Parallelization of algorithms by explicit partitioning

This item was submitted to Loughborough University's Institutional Repository by the author.

Additional Information:


Metadata Record: [https://dspace.lboro.ac.uk/2134/27004](https://dspace.lboro.ac.uk/2134/27004)

Publisher: © N.M. Bahoshy

Rights: This work is made available according to the conditions of the Creative Commons Attribution-NonCommercial-NoDerivatives 2.5 Generic (CC BY-NC-ND 2.5) licence. Full details of this licence are available at: [http://creativecommons.org/licenses/by-nc-nd/2.5/](http://creativecommons.org/licenses/by-nc-nd/2.5/)

Please cite the published version.
This item was submitted to Loughborough University as a PhD thesis by the author and is made available in the Institutional Repository (https://dspace.lboro.ac.uk/) under the following Creative Commons Licence conditions.

For the full text of this licence, please go to:
http://creativecommons.org/licenses/by-nc-nd/2.5/
Parallelization of Algorithms
by Explicit Partitioning

N. M. Bahoshy

A thesis submitted in partial fulfilment of the requirements for the award of
Doctor of Philosophy degree of Loughborough University

December 1992

©N. M. Bahoshy, 1992
<table>
<thead>
<tr>
<th>Loughborough University of Technology Library</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date</td>
</tr>
<tr>
<td>Class</td>
</tr>
<tr>
<td>Acc. No.</td>
</tr>
</tbody>
</table>

09922155
Declaration

I declare that the work presented in here was carried out by myself and the thesis is of my own composition, and that neither the work nor the thesis has been submitted to this or any other institution for a higher degree.

Nimatallah Marcel Bahoshy

December 1992
Acknowledgements

I thank Professor D. J. Evans, the Director of PARC, for his continual help to me in academic as well as in other matters. I acknowledge my enormous debt to his kindness and generosity.

I also thank Father Tutcher, the university chaplain, for his wise counsel and guidance throughout my years at Loughborough.

Words cannot express my gratitude to my family for the financial and emotional support which have been lavished on me. I pray only to be worthy of them.
Abstract

In order to utilize parallel computers, four approaches, broadly speaking, to the provision of parallel software have been followed:

1. Automatic production of parallel code by parallelizing compilers which act on sequential programs written in existing languages.

2. “Add on” features to existing languages that enable the programmer to make use of the parallel computer. These are specific to each machine.

3. Full blown parallel languages. These could be completely new languages, but usually they are derived from existing languages.

4. The provision of tools to aid the programmer in the detection of inherent parallelism in a given algorithm and in the design and implementation of parallel programs.
All these approaches are based on two implicit assumptions about what a parallel program is. One is completely machine dependent, viz. a parallel program is one that runs on a parallel computer. The second is that certain paradigms of parallel execution exist, so that a parallel program is one that contains instructions for implementing these paradigms.

In this thesis a definition of a parallel program is presented which is independent of any particular machine, and, following the conventions adopted in (sequential) structured languages, a distinction is drawn between the parts of a parallel program that control the order of execution and the other parts of the program. In order to implement such a distinction in practice, the manager/worker technique is adopted with the result that the control part of the parallel program becomes separate from the rest. This is called explicit parallel programming.

Not only does this separation allow the development of parallel control structures, it also allows the implementation of a general purpose harness for controlling the execution of an algorithm in parallel. Such a harness is designed and implemented. The user is required to input the tasks that make up the algorithm along with control functions, called sub-managers, and an overall schedule of the tasks. The harness is called exdata, standing for explicit data and task assignment.

The harness, which is implemented on the Sequent Balance parallel computer, is then tested. The performance of programs run under its control is compared with parallel programs written using the Sequent parallel library routines. In particular, a block diagonal bordered system of linear equations, which arises in the finite element method, is solved using various methods. Another example is provided by implementing Romberg integration for single and double integrals. It is shown how explicit data and function partitioning using the harness lead to natural parallel implementation. In particular it enables the partitioning of nested loops, which is not possible with existing
constructs.

The use of the harness is developed further by applying it to cases where the scheduling of tasks has to be prioritized. Another extension is provided by implementing a new strategy for partitioning tasks with good load balancing called scheduling with partial dependency.

Finally the partitioning of nested loops is generalized to arbitrary nested loops with arbitrary indices, and two implementations are given.
## Contents

1 Parallel Processing  
  1.1 Introduction ......................................... 1  
  1.2 Parallel Processing .................................... 3  
    1.2.1 Flynn's Classification ............................... 3  
    1.2.2 Händler's Classification ............................ 5  
    1.2.3 Shore's Classification ............................... 7  
    1.2.4 Kuck's Classification ............................... 9  
    1.2.5 Other Classifications .............................. 11  
  1.3 Parallel Architectures .................................. 12  
    1.3.1 Pipeline and Vector Processors .................... 12  
    1.3.2 Array Processors ................................... 14  
    1.3.3 Multiprocessors .................................... 15  
    1.3.4 Interconnection of Processors ...................... 17  
    1.3.5 Data Flow Machines ................................. 21  
  1.4 The Sequent Balance ................................... 23  

2 Parallel Programming  
  2.1 Introduction .......................................... 25  
  2.2 Issues in Parallel Software ............................ 27  
    2.2.1 Dependencies ....................................... 27  
    2.2.2 Parallelization of Loops ............................ 30
4.6.2 Data Partitioning in Strand .......................... 103
4.6.3 A Final Word on Strand ............................. 104
4.7 Timing Results ........................................... 106

5 Exdata: A Utility for Explicit Parallel Programming 110
5.1 Introduction .............................................. 110
5.2 The Input ................................................ 112
  5.2.1 Declarations Specific to exdata ..................... 113
5.3 A General Purpose Harness for Implementing exdata .... 115
  5.3.1 The Manager ......................................... 116
  5.3.2 Sub-Managers and Secretaries ....................... 118
  5.3.3 The Workers ......................................... 121
  5.3.4 Data Structures .................................... 123
  5.3.5 Initialization ....................................... 128
  5.3.6 Locks ............................................... 129
5.4 Processing the Input .................................... 130
  5.4.1 Introduction ....................................... 130
  5.4.2 The Declarations ................................... 132
  5.4.3 A Diversion: lex and yacc ......................... 134
  5.4.4 Lexical Analysis of the Declarations ............... 135
  5.4.5 Syntactic Analysis of the Declarations ............ 136
  5.4.6 Producing the C Functions ......................... 137
5.5 Other Considerations .................................... 138

6 Solving a Block Diagonal Bordered System 140
6.1 Introduction ............................................. 140
6.2 The Finite Element Method ............................. 142
  6.2.1 Structural Analysis ................................ 143
  6.2.2 Partial Differential Equations and Variational Methods 150
6.2.3 From Variational Methods to Finite Elements .................................. 158
6.2.4 Internal and Shared Nodes ......................................................... 164
6.3 Two Methods of Solving Linear Equations ......................................... 166
  6.3.1 Gaussian Elimination ............................................................... 167
  6.3.2 Choleski Factorization ............................................................ 176
6.4 Data Structures for Block Matrices ................................................. 184
6.5 Timing Results ........................................................................... 187
  6.5.1 Timing Results of the Gaussian Elimination Routines .................. 192
  6.5.2 Load Balancing in \textit{exdata} for Gaussian Elimination .............. 201
  6.5.3 Timing Results of the Choleski Factorization Routines ............... 204

7 Parallel Romberg Integration ............................................................. 211
  7.1 Introduction .................................................................................. 211
  7.2 Romberg Integration ..................................................................... 212
    7.2.1 The Trapezoidal Rule ............................................................. 212
    7.2.2 Romberg Integration ............................................................. 218
    7.2.3 Double Integrals .................................................................... 221
    7.2.4 Romberg Integration for the Double Integral ......................... 224
  7.3 Parallel Implementation of Romberg Integration ............................. 225
    7.3.1 The Single Integral ............................................................... 225
    7.3.2 Algorithm for the Double Integral ........................................ 234

8 Data and Function Partitioning, and Scheduling ................................... 241
  8.1 Introduction .................................................................................. 241
  8.2 Scheduling Different Sized Tasks .................................................. 243
    8.2.1 Priority Scheduling ............................................................... 245
  8.3 Scheduling with Partial Dependence .............................................. 254
    8.3.1 Practical Considerations ......................................................... 261
    8.3.2 Partial Dependence with \textit{Exdata} ........................................ 265
8.3.3 Partial Dependence with the Parallel Library Routines 271
8.3.4 The Rôle of Synchronization in Partial Dependence 275
8.3.5 Scheduling with Partial Dependence and Loop Spreading 276
8.3.6 Gaussian Elimination with Partial Dependence 278

8.4 Nested Loops and Sub-Managers 280
8.4.1 A universal Mapping for Nested Loops 285
8.4.2 An Explicit Partitioning Approach 289
8.4.3 Example 1 293
8.4.4 Example 2 294
8.4.5 Sub-Managers as Loop Control Statements 296

9 Conclusions 297
9.1 Conclusions 297
9.2 Suggestions for Further Work 301

A Appendix 303

Bibliography 304
Chapter 1

Parallel Processing

1.1 Introduction

The field of parallel computing can be divided into three areas: parallel hardware, parallel software, and parallel algorithms. This thesis is concerned with the second of these areas, i.e. with the provision of software to enable the encoding of parallel algorithms for execution on parallel computers.

It should be obvious that, just as in uniprocessor serial computing, these three areas are not completely separate from each other. The hardware plays a part in the development of software, which in turn affects the design of algorithms. And it can also be said that the need for certain kinds of algorithms affects the development of languages and software tools; also that the need for such languages, for example for logic programming, can affect the design and development of the hardware.

This chapter describes briefly the physical properties of parallel computers. The description is necessarily brief because hardware design and considerations did not form part of the work presented in this thesis. At the end of this chapter the hardware of the Sequent Balance computer is described in a little more detail since all the ideas presented here were implemented on
such a machine.

Throughout this thesis a distinction is maintained between an algorithm and its implementation in a particular language. Chapter 2 is a broad survey of developments in parallel software, by which is meant software facilities for making use of parallel computers, and includes parallel languages, extensions to existing languages, automatic parallelizing compilers, and tools that aid in the detection of inherent parallelism in a program, as well as others that aid in the design and/or construction of parallel programs. It will be seen that most of these developments are very closely linked to specific architectures and even to specific machines.

Chapter 3 then introduces and defines the basic concepts involved in parallel programming, as well as some software paradigms of parallel programming. For the first time a distinction is made between the control of execution and what a program actually does — its underlying algorithm, and using this distinction a definition of a parallel program will be given.

Chapter 4 explores further this idea of separating parallel control from the underlying algorithm, and using the manager/worker technique it is shown how such a separation can be achieved in practice. The advantages of such an approach are given.

Chapter 5 gives the details of an implementation of a general purpose harness for the execution of a parallel program which is input to the harness in the form of tasks and a schedule for these tasks. (The meaning of the word “schedule” here is different from that usually associated with it, and this will be explained and justified also.)

Chapters 6 and 7 describe in detail two applications in the use of the harness. The performance of the harness is assessed and compared with that of programs written without the harness.

Chapter 8 develops further the use of the harness, and in particular it is shown how its use can lead to new ideas in scheduling; this is done through
the presentation of a new scheduling strategy and its implementation with the harness. This chapter also presents a new technique for parallelizing nested loops.

Chapter 9 concludes the thesis with a résumé of the ideas and the work presented and a discussion of their possible future development.

1.2 Parallel Processing

Put simply, parallel processing means performing several tasks at the same time, in contrast with serial processing where, at any given moment, only one task can take place. In order to clarify and build on this idea, systems for classifying parallel processing have been proposed. Five such classifications are given below.

1.2.1 Flynn’s Classification

Flynn’s classification system [25, 26] is based on the way in which instructions are applied to data. To this end, an "instruction stream" is defined as a sequence of instructions executed by the computer, and a "data stream" is defined as a sequence of data on which the instructions are executed. By considering the multiplicity of such streams, four categories, or classes, of processing emerge: (the accompanying diagrams are from [42])

SISD (= single instruction stream single data stream)

This class represents most ordinary serial computers, where instructions are executed sequentially. However, the instructions may be overlapped by pipelining (see next section), and there might be a multiplicity of functional units, but always under one control unit. See figure 1.1(a).

SIMD (= single instruction stream multiple data stream)
Figure 1.1: Pictorial representation of Flynn's classification.
In this class several processor units receive and execute the same instruction from a central control unit, but each processor unit operates on different data from the others. See figure 1.1(b).

**MISD** (= multiple instruction stream single data stream)

This class is illustrated in figure 1.1(c) where it is seen that several processors, each with its own instructions, operate at the same time on the same data. This class is not talked about much in the literature; for example [70] covers this whole class by the single sentence: 'No computers fit into this category'.

**MIMD** (= multiple instruction stream multiple data stream)

This is the most general of all four classes. Here, several processor units can simultaneously execute different instructions on different data. This is illustrated in figure 1.1(d)

### 1.2.2 Händler's Classification

Händler[37] proposed a notation for classifying parallel processing at three different levels:

- The processor control unit (PCU). This is the processor or the CPU.
- The arithmetic logic unit (ALU). This corresponds to a processing element in array processors (see next section).
- Bit-level circuit (BLC). This corresponds to logic circuitry required to carry out 1-bit operations in the ALU.

A computer system $C$ is then described by a triple made up of six independent integers:

$$ T(C) = \langle K \times K', D \times D', W \times W' \rangle $$  \hspace{1cm} (1.1)
where

\[ K = \text{the number of PCUs within } C \]
\[ K' = \text{the number of PCUs that can be pipelined} \]
\[ D = \text{the number of ALUs under the control of a single PCU} \]
\[ D' = \text{the number of ALUs that can be pipelined} \]
\[ W = \text{the length, in bits, of an ALU word} \]
\[ W' = \text{the number of pipeline segments in all the ALUs} \]

or in a single processing element

if any of \( K', D', \) or \( W' \) is 1, it is not written because it is not meaningful to talk about a pipeline of one unit.

Where it is necessary to describe the operation of a system \( C \) in which different kinds of processors are used, the operator \( \times \) is used to link together the various descriptions. For example the Control Data 6600 [70] has one CPU. The ALU has ten specialized functional units, and all ten can be pipelined together, and each unit has a word 60 bits long. In addition, there are ten peripheral input/output processors that can work in parallel, and each of these input/output processors has a single ALU with a word length of 12 bits. The CDC6600 is therefore described by two tuples linked with the operator \( \times \) as follows:

\[
T(\text{CDC6600}) = T(\text{central processor}) \times T(\text{I/O processors})
\]
\[
= (1, 1 \times 10, 60) \times (10, 1, 12)
\]

If the system \( C \) can be connected in different ways, then its descriptions are linked together with the operator \( + \). For example the C.mmp [42] system is made up of 16 PDP-11 minicomputers, the word length of each being 16. It is also possible to connect up the system so that it behaves like an SIMD computer by controlling all 16 minicomputers by a single Central Unit. To
describe such a system fully, descriptions of its two modes are linked together using +, thus

\[ T(C.mmp) = (16, 1, 16) + (1, 16, 16) \]

1.2.3 Shore's Classification

This classification [75] is based on the organization of a computer. For this classification memory is divided into instruction memory (IM) and data memory (DM). The other constituents of a computer are, as in Flynn's classification, the control unit and the processor unit. The processor unit is either horizontal or vertical. If the data memory is thought of as a two dimensional array, then a horizontal processor unit is one that reads one word, or a row, at a time from the DM, which is now called a word-slice DM. A vertical processor unit is one that reads the first bit of every word in the DM, or the first column of the array; the DM is now said to be bit-sliced.

According to this classification, computers can be divided into the following six classes. (Figure 1.2, which is from [40], illustrates these.)


This is just the usual serial von Neumann architecture with one control unit, and one processor unit. The processor is horizontal and the DM is therefore word-sliced. The PU may be vector or scalar, pipelined or not. See figure 1.2(a).


This is the same as machine I except that the processor unit is vertical, and the DM is bit-sliced. See figure 1.2(b).

3. Machine III:

This is machine I and machine II put together. There is still only one control unit, but here there are two processor units, one of which is
(a) Machine I: word-serial, bit-parallel

(b) Machine II: word-parallel, bit-serial

(c) Machine III: orthogonal

(d) Machine IV: unconnected array

(e) Machine V: connected array

(f) Machine VI: logic-in-memory array

Figure 1.2: Pictorial representation from [40] of Shore’s classification.
vertical and the other horizontal, so that the DM must be both word- and bit-sliced. Such a machine is also called an orthogonal machine in [74]. See figure 1.2(c).


In this class a processing element (PE) is a processor unit and its data memory. There is one control unit which is used to control several PEs. There is no communication between the PEs except through the CU. See figure 1.2(d).


This is the same as machine IV, but here each processing element can communicate directly with its immediate neighbour. See figure 1.2(e).


Unlike all the other classes, where the DC is separate from the PU, in this class the logic circuitry is distributed throughout the DM. See figure 1.2(f). An associative memory is an example of this class.

1.2.4 Kuck's Classification

This classification [52] is similar to Flynn's. Both of them use the fact that a stream can be either single or multiple as a basis for the classification. But whereas Flynn uses instruction and data streams, Kuck uses instruction and execution streams for the classification. Strictly speaking, this classification applies only to the control unit of a computer system. Given such a system, the (global) control unit refers to all the hardware involved in preparing instructions for execution. The control unit is envisaged as having as its input instruction streams, which the unit then decodes to produce execution streams as its output (see figure 1.3). If a computer has several control units, then the global control unit refers to all of them collectively.
A multiple instruction stream implies that several "programs" are being handled by the control unit. A multiple execution stream means that the control unit is able to output more than one operation at the same time.

Based on the above, computer systems, or rather their control units, are classified into the following four classes:

**SISE** (= single instruction stream single execution stream)

To this class belong all uniprocessor serial computers.

**SIME** (= single instruction stream multiple execution stream)

To this class belongs, for example, the CDC 6600 computer mentioned earlier and classified as \((1,1 \times 10,60)\) using Händler's classification.

**MISE** (= multiple instruction stream single execution stream)

To this class belongs, for example, the CDC 6600 mentioned earlier and classified as \((10,1,12)\) using Händler's classification.

**MIME** (= multiple instruction stream multiple execution stream)

To this class belong all multiprocessors, such as the Sequent computers.

In order to be able to describe and include vector processors in the classification, the concepts of scalar and array instruction streams, and scalar and array execution streams are introduced. Thus, each of the above four classes can be further divided into four classes (scalar scalar, scalar array, array scalar, and array array) giving 16 classes in all. The relationship between these 16 classes and Flynn's four classes is set out in [5].
Finally, it is worth mentioning that in [65] and [63] Kuck's 16 classes are reduced to just 4, based solely on the execution streams, without considering the instruction streams at all. These four classes are:

**SES** (= single execution scalar) Which are uniprocessor machines without pipelining.

**SEA** (= single execution array) Which are uniprocessor machines which are pipelined, or with a vector processor, or have an array of processing elements.

