THE APPLICATION OF TRANSPUTER ARRAYS TO NUMERICAL ANALYSIS AND COMPUTER-AIDED ENGINEERING

by

ROBERT ARTHUR MAXWELL ALLEN

A thesis
presented to the University of Manitoba
in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

Winnipeg, Manitoba, Canada
March 1988

ISBN 0-315-44199-2

THE APPLICATION OF TRANSPUTER ARRAYS TO NUMERICAL ANALYSIS AND COMPUTER-AIDED ENGINEERING

BY

ROBERT ARTHUR MAXWELL ALLEN

A thesis submitted to the Faculty of Graduate Studies of
the University of Manitoba in partial fulfillment of the requirements
of the degree of

DOCTOR OF PHILOSOPHY

To my parents

I hereby declare that I am the sole author of this thesis. I authorize the University of Manitoba to lend this thesis to other institutions or individuals for the purpose of scholarly research. I further authorize anyone to reproduce this thesis by photocopying or by other means, in total or in part, for the purpose of scholarly research.

Robert A.M. Allen

ABSTRACT

The objective of this dissertation is to explore ways of achieving high performance for numerical analysis and Computer-Aided Engineering (CAE) algorithms with minimal cost. The transputer microprocessor and its accompanying language (Occam), which provide a readily accessible means to perform Multiple Instruction/Multiple Data parallel processing, were found to ideally satisfy these goals.

The main category of algorithms studied deals with matrix and vector operations since these are so pervasive in the targeted fields. Monadic and dyadic vector operations are shown to be efficient on linear transputer arrays, while scalar product and dense matrix-vector multiplication algorithms are more suited to the shuffle-exchange network. A novel sparse matrix-vector multiplication algorithm is demonstrated for banded matrices.

Matrix solution methods studied include Gaussian elimination and the Polynomial Preconditioned Conjugate Gradient (PPCG) techniques. The Gaussian elimination algorithm is shown to achieve nearly the maximum theoretical efficiency on a linear array of transputers by virtue of communication and computation overlap. A novel implementation of the PPCG algorithm is described, and applied to the finite-difference technique.

As an example of the implementation of a complex CAE algorithm, the MANitoba Integrated Transputer/Occam Boundary element Accelerator (MANITOBA) is described. MANITOBA consists of a novel algorithm that tightly couples a parallelized version of the boundary element method (BEM) with a Gaussian elimination matrix solver on a linear transputer array. Fundamental theory of the BEM is presented, along with architectural and algorithmic details of the implementation. The performance of MANITOBA is compared to a similar serial algorithm running on a workstation.

ACKNOWLEDGEMENT

I would like to express my sincere appreciation to my advisor and friend Dr. A. Wexler. The trust he has placed in me throughout this research was most encouraging. I am also deeply indebted to my friends and colleges. In particular, Mr. N. Aitken, Mr. A. Ang, and Dr. J. Poltz are thanked for their advice on the theory and implementation of the boundary element algorithm. I would like to acknowledge the financial support of the Natural Sciences and Engineering Research Council of Canada, the Province of Manitoba, and the University of Manitoba. In addition, the financial and moral support of my wife and best friend Val was indispensable. Finally, I would like to thank my parents, whose faith and inspiration have given me the strength to persevere.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF PRINCIPAL SYMBOLS

Symbol	Meaning
	Chapter III
${T_s}^+$	Time for scalar addition.
${T_s}^-$	Time for scalar subtraction.
${T_s}^\times$	Time for scalar multiplication.
${T_s}^\div$	Time for scalar division.
${T_v}^+$	Time for vector addition.
${T_v}^-$	Time for vector subtraction.
${T_v}^\times$	Time for vector multiplication.
${T_v}^\div$	Time for vector division.
${T_v}^{sc}$	Time to scale vector.
${T_v}^=$	Time to assign vector.
${T_s}^X$	Time to transmit scalar.
${T_v}^X$	Time to transmit vector.
T_1	Uniprocessor execution time.
T_{N_p}	Multiprocessor execution time.
E_{N_p}	Efficiency of multiprocessor algorithm.
N_p	Number of processors.
P_i	Processor number.
N	Problem size.
P	Matrix row bandwidth.
Q	Matrix column bandwidth.
I	Nearest neighbour communication index.
	Chapter IV
$\underline{r}$	Residual vector.
$\mathbf{A}, \mathbf{S}$	System matrices.
$\underline{x}, \underline{\sigma}$	Linear system unknown vectors.
$\underline{b}$	Linear system source vector.
$\underline{h}$	Exact solution of linear system.
$\underline{d}$	Direction vector.

Symbol	Meaning
$F(\underline{x})$	Quadratic energy functional.
$g(\underline{x})$	Gradient of $F(\underline{x})$.
$E(\underline{x})$	Error functional.
α	Optimum line search constant for $\underline{x}$ and $\underline{r}$.
β	Optimum line search constant for $\underline{d}$.
$\mathbf{I}$	Identity matrix.
$\mathbf{M}$	Part of system matrix splitting.
$\mathbf{M}^{-1}$	Splitting of $\mathbf{A}$ chosen to be $(\mathrm{DIAGONAL}(\mathbf{A}))^{-1}$.
$\mathbf{N}$	Part of system matrix splitting.
$\mathbf{K}$	Preconditioning matrix.
$\mathbf{K}_z^{-1}$	Approximate inverse of system matrix.
$\mathbf{L}$	Lower triangular factor of $\mathbf{K}$.
ρ	Spectral radius.
z	PPCG accuracy parameter.

Chapter V

B_e	Exterior region boundary.
B_i	Interface boundary.
ϕ	Electric potential.
ϕ'	Normal derivative of ϕ.
ϵ_0	Free-space permittivity.
ϵ	Permittivity.
$\vec{r}$	Field point.
$\vec{r}_0$	Reference point.
$\vec{r}'$	Source point.
$\hat{n}$	Normal vector pointing outward from a region.
$G(\vec{r}\,\vert\,\vec{r}')$	Green function.
$G'(\vec{r}\,\vert\,\vec{r}')$	Normal derivative of Green function.
$\sigma(\vec{r})$	Charge density distribution.
F	Energy functional.
α	Lagrange interpolation function.
ξ	Parametric coordinate.
J	Jacobian.

CHAPTER I

INTRODUCTION

The goal of this dissertation is to implement high-performance accelerators for numerical analysis and Computer-Aided Engineering (CAE) algorithms in a microcomputer or workstation environment. Using such platforms for CAE is desirable because of their low cost and easy accessibility when compared to operating within a mainframe environment.

Moreover, software provided in a mainframe environment is constrained by the information bandwidth between the mainframe and the user's terminal, and as a result tends not to be user-friendly. The workstation environment, on the other hand, allows high-bandwidth communication to a user's screen.

The problem we address here is that, more often than not, the workstation lacks the power needed to execute CAE algorithms. One strategy to overcome this shortcoming is to use the workstation as a graphical design device with the heavy computation off-loaded onto a mainframe through a network. This best of both worlds approach achieves the desired results but requires the presence of a mainframe computer locally with subsequent high costs.

Another strategy, which we adopt here, is to seek ways of speeding up the workstation itself so that it can solve difficult problems in a reasonable amount of time. Again, because we are operating in a low-cost environment, it is imperative that the methods used be sufficiently inexpensive to justify their use. Naturally, these goals are somewhat conflicting, and any practical system produced must be a compromise between them.

The workstation can be accelerated in a number of ways. For example, time consuming operations such as multiplication and division could be augmented with special-purpose hardware. Indeed, most present day microprocessors have companion co-processors to perform such tasks. The problem with this approach is that one can only go so far before the cost becomes prohibitive because of the speedup in the support circuitry necessary to keep the fast computation unit busy.

In addition, fundamental physical limits are being reached in the speed at which conventional computer hardware can operate, so even the most optimistic advances in present-day circuit technology can only hope to provide marginal improvements. Even if some unforeseen new technology evolves, it is unlikely that it will be inexpensive enough to find use in desktop computers in a timely fashion.

A better way to achieve speedup is to exploit parallelism in the problems at hand. In the present context, the algorithms used in numerical analysis and CAE need to be dissected into pieces that can be executed in parallel on different processors.

The level at which an algorithm is partitioned is referred to as its *granularity* (Geist et al. [1987,238]). An algorithm is said to be *coarse-grained* if each processor is given a significant portion of the problem data to operate on in each computation step. For example, many of the algorithms presented in this dissertation are coarse-grained since each processor is given many columns of a matrix to operate on.

Conversely, we say an algorithm is *fine-grained* if each processor has only a small amount of the computation to perform in each step. An example is the distributed sum algorithm in Chapter 3, where each processor has only to perform a single multiplication and addition in each step of the algorithm.

While the possible partitionings of an algorithm depend intimately upon the algorithm itself, they also depends upon the architecture of the target parallel system. The two main factors are the power (size) of the processors in the system, and the provisions made for communication between them.

Parallel architectures can be broadly categorized into two main groups (Flynn [1972]); Multiple Instruction stream/Multiple Data stream (MIMD); and Single Instruction stream/Multiple Data stream (SIMD). In practice, these groupings support well defined types of parallelism (coarse and fine-grain respectively).

A MIMD machine (multiprocessor) consists of many autonomous computers, each executing a different program (multiple instruction stream) on data in its local memory. Typically, each computer communicates to other processors through either shared memory (bus based multiprocessor) or through point-to-point communication networks. Since every node is a computer in its own right, it is by necessity rather large. Therefore, it is not usual to see large numbers of processors in a system, and small networks of relatively powerful computational nodes are favoured. Because the processors are very powerful, systems of this type most naturally address coarse-grain parallelism.

A SIMD machine consists of many processors, each obeying the commands of a common master controller which interprets a single instruction stream. The processors operate in lockstep (performing identical instructions at identical times) upon data stored in local memory. This architecture tends to favour large numbers of very simple processors. Examples include the Connection Machine with 65,536 single-bit processors (Hillis [1985]), and the ICL DAP with 4096 single-bit processors (Hockney and Jesshope [1981,178-192]). Since there are so many processors, bus based communication through shared memory is impractical, and communication networks are used to share information.

Since it is a rare algorithm that can be partitioned into pieces that are not inter-dependent in some way, efficient and versatile interprocessor communication is a key requirement of a parallel system. Making the necessary data available to a processor at the proper time encompasses much of a partitioning strategy.

Obviously, if every processor was directly connected to every other processor, this would not be a problem, and all communication could take place in one timestep. However, engineering and cost constraints prevent this for networks of any appreciable size, so we are forced to limit the number of processors to which a given processor can be connected.

Since two processors that need to communicate may not be directly connected, strategies that can facilitate communication through intervening processors must be developed. As such, we are required to continually weigh the gains in parallelism against the communication cost introduced by a particular partitioning on a given communication network.

Many different network topologies have been devised in an attempt to address these issues (Haynes et al. [1982], Gottlieb and Schwartz [1982]). Most represent an attempt to minimize communication costs while maximizing connectivity, and are usually successful only on certain classes of problems.

The hypercube is perhaps the most versatile of the "exotic" interconnection networks. It has the appealing property that, in a 2^K processor hypercube, a processor is never any further than K communication steps from any other processor. This property comes at a the price of hardware complexity, however, since it requires that each processor have K communication links. This in turn translates into high system cost due to the large amount of wire routing required. The other networks step back from this ideal case, and sacrifice communication distance so that hardware requirements will be less stringent.

It can be readily seen that parallel processing is not a panacea. The issues are further complicated by the fact that most of the development of numerical and non-numerical algorithms has been geared toward implementation on scalar (non-parallel) processors. The conversion of scalar algorithms to parallel algorithms is not a straightforward process. In fact there is no way of predicting whether or not an efficient parallel implementation of a given algorithm even exists. The crux of the problem is to find the right algorithms for a given architecture.

Working in the parallel realm requires a total rethinking of one's approach to programming. It is no longer the case that every variable and procedure in a program is available for use by all other parts. One has to adapt the algorithm so that its parallel components have available (or can obtain through communication) those resources and data that they need to carry out their duties.

After consideration of the above issues, it was apparent that the transputer microprocessor (Barron et al. [1983], INMOS [1985a,1985b], Whitby-Strevens [1985], Homewood et al. [1987]) was very well suited as the basis of a hardware accelerator. The transputer and its companion language Occam (Taylor and Wilson [1982], INMOS [1983]) directly address many of the above-mentioned issues.

The transputer microprocessor is unique in providing direct support for parallel processing. As we will see in Chapter 2, it has been designed from the ground up for this purpose. In many respects it allows one to break the rule that MIMD processors be few in number by allowing the construction of a powerful computational node in minimal circuit board area. This in turn translates into cost effective processing power.

With four high-speed serial I/O channels, a transputer directly addresses communication issues. This allotment of channels allows construction of point-to-point communication networks such as a linear array, square array, shuffle-exchange, 16-node hypercube, and cube connected cycle (Hill [1986]). Thus one can make use of existing algorithms for these network topologies.

Conversely, it should be emphasized that while this work concentrates on transputer networks, the algorithms developed are not limited to them. Any MIMD architecture with point-to-point communication can provide a suitable platform.

The transputer's only advantage is that it provides these required features at low cost.

We begin in Chapter 2 by describing the transputer and Occam in some detail so that the intimate relationship between the two can be better appreciated. The special features that support parallel processing are described.

In Chapter 3, matrix and vector algorithms are discussed. Such algorithms form the core of numerical analysis and CAE and, as such, it is imperative that efficient ways of implementing them exist. Basic vector operations such as vector addition and subtraction are implemented on a linear array. A scalar product algorithm running on a shuffle-exchange network is also implemented.

Two matrix-vector multiplication algorithms are also presented. The first, for dense matrices, makes use of the shuffle-exchange network in a manner quite similar to the scalar-product algorithm to achieve high efficiencies. A novel algorithm for banded sparse matrices, which requires a linear array of processors, is also presented.

Chapter 4 discusses two matrix solution techniques. The first, a novel implementation of Gaussian elimination, is applied to dense matrix systems. This solver was designed in conjunction with the boundary element algorithm discussed in Chapter 5. The second algorithm is a novel implementation of the preconditioned conjugate gradient method for the solution of banded sparse matrices.

Finally, Chapter 5 presents a novel parallel system that combines the Gaussian elimination solver from Chapter 4 with a Boundary Element Method (BEM) matrix generator. The objective was to integrate the two algorithms so that they would execute cooperatively on a problem with a minimum of data movement, so that high algorithmic efficiencies could be achieved.

CHAPTER II

OCCAM AND THE TRANSPUTER

Occam's Razor: *"Entities should not be multiplied unnecessarily"*

— William of Occam [Ockham], Quodlibeta Septum, c. 1320.

This chapter describes the Occam programming language and the transputer microprocessor which were used as parallel processing tools for the work described in this dissertation. There were a number of reasons for this choice. Firstly, the transputer is designed to allow easy assembly of multiprocessor systems. Secondly, Occam allows the specification of parallel systems in a simple and concise manner. Also important is the fact that they allow inexpensive and powerful parallel processing systems to be constructed.

Together, Occam and and the transputer provide a means of implementing multiple instruction stream/multiple data stream (MIMD) parallel algorithms (Flynn [1972]). As such, many algorithms designed for multiprocessor systems will apply to them. However, some architectural features, such as the lack of shared memory, will invalidate others.

The version of Occam described here is more properly known as Occam 2. It supersedes the original version of the language which has become known as proto-Occam. Proto-Occam was intended only as a demonstration of the Occam parallel processing paradigm and lacked many features consistent with a useful programming language (a major omission being floating-point number support). Occam 2 improves on proto-Occam in many areas and is at least as useful for numerical programming as FORTRAN, but with the advantages that a strongly typed structured language provides.

While this chapter presents some examples of Occam programming, it is not meant as a tutorial. The purpose is to convey the main features and philosophies of the language. Should a tutorial be desired, see Pountain [1987], or Appendix A – which provides an in-depth programming example of a tutorial nature. It should be noted that since Occam is an evolving language, the examples presented here may not be syntactically correct according to future language specifications.

Because technology is fast moving, transputer hardware specifics will only be presented where absolutely necessary. Instead, the generic attributes of a typical transputer are described, with major emphasis on those features that directly support the Occam programming paradigm. Above all else, it is these features that make a transputer what it is.

2.1 Occam

Occam has its roots in the theory of Communicating Sequential Processes (CSP) pioneered by C.A.R. Hoare [1978,1985]. It borrows many of its concepts from that work, but differs from the formal CSP model where required for practical implementation. For example, CSP theory allows recursively defined parallel procedures. Since a hardware implementation of such a construct would be difficult or impossible to realize, such capability was excluded from the language.

Practicality is perhaps the overriding tenet of Occam. True to Occam's razor, it is kept as simple as possible to allow direct hardware implementation of its fundamental features. Thus, on the proper hardware (i.e. the transputer), Occam can execute with an efficiency approaching that of assembly language while still providing the benefits of a high level language. Moreover, the Occam model is simple enough so that such hardware can be inexpensive.

In keeping with its CSP heritage, Occam provides a tool for describing parallel systems which consist of sequentially coded sections (called processes) executing in parallel. These processes are completely separate from one another, except for the existence of communication channels between them. Shared memory is not part of the programming model, avoiding the memory bottlenecks and expense of that multiprocessing technique.

Occam provides a little bit old and a little bit new. A programmer is able to carry over sequential programming skills when writing the sequential parts of Occam programs, while having available parallel execution of the component sequential processes.

2.1.1 Occam Processes

Occam programs are constructed from three primitive processes.

1. The assignment process

   ```
   a := <expression>
   ```

   The effect is the same as in other programming languages. The variable **a** is given the value of **<expression>**.

2. The input process

   ```
   ChannelName ? a
   ```

   The value received on the communication channel **ChannelName** is assigned to variable **a**. The value input can be of any type.

3. The output process

   ```
   ChannelName ! <expression>
   ```

   The value of **<expression>** (which can be of any type) is output on the communication channel **ChannelName**. The output process can be treated as a way of assigning a value to a variable in another process. Since shared memory is not part of the Occam programming model, the only way of doing this is through a channel.

An Occam program is created by combining these primitive processes in a hierarchical fashion. The following sections describe the special features of the language which facilitate this process for the construction of both sequential and parallel programs.

2.1.2 SEQ Constructor

The simplest way to combine the primitive processes is with the SEQ constructor. It identifies a list of processes (primitive or complex) that are to be executed in a sequential fashion. Thus SEQ allows the specification of conventional sequential programs. As an example, consider the following Occam code fragment (text that follows a double-dash is a comment):

```
CHAN OF ANY FromKeyboard: -- CHAN OF ANY can transmit
CHAN OF ANY ToScreen:     --   any variable type.
BYTE Temp:
SEQ
  FromKeyboard ? Temp     -- Get keyboard data
  ToScreen ! Temp         -- Send it to the screen
```

The example displays a number of Occam syntactical and semantic conventions. The indentation of the two lines after the SEQ indicates that they are combined into a single non-primitive (complex) process by the SEQ. Thus, the effect of the SEQ (or any other constructor) is to combine the processes at the same indentation level under it into what is treated as a single process. This can be carried on indefinitely with the component processes themselves being defined in terms of SEQ.

The example also shows how identifiers used by a program are declared. Notice that channels are treated just like any other variable in the program, and must be explicitly declared before they are used. The three identifiers (at the same indentation level as the **SEQ**) are tied to the process created by the **SEQ** and are only valid within the scope of that process. In a hierarchy of processes, variables declared in the inner processes are not known outside of those levels (i.e. syntactically at smaller indentation levels).

In terms of Hoare's CSP theory, the **SEQ** constructor provides the first necessary condition for a CSP computation environment, namely, a way of defining conventional sequential processes (programs). The final step is to combine these processes into parallel programs composed of sequential processes executing in parallel with each other.

2.1.3 PAR Constructor

Consider the parallel processing system shown in Figure 2.1. We have two processes that are to execute in parallel with communication over a channel. The processes are totally separate except for communication over the channel. Most importantly, they run asynchronously from one another (except when they communicate), and do not share memory. In Occam, such systems may be specified using the **PAR** constructor.

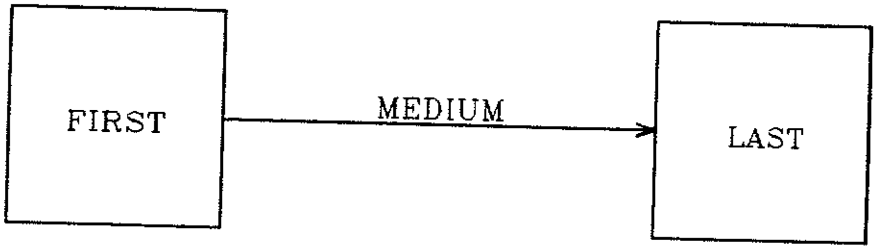

Figure 2.1: Example of a parallel system. This represents the basics of the Occam parallel processing paradigm.

Introducing some more Occam syntax, consider the following named procedures:

```
PROC First(CHAN OF ANY ToLast)
  INT A:                  -- Integer variable
  SEQ                     -- Construct sequential process
    A := 10               -- Assignment primitive
    ToLast ! A            -- Output on channel ToLast
:                         -- End of PROC
PROC Last(CHAN OF ANY FromFirst)
  INT B:
  SEQ
    FromFirst ? B         -- Input on channel FromFirst
    B := B * B
:
```

As in conventional languages, the two code sequences are given names by which they can be referred. Additionally the procedures (processes) can be given arguments which in this case happen to be channels. Using PAR the two processes may be combined into a single concurrent/parallel program.

```
CHAN OF ANY Medium: -- Declare inter-process Comm. CHAN
                    -- Ends of Medium are assigned to an
                    --  input and an output process.
PAR                 -- Denote parallel construct
  First(Medium)     -- Component processes of PAR
  Last(Medium)      --  execute in parallel.
```

2.1.4 ALT Constructor

It is often necessary to construct a process that services a number of channels. Consider for instance, the situation depicted in Figure 2.2, in which a process has to take messages from channels A, B, and C and multiplex them down channel D. A parallel construct cannot be used, since it is not possible to share channel D among three processes servicing the input channels separately.

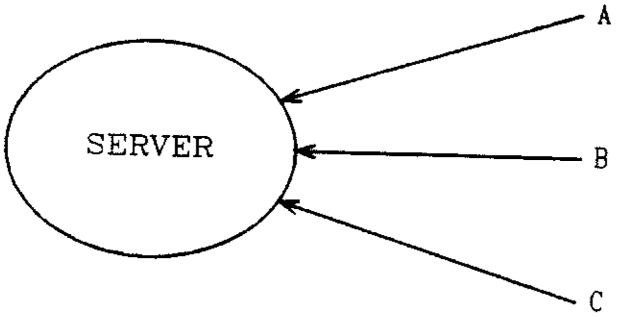

Figure 2.2: A sequential process servicing three inputs.

A possible solution would be to input from each of the channels in sequence, which will work if messages come often and regularly.

```
WHILE TRUE          -- Loop forever
  SEQ
    ChanA ? Msg
    ChanD ! Msg
    ChanB ? Msg
    ChanD ! Msg
    ChanC ? Msg
    ChanD ! Msg
```

The problem with this approach is that once an input statement is entered, the program will wait until it receives an input on that channel. If an input never comes (or comes at random times), the program will be deadlocked. Even though there may be messages ready on the other channels, they will never be serviced.

To handle such situations, the ALT constructor is provided. It provides a way of testing a number of input channels to see if they have messages pending. The first input channel to have a message on it is serviced. In the case of ties, the channels are serviced in an unspecified order. The following Occam code fragment shows how an ALT can be used to solve the present problem.

```
WHILE TRUE          -- Loop forever
  ALT
    ChanA ? Msg     -- Inputs if message present.
      ChanD ! Msg   -- Process executed if message present.
    ChanB ? Msg
      ChanD ! Msg
    ChanC ? Msg
      ChanD ! Msg
```

Once the ALT inputs from a channel, it terminates. It therefore must be set up in a loop if it is to repeatedly service all the channels. Note, that the clauses under the ALT are combined into a single process, and that each clause itself forms a process underneath it.

2.1.5 Characteristics of Occam Channels

We have seen that communication between parallel Occam processes takes place over channels. First of all, it should be emphasized that channel communication only makes sense between two processes that are executing in parallel. It is not possible to use channel communication between two processes that are components of the same sequential process. In any case, such communication wouldn't be necessary since processes that are components of a sequential process can pass information through variables in the normal way.

An Occam channel is a one way communication link between an input and an output process. Additionally, because (as we will see) channels are identified with fixed hardware resources, they cannot be shared among parallel components of a program. Once a channel is identified as a connection between two parallel components of a program, it can only be used between those two components and in only one direction until those components terminate. In terms of Occam syntax, this requirement is enforced via variable/channel scoping conventions.

Communication on channels can only take place when both the input and output processes on either end of the channel are ready to communicate. If either of them is not ready to enter into communication, the first must wait until the other is ready. Thus, channel communication serves to synchronize the two processes and the programmer does not need to explicitly manage this.

The characteristics of Occam channels, along with the various features of the Occam programming model were not chosen arbitrarily. A major consequence of these design decisions is that a simulation of an Occam program running on a single processor multitasking system will run exactly the same as it would with its component processes split up among a number of processors. This gives the program designer great freedom when partitioning a parallel program to run on a network of processors. The program is guaranteed to run identically no matter how the partitioning is done. The only restriction is that the resources of the target hardware must be accommodated. One cannot place a process that requires five I/O channels onto a processor that has only four available.

2.1.6 Occam Time

In most languages the concept of time is a superfluous one. Occam makes it an integral part of the language by providing each process with access to a special channel of type TIMER. Channels of this type return the value of a counter that is incremented at regular intervals (ticks). For example, a computation can be timed with the following code fragment:

```
VAL TicksPerSecond IS 1024:  -- Hardware dependent value
TIMER Time:                  -- TIMER channel
INT StartTime,EndTime,TotalTime:
SEQ
  Time ? StartTime
  ... Computation Process
  Time ? EndTime
  TotalTime := (EndTime - StartTime)/TicksPerSecond
```

Because Occam processes are independent, the Occam sense of time is necessarily local to each process. While the above code will yield identical times on two identical processors within a system, there is no guarantee that StartTime and EndTime will be given the same values on the two processors. Only the difference between them will be the same on each processor.

The timer can also be used to introduce timed delays. For example, the process

```
VAL DelayTicks IS 40:
TIMER Time:
INT CurrentTime:
SEQ
  Time ? CurrentTime
  Time ? AFTER (CurrentTime + DelayTicks)
```

will delay until the timer counter is greater than the sum of **CurrentTime** and **DelayTicks** (for simplicity, the case where this sum exceeds the maximum integer representation is ignored).

The latter usage of the Occam timer is important when channel communication is unpredictable (the result of random external events, for example). In these cases it allows, through the **ALT** statement, a way to test for input from a channel or a number of channels and escape to do other work if no input is present. Consider the following code fragment:

```
VAL TimeOutTicks IS 40:
TIMER Time:
INT CurrentTime,TimeOutTime:
INT RandomInput
CHAN OF ANY Random:
SEQ
  Time ? CurrentTime
  TimeOutTime := CurrentTime + TimeOutTicks
  ALT
    Random ? RandomInput
      ... service random input
    Time ? AFTER TimeOutTime
      ... service time out
  ... other processing duties
```

If no message on channel **Random** is received before 40 timer ticks have elapsed, the last clause of the **ALT** is entered, allowing other processing duties to be performed. Without the last clause, the **ALT** would wait until a message was received on **Random**. This mechanism is important as it allows one to have asynchronous interprocess communication.

2.2 The Transputer

We have seen that Occam provides a simple and concise (if not powerful) way to express parallel systems. By itself it might be a useful tool. What makes Occam special is the existence of the transputer microprocessor. The transputer is designed to implement in hardware what the language Occam describes in software. Together they provide a potent tool for the implementation of parallel algorithms.

2.2.1 Architecture

This section describes the architecture of the transputer microprocessor. The features described are that of a generic transputer and represent the minimum hardware requirements that a microprocessor must have to be called a transputer.

As can be seen in Figure 2.3, the main features of a transputer are:

1. CPU: Currently a reduced instruction set (RISC) design. The compactness of the CPU allows other essential features to be integrated on the same die.
2. RAM: On-board static RAM that can be accessed at least three times faster than external memory. Current transputers possess 2K or 4K bytes. This is general system RAM, and is not used as cache.
3. Communication Links: Current transputers possess four INMOS bidirectional communication links which implement a 10 or 20 MHz Occam channel in each direction (1 or 2 megabytes/second).
4. Programmable memory interface: Interfaces the transputer to external memory with the minimum of external interface logic by providing all necessary refresh, timing, and control signals.
5. Special purpose function units: architectural extensions that allow transputers to be tailored to specific applications.

