] R. F. Harrington. *Field Computation by Moment Methods.* Robert E. Krieger, Malabar, Florida, 1982. Reprint of 1968 edition.

[Har87] R. F. Harrington. The method of moments in electromagnetics. *Journal of electromagnetic waves and applications*, 1(3):181–200, 1987.

[Har90] R. F. Harrington. Origin and development of the method of moments for field computation. *IEEE Antennas Propagat. Magazine*, 32(3):31–36, June 1990.

[Hel78] D. Heller. A survey of parallel linear algorithms in numerical linear algebra. *SIAM Review*, 20(4):740–777, October 1978.

[HJ88] R. W. Hockney and C. R. Jesshope. *Parallel Computers 2.* Adam Hilger, Bristol, 1988.

[HM89] H. A. Haus and J. R. Melcher. *Electromagnetic Fields and Energy.* Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

[INM89] INMOS. The transputer databook, 2nd edition. 1989.

[INM91] INMOS. The T9000 transputer products overview manual, 1st edition. 1991.

[Jam86] G. L. James. *Geometrical Theory of Diffraction for Electromagnetic Waves.* Peter Peregrinus, 3rd revised edition, 1986.

[Jen85] A. Jennings. *Matrix Computation for Engineers and Scientists.* John Wiley and Sons, Chichester, 1985.

[Jon87] D. S. Jones. *Methods in Electromagnetic Wave Propagation.* Oxford University Press, Oxford, 1987. Volume 1: Theory and Guided Waves.

[JvR90] D. J. Janse van Rensburg. *On the computation of electromagnetic observables of conducting thin wire radiators.* PhD dissertation, Dept. Electronic & Computer Engineering, University of Pretoria, 1990.

[JvRM88] D. J. Janse van Rensburg and D. A. McNamara. Conjugate gradient algorithm based on matrix representation of linear operators in finite dimensional Hilbert spaces. *Electronic Letters*, 24(7):405–406, 31st March 1988.

[JvRM89] D. J. Janse van Rensburg and D. A. McNamara. A rapidly convergent Galerkin-Collocation (GC) method for the analysis of electromagnetic scattering from thin-wire structures. *Microwave and Optical Technology Letters*, 2(10):355–357, October 1989.

[JvRM90] D. J. Janse van Rensburg and D. A. McNamara. On quasi-static source models for wire dipole antennas. *Microwave and Optical Technology Letters*, 3(11):396–398, November 1990.

[KF90] A. H. Karp and H. P. Flatt. Measuring parallel processor performance. *Communications of the ACM*, 33(5):539–543, May 1990.

[KH90] R. Kastner and N. Herscovici. A concise conjugate gradient computation of plate problems with many excitations. *IEEE Trans. Antennas Propagat.*, 38(8):1239–1243, August 1990.

[Kon86] J. A. Kong. *Electromagnetic Wave Theory.* John Wiley and Sons, New York, 1986.

[Kre78] E. Kreyszig. *Introductory Functional Analysis with Applications.* John Wiley and Sons, New York, 1978.

[lR89] J. J. le Roux. *The Numerical Electromagnetic Analysis of Complex Structures with Specific Reference to HF Radiation from the Aerospatiale SA 330 Puma Helicopter.* Master's thesis, Dept. E & E Engineering, University of Stellenbosch, 1989.

[lRBdPC88] J. J. le Roux, P. J. Bakkes, J. J du Plessis, and J. H. Cloete. Execution of the NEC2 electromagnetic moment method code on the INMOS T800 transputer. *Electronic Letters*, 24(16):991–992, 4th August 1988.

[Lud87] A. C. Ludwig. Wire grid modelling of surfaces. *IEEE Trans. Antennas Propagat.*, AP-35(9):1045–1048, September 1987.

[LWA91] A. D. Lipworth, A. J. Walker, and H. J. Annegarn. FORTRAN package renewal using object-centred design techniques. *The Transactions of the South African Institute of Electrical Engineers*, 82(1):43–51, March 1991.

[Mal90] J. F. Malan. Private communication. 1990. University of Stellenbosch, Stellenbosch, South Africa.

[MAT89] MATLAB: The MathWorks, Inc. Version 3.5 User's Guide. 1989. The MathWorks, Inc.