**MES** (= multiple execution scalar) Which are multiprocessor machines.

**MEA** (= multiple execution array) Which are multiprocessor machines with vector processors.

### 1.2.5 Other Classifications

In addition to the classifications mentioned above, there are others. For example [24] attempts to measure the degree of parallelism and to use this as a basis for classification. Hockney and Jesshope [40] introduce a rather involved algebraic notation for describing computer architecture and then use this for a basis of classification. While [51] discusses the notion of classifying computers according to various criteria, such as data and granularity, and from these he builds up a classification tree. It is fair to say that the oldest classification mentioned, that of Flynn's, is always mentioned in the literature; and though it is invariably criticized for its inability to distinguish pipelined computers from those that are not pipelined, and for producing a class to which no computer system belongs (the MISD), this classification is widely used to describe parallel computers.
1.3 Parallel Architectures

1.3.1 Pipeline and Vector Processors

Instruction pipelining means breaking down the process of executing an instruction into several stages, and arranging these stages into a pipeline so that the output of each stage becomes the input to the next stage. If the same instruction is to be repeated several times, the pipeline gets filled up so that at any one time there are \( n \) different instances of the instruction being executed, where \( n \) is the number of stages into which the instruction is broken down. This is best illustrated by an example (the following example is taken from [62]). The operation of addition can be broken down into four stages. Let the operands be \( A \) and \( B \) whose binary representations are given by

\[
A = P \times 2^Q \\
B = R \times 2^S
\]

then the four stages are

**Stage 1** Compare the exponents \( Q \) and \( S \) to get \( Q - S \).

**Stage 2** Shift \( P \) by \( Q - S \) places with respect to \( R \).

**Stage 3** Add \( P \) and \( R \).

**Stage 4** Normalize the sum by shifting it until the leading non-zero digit is next to the binary point.

Assume that each of these stages takes \( T \) seconds; then, on a serial, non-pipelined processor the operation of addition takes \( 4T \) seconds. If \( N \) addition operations are to be carried out they would take \( 4NT \) seconds.

If the stages are now pipelined, and \( N \) addition operations are to be carried out, it is clear that the second of these operations can start as soon
Figure 1.4: A pipeline. The shaded area shows the stages when the pipeline is empty.

as the first is at stage 2, and so on with the other stages and operations. This is shown in figure 1.4. The state of the pipeline is seen at any given instant in time by looking at a line parallel to the vertical axis. After $\frac{3r}{T}$ seconds have elapsed, the pipeline becomes full and there are four instances of the addition operation at any one time thereafter.

In general, if there are $n$ stages in the pipeline, then the time it takes to process one instruction is $nT$. To process $N$ instructions on a serial processor without pipelining takes, therefore, $nNT$ seconds. With pipelining the time becomes

$$nT + (N - 1)T$$

because it takes $nT$ seconds to output the result of the first instruction, after which a result is produced after each $T$ seconds.

The benefit, in reduced time of processing, that pipelining can achieve depends on how many times an instruction is to be repeated. One way that this can be achieved is by applying the operation to a vector quantity. The (scalar) values of the vector are streamed into the pipelined processor. This is called *data pipelining* in [70], and the processor is then called a pipelined
vector processor. Examples of pipelined vector processors are the Cray-1 and the Cyber-205 of Control Data Corporation. A very good reference for pipelining is [38].

The parallelism achieved by pipelining is called *temporal* parallelism in [42].

### 1.3.2 Array Processors

An array processor is one which contains several arithmetic logic units, called processing elements (PE). All the PEs are under the control of a single control unit (CU), and all are synchronized in a lock-step fashion. Each PE has its own local memory. If there are $P$ such elements and each one has different data in its memory, an instruction can be performed on $P$ values of the data in the time it would take a serial processor to perform the instruction on just one value. Array processors thus allow *spatial* parallelism [42], and another way of achieving vectorization. Figure 1.5 is a representation of an array processor.
The performance of an array processor depends on how quickly data can be read into each PE’s memory, and so the way that these elements are connected is very important. For the different ways in which this can be done and the advantages of each way see [70] and [24].

Examples of array processors are the Illiac IV [10], the DAP [61], and the Massively Parallel Processor [11], which contains 16,384 PEs arranged as a grid of 128×128.

1.3.3 Multiprocessors

These consist of a collection of fully programmable processors, each of which can be programmed separately. Depending on the way memory is allocated to and used by the processors, and on the way in which the processors are connected to each other, multiprocessors are divided into shared memory machines and distributed memory machines.

In a shared memory machine, all processors share access to a single virtual memory, as shown in figure 1.6. In a distributed memory machine each processor has its own memory, and the processors are connected to each other by some form of switching mechanism, as shown in figure 1.7. The diagram shows that each processor has its own memory, called cache memory. This is provided in order to reduce the amount of data that has to be transferred to and from the main memory. Each processor works with a copy of the data in its cache memory; when this is updated by one processor, and another processor needs to use the updated values, the system is responsible for updating, or refreshing, the copy in the cache.

Shared memory machines are also described as tightly coupled, and distributed memory machines are also described as loosely coupled. This is done in, for example, [62]. Elsewhere, in [70] for example, the term shared memory machine is used to describe both types of computer described above,
and they are distinguished by being called loosely coupled and tightly coupled. In [70] a distributed memory machine is used to refer to one where all communication between the processors is carried out by messages sent from one processor to another.

Examples of tightly coupled shared memory computers are the Sequent Balance, where all processors share a bus for access to the shared memory; and the C.mmp, where access to the shared memory is done by a crossbar switch [31]. Examples of loosely coupled computers is the HEP [42] ([70] gives the HEP as an example of of a tightly coupled machine); and the C.m* [79, 76]. The iPSC [71] is an example of a multiprocessor where all communication between processes is carried out using message passing.
Figure 1.8: Two dimensional mesh networks [70]. (a) A mesh with no wrap-around. (b) A mesh with wrap-around in same row and column. (c) A mesh with wrap-around between rows and columns.

1.3.4 Interconnection of Processors

Figures 1.5, 1.6 and 1.7 above are simplified sketches of the different types of parallel architectures. The way in which processors are organized and connected to each other and to the memory can be classified into five main types [82].

Mesh  The processors are arranged in an n-dimensional mesh. Figure 1.8(a) shows a 2-dimensional mesh. Each internal processor can communicate directly with the $2n$ neighbouring processors. The processors at the extremities of the mesh may also be connected to each other in the same row and column as in figure 1.8(b), or between successive rows and columns in a wrap-around fashion as shown in figure 1.8(c).

A mesh is suitable for certain algorithms such as sorting and matrix multiplication, but not for others. For details see [70, 33].

Pyramid  The processors here are connected to form a pyramid as shown in figure 1.9. If $p$ is the size of the pyramid, then the height is $\log_4 p$. The processors are connected so that on each level a mesh is produced. The total number of processors is $(4p/3) - (1/3)$. The number of processors at the base
Figure 1.9: A Pyramid Network [70].

Shuffle-Exchange Network The idea behind the shuffle-exchange network is that, if \( n \) processors are connected, then an item of data requires at most \( \log_2 n \) shuffle operations to get from any processor to any other processor. If \( n = 2^k \), the processors are numbered from 0 to \( n - 1 \). There are two kinds of connection, the shuffle and the exchange. Exchange connections are made between every pair of processors whose numbers differ in their last significant bits only, i.e. the pairs (0,1), (2,3), etc. The shuffle connection links all pairs \((i, 2i \mod (n - 1))\), except for \( i = n - 1 \) which is connected to itself. Figure 1.10 shows a shuffle-exchange network with \( n = 8 \).

If an item of data is at processor \( i \), then \( k \) shuffle operations will bring it back to processor \( i \). The route taken by the item of data is called the
Figure 1.10: A Shuffle-Exchange Network [70]. The double headed arrows the exchange connections, and the single headed arrows show the shuffle connection.

Necklace of \( i \). The longest necklace is thus of length \( k = \log_2 n \).

**Butterfly**  In a butterfly network there are \((k + 1)2^k\) processors divided into \((k + 1)\) rows each with \(2^k\) processors. The way they are connected is as follows: Let the rows be numbered 0 to \( k \), and let \( p(i,j) \) denote the \( j \)th processor on the \( i \)th row. Then, for \( i > 0 \), \( p(i,j) \) is connected to the two processors \( p(i - 1,j) \) and \( p(i - 1,m) \), where \( m \) is the number obtained by inverting the \( i \)th most significant bit in the binary representation of \( j \). This is shown in figure 1.11 for the case \( k = 3 \). The pattern produced by the interconnection gives this network its name.

**Hypercube**  A hypercube network, as the name implies, is a network whose geometric shape is a hypercube. The processors sit at the corners and the sides of the hypercube form the connections between the processors. In two dimensions, the hypercube is a square; in 3 dimensions the hypercube is a cube; and in \( n \) dimensions it is an \( n \)-dimensional hypercube. Figure 1.12 depicts a 4-dimensional hypercube.

A hypercube connection is equivalent to a butterfly connection with the columns collapsed into a single point, to give the hypercube depicted in figure 1.13 which is another way of representing the 4-D hypercube of figure 1.12.
Figure 1.11: A Butterfly Network with \( k = 3 \)\cite{70}.

Figure 1.12: A 4-D Hypercube Network.
1.3.5 Data Flow Machines

All the previous architectures considered are control driven, so that the order in which instructions are executed is determined by a control unit. This order is determined by the way a program is written. In a data flow machine, all statements of a program are considered "ready" to be executed, and the order of the execution of the statements depends entirely on the availability of data required by the statements, hence the control of execution is determined by data flow. (See the section on Strand in the next chapter.) As far as parallelism is concerned, it should be clear that any inherent parallelism is immediately exposed by data flow execution. For example, the assignment

\[ a = b \times c - d \times e \]

can be executed on a data flow machine as shown in figure 1.14 so that \( b \times c \) and \( d \times e \) are performed simultaneously, without any need for parallel commands.

There are no commercial data flow machines. An experimental model has been built in Manchester [36] and figure 1.15 is an outline of its architecture taken from [62].
Figure 1.14: An illustration of data flow execution.

Figure 1.15: Outline of the Manchester data flow computer.
1.4 The Sequent Balance

The computer used throughout the work presented here is the Sequent Balance B8[72]. It is a multiprocessor computer with 12 32-bit microprocessors, all of them identical to each other. The memory is shared, and access to the memory by the processors is via a single high speed bus. The bus, in addition, connects the I/O controllers.

Each processor operates independently and can execute both user code and operating system code.

The operating system is Dynix, which is derived from Unix. In particular, the scheduler in Dynix has the choice of any one of the 12 processors to allocate tasks to, so that even if no parallel program is being run, the total work load is distributed amongst the available processors. This is called transparent multiprogramming.

Communication between processors and external storage devices, the disk and magnetic tape, as well as any other external device, is managed by System Link and Interrupt Controller (SLIC) chips, and these are connected to the storage devices via a special bus.

In addition to the shared memory, each CPU also has 8K of cache RAM. This contains copies of the most recently read blocks of memory, in order to reduce the number of memory lookups, and so reduce the traffic on the bus.

Locks (see chapter 3) are provided by the hardware also. Part of the memory is called ALM (atomic lock memory) and this provides atomic test-and-set operations for implementing mutual exclusion.

Figure 1.16 is an outline of the architecture of the Sequent Balance.
Figure 1.16: The Sequent Balance
Chapter 2

Parallel Programming

2.1 Introduction

In the previous chapter parallel computers were introduced. In order to utilize these computers and make use of their parallel capabilities, software must be provided. In this chapter parallel software will be introduced. As mentioned in the introduction to the previous chapter, the development of parallel software is partly dependent on the hardware available; for example, in the most general model of parallel architecture, the MIMD (with or without pipelining and array processing), there must exist language constructs that allow the programmer to program the individual processors and to define the data on which they are to operate. At the same time, software development also affects the design of the hardware; for example, data flow architectures are proposed in order to allow for the efficient implementation of data flow languages. In addition, it must also be borne in mind that computers, parallel or otherwise, are there to solve problems, in other words to implement algorithms; and since the implementation is always in some high level language, parallelism on the algorithm level needs must also affect software for parallel software. Based on these observations, it is possible
to regard parallel software, considered merely as software to utilize parallel computers, as being of two types. The first is constructs that are specific to each type of computer architecture, or even to each specific machine; and the other is constructs that can implement parallel paradigms derived from algorithmic considerations alone. Although in the end all software must run on some machine, it is useful to keep this distinction in mind when surveying the state of parallel software.

It should also be observed that parallel software need not be software that is produced by a programmer directly, but by some parallelizing algorithm. The programmer here simply writes a program in an existing (serial) language, and leaves it to a parallelizing compiler to produce object code which is parallel.

Thirdly, software tools have been produced in order to aid the programmer in turning an algorithm into a parallel program. These tools have been designed to help at different stages of the programming process; from the detection of parallelism, called inherent parallelism, in an algorithm; to restructuring an algorithm to increase its degree of parallelism; right up to writing code suitable for a specific machine.

With the three observations above firmly in mind, this chapter will introduce the development of parallel software. This will begin by considering first software from a programmer's point of view. The meaning of a parallel program is, put simply, one in which it is possible to execute more than one instruction at the same time. If this is at all possible, it must imply that some of these instructions do not depend on each other. So the discussion begins by describing dependencies. Next, it has been observed by trial, that loops provide a rich source of parallelism in a program; for this reason different loops will be considered and ways in which their parallelism can be exploited will be given.

The following section will discuss automatic parallelizers and the ideas
behind them. Such a discussion is useful because these ideas recur in all parallel programming.

In the next section several examples of constructs that are added to existing languages to enable them to make use of parallel machines will be given. Specifically, Sequent Fortran and Dynix's parallel library routines will be presented in some detail. The latter will be used extensively throughout this thesis, and programs written using these routines will be compared with those written using the techniques introduced and implemented in this thesis.

There exist full blown parallel languages and two such languages will also be presented. Occam was designed for the transputer, but even so some of the ideas behind it are those that must lie behind any parallel software. Strand is a more recent language derived from logic programming languages, especially Prolog, but claimed to be a general purpose programming language.

Parallel software tools are then considered. A brief survey of these tools will be given.

The chapter will conclude by attempting to sum up the aims of all these developments and the ideas behind them and to bring these together in order to see a possible way forward for parallel software. This is the basis of the work presented in the thesis.

2.2 Issues in Parallel Software

2.2.1 Dependencies

Data and control dependencies are topics that first arose in the field of automatic optimization of code by compilers [2]. They are essential for automatic parallel compilers and an understanding of them is also at the heart of all parallel programs.

There are four types of dependency [65, 60, 84, 85]:

27
True or flow dependency  Consider the following two statements

\[ s_1: \quad A = B \times C \]
\[ s_2: \quad D = A \times A \]

then clearly statement \( s_2 \) cannot be executed before statement \( s_1 \) has finished because \( s_2 \) needs the value of \( A \) assigned by \( s_1 \). This is called true or flow dependency, and is denoted by \( s_1 \delta s_2 \).

Antidependence  Consider the two statements

\[ s_1: \quad A = B \times C \]
\[ s_2: \quad B = D \times 4 \]

In this case \( s_2 \) must execute after \( s_1 \) in order for the execution to be correct. This is called antidependence and is denoted by \( s_1 \bar{\delta} s_2 \).

Output dependency  This may be illustrated by the statements

\[ s_1: \quad A = B \times C \]
\[ s_2: \quad D = A \times A \]
\[ s_3: \quad A = E + F \]

Here, \( s_1 \delta s_2 \) and \( s_2 \bar{\delta} s_3 \). Consider \( s_1 \) and \( s_3 \). Clearly, \( s_3 \) must be executed after \( s_1 \), for otherwise the variable \( A \) would have the wrong value \((B \times C)\) at the end of the three statements. This is called output dependency, and is denoted by \( s_1 \delta^\circ s_3 \).

Control dependency  Consider the following segment of a program

\[ s_1: \quad A = B \times C \]
\[ s_2: \quad \text{if } (X \geq 0) \{ \]
\[ s_3: \quad \quad A = A \times A \]
\[ \} \]
\[ s_4: \quad D = A \times 3 \]
With the three types of dependencies considered so far, it is true to say that \( S_1 \delta S_4 \) and \( S_3 \delta S_4 \). However, when the program is actually run, then \( S_3 \delta S_4 \) is the only one of the two relations which is true if \( X \geq 0 \). To distinguish this dependency from the others, it is called control dependency, and denoted by \( s_2 \delta^c s_4 \).

**Order of execution, and input and output sets** In order to formalize the relations of dependency above, order of execution, and input and output sets are defined.

Given two statements, \( s_i \) and \( s_j \), of a program, \( s_i \) is said to precede \( s_j \), denoted by \( s_i < s_j \), if in the serial order of execution of the program \( s_i \) is executed before \( s_j \). If \( s_i < s_j \) then \( s_j > s_i \).

The input set of a statement \( s \), denoted by \( \text{IN}(s) \), is the set of variables that appear on the right hand side of the assignment operator in the statement. The idea can clearly be extended to a block of statements.

The output set of a statement \( s \), denoted by \( \text{OUT}(s) \), is the set of variable(s) that appear on the left hand side of the assignment operator in the statement. Again, the idea can be readily extended to a block of statements.

Given two statements or block of statements, \( s_1 \) and \( s_2 \), then the Bernstein conditions [13] for \( s_1 \) and \( s_2 \) to be executed in parallel such that the execution of the program remains correct are

\[
\text{IN}(s_1) \cap \text{OUT}(s_2) = \emptyset \tag{2.1}
\]
\[
\text{IN}(s_2) \cap \text{OUT}(s_1) = \emptyset \tag{2.2}
\]

Using these notions it is now possible to define exactly the four types of dependency:

**flow dependency** \( s_1 \delta s_2 \) iff \( \text{OUT}(s_1) \cap \text{IN}(s_2) \neq \emptyset \) and \( s_1 < s_2 \).

**antidependency** \( s_1 \tilde{\delta} s_2 \) iff \( \text{IN}(s_1) \cap \text{OUT}(s_2) \neq \emptyset \) and \( s_1 < s_2 \).
output dependency \( s_1 \delta^o s_2 \) iff \( \text{OUT}(s_1) \cap \text{OUT}(s_2) \neq \emptyset \) and \( s_1 < s_2 \).

control dependence \( s_1 \delta^c s_2 \) iff \( s_1 \) is a conditional statement and \( s_2 \) is in the scope of \( s_1 \).

So far only statements have been considered. When the statements are within loops, then the notation and definitions have to be extended. See the references given at the beginning of this section. For the clearest explanation see [84]. [86, 8, 9] explain how these dependencies, when they occur within loops, can be manipulated automatically by parallelizing compilers.

2.2.2 Parallelization of Loops

Only a few of the transformations of loops for the purpose of turning them into parallel loops will be given here. The purpose is to give a general idea of the subject only. The references for this section are the same as those given above for dependencies.

A parallel loop Before giving the transformations themselves, it will be stated first what is meant by a parallel loop. Throughout the literature a distinction is made between a loop that can be vectorized and one that can be concurrentized. For the purpose of illustrating parallel software, however, no such distinction will be made, and by a parallel loop will be understood a loop whose iterations are independent of each other. Take for example the loop

\[
\text{for } i = 1 \text{ to } n \{ \\
\quad A(i) = B(i) \\
\} 
\]

Each iteration of the loop is independent of all other iterations, so the iterations can all be performed at once on \( n \) processors if that many processors
are available; it can be vectorized if a vector processor is available; and if less than \( n \) processors are available then the iterations can be distributed among the available processors in any way; the results would be correct.

**Removal of output dependence and antidependence** Consider the example given above to illustrate output dependence

\[
\begin{align*}
\text{s}_1: & \quad A = B \times C \\
\text{s}_2: & \quad D = A \times A \\
\text{s}_3: & \quad A = E + F
\end{align*}
\]

where \( s_1 \) \( \delta \) \( s_3 \). This dependence can be removed by a technique called *variable renaming*. Use another variable, say \( A_1 \), to replace \( A \) in \( s_1 \) and \( s_2 \) to obtain

\[
\begin{align*}
\text{s}_1: & \quad A_1 = B \times C \\
\text{s}_2: & \quad D = A_1 \times A_1 \\
\text{s}_3: & \quad A = E + F
\end{align*}
\]

It is seen now that the output dependence \( s_1 \) \( \delta \) \( s_3 \) no longer holds.