Figure 2.4 depicts the simplest possible view of a transputer – a black-box with four (or more) communication links (an input channel and an output channel are coupled together in pairs to form a link). As such, the transputer can realize any

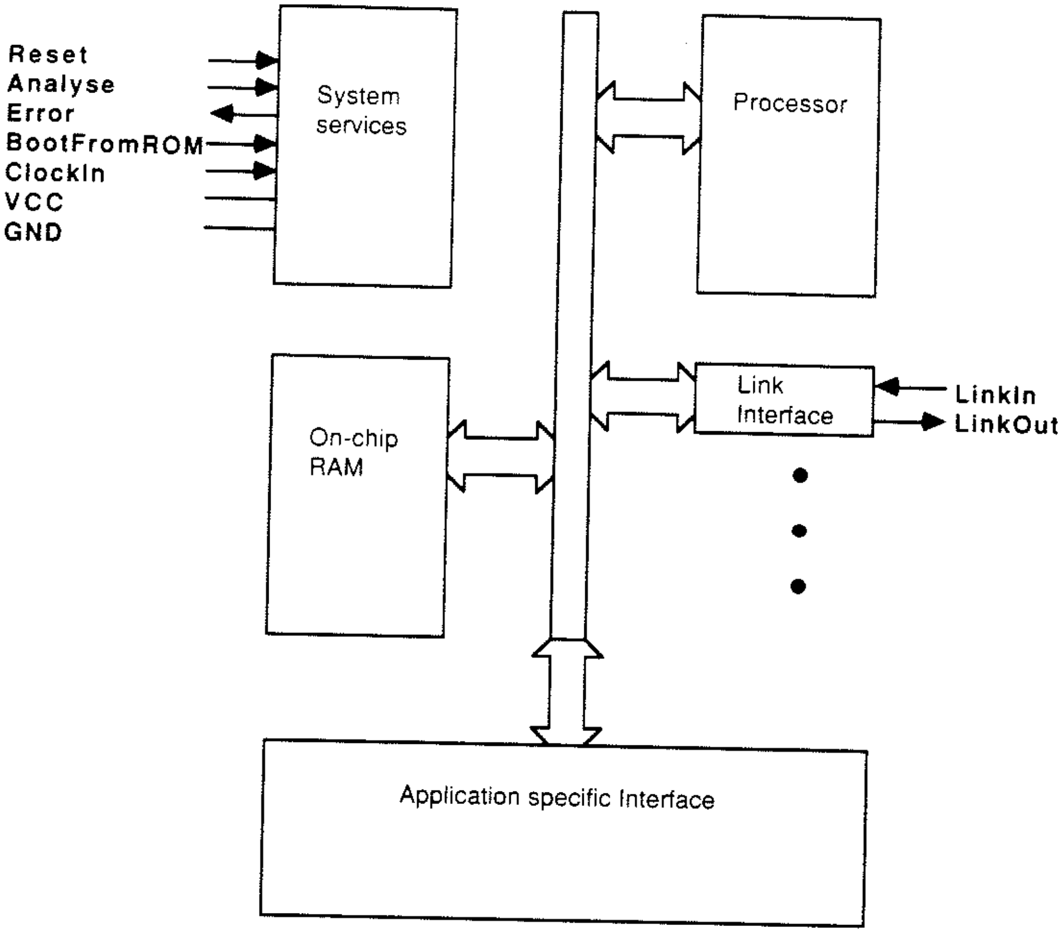

Figure 2.3: The transputer architecture. Copyright INMOS Corporation. Reproduced with permission.

network which requires four or fewer communication channels per computational node. In this work, they are used in linear array and shuffle-exchange configurations. Other possible networks include the square mesh and the toroidal mesh, as well as various multistage networks and the cube connected cycle (Hill [1986]). This kind of flexibility is important as it allows one to tailor the network topology to the data flow properties of the target algorithm.

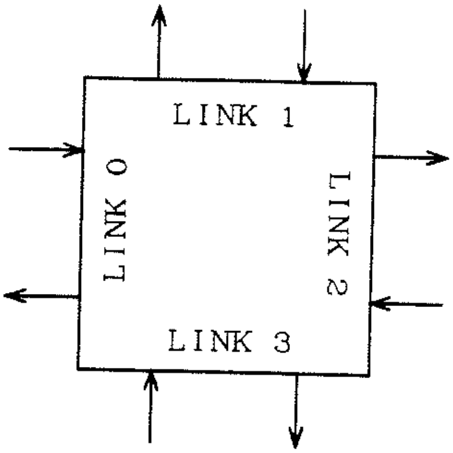

Figure 2.4: Transputer communication model.

The transputer definition allows for special-purpose hardware over and above what is pictured. For example, in one version of the transputer (the T-800), an on-board floating point unit is added (Homewood et al. [1987]), while in another a disk-controller is featured (Moore [1986]). Each special purpose transputer then plays a particular role in a given parallel system, with the links serving to tie the entire system together.

With these features, the transputer is really a microcomputer on a chip, possessing built in CPU, communication links, and RAM. In fact, it is capable of functioning with only external power and clock supplied.

The transputer's architecture is designed to support parallel processing. As such, the transputer lacks features desirable in single processor systems that may compromise its applicability to parallel processing (e.g. memory management). The success of this approach can be measured in a number of ways. We will see that the transputer has been designed to minimize cost, communication complexity, and board area - making it ideally suited for constructing large networks.

2.2.2 Support for Occam

The very reason for the transputer's existence is to provide hardware support for the Occam parallel processing model. Perhaps the most visible and important of these features are the on-chip communication channels. They provide the means by which Occam processes can communicate, and have characteristics entirely compatible with the definition of Occam channels in Section 2.1.5. It is important to realize that these are point to point communication channels (as is dictated by the Occam channel model), and there can only be one transputer at either end of a channel. It is not possible to connect one channel to many transputers in a broadcasting type of topology.

The point to point communication architecture that transputers implement has a number of advantages over conventional bus-based systems. Firstly, there is no contention for communication resources as might be found on a bus shared among many processors. Also, as the system grows, total communication bandwidth grows with it. This assumes that the communication requirements of the algorithms to be implemented are predominantly local.

To provide for seamless distribution of Occam processes over a network of processors, the transputer CPU possesses a multitasking kernel in hardware. It maintains two queues of Occam processes (low and high priority) which it timeslices automatically. If a process is waiting to communicate, it is descheduled until its companion process is ready to communicate with it. In this way, CPU cycles are not wasted in busy-wait (polling) loops and communication is carried out with the maximum possible efficiency.

In accordance with the Occam model, these parallel Occam processes communicate via channels. Processes on the same chip communicate via internal virtual channels, while processes on different chips use the external hardware channels. Since channel communication synchronizes communicating processes, the transputer is able to execute any number of them as if each resided on its own processor. This allows a given Occam program (essentially a collection of communicating processes) to be partitioned on a network as one desires (as long as enough hardware channels are available to support the connectivity of the partitioning).

To handle Occam's sense of time (see Section 2.1.6), each transputer incorporates a timer. In current transputer implementations, the high-priority process timer has greater resolution than the timer for low-priority processes (under the assumption that high-priority processes are more time critical). Occam sees the timer as a special channel which is available to all processes within a given transputer (normal channels can have only one originating and terminating process).

Finally, the transputer's CPU instruction set is designed to optimally implement the various features of the Occam language. We have already seen that the CPU is able to directly implement a `PAR` Occam construct. It also has instructions for the installation of processes on execution queues, the `ALT` operation, as well as instructions to perform channel input and output.

2.2.3 Constructing Large Transputer Networks

Since the intention of the transputer is to provide a means of doing parallel computation, it is important that such systems be easy to build. A number of features of the transputer make the system engineer's job easier.

The first feature is that the transputer does not require the distribution of a high-frequency clock. Conventional microprocessors require that an external clock be supplied at their internal clocking rate, which could be as high as 20-30 MHz. Such high frequency signals cause much grief for board designers, especially if they are required to travel large distances on the circuit board.

Transputers, on the other hand, only require an external clock of 5 MHz no matter what the internal device clock-rate. This standard input clock is then used to derive a higher internal clock speed. The system designer need only distribute this relatively low speed clock throughout a circuit board.

The engineering characteristics of transputer communication channels also support the construction of large networks. Transputer channels are implemented as bidirectional serial communication lines as shown in Figure 2.5, and provide one Occam channel in either direction.

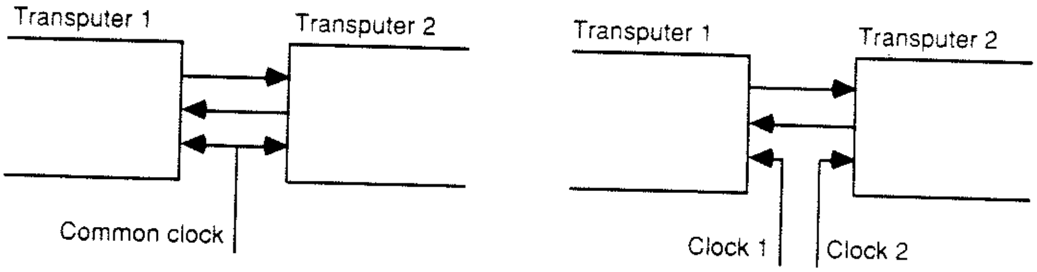

Figure 2.5: Transputer communication link. Copyright INMOS Corporation. Reproduced with permission.

Such an architecture has a number of important characteristics. For one, it is much less costly to distribute a three-line bus than it would be to distribute wide parallel buses. Since transputers may number in the hundreds in a given parallel computation system, reducing cost is paramount.

Besides the obvious ease of routing 3 lines, one does not pay a capacitive load penalty as more transputers are added to the system. In a bus-based system, the whole system must slow down as the bus is lengthened to accommodate more processors. In a transputer system, this penalty is more or less constant for each transputer no matter what the size of the system. Moreover, link communication is insensitive to clock phase between the sending and receiving transputer. All that is necessary is to maintain the clock frequency within fairly loose tolerances, allowing the system designer flexibility in routing clock signals. As Figure 2.5 shows, the input transputer clocks need not come from the same source.

When building systems with a large number of processors, it is desirable to pack as many processors on a single board as possible. The usual limiting factor is the amount of support circuitry that must be placed around a microprocessor to provide interfaces to memory and peripherals. For the most part INMOS channels can interface to peripherals with no external circuitry, offering great savings in board area. Furthermore, the transputer's programmable memory interface virtually eliminates the need for external memory interfacing circuitry by generating all timing, control, and refresh signals for a variety of external memory types.

With these space-saving features, it is possible to construct a transputer node with two Megabytes of RAM in as little as 51 cm^2 of board area (Electronics [1988]). If an application did not require much RAM, the node size could be reduced still further by utilizing only the internal transputer RAM and running the transputer without any external support circuitry save clock generation (which could be shared among all transputers on a single board).

2.2.4 Hardware Considerations for Occam Programming

While the transputer hardware is designed to support the Occam programming model, it has architectural features that enhance program performance which are not required by the model. This section outlines two of the most important ones. Further notes about performance issues can be found in Atkin [1987].

Using On-Board RAM

Unlike microprocessors with on-board cache, the on-board RAM of a transputer is part of its normal address space. Its advantage over external memory is that it can be accessed at least three times faster. Since it is normal memory, it can be used to store both code and/or data.

Since the transputer encodes four instructions within a single word, it fetches instructions faster than it does data words. It is therefore desirable to place critical data into the on-chip RAM, to speed access to it. Indeed, the current Occam compiler places data on chip in preference to code. If one has large vectors, it is impossible to place code on-chip in preference to data because the vectors will take up all of the on-chip RAM. It is possible that this restriction may be removed in the future, however.

Making use of internal RAM for data storage is (at the moment) an implicit rather than an explicit process. The programmer is given details of how variables defined in the Occam source are placed into memory. By arranging the source code declarations properly, the desired critical variables can be placed into internal RAM.

Channel Communication

Each link on a transputer possesses an independent controller that has direct memory access (DMA) to the transputer's memory space. Once a transfer is initiated on a link, the controller takes over responsibility for the transaction and frees the processor to do other tasks. Thus each link transfer is a direct memory to memory transfer between the processors at either end of the link.

An important feature of the link controller, called *slice communication*, is the ability to transfer blocks of memory in addition to single byte and word transfers. With processor overhead equivalent to a single word transfer, the CPU is able to initiate a multi-word transfer that is totally controlled by the link hardware. The CPU is then able to perform other tasks while the transfer is taking place. Moreover, slice communication can be performed on three links simultaneously without CPU interaction, and on all four links with only a small CPU degradation because of memory-bus contention.

To use this feature of the transputer, Occam provides a special syntax on the output and input process statements. For example, an integer array of length 1000 could be sent down a channel by the output process

```
OutChannel !  [IntArray FROM 0 FOR 1000]
```

and received by the input process

```
InChannel ?  [IntArray FROM 0 FOR 1000]
```

Slice communication allows the transfer of large amounts of information between processors while computation is being performed at full or nearly full speed. It thus has the potential to provide nearly transparent loading/unloading of data for the next computation step, while the current step is proceeding. This could be especially important in applications such as computer vision, where the next image could be loaded in while the previous one is being analyzed.

CHAPTER III

MATRIX AND VECTOR OPERATIONS

Vector and matrix operations are almost ubiquitous when it comes to algorithms in numerical analysis. For this reason it is particularly important that efficient ways of handling them exist. This chapter details ways of implementing important representatives of these algorithms on transputer arrays.

The most important aspect of the partitioning strategy is the way vector and matrix storage is distributed among the processors. This not only dictates what computations a processor can perform, but what communication is needed. The important thing is to choose a partitioning that minimizes global communication as much as possible.

The algorithms presented in this chapter view an array of processors with local memory as a distributed storage and computation network. The storage of each vector and matrix is divided up more or less equally between the processors, allowing each processor to work on its portion independent of the others.

For the sake of simplicity, and in keeping with the goals of this research, the implementations are restricted to linear arrays of transputers or simple shuffle-exchange (SE) graphs (Stone [1971]). The linear array accommodates algorithms with nearest neighbour communication requirements while the SE network is particularly efficient for performing global accumulations.

The beauty of these two networks is that the transputer is capable of supporting both of them simultaneously, which allows a much greater range of algorithms to be implemented efficiently. Unfortunately, restrictions in the hardware used for this dissertation prevented the simultaneous use of the two networks. This would have been desirable for the conjugate gradient algorithm presented in Chapter 4.

A final note is in order. The reader will notice that while this chapter contains estimates of algorithm performance, it lacks results from actual implementation. This was done for a number of reasons, the first being that experience has shown the estimates to be quite accurate. The major reason, however, is that most of the algorithms described here are used for the preconditioned conjugate gradient algorithm in Chapter 4. As actual results are given for that implementation, the merits of the algorithms of this chapter can be inferred from there.

3.1 Estimating Algorithm Execution Times

For the algorithms considered here, it is possible to accurately estimate their execution times on transputer or MIMD networks by counting the sequential vector and scalar operations and multiplying by the empirically derived times for these operations. When identical operations are performed in parallel, they count only once toward the final tally (i.e. they are treated as one sequential operation).

Table 3.1 shows timing results for some "primitive" operations on the T-414 and T-800 (a T-414 with floating-point support) transputer microprocessors. Since the T-800 was unavailable, timings for it had to be estimated as indicated in the table. While the list is not exhaustive, it is sufficient to give a fairly accurate estimate of execution times for matrix-vector algorithms.

Some things to note about the table:

1. the timings were performed for 32-bit floating-point operations and 32-bit word transfers on a 15 MHz T-414;
2. vector timings include the overhead of looping and array subscripting, and are given as the execution time per vector component;

Table 3.1: Timings for primitive operations on T-414 and T-800.

Symbol	Operation	T-414 (μs)	T-800 (μs)
T_s^+	Scalar Addition	20.3	2.03‡
T_s^-	Scalar Subtraction	19.8	1.98‡
$T_s^\times$	Scalar Multiplication	15.4	1.54‡
$T_s^\div$	Scalar Division	18.5	1.85‡
T_v^+*	Vector Addition	26.0	2.60‡
T_v^-*	Vector Subtraction	26.6	2.66‡
$T_v^\times$*	Vector Multiplication	21.6	2.16‡
$T_v^\div$*	Vector Division	24.3	2.43‡
T_v^{sc}*	Scale Vector by Constant	19.8	1.98‡
$T_v^=$*	Vector Assignment	4.8	4.8
T_s^X†	Transmit Scalar Word	10.8	10.8
T_v^X†*	Transmit Vector Word	8.4	8.4

* (per vector component)
† (using 10 MHz links)
‡ (estimated to be 0.1 of the T-414 value)

3. Vector communication utilizes the “slicing” feature of the transputer hardware discussed in Section 2.2.4; and
4. use of on-chip transputer RAM (which is a minimum of three times faster than external RAM) was avoided as it would have artificially decreased the measured timings (thus the numbers presented contain the penalty for off-chip RAM access).

By using actual execution times as weights for the operation counts it is possible to provide clock time estimates for the algorithms of this and the next chapter. This is much more useful than the operation counts alone, as it allows comparison to similar algorithms running on other hardware.

The metric used to characterize the algorithms in this dissertation is speedup efficiency (E_{N_p}). Letting the time taken for 1 and N_p processors to solve a problem be T_1 and T_{N_p} respectively, the speedup efficiency is defined as

$$E_{N_p} = 100\frac{(T_1/N_p)}{T_{N_p}} \tag{3.1}$$

and represents the percentage of ideal speedup that is attained. In an ideal world E_{N_p} would be 100%, but in practice it is less than 100% due to communication and synchronization overhead.

3.2 Vector Operations

We consider two classes of vector operations in this section. The first class includes monadic and dyadic vector operations that yield a vector result. The second is the scalar product, which combines two vectors to produce a scalar result.

These two classes have very different requirements for parallel execution because of their communication requirements. The monadic and dyadic vector operations are such that they require no inter-processor communication when properly partitioned. However, data input and result output operations are still necessary, so we will implement them on a linear array of processors as shown in Figure 3.1.

The scalar product, on the other hand, requires a great deal of communication, since partial-sums from all of the processors in the network must be brought together for a final summation. While the linear array is fairly efficient for the operation when the network contains a small number of processors, the efficiency would degrade for larger networks. For this reason, Section 3.2.2 introduces the shuffle-exchange network for this operation.

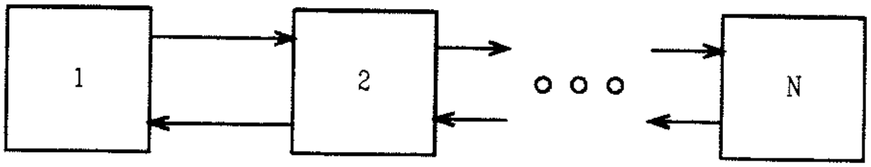

Figure 3.1: A linear array of processors.

3.2.1 Vector Combination

The dyadic vector combination operations considered include addition, subtraction, multiplication, and division. Monadic operations include vector assignment and scaling. In all these cases, vectors are combined or operated on in a component-by-component fashion, giving a vector result. The most notable property of these component operations is that they are independent from one another.

Since each component operation is independent, no inter-processor communication is required for the algorithm if a given processor contains all the information necessary to carry out that operation. In the case of the dyadic operations we require that a processor contain corresponding components of the two vectors that are to be combined. A consequence of this is that we are free to choose any partitioning for the vectors as long as we agree to partition each of them identically.

We are ignoring the case where different processors are operating on disjoint sets of vectors. Each processor could store all the components of the vectors it is operating on and process them independently from the other processors. This is a more macroscopic (larger grain) type of algorithm, however. While it gives an average speedup, it does nothing to speed up an individual operation such as vector addition.

Figure 3.2 shows the vector partitioning that was used in this work. This is not the only imaginable partitioning, merely the most orderly (a special application might require a different partitioning). Each vector is divided into a number of contiguous segments, with each segment stored on a processor within a linear array. The segments are assigned to the processors such that segment *i* is stored in processor *i*.

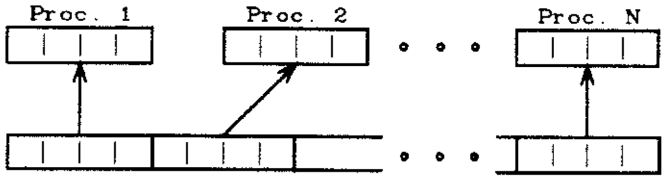

Figure 3.2: Partitioning of vectors for parallel computation.

Given this partitioning, Occam allows us to express the vector operations quite simply. Consider the Occam procedure **VectorAdd**:

```
PROC VectorAdd(     []REAL32 A,          -- result
               VAL []REAL32 B,C,         -- operands
               VAL INT NumComponents)
  SEQ I = 0 FOR NumComponents
    A[I] := B[I] + C[I]
:
```

Each processor in the network would contain a local copy of this procedure along with the storage for the segments from the vectors involved. A parallel system of four processors could then be denoted (ignoring declarations of the variables):

```
PAR
  VectorAdd(A,B,C,NumComponents)
  VectorAdd(A,B,C,NumComponents)
  VectorAdd(A,B,C,NumComponents)
  VectorAdd(A,B,C,NumComponents)
```

or more succinctly (where **NumProcessors** $= 4$)

```
PAR I = 0 FOR NumProcessors
  VectorAdd(A,B,C,NumComponents)
```

Given knowledge about how many components of the vector it is storing, each processor (each component of the PAR) can operate with virtually 100% efficiency.

3.2.2 Scalar Product

The scalar product between two vectors presents a very different problem. Given that each processor can compute a partial sum of the scalar product, we are left with a single number in each processor that must be summed with all the others. What remains then, is the problem of communicating these numbers between the processors so that a final sum can be computed.

First, consider N_p processors arranged in a linear array. The sum could be obtained in $(N_p - 1)$ communication steps and $(N_p - 1)$ addition steps simply by passing a partial sum from processor to processor — leaving the final answer in the last processor in the chain. If the result is needed in all the processors, an additional $(N_p - 1)$ communication steps are required to distribute the final result, giving a total of $2(N_p - 1)$ communication steps.

This simplistic approach can be improved upon by recognizing that the accumulation of the partial sum is independent of the order in which it is done. Instead of forming the sum from one end of the array to the other, we can have two parallel accumulations proceeding from each end to the middle of the array. With this scheme, the scalar product can be calculated in $\frac{1}{2}N_p$ computation steps and $(N_p - 1)$ communication steps (including distribution of the final sum to all processors).

In both these cases, we are constrained by the linear array, and are forced to compute the scalar product in almost a sequential fashion, with considerable communications overhead. As the number of processors increases the operation can become quite costly. This bottle-neck can be overcome using the shuffle-exchange (SE) network shown in Figure 3.3.

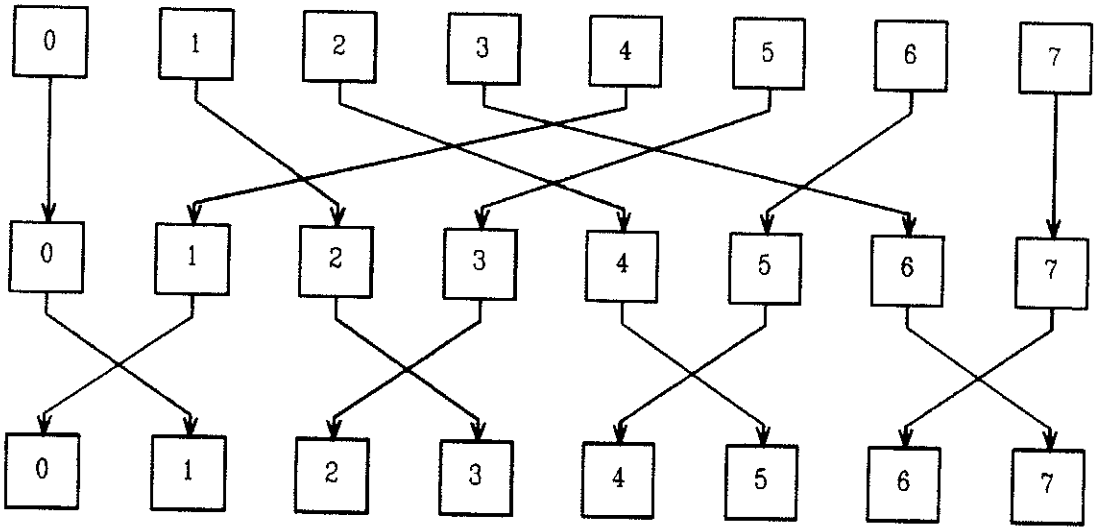

Figure 3.3: Time sequence of shuffle-exchange operations. Each row of processors represents the *same* set of processors at a subsequent step in the communication sequence. The lines represent the interconnections within the set of processors.

The SE network was first popularized by Stone[1971]. He demonstrated its applicability to a wide variety of problems including the Fast Fourier Transform (FFT) and matrix transposition. Lang and Stone[1976] showed its usefulness as a permutation network. Of interest here is its ability to sum N_p numbers in $\mathcal{O}(\log_2 N_p)$ steps.

Formally, the SE topology is defined for networks of $N_p = 2^K$ processors (K an integer) which we number 0 to $(2^K - 1)$. It combines the so called perfect shuffle interconnection with a pair-wise exchange to form a simple, yet powerful interconnection network. The perfect shuffle connects processor P_i to a processor whose index is given by the following rule (a permutation mapping):

$$P_i \longrightarrow \begin{cases} P_{2i}, & 0 \le i \le N_p/2 - 1 \text{ (lower half)}, \\ P_{2i+1-N_p}, & N_p/2 \le i \le N_p - 1 \text{ (upper half)}. \end{cases} \tag{3.2}$$

The effect of this operation can be seen in Figure 3.3. Messages from the lower and upper halves of the array are shuffled through the array in an alternating fashion, much like the two halves of a deck of playing cards would be when perfectly shuffled.

The shuffle operation is followed by a bi-directional exchange between adjacent pairs of processors according to the rule:

$$P_{2i} \longleftrightarrow P_{2i+1} \quad 0 \le i \le N_p/2 - 1. \tag{3.3}$$

Keeping in mind that these connections involve the same set of processors, Figure 3.3 can be collapsed to give a truer picture of the SE interconnections as shown in Figure 3.4.

Given the SE topology, we can now describe an algorithm for summing a set of N_p numbers (with one number located in each processor). Let each processor contain one of the numbers to be summed in variable `A`. Further, let there be a temporary location `B` in each processor. The following algorithm, which is performed identically by each processor, will sum the N_p values of `A` together:

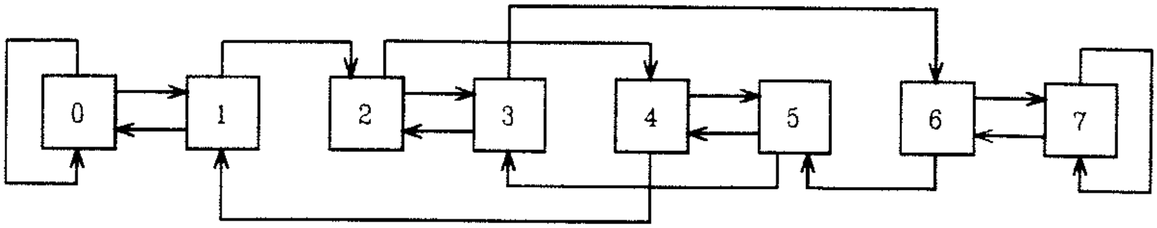

Figure 3.4: The shuffle-exchange interconnection network.

1. send A out on shuffle-out port and input into B on shuffle-in port (Shuffle operation);
2. output B on exchange-out port and input value into A on exchange-in port (Exchange operation);
3. sum B into A (Add operation); and
4. loop back to step 1.

If there are $N_p = 2^K$ processors, then after $\log_2(N_p) = K$ iterations each processor will contain the sum of all the numbers in variable A. Table 3.2 shows a detailed listing of these operations for an eight processor network adding the numbers 0 to 7 inclusive.

This scheme has two advantages over the linear interconnection network:

1. the operation is completed in logarithmic time ($2\log(N_p)$ communication steps and $\log(N_p)$ computation steps to sum N_p numbers); and
2. each processor ends up with the final sum, so there is no need to explicitly distribute it.

Additional insight into the workings of the SE algorithm presented here can be gleaned by following the path of a number from its starting point (the so-called

Table 3.2: Shuffle-exchange vector sum. Each row shows the values contained in A and B *after* the operation in the first column is performed (S = Shuffle, E = Exchange, and A = Add).

OP	PROCESSOR															
	0		1		2		3		4		5		6		7	
	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B
	0	*	1	*	2	*	3	*	4	*	5	*	6	*	7	*
S	0	0	1	4	2	1	3	5	4	2	5	6	6	3	7	7
E	4	0	0	4	5	1	1	5	6	2	2	6	7	3	3	7
A	4	*	4	*	6	*	6	*	8	*	8	*	10	*	10	*
S	4	4	4	8	6	4	6	8	8	6	8	10	10	6	10	10
E	8	4	4	8	8	4	4	8	10	6	6	10	10	6	6	10
A	12	*	12	*	12	*	12	*	16	*	16	*	16	*	16	*
S	12	12	12	16	12	12	12	16	16	12	16	16	16	12	12	16
E	16	12	12	16	16	12	12	16	16	12	12	16	16	12	12	16
A	28	*	28	*	28	*	28	*	28	*	28	*	28	*	28	*

* (don't care)

radioactive tracer technique). By marking the processors touched by that number and any numbers affected by it (directly or indirectly) as they cycle through the array, one can see the path fan-out in a binary tree like fashion. At the end of $\log_2(N_p)$ cycles all processors in the array will have received the number or one of its descendants. Since a number and its descendants never cross paths, they always contribute to an independent sum. Moreover, because this happens from all the processors, N_p independent sums are accumulated.

We are now in a position to compare the estimated execution times for the two scalar product/distributed addition algorithms outlined here. Since the time needed to calculate the initial partial sum in each processor is the same for all the algorithms, it is not included in the times given. Figure 3.5 shows the results for

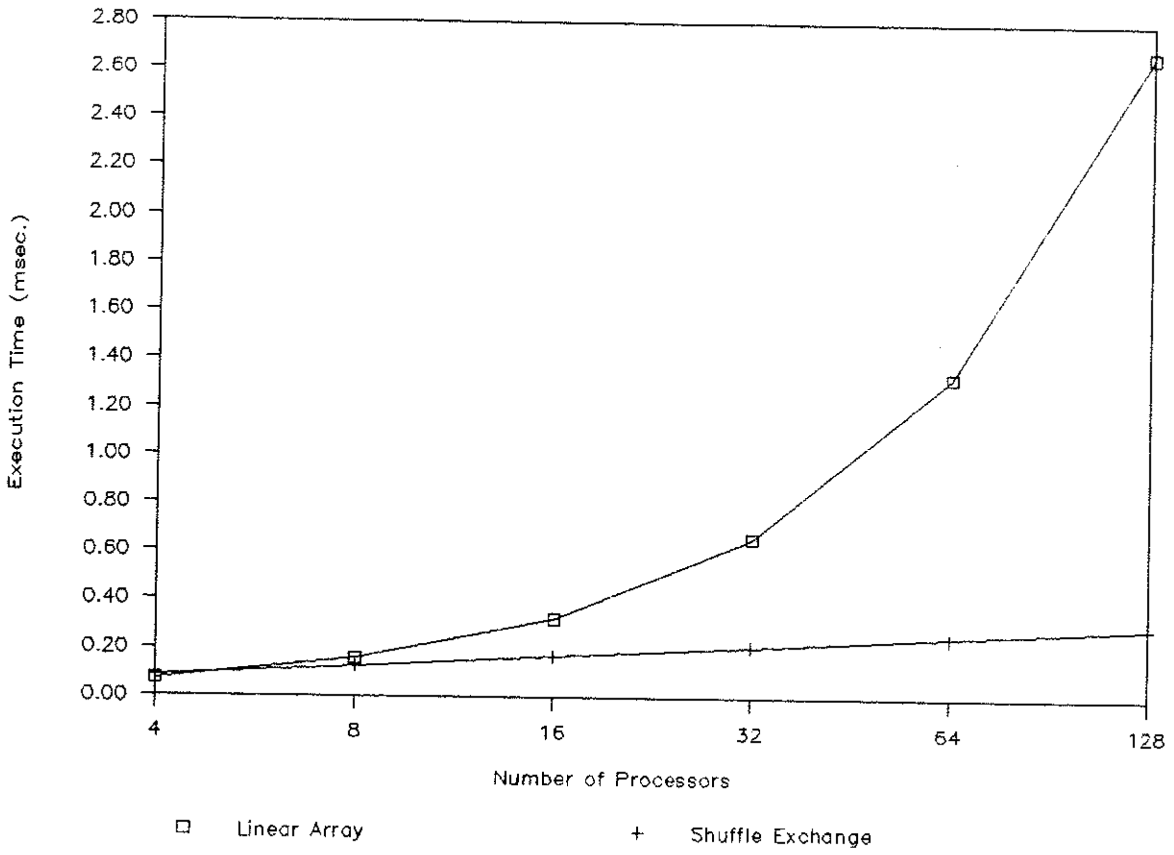

Figure 3.5: Comparison of the scalar-product (distributed addition) algorithms.

various network sizes (constrained by the requirements of the SE network). In all cases of interest, the SE network outperforms the linear array.