[Max54] J. C. Maxwell. *A Treatise on Electricity and Magnetism.* Dover, New York, 1954. Republication of Clarendon Press 1891 (3rd edition).

[Mil88] E. K. Miller. A selective survey of computational electromagnetics. *IEEE Trans. Antennas Propagat.*, 36(9):1281–1305, September 1988.

[Mil90a] E. K. Miller. PCs for AP. *IEEE Trans. Antennas Propagat. Magazine*, 32(5):48–51, October 1990.

[Mil90b] E. K. Miller. Private communication. June 1990. 1990 IEEE AP-S International Symposium, Dallas, Texas.

[Mil91a] E. K. Miller. Private communication. October 1991. Examiner's report.

[Mil91b] E. K. Miller. Private communication. June 1991. 1991 IEEE AP-S International Symposium, London, Ontario, Canada.

[Mit73] R. Mittra, editor. *Computer Techniques for Electromagnetics.* Pergamon, Oxford, 1973.

[MK75] R. Mittra and C. A. Klein. Stability and convergence of moment method solutions. In R. Mittra, editor, *Numerical and Asymptotic Techniques in Electromagnetics*, Springer-Verlag, New York, 1975.

[Mod88] J. J. Modi. *Parallel algorithms and matrix computation.* Clarendon Press, Oxford, 1988.

[Mor90] M. A. Morgan, editor. *Finite Element and Finite Difference Methods in Electromagnetic Scattering. Progress in Electromagnetic Research Series Number 2*, Elsevier, New York, 1990.

[MP86] J. Moore and R. Pizer, editors. *Moment Methods in Electromagnetics Techniques and Applications.* Research Studies Press, Letchworth, Hertfordshire, 1986.

[MPM90] D. A. McNamara, C. W. I Pistorius, and J. A. G. Malherbe. *The Uniform Geometrical Theory of Diffraction.* Artech House, Boston, 1990.

[Nah88] P. J. Nahin. *Oliver Heaviside: Sage in Solitude.* IEEE Press, New York, 1988.

[NdV78] D. H. Norrie and G. de Vries. *An Introduction to Finite Element Analysis.* Academic Press, New York, 1978.

[Ney85] M. M. Ney. Method of moments as applied to electromagnetic problems. *IEEE Trans. Microwave Theory Tech.*, MTT-33(10):972–980, October 1985.

[NF90a] D. C. Nitch and A. P. C. Fourie. Adapting the numerical electromagnetics code to run in parallel on a network of transputers. In *Proceedings of the 1990 IEEE/SAIEE Joint AP-MTT Symposium*, pages 31–38, August 1990. Held in Somerset West, South Africa.

[NF90b] 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*, 5(2):76–86, Winter 1990.

[NRS87] K. Nayanthara, S. M Rao, and T. K. Sarkar. Analysis of two-dimensional conducting and dielectric bodies utilizing the conjugate gradient method. *IEEE Trans. Antennas Propagat.*, AP-35(4):451–453, April 1987.

[OG76a] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319–340, 1976.

[OG76b] S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *Communications of the ACM*, 19(5):279–285, May 1976.

[Par88] Parallel FORTRAN. User Guide Version 2.0.1. 1988. 3L Ltd., Peel House, Ladywell, Livingstone EH54 6AG, Scotland.

[PM73] A. J. Poggio and E. K. Miller. Integral equation solutions of three dimensional scattering problems. In R. Mittra, editor, *Computer Techniques for Electromagnetics*, Pergamon, Oxford, 1973.

[PM85a] A. F. Peterson and R. Mittra. Method of conjugate gradients for the numerical solution of large-body electromagnetic scattering problems. *Journal of the Optical Society of America, Part A*, 2(6):971–977, June 1985.

[PM85b] A. F. Peterson and R. Mittra. On the implementation and performance of iterative methods for computational electromagnetics. December 1985. Technical report No 85-9, Electromagnetic Communication Lab., Dept. of Electrical and Computer Engineering, Univ. of Illinois, Urbana, IL.

[PSM88] 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.*, 36(8):1177–1179, August 1988.

[Reg87] E. Regis. *Who got Einstein's office?* Addison-Wesley, Reading, MA, 1987.