This technique can be applied to loops as well, but there it is a more difficult problem. Another technique is therefore used called *node splitting*. The loop

\[
\text{for } i = 1 \text{ to } n \{ \\
\text{s}_1: & \quad A(i) = B(i) \times C(i) \\
\text{s}_2: & \quad D(i) = A(i) \times A(i + 1) \\
\}
\]

is not a parallel loop because the value of \( D(i) \) depends on \( A(i + 1) \), and so it is not possible to modify \( A(i + 1) \) before iteration \( i \) has been performed. The solution is to remove this dependency by copying the original values of the array \( A \) into another, \( AA \) say, as follows:
for $i = 1$ to $n$ 
\begin{align*}
  s_1: & \quad AA(i) = A(i + 1) \\
  \end{align*}

for $i = 1$ to $n$ 
\begin{align*}
  s_2: & \quad A(i) = B(i) \ast C(i) \\
  s_3: & \quad D(i) = A(i) \ast AA(i) \\
  \end{align*}

The second loop must be performed after the first loop, but each loop is now a parallel loop.

**Loop interchanging** In a nested loop it is sometimes possible to interchange the loops, i.e. to make the inner loop the outer loop, and vice versa, and to create thus a parallel outer loop.

Consider

for $i = 2$ to $n$ 
\begin{align*}
  & \text{for } j = 1 \text{ to } n \{ \\
  & \quad \quad A(i, j) = A(i - 1, j) \ast B(i) \\
  & \text{\} } \\
  & \\}
  \end{align*}

The inner loop is a parallel loop and can therefore be executed in parallel. However, on a multiprocessor machine, it would be more profitable to parallelize the outer loop because that would increase the amount of work each processor does in parallel with the others (see *granularity* in chapter 3). In this particular example, the loops can be interchanged without affecting the result, thus

for $j = 1$ to $n$ 
\begin{align*}
  & \text{for } i = 2 \text{ to } n \{ \\
  & \quad \quad A(i, j) = A(i - 1, j) \ast B(i) \\
  & \text{\} } \\
  & \\}
  \end{align*}
and the outer loop is now a parallel loop, so its iterations can be performed in parallel, where each iteration is the whole of the inner loop.

Strip mining  Strip mining is a technique for turning a loop into a nested loop with the aim of increasing the amount of work in each iteration. It is so called because the loop can now be 'excavated in chunks, just as a strip mine is excavated in shovelfuls'[60].

The loop

\[
\text{for } i = 1 \text{ to } n \{ \\
\quad A(i) = B(i) \times B(i) \\
\quad C(i) = D(i) \times D(i) \\
\}
\]

can be turned with strip mining into

\[
\text{for } j = 1 \text{ to } n, \text{ step } 32 \{ \\
\quad \text{for } i = j, \min(j + 31, n) \{ \\
\quad \quad A(i) = B(i) \times B(i) \\
\quad \quad C(i) = D(i) \times D(i) \\
\quad \}
\}
\]

and the outer loop is a parallel loop, where each iteration is made up of 32 iterations of the inner loop.

Run-Time dependence checking  In an indirectly indexed loop, that is one where its index is used in a function to work out the indices of array variables, or where the loop index is used to look up the array variables' indices in another array, it is not possible to decide, just by looking at the code, whether or not the loop is parallel. As an example, take the loop

\[
\text{for } i = 1 \text{ to } n \{
\}
\]
\[ j = \text{table}(i) \]
\[ A(i) = B(j) \times C(j) \]

The solution to this problem is to carry out the dependency check at run time. For details of how this can be performed see [63].

### 2.3 Automatic Parallelizers

Automatic parallelizers are in essence program restructurers. The input is a serial program and the output is object code for execution on a parallel machine. Just as compilers turn high level code to runnable object code for a specific machine, so these turn serial programs into runnable parallel object code. Compilers also perform some program restructuring in order to improve the efficiency of the resulting code [2], but, of course, a complete analysis of a program with the aim of producing the "best" code is not possible, so that all these restructuring activities are based on recognizing certain constructs or uses of variables and applying the relevant transformation. The same is true of these automatic parallelizers, but here the scope for restructuring is much wider than that in serial compilers.

All automatic parallelizers must start by examining the data dependencies that exist between the statements of the program. These have already been presented above. From this analysis a data dependence graph (DDG) is built. From this graph it can be determined if restructuring the program can increase the degree of parallelism in the program. For example, in the discussion on loops above, it was observed that antidependence and output dependencies can be removed by variable renaming and node splitting. As well as increasing the degree of parallelism, common compiler optimizations can also be applied. Such restructuring is independent of the target machine. There are also restructuring techniques that are employed for specific machine
architectures. For example whether to optimize a loop for a vector processor or for a multiprocessor. As mentioned earlier, these transformations, like the optimizing transformations applied by compilers, are, to give them a proper technical description, not confluent [41], or they do not satisfy the lattice condition [50, 39]. In other words, given a program $P$ on which it is possible to apply the transformations $T_1$, $T_2$, and $T_3$, in that order, to yield a program $P_1$, it is not known in general if applying the same transformations in a different order to yield a program $P_2$, whether $P_1$ and $P_2$ will be the same. Semantically $P_1$ and $P_2$ will be identical, but it cannot be known which of them, if they are different, will be the more efficient. Worse, it cannot in general be known whether, having applied transformation $T_2$, say, to $P$, it will then be possible to apply transformation $T_1$. This problem is a very difficult theoretical problem which does not have a solution. If a solution existed, then it would be possible to have an automatic restructurer for turning any program to some "best" semantically equivalent program.

Following these restructuring transformations, code is generated. The code generated depends on the target machine. In addition, some automatic parallelizers also perform analysis on the costs of communication and synchronization and attempt thereby to determine a schedule for running the program, but such attempts are still at the research stage.

Examples of automatic parallelizers are Parafrase-2 [66], PFC+ [4], and SUPERB [87].

### 2.4 Extensions to Serial Languages

As mentioned in the introduction, developments in parallel software are often motivated by the need of making use of specific architectures and machines. Fortran, being the language of numerical algorithms, has been extended by various machine manufacturers for this purpose. In this section, two such
Fortran extensions will be presented briefly. The Fortran 200 for the CDC Cyber machine, and Sequent Fortran for Sequent machines. Not all the developments in software were made to take advantage of specific architectures of course, and two other language extensions intended to produce a general purpose parallel language will also be described briefly, Pascal Plus and the Sequent extensions to C.

2.4.1 Fortran 200

This is a superset of Fortran for the CDC Cyber 205 parallel computer. The extensions are added in order to facilitate specifying vector operations to take advantage of the computer's vector processors [16].

Let $VEC$ be a scalar array declared as

$$\text{DIMENSION } VEC(100)$$

then it is possible to use $VEC$, or parts of it, as a vector, by referring to it using

$$VEC(base; length)$$

where $base$ refers to the subscript corresponding to the vector's first element, and $length$ is the number of elements in the vector. For example $VEC(10; 3)$ is the vector whose elements are $VEC(10)$, $VEC(11)$, and $VEC(12)$. Vectors can also be defined on multidimensional scalar arrays.

A new data type, DESCRIPTOR, is also provided. A DESCRIPTOR is a pointer to a vector. It is declared in the usual way. For example

$$\text{DECLARATOR } VP$$

and can be initialized either at compile time using a DATA statement, or at run time with another extension to the language: an ASSIGN statement.

These vectors can be used in expressions. For example
\[ VEC(1;10) = VEC(11;10) + VEC(21;10) \]

is a vectorization of the loop

\[
\begin{align*}
\text{DO 10 } & \quad I = 1,10 \\
10 & \quad VEC(I) = VEC(I + 10) + VEC(I + 20)
\end{align*}
\]

Vectors can also be combined with scalars in expressions, for example

\[ VEC(1;10) = VEC(21;10) \ast X \]

is a vectorization of

\[
\begin{align*}
\text{DO 10 } & \quad I = 1,10 \\
10 & \quad VEC(I) = VEC(I + 20) \ast X
\end{align*}
\]

Vectors can also be used in logical expressions where the result is a vector of logical values. For example

\[ VEC(1;10) \ast \text{GT}. VEC(51;10) \]

has the value \( BV \), say, where \( BV \) is a vector of bits (1 or 0) such that \( BV(i) \), \( 1 \leq i \leq 10 \), is 0 if \( VEC(i) \leq VEC(i + 50) \), and 1 if \( VEC(i) \geq VEC(i + 50) \).

Vector logical expressions are used in another extension to the language corresponding to the \textsc{if} statement. This is the \textsc{where} statement. For example

WHERE \((VEC(1;10) \ast \text{GT}. VEC(51;10)) \)

\[ VEC(51;10) = VEC(1;10) \]

is semantically equivalent to

\[
\begin{align*}
\text{DO 10 } & \quad I = 1,10 \\
10 & \quad \text{IF } (VEC(I) \ast \text{GT}. VEC(I + 50)) \quad VEC(I + 50) = VEC(I)
\end{align*}
\]

A block WHERE statement is also provided to correspond to the block \textsc{if} statement. For example
WHERE \((V_{EC}(1; 10) .GT. X)\)
\[ V1(2; 10) = X \]
OTHERWISE
\[ V1(2; 10) = -X \]
END WHERE

is semantically equivalent to

\[
\text{DO 10 } I = 1, 10 \\
\text{IF } (V_{EC}(I) .GT. X) \\
\quad V1(I + 1) = X \\
\text{ELSE} \\
\quad V1(I + 1) = -X \\
\text{ENDIF} \\
10 \text{ CONTINUE}
\]

It is also possible to write vector functions, and some intrinsic functions for vectors are provided. For example, the function \(Q8V\text{GATHER}(V, I; U)\) operates on two vectors, \(V\) and \(I\), and produces its result in the vector \(U\). The function places in \(U\) the elements of \(V\) selected by \(I\). For example if \(V\) is the vector \(\{5, 6, 7, 8\}\), and \(I\) is the vector \(\{4, 1, 2, 3\}\), then a call to \(Q8V\text{GATHER}\) places the values \(\{8, 5, 6, 7\}\) in \(U\).

Finally, Fortran 200 allows the programmer to include object code in the program to allow maximum possible use to be made of the machine.

### 2.4.2 Sequent Fortran

Sequent Fortran [58] extends Fortran with directives for parallelizing DO loops, as well as providing a library of parallel routines that be called from within a program. These can also be called from programs written in C and Pascal (see section 2.4.4 below). There are just three directives in Sequent Fortran for parallelizing DO loops. They are

- C$ DOACROSS (options)
• C$ ORDER name of order section
• C$ ENDORDER name of order section

All the directives begin with C$ starting at the first column. In this way the program can be compiled by standard Fortran compilers which would treat the lines containing the directives as comments.

The options following the DOACROSS directive are determined by the variables in the loop. The programmer is required to analyze these variables and group them into five types: shared, shared ordered, shared locked, reduction, and local variables.

A shared variable is either a read-only variable or an array such that each of its elements is referenced by one loop iteration only. For example, in the loop

\[
\text{DO 10 } I = 1, 10 \\
10 \quad A(I) = B(I)
\]

both \( A \) and \( B \) are shared variables. To parallelize this loop in Sequent Fortran, the directive

C$ DOACROSS SHARE(A, B)

is added just before the loop which becomes

C$ DOACROSS SHARE(A, B) \\
\text{DO 10 } I = 1, 10 \\
10 \quad A(I) = B(I)

A local variable is one which is initialized before its value is used in each iteration. For example the loop

\[
\text{DO 10 } I = 1, 10 \\
\quad \text{DO 10 } J = 1, 10 \\
10 \quad A(I, J) = B(I, J)
\]
contains two shared variables, $A$ and $B$, and one local variable $J$. The outer loop can be parallelized thus

\[
\text{C$\S$ DOACROSS SHARE}(A, B), \text{LOCAL (} J \text{)}
\]

\[
\text{DO 10 } I = 1, 10
\]

\[
\text{DO 10 } J = 1, 10
\]

\[
10 \quad A(I, J) = B(I, J)
\]

A reduction variable is one which appears in the body of the loop in a statement of the form

\[
\text{var} = \text{var op exp}
\]

where \text{var} is the reduction variable, \text{op} is an associative commutative operation, and \text{exp} is any expression that does not include \text{var}. Further, there can be several statements of this form provided the operation \text{op} is always the same. In the following loop $R$ is a reduction variable.

\[
\text{DO 10 } I = 1, 10
\]

\[
10 \quad R = R \ast A(I)
\]

Marking this loop for parallel execution turns it into

\[
\text{C$\S$ DOACROSS SHARE}(A), \text{REDUCTION}(R)
\]

\[
\text{DO 10 } I = 1, 10
\]

\[
10 \quad R = R \ast A(I)
\]

A shared ordered variable is one that is not a shared, local, or a reduction variable, and if the loop iterations are performed in an order other than that implied by the serial execution order, the result would be wrong. In the loop

\[
\text{DO 10 } I = 1, 10
\]

\[
S(I) = A(I) \ast B(I)
\]

\[
10 \quad T = S(I + 1) - S(I)
\]
the variable $S$ is a shared ordered variable for the two reasons given above. To parallelize the loop in Sequent Fortran, an order section must be named for each ordered variable. This is marked by the ORDER, and ENDORDER directives. Thus the loop above becomes

```
C$ DOACROSS SHARE(A, B, S), LOCAL(T), ORDER(SO)
   DO 10 I = 1, 10
   C$ ORDER SO
       S(I) = A(I) * B(I)
       T = S(I + 1) - S(I)
   C$ ENDORDER SO
   10 CONTINUE
```

Note that the shared ordered variable is also given as an argument to the SHARE option.

A shared locked variable is a variable that is assigned from several iterations, and (unlike a shared ordered variable) if the loop iterations are performed in any order (but not concurrently) the result would still be correct. In the program segment

```
BIG = 0.0
DO 10 I = 1, 10
   A(I) = B(I) * C(I)
   IF (A(I) .GT. BIG) THEN
      BIG = A(I)
   ENDIF
10 CONTINUE
```

The variable $BIG$ is a shared locked variable. The treatment of shared locked variables in Sequent Fortran is slightly different from the other four types of variable. Whereas the others are treated using the directives only, shared locked variables require a pair of parallel library routines, m_lock and m_unlock (see section 2.4.4 below), but these calls are preceded by the C$ directive so that the resulting program would still be a correct standard
Fortran program. In order to parallelize the program segment above it is rewritten as

\[ \text{BIG} = 0.0 \]

C$ DOACROSS SHARE (A, B, C, BIG)
   DO 10 I = 1, 10
      A(I) = B(I) \times C(I)
   C$
      CALL m.lock
      IF (A(I) .GT. BIG) THEN
         BIG = A(I)
      ENDIF
   C$
      CALL m.unlock
   10 CONTINUE

As can be seen, the shared locked variable is given as an argument to the SHARE option of the DOACROSS loop directive.

Notice that in Sequent Fortran the programmer does not specify which parts of the program are to be executed in parallel. The compiler decides this according with the variable types, which are the responsibility of the programmer. Further, these directives apply only to loops, and in section 2.4.4 below the parallel library routines will be discussed as well as the few extensions to C made by Sequent for providing general parallel programming facilities.

Finally, before leaving Fortran, it should be mentioned that the Fortran 90 standard defines operations on vectors and arrays that are similar to the Fortran 200 just discussed. See [30].

2.4.3 Pascal Plus

Pascal Plus [83] is an extension of Pascal. The extensions are made for two purposes: to allow for modularization and concurrent processing. Modularization is achieved by the envelope structure, which defines data structures, the operations that can be performed on the data, and a control structure
which envelops the block within which it is declared. The envelope structure will not be described any further here. Concurrent processing is achieved by providing a means of defining tasks that can be performed in parallel via the process, and a way of controlling access to shared data by concurrent tasks via the monitor. There is in addition the condition monitor to enable processes to synchronize their action when using shared resources. A process in Pascal Plus is a routine of which there may be several instances at the same time. This is defined by the reserved word process as in the following example which defines a process called PP.

```pascal
process PP;
    (* local declarations and definitions *)
begin
    (* the statements making up the process *)
end;
```

To indicate how many instances of the process can execute concurrently, the instance declaration is used. The instances can be named, as in

```pascal
instance
    PP1, PP2: PP;
```

or they can be an array, as in

```pascal
instance
    PPS: array[1..N] of PP;
```

The activation point of the concurrent processes is the inner statement in the body of the block which contains their declaration. For example, in the procedure

```pascal
procedure PAR;
    process PP;
```
The statement * * * is the inner statement. The procedure PAR begins by executing the initialization statements (if any). At the inner statement all N instances of PP, PP[1] to PP[N], become active and all run concurrently. Upon the termination of all N instances, the finalization statements are executed.

Monitors in Pascal Plus control access to shared data by processes. This is done by placing all shared data in the monitor. A process that wants to access the shared data must call the monitor. The monitor allows only one process at a time to execute its code. (It can be seen then that monitors perform the rôle of lock. See chapter 3.)

The structure of a monitor is as follows:

```pascal
monitor MON;
(* local declarations *)
procedure *ACTION1(argument: argument_type);
   begin
      (* statements of ACTION1 *)
      end; (* procedure ACTION1 *)
(* other local procedures and functions *)
begin
   (* initialization *)
   * * *
   (* finalization *)
end; (* MON *)
```
The initialization statements are executed upon declaration of the monitor. The asterisk, ‘*’ which precedes ACTION1 in the example above means that this procedure can be called by statements outside the monitor. Similarly, in any data declaration inside the monitor, a * before the name of the variable indicates that it may be accessed from outside the monitor, in read-only mode. All other variables, procedures, and functions are local to the monitor and may not be accessed from outside. Instances of a monitor are created by the instance declaration as before.

In fact, all process definitions can appear only within a monitor definition, and the program block itself is considered a monitor definition.

In order to synchronize processes, a predefined monitor called CONDITION is available. It allows processes that want access to it to be queued, and their calls to the monitor are then executed on a first come first served basis. The processes are also allowed to have different priorities on the queue. The definition of CONDITION and its use in an example to implement a bounded buffer (see next chapter) is given in [62].

2.4.4 Parallel Programming in C on the Sequent Balance

There are only two additions to the C language as such, both these additions are type qualifiers. The first is the shared type qualifier used to make data shared, or global, to all processes. And the private type qualifier used to ensure that the data declared can only be accessed by one process. In order to facilitate parallel programming in C (as well as in Fortran and Pascal) a set of library routines is provided [73]. These routines can be called from within C programs. It would not be appropriate to list all these routines and comment on them here. The discussion below is of what these routines are intended to achieve.
Perhaps the most important of these routines is the `m_fork()`. This takes as its arguments the name of a function and the function's arguments. It causes several processes to be spawned each of which will start executing the function, and all the processes run concurrently. The number of processes spawned is determined by a call to another routine. The system checks that there are as many processors as there are processes so that each process is assigned to run on a processor. If there are not sufficient processors, `m_fork()` will spawn only one half the number of processors available. The process that calls `m_fork()` is known, following the terminology used in the Unix operating system [48], as the parent process and the processes created by the call as the children processes. All the processes, parent and children, are identical to each other except for their id numbers. The processes then execute the function and when they finish, all wait until the last one has returned to the calling process, the parent. At which point the parent continues by executing the code following the call to `m_fork()`. All the children processes will be spinning, i.e. idle. The programmer can kill them by issuing a call to the routine `m_kill_procs()`; or they can be parked so that they remain in the system to be used later, but they are actually removed from the processors which are now free to run other code. This is done by a call to another routine `m_park_procs()`. When the processes are needed later, they can be reactivated with a call to `m_rele_procs()`. The programmer can also leave the children processes spinning and use them with another call to `m_fork()`.