Thus the SE network is very useful for combining objects that are distributed amongst a collection of processors. For small numbers of processors, the gains are not great, but owing to the logarithmic growth in costs, the gains are very significant for larger numbers of processors.

This example also serves to illustrate the ability of the transputer/Occam programming model to address both coarse-grain and fine-grain parallelism. The distributed addition algorithm is essentially a fine-grain algorithm, since very little computational work is done at each step. In the next section, we will see the same

SE network used for a coarse-grain algorithm (dense matrix-vector multiplication). This property allows great flexibility when designing algorithms.

An Occam example in Appendix A shows how a SE network could be implemented on a transputer array. The code demonstrates the distinct advantage of the SE network on transputer arrays, namely, ease of implementation. A comparison of the code to implement the bi-directional sum on a linear array to the SE network shows the code for the SE to be simple and elegant, requiring no logic for its execution. The linear array implementation, on the other hand, requires considerable logic to handle special cases in communication, and is generally "messy" in comparison.

3.3 Matrix-Vector Multiplication

Matrix-vector multiplication represents a step up in complexity from the vector algorithms described in this chapter. The biggest problem arises from the interaction between storage partitioning of the matrix and the communication requirements that are dictated by that partitioning.

An algorithm for dense matrix-vector multiplication is presented first. It is shown that an efficient algorithm is possible using a shuffle-exchange network. As for the scalar product, the SE network's ability to sum N_p objects in $\log_2 N_p$ steps is the important factor. The only difference is that instead of summing numbers, we are summing vectors.

Next, an algorithm for sparse matrix-vector multiplication is presented. In this case, the SE network did not offer a viable solution. Instead, it was necessary to restrict the application of the algorithm to banded matrices, which allows an efficient implementation on a linear array of processors. This algorithm forms the kernel of the conjugate gradient algorithm described in Chapter 4.

3.3.1 Dense Matrices

Unlike the vector operations, partitioning of matrix operations is not straightforward. To extract any kind of parallelism at all, it would seem necessary to distribute the storage of the vector and matrix over the array of processors. Figures 3.6 and 3.7 illustrate two ways of performing such a partitioning.

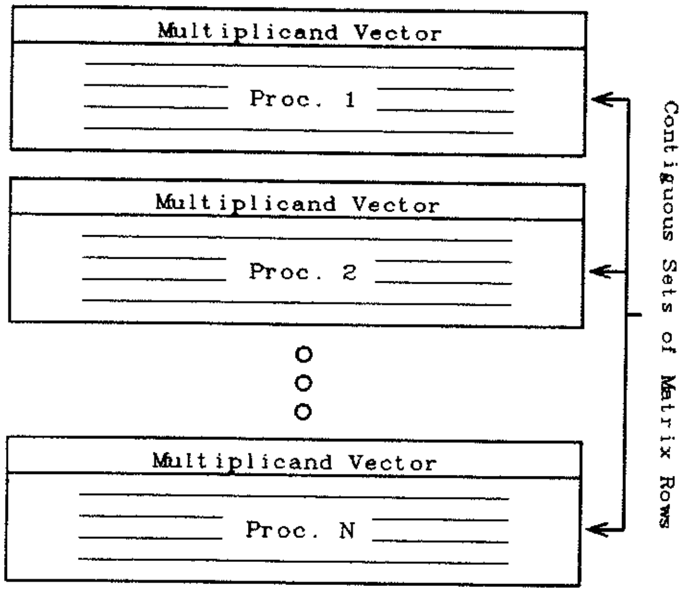

Figure 3.6: Row-wise matrix and vector partitioning. The entire vector multiplicand must be present in each processor.

Each of these partitionings has some advantages and disadvantages. With the row-wise scheme, each processor produces a fully summed segment of the resultant vector (of length equal to the number of rows that a processor stores). If the resultant vector is to be used in another multiplication, all of the segments must be communicated to all of the other processors, resulting in considerable communications overhead.

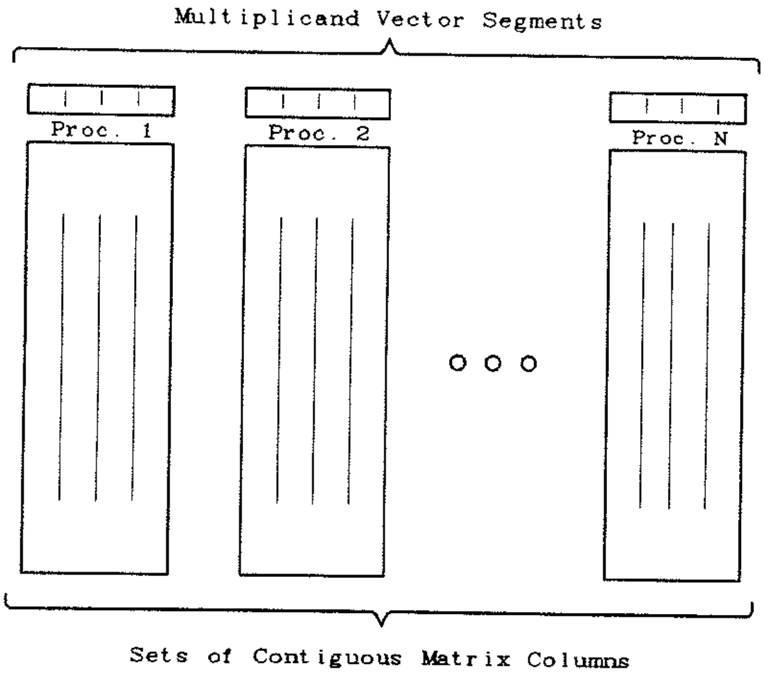
Multiplicand Vector Segments
Proc. 1
Proc. 2
Proc. N
Sets of Contiguous Matrix Columns

Figure 3.7: Column-wise matrix and vector partitioning. Only a segment of the vector multiplicand is needed in each processor.

The column-wise storage scheme also carries with it considerable communications overhead. However, it has the virtue of only requiring a segment of the multiplicand vector in each processor, and is compatible with the vector partitioning for parallel vector operations. Because of this property, column-wise partitioning is used here, as it allows the simultaneous application of matrix-vector multiplication and the monadic and dyadic vector operations described previously.

With this partitioning, matrix-vector multiplication produces (in each processor) a full length vector whose components are partial sums of the resultant vector. To form the final result, it is necessary to perform a sum of all such vectors. As in the scalar-product case, we must combine N_p objects distributed over N_p processors. Again, a solution is offered in the form of the shuffle-exchange network, the only difference being that, instead of scalars, vectors are transmitted and summed in each iteration of the algorithm.

This algorithm is not optimal in that it performs more communication than is necessary. Each processor only needs to end up with the segment of the resultant vector it will store, but instead ends up with the whole vector. In the case of transputer arrays, however, the communication cost of the algorithm is not significant when compared with the computation cost, and we can trade off algorithmic efficiency for algorithmic simplicity.

As a measure of the algorithm's performance, its efficiency can be estimated in terms of calculated multiprocessor and uniprocessor execution times. Consider a square matrix of size N. No generality is lost by requiring that the number of columns N be evenly divisible by the number of processors N_p. A single processor has to perform N^2 multiply-adds in order to form a matrix-vector product. In terms of the constants defined in Table 3.1, the execution time can be estimated as:

$$T_1 = N^2(T_v^+ + T_v^\times). \tag{3.4}$$

Note that vector operation timing constants have been used so that array indexing costs are implicitly catered for.

The multiprocessor execution time consists of two components. The first is the time required to form the partially summed vector (the computational component). This is an identical operation to what is done in the single processor case, except that it is applied to N/N_p matrix columns instead of N columns.

The second component is the time needed to sum the N_p vectors using the SE addition algorithm. An iteration of the SE algorithm consists of two vector communication steps (shuffle vector, exchange vector) followed by a vector sum. Since the vectors involved are of length N, the cost of an iteration is $N(2T_v^X + T_v^+)$, with a total of $\log_2(N_p)$ such iterations required to complete the sum.

Combining the execution times of these two components of the algorithm gives

$$T_{N_p} = \underbrace{(N^2/N_p)(T_v^+ + T_v^\times)}_{\text{computation}} + \underbrace{N \log_2(N_p)(2T_v^X + T_v^+)}_{\text{shuffle-sum}}. \tag{3.5}$$

Equations (3.4) and (3.5) can be used to calculate the algorithmic efficiency defined in Section 3.1. The results are summarized in Figure 3.8 for various problem and network sizes. We can see that the algorithm has the desirable characteristic of maintaining efficiency for larger problem sizes when the number of processors is increased. Moreover, the efficiencies are high, indicating that the overhead due to communication is small relative to the cost of computation.

3.3.2 Sparse Matrices

In anticipation of the needs of Section 4.2, the next task is to examine the behaviour of the algorithm for sparse matrices. † If one takes advantage of the sparseness of a matrix, the effect is to reduce the amount of computation required to perform a matrix-vector product. We still must communicate full length vectors when shuffling, however, so the sparseness of the matrix does not reduce these costs. As the matrix becomes more sparse, the communication costs then become dominant and the efficiency of the algorithm drops.

We may use equations (3.4) and (3.5) to calculate the efficiency for a randomly sparse matrix by augmenting them to represent the reduced computation costs. Let N_{nz} denote the average number of non-zero coefficients in a matrix column. If we take advantage of the sparsity and only compute with non-zero matrix coefficients, (3.4) and (3.5) become:

$$T_1 = N_{nz} N (T_v^+ + T_v^\times), \tag{3.6}$$

† The sparse matrix-vector multiplication algorithm described in this section was presented by the author at the Third Occam Users Group Meeting in Chicago IL, September 1987.

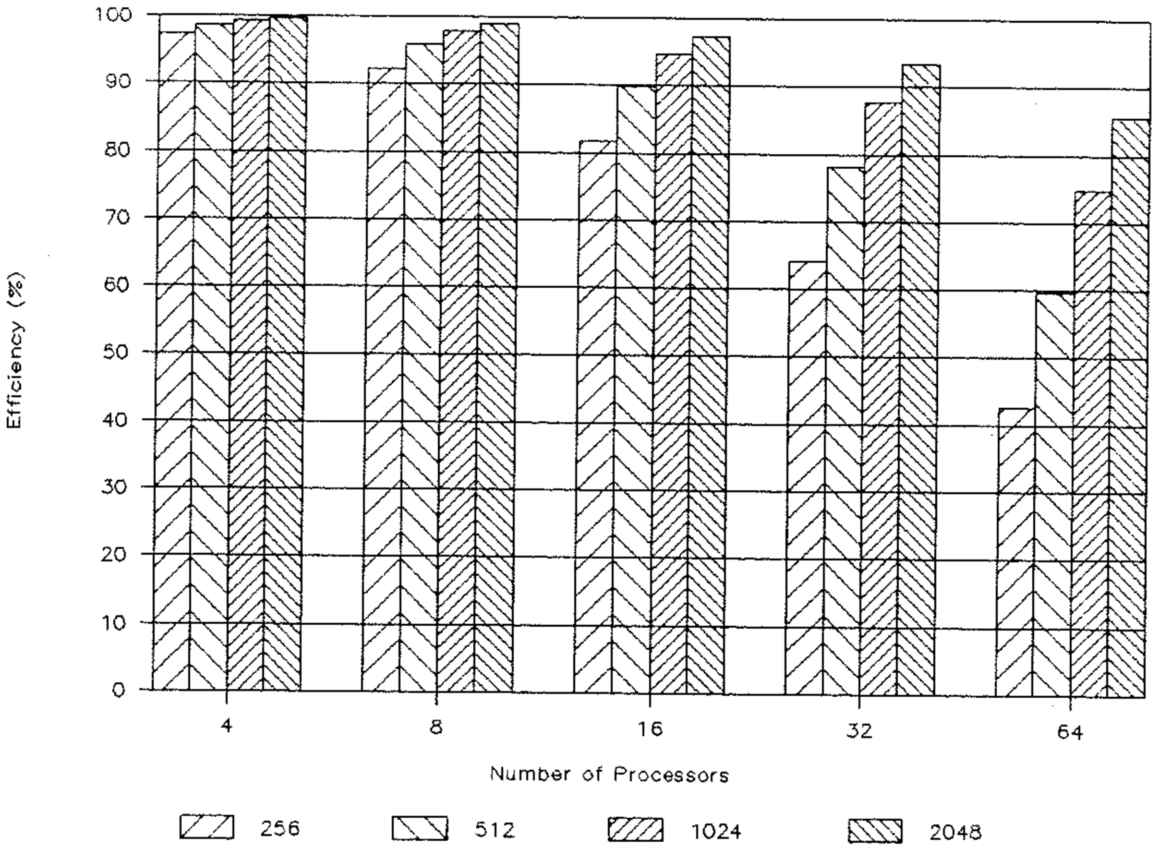

Figure 3.8: SE efficiency for dense matrices (T-414). Estimates are shown for matrices of size 256, 512, 1024, and 2048.

and

$$T_{N_p} = \underbrace{(N_{nz}N/N_p)(T_v^{+} + T_v^{\times})}_{\text{computation}} + \underbrace{N\log_2(N_p)(2T_v^{X} + T_v^{+})}_{\text{shuffle-sum}}. \tag{3.7}$$

Figure 3.9 shows the efficiencies estimated using equations (3.6) and (3.7) for a 1024 × 1024 matrix with 31 non-zeros per column. As expected, the efficiency drops dramatically for larger numbers of processors. In fact, the expected execution time for 64 processors is actually larger than that for 32 processors – a dismal result indeed.

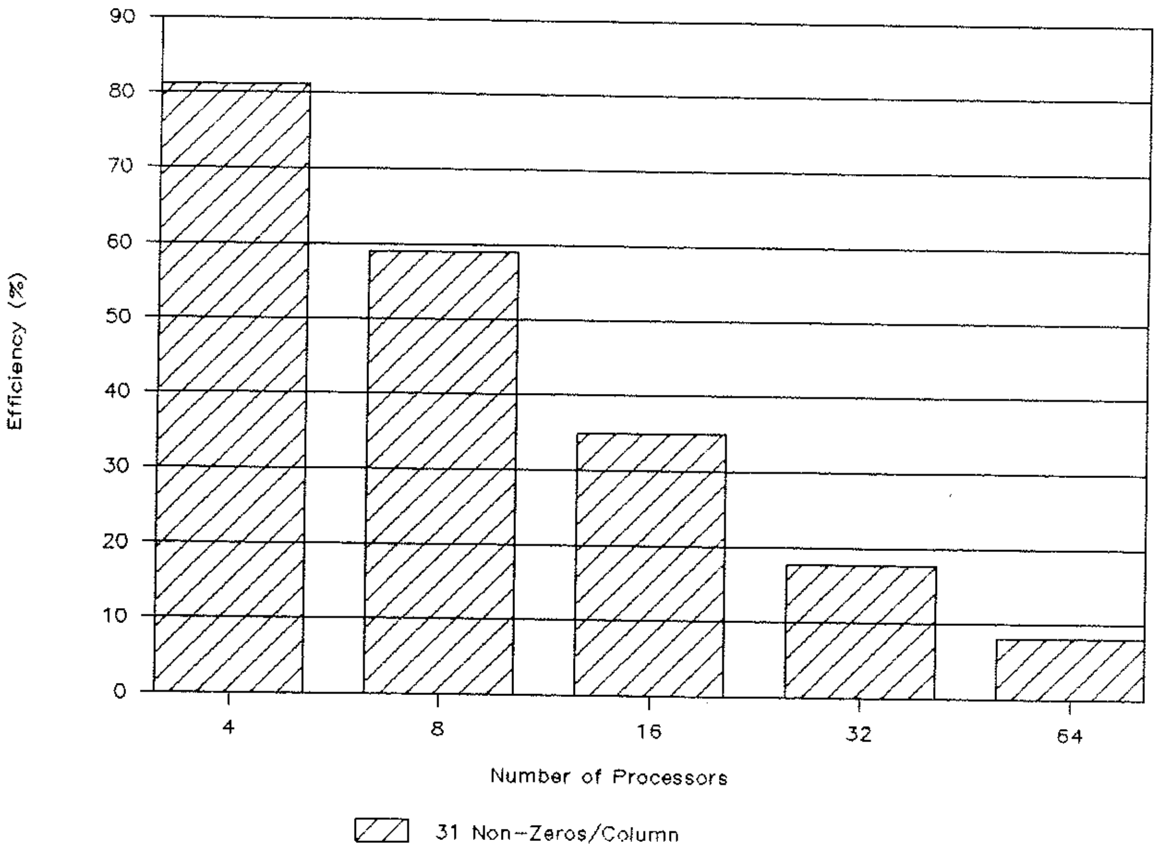

Figure 3.9: SE efficiency for 1024 × 1024 sparse matrix. Efficiency calculated assuming 31 non-zero coefficients per column.

With this realization, it can be seen that a new algorithm is needed in the case of sparse matrix-vector multiplication. The basic problem is that the global communication inherent in the SE network is too costly relative to the small amount of computation required for a sparse matrix-vector multiplication.

Instead, consider an implementation on a linear array of processors, with the matrix columns and vectors partitioned as before. If we are computing $\mathbf{A}\underline{x} = \underline{b}$, a typical component of $\underline{b}$ is given by

$$b_i = \sum_{j=1}^{N} a_{ij} x_j, \tag{3.8}$$

where it is assumed that no calculations are performed when $a_{ij} = 0$.

Because vectors are partitioned across the array of processors, each of the b_i is located on a particular processor (the one which contains column i of the matrix). As a result, terms that contribute to b_i must be communicated to that processor if they are not generated locally. Given a randomly sparse matrix, however, it is possible for non-zero terms in equation (3.8) to be produced in any of the processors in the network. It would therefore be necessary to perform a great deal of global communication to accumulate the components of $\underline{b}$ in the proper processors.

For randomly sparse matrices, then, the picture is rather gloomy, as global communication translates into algorithmic inefficiency. Ideally, one would like to have nearest or next-nearest neighbour communication only. It is possible to arrange this, but we must sacrifice the generality of the algorithm in the process.

Consider equation (3.8) again. If we restrict the range over which non-zero coefficients occur in row i, we can effectively limit the range of processors which contribute to b_i since columns are topologically contiguous across the array. In general terminology, we must restrict the application of the algorithm to banded matrices as shown in Figure 3.10.

Restriction to banded matrices is a common requirement for parallel sparse matrix-vector multiplication. For example, systolic algorithms that have been developed require that the matrix be banded (Kung [1982]). Moreover, they have the uncomfortable requirement that there be as many processors as the band is wide. This requirement reduces their flexibility, and makes no provision for the band itself being sparse – causing them to compute with zero coefficients. Vector computers require similar restrictions, although they can avoid calculating with zero coefficients (Madsen et al. [1976]).

Let a matrix of size $N \times N$ be distributed over a linear array of processors such that $C_p \times N_p = N$, where C_p is the number of columns stored in each processor. Defining P to be the row bandwidth, and Q to be the column bandwidth (as shown

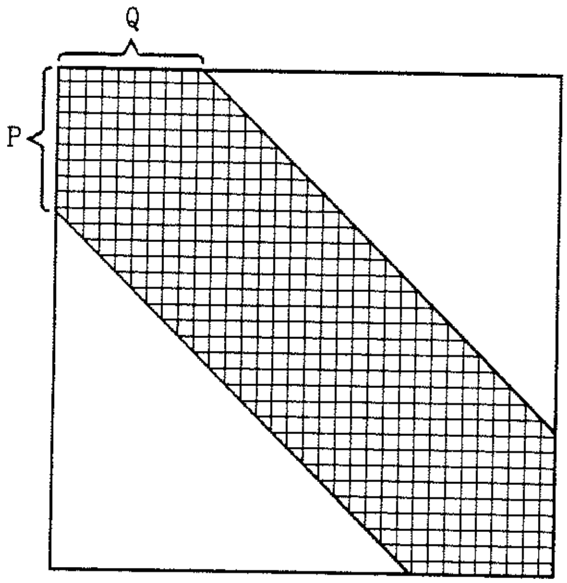

Figure 3.10: A banded matrix. P and Q define the range over which non-zero coefficients occur. The bandwidth of the matrix is defined as $P + Q - 1$.

in Figure 10), the key to an efficient algorithm is the relative size of P, Q, and C_p. We can insure Ith neighbour communication if

$$P, Q \leq I\,C_p + 1. \tag{3.9}$$

For any given row i, there will not be any non-zero coefficients in the $(I+1)$th processor to the left or right under these conditions. By restricting the band, we restrict the range of processors which can contribute to a given row-sum of the product, which in turn minimizes the communication requirements.

The efficiency of this algorithm can be estimated in a manner similar to the SE case. The major difference is that the communication requirements are vastly reduced. For simplicity, let P and Q be such that nearest neighbour communication is possible ($I = 1$ in equation (3.9)). Each processor will accumulate row-sums for all of its non-zero rows. If it contains the diagonal element of that row, it will not need to send that row-sum, as it is the accumulator for the corresponding b_i.

Rather, it will need to receive partial row-sums from neighbouring processors so that it can form the final accumulation for the b_i that it stores.

It is only necessary for each processor to send $(Q-1)$ row-sums to the neighbour to the left since the band only extends $(Q-1)$ rows above the row containing a processor's left-most diagonal element. Similarly, only $(P-1)$ row-sums will need to be communicated to a neighbour to the right since the band extends only $(P-1)$ rows below the row containing a processor's right-most diagonal element. This ignores the special cases which occur at each end of the array.

The time needed to calculate the row-sums is proportional to the number of non-zeros as in the SE case. Since transputer hardware makes it possible to send and receive values both to the left and right simultaneously, the communication time is proportional to $(\max(P,Q)-1)$ and not $(2(P-1)+2(Q-1))$ as it would be if all communication had to be done serially. Once the values are communicated, however, the final accumulation in each processor is a serial process and is thus proportional to $(P+Q-2)$. Tallying up the communication and computation costs yields:

$$T_{N_p} = \underbrace{(N_{nz}C_p)(T_v^+ + T_v^\times)}_{\text{computation}} + \underbrace{(\max(P,Q)-1)(T_v^X)}_{\text{communication}} + \underbrace{(P+Q-2)(T_v^+)}_{\text{final accum.}}. \qquad (3.10)$$

Equations (3.10) and (3.6) allow us to calculate the efficiency for the new algorithm. The results are shown in Figure 3.11 for a matrix with 5 non-zero coefficients per column. Results are shown for two bandwidths, to demonstrate the effect of a wider band on the efficiency of the algorithm.

This example presents what is just about the worst case for the algorithm. With only five non-zeros per column, there is very little computation relative to the communication involved. Despite this, efficiencies are still high, even for the larger bandwidth where communication costs are even greater.

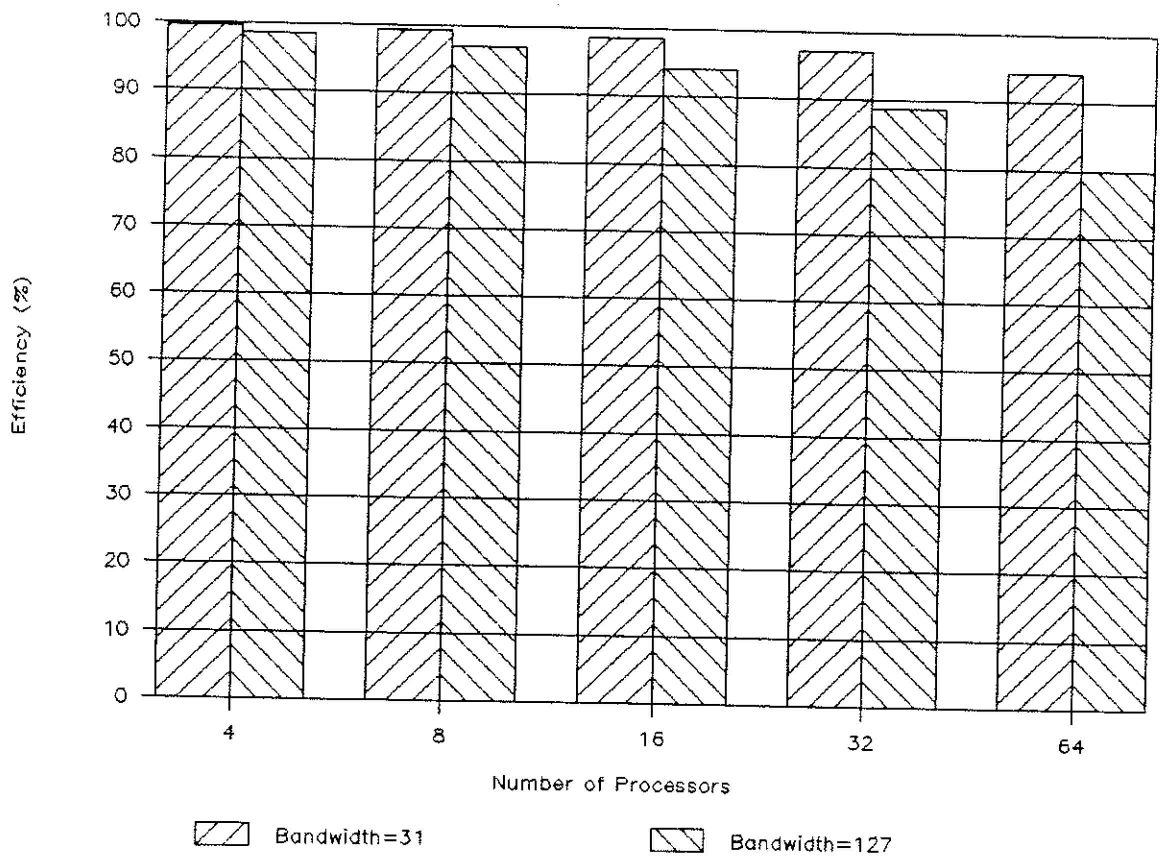

Figure 3.11: Estimated efficiency for banded matrix-vector multiplication. Calculated assuming a 4096 × 4096 matrix with 5 non-zeros per column.

Lest one get the impression that this "watered down" algorithm is not very useful, consider that it is used in conjunction with the conjugate gradient algorithm in Section 4.2 to solve linear systems of equations. The matrices considered there, are not only banded in this way, but result from common CAE techniques such as finite-differences (Forsythe and Wasow [1960]) and finite-elements (Zienkiewicz [1971]). Thus, while not a general algorithm, it is still very useful for its intended application.

CHAPTER IV

MATRIX SOLUTION ALGORITHMS

Matrix solution algorithms play an important role in numerical analysis and CAE algorithms. Virtually all methods of solution (even solutions of non-linear problems) produce a linear system of equations which must be solved.

Two algorithms are presented. The first is a fairly straightforward implementation of a Gaussian elimination solver with row pivoting on a linear array of transputers. The partitioning used is not the most efficient imaginable, but was cast this way to allow its use as part of the boundary element engine described in Chapter 5. The efficiency of the implementation is presented as part of the results in Chapter 5.

The second algorithm is a sparse-matrix solver based upon the Conjugate Gradient (CG) algorithm. While the algorithm is restricted to banded matrices with rather stringent properties, this does not hinder its usefulness, as such matrices are produced by a variety of standard numerical analysis techniques. The algorithm has the additional advantage of serving as a proving ground for many of the matrix-vector algorithms introduced in Chapter 3.

4.1 Gaussian Elimination

As we will see in Chapter 5, the Boundary Element Method (BEM) produces a matrix system that must be solved. This section outlines a parallel implementation of the Gaussian elimination matrix solution algorithm on a linear array of transputers (the same architecture as for the BEM matrix generation algorithm).[†] Thus the two algorithms are able to work cooperatively on a problem.

4.1.1 The Algorithm

Gaussian elimination (GE) is a systematic way of performing elimination on N equations in N unknowns. Consider the linear system

$$\mathbf{S}\underline{\sigma} = \underline{b}. \tag{4.1}$$

For convenience in implementing the algorithm, the $\underline{b}$ is amalgamated with the **S** matrix to form

$$\mathbf{S}' = \begin{pmatrix} s_{11} & s_{12} & \ldots & s_{1n} & b_1 \\ s_{21} & s_{22} & \ldots & s_{2n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ s_{n1} & s_{m2} & \ldots & s_{nn} & b_n \end{pmatrix} \tag{4.2}$$

This allows all the row-operations to be carried out in a single step, rather than needing a separate step to handle the right-hand side. Gaussian elimination solves such a system by using row-operations to eliminate all coefficients below the diagonal, forming an upper-triangular system that can be solved by back-substitution.

The first step in the elimination of column k is to choose a suitable pivot equation. In the partial-pivoting form of the algorithm described here, the system row $k' > k$ that has the largest coefficient in column k is chosen as the pivot equation and swapped with row k. This procedure is done to ensure stability of the algorithm and to avoid zero value diagonal elements which will cause execution errors.

[†] Presented by the author at the Second Occam Users Group Meeting in Santa Clara CA, March 1987.