[Ric91] J. H. Richmond. On the variational aspects of the moment method. *IEEE Trans. Antennas Propagat.*, 39(4):473–479, April 1991.

[RP88] S. L. Ray and A. F. Peterson. Error and convergence in numerical implementations of the conjugate gradient method. *IEEE Trans. Antennas Propagat.*, 36(12):1824–1827, December 1988.

[SA85] T. K. Sarkar and E. Arvas. On a class of finite step iterative methods (conjugate directions) for the solution of an operator equation arising in electromagnetics. *IEEE Trans. Antennas Propagat.*, AP-33(10):1058–1066, October 1985.

[Sar86] T. K. Sarkar. The conjugate gradient method as applied to electromagnetic field problems. *IEEE Antennas Propagat. Society Newsletter*, 28(4):5–14, August 1986.

[Sar87a] T. K. Sarkar. On the application of the generalized biconjugate gradient method. *Journal of Electromagnetic Waves and Applications*, 1(3):223–242, 1987.

[Sar87b] T. K. Sarkar. Some of the misconceptions associated with the conjugate gradient method. In *Symposium digest, Vol I, of the 1987 IEEE AP-S International Symposium*, pages 84–86, June 1987. Held in Blacksburg, Virginia, USA.

[Sar88] T. K. Sarkar. Comments on "Comparison of the FFT conjugate gradient method and the finite-difference time domain method for the 2-D absorption problem" and reply by D. T. Borup and O. P. Gandhi. *IEEE Trans. Microwave Theory Tech.*, 36(1):166–170, January 1988.

[Sch90] A. M. Schuilenburg. Parallelisation of basic scattering code. In *Proceedings of the 1990 IEEE/SAIEE Joint AP-MTT Symposium*, pages 259–266, August 1990. Held in Somerset West, South Africa.

[SD90] P. Steyn and D. B. Davidson. Solution stability of iterative schemes utilizing the FFT. In *Symposium digest Vol II of the 1990 IEEE AP-S International Symposium*, pages 614–617, May 1990. Held in Dallas, Texas, USA.

[SF90] P. P. Silvester and R. L. Ferrari. *Finite Elements for Electrical Engineers.* Cambridge University Press, Cambridge, 2nd edition, 1990.

[Skw81] J. K. Skwirzynski, editor. *Theoretical methods for determining the interaction of electromagnetic waves with structures.* Sijthoff and Noordhoff, Alphen aan den Rijn, The Netherlands, 1981.

[SPM89] 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.*, 37(11):1490–1493, November 1989.

[SPM90] C. F. Smith, A. F. Peterson, and R. Mittra. The biconjugate gradient method for electromagnetic scattering. *IEEE Trans. Antennas Propagat.*, 38(6):938–940, June 1990.

[ST81] W. L. Stutzman and G. A. Thiele. *Antenna Theory and Design.* John Wiley and Sons, New York, 1981.

[Sto90a] V. Stover. Rendezvous with a computer scientist: methods of software validation. *Applied Computational Electromagnetics Society Newsletter*, 5(2):16–23, July 1990.

[Sto90b] V. Stover. Rendezvous with a computer scientist: software testing. *Applied Computational Electromagnetics Society Newsletter*, 5(1):36–43, March 1990.

[Sun88] D. Sundholm. A block preconditioned conjugate gradient method for solving high-order finite element matrix equations. *Computer Physics Communications*, 49, 1988.

[Tai71] C. T. Tai. *Dyadic Green's Functions in Electromagnetic Theory.* Intext Educational Publishers, Scranton, 1971.

[TN81] G. A. Thiele and T. H. Newhouse. A hybrid technique for combining moment methods with the Geometrical Theory of Diffraction. In R. C. Hansen, editor, *Geometric Theory of Diffraction*, pages 385–392, IEEE Press, New York, 1981.

[vdV88] J. G. G. van de Vorst. The formal development of a parallel program performing LU-decomposition. *Acta Informatica*, 26:1–17, 1988.

[vdVB89] J. G. G. van de Vorst and R. H. Bisseling. Parallel LU-decomposition on a transputer network. In *Lecture Notes in Computer Science*, pages 61–77, Springer-Verlag, Berlin, 1989.