In the example below, the arrays `A[0 ... 3]`, `B[0 ... 3]`, and `C[0 ... 3]`, are shared. The main program forks four processes (including the parent) with ids 0, 1, 2 and 3. Each of these then performs a task according to the function `par()`:

```c
shared int A[4], B[4], C[4];
void par();
main()
```
{ 
  /* read in the values of A and B */
  m_set_procs(4); /* sets the no. of forked processes to 4 */
  m_fork(par);
  m_kill_procs(); /* print A */
  return;
}

void par() {
  int i;
  i = m_get_myid(); /* this returns the process's id */
  switch(i) {
    case 0: C[i] = A[i] + B[i]; break;
    case 1: C[i] = A[i] * B[i]; break;
    case 2: C[i] = A[i] - B[i]; break;
    case 3: C[i] = A[i]/B[i]; break;
  }
  return;
}

Access to shared data and code can be controlled using locks; and two types of lock are provided. The first is called a spinning lock. It is set, or locked, by a call to s_lock(), and unlocked by a call to s_unlock. If a lock \( \ell_1 \), say, is already set, or locked, when a process attempts to set it with a call to \( s\_lock \) then the process does a busy wait, or spins, until the lock is released. The other type of lock which is set by a call to \( s\_lock \) operates differently. If \( \ell_2 \), say, is locked when a process calls \( s\_lock \), the call immediately returns a value to the process, which can then decide what to do depending on the returned value. It can either keep trying to get the lock, or it can execute other code and attempt to get the lock later. If only one lock is required in a program, the routines \( m\_lock \) and \( m\_unlock \) can be used. An example
of their use, from a Fortran program, was given earlier in section 2.4.2.

Synchronization of parallel processes is dealt with using *barriers*. A barrier is a data structure that acts as a rendezvous for several processes. If a barrier is defined for $n$ processes, say, with the call `s_init_barrier()` then each of the $n$ processes has to include a call to `s_wait_barrier()`; at this point the processes wait until all $n$ of them have reached that point. When this happens all them proceed.

There are also routines for dynamically allocating shared memory which are similar to those in the standard C libraries. For example the routine `shmalloc()` allocates memory much the same way as `malloc()` does, albeit the allocated memory is global, or shared, memory.

### 2.5 Parallel Languages

In this section two languages for parallel programming will be described briefly. In contrast with the languages of the previous sections, these two languages are not extensions added to an existing language. Having said this, the second of these, *Strand*, is remarkably similar to Prolog; but, as will be seen, the semantics of *Strand* are different.

#### 2.5.1 Occam

Occam [44] is a language for parallel programming on a distributed memory machine, based on the Communicating Sequential Processes (CSP) model [21]. The language was developed by the Inmos Company mainly to be used on their particular parallel computer, the transputer. What follows is a brief description of the language; for more details see [43, 67].

Processes communicate via channels. A channel in occam is a zero-buffered, unidirectional path for data. There is no limit on the number
of channels a process can have, and each channel must be from one process to another (one) process. Since channels are unidirectional, two-way communication between two processes requires two channels. A channel is declared by the word chan, for example

chan $CC$:

declares a channel with the name $CC$. The statement

chan $CC[21]$:

declares an array of 21 channels $CC[0 \ldots 20]$.

A process can output a value on a channel using '! ' and input a value from a channel using '? '. For example if process 1 outputs $x$ on channel $CC$, and process 2 inputs the value from channel $CC$ in $y$, this would be achieved by

$CC! x$

in process 1, and

$CC? y$

in process 2. The transfer of data on a channel can only happen if both are active. In the example above, if process 2 is not active, then process 1 does not output the value of $x$ until process 2 does become active.

A program in occam is considered to be a hierarchy of processes, with each process determining the execution of its constituent processes. There are three constructors for determining the way the constituent processes are executed: the seq constructor for sequential execution, the par constructor for parallel execution, and the alt constructor for selective execution.

Indentation is used by occam to define the scope of processes, so that to specify that the statements of a processes are to be executed sequentially, we would have to indent all its statements as in
seq

\[
CC1 ? x \\
y := x \times x \\
CC2 ! y
\]

defines a process that is made of three sequential processes: one that reads the value on channel \(CC1\) and places it in the variable \(x\), the second assigns the variable \(y\) the value of \(x^2\), and the third places the value of \(y\) on channel \(CC2\). The code below shows two concurrent processes

par

seq

\[
CC1 ? xin \\
xout := xin + 1 \\
CC2 ! xout
\]

seq

\[
CC3 ? yin \\
yout := yin + 2 \\
CC4 ! yout
\]

each of which is made up of three sequential processes. The alt constructor is similar, in syntax, to seq and par, except that each process has a guard and the process whose guard is the first to be tested and found to be true is executed, and none of the other processes is. The guard can be a logical expression, or it can be an expression for reading a value from a channel, or both. If it reads a value from a channel then the first process to get a value on its channel is the one that is executed.

In addition to the three constructors, a replicator is provided which creates multiple instances of the same process. For example

chan CC[21]:
par i = [0 for 20]
while TRUE
    var X:
seq

\[
CC[i] ? X \\
CC[i+1] ! X
\]

declares an array of 21 channels, and 20 parallel processes each of which reads a value on a channel and places it on the input channel of another process.

### 2.5.2 Strand

Strand [27, 77] is derived from fiat concurrent prolog [28, 81], which is a concurrent programming language. Even so, it was developed as a general purpose programming language. As will be seen shortly, Strand is, semantically, a completely parallel language in the sense that there is no order imposed on the execution of the statements making up a program by explicit control statements. Processes execute when the data they require become available. In this respect at least, Strand is a data flow language. In practice, unless a data flow machine is available, the above remains a semantic description of the order of execution only.

A program in Strand consists of a set of guarded clauses. A guarded clause is of the form

\[
H : - G_1, \ldots, G_m | B_1, \ldots, B_n.
\]

where \(m, n \geq 0\)

where \(H\) is the clause head. The clause head consists of a name, \(N_H\), and a number of arguments, \(T_1, \ldots, T_p\), enclosed in brackets. Thus

\[
N_H(T_1, \ldots, T_p)
\]

where \(p \geq 0\)

The name and the number of arguments, \(p\), called the arity of the clause, determine the clause, which can then be referred to as \(N_H/p\). The clause whose head is \(op1(T_1, T_2)\) will be referred to as \(op1/2\). This is different from another clause with the same name and a different arity, \(op1(T_1, T_2, T_3)\), say, which will be referred to as \(op1/3\).
Each $G_i$, $1 \leq i \leq m$, is called a *guard*. A guard is a logical expression which is either true or false. In Strand, a guard can only contain predefined operations, called *kernels*, and no user defined operations unless these are written in a "foreign" language and linked with the Strand program using Strand's foreign language interface (see below). A guarded clause can have zero guards.

The symbol ‘[‘ is called the *commit* operator, and this will be explained shortly.

The set of $B_i$, $1 \leq i \leq n$, with the $B_i$'s separated by a comma and the set terminated by a full stop, is called the *body*, and each $B_i$ is referred to as a *body call*. A $B_i$ is either the head of a guarded clause (which must be defined elsewhere), or a predefined operation. It is possible for a guarded clause to have zero body calls.

An item of data in Strand is called a *term*. A term is one of the following: a number, a variable, a string, a list, or a tuple. A number is just a number, real or integer, and is denoted by its value. A variable is an unknown quantity which is not yet assigned. Until a variable is assigned it can stand for any of the other types of term, a number, a string, a list, or a tuple. However, once a variable is assigned a value, it may not be assigned again; thus Strand has the single assignment rule. A string is a sequence of symbols that stands for itself. In Strand variables are distinguished from strings by the following: a variable always starts with an upper case letter unless it is quoted. For example $A$, $B$, and $C$ are variables, but "$A$", "$B$", and $c$ are strings. A tuple is a collection of items enclosed by braces, "{" and "}", and separated from each other by commas. The number of items in a tuple is called the tuple's arity. An $n$-ary tree is used to represent a tuple of arity $n$. With a tuple, one can address the items within directly. A list is, of course, a very well known and widely described data structure [49, 1]. A list in Strand is made up of two parts, the *head* and the *tail*. These are enclosed in square brackets, '["}']
and', and the head is separated from the tail by a bar, '|'. Thus a list is

<math>
\text{[Head|Tail]}
</math>

If there is only one item in the list, then the tail is just the empty list, [I], so that a list of just one item is [Head][I]. A list of several items will have as its head the first item and the tail is itself a list of the other items in it. So, for example, a list with the items I_1, I_2, I_3 in it is

<math>
[I_1|[I_2|[I_3]]]
</math>

and this will be written as [I_1, I_2, I_3] for convenience.

Note that whereas a tuple has a fixed number of items, there is no such restriction on a list. A list is represented by a binary tree, where each node points to two children, one of which is the head, and the other is itself a tree representing the tail. A list must always terminate with the empty list.

A procedure in Strand is one or more guarded clauses with the same clause head. A procedure is executed by what is called reduction. At any one point there is a collection of processes (semantically all the processes are active concurrently). A process is chosen that matches the clause head, where matching proceeds according with the same rules as those for Prolog [14, 56] and these will not be explained here. Of a procedure's possible many guarded clauses, one is chosen and its guards are evaluated. If all of them are true, the commit operator then means that that process is reduced. If the body of the clause is a predefined kernel operation, it is immediately executed and the process terminates. Otherwise a process is added for each body call to the pool of processes — this is called forking — and then execution continues with a process chosen from the pool. If the body of a clause is made up of just one body call, then the addition of a process corresponding to that body call to the processes pool is called changing state rather than forking. If the body is empty then the process again terminates. It can thus be seen
that a process, or processes, corresponding to a body of a guarded clause, is matched against procedures, from which one clause is chosen. This results in one of three types of reduction: termination, forking, or changing of state.

It is not always possible to reduce a process, either because it cannot be matched to any procedure, or, having been matched to some procedure, it cannot be matched to a specific guarded clause, or, having been matched to a specific guarded clause, the guards do not all evaluate to true. In any of these cases the process is suspended; it is thrown back in the pool of processes to be picked up later. This happens most commonly when a process requires data not yet available. In this way, data flow determines the order of execution of the program statements.

The following example of a program made up of two procedures illustrates the execution of a program in Strand. The example is taken from [27]. The program adds up the numbers in a list L, and puts the sum in the variable Sum.

\[
\text{sum}([X \mid Xs], A, \text{Sum}) \leftarrow \text{sum}([X \mid Xs], A, \text{Sum}) : \leftarrow A1 \text{ is } A + X, \text{sum}(Xs, A1, \text{Sum}). \\
\text{sum}([\), A, \text{Sum}) \leftarrow \text{Sum} \leftarrow A.
\]

The program consists of two procedures, \text{sum}/2 and \text{sum}/3. Note that even though the two procedures have the same name, they are different because of their different arities. \text{sum}/2 is made up of one guarded clause (the number of guards is zero). \text{sum}/3 is made up of two guarded clauses, and again the number of guards for both is zero. Let the program be started, from the environment, with the two processes

(1) \text{sum}([1] \mid \{T\}, S), and (2) \text{T} := [2, 3].

The execution of the program may proceed as shown in table 2.1.

The second column in the table gives the number of the process which is picked from the pool of processes. As far as the language is concerned,
<table>
<thead>
<tr>
<th>Step</th>
<th>Process Picked</th>
<th>Reduce/Suspend</th>
<th>Processes in Pool</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>(1) sum([1], T), S&lt;br&gt;(2) T := [2, 3]</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>(1)</td>
<td>change state</td>
<td>(1) sum([1], 0, S)&lt;br&gt;(2) T := [2, 3]</td>
</tr>
</tbody>
</table>
| 2    | (1)           | fork           | (1) A1 is 0 + 1  
                                      | (2) T := [2, 3]  
                                      | (3) sum(T, A1, S) |
| 3    | (3)           | suspend        | as in step 2     |
| 4    | (2)           | terminate      | (1) A1 is 0 + 1  
                                      | (2) sum([2, 3], A1, S) |
| 5    | (2)           | fork           | (1) sum([3], A2, S)  
                                      | (2) A1 is 0 + 1  
                                      | (3) A2 is A1 + 2 |
| 6    | (2)           | terminate      | (1) sum([3], A2, S)  
                                      | (2) A2 is 1 + 2 |
| 7    | (1)           | fork           | (1) A2 is A2 + 3  
                                      | (2) sum([1], A3, S)  
                                      | (3) A2 is 1 + 2 |
| 8    | (1)           | suspend        | as in step 7     |
| 9    | (3)           | terminate      | (1) A3 is 3 + 3  
                                      | (2) sum([], A3, S) |
| 10   | (1)           | terminate      | (1) sum([], 6, S) |
| 11   | (1)           | terminate      | R := 6           |
| 12   | (1)           | terminate      | <empty>          |

Table 2.1: Execution of the sum program. (From [27]).
there is no restriction at all on which process is actually picked; if each
process existed in a separate processor, then all of the processes are picked
simultaneously or concurrently and would get to step 4 from step 0 in one
step.

A suspension takes place in step 3 because \( \text{sum}(T, A1, S) \) cannot be
matched with either of the clauses of the procedure \( \text{sum}/3 \). To match it with
the first clause, the variable \( T \) would have to be matched against \([X \mid Xs]\) but
it is not known at this stage what \( T \) is. To match it with the second clause, \( T \)
would have to be matched against the empty list \([\ ]\), and again this cannot be
done because \( T \)'s value is not known. The process is therefore suspended and
returned to the pool. One way of describing what happens is that process (3)
in step 3 cannot execute because it requires data that is not yet available. In
step 8, process (1) is suspended because the value of \( A2 \) is not available.

At step 5 the process that was suspended and returned to the pool at
step 3 is picked up again. This time it does reduce because \( T \) has been as­
signed a value in step 4. It is seen then how the availability of data determines
the order of execution.

In the procedures of the program there are no variables \( A2 \) and \( A3 \) as
shown in table 2.1. Because of the single assignment rule of the language,
unless variable names are matched by the procedures themselves, a new vari­
able is created at each reduction. On the other hand, if a variable is used
in more than one procedure, then its assignment by one of them enables the
other to use its value. In this way shared variables can be used to pass data
between processes. In the next chapter this will be seen in the section on the
the basic techniques for writing programs in Strand.

From the above it can be seen that Strand, being a data flow language,
can bring to the fore any inherent parallelism in a program. That is the
situation as far as its semantics are concerned. In practice, a Strand pro-
gram does not, when running on a multiprocessor, in any way attempt to
distribute the processes on the available processors. It is the responsibility of the programmer to partition the program and allocate its parts on individual processors. The language does provide virtual mappings of the available processors, such as a straight line, an infinite grid, a torus, so the programmer can use them to write code that would run on different machine configurations.

Although communication between processes is specified by the language itself, in practice the programmer would have to specify communication streams explicitly if any meaningful gain is to be had.

Finally, Strand is a general purpose parallel programming language, but it lacks a variety of data structures, such as arrays, which are very important in general purpose programming. Further, the single assignment rule would make it impractical to use the language for applications where variables need to be updated many times. And so a "foreign" language interface is provided. The interface allows functions and subroutines written in Fortran and C to be incorporated into a Strand program. The interface is provided with another idea in mind; because Strand can perform synchronization and communication effortlessly, it is envisaged that a programmer can take an existing program in Fortran, say, and partition it into modules. A Strand program would then be written to control the running of these modules.

Strand was in fact used in the work presented here; and some results and comments will be found in chapter 4.

2.6 Tools for Parallel Software

There are several tools that aid in the design and writing of parallel software. These tools vary widely in several aspects. [53] list some 15 of these tools and classify them according to the parallel computing paradigms they assume. Most of them assume a multiprocessor paradigm, shared or dis-
tributed memory, though some of them assume vector processing or systolic arrays. Next the platform upon which these tools operate is given. This is either the physical machine, e.g. Sun or Macintosh, or the operating system, such as Unix or X-Windows, or the kind of input to the tool, whether it is graphics or text. Following this the target of the tool is given. Again this varies from being a language, to a specific machine, or a type of machine, or even code that becomes the input to another tool. Finally the life cycle of a parallel program is broken down into phases, and the tools are distinguished by the phase or phases of a program's life cycle they apply to, such as code generation, debugging, restructuring of serial code, optimization of code for performance, and so on. For more information on these tools consult the reference above and the references therein.

It is worth pointing out that all the tools which have as their aim the production of parallel code, are collections of routines (a tool box) which are to be made use of from within the program. As far as this goes it can also be argued that the Sequent parallel programming library described earlier is a tool box.

Only one tool will be mentioned here, and it is called Schedule [22]. It is a tool box of Fortran routines that enable users to specify a parallel program. The target machine is a vector processor, the Alliant FX-8. It assumes a shared memory paradigm and its platform is a Sun workstation with the input in graphics (which requires another tool called Build) and text. The idea behind Schedule is that it can provide a uniform interface to various machines for the user. Also, being made up of a set of routines, an existing program can be parallelized by appropriate calls to these routines. A program is broken down into a set of tasks whose execution order is determined by the programmer and this order is expressed using certain Schedule routines. The routines also provide for declaring shared variables, synchronization of processes, forking new processes, controlling access to shared variables, and
so on. A very good background to Schedule is provided in [54].

2.7 Algorithms and Control Flow

All programs are implementations of algorithms along with data structures. An algorithm as well as a program contain statements whose execution means action is performed on some data, and statements whose execution changes the order of execution or the control flow. This control flow is an inseparable part of the algorithm.

When programs become parallel programs, there might be a need to modify the data structures, an array might become a vector, say. But the control flow, as far as the algorithm is concerned does not change. Other control statements for forking, communication, and synchronization, may be added. But these cannot and must not interfere with the underlying algorithm.

Of course, it is possible that when two algorithms are available to solve a problem, one of them might be more suitable for parallel implementation than the other, whereas the other is more suitable for serial execution. There is, however, a very important point to make. This will be put in two parts:

- It is possible to have a serial program that cannot be parallelized.
- It is impossible to have a parallel program that cannot be serialized.

In other words: there is nothing that a parallel program can do that cannot be done by a serial program. With this in mind, it should be possible to have parallel control constructs that are completely separate from the rest of the program. This would be different from the approaches to parallel software presented in this chapter, where it is assumed, or presumed, that parallel control is part of the program itself. This idea of separate control is one of the ideas from which the work presented here grew. It will be taken up
again briefly in the next chapter and then considered in detail in chapter 4 and subsequent chapters.
Chapter 3

Basic Concepts in Parallel Processing

3.1 Data and Function Partitioning

These are two basic ways of turning an algorithm into a parallel program, and they correspond to the classes SIMD and MIMD of Flynn (see chapter 1). Data partitioning means the creation of identical processes which run, or can run, concurrently with each operating on different data from those operated on by the others. In other words, the data to be processed is distributed, or partitioned, among several processes. Examples of data partitioning have been given: all the loops considered in the previous chapter, for example, were being considered for data partitioning. A full example will be given in the next chapter, and there will be many examples in the rest of the thesis, and these can be consulted.

Function partitioning means the creation of different processes, so that it is the functions that the program has to perform that are distributed, or partitioned. Pipelining is an example of function partitioning. Other examples of function partitioning are hard to come by, but in chapter 8 a schedul-
ing strategy that requires function partitioning will be given, discussed and implemented in detail.

Data partitioning is easier to implement than function partitioning simply because data is easier to manipulate than functions. One way in which the two can be distinguished further is to say that function partitioning means that the control flow of the program is altered, but this does not happen with data partitioning. It was observed at the end of chapter 2 that parallel control flow does not form part of an algorithm as such. It will be seen in chapter 8 how, once parallel control is separated from the rest of the program, function partitioning can be implemented using the same ideas and methods used for data partitioning.

3.2 Speedup and Efficiency

Speedup and efficiency are measures of performance in parallel computing. There is only one purpose for parallel computing, and that is speed. Speedup is a measure of how much faster a program runs in parallel than serially. Efficiency is a measure of how well the parallel computer is being utilized.