After the pivoting procedure, the next step is to eliminate the entries below the diagonal in the pivot column by subtracting an appropriate multiple of the pivot row k from the rows below it (this operation is also applied to the right-hand side of the system). In equation form, the elimination of the k'th column can be represented as

$$s_{ij}^{k} = s_{ij}^{k-1} - \frac{s_{kj}^{k-1}}{s_{kk}^{k-1}} s_{ik}^{k-1} \qquad \begin{cases} k+1 \leq i \leq N \\ k+1 \leq j \leq N+1 \end{cases} \tag{4.3}$$

where the superscript denotes the elimination step and we have included the right-hand side of the system in an extra column of the system matrix (hence the range to $N+1$ for the column index).

After this algorithm is applied iteratively to the first $N-1$ columns of the system, the diagonalized system is then solved through the following back-substitution process. The first step is to solve for σ_n giving

$$\sigma_n = \frac{s_{N,N+1}}{s_{NN}}. \tag{4.4}$$

Thereafter, each succeeding σ_i is determined by

$$\sigma_i = \frac{s_{i,N+1} - \sum\limits_{j=i+1}^{N} s_{ij}\sigma_j}{s_{ii}} \qquad i = n-1, n-2, \ldots, 1. \tag{4.5}$$

4.1.2 Partitioning of Forward Elimination Phase

The GE algorithm has the advantage of having an obvious partitioning for parallel processing. Before detailing the partitioning, a few observations are worth noting:

1. All information necessary for the elimination of a column k is derived from the coefficient values present in that column. This implies that a processor which possesses all the information within a column can determine the actions and coefficients necessary to eliminate that column.
2. When eliminating column k, each coefficient in rows $\{k+1, k+2, \ldots, N\}$ is augmented by a multiple of the pivot equation coefficient in row k in the column directly above it. Furthermore, the value of the multiplying coefficient is derived from the pivot column. In short, calculations in all columns (including the right-hand side vector) to the right of the pivot column require information from the pivot column.
3. No calculations are performed within a column after it has been eliminated (except when back-substitution is being performed). This fact has important implications on the maximum possible speedup efficiency of the algorithm.

To execute the GE algorithm in parallel, each processor is given a contiguous series of columns within the matrix in which it is responsible for all computation as was shown in Figure 3.6. Because all the information necessary for eliminating a column is contained within that column, it is possible for each processor to play master and direct the elimination process in the processors to the right while the pivot column is one of the columns it is responsible for. This is done by determining which matrix rows have to be swapped and the values of the coefficients of elimination for each row and sending this information to the next processor on the right. The pivot processor then goes on and finishes the elimination computation in the columns to the right of the pivot column for which it is responsible.

All processors to the right of the pivot processor, upon receiving the message, pass it on (if there is another processor to the right) and use the information received to compute the results of the current elimination step. The end result is that the work of performing the current elimination step is shared between a number of processors.

The global communication required in this partitioning is compensated for by the fact that computation and communication are overlapped. After a processor passes on the message, it is free to perform calculations. Because computation time dominates communication time (for most practical problems and processing systems) the processors are kept busy most of the time. Thus global communication does not impact the efficiency of the algorithm to a great extent.

What does impact the efficiency, though, are the implications of observation 3. Once a processor has supervised the elimination of its columns, it has nothing to do until the back-substitution phase. It therefore sits idle and does not contribute. The only compensating factor is that while fewer processors come into play, the work that each of the remaining active processors have to do for each column becomes less (i.e. the elimination involves a shorter length in the column).

The theoretical expressions for efficiency and execution times are derived in Chapter 5, where they are compared to experimental results. Since the derivation ignores all communication costs, we arrive at an expression that gives the maximum possible efficiency. Comparing this number to the experimental results allows us to show that communication and computation are overlapped, thus lessening the effect of global communication. The end result is that the algorithm as implemented does not have a theoretical maximum efficiency of 100%. As we will see in Chapter 5, the theoretical efficiency with two processors is around 70% for the forward elimination phase, and falls slowly as the number of processors is increased.

This problem could be avoided to a large extent by choosing a different partitioning of the system matrix. For example, instead of using contiguous sets of matrix columns for each processor, we could have assigned successive columns to different processors in a cyclic fashion. In this way processors would have columns from all regions of the matrix and could be kept busy doing computation all of the time. This strategy combined with full Gauss-Jordan elimination leads to an efficient algorithm (Li et al. [1987]).

One problem with this approach is that it would require more logic and more communication to implement, perhaps offsetting some of the computational gains. The column elimination information would have to be shipped to all processors (and would always be of length N), instead of only those to the right, which represents a sizable increase in communication costs.

Further to the point, the present algorithm was designed to work in consort with the BEM algorithm of Chapter 5. Without a matrix partitioned into contiguous sets of columns, the implementation of an efficient BEM matrix generator would have been much more difficult, if not impossible. It was therefore deemed acceptable to sacrifice some efficiency in the solver so that the combined algorithms could be more efficient.

4.1.3 Partitioning of the Back-substitution Phase

In comparison to the elimination process, the amount of calculation involved in the back-substitution phase of the algorithm is of little consequence. While there is some parallelism present, it is debatable if exploiting it would be worthwhile. As such, we seek a method which caries out the operations in a simple manner, and will ignore the parallelism present.

Examination of equation (4.5) reveals that, while each processor can evaluate the terms in the summation in parallel, we are still faced with the sequential accumulation of the sum. Worse, the accumulation requires transfer of partial sums from the last processor in the array to the processor containing the column corresponding to the unknown being solved for (column i). This communication is wasteful since the transputer communicates vectors more efficiently than it does a single value (due to overhead involved in initiating the communication sequence).

If it can be arranged that we transmit vectors instead of scalars, the algorithm will be more efficient.

This can be accomplished by restructuring the back-substitution process so that all the calculations performed with a matrix column are done in one step. In this way, each σ_i is partially accumulated as each column i $\{i = (n-1), (n-2), \ldots, 1\}$ is dealt with in turn. We are then able to pass $\underline{\sigma}$ backwards as a vector rather than scalar partial sums. An additional saving occurs at the end of the process when the final solution vector is left in processor one, and can be extracted to the controlling processor in one step.

The first step is to assign the right-hand side to $\underline{\sigma}$. Then for each column k $\{k = N, (N-1), \ldots, 1\}$ of the S-matrix we calculate

$$\sigma_k = \frac{\sigma_k}{s_{kk}}, \tag{4.6}$$

followed by

$$\sigma_i = \sigma_i - s_{ik}\sigma_k \qquad i \;=\; 1, 2, \ldots, (k-1), \tag{4.7}$$

where this last step (4.7) is not performed when $k = 1$.

Each processor, proceeding from last to first, performs these calculations, and passes the intermediate $\underline{\sigma}$ back to the proceeding processor. Finally the first processor, after having finished all its calculations, passes the now complete $\underline{\sigma}$ back to the controlling processor.

The disadvantage of this approach is that we sacrifice the parallel evaluation of the summation terms in favour of communication efficiency. The choice made will depend upon the relative speed of communication and computation on the transputer being used. For example, the T-800's floating performance might dictate that communication be favored.

4.2 The Conjugate Gradient Algorithm

This section describes a parallel implementation of the polynomial preconditioned conjugate gradient method (PPCG) on a linear array of transputers.† We begin by describing the conjugate gradient (CG) method, which is a semi-iterative technique for the solution of sparse–symmetric sets of linear equations.

This method is most useful in the preconditioned (PCG) form described in Section 4.2.2, and has received much attention as a solver for finite-difference problems on vector supercomputers (Kightley and Jones [1985], Wong and Jiang [1987], Wong [1987]). It has also been implemented on a single instruction/multiple data (SIMD) processor by Allen[1983].

Most of these parallel implementations make use of the polynomial preconditioning technique described in Section 4.2.3 (Dubois et al. [1979]). This method of preconditioning has proved to be pivotal in allowing efficient parallelization of the PCG algorithm.

4.2.1 The Classical Conjugate Gradient Algorithm

The conjugate gradient algorithm is derived from an optimization point of view.‡ An iterative method is developed that seeks a solution to a set of linear equations by requiring that each iterant minimize an error functional. The error functional is designed to give a measure of the current iterative solution's "closeness" to the exact solution, and as such, the solution vector that minimizes the error functional is the solution to the system of linear equations.

The solution to the system of equations

$$\mathbf{A}\underline{x} = \underline{b}, \tag{4.8}$$

† Presented by the author at the Third Occam Users Group Meeting in Chicago IL, September 1987.

‡ The presentation of the theory of the conjugate gradient method in this and the following sections is derived from a previous work of the author (Allen [1983]), and follows Axelsson [1977].

is sought, where $\mathbf{A}$ is a symmetric positive definite $N \times N$ matrix, and $\underline{x}$ and $\underline{b}$ are respectively the unknown and forcing vectors (length N). Let the exact solution to the equation (4.8) be denoted by

$$\underline{h} = \mathbf{A}^{-1}\underline{b}. \tag{4.9}$$

Given an estimate $\underline{x}$ of the solution vector, define the residual to be

$$\underline{r} = \underline{b} - \mathbf{A}\underline{x}. \tag{4.10}$$

With the above definitions, consider the quadratic functional (Mikhlin [1964], Axelsson [1977,5-6])

$$F(\underline{x}) = \frac{1}{2}\langle \underline{x}, \mathbf{A}\underline{x}\rangle - \langle \underline{b}, \underline{x}\rangle, \tag{4.11}$$

which is a so-called energy functional. The solution which minimizes (4.11) is the solution of minimum energy. Note that $\langle\ ,\ \rangle$ denotes the standard inner product, which is assumed valid for real spaces.

As the name of the CG method implies, information about the gradient of the functional (4.11) is used to determine a path to its minimum. The gradient of (4.11) is given by

$$\underline{g}(\underline{x}) = \nabla(F(\underline{x})) = \mathbf{A}\underline{x} - \underline{b}. \tag{4.12}$$

Noting the definition of the residual, (4.12) can be rewritten as

$$\underline{g}(\underline{x}) = -\underline{r}. \tag{4.13}$$

Observe here, that in following a path to the minimum of the functional, the negative of (4.13) is used since it is in the direction of the minimum.

Rewriting (4.11) in the form

$$F(\underline{x}) = \frac{1}{2}\langle(\underline{h} - \underline{x}), \mathbf{A}(\underline{h} - \underline{x})\rangle - \frac{1}{2}\langle\underline{h}, \mathbf{A}\underline{h}\rangle, \tag{4.14}$$

and using the fact that the last term is constant, it can be seen that minimizing (4.11) is equivalent to minimizing

$$E(\underline{x}) = \frac{1}{2}\langle(\underline{h} - \underline{x}), \mathbf{A}(\underline{h} - \underline{x})\rangle, \tag{4.15}$$

which shall be called the error functional. Two alternate forms of (4.15) are

$$E(\underline{x}) = \frac{1}{2}\langle\underline{r}, \mathbf{A}^{-1}\underline{r}\rangle, \tag{4.16}$$

and

$$E(\underline{x}) = \frac{1}{2}\langle\underline{g}(\underline{x}), \mathbf{A}^{-1}\underline{g}(\underline{x})\rangle, \tag{4.17}$$

where $\underline{g}(\underline{x})$ is given by (4.13). Note, that the gradients of (4.11), (4.15), (4.16), and (4.17) are equal.

In a CG iteration, one constructs a path through the space of solution vectors such that (4.17) is minimized and a solution is obtained on the N'th step. Each iterative step may be considered as an exact line search of the form

$$\underline{x}^{k+1} = \underline{x}^{k} + \alpha_k \underline{d}^{k}. \tag{4.18}$$

That is, in proceeding from the current solution vector to the next, one travels along a direction $\underline{d}^k$ a distance α_k. The direction vector is chosen with some idea of the gradient, and the parameter α_k is chosen so that $\underline{x}^{k+1}$ will be located at the minimum of (4.17) along the line $\underline{d}^k$. The requirement that $E(\underline{x})$ be minimized successively by each step of the CG algorithm allows the value of α_k to be determined.

Applying this search to (4.17) we observe that

$$E(\underline{x}^k + \alpha_k \underline{d}^k) = \langle(\underline{b} - \mathbf{A}(\underline{x}^k + \alpha_k \underline{d}^k)), \mathbf{A}^{-1}(\underline{b} - \mathbf{A}^{-1}(\underline{x}^k + \alpha_k \underline{d}^k))\rangle, \tag{4.19}$$

may be written as

$$E(\underline{x}^k + \alpha_k \underline{d}^k) = -2\alpha_k \langle \underline{r}^k, \underline{d}^k \rangle + \alpha_k^2 \langle \underline{d}^k \mathbf{A} \underline{d}^k \rangle. \tag{4.20}$$

Setting the derivative with respect to α_k of (4.20) equal to zero gives the minimization requirement that

$$\alpha_k = \frac{\langle \underline{r}^k, \underline{d}^k \rangle}{\langle \underline{d}^k, \mathbf{A}\underline{d}^k \rangle}. \tag{4.21}$$

Now consider the calculation of the residual vectors for each CG iteration. While they may be calculated from (4.10), the matrix multiplication involved is not useful anywhere else in the algorithm. Using (4.10) and (4.18), the following recursive definition for the residual is obtained:

$$\underline{r}^{k+1} = \underline{r}^k - \alpha_k \mathbf{A} \underline{d}^k. \tag{4.22}$$

This expression involves a matrix product that is used elsewhere in the algorithm, giving greater efficiency.

At this juncture, recursive expressions for both $\underline{x}$ and $\underline{r}$ have been determined. All that is needed to complete the algorithm is a definition for $\underline{d}$. It is the choice made for $\underline{d}$ that distinguishes the CG method from the more general conjugate direction (CD) method. The conjugate direction method does not specify how the direction vectors are to be derived, save that they be A-orthogonal (i.e. $\langle \underline{d}, \mathbf{A}\underline{d} \rangle = 0$). The conjugate gradient method, on the other hand, requires that the direction vectors be constructed via A-orthogonalization of the residual vectors generated by (4.22).

The orthogonalization could be realized by a Gram-Schmidt process (Lang [1972,138-139]), but it is undesirable to store each $\underline{r}$ vector that is generated. Instead, the following iterative procedure is used:

$$\underline{d}^{\circ} = \underline{r}^{\circ}, \tag{4.23}$$

followed by

$$\underline{d}^{k+1} = \underline{r}^{k+1} + \beta_k \underline{d}^{k}. \tag{4.24}$$

To prove the validity of this process, the orthogonality of the residual vectors must be demonstrated. With that fact, the A-orthogonality of the direction vectors can be proved, and finally the value of β_k determined.

Using (4.23), (4.22) can be rewritten as

$$\underline{r}^{k} = (I + C_1\mathbf{A} + C_2\mathbf{A}^2 + \ldots + C_k\mathbf{A}^k)\underline{r}^{\circ}, \tag{4.25}$$

or

$$\underline{r}^{k} = (I + P_k(\mathbf{A}))\underline{r}^{\circ}, \tag{4.26}$$

where $P_k(\mathbf{A})$ is a polynomial of degree k in $\mathbf{A}$ with no constant term. Substituting (4.26) into the error functional (4.16) produces

$$E(\underline{x}^{k}) = \langle (I + P_k(\mathbf{A}))\underline{r}^{\circ}, \mathbf{A}^{-1}(I + P_k(\mathbf{A}))\underline{r}^{\circ}\rangle. \tag{4.27}$$

Interpreting (4.27) as defining the square of a norm with respect to the matrix $\mathbf{A}^{-1}$, the minimization of (4.27) is equivalent to requiring that $-P_k(\mathbf{A})\underline{r}^{\circ}$ be an approximation to $\underline{r}^{\circ}$ – so that their sum is zero. The best approximation to $\underline{r}^{\circ}$ occurs when the error is orthogonal (in the $\mathbf{A}^{-1}$ norm) to the basis of approximating vectors (Davis [1975,176]).

It is therefore required that

$$\langle (I + P_k(\mathbf{A}))\underline{r}^\circ, \mathbf{A}^{-1} P_j(\mathbf{A})\underline{r}^\circ \rangle = 0 \qquad (j \ \leq \ k). \tag{4.28}$$

But

$$P_j(\mathbf{A}) = \mathbf{A} + \mathbf{A} P_{j-1}(\mathbf{A}). \tag{4.29}$$

Substituting (4.29) into (4.28) gives

$$\langle (I + P_k(\mathbf{A}))\underline{r}^\circ, (I + P_j(\mathbf{A}))\underline{r}^\circ \rangle = 0 \qquad (j \ \leq \ k-1), \tag{4.30}$$

which (using (4.26)) demonstrates the orthogonality of the residual vectors.

With the result in (4.30), the A-orthogonality of the direction vectors can be proved. Let j be less than k, then (using (4.22) and (4.24))

$$\langle \underline{d}^k, \mathbf{A}\underline{d}^j \rangle = \frac{\beta_{j-1}}{\alpha_k} \langle (\underline{r}^k - \underline{r}^{k+1}), \underline{d}^{j-1} \rangle. \tag{4.31}$$

Extending (4.31) by induction leaves

$$\langle \underline{d}^k, \mathbf{A}\underline{d}^j \rangle = \frac{\beta_{j-1}\beta_{j-2}\dots\beta_\circ}{\alpha_k} \langle (\underline{r}^k - \underline{r}^{k+1}), \underline{d}^\circ \rangle. \tag{4.32}$$

Using (4.23) and the orthogonality of the residual vectors shows that (4.32) equals zero, which proves the A-orthogonality of the direction vectors.

It remains for the value of β_k to be determined. The A-orthogonality of the direction vectors gives

$$\langle \underline{r}^{k+1} + \beta_k \underline{d}^k, \mathbf{A}\underline{d}^k \rangle = 0. \tag{4.33}$$

Solving for β_k produces

$$\beta_k = -\frac{\langle \underline{r}^{k+1}, \mathbf{A}\underline{d}^k \rangle}{\langle \underline{d}^k, \mathbf{A}\underline{d}^k \rangle}. \tag{4.34}$$

It is profitable to put (4.21) and (4.34) into more computationally efficient forms. Using (4.24) and the orthogonality of the residual vectors gives

$$\alpha_k = \frac{\langle \underline{r}^k, \underline{r}^k \rangle}{\langle \underline{d}^k, \mathbf{A}\underline{d}^k \rangle}. \tag{4.35}$$

Substituting (4.22) and then (4.35) into (4.34) yields

$$\beta_k = \frac{\langle \underline{r}^{k+1}, \underline{r}^{k+1} \rangle}{\langle \underline{r}^k, \underline{r}^k \rangle}. \tag{4.36}$$

The derivation of the classical conjugate gradient algorithm is now complete. Using equations (4.10), (4.18), (4.22), (4.23), (4.24), (4.35), and (4.36), the algorithm can be summarized as follows:

$$\underline{x}^\circ = ARBITRARY, \tag{4.37}$$

$$\underline{r}^\circ = \underline{b} - \mathbf{A}\underline{x}^\circ, \tag{4.38}$$

$$\underline{d}^\circ = \underline{r}^\circ, \tag{4.39}$$

$$\alpha_k = \frac{\langle \underline{r}^k, \underline{r}^k \rangle}{\langle \underline{d}^k, \mathbf{A}\underline{d}^k \rangle}, \tag{4.40}$$

$$\underline{x}^{k+1} = \underline{x}^k + \alpha_k \underline{d}^k, \tag{4.41}$$

$$\underline{r}^{k+1} = \underline{r}^k - \alpha_k \mathbf{A}\underline{d}^k, \tag{4.42}$$

$$\beta_k = \frac{\langle \underline{r}^{k+1}, \underline{r}^{k+1} \rangle}{\langle \underline{r}^k, \underline{r}^k \rangle}, \tag{4.43}$$

and

$$\underline{d}^{k+1} = \underline{r}^{k+1} + \beta_k \underline{d}^k, \tag{4.44}$$

where $k = 0, 1, \ldots \infty$ and the iteration terminates when the Euclidean norm of the residual vector is less than some prescribed value.

4.2.2 The Preconditioned Conjugate Gradient Algorithm

Theoretically, the CG method should terminate in a finite number of steps (less than or equal to N - the dimension of the linear system). However, if round-off error occurs, or if the system matrix has a large spectral condition number (defined as the ratio of the largest to the smallest eigenvalue), convergence may never occur or may take considerably more than N iterations. Also, for large N, even a well-conditioned system will require a large amount of execution time owing to the fact that each CG iteration is fairly expensive.

The slow convergence rate of the CG algorithm can be improved by performing a preconditioning process on the system matrix. The effect is to reduce the spectral condition number of the system matrix which in turn improves the convergence rate (Kershaw [1978,46], Axelsson [1977,17-23]).

Preconditioning is realized by multiplying the original system (4.8) by matrix $\mathbf{K}^{-1}$ which is an approximate inverse to the system matrix. The optimal preconditioning matrix would be the precise inverse of the system matrix, since multiplying by such a matrix would solve the system exactly in one iteration. The effect of preconditioning, then, is to put the linear system "closer" to its solution.

Since our system matrix is symmetric, the matrix $\mathbf{K}$ will also be symmetric, and can be written as

$$\mathbf{K} = (\mathbf{L}\mathbf{L}^T). \qquad (4.45)$$

Multiplying (4.8) by $\mathbf{K}^{-1}$ gives the new system

$$(\mathbf{L}\mathbf{L}^T)^{-1}\mathbf{A}\underline{x} = (\mathbf{L}\mathbf{L}^T)^{-1}\underline{b}. \qquad (4.46)$$

For the present purpose, it is necessary to rewrite (4.8) as

$$(\mathbf{L}^{-1}\mathbf{A}\mathbf{L}^{-T})(\mathbf{L}^{T}\underline{x}) = (\mathbf{L}^{-1}\underline{b}), \tag{4.47}$$

or (to define the primed quantities),

$$\mathbf{A}'\underline{x}' = \underline{b}'. \tag{4.48}$$

Since $(\mathbf{L}\mathbf{L}^{T})^{-1}\mathbf{A}$ and $(\mathbf{L}^{-1}\mathbf{A}\mathbf{L}^{-T})$ are similar matrices and have the same eigenvalues, the convergence properties of (4.48) and (4.46) will be identical. The CG method, now called the Preconditioned Conjugate Gradient (PCG) method, can be applied to the system (4.48) in a manner identical to its application to (4.8). This results in a set of equations "identical" to equations (4.37)-(4.44), but with primed quantities replacing the normal quantities.

$\mathbf{A}', \underline{b}'$, and $\underline{x}'$ are as defined in (4.48). As for $\underline{r}'$, rewriting (4.16) in the form

$$E(\underline{x}^{k}) = \frac{1}{2}\langle(\mathbf{L}^{-1}\underline{r}^{k}), (\mathbf{L}^{-1}\mathbf{A}\mathbf{L}^{-T})^{-1}(\mathbf{L}^{-1}\underline{r}^{k})\rangle, \tag{4.49}$$

allows the definition of $\underline{r}'$ as

$$E(\underline{x}^{k}) = \frac{1}{2}\langle\underline{r}'^{k}, \mathbf{A}'^{-1}\underline{r}'^{k}\rangle, \tag{4.50}$$

giving finally

$$\underline{r}'^{k} = \mathbf{L}^{-1}\underline{r}^{k}. \tag{4.51}$$

Similarly to (4.23),

$$\underline{d}'^{\circ} = \underline{r}'^{\circ}. \tag{4.52}$$

The relationship between $\underline{d}$ and $\underline{d}'$ is somewhat arbitrary. The choice

$$\underline{d}'^{k} = \mathbf{L}^{T}\underline{d}^{k} \tag{4.53}$$

is made as it results in a considerable simplification in the equations defining the PCG method.

With the above definitions for the primed variables, it is possible to transform the equations back to normal variables. The resulting algorithm is given in the following equations:

$$\underline{x}^{\circ} = ARBITRARY, \tag{4.54}$$

$$\underline{r}^{\circ} = \underline{b} - \mathbf{A}\underline{x}^{\circ}, \tag{4.55}$$

$$\underline{d}^{\circ} = \mathbf{K}^{-1}\underline{r}^{\circ}, \tag{4.56}$$

$$\alpha_k = \frac{\langle \underline{r}^k, \mathbf{K}^{-1}\underline{r}^k \rangle}{\langle \underline{d}^k, \mathbf{A}\underline{d}^k \rangle}, \tag{4.57}$$

$$\underline{x}^{k+1} = \underline{x}^k + \alpha_k \underline{d}^k, \tag{4.58}$$

$$\underline{r}^{k+1} = \underline{r}^k - \alpha_k \mathbf{A}\underline{d}^k, \tag{4.59}$$

$$\beta_k = \frac{\langle \underline{r}^{k+1}, \mathbf{K}^{-1}\underline{r}^{k+1} \rangle}{\langle \underline{r}^k, \mathbf{K}^{-1}\underline{r}^k \rangle}, \tag{4.60}$$

and

$$\underline{d}^{k+1} = \mathbf{K}^{-1}\underline{r}^{k+1} + \beta_k \underline{d}^k, \tag{4.61}$$

where $k = 0, 1, \ldots \infty$ and the iteration terminates when the Euclidean norm of the residual is less then some prescribed value.

4.2.3 Polynomial Preconditioning

The PCG algorithm presented in (4.54) – (4.61) is a special case of the generalized CG method first presented by Hestenes [1956,83-102]. In his derivation, Hestenes places no requirements on the properties of the matrix **K**. As a result, the choice for **K** is not entirely obvious. The derivation presented here has the advantage of indicating exactly what properties **K** should possess, namely that $\mathbf{K}^{-1}$ be an approximate inverse of the system matrix. Moreover, the method by which $\mathbf{K}^{-1}$ is obtained has not been specified.

For scalar processors, the most efficient way to obtain an approximate inverse seems to be the incomplete Cholesky factorization method put forth by Meijerink and van der Vorst [1977,148-162] (An implementation of which is discussed by Kershaw [1978,43-65]). Also, Nakonechny [1983] has shown that the the Incomplete Cholesky Conjugate Gradient (ICCG) method has the advantage of allowing efficient implementation of a linked-list sparsity scheme.

Despite its advantages, ICCG is not suitable for implementation on a parallel computer (Webb et al. [1982,325-329]). The incomplete Cholesky decomposition is inherently a recursive process that does not lend itself to parallel implementation. Successive column eliminations must proceed serially. While some parallelism can be extracted from a column elimination when a sparsity scheme is not used, with a sparsity scheme, parallel implementation is hopeless.

The goal here, then, is to arrive at an algorithm that:

1. incorporates preconditioning in its framework;
2. allows sparse storage of sparse system matrices; and
3. is efficiently implementable on a transputer array.

The above requirements are met by combining the basic PCG algorithm with a class of polynomial preconditioners discussed by Dubois et al. [1979,257-268] and Johnson et al. [1983,362-376].

The Polynomial Preconditioned Conjugate Gradient (PPCG) algorithm approximates the inverse of the system matrix $\mathbf{A}$ by a truncated Neumann series expansion. This approximate inverse is then used as $\mathbf{K}^{-1}$ in the PCG algorithm ((4.54) – (4.61)). Consider the splitting of the system matrix

$$\mathbf{A} = (\mathbf{M} - \mathbf{N}) = \mathbf{M}(\mathbf{I} - \mathbf{M}^{-1}\mathbf{N}). \tag{4.62}$$

In exact analogy with the theory of scalar series (as opposed to matrix series), $\mathbf{A}^{-1}$ can be represented exactly by

$$\mathbf{A}^{-1} = (\mathbf{I} - \mathbf{M}^{-1}\mathbf{N})^{-1}\mathbf{M}^{-1}, \tag{4.63}$$

which can be written as

$$\mathbf{A}^{-1} = \left(\sum_{i=0}^{\infty} (\mathbf{M}^{-1}\mathbf{N})^{i}\right)\mathbf{M}^{-1}, \tag{4.64}$$

or as

$$\mathbf{A}^{-1} = \left(\sum_{i=0}^{\infty}(\mathbf{I} - \mathbf{M}^{-1}\mathbf{A})^{i}\right)\mathbf{M}^{-1}. \tag{4.65}$$

Equation (4.65) assumes that

$$\rho(\mathbf{M}^{-1}\mathbf{N}) = \rho(\mathbf{I} - \mathbf{M}^{-1}\mathbf{A}) < 1, \tag{4.66}$$

where ρ is the spectral radius of its matrix argument (Mirsky [1982,332]). The latter point (4.66), follows from the fact that a matrix raised to higher and higher powers will approach zero only if its spectral radius is less than one (Mirsky [1982,328]). Wong [1987] discusses strategies for handling cases where the matrix does not meet this requirement.

Owing to the fact that the calculation of (4.65) is impossible, an approximation to the inverse of the system matrix can be constructed by truncating the series (4.65) after a few terms (typically 1 to 4). Let the truncated inverse be defined by

$$\mathbf{K}_z^{-1} = \left(\sum_{i=0}^{z-1} (\mathbf{I} - \mathbf{M}^{-1}\mathbf{A})^i \right) \mathbf{M}^{-1}, \tag{4.67}$$

where the possible z values (one to infinity) determine the degree to which $\mathbf{A}^{-1}$ is approximated. The PPCG algorithm is therefore parameterized by the quantity z and shall, hereafter, be denoted as PPCG(z).

Since the matrix $\mathbf{K}^{-1}$ is needed only for the matrix vector products in (4.54) – (4.61) of the form

$$\underline{c} = \mathbf{K}_z^{-1} \underline{r}, \tag{4.68}$$

the product can be evaluated whenever it is needed and $\mathbf{K}^{-1}$ never explicitly stored (which represents a considerable saving on storage since $\mathbf{K}^{-1}$ will usually be denser than $\mathbf{A}$ itself). This is a great advantage over the ICCG method which requires additional storage equal to the storage required for the $\mathbf{A}$ matrix. PPCG(z) requires only the storage of an additional vector of length N over the basic CG algorithm.

The value of z should be user-specifiable since the matrix vector multiplication required to evaluate (4.68) is very expensive. For increasing values of z, there is a definite trade-off between:

1. the decrease in total execution time resulting from the decrease in the number of iterations needed to achieve a specified accuracy; and
2. the increase in total execution time resulting from the increased execution time of a single iteration for higher z values.