[Vil89] N. Viljoen. An overview of transputer based computers and a description of the MC^2 machine. In H. Neishlos, editor, *Parallel processing: technology and applications*, pages 131–137, IOS,Van Diemenstraat 94, 1013 CN Amsterdam, 1989. Proceedings of the International Symposium on Parallel Processing, 26-28 October 1988, Johannesburg, South Africa.

[Wan90a] J. J. H. Wang. Comments on "From 'reaction concept' to 'conjugate gradient': have we made any progress?" and reply by T. K. Sarkar. *IEEE Trans. Antennas Propagat. Magazine*, 32(1):64–65, February 1990.

[Wan90b] J. J. H. Wang. Further comments on "From 'reaction concept' to 'conjugate gradient': have we made any progress?" and reply by T. K. Sarkar. *IEEE Trans. Antennas Propagat. Magazine*, 32(4):70–71, August 1990.

[Wan91] J. J. H. Wang. *Generalized Moment Methods in Electromagnetics.* John Wiley and Sons, New York, 1991.

[WK76] T. T. Wu and R. W. P. King. The tapered antenna and its application to the junction problem for thin wires. *IEEE Trans. Antennas Propagat.*, AP-24(1):42–45, January 1976.

[Zwa91] A. P. M. Zwamborn. *Scattering by Objects with Electric Contrast.* PhD dissertation, Faculty of Engineering, Technical University of Delft, June 1991.

Appendix A

CG Configuration Code

Notes:

1. The parameters `number.of.transputers`, `num.nodes` and `num.leaves` are defined in a separate file of constants.
2. This configuration code was developed for the Occam 2 TDS 3 enviroment.
3. The {{{ and }}} indicate the start and end of "folds".

```
--{{{  hard configuration
--{{{  Compiler USAGE checking notes
-- Due to compiler defects, the USAGE checking option must be disabled
-- to compile the following code. It should ONLY be disabled once the rest of
-- the code is working. Note that the construct is valid, accessing disjoint
-- arrays in parallel.
--}}}
--{{{  Notes on the configuration
-- The configuration is designed to be as general purpose as possible.
-- The foloowing important limitations must, however, be noted:
-- 1. The MCC can only switch even to even and odd to odd links to the
--    the hardware switching network.
-- 2. A boot path must be provided to allow the root to boot the whole
--    network over only one link. This is provided by a special treatment
--    of processors 1 and 2, whose lower links follow the pattern
--    of the rest of the network but whose upper links are special and
--    these processors also have side-ways links to provide a boot path.
--    Hence processors 1 and 2 are excluded from the nodal declarations.
--
--    This method is valid for all except the case of only two
--    worker processors; this can however be fixed in the library
--    file ''cgtpts'' by declaring num.leaves as 0, not 2, for the two
--    processor case. Unfortunately, the configuration language is not
--    sufficiently powerful to handle this special case in the code.
--
--}}}
--{{{  channel declarations
```

```
[2]CHAN OF messages boot.path :
[number.of.transputers+5]CHAN OF messages down              :
[number.of.transputers+5]CHAN OF messages up                :
-- +1 because first channel element (element 0) not used
-- and additional +4 for case of two processors (used instead
-- of dummy channels for that case).
[(number.of.transputers/2)+1]CHAN OF messages dummy.down.left :
[(number.of.transputers/2)+1]CHAN OF messages dummy.down.right:
[(number.of.transputers/2)+1]CHAN OF messages dummy.up.left   :
[(number.of.transputers/2)+1]CHAN OF messages dummy.up.right  :
--}}}
PLACED PAR -- See Galletly p123 for details.
  -- processor 1 and 2 ; special treatment to provide boot path
  --{{{  processor 1
  -- processor 1
  PLACED PAR
    PROCESSOR 1 T8
      PLACE down[3]  AT link1.out :
      PLACE up[3]    AT link1.in  :
      PLACE down[4] AT link2.out :
      PLACE up[4]   AT link2.in  :
      PLACE down[1] AT link3.in  :
      PLACE up[1]   AT link3.out :
      PLACE boot.path[0] AT link0.out :
      PLACE boot.path[1] AT link0.in:
      nec.worker(down[1],up[1],
      down[3],up[3],down[4],up[4])
  --}}}
  --{{{  processor 2
  -- processor 2
  PLACED PAR
    PROCESSOR 2 T8
      PLACE down[5]  AT link1.out :
      PLACE up[5]    AT link1.in  :
      PLACE down[6] AT link2.out :
      PLACE up[6]   AT link2.in  :
      PLACE down[2] AT link3.in  :
      PLACE up[2]   AT link3.out :
      PLACE boot.path[0] AT link0.in :
      PLACE boot.path[1] AT link0.out:
      nec.worker(down[2],up[2],
      down[5],up[5],down[6],up[6])
  --}}}
  -- nodes; note excludes nodes 1 and 2
  --{{{  odd numbered nodes
  -- odd numbered nodes
  PLACED PAR i = 2 FOR (num.nodes/2) -1
    VAL Index IS (2*i)-1:
    PROCESSOR Index T8
      VAL Parent IS Index:
      VAL Left  IS (2*Index)+1:
      VAL Right IS (2*Index)+2:
      PLACE down[Left]  AT link1.out :
      PLACE up[Left]    AT link1.in  :
      PLACE down[Right] AT link2.out :
      PLACE up[Right]   AT link2.in  :
      PLACE down[Parent] AT link3.in  :
      PLACE up[Parent]   AT link3.out :
      nec.worker(down[Parent],up[Parent],
      down[Left],up[Left],down[Right],up[Right])
  --}}}
  --{{{  even numbered nodes
  -- even numbered nodes
```

```
PLACED PAR i = 2 FOR (num.nodes/2) -1
  VAL Index IS (2*i):
  PROCESSOR Index T8
    VAL Parent IS Index:
    VAL Left  IS (2*Index)+1:
    VAL Right IS (2*Index)+2:
    PLACE down[Left]  AT link1.out  :
    PLACE up[Left]    AT link1.in   :
    PLACE down[Right] AT link2.out  :
    PLACE up[Right]   AT link2.in   :
    PLACE down[Parent] AT link0.in   :
    PLACE up[Parent]   AT link0.out :
    nec.worker(down[Parent],up[Parent],
    down[Left],up[Left],down[Right],up[Right])
--}}}
-- leaves; note must be commented out for the case of only two processors
--{{{  odd numbered leaves
-- odd numbered leaves
PLACED PAR i = (num.nodes/2)+1 FOR num.leaves/2
  VAL Index IS (2*i)-1:
  PROCESSOR Index T8
    VAL Parent IS Index:
    VAL Dummy.Index IS Index - (num.nodes+1): -- to minimize num of dummy chans
    PLACE down[Parent] AT link3.in   :
    PLACE up[Parent]   AT link3.out  :
    PLACE dummy.up.left[Dummy.Index]     AT link1.in:
    PLACE dummy.down.left[Dummy.Index]   AT link1.out:
    PLACE dummy.up.right[Dummy.Index]    AT link2.in:
    PLACE dummy.down.right[Dummy.Index]  AT link2.out:
    nec.worker(down[Parent],up[Parent],
    dummy.down.left[Dummy.Index],dummy.up.left[Dummy.Index],
    dummy.down.right[Dummy.Index],dummy.up.right[Dummy.Index])

--}}}
--{{{  even numbered leaves
-- even numbered leaves
PLACED PAR i = (num.nodes/2)+1 FOR num.leaves/2
  VAL Index IS (2*i):
  PROCESSOR Index T8
    VAL Parent IS Index:
    VAL Dummy.Index IS Index - (num.nodes+1): -- to minimize num of dummy chans
    PLACE down[Parent] AT link0.in   :
    PLACE up[Parent]   AT link0.out  :
    PLACE dummy.up.left[Dummy.Index]     AT link1.in:
    PLACE dummy.down.left[Dummy.Index]   AT link1.out:
    PLACE dummy.up.right[Dummy.Index]    AT link2.in:
    PLACE dummy.down.right[Dummy.Index]  AT link2.out:
    nec.worker(down[Parent],up[Parent],
    dummy.down.left[Dummy.Index],dummy.up.left[Dummy.Index],
    dummy.down.right[Dummy.Index],dummy.up.right[Dummy.Index])
--}}}
--}}}
```

Appendix B

LU Configuration Code

Notes:

1. This configuration code is for the Occam 2 Toolset.
2. The parameter `mesh.size` is defined in a separate file of constants.

```
#INCLUDE "linkaddr.inc" -- link addresses
#INCLUDE "hostio.inc"   -- host i/o constants
#INCLUDE "pmomlib.inc"   -- constant and protocol definitions
#USE "master21.c8h"     -- linked worker process
#USE "worker21.c8h"     -- linker master process
-- Declare the channels.
CHAN OF messages soft2.in, soft2.out :
[mesh.size]CHAN OF messages dummy1, dummy2 :
[mesh.size+1][mesh.size+1] CHAN OF messages left.in,
                        right.in,
                        up.in,
                        down.in :

-- Special channels (link 0) to communicate with host.
CHAN OF SP fs, ts:

-- Placement statements

PLACED PAR
  -- 00 process.
  VAL INT j IS 0 :
  VAL INT i IS 0 :
  PLACED PAR
    PROCESSOR ((i*mesh.size)+j) T8
      PLACE fs AT link0.in                : -- from host; special for 00
      PLACE ts AT link0.out               : -- to host;       "
      PLACE left.in[i][j]    AT link1.in    : -- left in; special for 00
      PLACE right.in[i][mesh.size-1] AT link1.out : --left.out;    "
      PLACE right.in[i][j]   AT link3.in    : -- right in
      PLACE left.in[i][j+1]  AT link3.out   : -- right out
      PLACE down.in[i][j]    AT link2.in    : -- down in
      PLACE up.in[i+1][j]    AT link2.out   : -- down out
      -- Declare internal soft channel for the 00 processor only
    CHAN OF messages to.master2, from.master2 :
    IF
      number.of.transputers > 1
        PAR
          master(fs,ts,to.master2, from.master2)
          -- 00 process (to run on master transputer)
          worker(left.in[i][j], right.in[i][mesh.size-1],
            right.in[i][j], left.in[i][j+1],
            from.master2, to.master2,
            down.in[i][j], up.in[i+1][j],i,j)
      number.of.transputers = 1
        master(fs,ts,to.master2, from.master2)


-- first row processes except last column
VAL INT i IS 0 :
PLACED PAR j = 1 FOR mesh.size -2
  PROCESSOR ((i*mesh.size)+j) T8
    PLACE left.in[i][j]    AT link1.in  : -- left in
    PLACE right.in[i][j-1] AT link1.out : -- left out
    PLACE right.in[i][j]   AT link3.in  : -- right in
    PLACE left.in[i][j+1]  AT link3.out : -- right out
    PLACE up.in[i][j]      AT link0.in  : -- up in  (unused)
    PLACE dummy1[j]        AT link0.out : -- up out (unused)
    PLACE down.in[i][j]    AT link2.in  : -- down in
    PLACE up.in[i+1][j]    AT link2.out : -- down out
    worker(left.in[i][j], right.in[i][j-1],
         right.in[i][j], left.in[i][j+1],
         up.in[i][j], dummy1[j],
         down.in[i][j], up.in[i+1][j],i,j)

-- upper right-most element
VAL INT i IS 0 :
VAL INT j IS mesh.size -1 :
PROCESSOR ((i*mesh.size)+j) T8
  PLACE left.in[i][j]    AT link1.in  : -- left in
  PLACE right.in[i][j-1] AT link1.out : -- left out
  PLACE right.in[i][j]   AT link3.in  : -- right in
  PLACE left.in[i][0]    AT link3.out : -- right out (wraparound)
  PLACE up.in[i][j]      AT link0.in  : -- up in  (unused)
  PLACE dummy1[j]        AT link0.out : -- up out (unused)
  PLACE down.in[i][j]    AT link2.in  : -- down in
  PLACE up.in[i+1][j]    AT link2.out : -- down out
  worker(left.in[i][j], right.in[i][j-1],
       right.in[i][j], left.in[i][0],
       up.in[i][j], dummy1[j],
       down.in[i][j], up.in[i+1][j],i,j)

-- first column processes
VAL INT j IS 0 :
PLACED PAR i = 1 FOR mesh.size -1
  PROCESSOR ((i*mesh.size)+j) T8
    PLACE left.in[i][j]    AT link1.in   : -- left in
    PLACE right.in[i][mesh.size-1] AT link1.out  : -- left out (wraparound)
    PLACE right.in[i][j]   AT link3.in   : -- right in
    PLACE left.in[i][j+1]  AT link3.out  : -- right out
    PLACE up.in[i][j]      AT link0.in   : -- up in
    PLACE down.in[i-1][j]  AT link0.out  : -- up out
    PLACE down.in[i][j]    AT link2.in   : -- down in
    PLACE up.in[i+1][j]    AT link2.out  : -- down out
    worker(left.in[i][j], right.in[i][mesh.size-1],
         right.in[i][j], left.in[i][j+1],
         up.in[i][j], down.in[i-1][j],
          down.in[i][j], up.in[i+1][j],i,j)

  -- interior processes
  PLACED PAR i = 1 FOR mesh.size -1
    PLACED PAR j = 1 FOR mesh.size -2
      PROCESSOR ((i*mesh.size)+j) T8
        PLACE left.in[i][j]     AT link1.in  : -- left in
        PLACE right.in[i][j-1]  AT link1.out : -- left out
        PLACE right.in[i][j]    AT link3.in  : -- right in
        PLACE left.in[i][j+1]   AT link3.out : -- right out
        PLACE up.in[i][j]       AT link0.in  : -- up in
        PLACE down.in[i-1][j]   AT link0.out : -- up out
        PLACE down.in[i][j]     AT link2.in  : -- down in
        PLACE up.in[i+1][j]     AT link2.out : -- down out
        worker(left.in[i][j], right.in[i][j-1],
          right.in[i][j], left.in[i][j+1],
          up.in[i][j], down.in[i-1][j],
          down.in[i][j], up.in[i+1][j],i,j)

  -- last column (except upper right-most element)
  VAL INT j IS mesh.size -1:
  PLACED PAR i = 1 FOR mesh.size -1
    PROCESSOR ((i*mesh.size)+j) T8
      PLACE left.in[i][j]     AT link1.in  : -- left in
      PLACE right.in[i][j-1]  AT link1.out : -- left out
      PLACE right.in[i][j]    AT link3.in  : -- right in
      PLACE left.in[i][0]     AT link3.out : -- right out
      PLACE up.in[i][j]       AT link0.in  : -- up in
      PLACE down.in[i-1][j]   AT link0.out : -- up out
      PLACE down.in[i][j]     AT link2.in  : -- down in
      PLACE up.in[i+1][j]     AT link2.out : -- down out
      worker(left.in[i][j], right.in[i][j-1],
           right.in[i][j], left.in[i][0],
           up.in[i][j], down.in[i-1][j],
           down.in[i][j], up.in[i+1][j],i,j)
```