Speedup is the ratio of the time taken to run a program serially to that taken to run the same program (with additions for parallel control) in parallel. Let \( T_1 \) be the time taken to run the program using one processor (serially), and let \( T_p \) be the time taken when \( p \) processors are used. Then the speedup, \( S_p \), with \( p \) processors is given by

\[
S_p = \frac{T_1}{T_p}
\]  

(3.1)

Now, in any program, some parts of it may be parallelized (say a loop with independent iterations) and others may not be. Let the time taken for the execution of the serial part be \( t_s \) and that for the execution of the parallelizable
part be $t_p$. Then

$$T_1 = t_s + t_p$$  \hspace{1cm} (3.2)

Consider the parallelizable part. If it is possible to distribute the work on $p$ processors so that each processor did exactly $1/p$'th of the work, then the time taken for this part becomes $t_p/p$. If such a perfect partition existed, we talk of **perfect load balancing**. Even with perfect load balancing for the parallelizable part of the program, as long as part of it must be executed serially, perfect load balancing for the whole program cannot be achieved.

The efficiency, $E_p$, with $p$ processors is just the speedup divided by $p$,

$$E_p = S_p/p = T_1/pT_p$$  \hspace{1cm} (3.3)

It can be seen that the minimum possible value for $T_p$ is when a perfect load balance is achieved for the parallelizable part, so that

$$T_p|_{min} = t_s + t_p/p$$  \hspace{1cm} (3.4)

so that

$$S_p|_{max} = T_1/(T_p|_{min})$$

$$= \frac{1}{t_s/T_1 + (T_1 - t_s)/T_1p}$$

$$= \frac{1}{R - (1 - R)/p}$$  \hspace{1cm} (3.5)

where $R = t_s/T_1$, is the fraction of the program that must be performed serially. Clearly, this sets an upper limit on the speedup achieved with any number of processors, given by

$$\lim_{p \to \infty} S_p = 1/R$$  \hspace{1cm} (3.6)

so that, for example, with $R = 0.1$, that is when one tenth of a program is unparallelizable, the speedup can never exceed 10 no matter how many processors are used. Another way of expressing equation (3.6) is to say that the speedup of a program reaches $S_\infty$ asymptotically.

63
[40] define a work segment as the basic unit of a program. When a program is partitioned for parallel processing, some of these segments will be run concurrently, and in order to ensure the correct order of execution, they have to be synchronized. One can then think of the parallel program as a collection of work segments with synchronization points. The synchronization takes up execution time that is not needed in serial execution. So synchronization must also be taken into account when a program is partitioned for parallel execution, and the size of the work segments is therefore very important. It should not be so small that any gain obtained is wiped out by the synchronization cost. The size of the work segment defines the granularity of the parallelism. If it is small then the parallelism is said to be small- or fine-grain; and if it is large, then the parallelism is said to be large- or coarse-grain. However, just what size of work segment constitutes fine or coarse grain parallelism cannot be defined in general, because it depends on the actual machine being used. For example the loop

\[
\text{DO 10 } I = 1,64 \\
10 \quad A(I) = B(I) + C(I)
\]

can be parallelized since all its iterations are independent of each other. On a vector machine, the whole loop can be vectorized, so that the work segment here becomes just one statement of the form \( x = y + z \). On a shared memory multiprocessor, such data partitioning would be fine-grain indeed, and no speedup can be expected. In such a case one would increase the size of the work segment by assigning, say, two processors for the loop with each one carrying out half the iterations. See strip mining in chapter 2.

Finally there is the communication cost to take into account when partitioning a program for parallel execution. It is not always possible to have work segments that are completely independent of each other, and some results worked out in some segment might be needed by another segment.
One would then have to consider whether it is worth running both segments concurrently and, using suitable synchronization, make one of them wait for the results from the other, or to run the segments sequentially, or even to partition the program differently. Communication cost is also a factor even when all the segments are independent of each other. With vector processors the processing time is less than that for fetching data from memory, but this problem is not unique to parallel processing since it arises in serial processing on one processor as well.

Before leaving this section on speedup of programs with parallel execution, it is worth mentioning two related matters. The first is that the limit on the speedup of a program imposed by the proportion of nonparallelizable code in the program is known as Amdahl's law, after Amdahl [6]. [70] and [63] counter this by stating that there are many programs where the proportion of nonparallelizable code is very small indeed, with the latter illustrating this with several examples.

The second point is that taking into consideration that any program must have some nonparallelizable code, and the costs of communication and synchronization, the best speedup would appear to be what is called linear speedup, that is when the speedup equals the number of processors used, so that a plot of speedup vs. the number of processors would yield a straight line with a 45° slope upwards. It is in fact possible to have better speedup, albeit in special cases. For example a program that searches an unordered array of $1 \times 10^6$ elements, say, for a specific item by looking up each element in turn and which takes $t$ seconds to find the item whose location is 500,001st in the array, can be parallelized and made to run on two processors with each one searching one half of the array, 1 to 500,000 and 500,001 to 1,000,000. The parallel program would now take only about $t/500,000$ seconds to find the item. This represents a speedup of $5 \times 10^5$ with only two processors!
3.3 Communication and Synchronization

Communication means the transfer of data between processes, and synchronization is the operation by which processes are made to run in the correct order. For example, let there be two processes, A and B. Process A works out a value and assigns it to a variable, \( x \), say, and process B needs the value of \( x \), as worked out by A, for its operations. Then process B must not run until process A has finished in order to ensure that it uses the correct value of \( x \). Or, if B does run before A has finished, then it must wait until \( x \) is assigned before it uses its value. This is synchronization. Getting the value of \( x \) from A and making it available to B is communication.

On a shared memory machine, such as the Sequent Balance, communication is most readily carried out using the shared memory. The variable \( x \) in the example above can be declared with the qualifier shared and then its value, throughout the program, will be known by all processes. On a distributed memory machine message passing is used for communication. This is also possible on a shared memory machine using operating system calls. The Unix operating system provides four such calls: `msgsend()`, `msgrcv()`, `msgget()`, and `msgctl()`, for receiving, sending and controlling messages between processes. The user can associate a type with each message, and the system automatically queues the messages so that they are received in the correct order. Also, requests for messages are queued and the system ensures that only one processor receives any one message, which is then removed from the queue. The messages themselves are always text messages, so that sending other types of values, say integers, requires the user to convert all data to be communicated to text, and then to be converted, when received, to the appropriate type. Using messages for communication is clearly not as easy as using shared memory.

If two or more processes read and write the same variable, the processes
need to be synchronized to ensure the variable is only being written by one process at a time, and also that no process reads the variable while it is being written by another. An example of this situation was given in section 2.4.2 earlier, when shared locked variables were considered. Synchronization can therefore be carried out using locks. This example is a specific case of synchronization by mutual exclusion, where it is required that only one process at a time performs certain operations, and all other processes are excluded. This requirement arises not only in parallel processing, but also in operating systems. The code which can only be executed by one process at a time is called the critical section or the critical region. Binary semaphores were proposed by [20] to implement mutual exclusion. They are analogous to railway semaphores (or flags) used to stop a collision taking place when two trains run on the same track. A binary semaphore (also called boolean semaphore), or lock, is a variable which, as its name implies, can take only one of two values, and only two operations are permitted on it: if its value is 1, then it can be set to 0; and if its value is 0, then it can be set to 1. The idea is that before a process enters its critical region, it checks the value of the semaphore. If it is 1, then it sets it to 0 and enters the critical region. When it leaves the region it sets the semaphore to 1. If a processor finds that the semaphore is 0, it does not enter the critical region. In order to avoid the situation where two or more processes check the semaphore at exactly the same time and both or all then enter the critical region, the checking and setting of the semaphore to 0 form one indivisible operation, called a test-and-set operation, which is usually implemented by the hardware.

What happens when a process tests a lock and finds it 0 depends on the kind of lock that it is. The process can be blocked and only awakened when the lock is released, or the process can keep spinning trying to get the lock, or a value is returned to the process to indicate that the lock is unavailable and let the programmer decide what to do in such a situation.
Another way of handling mutual exclusion which has been proposed is the use of monitors [59] (see the section on Pascal Plus in chapter 2), which is a structure of shared data and the operations to be performed on the data. When a process wants to perform one of these operations its makes a request to the monitor which is then responsible for ensuring mutual exclusion.

If a process has acquired a lock, and several processes try to obtain the lock also, then a priority can be imposed on the order in which the processes acquire the lock by queuing the requests for the lock. The queue is managed using a counting semaphore. A counting semaphore is a shared integer variable, access to which is protected by a lock. The counting semaphore takes values \( n \), where \( n \leq 1 \). If \( n = 1 \) it means that a process may enter the critical region. On exit it increments \( n \). If \( n \leq 0 \) then a process may not enter the critical region; it decrements \( n \), and puts its id number in the queue. When a process leaves the critical region, it checks the value of \( n \), and if this is less than 0, it notifies the process at the head of the queue, then it increments \( n \).

Another synchronization method is the barrier (section 2.4.4) which is a rendezvous point. Only when the set number of processes arrive at the barrier, are the processes allowed to continue execution.

A very general way of synchronization is the use of signals, or software interrupts. On the Unix operating system, processes can send signals to each other. The programmer can define what a process should do upon receiving these signals. In the implementation of exdata described in chapter 5, signals are used in the following way. A process which is not needed for a period suspends itself, and when work becomes available, or at the end of the program, the suspended processes are woken up with appropriate signals. Signals are much more versatile than barriers because their use does not depend on all processes reaching a certain point; further the programmer can decide, by associating operations with the different signals, what operations are carried out upon receipt of the signals. Care must be taken however that not all
processes suspend themselves without leaving any active process to send the wake-up signal.

3.4 Scheduling and Parallel Programs

According to [65] processor allocation means the assignment of tasks to processors but not actually binding the tasks to specific physical processors, and scheduling is processor allocation under time constraints and binding the tasks to specific physical processors. A schedule is either static or dynamic. A static schedule is one that is determined before the running of the program, and does not change throughout the running period. A dynamic schedule is one that is determined during the running of the program, and can therefore vary depending on the data and the number of processors used.

[70] defines a schedule as the allocation of tasks to processors, and a schedule is either deterministic or nondeterministic. The aim of a schedule is to minimize the overall execution time of a program, and then the schedule is said to be optimum. In a deterministic schedule, all the dependency relations and all the execution times of the tasks are known in advance. The result is a static schedule. In a nondeterministic schedule the execution times of the tasks are not known in advance, and random numbers are used to represent them in order to find a schedule.

In general, the problem of determining an optimum schedule is an \( NP \)-complete problem [32]. In chapter 8 a simple heuristic is used to schedule tasks of different sizes.

The binding of tasks to physical processors does not form part of the work presented here. But the word scheduling will be used, nonetheless, although its meaning will be slightly different.

Observe first that on a shared memory multiprocessor machine, such as the Sequent Balance, all the processors are identical. The job of allocating
processes to physical processors is carried out by the system. It would therefore be more appropriate to talk about allocating tasks to processes (rather than processors). In a way, this is what partitioning a program for parallel execution actually means: dividing the program, using function and data partitioning, and assigning the resulting tasks to different processes. One advantage of this is that a parallel program need not change with the available number of processors.

Secondly, recall from section 2.7 that parallel control flow is in essence separate from the rest of the parallel program which is an encoding of an algorithm along with data structures. Once a program is partitioned the statements that must be added to it to control its execution in parallel can therefore be viewed as separate control flow statements, whose purpose is to ensure the correct order of execution, in accordance with the dependency requirements, and to attempt, where possible, to minimize the execution time. It is these statements, which deal exclusively with the parallel control flow, taken together, that form the schedule as it is to be understood in this thesis.

Since we are concerned only with the scheduling of the tasks as the way in which they are assigned to processes, the terms static and dynamic scheduling will be used to mean the following: a static schedule is one that assigns the available tasks one by one, regardless of how many processes are active; a dynamic schedule is one that assigns the available tasks in blocks whose size depends on the number of available processes. Where the tasks are given to specific processes, this will be called deterministic scheduling, and where the processes are not distinguished in the assignment of tasks, this will be called nondeterministic scheduling.

Subsequent chapters will show how this separation can be done in practice and, it is believed, to advantage. For now, it is useful to emphasize this by defining a parallel program as follows:
A parallel program is a partitioned (serial) program with a schedule which specifies its parallel order of execution, or its parallel control flow.

### 3.5 Six Parallel Programming Techniques

This section presents six parallel programming techniques for Strand. As will be readily seen from these techniques, programming in Strand is very different from programming in a command language. There are no explicit control flow statements, and all iteration is done via recursion. Note also the use of shared variables to communicate data between processes. Finally, note that the only data structures used are the list, which is used quite extensively, and, in the second technique only, the tuple. The six techniques are:

1. Producer/consumer
2. Incomplete messages
3. Bounded buffers
4. Difference lists
5. Short circuits
6. Blackboards

The presentation here follows that in [27].

The first three of the techniques are ways of specifying communication between processes. The fourth, difference lists, is a way of enabling several processes to build a single list. The short circuit technique allows the detection of termination of processes and their synchronization, and the blackboard technique enables cooperating processes to write to and read from a common data structure.
Producer/consumer  Given a loop

\[
\text{for } I = 1 \text{ to } N \{
\begin{align*}
  s_1: & \quad A(I) = B(I) \ast C(I) \\
  s_2: & \quad D(I) = A(I) \ast A(I)
\end{align*}
\}
\]

It is possible to regard \( s_1 \) as a producer of the values \( A(1), A(2), \ldots, A(N) \), and \( s_2 \) as the consumer of these values. With the discussion of dependencies of the last chapter, \( s_1 \delta s_2 \), so that \( s_2 \) must wait for \( s_1 \) to finish before it can execute. However, it is possible to let \( s_1 \) execute for all values of \( I \), and then \( s_2 \) can execute immediately after \( s_1 \)'s first iteration. Then two processes are set up, the first, the producer, is just the loop

\[
\text{for } I = 1 \text{ to } N \{
\begin{align*}
  s_1: & \quad A(I) = B(I) \ast C(I)
\end{align*}
\}
\]

and the second, the consumer, is the loop

\[
\text{for } I = 1 \text{ to } N \{
\begin{align*}
  s_2: & \quad D(I) = A(I) \ast A(I)
\end{align*}
\}
\]

In order to make sure that the order of execution is correct, the values of \( A(I) \) are put in a list \( L \), and then the consumer acts on the elements of the lists that have been assigned. In addition, the arrays \( B, C, \) and \( D \) will also be put in lists. The two loops above now become the two Strand procedures below:

\[
\text{producer}(l, N, [B \mid Bs], [C \mid Cs], L) :-
\begin{align*}
  l = & < N \\
  & \text{Temp is } B \ast C, \\
  & L := [\text{Temp} \mid L1].
\end{align*}
\]

72
ll is l + 1,
producer(ll, N, Bs, Cs, L1).
producer(N, N, _, _, L) :-
    L := [].

consumer([A | As], D) :-
    Temp is A * A,
    D := [Temp | D1].
consumer([], D) :-
    D := [].

The consumer thus only uses values of A(I) that have already been produced.
The only other thing to notice in this example is the use of an underscore, '_', in place of arguments. This means that the value of the argument is not important, and will therefore be matched by any value or type.

Incomplete messages With the producer/consumer technique, the consumer is able to know when the producer has produced the values it needs, but the producer does not know anything about the activities of the consumer. The producer/consumer technique is thus a way of implementing one way communication. The incomplete messages technique allows two-way communication.

Consider a router program that allocates printing jobs to two printers. It receives requests in a list, and sends each request to one of the printers, whichever one of them is not busy. The program acts as a producer for the two printers, but it needs to know when the printers are ready before it sends them more printing jobs. This is solved by the router sending a tuple made up of two elements, the printing job itself, and an unassigned variable. The printer that receives the job, does the job and then assigns a suitable value, say "ready", to the variable that accompanied the job. When the router wants to send more printing jobs, it checks first that the last variable it sent out has been assigned "ready", and only if it has been assigned, does it send the new printing job. This technique is called incomplete messages because
the message sent out by the producer has an empty slot in it which can only be filled by the consumer.

Thus the router program can be represented by a process with three lists: one for receiving printing requests, and one for each of the two printers. In addition there must also be two variables, one for each printer.

```
router(Requests, Printer1, Printer1) :-
    router(Requests, ready, Printer1, ready, Printer2).

router([R|Rs], ready, P1, V2, P2) :-
    P1 := [{R, V1}, P11],
    router(Rs, V1, P11, V2, P2).

router([R|Rs], V1, P1, ready, P2) :-
    P2 := [{R, V2}, P21],
    router(Rs, V1, P11, V2, P21).

router([], _, P1, _, P2) :-
    P1 := [],
    P2 := [].
```

**Bounded buffers** In the producer/consumer paradigm, the producer keeps producing values without regard to whether the consumer has consumed them or not. It sometimes happens that there is a limit on the capacity for storing produced, but as yet unconsumed, values. For example, if the producer is producing printing jobs to a printer, these jobs are held in a buffer before they are processed because the printer cannot normally print as fast as the printing jobs get produced. The size of the buffer clearly cannot be infinite, and so the bounded buffer technique is used to indicate to the producer when a buffer becomes full so that it stops producing, and also when space in the buffer becomes available so that it can resume producing.

In order to implement the technique, a list, Buffer, is used. This is initialized by as many items as there are spaces in the buffer. The producer has access to the head of Buffer, and it only produces if this is not empty; if it is not empty, it produces an item and then removes the head. The consumer
also has access to the list Buffer; as soon as it consumes an item, it adds an item to Buffer. When the consumer has done all its work, it terminates Buffer with the empty list.

Initially, then, Buffer is the list [E1, E2, ..., EN], and the producer and consumer are given by:

\[
\text{producer}([\text{Head} \mid \text{Tail}], \ldots) :- \\
\quad \text{Head} := \text{a new value}, \\
\quad \text{producer}(\text{Tail}, \ldots).
\]

\[
\text{consumer}(N, [\text{Head} \mid \text{Tail}], \text{Buffer}) :- \\
\quad N > 0 | \\
\quad N1 \text{ is } N - 1, \\
\quad \text{consume Head}, \\
\quad \text{Buffer} := [\text{Var} \mid \text{Vars}], \\
\quad \text{consumer}(N1, \text{T}, \text{Vars}).
\]

\[
\text{consumer}(0, \ldots, \text{Buffer}) :- \\
\quad \text{Buffer} := [].
\]

All the three techniques above are techniques of managing stream communication developed for operating systems [46]. The three techniques below are from logic programming.

**Difference lists** This technique, due to [15], enables several processes to build a single list. The idea is that the list to be built, L, is split into a beginning, Lb, and an end, Lr. A process that is defined as two processes can make the two cooperate in building L by splitting the latter further with a middle, Lm, and then handing one process Lb and Lm, and handing the other process Lm and Lr. Each of the two processes thinks it has the beginning and end of a list it has to build. This procedure can therefore continue indefinitely. When a process finishes, it assigns the beginning of its list to the end.