The choice for $\mathbf{M}^{-1}$ in (4.67) is of fundamental importance. It must be such that (4.66) holds true. An exceptional choice is to take

$$\mathbf{M}^{-1} = (DIAGONAL(\mathbf{A}))^{-1}, \tag{4.69}$$

since it allows any matrix-vector products involving $\mathbf{M}^{-1}$ in the evaluation of (4.68) to be replaced by a vector-vector product. With $\mathbf{M}^{-1}$ of this form, (4.66) is guaranteed to hold if $\mathbf{A}$ is strictly or irreducibly diagonally dominant (Varga [1965,73]). In particular, if $\mathbf{A}$ is a real $N \times N$ matrix, and $(a_{ij}) \leq 0$ for all $i = j$, then $\mathbf{M}^{-1}\mathbf{N}$ is nonnegative, irreducible, and convergent if (Varga [1965,84]):

1. $\mathbf{A}$ is nonsingular and $\mathbf{A}^{-1}$ is > 0; or
2. the diagonal entries of A are positive real numbers.

Matrices of this type arise in many cases of interest. Varga [1965,161-208] demonstrates that matrices with the above properties arise naturally from the finite-difference solution of elliptic partial differential equations.

It can be seen that the preconditioning algorithm presented here is only as good as the matrix-vector multiplication routine used to implement it. It is important that the chosen sparsity scheme allow for a very efficient routine to be coded. This point is especially important with parallel processors since the architecture will often limit the usable sparsity schemes with a resulting limitation in the options available for matrix-vector multiplication routines.

The implementation of PPCG(z) on parallel processors will involve compromises between sparsity schemes and multiplication routines.

4.2.4 Transputer Implementation

Since sparse matrix-vector multiplication is the most influential component of the PPCG algorithm, its implementation is wholly determined by this component algorithm. We saw in Section 3.3.2 that we can have an efficient multiplier on a linear array of transputers when the matrices are banded. Fortunately, the numerical methods for which PPCG is applicable (finite-differences and finite-elements) produce banded matrices. As long as the requirements for PPCG preconditioning are met, we can implement an efficient algorithm.

The vector operations also require a linear array of processors (see Section 3.2), so we are twice fortunate. Given the presence of the component algorithms, the only necessary constituent for a complete algorithm is a sparse storage scheme.

Since each transputer in the network is essentially a scalar processor, we can apply any of the linked-list sparsity schemes that have been developed for sequential processors (Zollenkopf [1971], Gentleman et al. [1976], Eisenstat et al. [1976]). The method adopted here stores each column as a linked list along with a pointer to the first location in a column. A particularly convenient scheme is used in which all storage (for pointers and coefficients) is contained in a single one-dimensional array. † Although not exploited at present, this would allow easy transferal of these matrices between transputers.

In order to make use of the principles presented in Chapter 3, the CG and PPCG solvers are implemented on a linear array of transputers. Each processor in the network is given an identical copy of the program, with logic to handle the special cases in communication requirements that arise at each end of the array. For example, being at the end of the array would imply that a processor not pass information to the right, and only accept information from the left (or however one wishes to define the directions).

† The key Occam techniques used in implementing the sparse matrix storage scheme were suggested by Andy Rabagliotti of INMOS Corporation.

Given the algorithms of Chapter 3, the coding of the equations defining the CG and PPCG algorithms is a straightforward substitution of procedure calls with additional logic to detect convergence. The only departure from Chapter 3 is that the scalar product algorithm is implemented on a linear array instead of a shuffle-exchange network. This is due to limitations in the available hardware which did not allow the simultaneous use of the two networks. Since the network used is small (8 processors), the difference will not be significant.

To test the algorithm, a test matrix was generated locally in each processor using the finite-difference CAE algorithm. In addition, the solver could be coupled with another physical modeller which generates sparse matrices such as the finite element method, or could receive all matrix data from the host. The latter case would incur some start-up overhead which is not considered here.

4.2.5 Results

We begin by constructing a theoretical model for execution time of the PPCG algorithm so that the behaviour of the algorithm can be studied for matrices larger than the specific problem used to test the implementation (and allowed by available hardware). This also allows characterization of the algorithm for larger numbers of processors.

In general, many iterations of the CG/PPCG algorithms are required to achieve a solution. The number of iterations is dependent upon the size of the system, as well as the conditioning of the matrix. Total execution time is therefore only useful in directly comparing CG to PPCG (or other preconditioning methods) where we are interested in the merit of the method as a whole.

For our purposes, the best way to characterize these algorithms is in terms of the execution time for a single iteration (T_{iter}). This gives a measure of the efficiency of the implementation, ignoring considerations of the specific problem at hand.

Consider the PPCG(z) algorithm of Sections 4.2.2 and 4.2.3. The following model makes use of the primitive execution times in Table 3.1 as weights, and is parameterized by five main factors:

1. the number of processors (N_p);
2. the matrix half-bandwidth W†;
3. the number of non-zero coefficients per column (N_{nz});
4. the number of columns stored per processor ($C_p = N/N_p$) where N is the problem size; and
5. the PPCG parameter z.

Table 4.1 gives the cost of the main components of the PPCG(z) algorithm in terms of these parameters, it being assumed that the requirements for nearest neighbour sparse matrix-vector multiplication outlined in Section 3.3.2 are met.

Table 4.1: Estimated cost for an iteration of PPCG(z) ($N_p \geq 2$). Adding up the terms in the cost column gives the execution time for an iteration of PPCG(z).

Operation	Cost
Matrix-Vector Mult.	$z[C_p N_{nz}(T_v^{\times} + T_s^{+}) + 2(W-1)T_v^{+} + (W-1)T_v^{X}]$
Vector Addition	$(z+1)[C_p T_v^{+}]$
Vector Subtraction	$z[C_p T_v^{-}]$
Vector Multiplication	$z[C_p T_v^{\times}]$
Scale Vector	$3[C_p T_v^{sc}]$
Vector Assignment	$1[C_p T_v^{=}]$
Scalar Product	$3[C_p N_{nz}(T_v^{\times} + T_s^{+}) + N_p T_s^{X}]$

† The calculation is performed for symmetric matrices for which $W = P = Q$ (as defined by Figure 3.10) and thus have bandwidth $2W - 1$.

As an example of the use of the PPCG algorithm, consider the finite-difference solution of Laplace's equation

$$-\nabla^2\phi = 0, \tag{4.70}$$

in the region depicted in Figure 4.1. We proceed by laying a regular grid of nodes $\phi_{i,j}$ over the region (including the boundaries) and forming a system of equations using the finite-difference approximation of the Laplacian (Forsythe and Wasow [1960])

$$(-\phi_{i-1,j} - \phi_{i,j-1} + 4\phi_{i,j} - \phi_{i,j+1} - \phi_{i+1,j}) = 0. \tag{4.71}$$

Here i and j represent the row and column index of a node (with $\phi_{1,1}$ in the top left corner of the region).

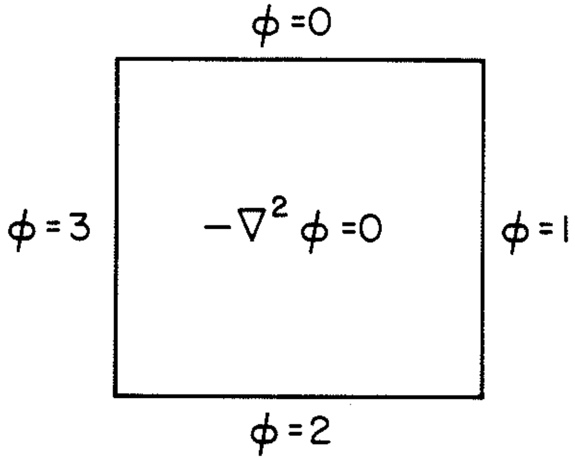

Figure 4.1: Example finite-difference problem.

Applying (4.71) to the nodes generates a system of equations which can be solved for the unknown ϕ. The traditional approach for solving problems of this sort on serial machines, however, is not to form this matrix at all. The coefficients of the matrix are generated as needed in an iterative scheme. Successive over-relaxation (SOR), for example, uses (4.70) directly to compute the next value of $\phi_{i,j}$ given existing values of the other ϕ's in the formula (Forsythe and Wasow [1960]). Newly computed values are used immediately in subsequent node evaluations. This process is iterated over all the nodes until the newly calculated ϕ cease to differ from the old ones by some prescribed measure – at which time the solution is obtained.

Parallelization of the SOR iterative sequence is difficult since values generated in the current step are used in the current step. This enforces an ordering on the calculations, and makes the algorithm highly sequential. In simple cases such as we have here, it is possible to "colour" the mesh into disjoint sets of nodes (the so called red-black ordering) which do not affect one another in an iterative sweep (Barlow and Evans [1982]). In this case, applying (4.71) to the red nodes produces the next set of black nodes. Similarly, the black nodes produce the next set of red nodes. Since the calculations involving a given colour are completely independent, a highly efficient algorithm is possible.

In more complicated problems with varying boundary conditions , a different operator, or arbitrary boundaries, the task becomes more difficult. First of all, considerable logic is needed to determine which equation applies to a particular node (this is also a problem in the sequential case). Secondly, given the right equation, we must ensure that we are able to colour connecting nodes differently. Any, or all, of the above complications might cause us to reach an impasse, creating the need for multi-colour schemes (Adams and Ortega [1982]).

To avoid the above complications, the entire matrix can be formed as we have done here. This approach has been taken by Wong and Jiang [1977], and Wong [1977], who use a method similar to PPCG to solve finite-difference problems with up to 65,000 unknowns on a CYBER 205 vector supercomputer (Hockney and Jesshope [1981]).

It is interesting to note that vector supercomputer implementations of PPCG suffer from limitations similar to the current algorithm, in that they must also restrict themselves to structured matrices (i.e. banded) to obtain efficient matrix-vector multiplication. The most common method, due to Madsen et al. [1976], relies on the matrix having a well defined diagonal band structure.

For the problem in Figure 4.1, equation (4.71) is applied only to the interior nodes, since the boundary nodes are specified by the boundary conditions (other types of boundary conditions have to be catered for differently). Furthermore, for nodes immediately next to the boundary nodes, (4.71) must be modified to reflect the fact that the value of ϕ for some of the nodes in the operator is already known by virtue of the boundary conditions. Consider the equation for $\phi_{2,2}$. In this case, we know that $\phi_{1,2} = 0$ and $\phi_{2,1} = 3$, so that equation (4.71) becomes

$$4\phi_{2,2} - \phi_{2,3} - \phi_{3,2} = 3. \tag{4.72}$$

Generating equations for all of the interior nodes results in the following matrix system (using a 5 × 5 grid with 9 interior nodes):

$$\begin{pmatrix} 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 4 & -1 & 0 & -1 & 0 & 0 & 0 \\ -1 & 0 & -1 & 4 & -1 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & -1 & 4 & -1 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 & -1 & 4 & -1 & 0 & -1 \\ 0 & 0 & 0 & -1 & 0 & -1 & 4 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 \end{pmatrix} \begin{pmatrix} \phi_{2,2} \\ \phi_{2,3} \\ \phi_{2,4} \\ \phi_{3,2} \\ \phi_{3,3} \\ \phi_{3,4} \\ \phi_{4,2} \\ \phi_{4,3} \\ \phi_{4,4} \end{pmatrix} = \begin{pmatrix} 3 \\ 0 \\ 1 \\ 3 \\ 0 \\ 1 \\ 5 \\ 2 \\ 3 \end{pmatrix}. \tag{4.73}$$

For a grid with N_s nodes on each side, a system with $N = (N_s - 2) \times (N_s - 2)$ unknowns is produced (with an associated matrix of size $N \times N$). Important for our purposes, is that the matrix is banded, with bandwidth $(2(N_s - 2) - 1)$. As we have mentioned, the matrix meets the requirements for PPCG convergence and efficient matrix-vector multiplication.

Table 4.2 compares the execution time of a CG/PPCG(z) iteration estimated by the computation time model in Table 4.1 (for both the T-414 and T-800 transputers) with an implementation of CG/PPCG(z) running on an 8-processor transputer array. It should be emphasized that the times given are for a *single* iteration, and are not the execution times needed to reach the final solution. These results are shown for a 66×66 node problem (which has 4096 unknowns). Also shown are results given in Allen [1983] which show the execution time per iteration for an identical problem running on a DAP processor array (Hockney and Jesshope [1981]) and on an Amdahl 5850 mainframe.

Table 4.2: T_{iter} estimates and results for PPCG(z) ($N_p = 8$). The time is given in milliseconds.

System	CG	PPCG(2)	PPCG(4)
T-414 Array	251	461	825
T-414 (Est.)	245	447	772
T-800 (Est.)	25	48	81
DAP	33	64	125
Amdahl	52	90	162

The first thing to notice from Table 4.2 is the accuracy of the estimated T-414 execution time relative to the time obtained on an actual array of transputers. Generally, the estimated values are within 7% of the actual values. This gives us some license to speculate on the behaviour of the algorithm on larger networks and problem sizes. While this is important in its own right, we are also given some confidence in the estimates for the T-800 transputer, allowing us to discuss the expected results for it as well.

Table 4.3: Estimated T_{iter} for PPCG(2) for different N_p. T_{iter} is given in milliseconds, and E_{N_p} in %.

N_p	T_{iter} (T-414)	E_{N_p} (T-414)	T_{iter} (T-800)	E_{N_p} (T-800)
4	884	99	94	98
8	447	98	48	96
16	229	96	26	91
32	123	89	15	76
64	74	74	12	49

Given this, we can tabulate estimates of execution time and efficiencies for other network sizes for both the T-414 and the T-800 transputers. The results are given in Table 4.3 for an array of processors executing the PPCG(2) algorithm.

We can see that in all cases, the efficiencies are quite high, only faltering when there are 64 processors (the maximum allowable for a problem of this size and bandwidth – while still maintaining nearest neighbour communication). Thus it represents the maximum in communication and the minimum in computation. We are consoled by the fact, then, that the efficiencies will rise when the problem size is increased.

The algorithm possesses at least one shortcoming. This example problem is admittedly quite special in that the bandwidth is quite small in comparison to the number of unknowns. Given different finite-difference operators, or more complicated shaped regions and grids, this may not be the case. It would then be harder to maintain nearest neighbour communication, causing efficiency to suffer. This problem can be controlled somewhat by judicious node numbering, but there may be cases where that may not be enough. Even in the event that next-nearest neighbour communication is required, the algorithm should still be quite efficient. What has to be avoided is having a bandwidth so large that values have to be passed across a significant portion of the array.

This example shows how transputers can provide high power at a reasonable cost. As the estimated results in Figures 4.2 and 4.3 show, a transputer array with 8 T-800 processors could execute the finite-difference problem faster than either the DAP or the Amdahl. Since the cost of these computers is on the order of millions of dollars, and the cost of an 8 transputer array is currently around $25,000, we can see that there is a considerable difference in the power/cost ratio. Thus, for this problem at least, we have achieved to goals set forth in the introduction.

CHAPTER V

PARALLELIZATION OF THE BEM ALGORITHM

This chapter presents a parallel implementation of a two-dimensional Boundary Element Method (BEM). This method was chosen as an example of a CAE algorithm, since it is currently being used in a commercial CAE system that is used to aid in the design of printed circuit boards (Poltz and Wexler [1986]).

The partitioning of the BEM algorithm described here is dependent upon a number of details specific to the algorithm used. Most important of these is the way in which one chooses to describe the geometry of the structures being modelled. The particular properties of the modelling method used here (see Section 5.2.2) cannot be guaranteed to carry over to other geometric discretization techniques such as cubic spline based modelling. As such, this chapter in no way describes a general attack on such problems. Its purpose is to present a particularly innovative partitioning of a CAE algorithm.

5.1 MANITOBA: A Boundary Element Accelerator

In this section, the MANitoba Integrated Transputer/Occam Boundary element Accelerator (MANITOBA)† is described. ‡ MANITOBA necessarily integrates two algorithms, because the end result of applying the BEM to a problem is a system of linear equations which must be solved for the unknowns. Since the execution times for these two separate phases are comparable, an implementation cannot consider only one, but must address both problems simultaneously. It is in cases like this that the art of parallel programming comes to the fore. The algorithms must be implemented in a complementary fashion, so that neither is unduly crippled by the architecture needed for the other.

Once it was decided that the transputer and Occam would be used, it was necessary to determine how the two algorithms could best be partitioned to allow efficient co-implementation. Decisions about the partitioning were made within the bounds of a number of competing constraints.

The first and perhaps most important constraint was that the algorithm could be adapted (or would adapt itself) to transputer networks of varying sizes. This is important because it is not reasonable to limit users to some standard configuration if it does not meet their performance requirements. If more power is needed, then more processors could be added to obtain the required performance characteristics.

Also important was the requirement that both the matrix generation and matrix solution phases of the algorithm be implemented on the same network topology. This avoids any waste or inefficiency involved in shoe-horning an algorithm onto an incompatible network configuration. It also avoids the possibility that some processors might have to sit idle if they are not part of the topology required for the other algorithm.

† In the spirit of self-reference (Hofstadter [1980]) – a self-referential acronym.

‡ MANITOBA was presented at the Second Occam Users Group meeting in Santa Clara CA, March 1987.

The last major criterion revolves around global communication in transputer networks. Since transputer networks are best at local communication and degrade greatly in efficiency if much random global communication takes place, we must avoid it if at all possible. Failing this, the required communication must be regular and predictable so that communication and computation can be overlapped in different parts of the array. After study of various matrix solution algorithms and the BEM algorithm it was decided that a linear array of processors as shown in Figure 5.1 was best suited to accommodating both the algorithms and the design criteria.

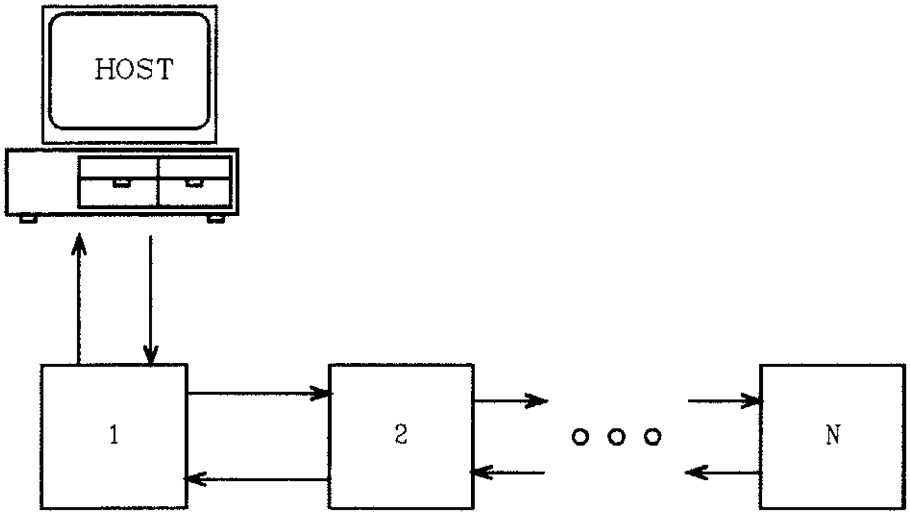

Figure 5.1: MANITOBA architecture.

Aside from allowing implementation of the BEM and matrix solution algorithms, the linear array has a side benefit in keeping with the goals of this research. If one is targeting the accelerator for small workstations, cost is a major consideration. Given constraints of cost, it is not feasible to offer a very large number of processors, so it is better to pick the architecture that is most versatile for the problems at hand.

MANITOBA is divided into two sections. The first section, which runs on a transputer located in the host computer, is responsible for communicating with the host computer and feeding data from the host to the computational array. It also maintains a protocol with the BEM array that allows it to control its operation.

The second part of MANITOBA, which is distributed over a linear array of transputers attached to the control processor, implements the BEM and matrix solution algorithms. Each processor is assigned a set of contiguous columns over which it is responsible for both matrix generation and solution as shown in Figure 5.2. Every processor $\{1, 2, \ldots, N\}$ contains a copy of the matrix generation and solution algorithms as well as a complete copy of the problem specification data so that each has all the resources necessary to carry out its task.

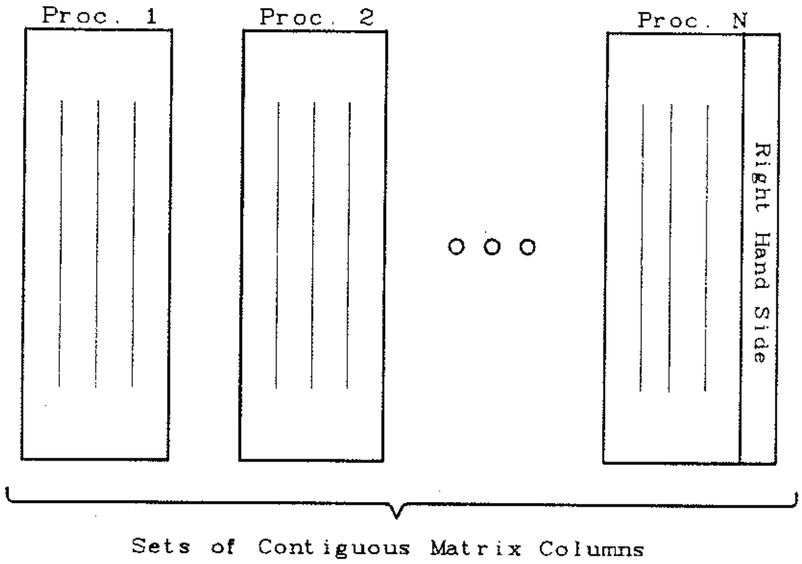

Figure 5.2: Matrix partitioning on an array of processors.

As with any solution to a complex problem, an acceptable solution represents a compromise in some respects. For example, the chosen matrix solution method was Gaussian elimination. This algorithm has the virtue of a rather obvious parallelism, but also requires global communication of coefficients of elimination. Fortunately, this global communication is very regular and can be overlapped with computation in a pipelined fashion, making for an efficient algorithm.

The matrix solution algorithm was the fundamental driving force behind choosing a linear array since having an efficient matrix solver is important to algorithm performance. Given this, it was necessary to put restrictions on the geometrical description given to the BEM matrix generator so that global communication would be minimized. Details of this are discussed in Section 5.2.4.

5.2 The Boundary Element Method

The focus of this chapter is the Boundary Element Method (BEM) which is a general technique for the solution of boundary value problems posed in an integral equation framework (Jaswon and Symm [1977]). It is applicable to many kinds of problems in both two and three dimensions.

For the purposes of this dissertation, two-dimensional electrostatic boundary-value problems are solved in piecewise homogeneous regions (i.e. a two-dimensional plane segmented into regions of differing dielectric permittivity ϵ that is constant throughout each region). The method used was first described by McDonald et al. [1974]. Further treatment of the method appeared in Jeng and Wexler [1977], Jeng and Wexler [1978], Lean and Wexler [1982], and Klimpke [1983]. The spirit of the implementation described here comes from these works, but has been extended to accommodate parallel partitioning on a linear transputer array.

5.2.1 BEM Theory

Consider a three-region problem consisting of the whole of exterior space (R_e) with free-space dielectric permittivity ϵ_o, and two interior regions R_1 and R_2 (or R_i generally) with permittivities ϵ_1 and ϵ_2 as shown in Figure 5.3. The labels B_e and B_i are the exterior and interior interface boundaries respectively, while $\vec{r}$ is an observation point and $\vec{r}'$ is the location of charges distributed on the various boundaries. Moreover, the *outward* pointing normals for the interior and exterior regions are denoted as $\hat{n}_i$ and $\hat{n}_e$ respectively, with *outward* pointing normals on the interface between regions 1 and 2 being denoted $\hat{n}_1$ and $\hat{n}_2$.

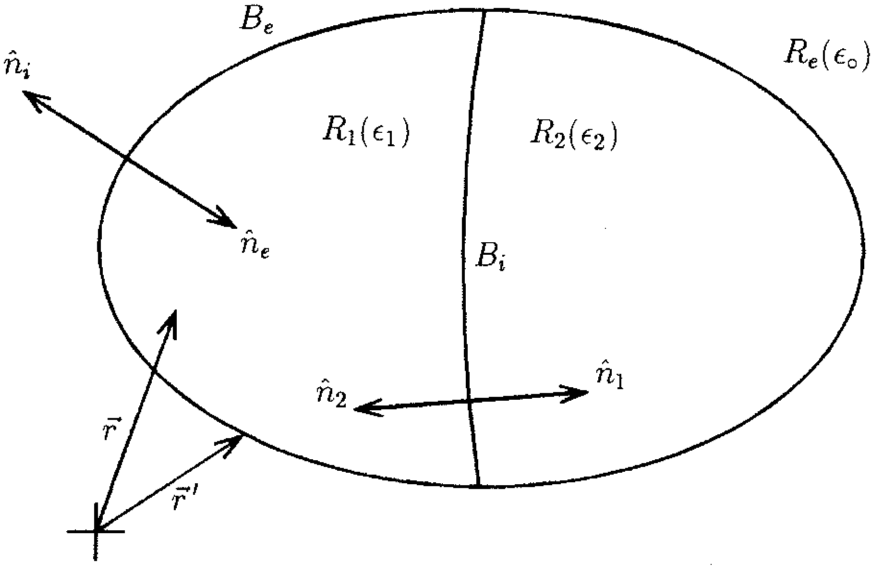

Figure 5.3: Region, surface, and normal definitions for multi-media regions.

The boundary element analysis that follows seeks the charge distribution along the problem boundaries shown in Figure 5.3. The effect is to replace the bounded regions of differing permittivity with *equivalent* charge distibutions on their boundary. This allows the problem to be viewed as a free-space distribution of charges, and in turn allows the use of the free-space Green function. Once the boundary charges have been determined, they can be used to calculate the potential throughout the two-dimensional plane.

The solution of Laplace's equation

$$\nabla^2\phi = 0 \tag{5.1}$$

in this two dimensional plane is sought. Remembering that in two-dimensions, the field of a point charge has logarithmic variation, consider the field due to a collection of N point charges q_i whose locations are given by various vectors $\vec{r}'_i$. Elementary superposition allows us to express the potential at some observation point $\vec{r}$ with respect to some reference point $\vec{r}_o$ as the sum

$$\phi(\vec{r}) - \phi(\vec{r}_\circ) = -\sum_{i=1}^{N} \frac{q_i}{2\pi\epsilon_\circ}(ln|\vec{r} - \vec{r}_i{}'| - ln|\vec{r}_\circ - \vec{r}_i{}'|). \tag{5.2}$$

If instead of discrete charges, we have a continuous charge density distribution $\sigma(\vec{r}')$ along a boundary curve B, then the sum is replaced by an integral and becomes

$$\phi(\vec{r}) - \phi(\vec{r}_\circ) = -\frac{1}{2\pi\epsilon_\circ}\int_B \sigma(\vec{r}')[ln|\vec{r} - \vec{r}'| - ln|\vec{r}_\circ - \vec{r}'|]dr'. \tag{5.3}$$

Unlike the three-dimensional case, the potential has logarithmic variation. Thus we cannot use infinity as a reference potential, and must instead choose $\vec{r}_\circ$ to be finite. Letting $\phi(\vec{r}_\circ) = C$ yields

$$\phi(\vec{r}) = -\frac{1}{2\pi\epsilon_\circ}\int_B \sigma(\vec{r}')[ln|\vec{r} - \vec{r}'| - ln|\vec{r}_\circ - \vec{r}'|]dr' + C. \tag{5.4}$$

Consider the second term of equation (5.4) for a moment. If we choose $\vec{r}_\circ \gg \vec{r}'$, then $ln|\vec{r}_\circ - \vec{r}'|$ is essentially a constant. We can therefore write the second term as

$$\frac{ln|\vec{r}_\circ - \vec{r}'|}{2\pi\epsilon_\circ} \int_B \sigma(\vec{r}')dr'.$$

If we impose the physically realistic constraint that the total charge be zero

$$\int_B \sigma(\vec{r}')dr' = 0, \tag{5.5}$$

we are left with

$$\phi(\vec{r}) = \frac{1}{\epsilon_\circ} \int_B G(\vec{r}|\vec{r}')\sigma(\vec{r}')dr' + C. \tag{5.6}$$

Here, $G(\vec{r}|\vec{r}')$ is the two-dimensional Green function

$$G(\vec{r}|\vec{r}') = -\frac{1}{2\pi} ln|\vec{r} - \vec{r}'|, \tag{5.7}$$

and plays the role of kernel for the Fredholm integral equation of the first kind.

Equation (5.6) describes the potential for Dirichlet boundary conditions. For Neumann boundaries, the derivative of the potential is specified. This is modelled by the Fredholm equation of the second kind (Stakgold [1979])

$$\phi'(\vec{r}) = \frac{1}{\epsilon_\circ} \int_B G_i{}'(\vec{r}|\vec{r}')\sigma(\vec{r}')dr' - \frac{\sigma(\vec{r})}{2\epsilon_\circ}, \tag{5.8}$$

in which ϕ' and G' are the derivatives of ϕ and G with respect to the outward normal to the boundary.

For the case where there is an internal interface between materials with different ϵ, the appropriate equation is (Jeng and Wexler [1978], Lean and Wexler [1982])

$$\left(\frac{\epsilon_1 - \epsilon_2}{\epsilon_\circ}\right) \int_B \sigma(\vec{r}')G_i{}'(\vec{r}|\vec{r}')dr' - \left(\frac{\epsilon_1 + \epsilon_2}{2\epsilon_\circ}\right)\sigma(\vec{r}) = 0. \tag{5.9}$$

While we are seeking $\phi(\vec{r})$ throughout the plane, it is not the unknown. Instead, we must first find the charge distribution $\sigma(\vec{r}')$ on the boundaries which will give rise to the spatial potential field under the prescribed Dirichlet, Neumann, and interface boundary conditions. Once the charge distribution is known, it can be substituted back into Equation (5.4) to calculate $\phi(\vec{r})$.