Appendix C

FORTRAN Precision Conversion

The procedure to convert the single precision FORTRAN version of NEC2 to a double precision version is summarized below. The procedure was carried out manually; as an one-off procedure, it was not deemed worthwhile automating the process.

- `IMPLICIT DOUBLE PRECISION (A-H,O-Z)` was added at the start of each program unit.
- `COMPLEX` was changed to `DOUBLE COMPLEX`.
- `REAL` (and `REAL*4`) declarations were changed to `DOUBLE PRECISION`.
- `REAL*8` was changed to `DOUBLE PRECISION` (for uniformity).
- `CMPLX` was changed to `DCMPLX`; note that `CMPLX` takes two `REAL*8` and creates a `COMPLEX*8` from them. This could lead to a baffling lack of precision if assigned to a `COMPLEX*16` data type, which will of course simply convert the `COMPLEX*8` to `COMPLEX*16` using FORTRAN's (un-type-checked) re-typing rules.
- `REAL` was changed to `DREAL` and `AIMAG` to `DIMAG`; there are no generic functions for this.
- Specific functions (eg `CSQRT`, `CEXP`) were changed to the equivalent generic function (viz `SQRT`, `EXP`).

The first six steps were done using the editor's search facility; the last was done by compiling and picking up errors. After the program compiled successfully, the cross-referenced listing was searched for (erroneous) `R*4` and

`C*8`, and the list of functions referenced was examined to look for similar errors. Note that the VAX compiler does *not* list `CMPLX`, `REAL` and `AIMAG` in its list of functions referenced; hence the need for great caution here.