For example, it is required to produce a list L that is the intersection of two lists, L1 and L2. This can be done by making Lb the list L itself,
which is not yet assigned, and Le the empty list []). A process can then remove the elements of L1 recursively, and pass each element to another process for comparing it with the elements of L2. This second process then recursively compares each element of L2 with the element it received from the first process. If the two are the same, it is added to the list. This goes on until L1 and L2 are exhausted. Thus:

\[
\text{intersect}(L1, L2, L) :\]
\[
\text{intersect}(L1, L2, L, []). \\
\text{intersect}([X | Xs], L2, Lb, Le) :- \\
\text{compare}(X, L2, Lb, Lm), \\
\text{intersect}(Xs, L2, Lm, Le). \\
\text{intersect}([], Lb, Le) :- \\
Lb := Le.
\]

\[
\text{compare}(X, [X | .], Lb, Le) :- \\
Lb := [X | L]. \\
\text{compare}(X, [X1 | Xs], Lb, Le) :- \\
X =\neq X1 | \\
\text{compare}(X, Xs, Lb, Le). \\
\text{compare}(., [], Lb, Le) :- \\
Lb := Le.
\]

**Short circuits** The short circuit technique [80] is used to detect the termination of a group of processes. When the processes are spawned they are chained together by a list. Each process is handed two links, such that the first link of the one is the second of the other. Thus

\[
\text{op}(\text{FirstLink}, \text{LastLink}) :- \\
\text{opA}(\text{FirstLink}, \text{MidA}), \\
\text{opB}(\text{MidA}, \text{MidB}), \\
... \\
\text{opZ}(\text{MidY}, \text{LastLink}).
\]

The processes then perform their actions, possibly spawning more processes in the same way. When a process finishes, it *short circuits* the chain.
by assigning its last link to its first (cf. difference lists above).

\[
\text{opC(FirstLink, LastLink) :-}
\]
\[
\text{LastLink := FirstLink.}
\]

As an example, the program below is the intersection program above with a chain threaded through it.

\[
\text{intersect1(L1, L2, L, D) :-}
\]
\[
\text{intersect1(L1, L2, L, [], done, D).}
\]

\[
\text{intersect1([X | L1], L2, Lb, Le, L, R) :-}
\]
\[
\text{compare1(X, L2, Lb, Lm, L, M),}
\]
\[
\text{intersect1(L1, L2, Lm, Le, M, R).}
\]

\[
\text{intersect1([], _, Lb, Le, L, R) :-}
\]
\[
\text{assign(Lb, Le, Done),}
\]
\[
\text{short_circuit(Done, L, R).}
\]

\[
\text{compare1(X, [X | .], Lb, Le, L, R) :-}
\]
\[
\text{assign(Lb, [X | Le], Done),}
\]
\[
\text{short_circuit(Done, L, R).}
\]

\[
\text{compare1(X, [X1 | L2], Lb, Le, L, R) :-}
\]
\[
X =\backslash = X1 |
\]
\[
\text{compare1(X, L2, Lb, Le, L, R).}
\]

\[
\text{compare1(\_, [], Lb, Le, L, R) :-}
\]
\[
\text{assign(Lb, Le, Done),}
\]
\[
\text{short_circuit(Done, L, R).}
\]

\[
\text{short_circuit([], L, R) :-}
\]
\[
R := L.
\]

The function \text{assign(X, Y, Done)} used in the procedure above is a pre-defined Strand function. It assigns to \(X\) the value of the variable \(Y\) and then it assigns the empty list [] to the variable \text{Done}. (If the variable \(Y\) is not assigned at the time the process is matched, then it is suspended). The two operations are done sequentially, so that the chain is not short circuited before the assignment has taken place.
The way in which the chain is short circuited after the assignment takes place is an example of sequencing processes in Strand. In the above, the variable Done is assigned an empty list, and the procedure short_circuit() is thus made to reduce in a specific order. This can be generalized so that the short circuit technique can be used in general for sequencing processes.

Blackboards A blackboard is a data structure used when several processes have to read and write the same data. The basic idea here is that the processes are not allowed access directly to the blackboard, where the shared data is held, but have to issue requests to a merger using incomplete messages. The merger then ensures that only one request gets through to the blackboard at a time. In the earliest implementation of Strand, the merger had to be user defined using a special data type, called the c_stream which could be written and read by more than one process. In later implementations, the c_stream was superseded by a predefined merger procedure. It can be seen that the blackboard technique is analogous to the monitor structure of Pascal Plus.
Chapter 4

Explicit Parallel Programming

4.1 Introduction

It was stated in Chapter 2 that attempts at providing general purpose means for writing parallel programs are characterized by their having, as their basis, the assumption that writing parallel programs involves a number of concepts which can be isolated and defined, and for whose implementation code may be provided for the user in one form or another. This is not dissimilar in outlook from structured programming and structured languages as implementations of a number of structures which a programmer is expected to use.

It was also observed that a better approach would be to isolate completely the partitioning process from the algorithm which is to be parallelized. This can be justified by three arguments:

- No general method for partitioning an algorithm can replace the requirement for good programming.

- Dissociating the partitioning process from the algorithm enables the programmer to look at the problem more clearly, and allow him/her to experiment with different partitioning strategies without the need for
extensive rewriting. It would also enable the parallelization of existing code without extensive rewriting.

- No specific paradigms are assumed and the programmer is free to invent and use any paradigms that best suit the problem at hand. It may well turn out that a few paradigms will be found to suffice in all applications, or for specific architectures. Be that as it may, an approach such the one advocated here will allow the development of paradigms instead of enforcing them.

This chapter will introduce and define the concepts of implicit and explicit partitioning. The manager/worker technique for achieving explicit partitioning will be described, and it will be shown how adopting such an approach leads naturally to the development of parallel programs in accordance with the definition given in Chapter 3. Throughout this chapter, all the ideas will be illustrated with an example algorithm. Finally, the developed parallel program will be compared with parallel implementations of the same algorithm in the parallel language Strand.

### 4.2 Implicit Data Assignment

By implicit data assignment is meant any way of partitioning data for processing on concurrent processes such that the partitioning forms part of the routine being processed. This is the way data is partitioned usually, and is illustrated with the following example.

**Example** The code in figure 4.1 performs numerical integration, using Simpson's rule, of the function \(4/(1 + x^2)\) with respect to \(x\) between the limits 0 and 1. By setting \(x = \tan \theta\), it can be seen readily that

\[
4 \int_0^1 \frac{1}{(1 + x^2)} dx = 4 \int_0^{\pi/4} d\theta = \pi
\]
and so the code in figure 4.1 evaluates π. This example was used by [7, 47] for comparing various parallel machines.

Parallelization of this algorithm is achieved by partitioning the data on which the body of the loop operates. The only data dependency here is caused by the variable sum to which values are added in each iteration. To parallelize the loop there are two solutions. The first, and more straightforward, is to protect the variable sum by a lock thus ensuring that only one process may increment its value at a time. But the use of a lock here would turn the would be parallel execution of the loop into an almost sequential execution. There would indeed be several processes executing the body of the loop concurrently, but only one of the processes will be allowed to execute the second statement (comprising one half of the loop body) at any one time. The second solution is to provide a private variable \( \text{par}_\text{sum} \) for each process, and to add them up at the end. This is the preferred solution. Figure 4.2 shows a complete parallel program implemented using the Sequent parallel programming library routines. (See section 2.4.4 above for a discussion of these. The interested reader is referred to [73] and to [58] for full details.) For the sake of clarity, the routines used in figure 4.2 are briefly described here in the order in which they appear and with the arguments used. (See also the comments given with the program.)

\texttt{m.set_procs(nprocs)} sets the number of concurrent processes created by \texttt{m.fork()} to \( n \) processes.

\texttt{m.fork(integrate, width, nprocs, interval)} creates a number of parallel processes, each of which will execute the routine whose name appears as the first argument, i.e. \texttt{integrate()}. The other arguments are the arguments to the latter.

\texttt{m.kill_procs()} kills all but one of the forked processes. The remaining process continues executing the rest of the program.
```c
#include (stdio.h)

main () {
    int interval;
    float width, pi;
    float integrate ();

    printf ("enter the number of strips:\n");
    scanf ("%d", &interval);
    width = 1.0/interval;
    pi = integrate (interval, width);
    printf ("pi =\n", pi * width);
    return;
}

float integrate (interval, width)
int interval;
float width;
{
float x, sum = 0.0;
int i;
    for (i = 1; i <= interval; i++) {
        x = (i - 0.5) * width;
        sum += 4.0/(1.0 + x * x);
    }
    return sum;
}

Figure 4.1: Evaluating $4 \int_0^1 \frac{1}{1 + x^2} dx = \pi$
```
...  

```c
#define MAXNPROCS 11

shared float part_sum[MAXNPROCS];
main () {
float width, sum = 0.0;
int i, interval, nprocs;
void integrate ();

input the values of interval and nprocs
width = 1.0/interval;
m_set_procs (nprocs);
m_fork (integrate, width, nprocs, interval);
m_kill_procs ();

for (i = 0; i < nprocs; i++)
sum += part_sum[i];

exit (0);
}

void integrate (width, nprocs, interval)
float width;
int nprocs, interval;
{
int i, j;
float x;

j = m_get_myid ();
part_sum[j] = 0.0;
for (i = j + 1; i <= interval; i += nprocs) {
x = (i - 0.5) * width;
part_sum[j] += 1.0/(1.0 + x * x);
}
return;
}
```

Figure 4.2: Parallel program of the integration routine using Sequent library routines
m_get_myid() returns the process’s id number, which is a unique integer associated with each of a set of concurrent processes created by a previous call to m_fork(). The returned number is among 0 to nprocs - 1.

m_next() (see figure 4.3) increments and returns the value of a global counter. A call to m_fork() resets the counter to zero.

The data partitioning used here is the easiest to implement given the library routines. If there are n concurrent processes, say, with id numbers 0, 1, ..., n - 1, then the process with id number j carries out the iterations of the loop where the loop index, i, takes the values i = j + 1, j + 1 + n, j + 1 + 2n, ..., and the work is thus divided among the processes. This is not, perhaps, how one would intuitively think of partitioning the loop; more likely, one would want to let the first process carry out the loop iterations i = 1, 2, ..., \([\text{interval}/n]\), and the second process i = \([\text{interval}/n]\) + 1, ..., and so on. This is, of course, also possible, and figure 4.3 shows another version of the routine integrate() for doing just this. It is not the intention here to demonstrate the use of the Sequent parallel library routines; these programs are only given in order to show how quickly using general purpose routines to solve even a very simple problem becomes unwieldy and interferes with the simple logic of the underlying algorithm. Nor should it be thought that the construction used here is made deliberately unclear; this is, in fact, the construction given by Sequent for data partitioning in their guide [58].

The salient feature of partitioning the data in this way is that the integration routine now performs two logically separate functions, partitioning the data and operating on it. This is not simply a result of choosing to write the code that performs the two operations in the same routine, so that the two functions can be separated by separating the lines that belong to one function from those that belong to the other and placing each group in a separate routine; the relationship between the functions is much deeper.
#define MAXNPROCS 11

shared float part_sum[MAXNPROCS];
main () {
    float width, sum = 0.0;
    int i, interval, nprocs;
    void integrate ();

    input the values of interval and nprocs
    width = 1.0/interval;
    m_set_procs (nprocs);
    m_fork (integrate, width, nprocs, interval);
    m_kill_procs ();

    for (i = 0; i < nprocs; i++)
        sum += part_sum[i];
    printf ("\pi = %f\n", sum * width * 4.0);
    exit (0);
}

void integrate (width, nprocs, interval)
float width;
int nprocs, interval;
{
    int i, j, start, finish, size;
    float x;

    j = m_get_myid ();
    part_sum[j] = 0.0;
    size = interval/nprocs;
    while ((start = (m_next () - 1) *size + 1) <= interval) {
        finish = start + size - 1;
        if (finish > interval)
            finish = interval;
        for (i = start; i <= finish; i++) {
            x = (i - 0.5) * width;
            part_sum[j] += 1.0/(1.0 + x * x);
        }
    }
    return;
}

Figure 4.3: A second way of parallelizing the routine integrate()
Further, this feature is not an accident but an inevitable consequence of the logic behind the routine. It may be described by saying that the parallelized routine has to assign to itself the data upon which it is to operate, and this will be called *implicit data assignment*.

**Definition 4.2.1** *A parallel program with data partitioning, where the partitioning of the data and its assignment to each of the concurrent processes running the same routine are carried out from within the routine itself, is said to use implicit data assignment or IDA.*

Before going on to describe how data partitioning may be disentangled from the rest of the routine, it is worth mentioning that the use of a DOACROSS-type command would have solved this problem, as far as the user is concerned anyway. But that would not allow any freedom in the partitioning of the data.

### 4.3 Explicit Data Assignment

In explicit data assignment, partitioning the data is logically and functionally separate from the routines that operate on the data. There are two immediate benefits: the routines will be identical to the original algorithm which they implement, and different strategies for data partitioning can be devised and tried without affecting the rest of the program. It will be seen that explicit data assignment brings other benefits as well.

**Definition 4.3.1** *A parallel program with data partitioning, where the partitioning of the data is carried out by a partitioning routine which then assigns the partitioned data to concurrent processes running the same routine, is said to use explicit data assignment or EDA.*

The *manager/worker* technique will be used to achieve explicit data assignment. This is a technique used in logic programming for load balancing.
It is based on the use of two generic processes, a manager and several workers. A worker requests work from the manager and acts in accordance with the received instructions. For EDA, the request will be for data, and the reply will be the data.

There will in general be three routines, manager() which is the partitioning routine, worker() which sends requests for work and receives the data, as well as the routine to be parallelized with data partitioning, par(), say. The actions of these routines will be as follows:

```c
manager ()
{
    return next data to be operated on;
}

worker ()
{
    loop:
    data = manager();
    par(data);
}

par () { ··· }
```

Access to the manager by the various workers has to be controlled; no more than one worker can have access at any one time, for otherwise two or more workers will receive the same data. The way access is controlled depends on the implementation. The implementation of such a scheme depends on the actual machine to be used; on a loosely coupled machine, communication between the workers and the manager would probably best be carried out using message passing, for in this case the requests from the various workers will be queued automatically by the system, thus providing orderly access to the manager and so ensuring correct data partitioning. On a tightly coupled machine, such as the Sequent, communication is best carried out using shared memory, and here a lock would be needed to control access to the manager; each worker has to obtain the lock before calling the manager,
and release it once it receives the data. The idea of a lock is identical to the atomic test-and-set operation used in operating systems to control access to shared resources.

Details of the implementation of such a scheme are not perhaps very important. It suffices to show that the scheme can be implemented efficiently. But there are parts of the implementation which can be made general and used for data partitioning in general, and there will be more about this in section 4.5 below and especially in Chapter 5. For this reason the integration example is reworked here using this scheme, and the program, for the Sequent is given here in full. A full program for an implementation using message passing is given in the appendix.

4.3.1 The Manager/Worker Technique

Figure 4.4 shows the full program for the integration example using the manager/worker technique for data partitioning. At the heart of the implementation are the two routines manager() and worker(). These have already been outlined above. The manager's action depends on the way in which data is to be partitioned, but the worker's action is the same for all applications; it consists of obtaining the manager's lock, a call to the manager, releasing the lock, and calling the routine which acts on the data returned by the manager.

The manager must be able to tell a worker to stop when all the data has been operated on and there is no more work. To this end the manager's reply is made up of two parts, the first part tells the worker either to "go" using the data in the second part, or to "stop".

The data returned by the manager are the three parameters, start, finish, and step. These are, respectively, the value of the loop index at the start, the value of the loop index at the last iteration, and the distance between successive iterations. In figure 4.4, the manager partitions the data in the
#include (stdio.h)
#include (parallel/parallel.h)
#include (parallel/microtask.h)

#define MAXWORKERS 10

struct job {
    int s_g;
    int start;
    int size;
};

shared float width, sum[MAXWORKERS];
shared int interval, size, start = 1;
shared slock_t lp;

void worker(), m_fork (), m_set_procs (), m_kill_procs();
    int m_get_myid();

struct job manager();

/*-----------------------*/
main (argc, argv)
int argc;
char *argv[];
{
    float fsum = 0.0;
    int nw, i;

    if (argc != 3) {
        printf ("usage: %s interval no_of_workers\n", argv[0]);
        return;
    }

    interval = atoi (argv[1]);
    nw = atoi (argv[2]);
    size = interval/nw;
    width = 1.0/interval;

Figure 4.4: Part 1 (of 3) of Data partitioning with the manager/worker technique.
s_init_lock(&lp);
m_set_procs(nw);
m_fork(worker);
m_kill_procs();

for (i = 0; i < nw; i++)
    fsum += sum[i];

printf("pi: %f
", fsum * width);
return;

/**--------------------------------*/
void worker()
{
    struct job j;
    int k, cid;
    float x, wsum = 0.0;
mbox
    s_lock(&lp);
    j = manager();
s_unlock(&lp);

    while (j.s.g == 'g') {
        for (k = j.start; k < j.start + j.size; k++)
            x = (k - 0.5) * width;
            wsum += 4.0/(1.0 + x * x);
        }
        printf("worker %d: start %d, size %d
", m_get_myid(),
                j.start, j.size);
    s_lock (&lp);
    j = manager();
s_unlock(&lp);

    cid = m_get_myid();
    sum[cid] = wsum;
    return;

    /**--------------------------------*/

Figure 4.4: Part 2 (of 3) of Data partitioning with the manager/worker technique.
struct job manager()
{
    struct job j;

    if (interval ≥ size) {
        j.s_g = 'g';
        j.start = start;
        j.size = size;
        start += size;
        interval -= size;
        return j;
    }
    else
    {
        if (interval > 0) {
            j.s_g = 'g';
            j.start = start;
            j.size = interval;
            interval = 0;
            return j;
        }
        else {
            j.s_g = 's';
            j.start = 0;
            j.size = 0;
            return j;
        }
    }
}

/* ----------------------- */

Figure 4.4: Part 3 (of 3) Data partitioning with the manager/worker technique.
int find (A, size, key);
...
{
    int i = 1;
    while (i++ <= size && A[i] != key);
    if (i > size) return 0; /* not found */
    else return i;
}

Figure 4.5: Searching an unsorted list

same way as that of the routine in figure 4.3, i.e. if the number of workers is $n$, and the total number of iterations is $\text{interval}$, then it divides the latter into blocks of size $\text{size} = \lfloor \text{interval}/n \rfloor$ each. The $i$'th request for work will then cause $\text{start}$ to be assigned the value $(i-1) \times \text{size} + 1$, $\text{finish}$ to be assigned the value $\text{start} + \text{size} - 1$, and $\text{step}$ to be assigned the value 1. Any iterations left over, i.e. $\text{interval} - \text{size} \times n$, will be assigned as a block to one of the workers.

The integration routine itself is now, up to the names of the variables used and the fact that the result is now written to an array element instead of being returned, identical to the original sequential routine of figure 4.1.

4.4 The Benefits of Explicit Data Assignment

The most immediate benefit of explicit data assignment is that the routine which is to be parallelized with data partitioning remains essentially unchanged.

Another benefit is that the way data is partitioned can be changed independently of the rest of the program. The only routine that is affected is the $\text{manager()}$. For example, in the integration problem, it is straightforward to
achieve an improved load balancing by modifying the manager so that the last block, where the number of iterations is less than the number of workers, can be partitioned into blocks of one iteration each. (For this example, such fine partitioning is at best of dubious benefit, but where the work to be carried out in each iteration is significant, such partitioning would be very efficient, and this will be seen in chapter 6 where the problem considered is much larger.) Further, it can be readily applied to partition data for a while or a do ... until loop. As a simple example, consider a routine which searches for a particular element in an unsorted list held in an array $A[\text{size}]$. Such a routine is shown in figure 4.5. In order to parallelize such a routine two problems have to be solved; the first is the partitioning of the array containing the list into blocks so that several processes can search it concurrently, each one searching one block. This problem is not different from that of partitioning the iterations of the integration example. But whereas in the previous example all the iterations have to be carried out, it is not necessary to go through the whole list here; indeed, once the element looked for is found, the program should stop. How to stop the operations of the concurrent processes is the second problem to be solved in partitioning the search. The solution comes easily using EDA since it already provides for the means of issuing a "stop" command by the manager to the workers when the latter request work. All that is needed is to make the manager check to see if the element has been found before going ahead with assigning the next block to be searched. Questions arise of how large each block should be made for maximum efficiency, and in what order is the list to be searched. It is not the intention here to address these problems; it suffices to note that all the problems can be considered and solved separately from the original algorithm. This is a considerable advantage. Figure 4.6 shows an outline solution of partitioning the algorithm using EDA.