With the above equations in hand we are able to solve for the potential in the plane in terms of the boundary charges necessary to produce that potential field. The variational process used to obtain the boundary charges is similar to that used to solve finite element problems. Consider equation (5.4) under Dirichlet boundary conditions $\phi(\vec{r}) = g(\vec{r})$. The integral operator for this case is

$$K = \frac{1}{\epsilon_\circ} \int_B G(\vec{r}|\vec{r}')dr' + C, \tag{5.10}$$

which produces the operator equation

$$K\sigma(\vec{r}) \;=\; g(\vec{r}). \tag{5.11}$$

Defining the inner product

$$\langle u, v \rangle = \int_B uv\, dr, \tag{5.12}$$

the solution of (5.11) is accomplished via a variational approach in which the stationary point of the "energy" functional (Jeng and Wexler [1977])

$$F = \langle K\sigma, \sigma \rangle - 2\langle \sigma, g \rangle, \tag{5.13}$$

coincides with the desired solution when the operator K is self adjoint

$$\langle K\sigma, \tau \rangle = \langle \sigma, K\tau \rangle. \tag{5.14}$$

Substituting (5.6) into (5.13), we obtain

$$F = \int_B \sigma(\vec{r}) \int_{B'} \frac{1}{\epsilon_0} G(\vec{r}|\vec{r}') \sigma(\vec{r}') dr' dr + \int_B \sigma(\vec{r}) dr C - 2 \int_B \sigma(\vec{r}) g(\vec{r}) dr. \quad (5.15)$$

To minimize this functional, a Rayleigh–Ritz procedure is used (Harrington [1968]) in which σ is expressed as the sum of orthogonal expansion functions with unknown coefficients. This generates a matrix system which can be solved for the coefficients of expansion, yielding the value of σ along the boundaries.

Let there be expansion functions $\{\alpha_1(\vec{r}), \alpha_2(\vec{r}), \ldots, \alpha_n(\vec{r})\}$ which are parameterized along the path of integration by coefficients of expansion $\{\sigma_1, \sigma_2, \ldots, \sigma_n\}$. We can then write σ as

$$\sigma(\vec{r}) = \sum_{i=1}^{n} \sigma_i \alpha_i(\vec{r}), \quad (5.16)$$

which after forming column vectors $\underline{\alpha}$ and $\underline{\sigma}$ (5.16) may be expressed in vector notation as

$$\sigma(\vec{r}) = \underline{\alpha}^T \underline{\sigma} = \underline{\sigma}^T \underline{\alpha}. \quad (5.17)$$

Equation (5.15) then becomes

$$F = \underline{\sigma}^T \int_B \underline{\alpha}(\vec{r}) \int_B \frac{1}{\epsilon_0} G(\vec{r}|\vec{r}') \underline{\alpha}^T(\vec{r}') dr' dr \underline{\sigma} + \underline{\sigma}^T \int_B \underline{\alpha}(\vec{r}) dr C - 2\underline{\sigma}^T \int_B \underline{\alpha}(\vec{r}) g(\vec{r}) dr. \quad (5.18)$$

The functional F is minimized by differentiating with respect to the variational parameter $\underline{\sigma}$ and setting the result equal to zero which yields

$$\int_B \underline{\alpha}(\vec{r}) \int_B \frac{1}{\epsilon_0} G(\vec{r}|\vec{r}') \underline{\alpha}^T(\vec{r}') dr' dr \underline{\sigma} + \int_B \underline{\alpha}(\vec{r}) dr C = \int_B \underline{\alpha}(\vec{r}) g(\vec{r}) dr. \quad (5.19)$$

Equation (5.19) can be written as the matrix system

$$\mathbf{S}\underline{\sigma} = \underline{b}, \tag{5.20}$$

where

$$s_{ij} = \int_B \alpha_i(\vec{r}) \int_B \frac{1}{\epsilon_\circ} G(\vec{r}\,|\vec{r}\,')\alpha_j(\vec{r}\,')dr'dr \tag{5.21}$$

and

$$b_i = \int_B \alpha_i(\vec{r})g(\vec{r})dr. \tag{5.22}$$

Equation (5.19), and its accompanying constraint equation

$$\int_B \underline{\alpha}^T(\vec{r})dr\underline{\sigma} = 0 \tag{5.23}$$

apply to all the Dirichlet boundaries in the problem.

Analysis similar to that producing (5.19) can be applied to the equations governing the Neumann and interface boundaries. In the case of Neumann boundaries, the analysis produces

$$\int_B \underline{\alpha}(\vec{r}) \int_B \frac{1}{\epsilon_\circ} G'(\vec{r}\,|\vec{r}\,')\underline{\alpha}^T(\vec{r}\,')dr'dr\underline{\sigma} - \frac{1}{2\epsilon_\circ}\int_B \underline{\alpha}(\vec{r})\underline{\alpha}^T(\vec{r})dr\underline{\sigma} = \int_B \underline{\alpha}(\vec{r})\phi'(\vec{r})dr, \tag{5.24}$$

where G' represents the normal gradient of the Green function with respect to the boundary in question and is given by

$$\nabla G \cdot \hat{n} = \frac{1}{2\pi}\frac{[(x'-x)|\hat{n}_x| + (y'-y)|\hat{n}_y|]}{[(x-x')^2 + (y-y')^2]}. \tag{5.25}$$

The only restriction implied by (5.24) is that the solution will only be valid in the region into which the normal to the Neumann surface is pointing. For example, if the surface normals of the Neumann boundaries all point into the exterior region, the solution obtained will be valid there. Conversely, if they point into the interior region, the solution will only be valid in that region.

For interface boundaries, (5.9) becomes

$$\left(\frac{\epsilon_1 - \epsilon_2}{\epsilon_\circ}\right) \int_B \underline{\alpha}(\vec{r}) \int_B G_1{}'(\vec{r}\,|\vec{r}\,') \underline{\alpha}^T(\vec{r}\,') dr' dr \underline{\sigma} \\ - \left(\frac{\epsilon_1 + \epsilon_2}{2\epsilon_\circ}\right) \int_B \underline{\alpha}(\vec{r}) \underline{\alpha}^T(\vec{r}) dr \underline{\sigma} = 0. \quad (5.26)$$

Here, the normal derivative of the Green function is calculated with respect to the normal pointing into region 2 (the outward normal of region 1).

Equations (5.24) and (5.26) accumulate into the system matrix and forcing vector in a fashion similar to the Dirichlet case (equations (5.21) and (5.22)).

Given the above equations for Dirichlet, Neumann, and interface boundaries, most of the machinery is in place for the solution of boundary element problems. The only detail left is to describe the expansion functions used to model the geometry and the charges. These dictate how the boundaries are subdivided and the matrix equation accumulated.

5.2.2 Surface and Charge Modelling

Applying the Rayleigh-Ritz variational method to (5.15) expresses the boundary charges in terms of orthogonal expansion functions with unknown coefficients. Solving for the coefficients gives us the boundary charges, which can in turn be used to calculate potentials in problem region(s).

It turns out to be convenient to carry this same framework over to the geometric model of the boundary surfaces so that both are modelled in the same way and to the same polynomial order. Formally this is called an isoparametric representation.

However, it is hard to imagine how one could model a complicated geometry with a single polynomial, since such geometries are made up of combinations of many lines and curves with geometric singularities. To get around this problem, the surfaces are broken up into many patches or *elements* (hence the name Boundary Elements). A polynomial is fitted over each patch separately, with the control points (called *nodes*) defining the path of the element on the two-dimensional plane.

The charges are then modelled isoparametrically by specifying their variation along each element in terms of their values at the nodes used to model the geometry. Interpolation between the nodal values gives the charge anywhere along the boundary.

Consider Figure 5.4, in which some function $f(x)$ is known at three points (nodes) in the range $[x_1, x_3]$. We shall call this set of three nodes an element. If we assume that the function varies quadratically we may construct a function to interpolate it over the whole range considered.

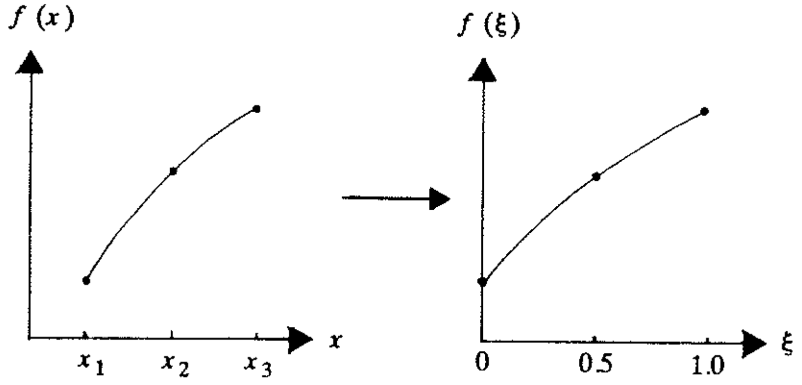

Figure 5.4: Mapping of a boundary element from global to local space.

This is done by first mapping the element into what is called the standard simplex. In this case, the simplex corresponds to the interval [0,1] on the ξ axis, where ξ is our parametric coordinate. We define a set of orthogonal quadratic polynomials α_i over this interval (called local or ξ space) which take on values

$$\alpha_i(\xi) = \begin{cases} 1 & \text{at node } i; \\ 0 & \text{at all other nodes.} \end{cases} \tag{5.27}$$

We may then express $f(\xi)$ in terms of its value f_i at the three nodes as

$$f(\xi) = \sum_{i=1}^{3} \alpha_i(\xi) f_i. \tag{5.28}$$

Note, that while we know f in terms of ξ, it is not easy to obtain a particular $f(x)$. This restriction is of no consideration here, however. The interpolation merely gives a parametric representation of the functions involved which can be used to evaluate the integral equations in a simple fashion.

The particular expansion functions used here are called the Lagrangian shape (or interpolation) functions (Wexler [1980], Lean and Wexler [1985]). Such functions may be defined for any order of approximation, but are restricted to quadratic representation here. In terms of the local space coordinate they are

$$\begin{aligned} \alpha_1 &= 2\xi^2 - 3\xi + 1, \\ \alpha_2 &= 4(\xi - \xi^2), \\ \alpha_3 &= 2\xi^2 - \xi. \end{aligned} \tag{5.29}$$

This same formulation is used to model both geometry and charge distribution for the algorithm. The charge distribution representation follows the description of $f(\xi)$ above, and allows a quadratic representation of the charge along a curve in space.

Curved boundaries are modelled in a similar way, but have a parametric representation for both the x and y coordinate. Given a curve in space , it is subdivided along its length into boundary elements (as in Figure 5.5) which are defined by three nodes $\{(x_1,y_1),(x_2,y_2),(x_3,y_3)\}$ along the curve. The location of any point on a boundary element is described parametrically as

$$x(\xi)=\sum_{i=1}^{3}\alpha_i(\xi)x_i \tag{5.30}$$

and

$$y(\xi)=\sum_{i=1}^{3}\alpha_i(\xi)y_i. \tag{5.31}$$

Thus a particular ξ on the interval [0,1] will generate a point in space (x,y) on the boundary element.

Since we are going to evaluate line integrals on the simplex element, it is desirable to know the Jacobian of the transformation (the magnitude of the incremental vector $d\vec{r}$) as a function of ξ. Given the above parametric representations for a boundary element defined by three real space coordinates, the Jacobian is defined such that $|d\vec{r}|=J\,d\xi$, which yields

$$J=\sqrt{\left(\frac{\partial x}{\partial\xi}\right)^2+\left(\frac{\partial y}{\partial\xi}\right)^2}, \tag{5.32}$$

where

$$\frac{\partial x}{\partial\xi}=\sum_{i=1}^{3}\frac{\partial\alpha_i(\xi)}{\partial\xi}x_i \tag{5.33}$$

and

$$\frac{\partial y}{\partial\xi}=\sum_{i=1}^{3}\frac{\partial\alpha_i(\xi)}{\partial\xi}y_i. \tag{5.35}$$

Also important to the proper evaluation of the boundary equations is the unit normal vector to a boundary element. By convention, we define the unit normal as pointing to the left when traversing the boundary element from node 1 to node 3. Specifically, the unit normal is defined as

$$\hat{n} = -\frac{1}{J}\left(\frac{\partial y}{\partial \xi}\hat{\imath} - \frac{\partial x}{\partial \xi}\hat{\jmath}\right). \tag{5.36}$$

5.2.3 Matrix Generation

Given the results of the two previous sections, we can now describe the matrix generation process in detail. The data given to the matrix generator consists of a list of boundary elements describing geometrically the various boundaries and interfaces present in the problem. Each element is assigned a boundary or interface condition and is defined by three nodes on the boundary as described in the previous section. Finally, a list indicating the (x, y) coordinates of each node is given. Each node is indexed by a unique number that is in one-to-one correspondence with the unknown coefficients of expansion for the charge along the boundary. Figure 5.5 shows a sample geometric discretization for a parallel plate capacitor.

With the problem discretized in this fashion, the task is to evaluate equations (5.19), (5.24), or (5.26) on the appropriate boundaries. Since the boundaries are discretized, so too are the evaluations of the integrals. Thus the limits of integration extend only over one element at a time, but are evaluated for all elements.

In the case of the double integral, both the inner and outer integrals are evaluated in this fashion. If there are N elements, the double integral must be evaluated N^2 times to perform the required integration over all of the boundaries $(1, 2 \ldots, N$ on the inner integral, and $1, 2, \ldots, N$ on the outer integral). In this way every element is integrated "against" every other element and itself (the self-element integration). When performing these integrations, the appropriate equation

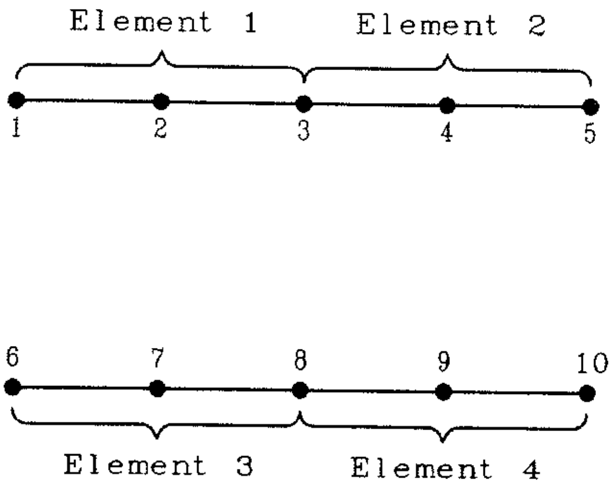

Figure 5.5: Boundary element discretization of a parallel plate capacitor.

(Dirichlet, Neumann, or interface) is chosen according to the boundary condition on the element used in the inner integration.

Note that, when performing the self-element integration, the Green function is singular (i.e. $\vec{r}$ and $\vec{r}'$ coincide, and a logarithmic singularity results). This situation requires special care in order to obtain accurate results. While the necessary techniques have been implemented in the program, they are not important to the present discussion. Details of the process are discussed by Lean [1981], Klimpke [1983], and by Lean and Wexler [1985].

Considering equation (5.19), let the inner integral be evaluated over element E^1 (defined by nodes N1, N2, and N3) and let the outer integral be evaluated over element E^2 (defined by nodes N3, N4, and N5). Applying the element-simplex transformation from Section 5.2.2 to (5.19) we obtain

$$\int_0^1 \underline{\alpha}(\xi) \int_0^1 \frac{1}{\epsilon_0} G(\vec{r}(\xi),\vec{r}(\xi'))\underline{\alpha}^T(\xi')J(\xi)'d\xi' J(\xi)d\xi\underline{\sigma} + \int_0^1 \underline{\alpha}(\xi)J(\xi)d\xi C = \int_0^1 \underline{\alpha}(\xi)g(\xi)J(\xi)d\xi, \quad (5.36)$$

where J and J' are the Jacobians for the outer and inner integrals respectively.

The important concept to be gleaned from (5.36) is the process by which matrix coefficients are accumulated, for it is the properties of this process which dictate how the BEM algorithm can be partitioned. Remembering the vector nature of $\underline{\alpha}$ and $\underline{\sigma}$, it can be seen that the multiplication of the row vector and column vector present in the double integral produces a 3 × 3 matrix of coefficient values. Thus the evaluation of (5.36) must be performed for each of these terms. In the present case we have

$$\underline{\alpha}_{OUT}\,\underline{\alpha}_{IN}^{T} = \begin{pmatrix} \alpha_3 \\ \alpha_4 \\ \alpha_5 \end{pmatrix} \begin{pmatrix} \alpha_1 & \alpha_2 & \alpha_3 \end{pmatrix} = \begin{pmatrix} (3,1) & (3,2) & (3,3) \\ (4,1) & (4,2) & (4,3) \\ (5,1) & (5,2) & (5,3) \end{pmatrix}. \qquad (5.37)$$

The first index in the ordered pairs of (5.37) indicates the row that the calculated coefficient will sum into, while the second coefficient indicates the column. Thus, for every element pair, the nine values resulting from the evaluation of (5.36) are summed into matrix locations as shown above.

After the element integrations have been performed, the integrations for the constraint equation (5.23), the C constant, and the right-hand side are performed. The results of these integrations are then placed into appropriate matrix locations giving, finally, the complete system matrix which can be solved for the charge σ and the constant C.

5.2.4 Parallel Matrix Generation

Parallel matrix generation must be done within the context of matrix inversion. The Gaussian elimination solver described in Chapter 4 requires that each processor in the array contain a set of contiguous matrix columns. Furthermore, we require that the columns be contiguous across the array of processors (column numbers must increase as we go from the beginning to the end of the processor array). Any generation scheme must respect this requirement, lest we suffer an inordinate penalty moving data into the required format.

Since the element-element integrations of equations (5.19), (5.24), and (5.26) are completely independent, the BEM algorithm has an obvious parallelism. One just divides the task of performing all of the integrations equally between all of the processors. This obvious solution does not work unless care is taken, however, because it ignores important details of interaction between the matrix accumulation process, the matrix solution algorithm, and the underlying parallel processing architecture.

For example, consider the implications of equation (5.37). Given two elements with arbitrarily numbered nodes, we see that the element-element integrations have the potential to generate coefficients that sum anywhere in the S-matrix. Since the architectural design has specified that a processor handles only a small set of contiguous columns of the S-matrix, it is highly probable that a processor will have to communicate a large number of coefficient values to other processors. Thus, while a simple partitioning of the work of integration will give near perfect speedup efficiency, it introduces unacceptable global communication costs. The problem, then, is to seek ways to minimize the global communication.

Careful examination of equation (5.37) reveals that the global node numbers of the elements on the inner loop of the double integral determine the matrix column that a coefficient will get summed into. If a processor had only to process nodes in its inner loop whose index numbers corresponded with the matrix columns it was accumulating, global communication would be eliminated. It would be impossible for a processor to generate coefficients that fall outside of its responsibilities. Note, that the outer integration loop still ranges over all of the elements, so it is necessary for each processor to contain data describing each of the elements.

We can arrange for this to happen by numbering the elements and nodes appropriately. All that is required is that the index numbers of the nodes comprising an element steadily increase as the element number increases (with the exception that the index number of node 3 of element i can equal the index number of node 1 of element $(i+1)$). If one has control over the element generation process, it can be constrained to produce data in this fashion. Failing that, the nodes could be renumbered. This does not restrict the geometries in any manner, but does require care when constructing the boundary element mesh. An example of such a data set is shown in Table 5.1.

Table 5.1: Example of node numbering required for parallel partitioning. The global index numbers of the nodes defining the elements are given.

Element	Nodes
1	1 2 3
2	3 4 5
3	5 6 7
4	7 8 9

With elements defined this way, the communication costs of the algorithm are greatly reduced. Because the node numbers making up the element in the inner integration specify which columns are accumulated into, it is easy to divide up the elements between the processors so that a processor performs only those integrations which accumulate into its matrix section. Consider, for example, two processors and the element definitions of Table 5.1. Assigning elements 1 and 2 to processor 1, and elements 3 and 4 to processor 2 would cause processor 1 to accumulate into columns 1 to 5 while processor 2 would accumulate into columns 5 to 9.

The only case where communication of coefficients between processors is required occurs when the last element assigned to one processor shares a node with the first element assigned to the next processor. In this case, both processors accumulate coefficients for the same matrix column. To simplify the matrix solution algorithm, it is deemed that the shared column of coefficients be passed backward (towards the host) in the processor array. Thus in the example above, processor 1 would be responsible for columns 1 to 5 and processor 2 would be responsible for columns 6 to 9.

5.3 Combining the Generator and Solver

The major points impacting the implementation of the BEM and Gaussian elimination algorithms on a linear array of processors have already been mentioned. Given these, the ways in which the algorithms impact each other are now detailed.

Most of the interaction stems from the fact that the boundary element is the smallest "quantum" with which one can deal when generating the system matrix. Because of this, it is not possible (or at least practical) to distribute the matrix columns between the processors as evenly as the matrix solution algorithm allows. Ordinarily, the maximum disparity in the number of columns per processor would be one, but because of this element quantization, the difference could be as much as three columns. This slows the algorithm down since processors with fewer columns will have to wait for processors with more columns.

For completeness (and robustness) of the algorithm it is necessary to consider the details of matrix accumulation and solution when the number of boundary elements is comparable to the number of processors. Again, since the element is the smallest "quantum" with which we can deal in generating the system matrix, a processor must generate at least three columns of the matrix. The only exception is the case when there are more processors than elements, where some processors sit idle during the whole of the algorithm execution.

While it may seem wasteful not to distribute the load of non-idle processors onto the idle ones, it is not. For most problems of practical interest, it is improbable that there will be more large-grain processors than there are elements. Moreover, the communication costs involved in redistributing the matrix columns would probably outweigh the gains in execution time.

In addition, there is the possibility of node sharing. If node 3 of the last element of processor i is the same as node 1 of the first element of processor $(i+1)$ (e.g. both with an identical node index number k), then both processors will sum into a common column k. To arbitrate this occurrence, it is deemed that processor $(i+1)$ will pass back its partially accumulated column to the ith processor. Thus the minimum number of columns that a processor handles (excluding zero) is two.

The maximum number of columns handled by a processor is limited only by available memory and round-off error. The actual number of columns it handles is equal to the number of distinct global node numbers present in the boundary elements processed in its inner integration loop (minus one if it shares a node with the previous processor in the pipeline).

5.4 Results

Initially a single transputer board (IMS B004) was used to develop the algorithm. Because of the mutual support provided by Occam and the transputer the multiprocessor version of the program could be run on the single transputer. The self-synchronizing nature of Occam programs guarantees that the program running in a simulated parallel processing environment (multitasking) would perform the same as it would on an array of transputers (multiprocessing).

Subsequently, an array of eleven transputers was obtained. As proof of the validity of the Occam/transputer programming model, the algorithm was adapted to run on the hardware array in a matter of hours — and ran correctly the first time.

To evaluate the performance of the algorithm, MANITOBA was applied to three problems of varying sizes (36 Elements - 81 Nodes, 77 Elements - 175 Nodes, and 130 Elements - 299 Nodes). This range of problem sizes corresponds to small, medium, and large respectively. Figures 5.6, 5.7, and 5.8 show the speedup efficiencies E_{N_p} (defined in equation (3.1)) attained by MANITOBA for the three problems using two to eleven processors. To further quantify the algorithm, E_{N_p} was calculated separately for the matrix generation and solution portions of the algorithm as well as for the algorithm as a whole.

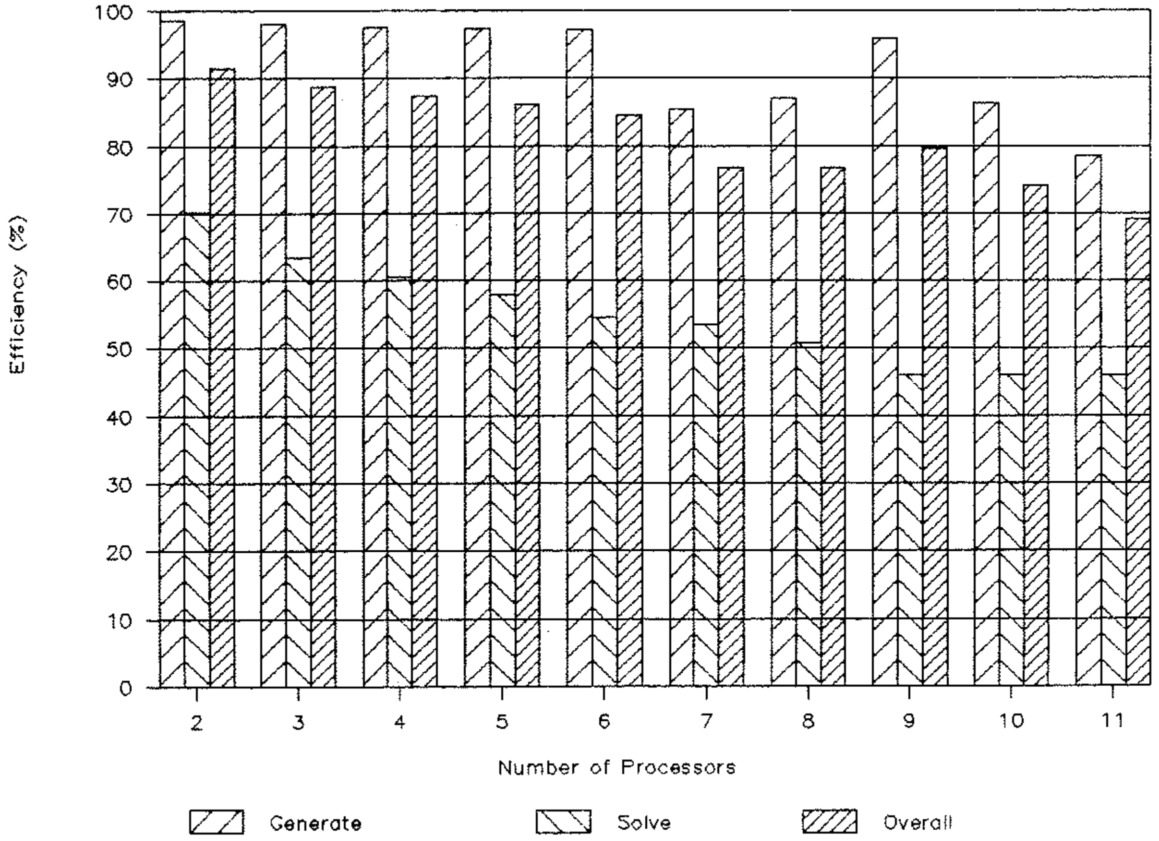

Figure 5.6: E_{N_p} for 36 element - 81 node problem.

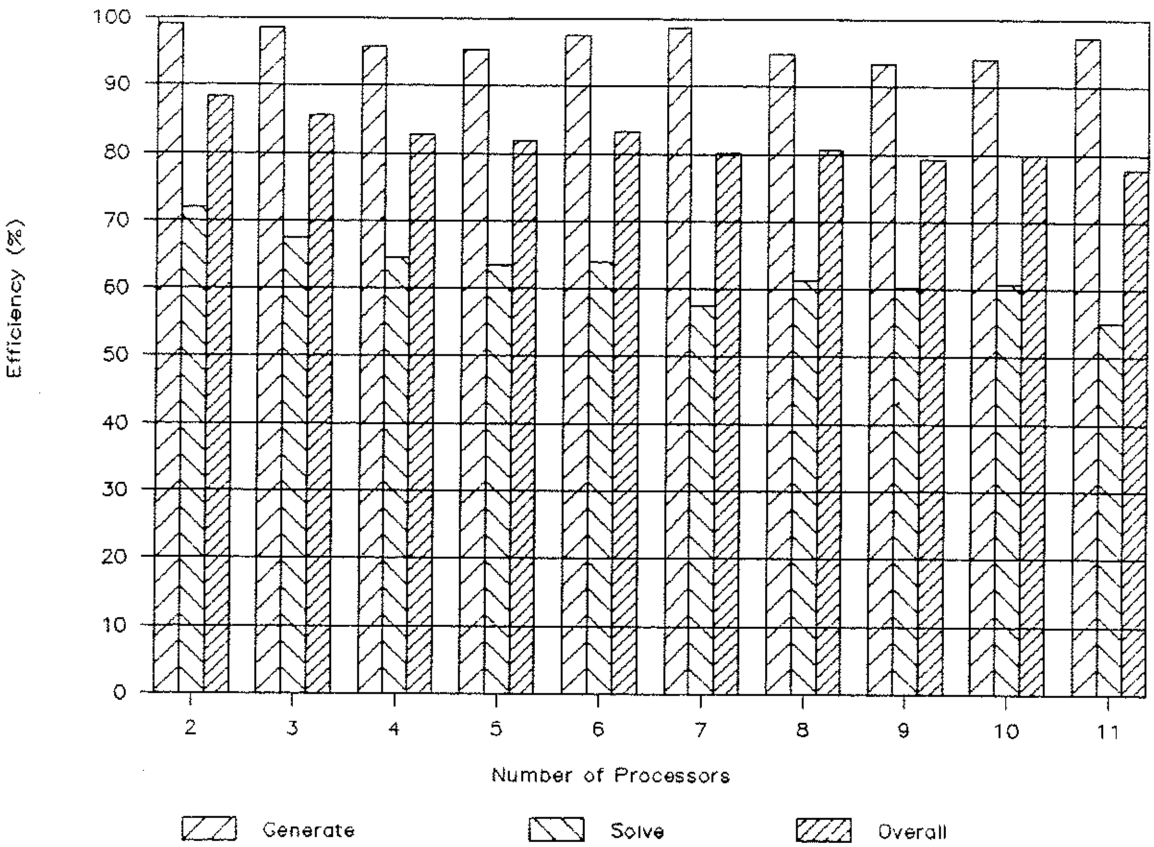

Figure 5.7: E_{N_p} for 77 element - 175 node problem.

It can be seen that excellent results are obtained for the the matrix generation portion of the algorithm. For the medium and large problems E_{N_p} is well over 90%, and only falls below 90% on the small problem in which communication costs become significant in comparison to computation costs. In fact, efficiencies rise steadily with increasing problem size, indicating that the matrix generation algorithm can be used on larger arrays of processors with little penalty.

A periodic rise and fall in efficiency can also be observed. While it is most predominant for the matrix generation algorithm, it also occurs for the matrix solution algorithm. The peaks in efficiency correspond to cases where there are an equal number of matrix columns assigned to each processor. This generally occurs

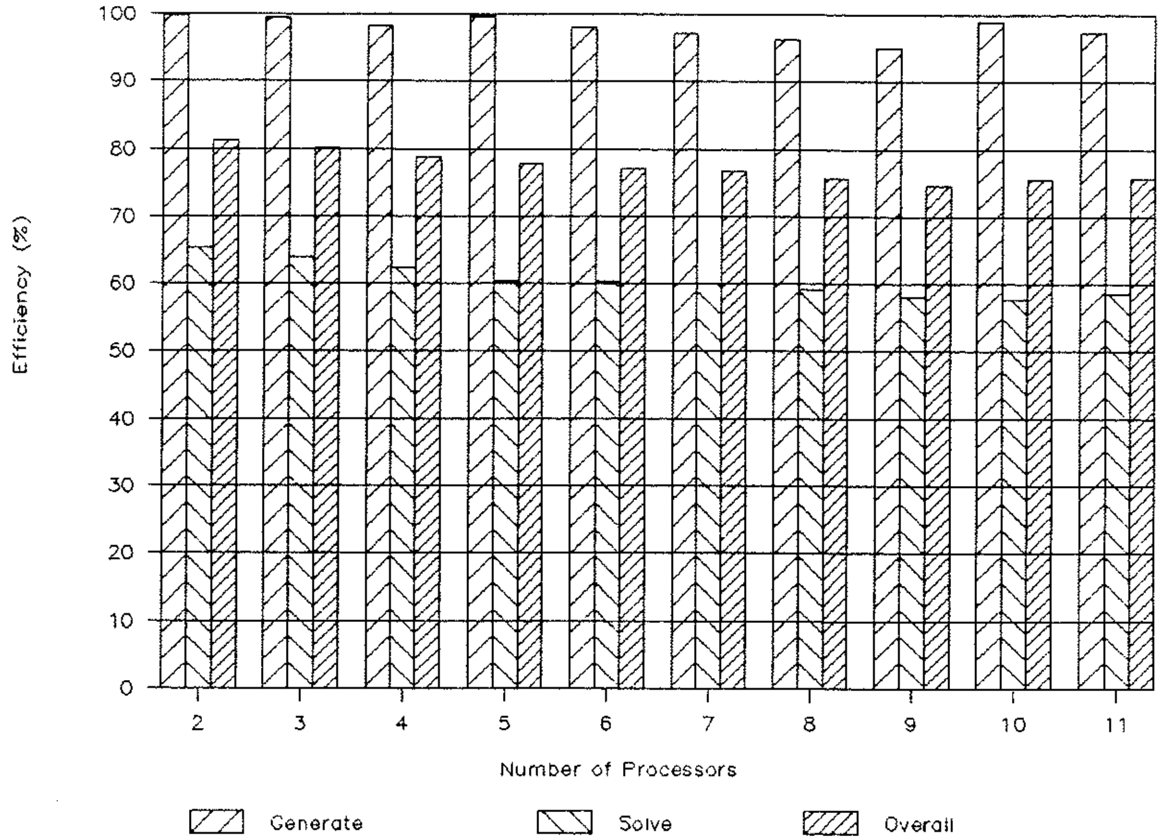

Figure 5.8: E_{N_p} for 130 element - 299 node problem.

when there are an equal number of elements assigned to the inner integration loop of each processor (see Section 5.3 for details). The word "generally" is used because saying exactly how many columns are assigned to a given processor is difficult. The possibility of of node sharing between elements makes the number of columns dependent upon details of the node numbering scheme.

Although it may not appear so at first glance, the results for the matrix solution portion of the algorithm are also good to excellent. To see this, we can calculate the theoretical efficiency when solving a $N \times N$ problem on N_p processors. In this case each processor will be assigned $C_p = N/N_p$ columns of the matrix to eliminate.

To calculate efficiency, we need both T_1 (the uniprocessor execution time) and T_{N_p} (the N_p processor execution time). When eliminating the ith column, row-operations are performed on $(N - i)$ rows, each of length $((N + 1) - i)$. Thus each elimination step requires $((N + 1) - i) * (N - i)$ multiply-subtractions (note that this number includes manipulation of the right-hand side of the linear system). To fully eliminate below the diagonal, $(N - 1)$ steps will be needed, since $(N - 1)$ rows have coefficients below the diagonal. Summing all of these operations gives

$$T_1 = (T_v{}^- + T_v{}^\times) \sum_{i=1}^{N-1} ((N + 1) - i) * (N - i). \tag{5.38}$$

Using the fact that

$$\sum_{i=1}^{N} i = \frac{N(N + 1)}{2}$$

and

$$\sum_{i=1}^{N} i^2 = \frac{N(N + 1)(2N + 1)}{6}$$

allows us to simplify (5.38), giving

$$T_1(N) = (T_v{}^- + T_v{}^\times)N(N - 1)(\frac{N}{3} - \frac{2}{3}). \tag{5.39}$$

The task of estimating T_{N_p} is simplified with the recognition that the last processor is the bottleneck in the system, and we need only to calculate its execution time to obtain the execution time of the whole algorithm. Until the last processor starts eliminating its own columns, it will be performing row operations upon all of its $(C_p + 1)$ columns (including the right-hand side). Since there are $(N - C_p)$ columns to the left, the last processor will have to do row operations on each of its columns $(N - C_p)$ times (for a length $(N - i)$ in the ith elimination step). Having done that, it eliminates a sub-matrix of dimension C_p on its own, giving an execution time formula identical to (5.39), but with C_p substituted for N (i.e. $T_1(C_p)$). Pulling all of these contributions together yields

$$T_{N_p} = (T_v{}^- + T_v{}^\times)\left[\left[\sum_{i=1}^{N-C_p}((C_p+1)-i)*(N-i)\right] + T_1(C_p)\right], \qquad (5.40)$$

which can be simplified to give

$$T_{N_p} = (T_v{}^- + T_v{}^\times)\left[(C_p+1)(N-C_p)[N - \frac{(N-C_p+1)}{2}] + T_1(C_p)\right]. \qquad (5.41)$$

The execution time predicted by (4.39) and (4.41) is for forward elimination only, and does not include the overhead of communicating the coefficients of elimination. It is instructive to see how close the predicted performance comes to experimental performance, as it offers a measure of how effective the overlapping of communication and computation is.

Figures 5.9 and 5.10 compare predicted elimination execution times and efficiencies for the 175 node BEM problem. The fact that the curves follow each other so closely indicates that communication time is not much of a factor in the total execution time of the algorithm, and that communication and computation are effectively overlapped in the Gaussian elimination algorithm. Also, since the theoretical expressions ignore the cost of back-substitution, the result also indicates that we are justified in assuming that the cost of back-substitution is small when compared to the cost of elimination. As such, these results indicate that the implementation of Gaussian elimination is close to optimal for this architecture.

Since the results indicate that communication is of small cost in the algorithm, we can expect that we will still have relatively good efficiencies with the faster T-800

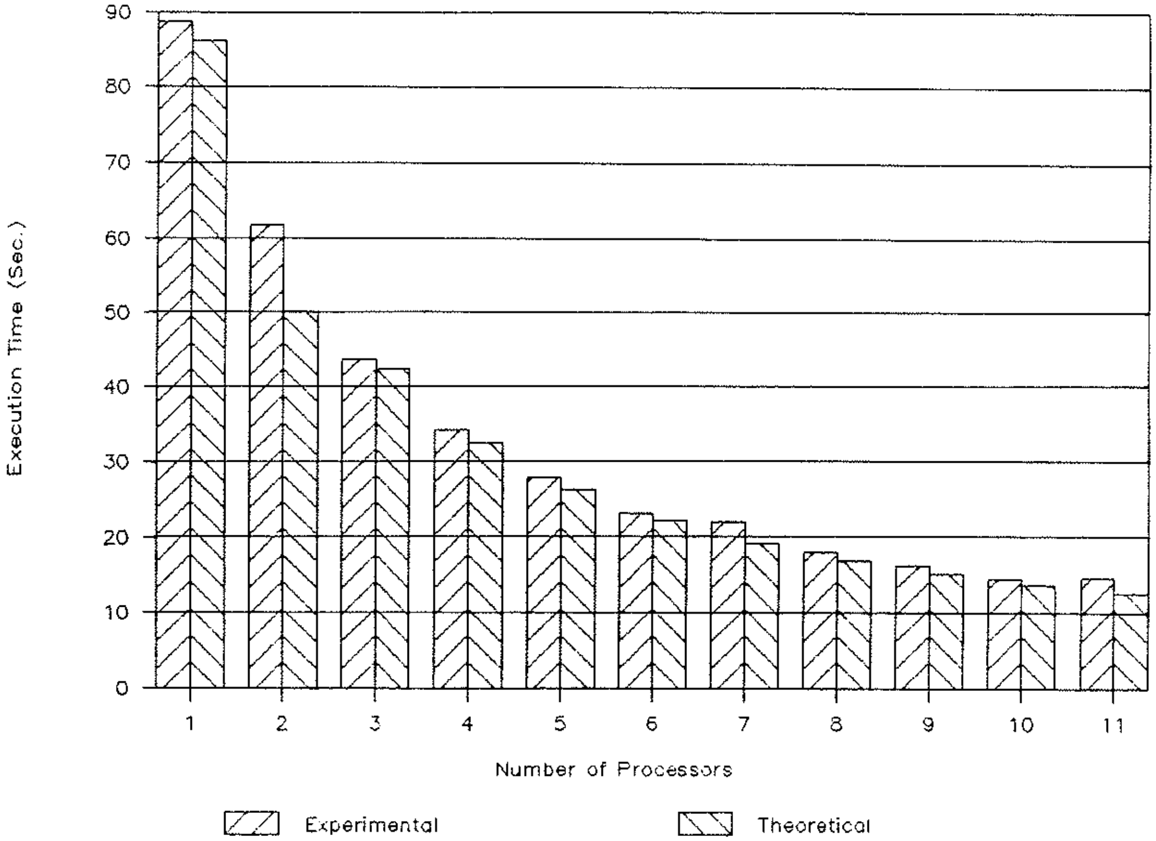

Figure 5.9: Theoretical vs. measured elimination execution times. Gaussian elimination results are compared for the 175 node BEM problem for 1 to 11 processors.

transputer. The efficiencies will definitely be lower, however, as the computational costs will be reduced by approximately a factor of 10 with the T-800.

Finally, looking at the overall efficiency of the algorithm shown in Figures 5.6, 5.7, and 5.8, we can see that it ranges between 88% and 75% for the medium and large problems. Since the decline in efficiency with increasing numbers of processors is not very steep, the implementation could also be applied to larger arrays. In general, for arrays of practical size, it can be seen that MANITOBA provides a realistic route to obtain significant boundary element performance in a desktop workstation environment.

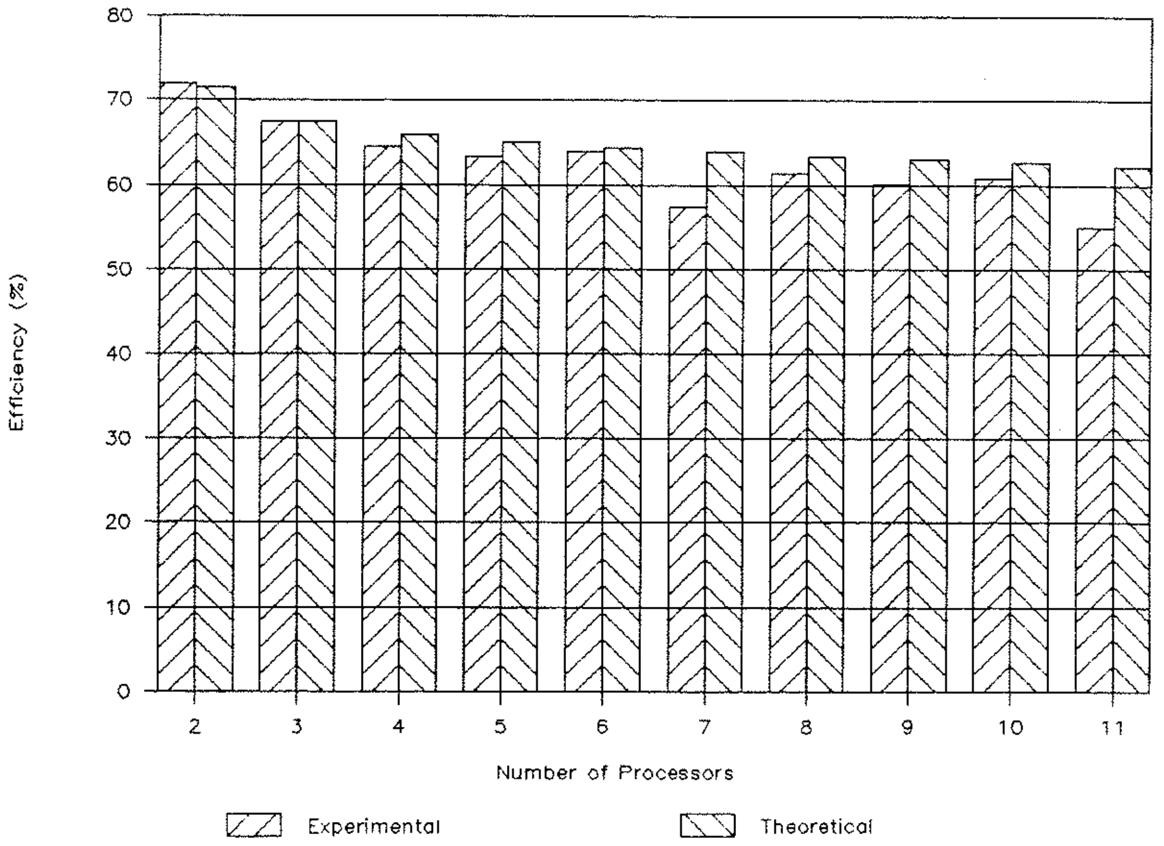

Figure 5.10: Theoretical vs. measured elimination efficiencies. Gaussian elimination results are compared for the 175 node BEM problem for 2 to 11 processors.

It is easily seen, however, that the major limitation of the combined algorithm is the matrix solver and not the matrix generator. It is the solver that will ultimately limit the number of processors that can be used efficiently.

It is doubtful that this imposes a *practical* limit however. Since a processor must have at least one element on its inner integration loop, it must be responsible for at least three matrix columns on average. Examination of theoretical efficiency for a 300 node problem indicates that with 100 processors, the efficiency would be still around 50%. Doubling the problem size to 600 nodes only increases the efficiency.

Thus, a 100 processor array can solve a 600 node problem at least as efficiently as a 300 node problem. However, without care, solving a matrix of that size using Gaussian eliminatior can incur problems with round-off error. Using double precision would allow larger problem sizes, but this would only increase the efficiency of the elimination algorithm by increasing the cost of computation, and further reducing the effects of communication. Thus, if one is happy with 50% efficiency, the parallel algorithm will always be practical within the constraints of Gaussian elimination itself.

To gauge the relative performance of MANITOBA, it was compared against a uniprocessor implementation running on a HP 9000-320 workstation consisting of a Motorola 68020 (16 MHz) assisted by a 68881 numeric coprocessor (12 MHz). Figure 5.11 summarizes the results for the small, medium, and large problems.

With eleven processors MANITOBA is about 6 times faster than the 68020 based machine. This result speaks well for the speed of the T-414 transputer microprocessor, since it is an integer chip with no floating point support. However, using the T-800 would decrease the computation component of the algorithm by about a factor of 10 (Electronics [1987]). Conservatively speaking, then, MANITOBA would perform at about 30 to 40 times the 68020 workstation's speed if it used eleven T-800 transputers. An impressive result indeed.

This chapter has addressed a two-dimensional BEM formulation. In three-dimensions, elements are two-dimensional patches instead of line-elements, and the issues become more complicated. With line elements we only had to worry about common node numbers at each end of the element. With patches, however, common node numbers can occur at any node on the boundary of the element. This makes it harder to divide up the inner loop as nicely as we have done in the two-dimensional case, so that matrix generation may require more communication.

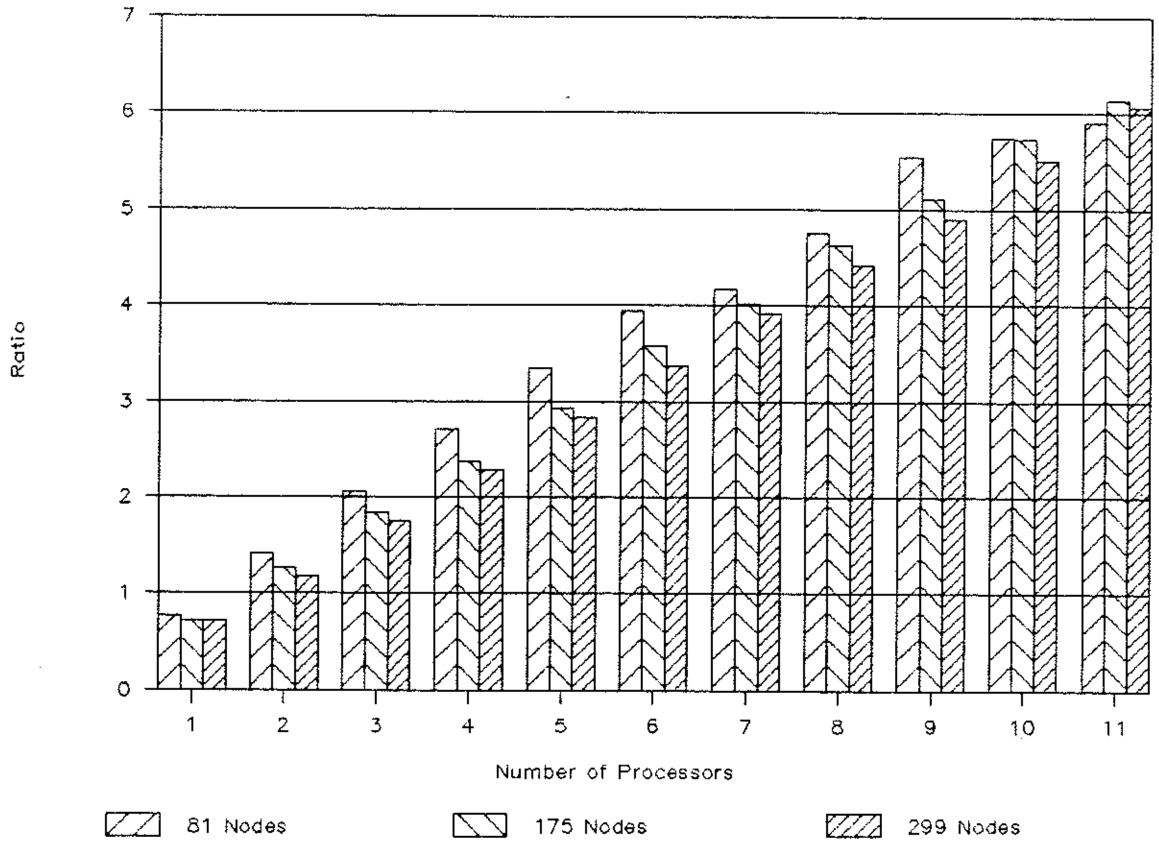

Figure 5.11: Comparison of MANITOBA to 68020 based workstation. This figure gives the ratio of the execution time for a 16 MHz 68020 based workstation (with 12 MHz 68881) to the execution time of MANITOBA with one to eleven transputers.

Again it should be possible to put suitable restrictions on the mesh node numbering to minimize communication between processors. However, the process will be more complex than the two-dimensional case.

CHAPTER VI

CONCLUSION

The goal of this dissertation was to show ways of achieving high performance for numerical analysis and CAE algorithms on microcomputers and workstations. The fact that we were working in a low cost environment made this task harder by requiring that the acceleration mechanism be low-cost. Consideration of this issue lead us to adopt the transputer as the implementation platform.

Again, it should be emphasized that this is the transputer's main advantage over other MIMD systems. All of the algorithms presented here can be adapted equally well to any MIMD system that utilizes simple point-to-point communication and local memory. Shared memory multiprocessing systems can also be used, but with fewer processors, since memory and bus contention would slow them down. Therefore, what is presented in this dissertation should be taken in a more global context, and not applied specifically to transputer networks.

If one wishes to assess the algorithms presented here on other MIMD processors, a table similar to Table 3.1 can be constructed and used in conjunction with the execution time formulas provided (except for BEM matrix generation). This will weight operation counts appropriately, and allow valid comparisons between different architectures, which are sure to have different strengths and weaknesses.

In addition to affecting the choice of implementation platform, the low-cost requirement had other far reaching implications. The major one being that the algorithms had to be developed for a limited number of processors. Since there were few processors, the simple linear array and shuffle-exchange communication networks were sufficient for our purpose. Given access to an unlimited number of processors, it would be necessary to consider more complex networks.

It can be seen that cost was a very influential factor in the algorithms presented here. Experience and the results presented show that the choice of the transputer was a good one. The PPCG(z) algorithm in Chapter 4 is able to compete favourably with computers costing considerably more on just eight T-800 transputers (based upon theoretically derived data).

We began in Chapter 3 with a study of basic matrix and vector algorithms. By carefully choosing a partitioning of vectors and matrices, it was shown that dyadic and monadic vector operations could co-exist with matrix operations on a linear network augmented with a shuffle-exchange network. These algorithms serve to demonstrate that programming MIMD computers is basically a problem in data organization. One has to insure that the data is where it is needed, or can be communicated there easily.

The monadic and dyadic vector operations present an almost ideal case, since all the component operations are independent. It was thus possible to process them on a linear array with virtually 100% efficiency. This presupposes that the coefficients of the vectors can be ordered and distributed identically on all processors.

The scalar product presented difficulty since values from all parts of the array had to be combined. This necessitated the introduction of the shuffle-exchange (SE) network because of its superior capabilities in that area. This is really only necessary for larger numbers of processors, however, since the advantage over the linear array is small for smaller array sizes.

The same SE network also allowed an efficient dense matrix-vector multiplication since matrix-vector multiplication is essentially a number of independent scalar products. The communication requirements were too expensive when applied to randomly sparse matrices, however, and we were led to consider a restricted class of sparse matrices (banded) in order to have an efficient algorithm.

The tools developed in Chapter 3 enabled a novel PPCG algorithm to be developed, since PPCG is entirely composed of these algorithms. This opened the door to parallel implementation of the finite-difference method (FDM), and we were able to solve a simple problem with high efficiency. Since the finite-element method (FEM) also produces sparse matrices of the type required by the PPCG algorithm, we can expect that the solver.can also be applied to that CAE technique.

This is really only the first step in the process, however, as it does not consider the production of the matrices. In the simple finite-difference problem presented in Chapter 4 it was very easy to generate the matrix in parallel, but the general application of the FDM or the FEM will not adapt as easily to parallel partitioning.

In the case of the FEM, the integration over each element is independent, but coupling due to common nodes between elements may result in large communication costs for matrix generation. In addition, matrices with unacceptable bandwidth may be produced, invalidating the use of the banded sparse matrix-vector multiplication routine in the PPCG routine. Thus, the major issue involved is keeping the bandwidth of the matrix narrow so that we may solve the matrix equation efficiently. To do this, one may be forced to compromise the performance of the solver in favour of the generator or vice versa.

MANITOBA, on the other hand, successfully addresses both of these issues, and represents a complete CAE algorithm. By making restrictions on the the input data, the matrix generation algorithm was partitioned efficiently, and in such a way that very little data movement was required to put the matrix into the format required by the Gaussian elimination routine.

It is argued by some that the types of algorithms studied here are most suitable for vector supercomputers. Cost issues aside, the results presented here and elsewhere indicate that MIMD parallelism is also suitable for such algorithms. Of course there are arguments for both sides, but one should not dismiss MIMD parallelism out of habit.

The problem with MIMD parallelism, when compared to vector computers, is that a MIMD computer faces problems when algorithms require global access to the data. A vector computer merely has to fetch a value from memory, while a multiprocessor may have to communicate the value through several intervening processors.

However, the unrestricted access to data has problems associated with it. Memory becomes a bottle-neck since all accesses must come through one place. While schemes such as interleaving can be implemented, they quickly drive up the cost of the computer. In fact, it can be argued that most of the resources (and hence cost) of a vector computer are dedicated to memory systems whose purpose is to feed the data-hungry pipelines.

MIMD processors do not suffer this bottle-neck (excluding shared-memory types), since they all access independent memory. This effectively multiplies the speed of the memory by a factor equal to the number of processors, giving very high effective memory bandwidth. Also, since increasing the speed of a memory system causes costs to grow exponentially, the slower memory system of the MIMD machine can be very inexpensive when compared to the vector computer. This in turn, implies that each node in a MIMD machine can be disproportionately cheaper than the vector counterpart.

This work has shown that programming of MIMD machines is not a trivial task. Generally, the kind of parallelism they support requires that one have a very extensive idea of what is happening in the algorithm at all levels of parallelism. Where a vector computer attains speed by shear brute-force, a MIMD algorithm must approach a problem with surgical-like precision.

Consider MANITOBA for a moment. MANITOBA represents a tight fusion of two very different algorithms. The requirements on input data, although basically simple, could not have been realized without a thorough "global" picture of data dependency in the algorithm. In fact, the necessary conditions are born of the innermost loop in the program combined with the method of geometric modelling. Having a machine recognize such a tactic would be enormously difficult, if not impossible.

Because MIMD programming is essentially a "thought" process, it does not seem likely that automated MIMD programming will be possible in the near future, so perhaps the best that can be hoped for is a "programming assistant". Since this work has shown that a linear array (or more generally, a ring of processors) and the shuffle-exchange network allow efficient MIMD algorithms for both numerical analysis and CAE, it is proposed that the combination of the two networks can be the basis of a versatile hardware accelerator. This allows much flexibility within a fixed network topology. Given a fixed topology, it might be possible to automate some aspects of algorithm development. Since the transputer with four links is capable of implementing both network topologies together, it would be an ideal implementation vehicle.

APPENDIX A

An Occam-Transputer Programming Example

This Appendix is included to better explain the methodology of programming in Occam for a transputer array. Specifically, it shows how the Transputer Development System (TDS), marketed by INMOS Corporation, is used to develop parallel programs. While environments using other languages are appearing, they are not considered here.

The TDS is discussed first. This is necessary to give a feel for the programming environment in the hope that it will make what follows as clear as possible. This will also explain why the host-accelerator concept is used throughout this work. Development using the TDS mandates it.

The algorithm chosen is the distributed addition of N_p numbers on a N_p processor shuffle-exchange (SE) network described as described in Section 3.2.2. It has the virtue of being simple to understand and present, while still illustrating some important Occam/transputer programming principles. The thought processes required at each step of the algorithm's development are discussed.

A.1 The Transputer Development System

The TDS is a complete programming environment for the development of transputer programs. Although other languages are available, it is primarily a vehicle for Occam programming. The author's system is hosted on an IBM PC containing an IMS B004 transputer board (Ghee [1986]) as shown in Figure A.1. The PC and the B004 are interfaced through the PC bus and a transputer link adapter which effectively gives the PC a transputer link.

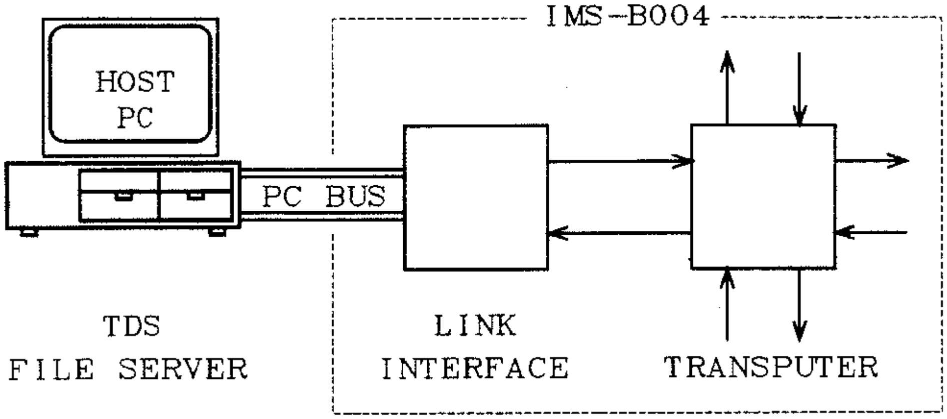

Figure A.1: PC-based transputer development system.

The TDS used here consists of two parts:

1. a file and screen server program running on an IBM PC platform; and
2. a compiler-editor system running on the transputer located on the B004 board.

All file and screen I/O required by the server takes place over a standard transputer link (see Chapter 2).

With the TDS, one is able to code and execute Occam programs entirely on the B004's transputer. Any parallel components of these programs can be executed concurrently using the transputer's hardware time-slicing abilities. This is usually the first step when developing a parallel algorithm, as it is much easier to debug programs running on the host processor. Moreover, the laws of Occam programming guarantee that the behaviour of the program will be identical when distributed over an array of transputers.

When a program is debugged, additional facilities exist for specifying how to distribute it over an array of transputers. This process, called configuration, allows the programmer to specify which process should be placed on a given transputer, and which transputer links to associate with its input and output channels.

A configured program can then be loaded into a network of transputers that is connected via a transputer link to one of the free links of the transputer on the B004 (assuming, of course, that the network is interconnected in the appropriate topology). This loading process is entirely transparent to the user, as the TDS includes a distributing network loader. A program running on the host B004 can then be used to provide any data required by the computational array and to receive the computational results.

A.2 Specification in Occam

As an example we will implement the SE network discussed in Chapter 3. It will be used to sum a set of numbers, one in each processor of the network. The collapsed view of the SE network is shown in Figure A.1 (reproduced from Chapter 3).

We first describe how this network and algorithm can be described for execution on a single transputer. This is usually the first target for new algorithms, as it provides a more convenient testing environment. The next section will describe how this description can be adapted to run on a network of transputers with I/O links connected in the SE topology. The Occam/transputer programming model guarantees that the single processor and distributed versions will behave identically.

First a practical note: since present transputers possess four I/O links, any process that we construct for placement on a single transputer must not require more than this number. We could, if we choose to, implement a link multiplexing

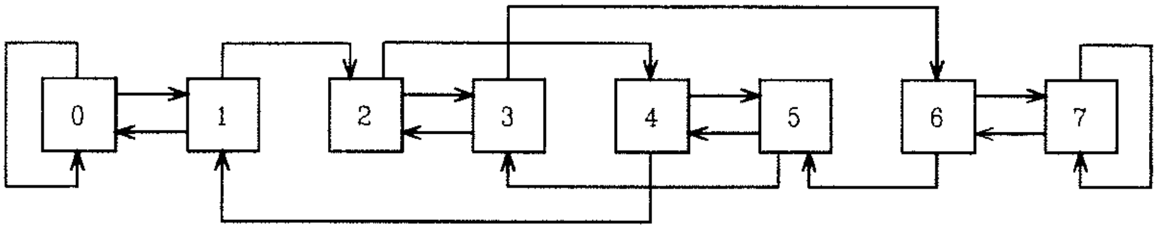

Figure A.2: The SE interconnection network.

scheme, but this may carry with it performance penalties, as well as introducing complexity to the algorithms. The SE network is fortunate in that it only requires a maximum of three I/O links per computational process.