The ease of changing the partitioning method implies that the granularity
shared found = 0;

main () { ··· }

struct work manager() {
    if (more_work && !found)
        return GO with next block;
    else
        return STOP;
}

void worker () { /* this is essentially the same as the worker of figure 4.4 */ }

void find (A, start, finish, key) {
    while (start ++ <= finish && A[start] != key);
    if (start <= finish) found = start;
    return;
}

Figure 4.6: Parallelization of the search algorithm using EDA
of the program can also be varied easily and without the need to change any routine other than the manager.

It was stated in section 4.1 that no specific paradigms are imposed on the user, who is free to devise and employ whatever method is suitable to the problem at hand. This has been borne out by the examples shown. It was also stated that paradigms may emerge through the use of this technique, and, clearly, the manager used for partitioning the loop of the integration problem can be reused in many other cases where a loop is to be partitioned.

Finally, a criticism may be made that for these advantages a high price has to be paid in terms of the length of the resulting program (compare the programs of figures 4.2 and 4.4) and the need for machine-specific knowledge to produce it. This is valid for the examples given here, and were matters to rest as they are the technique would have little to recommend it, when its disadvantages are considered, as anything more than interesting. The criticism will be answered in the next section.

4.5 Explicit Task Assignment

Pipelining apart, data partitioning and function partitioning are the only means by which an algorithm can be parallelized. In the last two sections it was shown how to separate data partitioning from the routine which is to execute concurrently on the different data using the manager/worker technique. This technique can be extended to separate function partitioning from the rest of the program, and this will be called explicit task assignment. The tasks are assigned to workers who request a manager for work. The implementation of such a scheme can be made general enough to accommodate almost any program, and in chapter 5 all the problems associated with such an implementation and their solutions are discussed fully. For the rest of this section, the scheme itself will be discussed, and it will be shown here how it
allows the writing of parallel programs in general.

In function partitioning, a program is divided into various parts, each of which is called a task. The tasks may run concurrently, or some of them may run concurrently while others have to wait for these to finish before they can run. All this, of course, depends on the algorithm and the way partitioning is carried out. The amount of work (e.g. the number of statements or the number of iterations in a loop) in each task defines the granularity of the partitions; a partition where the tasks are very small is said to have fine granularity, and one where the tasks are large is said to have coarse granularity. These concepts have already been defined and discussed in chapter 3. The concern here is with their implementation.

A general parallel program was defined as a set of tasks with a schedule (of the tasks). In the previous examples the functional partitioning of the programs, i.e. the partitioning into tasks, was not carried out, nonetheless it can be said that these programs consisted, logically and practically, of a set of tasks together with a schedule.

To see this, recall first what is meant by a schedule here is not what is meant by the same word in the world of operating systems, where it is most commonly used; a schedule here is a strict ordering of the tasks to be run based on criteria determined by the program itself. With this in mind, it can be said that any program, whether sequential or parallel, is a set of tasks with a schedule. At the level of a high level language, the tasks are just the statements, and the schedule is the implicit ordering of the statements by the rules of the language (sequencing, selection, pre-check and post-check loops, and jumps to specified statements). It is, of course, also possible to view the tasks as whole subroutines, or as blocks of sequential statements, or whatever, and the aforementioned would still hold.

It can now be seen that the previous examples did indeed consist of a set of tasks with a schedule. All the statements before the routine integrate()
had to be executed in the order implied by the C language before concurrent processes were created to execute the routine; and after this the rest of the program was executed. Logically, then, all the previous examples consisted of three sequential tasks; the first task consisted of inputting the data and forking concurrent processes; the second task was the integration routine with associated data partitioning; and the third task consisted of completing the integration and printing the result. The scheduling of these tasks was defined by the rules of the language.

As in implicit data partitioning, it is possible to define the schedule of a parallel program with function partitioning from within the program itself; but as with explicit data partitioning, it is advantageous to make the scheduling explicit, leaving the programmer free to concentrate on the partitioning without having to consider how it affects the rest of the algorithm.

As stated earlier, the manager/worker technique will be used to achieve explicit function partitioning, and in view of the fact that tasks are going to be assigned to workers by the manager, this will be called explicit task assignment.

**Definition 4.5.1** A parallel program with function partitioning, where the scheduling of the tasks and their assignment to the concurrent processes running the tasks are independent of the tasks themselves, is said to use explicit task assignment or ETA.

A parallel program can have both data and function partitioning, so that a task is executed by several concurrent processes operating on different data. To distinguish between different tasks and identical tasks with different data, the former will be called main tasks and the latter sub-tasks. Correspondingly, the manager process which assigns main tasks to workers will be called simply the manager and those which partition and assign data will be called sub-managers. It should be obvious that a parallel program with function
and data partitioning can have more than one sub-manager.

A criticism was made at the end of the last section to the effect that using EDA would result in programs that are considerably longer and more difficult than those developed using other means of data partitioning. To counter this it is simply stated here that the implementation of both EDA and ETA can be made general so that programs can be developed very easily using these techniques. Such an implementation is given in full detail in the next chapter. Below, it is shown how to turn the routine of figure 4.1 into a general parallel program with explicit data and function partitioning.

4.5.1 An Explicit Parallel Program

The general implementation of EDA and ETA described in the next chapter is in the form of a program called \textit{exdata} (standing for explicit data and task assignment). It takes as its input a "program" consisting of declarations of data, main tasks and sub-managers, a specification of the schedule, and the main tasks and the sub-managers themselves. It produces as its output a compiled parallel program which can be run on any number of processors.

For the integration problem, the function partitioning is quite trivial, for the program divides quite naturally into the initial part which reads in the parameters \textit{interval} and works out the width of each strip; then there is the subroutine \text{integrate()} which can be parallelized with data partitioning; and finally the variables \text{part_sum}_i are added, and the result output. Call these main tasks \text{initialize}(), \text{integrate}(), and \text{adprint}(), respectively. The sub-manager for \text{integrate} will be called \text{integrate.sm}().

The schedule of these tasks is very simple and proceeds thus

\begin{align*}
\text{initialize}() & \rightarrow \text{integrate}() \rightarrow \text{adprint}()
\end{align*}

Figure 4.7 shows the declarations for the data, the main tasks, the sub-manager, and the dependencies between the tasks from which a schedule is
shared float part_sum[MAXWORKERS], interval, width;
{}
/* declarations specific to exdata */
Main_Task initialize(), integrate(), adprint();
Sub_Man integrate_sm();
SMVar int ii;

DEP {
    /* dependency definitions for the schedule */
    initialize: ; integrate; .
    integrate: initialize; adprint; integrate_sm.
    adprint: integrate; ; .
} /* end dependency definitions */
}} /* end declarations specific to exdata */

Figure 4.7: Data, task, and dependency declarations for the integration problem

derived. All these are straightforward and will, in any case, be fully explained in the next chapter. But it is worth noting here that the dependency declaration consists of one definition for each main task. Each definition begins with the name of the defined task, followed by a colon. This is followed by three fields separated by semicolons. The first field contains the names of the tasks on which the defined task depends, the second field contains the names of the tasks which depend on the defined task, and the third field contains the name of the sub-manager for the defined task. A field may be empty, and a definition is terminated by a full stop.

The tasks initialize() and adprint() are trivial and are not shown here (see the appendix for the full program), and the task integrate() and its sub-manager are identical to those derived earlier for this problem using EDA.

4.6 Parallel Programming in Strand

It is useful to compare the programs of the preceding sections which have all been written using what are essentially extensions to an existing (sequential)
high level language with programs written in a general purpose parallel lan-
guage. Strand is such a language, and it has already been described in some
detail in chapters 2 and 3.

An important point to keep in mind about Strand is this: even though it
is a parallel language, a program written in it does not automatically run in
parallel. The parallelism of the language is semantic only; each line of code
of a program is an independent process which can theoretically run as soon
as the variables it needs have been assigned appropriate values, as explained
earlier. In practice all these independent processes run on the same processor
unless otherwise specified. This means that the programmer must determine
how to partition the program functionally; the language, or at least the
available implementation of the language\(^1\), will not do it. Having said this,
however, the programmer need not specify a schedule for the program; data
dependencies ensure correct scheduling.

Another point to keep in mind is that data partitioning must also be
done by the programmer through specific instructions. There is nothing
in the semantics of the language itself nor in its implementation for data
partitioning.

Notwithstanding the above, the language does provide several possibilities
for solving the integration problem in parallel. The most obvious solution
would be to use function partitioning to set up a pipeline. The pipeline
would consist of a number of processors, each of which performs a small
part of the computation. The way these processors can communicate is
readily provided by the language through the difference lists technique and
the producer consumer paradigm already mentioned.

Another way would be to use data partitioning. Although this would
require more effort on the part of the programmer, it allows more flexibility

\(^1\)The implementation available is STRAND.88, made by Strand Software Technologies
Ltd.
in the choice of the number of processors than the pipeline.

4.6.1 A Pipeline

A pipeline is set up consisting of processes each of which can be considered a producer for the next process, and a consumer of the output of the previous process. The input to and the output from each is a list. The use of difference lists allows the processes to run concurrently without the need for each of them to wait for the output list of the previous process to terminate. It is important to make each process perform as much work as every other process to avoid bottle necks. The processes which make up the pipeline are as follows:

- `generate_indices()` generates a list, Indices, of every value of the loop index.
- `sub()` subtracts 0.5 from each element of Indices and places the results in a list Sub.
- `mult()` multiplies each element of Sub by the width of each strip and places the results in a list, Mult.
- `square()` squares each element of Mult and places the results in a list, Sq.
- `add()` adds 1 to each element of Sq and places the results in a list, Add.
- `div()` divides 1.0 by each element of Add and places the results in a list, Strip.
- `sum()` sums up all the values for the areas obtained for each strip, which are the elements of Strip.

Figure 4.8 shows some of the processes above. The full program is in the appendix.

All that is needed now is a main program which assigns each of these processes to a separate processor. This, again, is straightforward and is given below:
generate_indices(Start, Finish, Indices) :-
    Start ≤ Finish |
    Indices := [Start|Indices1],
    Start1 is Start + 1,
    generate_indices(Start1, Finish, Indices1).

generate_indices(Start, Finish, Indices) :-
    otherwise |
    Indices := [].

sum([Strip|Strips], PartSum) :-
    PartSum1 is PartSum + Strip,
    sum(Strips, PartSum1).

sum([], PartSum).

Figure 4.8: The first and the last stages of the pipeline

pi(Interval, Pi) :-
    Width is 1.0 / interval,
    module_name: generate_indices(1, Interval, Indices)@1,
    module_name: sub(Indices, Sub)@2,
    module_name: mult(Sub, Width, Mult)@3,
    module_name: square(Mult, Sq)@4,
    module_name: add(Sq, Add)@5,
    module_name: div(Add, Strip)@6,
    module_name: sum(Strip, PartSum)@7,
    Pi is PartSum * 4.0 * Width.

Even though the full program is rather long, it is very logical and easy to write and understand. Further, setting up a pipeline using C and the parallel library routines would have presented serious problems in communication and synchronization. The problem has been solved using function partitioning without too much effort, and the resulting program follows very closely intuitive ideas of what a pipeline is. Strand solves all the communication and synchronization problems, and even though the program may seem to be too long, it must be remembered that the problem itself is very small;
had the problem been larger, setting up all the different processes would have been a very small cost indeed for the benefits that should ensue. One can immediately see the potential of the language for parallel applications. But the proof of the pudding is in the eating, and in the next section results of this program will be given, and it will be seen that at best it is considerably slower than the other programs in this chapter.

4.6.2 Data Partitioning in Strand

Here the program must partition the data and specify the processors on which the (identical) functions which operate on the data are to run. The full program is given in the appendix. The partitioning is determined by the user of the program, and the size of each block (of iterations) is part of the input to the program. The program spawns processes, specifies processors for them, and sets them to run on the data; the function which carries this out is called allocate() and is shown in figure 4.9. This function, like all the functions of the pipeline, is very logical and easy to understand. The spawned processes cooperate in building the list containing the partial sums using difference lists. The program here is not as intuitive as the pipeline, but this is to be expected since Strand does not provide data partitioning facilities. Once a few techniques are grasped, writing programs for data partitioning becomes easy and natural.

Again: the proof of the pudding is in the eating, and the procedure of figure 4.9, though correct, logical, and seemingly adequate, is in fact again considerably slower than those written in C.

The reason for the poor performance is the cost of communication between and synchronization of the different processors; this is the reason given by the people who implemented the language. In fact, the only way reasonable performance can be achieved is by:
allocate(N, I, W, Size, S, F, L, R) :-
    N > 0,
    loop(I, W, Size, S, F, L, M),
    N1 is N - 1,
    S1 is S + Size,
    F1 is F + Size,
    module_name: allocate(N1, I, W, Size, S1, F1, M, R)@next.

allocate(0, _ ,_ , _ ,_ ,_ , L, R) :-
    L := R.

Figure 4.9: Spawning processes and allocating data to them (the variables used are: N for number of processes, I for the interval, W for the width, S for the start, F for the Finish, L, M, and R for the left, middle, and right of the difference list.)

- Using a manager/worker strategy for data partitioning.
- Specifying explicitly communication channels between the manager and the workers.
- Allocating a whole processor for the manager.
- Using an explicit merger process for queuing requests to the manager.

A program incorporating all of this is given in the appendix. Note the use of incomplete messages for the requests. Timing results are given in the next section. Although these are better than those for the two other programs in Strand, they still leave a great deal to be desired.

4.6.3 A Final Word on Strand

Strand has its origins in concurrent logic programming and Flat Concurrent Parlog (or FCP), which in turn has its origins in Prolog [14, 56], and the similarity in syntax between it and these logic programming languages is striking. But according to the literature produced by the inventors and the
implementors of the language, Strand is a general purpose parallel program­
ming language, whereas FCP is a parallel logic programming language. This
claim cannot be sustained; the timing results of the programs of this section
should make this obvious. Of course one can always object to this on the
grounds that the example chosen is too small and trivial to have any meaning.
Well then consider a large program where each concurrent process performs
hundreds, thousands, or millions of operations on large data structures. It is
impractical to encode these operations in Strand for three reasons: the first
is that the language does not provide any data structures other than lists and
tuples; the second is that even if all the data were arranged and held in lists
and tuples, the single assignment rule means that the environment would
have its work cut out collecting garbage; and the third is that the language,
high level and based on logic programming as it is, does not lend itself to the
efficient coding of simple mundane operations like branching, incrementing a
variable, etc., which form the skeleton of any algorithm. All this seems to be
accepted by the inventors and the implementors of the language, and to solve
the problem they have provided a foreign language interface which enables
functions written in C and FORTRAN to be linked with main programs writ­
ten in Strand. There is another advantage in providing this interface, for it
allows the parallelization of existing code, with the main program acting as
a harness for parallel execution, providing communication and synchronizing
the tasks. Trials of this method have been made in the Argonne National
Laboratory and other places and the results are said to be encouraging. But
using the foreign language interface is not an easy matter; the difficulty is in
allocating memory from the heap to the foreign language application which
does not get relocated when the garbage collector does its job. This is a
complicated technical issue, and the solution provided in the shape of the in­
terface is correspondingly technically complicated. Further, it never behaved
as it should.
Apart from the two points raised above, there is a much more important criticism. The language sets out as a parallel language. Yet, it was observed that to write a parallel program, the parallelism must be programmed explicitly: concurrent processes must be assigned processors, data partitioning must be carried out by the programmer, and communication channels must be declared and defined. Further, the semantics of the language, which define each line of code as a separate process, mean that whenever statements are intended to run sequentially, or in any specific order, extra work must be done by the programmer to ensure the correct order of execution.

From a general purpose parallel language to a program partitioned by the programmer; from automatic to manual communication; from inbuilt to explicit synchronization; there is a lesson here for all who would be technical giants.

4.7 Timing Results

In this section timing results are given for the programs discussed in this chapter. Two observations are made regarding them. The first is that the performances of the programs written using the Sequent parallel library routines, with static and dynamic scheduling, and using the manager/worker technique are very close to each other. The performance of the programs written in Strand, on the other hand, differ greatly, with only the program that uses the manager/worker technique exhibiting any sort of speedup.
Table 4.1: Times (in seconds) for the \( \pi \) evaluation problem using library routines, with interval = 120,000.

<table>
<thead>
<tr>
<th>NProcs</th>
<th>Sequential</th>
<th>Static</th>
<th>Dynamic</th>
<th>Man/Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>13.29</td>
<td>13.95</td>
<td>13.88</td>
<td>14.50</td>
</tr>
<tr>
<td>2</td>
<td>7.10</td>
<td>7.07</td>
<td>7.36</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4.95</td>
<td>4.93</td>
<td>5.08</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3.77</td>
<td>3.78</td>
<td>3.90</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>3.16</td>
<td>3.14</td>
<td>3.21</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2.74</td>
<td>2.73</td>
<td>2.75</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>2.45</td>
<td>2.44</td>
<td>2.46</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>2.25</td>
<td>2.25</td>
<td>2.28</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>2.11</td>
<td>2.10</td>
<td>2.07</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>2.01</td>
<td>2.00</td>
<td>1.96</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2: Times (in seconds) for the \( \pi \) evaluation problem using Strand, with interval = 12,000.

<table>
<thead>
<tr>
<th>NProcs</th>
<th>Pipeline</th>
<th>DataPart</th>
<th>Man/Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>63.90</td>
<td>39.84</td>
<td>22.35</td>
</tr>
<tr>
<td>2</td>
<td>63.86</td>
<td>27.79</td>
<td>24.24</td>
</tr>
<tr>
<td>3</td>
<td>62.02</td>
<td>23.34</td>
<td>15.05</td>
</tr>
<tr>
<td>4</td>
<td>48.09</td>
<td>21.29</td>
<td>11.35</td>
</tr>
<tr>
<td>5</td>
<td>44.11</td>
<td>21.44</td>
<td>13.21</td>
</tr>
<tr>
<td>6</td>
<td>37.69</td>
<td>19.28</td>
<td>11.34</td>
</tr>
<tr>
<td>7</td>
<td>33.88</td>
<td>20.83</td>
<td>9.60</td>
</tr>
<tr>
<td>NProcs</td>
<td>Sequential</td>
<td>Static</td>
<td>Dynamic</td>
</tr>
<tr>
<td>--------</td>
<td>------------</td>
<td>--------</td>
<td>---------</td>
</tr>
<tr>
<td>1</td>
<td>1.00</td>
<td>0.95</td>
<td>0.96</td>
</tr>
<tr>
<td>2</td>
<td>1.87</td>
<td>1.88</td>
<td>1.81</td>
</tr>
<tr>
<td>3</td>
<td>2.68</td>
<td>2.70</td>
<td>2.62</td>
</tr>
<tr>
<td>4</td>
<td>3.53</td>
<td>3.52</td>
<td>3.41</td>
</tr>
<tr>
<td>5</td>
<td>4.21</td>
<td>4.23</td>
<td>4.14</td>
</tr>
<tr>
<td>6</td>
<td>4.85</td>
<td>4.89</td>
<td>4.83</td>
</tr>
<tr>
<td>7</td>
<td>5.42</td>
<td>5.45</td>
<td>5.40</td>
</tr>
<tr>
<td>8</td>
<td>5.91</td>
<td>5.91</td>
<td>5.83</td>
</tr>
<tr>
<td>9</td>
<td>6.30</td>
<td>6.33</td>
<td>6.42</td>
</tr>
<tr>
<td>10</td>
<td>6.61</td>
<td>6.65</td>
<td>6.78</td>
</tr>
</tbody>
</table>

Table 4.3: Speedup of the Sequent library programs.

Figure 4.10: Plot of the speedup figures of table 4.3.
Table 4.4: Speedup of the Strand programs.