This is only a concern if one intends to execute the algorithm on a real network of transputers. If one is only interested in developing parallel algorithms, with no intention of actual parallel execution on a specific hardware target, a process with any channel requirement can be created.

Isolating a single processor from Figure A.2, we can view it as a black box with two input channels and two output channels as shown in Figure A.3. The channels `ShuffOut` and `ShuffIn` serve to implement the perfect shuffle network, while `ExchgIn` and `ExchgOut` mediate the exchange operation (see Section 3.2.2 for definitions of these operations).

In Occam (ignoring the contents of the black box for the moment), we could express Figure A.3 as follows:

```
SE.Proc(CHAN OF ANY ShuffIn,ShuffOut,ExchgIn,ExchgOut)
```

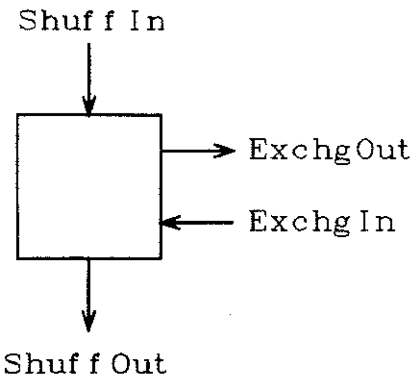

Figure A.3: A single SE processor.

Note, the direction of communication on a channel is not explicitly defined in Occam. Rather, it is implicitly defined to be from the output process on one end to the input process on the other. The channel itself carries no information about its direction. By appending the channel's name with `In` and `Out` we define the direction mnemonically.

Somehow we must arrange that a number of copies of `SE.Proc` be used with proper channel arguments to implement the SE topology. This, of course, must be done with the constraint that the number of processors must be an integral power of 2 (i.e. $N_p = 2^N$).

Since we may wish to look at the results of the computation, we will use the host-accelerator architecture that has been used throughout this work (see Figure 5.1 for example). This requires that we designate a single processor as the interface between the host and the SE network. Conveniently, we choose processor 0 for this purpose. This will require that the homogeneity of the network be destroyed somewhat. Where we could have placed identical programs in each processor, it is necessary to place a somewhat modified one in processor 0 to handle external communication.

A side effect of this is that we must reserve one channel of processor 0 for communication with the host. Thus we must arrange that the process(program) we place in that processor only require at most, 3 links. Looking at Figure A.2, we can see that processor 0 only requires 3 links, so no extra effort is required to address the problem. In addition, we should notice that the shuffle operations on processor 0 and 7 ($N_p - 1$) are redundant, and could be replaced with internal memory to memory copy. Eliminating them would free up links at each end for other purposes (multiple links to host or between each end of array).

This introduces additional complexity, but does not require we burden the processors with logic to arbitrate the special cases for communication at each end of the array. We now require three procedures for the beginning, middle, and end of the array:

```
SE.Proc.0 (CHAN OF ANY FromHost,ToHost,ExchgIn,ExchgOut)
SE.Proc.I (CHAN OF ANY ShuffIn,ShuffOut,ExchgIn,ExchgOut)
SE.Proc.L (CHAN OF ANY ExchgIn,ExchgOut)
```

Each of them will be designed optimally for their role in the communication and computation process.

Recalling the definition of the perfect shuffle from Chapter 3, the i'th processor P_i "shuffles-out" to another processor whose index is:

$$P_i \longrightarrow \begin{cases} P_{2i}, & 0 \leq i \leq N/2 - 1, \\ P_{2i+1-N}, & N/2 \leq i \leq N - 1. \end{cases} \tag{A.1}$$

The bi-directional exchange connections between the processors are defined by:

$$P_{2i} \longleftrightarrow P_{2i+1} \quad 0 \leq i \leq 2^{N-1} - 1. \tag{A.2}$$

A straightforward translation of the SE operation yields the following Occam procedure (where we now use a more descriptive name for the procedure):

```
--
-- Middle processors (I'th processor)
--
PROC SE.Add.I(CHAN OF ANY ExchgIn,ExchgOut,
              CHAN OF ANY ShuffIn,ShuffOut,
              VAL INT Log2Np,
              VAL INT ProcNum)
  INT MyNum,NextNum:
  SEQ
    MyNum := ProcNum                -- Sum all the ProcNums
    SEQ I = 0 FOR Log2Np
      SEQ
        PAR                         -- Parallel shuffle
          ShuffOut ! MyNum
          ShuffIn ? NextNum
        PAR                         -- Parallel exchange
          ExchgOut ! NextNum
          ExchgIn  ? MyNum
        MyNum := MyNum + NextNum    -- Accumulate
:
```

This represents the generic SE procedure, and is used for all shuffle processes except the first and the last. It is important to notice the use of PAR in the communication sections. This allows input and output communication to proceed concurrently. When running entirely on a single transputer this is not as important, but on a multiprocessor application, true concurrency can be achieved on link communication because of the transputer's link architecture.

The parallel communication also eliminates the possibility of deadlock for the shuffle and exchange operations in a transparent fashion. Otherwise, the programmer would have to devise an orderly communication scheme to prevent deadlock.

This is most easily seen in the case of the exchange operation. If the communication was done sequentially, and both processes tried to exchange-out, they would be deadlocked since neither could receive the others message. This could be circumvented by ensuring that one of the pair was exchanging out while the other was exchanging in, but this would add more complexity than is necessary, while ignoring the performance benefits discussed above.

The last processor in the chain requires some special consideration. As was already mentioned, the shuffle operation feeds the information back on itself and could be replaced by an internal data transfer. Doing this would have the side benefit of releasing precious channel resources at the end of the array for other purposes should the need arise. Taking the shuffle operation inside the process results in the following Occam procedure (suffix "L" indicates last):

```
--
-- Last processor - internal shuffle step.
--
PROC SE.Add.L(CHAN OF ANY ExchgIn,ExchgOut,
              VAL INT Log2Np,
              VAL INT ProcNum)
  INT MyNum,NextNum:
  SEQ
    MyNum := ProcNum                -- Sum all the ProcNums
    SEQ I = 0 FOR Log2Np
      SEQ
        NextNum := MyNum            -- Internal shuffle
        PAR                         -- Parallel exhange
          ExchgOut ! NextNum
          ExchgIn  ? MyNum
        MyNum := MyNum + NextNum    -- Accumulation
:
```

The same issues apply to the first processor (0), so the shuffle operation is also placed internally for it. There is however an additional requirement placed upon it. Namely, that it communicate the result of the sum operation back to the host. We therefore must add channels to it for this purpose, along with the code to communicate the final result, which yields:

```
--
-- First Processor - has internal shuffle step
--                 - communicates answer to host
--
PROC SE.Add.0(CHAN OF ANY ToHost,FromHost,
              CHAN OF ANY ExchgIn,ExchgOut,
              VAL INT     Log2Np,
              VAL INT     ProcNum)
  INT MyNum,NextNum:
  SEQ
    MyNum := ProcNum                  -- Sum all the ProcNums
    SEQ I = 0 FOR Log2Np
      SEQ
        NextNum := MyNum              -- Internal shuffle
        PAR                           -- Parallel exchange
          ExchgOut ! NextNum
          ExchgIn  ? MyNum
        MyNum := MyNum + NextNum      -- Accumulation
    ToHost ! MyNum                    -- Give host the answer
                                      -- (this processor only)
:
```

Given the above procedures, we must connect them up with channels in the proper fashion (i.e. the SE topology). This is most easily done using an array of channels (defined in the same way as an array of variables) and the Occam replicated PAR construct. It also allows us to target different size SE networks in a convenient fashion. We will need separate arrays for both the shuffle connections and the exchange connections, with each input and output treated separately. Also, the arrays for the exchange operation need only be half the size of the shuffle channel arrays.

```
VAL Np        IS 8:                      -- number of processors
VAL Log2Np    IS 3:                      -- log base 2 of Np
[Np] CHAN OF ANY Shuff:                  -- shuffle channels
[Np/2] CHAN OF ANY ExLtoR,ExRtoL:        -- exchange channels
```

We use the convention that each processor i has channel `Shuff[i]` as its shuffle-out channel. Given this, we must determine the channel to use for input. Equation (A.1) is not immediately helpful, as it indicates the connection between processors and not channel resources. Since we are making a one to one identification of output channel numbers and processor numbers, we can invert equation (A.1) to give

$$P_i \longleftarrow \begin{cases} P_{i/2}, & i \text{ even}, \\ P_{(i-1+N)/2}, & i \text{ odd}, \end{cases} \tag{A.3}$$

which indicates the processor that P_i inputs from. Given this information, we immediately know the channel index to use.

Exchange operations will be keyed from their representation in Figure A.2 (left to right and right to left). The only thing to watch is that we map a given channel between the proper set of processors.

The following Occam code fragment embodies the above considerations in the connection of a SE array:

```
--
-- Parallel shuffle-adds
--
PAR
  SE.Add.0(ToHost,FromHost,          -- Processor 0
      ExRtoL[0],ExLtoR[0],Log2Np,0)
  PAR I = 1 FOR ((Np-2)/2)           -- Middle processors
    PAR
      VAL INT PN IS ((2*I) -1):      -- Proc. Num  (odd)
      SE.Add.I(ExLtoR[I-1],ExRtoL[I-1],
          Shuff[((PN-1)+Np)/2],Shuff[PN],
          Log2Np,PN)
      VAL INT PN IS (2*I):           -- Proc. Num  (even)
      SE.Add.I(ExRtoL[I],ExLtoR[I],
          Shuff[PN/2],Shuff[PN],
          Log2Np,PN)
  SE.Add.L(ExLtoR[(Np-2)/2],         -- Processor (NumProcs-1)
      ExRtoL[(Np-2)/2],Log2Np,(Np-1))
```

Note the use of the replicated PAR. By changing the constants Np and Log2Np we can construct SE arrays of any size. These numbers *must* be constants however, as dynamic use of the PAR is not allowed.

All the preceding code fragments can then be wrapped up into a single Occam procedure SE.Add.Unit.

```
PROC SE.Add.Unit(CHAN OF ANY ToHost,FromHost)
  ... PROC SE.Add.O()
  ... PROC SE.Add.I()
  ... PROC SE.Add.L()
  VAL Np       IS 8:                  -- number of processors
  VAL Log2Np   IS 3:                  -- log base 2 of Np
  [Np] CHAN OF ANY Shuff:             -- shuffle channels
  [Np/2] CHAN OF ANY ExLtoR,ExRtoL:   -- exchange channels
  ... Parallel shuffle-adds
:
```

Note, that we have used the convention of the INMOS TDS whereby sections of hidden text are denoted by "..." on the left margin. Conceptually this represents a "crease" in the program listing, which hides the text. This is a useful way of seeing the overall structure of the program without the distraction of seeing all the detail.

Procedure **SE.Add.Unit** is a model for the computational system as it would exist on an actual array of transputers. The properties of Occam and the transputer together ensure that this procedure will execute identically on one transputer, or with its parallel components distributed over many. It and its component procedures have been designed with multiprocessor implementation in mind. The process of taking the single processor version and placing it on an array of transputers is described in the next section.

A.3 Mapping onto a Transputer Network (Configuration)

The following discussion makes a number of assumptions regarding the equipment of the user. First of all, it applies only to the Beta-2 release of the TDS. Second, it assumes that a set of transputers is available with access to *all* the links of each transputer. The author's hardware does not meet this requirement (Vadher and Walker [1986]), as each set of four processors is hard-wired as a 2 × 2 square array. This restricts the assignment of channels and introduces an unnecessary complication to the configuration process.

The basic idea behind configuration is to assign hardware (physical) resources to the virtual resources defined in the previous section. For example, you might tell the TDS to put process "A" on transputer number 0. Further, you may also request that the input or output side of a transputer link "i" be associated with a particular channel of process A. Figure A.4 shows how the four links of a transputer are numbered. Further, each link consists of two channels (input and output), whose numbers are also shown. These index numbers are used to refer the channels when allocating resources.

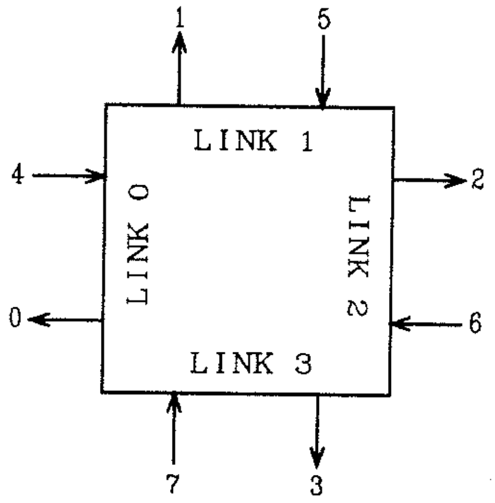

Figure A.4: Transputer link numbering.

Configuration replaces PAR by the construct PLACED PAR. Each processor that is placed is identified by a number and type specification (which indicates what version of transputer is used) via the PROCESSOR statement. For example, the line

```
PROCESSOR 2 T4
```

specifies that a T-414 transputer is being identified as processor number 2 in the network being defined.

The PROCESSOR statement is followed by PLACE statements which assign named channels to specific hardware channels of the transputer. The statement

```
PLACE ToHost AT 2:
```

associates channel 2 (part of link 2) with the channel name ToHost. This name can than be used as an argument to an instance of one of the procedures that are being placed in the network.

Here in complete detail is the configuration specification for the SE network (assuming previous definitions for the SE.Add procedures). For simplicity, the same links (channels) on each transputer are used in an identical role. Link 1 is used for shuffling-out, link 2 for shuffling-in, and Link 3 for both exchange channels. Since ExchIn and ExchOut are both between the same processor we are able to assign them to a single transputer link. The shuffle channels, on the other hand, are between different processors, so each require a separate link.

```
VAL Np       IS 8:                 -- number of processors
VAL Log2Np   IS 3:                 -- log base 2 of Np
CHAN OF ANY ToHost,FromHost:       -- host i/o channels
[Np] CHAN OF ANY Shuff:            -- shuffle channels
[Np/2] CHAN OF ANY ExLtoR,ExRtoL:  -- exchange channels
VAL ShuffInChan  IS 1:  -- Shuffle's must be on different
VAL ShuffOutChan IS 6:  --   links (1 and 2).
VAL ExchInChan   IS 7:  -- Exchange uses both channels
VAL ExchOutChan  IS 3:  --   of link 3.
PLACED PAR
  PROCESSOR 0 T4                   -- First proc.
    PLACE ToHost    AT 0:          -- Link 0 to host
    PLACE FromHost  AT 4:
    PLACE ExRtoL[0] AT ExchInChan:
    PLACE ExLtoR[0] AT ExchOutChan:
    SE.Add.0(ToHost,FromHost,ExRtoL[0],ExLtoR[0],Log2Np,0)
  PLACED PAR I = 1 FOR ((Np-2)/2)
    PLACED PAR
      VAL PN IS ((2*I) - 1):
      PROCESSOR PN T4             -- Odd middle procs.
        PLACE ExRtoL[I-1]          AT ExchOutChan:
        PLACE ExLtoR[I-1]          AT ExchInChan:
        PLACE Shuff[((PN-1)+Np)/2] AT ShuffInChan:
        PLACE Shuff[PN]            AT ShuffOutChan:
        SE.Add.I(ExLtoR[I-1],ExRtoL[I-1],
            Shuff[((PN-1)+Np)/2],Shuff[PN],Log2Np,PN)
      VAL PN IS (2*I):
      PROCESSOR PN T4             -- Even middle procs.
        PLACE ExRtoL[I]   AT ExchInChan:
        PLACE ExLtoR[I]   AT ExchOutChan:
        PLACE Shuff[PN/2] AT ShuffInChan:
        PLACE Shuff[PN]   AT ShuffOutChan:
        SE.Add.I(ExRtoL[I],ExLtoR[I],Shuff[PN/2],Shuff[PN],
            Log2Np,PN)
  PROCESSOR (Np - 1) T4            -- Last proc.
    PLACE ExRtoL[(Np-2)/2] AT ExchOutChan:
    PLACE ExLtoR[(Np-2)/2] AT ExchInChan:
    SE.Add.L(ExLtoR[(Np-2)/2],ExRtoL[(Np-2)/2],
        Log2Np,(Np-1))
```

While this certainly looks more complicated, the similarity to what we had before is quite evident. Logically, nothing has changed. We have merely specified a physical mapping for the parallel components of the algorithm.

A.4 Observations

One of the major points to be observed in the previous sections is that programming parallel algorithms using Occam is in no sense an automatic process. Considerable effort is required on the part of the programmer to partition an algorithm for the Occam programming model (and the transputer architecture). Some additional effort is then required to map the algorithm onto a transputer array.

Placing the onus on the programmer is not without its benefits, however. It is unlikely that "automatic parallelization" will reach a level that will allow it to partition problems efficiently in the near future. The process requires too much of a global picture of a given algorithm to expect it. The human mind is capable of assembling such global pictures, and hence should be expected to come up with novel approaches which would escape machine analysis of the problem.

Configuration, however, is one area that might be improved. While it is fairly straightforward, there is no reason that it could not be automated, removing an extra step from the programming process. The idea of configuration seems to have resulted from the fact that a transputer network is usually a fixed entity. Once it is connected in a configuration, it will probably stay that way. Thus the designer needed a strict way of specifying the topology that was to be used. In the case of the author's hardware (Vadher and Walker [1986]), this capability was absolutely necessary since the B003 board had the four transputers hard-wired into a 2 × 2 array.

With the advent of the IMS C004 (Hill [1987]) the situation has totally changed. The C004 is a 32 × 32 INMOS transputer link crossbar switch. That is, it is capable of connecting any of the inputs and outputs of 32 separate links (each input and output can be connected only once – many-to-one interconnections are not allowed). Moreover, the connection topology is entirely software controlled.

With the C004, the configuration process could be automated. Given the Occam representation of the algorithm, the C004 connection pattern could be generated, saving the programmer the necessity of producing the configuration himself. The manual connection of links using patch-cords would also be avoided.

Also evident is that the TDS is able to produce programs for imbedded systems. That is, systems that do not require any external support. For example, once the SE array is loaded, it operates totally independent of the host. In principle, the code could be placed in read only memory (ROM) for each transputer so that the array could be autonomous. In digital electronics parlance, the TDS produces ROMable code. Because of this, the transputer can be used in applications where there may be no host computer to boot it. Again, the transputer is able to provide a low-cost, yet powerful system.

All this aside, one thing speaks for itself. The Occam/transputer combination provides inexpensive, powerful, and accessible parallel processing.

REFERENCES

Adams, L., and Ortega, J., "A Multi-Color SOR Method for Parallel Computation", in *Proc. of the 1982 International Conference on Parallel Processing*, IEEE, pp. 53-61, 1982.

Allen, R.A.M., *Parallelization of the Preconditioned Conjugate Gradient Method Using a Processor Array.* Masters Thesis, The University of Manitoba, Winnipeg, Manitoba, Canada, 1983.

Atkin, P., *Tech. Note 17: Performance Maximization.* INMOS Ltd. publication 72 TCH 017 00, March 1987.

Axelsson, O., "Solution Of Linear Systems of Equations: Iterative Methods", in *Sparse Matrix Techniques (Lecture Notes in Mathematics #572).* Barker, V.A. (ed.), Springer-Verlag, Berlin, Germany, pp. 1-51, 1977.

Barlow, R.H., and Evans, D.J., "Parallel Algorithms for the Iterative Solution to Linear Systems", in *The Computer Journal*, Vol. 25(1), pp. 56-60, 1982.

Barron, I., Clayill, P., May, D, and Wilson, P., "Transputer does 10 or more MIPS, even when not used in parallel", *Electronics*, pp. 109-115, Nov. 17, 1983.

Davis, P.J., *Interpolation and Approximation.* Dover Publications Inc., New York, 1975.

Dubois, P.F., Greenbaum, A., and Rodrigue, G.H., "Approximating the Inverse of a Matrix for Use in Iterative Algorithms on Vector Processors", in *Computing*, Vol. 22, pp. 257-268, 1979.

Eisenstat, S.C., Schultz, M.H., and Sherman A.H., "Considerations in the Design of Software for Sparse Gaussian Elimination", in *Sparse Matrix Computations.* Bunch, J.R., and Rose, D.J. (eds.), Academic Press Inc., New York, pp. 263-273, 1976.

Electronics, "Credit-Card Size Transputer Modules Can Turn A PC Into a Super-Mini", *Electronics*, pp. 85, Jan. 21, 1988.

Flynn, M.J., "Some Computer Organizations and Their Effectiveness", in *IEEE Transactions on Computers*, Vol. C-21 (9), pp. 948-960, 1972.

Forsythe, G.E., and Wasow, W.R., *Finite-Difference Methods for Partial Differential Equations.* John Wiley and Sons, Inc., New York, 1960.

Geist, G.A., Heath, M.T., and Ng, E., "Parallel Algorithms for Matrix Computations", in *The Characteristics of Parallel Algorithms*, Jamieson, L.H., Gannon, D.B., and Douglass, R.J. (eds.), The MIT Press, pp. 233-251, 1987.

Gentleman, W.M., and George, A., "Sparse Matrix Software", in *Sparse Matrix Computations.* Bunch, J.R., and Rose, D.J. (eds.), Academic Press Inc., New York, pp. 243-261, 1976.

Ghee, S., *Tech. Note 11: IMS B004 IBM PC add in board.* INMOS Ltd. publication 72 TCH-011 00, 1986.

Gottlieb, A., and Schwartz, J.T., "Networks and Algorithms for Very-Large-Scale Parallel Computation", in *IEEE Computer*, Vol. 15(1), pp. 27-36, 1982.

Harrington, R.F., *Field Computation By Moment Methods.* Reprinted by R.F. Harrington, R.D. 2, West Lake Road, Cazenovia, N.Y., 13035, 1968.

Haynes, L.S., Lau, R.L., Siewiorek, D.P., Mizell, D.W., "A Survey of Highly Parallel Computing", in *IEEE Computer*, Vol. 15(1), pp. 9-24, 1982.

Hestenes, M.R., "The Conjugate-gradient Method for Solving Linear Systems", in *Proceedings of the Sixth Symposium in Applied Mathematics of the American Mathematical Society*, Curtiss J.H. (ed), McGraw-Hill Book Company, Inc., Toronto, 1956.

Hill, G., *Tech. Note 13: Transputer networks using the IMS B003.* INMOS Ltd. publication 72 TCH-013 00, 1986.

Hill, G., *Tech. Note 19: Designs and Applications for the IMS C004.* INMOS Ltd. publication 72 TCH-019 00, June, 1987.

Hillis, W.D., *The Connection Machine.* The MIT Press, Cambridge, Massachusetts, 1985.

Hoare, C.A.R., "Communicating Sequential Processes", *Communications of the ACM*, Vol. 21 (8), pp. 666-677, 1978.

Hoare, C.A.R., *Communicating Sequential Processes*, Prentice-Hall International, U.K., Ltd., 1985.

Hockney, R.W. and Jesshope, C.R., *Parallel Computers.* Adam Hilger Ltd., Bristol, 1981.

Hofstadter, D.R., *Gödel, Escher, Bach: An Eternal Golden Braid.* Vintage Books, New York, 1980.

Homewood, M., May, D., Shepherd, P., and Shepherd, R., "The IMS T800 Transputer", in *IEEE Micro*, Vol. 7(5), pp. 10-26, 1987.

INMOS, *Occam Programming Manual.* INMOS Ltd. publication OPS-002 000, July 1983.

INMOS, *Transputer Architecture Reference Manual.* INMOS Ltd. publication 72 TRN-048 01, 1985a.

INMOS, *IMS T414 Transputer Product Data.* INMOS Ltd. publication 72 TRN-049 01, 1985b.

Jaswon, M.A., and Symm, G.T., *Integral Equation Methods in Potential Theory and Elastostatics.* Academic Press, New York, 1977.

Jeng, G., and Wexler, A., "Isoparametric, Finite Element, Variational Solution of Integral Equations for Three-Dimensional Fields", in *International Journal for Numerical Methods in Engineering,* Vol. 11, pp. 1455-1471, 1977.

Jeng, G., And Wexler, A., "Self-Adjoint Variational Formulation of Problems Having Non-Self-Adjoint Operators", in *IEEE Transactions on Microwave Theory and Techniques*, Vol. MTT-26(2),pp. 91-94, Feb., 1978.

Johnson, O.G., Micchelli, C.H., and Paul, G., "Polynomial Preconditioners for Conjugate Gradient Calculations", in *Siam J. Numerical Analysis*, Vol 20 (2), pp. 362-376, April 1983.

Kershaw, D.S., "The Incomplete Cholesky-Conjugate Gradient Method for the Iterative Solution of Systems of Linear Equations", in *Journal of Computational Physics*, Vol. 26, pp. 43-65, 1978.

Kightley, J.R., and Jones, I.P., "A Comparison of Conjugate Gradient Preconditionings for Three-Dimensional Problems on a CRAY-1", in *Computer Physics Communications*, 37, pp. 205-214, 1985.

Klimpke, B.W., *A Two-Dimensional, Multi-Media, Boundary Element Method.* Masters Thesis, The University of Manitoba, Winnipeg, Manitoba, 1983.

Kung, H.T., "Why Systolic Architectures?", in *Computer*, Vol. 15(1), pp. 37-46, 1982.

Lang, S., *Linear Algebra.* Addison-Wesley Publishing Company, Don Mills, Ontario, 1972.

Lang, T, and Stone, H.S., "A Shuffle-Exchange Network with Simplified Control", in *IEEE Transactions on Computers*, Vol. C-25 (1), pp. 55-65, 1976.

Lean, M.H., *Electromagnetic Field Solution With The Boundary Element Method.* Ph.D. Thesis, The University of Manitoba, Winnipeg, Manitoba, July 1981.

Lean, M.H., and Wexler, A., "Accurate Field Computation With the Boundary Element Method", in *IEEE Transactions on Magnetics*, Vol. MAG-18(2), pp. 331-335, March, 1982.

Lean, M.H., and Wexler, A., "Accurate Numerical Integration of Singular Boundary Element Kernels Over Boundaries with Curvature", in *International Journal for Numerical Methods in Engineering,* Vol. 21, 211-228, 1985.

Li, Q., Klien, D., and Field Jr., W.B., "A Method for Solving Linear Equations on a Transputer System", in *Proc. of the Third Occam Users Group Meeting,* INMOS Corp., Chicago IL, 1987.

Madsen, N.K., Rodrigue, G.H., and Karush, J.I., "Matrix Multiplication by Diagonals on a Vector/Parallel Processor", in *Information Processing Letters*, Vol. 5(2), pp. 41-45, 1976.

McDonald, B.H., Friedman, M., and Wexler, A., "Variational Solution of Integral Equations", in *IEEE Transactions on Microwave Theory and Techniques,* Vol. MTT-22, No. 3, March 1974.

Meijerink, J.A., and van der Vorst, H.A., "An Iterative Solution Method for Linear Systems of Which the Coefficient Matrix is a Symmetric M-Matrix", in *Mathematics of Computation*, Vol. 31(137), pp. 148-162, Jan. 1977.

Mikhlin, S.G., *Variational Methods in Mathematical Physics.* MacMillan, New York, 1964.

Mirsky, L., *An Introduction to Linear Algebra.* Dover Publications, Inc., New York, 1982.

Moore, P., *Tech. Note 15: IMS B005 - Design of a disk controller board with drives.* INMOS Ltd. publication 72 TCH-015 00, 1986.

Nakonechny, R.L., *A Preconditioned Conjugate Gradient Method Using a Sparse Linked-List Technique For the Solution of Field Problems.* Masters Thesis, The University of Manitoba, Winnipeg, Manitoba, 1983.

Poltz, J., and Wexler, A., "Transmission-Line Analysis of PC Boards", in *VLSI Systems Design*, CMP Publications, Inc., pp. 38-43, March, 1986.

Pountain, D., *A Tutorial Introduction to Occam Programming.* INMOS publication, 72 OCC 046 00, March, 1987.

Stakgold, I., *Green's Functions and Boundary Value Problems.* John Wiley and Sons, New York, 1979.

Stone, H.S., "Parallel Processing with the Perfect Shuffle", in *IEEE Transactions on Computers*, Vol. C-20 (2), pp. 153-161, 1971.

Taylor, R., and Wilson P., "OCCAM: Process-Oriented Language Meets Demands of Distributed Processing", *Electronics*, McGraw-Hill, Nov. 30, 1982.

Vadher, A., Walker, P., *Tech. Note 10: IMS B003 - Design of a multi-transputer board.* INMOS Ltd. publication 72 TCH-010 00, 1986.

Varga, R.S., *Matrix Iterative Analysis.* Prentice-Hall, Inc., Toronto, 1962.

Webb, S.J., McKeown, J.J., and Hunt, D.J., "The Solution of Linear Equations on a SIMD Computer Using a Parallel Iterative Algorithm", in *Computer Physics Communications*, Vol. 26, pp. 325-329, 1982.

Wexler, A., *Finite Elements for Technologists.* Department of Electrical Engineering Technical Report TR-80-4, University of Manitoba, Winnipeg, Manitoba, Canada, 1980.

Whitby-Strevens, C., "The transputer", *The 12'th Symposium on Computer Architecture*, IEEE, pp. 292-300, 1985.

Wong, Y.S., "Solving Large Elliptic Difference Equations on the CYBER 205", in *Parallel Computing*, (to appear), 1987.

Wong, Y.S., and Jiang, H., "Approximate Polynomial Preconditioning Applied to Biharmonic Equations on Vector Supercomputers", in *NASA Technical Memorandum 100217*, ICOMP-87-5, 1987.

Zienkiewicz, O.C., *The Finite Element Method in Engineering Science.* McGraw-Hill, New York, 1971.

Zollenkopf, K., "Bi-Factorization: Basic Computational Algorithm and Programming Techniques", in *Large Sparse Sets of Linear Equations.* Reid, J.K., (ed.), Academic Press, New York, pp. 75-96, 1971.