<table>
<thead>
<tr>
<th>NProcs</th>
<th>Pipeline</th>
<th>DataPart</th>
<th>Man/Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>1.00</td>
<td>1.43</td>
<td>0.92</td>
</tr>
<tr>
<td>3</td>
<td>1.03</td>
<td>1.71</td>
<td>1.49</td>
</tr>
<tr>
<td>4</td>
<td>1.33</td>
<td>1.87</td>
<td>1.97</td>
</tr>
<tr>
<td>5</td>
<td>1.45</td>
<td>1.86</td>
<td>1.69</td>
</tr>
<tr>
<td>6</td>
<td>1.70</td>
<td>2.07</td>
<td>1.97</td>
</tr>
<tr>
<td>7</td>
<td>1.89</td>
<td>1.91</td>
<td>2.33</td>
</tr>
</tbody>
</table>
Chapter 5

Exdata: A Utility for Explicit Parallel Programming

5.1 Introduction

Explicit parallel programming was defined in the previous chapter, and it was there proposed as a solution to the difficulties of parallel programming in general. It is in the nature of the problem that there does not exist a solution in the way a solution exists for, say, a mathematical problem, any more than methods for developing programs in disciplined and structured ways are solutions to the problem of bad programming; or high level structured languages are solutions to the problem of unstructured programs. In all these situations the difficulties are in formulating the problems in the first place. The requirement for efficiency in coding is incompatible with that for clarity; similarly, the requirement for speed in execution is incompatible with that for making programs general and transportable across different machines; restricting the structures which a programmer may employ may reduce the likelihood of error but may also force him to adopt logically clumsy solutions. All this makes it clear that while it is highly desirable to have good programs
and good programming methods, it is difficult to say just what a good pro-
gram is. Nonetheless, few people would doubt the usefulness of high level
programming languages, or dispute the worth of transportable code; and
nobody would think that the algorithms which programmers are expected
to design and code are simple enough to make abstract data structures re-
dundant. Even so, a value judgement has to be made about where the line,
which divides the things a programmer ought to be provided with from those
which he has to design and build, should be drawn. So in the early days of
FORTRAN a programmer was expected to make do with arrays by way of
data structures, and sequencing, iteration, (primitive) selection, and uncondi-
tional jumps by way of control flow. In Pascal, the programmer is provided
with records, variant records, and sets for holding data and building arbi-
trarily complex abstract data types, and pre- and post-check loops, as well as
other structures to encourage, or even to enforce, structured programming.
Nobody has proved that the control structures are sufficient to encode any
algorithm; indeed there are flow structures which cannot be encoded using
the usual structures. See, for example,[35].

The problem to be solved here is, simply, how to make the best of parallel
computers. The solution proposed is the use of explicit programming, which
overcomes many of the difficulties and limitations of the other solutions; but
this, too, has problems of its own, viz. the resulting programs are consid-
ervably longer than those that would result using the other solutions, and a
very thorough knowledge of the machine is required. However, the code re-

---

1This ought not to be the question at all if solving problems in the “real world” is the
motivation behind the development of parallel computers, for then the parallel computers
would themselves be the solution to these “real” problems and that would be the end of
the affair! The view taken here and throughout the (computing) world is that computers
are useful general purpose machines, whose availability and user-friendliness alone create
more and more uses of them.

111
quired for implementing an algorithm in parallel using explicit programming is, by definition, separate from the algorithm itself, and so it is possible to provide a tool for carrying out these operations, leaving the programmer free to partition his program without considering how the partitioning will affect the code for the algorithm itself. An example of this has already been given in the previous chapter. In this chapter, the implementation of the tool, called \textit{exdata} (explicit data and task assignment) is given in full detail.

Before going on with the details of the implementation, it is noted here that just as in structured high level programming languages there are degrees of structuredness and "height" reflecting the designers' opinion — reached by whatever route — of how labour is to be divided between language constructs and programmer, so here the choice is wide open between, say, providing specific sub-managers and requiring the users to supply their own, and between requiring that the schedule be specified in the form of dependency definitions or through a manager routine analogous to the sub-manager routines and thus allowing for greater flexibility, and so on; and the implementation discussed below is only one of many possible implementations. This subject will be discussed more fully at the end of this chapter.

5.2 The Input

The aim is to allow a parallel program to be written in the form of subroutines that are to be executed in accordance with a specified order. Each subroutine is called a \textit{main task}, and the order of execution is the program's \textit{schedule}. In addition, a main task may be executed concurrently on more than one process, with each instance of the main task acting on different data from that acted on by the other instances. To achieve this a \textit{sub-manager} has to be specified along with the main task, and the sub-manager routine partitions the data among the instances of the main task, each of which is called a
Figure 5.1: The general form of declarations specific to \textit{exdata}

\textit{sub-task}. Further, variables used by the sub-managers for partitioning the data must be declared separately.

Now all the declarations of variables and functions that are specific to \textit{exdata} are the same as those of an ordinary program, except for their types. It is required only that they be enclosed by special delimiters. This requirement is necessary in order to facilitate the implementation, which is, necessarily, of an experimental nature. It would have been possible to arrange matters so that the input to \textit{exdata} looked like any other program file; but the time and effort needed to achieve this considerably outweigh the benefits.

Thus the input is made up of three parts. Ordinary inclusions, definitions, and data and function declarations, followed by declarations specific to \textit{exdata} which are enclosed by special delimiters, and finally the subroutines themselves, i.e. the main tasks and the sub-managers, if any. The following subsection describes the second part.

\subsection{5.2.1 Declarations Specific to \textit{exdata}}

These have the general form shown in figure 5.1.

Two pairs of curly brackets, "\{" and "\}"", serve as delimiters for these
declarations. Subroutines which carry out main tasks are declared to be of
type Main_Task, and the sub-managers are declared to be of type Sub_Man.
Variables that are used by the sub-managers have to be declared here using
the type qualifier SMVar.

The schedule of the main tasks is specified by means of dependency def-
initions which consist of the word DEP followed by a number of definitions,
one definition for each main task declared, enclosed by a pair of curly brack-
ets. As can be seen from figure 5.1 a dependency definition consists of the
following fields:

- The name of the main task whose dependency is being defined. This
  is followed by a colon.

- A list of the names of the main tasks on which the defined main task
  depends. The names in the list are separated by commas, and the
  field is terminated by a semicolon. It is, of course, possible to have a
  main task which does not depend on any other (indeed, there must be
  at least one such main task, for otherwise the program cannot start),
  and in such a case this field will be empty.

- A list of the names of the main tasks which depend on the defined main
  task. Again, the names in the list must be separated by a comma, and
  the field is terminated by a semicolon. It is, of course, possible to have
  a main task on which no other depends (indeed, there must be at least
  one such main task, for otherwise the program does not terminate),
  and in such a case this field will be empty.

- Finally, if the defined main task will be run with data partitioning, then
  it requires a sub-manager. The name of the sub-manager is placed in
  the last field, which is terminated by a full stop. This also terminates
  the definition.
Given an input as described in this section, \textit{exdata} then manipulates the input and, using the dependency definitions, builds up a parallel program, compiled and ready to be run on a \textit{Sequent} shared memory machine. To get from the input to the output, two stages are involved: source to source transformation, and compiling the results of this and linking it with a \textit{general purpose harness} to produce the runnable code. The first stage is clearly what is generally called a preprocessor and transforms the input part which is specific to \textit{exdata} into ordinary C. The second stage then involves no more than calling the system's C compiler. It would be meaningless to explain the preprocessing stage without first discussing in detail the design and working of the general purpose harness; and it is this which is explained now.

5.3 A General Purpose Harness for Implementing \textit{exdata}

Recall that the manager/worker technique is being used here to implement a general scheme for explicit parallel programming. The rôle of the harness is to use this technique in order to run the main tasks in accordance with the schedule and using the sub-managers for data partitioning. Needless to say, the harness must be general and not specific to any one application. It is entirely reasonable to give this a different name, such as scheduler, task master, or any other name that described all or some of its function. The name \textit{harness} has been chosen here simply because it has appeared in the literature e.g. [27, 77], but note the use of the qualifying \textit{general purpose} here since the harnesses mentioned elsewhere are for specific applications.

Before going on any further, it is worthwhile to clarify and distinguish between four terms that have been used, namely, explicit parallel programming, the manager/worker technique, \textit{exdata}, and general purpose harness.
Explicit parallel programming refers to a way of writing parallel programs such that function and data partitioning operations are separate from the algorithm of which the program is an implementation. The manager/worker technique is a technique borrowed from logic programming (where it is used for load balancing) to achieve explicit parallel programming. Exdata is a tool for facilitating explicit parallel programming which is based on the manager/worker technique. Thus exdata manages the running of main tasks in accordance with a specified schedule, and so functionally it is a harness.

The harness is made up of two generic routines, manager() and worker(), with the code in each being general and not restricted to any one application.

5.3.1 The Manager

The manager receives requests for work from free workers and allocates tasks according with the schedule. In this implementation the manager is a subroutine which may be executed by any of the processors cooperating in the running of the program. Such a scheme is suitable for a shared memory machine, but need not necessarily be suitable for a distributed memory machine where message passing would be used for communication. Indeed, if message passing were used for communication, then even on a shared memory machine, it would be more suitable to implement the harness such that the manager code ran on a process by itself. This aspect of the implementation has already been discussed in the previous chapter and an example of how it can be done is provided in the appendix.

The requests then are just calls to the subroutine manager(). Since there are as many copies of the subroutine as there are processors running the program, the calls must be protected by a lock. In fact, the need for locks arises on several occasions and a discussion of the use of locks is given a separate subsection.
The action of the manager is to consult a list of runnable main tasks and return an appropriate task to the worker requesting work. This list of runnable main tasks is one of several data structures that are built and maintained by \textit{exdata} for purposes of bookkeeping. All the data structures will be discussed fully later, for now it suffices to say that, initially, \textit{exdata} uses the dependency definitions to place all the tasks that are not dependent on any others, and are therefore immediately runnable, in a list called the \textit{independent task list}. As the tasks on the list are executed and finish, they are removed from the list; and as other tasks become ready to run, they are placed on this list. The independent task list thus holds, at any one time, all the runnable tasks. A main task on the list remains there until it finishes.

After consulting the list, the manager returns one of three kinds of reply, depending on what is in the list. If the list is empty, then this means that all the main tasks have finished, and the reply is \texttt{STOP}, which causes the worker that receives it to terminate. If the list is not empty, then there are two possibilities: Either all the tasks on the list have all the workers they require, in which case no more workers would be needed at that time and the manager returns a \texttt{WAIT} reply. This causes the worker that receives it to sleep. (It will be woken up as soon as a new main task becomes runnable.) Or one or more of the tasks on the list require more workers, in which case the manager returns the first such main task, and the worker starts executing it.

It will be seen later, when the data structures are discussed, that the main tasks are referred to using positive integers. It is, therefore, convenient to use integers as the type of reply that the manager sends. A positive integer is just the number of the main task to be executed, zero is used to indicate \texttt{WAIT}, and \texttt{-1} to indicate \texttt{STOP}.

The outline of the manager code is shown in figure 5.2.
```c
int manager(···)
{
p is pointer to first element of independent task list;
job = unassigned;
    if (p == NULL)        /* list is empty */
        job = -1;         /* STOP */
    else {
        do {
            if (task pointed to by p needs more workers)
                job = task number;
            else
                p = next element in list;
        } while (p != NULL && job == unassigned)
        if (p == NULL)    /* no one needs more workers now */
            job = 0;       /* WAIT */
    }
return job;
}
```

Figure 5.2: Outline of the manager’s code

5.3.2 Sub-Managers and Secretaries

It has been stated on several occasions in this thesis that a main task need not necessarily have a sub-manager, and that a sub-manager is only needed if data partitioning is required within the main task. This is, of course, true as far as the user (or each application) is concerned. It will become apparent when such issues as waking up waiting processes and maintaining the independent task list, that there are operations that need to be carried out by the harness, and that it is most convenient to associate the functions for carrying out these operations with each main task. So, as far as a user application is concerned, a sub-manager is there only for data partitioning if required; for the harness, a sub-manager performs various functions one of which might be data partitioning. In order to distinguish between the two, a sub-manager as seen from the harness’s point of view, will be called a
secretary.

The duties of a secretary are:

- Maintaining information about whether or not the main task requires more workers. This is used by the manager before replying to a worker.

- Determining when a main task is finished. This involves keeping track of the workers active running the main task, as will be seen later. When a main task finishes, it must be removed from the independent task list; further, all main tasks that depend on it have to be checked, and if any of them becomes independent, they must be added to the list (see subsection on data structures).

As with the manager, there are as many copies of each secretary as there are processors running the program. The use of locks ensures that at any one time only one instance of a secretary is allowed to run. Consequently, the operations that have to be carried out when a main task finishes are assigned to a worker to perform.

- Partitioning data where necessary. In contrast with the duties above, the code for this is not general but specific to each application and is supplied by the user.

When the manager sends a requesting worker the number of a main task, the worker then makes another request to the main task's secretary. The request is just a call to the secretary subroutine, and the latter's reply is just the data it returns. The returned data is a structure made up of two parts. The first part is one of three values to indicate STOP, GO, or SPECIAL. If the first part is GO, the second part is just the partitioned data.

A secretary tells a worker to STOP if there is no more work to be done by the main task, i.e. when the main task does not need any more workers. (Notice that this is not the same as saying that the main task has finished.)
But before the manager replies to a worker, it checks to ensure that the main task it is returning does require more workers, and it would appear, therefore, that a secretary would never need to tell a worker to STOP. In fact, such a situation can arise for two reasons. The first is that the manager may have sent a worker a main task before the main task's secretary has indicated that it does not need any more workers. The second is that a worker stays with a main task until it finishes, so that a worker which has just finished a sub-task requests more data from the secretary, and only after the latter tells it to STOP does it go to the manager for more work.

When the secretary detects the termination of its main task, it sends a SPECIAL reply, and the worker performs the cleaning up operations mentioned earlier.

Outline code for the secretary is shown in figure 5.3. Where a main task does not involve data partitioning, no sub-manager will be supplied. However, in order to make the operations the same for all main tasks, a sub-manager subroutine for all such main tasks is included in the harness. This sub-manager does not perform any data partitioning but its provision means that the code of figure 5.3 is true for all secretaries.
5.3.3 The Workers

From the foregoing it is clear that each worker has to make two requests, in general, before it can get on with executing its share of the program. The first request is to the manager, whose reply is one of three types, as seen earlier. If the manager’s reply is STOP, then the worker exits; if the reply is WAIT, then the worker goes to sleep, or pauses, awaiting a wake up call; but if the reply is a main task, then the worker makes another request, this time to the secretary of the main task. If the secretary replies with GO, then the worker executes the allotted sub-task, i.e. it executes the application subroutine with the data returned by the secretary. Upon completion of the sub-task, the worker makes another request to the secretary, and so on, so that the worker stays with the same main task until it is finished. When a worker receives a STOP reply from the secretary, it goes back to the manager requesting another main task. If the secretary replies with a SPECIAL, then the worker carries out the special cleaning up operations necessary when a main task finishes before going back to the manager for more work.

A worker can be made to wait in a variety of ways. For example, a worker can be made to loop a fixed number of times, and therefore a fixed period of time, before making another request to the manager to see if any work has become available. This has the problem that if the looping period is made too small then a bottleneck will be caused by too many workers vying for access to the shared data structures of the harness, in particular the independent task list, and thus not giving a chance for the other workers to update it. On the other hand, making the looping period long might mean that new main tasks become runnable and are executed before the waiting workers make their request to the manager. Another way of making a worker wait is to make it loop while some shared variable has a particular value; which is changed whenever a new main task becomes runnable. This has
void worker(...)
{
    task = manager();
    while (task != STOP) {
        if (task == WAIT)
            sigpause();
        else {
            sub_task = secretary();
            while (sub_task == GO) {
                execute allotted sub_task;
                sub_task = secretary();
            }
            if (sub_task == SPECIAL)
                do clean up operations;
        }
    }
    task = manager();
}
return;

Figure 5.4: Outline code for a worker

the disadvantage that all the idle workers will be taking up system time. As far as the correct functioning of the harness is concerned, there is no reason to prefer any one way to the others; but as far as efficiency is concerned, great care must be taken that waiting workers do not use up resources and so cause inefficiency. Accordingly, the way chosen here is using signals (software interrupts) that are provided by the operating system. When a worker is told to wait, it issues a sigpause() system call; and when a new main task becomes runnable and is added to the independent task list, a sigcont() is issued to each sleeping worker which causes it to carry on executing where it left off. There will be more about this in the subsection on waiting below. Figure 5.4 shows outline code for the worker.
5.3.4 Data Structures

Like every other program the harness is made up of data structures and instructions for performing operations on the data structures. As always, there is a degree of freedom in deciding how much should be done through the data structures and how much by the instructions. Already two approaches have been used for allocating tasks to workers. In the main tasks, a data structure (the independent task list) is used; in the case of sub-tasks, the sub-manager routine is used. In a later section in this chapter (section 6.5) another way of implementing the harness where the manager is done away with altogether is discussed. The other functions or operations the harness has to perform are best described by describing the data structures, all of which play the role of bookkeeping.

The Independent Task List

As mentioned earlier, the harness maintains a list of runnable tasks. At any one time, except when it is being updated, the list contains those tasks, and only those tasks which are runnable or running.

At the start of the program the list is built up using the information provided in the dependency definitions (after these have been suitably processed) and consists of the main tasks that do not depend on any others. Other tasks which are not independent are not placed on the list.

As the program runs, the tasks on the list are executed and (if all is well) finish. When they do, they are removed from the list (see below on detection of termination); and as other tasks become runnable (see below on keeping dependency information) they are placed on the list.

It follows that the independent task list is only empty when all work has been done. Conversely, when this list becomes empty, then all the main tasks have finished.
The implementation of this list is simply as a linked list, with each element of the list being a structure with two fields. The first field is a main task and the other is a pointer to the next element in the list.

The operations that are carried out on the list are: searching the list to find a main task that requires more workers — this is done by the manager; searching the list to find a particular main task to remove it — this is done by the secretary; and inserting a main task once it becomes runnable — this is done by the secretary. All these are standard operations for a linked list.

Detecting Termination of Main Tasks

Detecting when a main task has finished execution, or terminated, is essential for the correct running of a parallel program. If only one worker is required for a main task, then it is a trivial matter to detect termination: as soon as the worker makes a second request to the secretary, then it is known that the main task has terminated. But given that there are several workers cooperating in the running of the program, it is important, even in this trivial case, for the secretary to know which worker is executing the main task, for otherwise it might assume upon receiving a second request that the main task has terminated when in fact it is still running and the second request was made by another worker. With more than one worker executing a main task, the situation is more complicated.

To detect termination then a record has to be kept of which workers are executing each main task. As all the sub-tasks are done and the workers request the secretary for more data, they are told to STOP and the record updated. When the last worker comes back for more, it is sent the SPECIAL reply, and it goes and performs the necessary operations.

The record is implemented as an integer array, \texttt{active_w[]}, for each main task. The size of the array is the number of workers, one cell for each worker.
/* see figure 5.3 */
if (j.stop.go == STOP {
    ...
    if (active_w[worker id]) {
        ...
        active_w[worker id] = 0;
        w_out--;
        if (w_out == 0)
            j.stop.go = SPECIAL;
    }
} else
    if (!active_w[worker id]) {
        active_w[worker id] = 1;
        w_out++;
    }
return;

Figure 5.5: Outline code for detecting termination

The entry in a cell is 1 if the corresponding worker is executing that main
task, and 0 otherwise. In addition, an integer variable, w_out, is used to
keep the number of workers executing the main task. Each time a secretary
receives a request, it checks to see if the worker is requesting data for the
first time by checking its entry in the array. If it is there, then depending
on whether the reply is STOP or GO, its entry in the array active_w[] is
removed and w_out is decremented or no change to the record is made. If it
is not there, then depending on whether the reply is STOP or GO, no change
to the record is made or it is entered in the array active_w[] and w_out is
incremented. If w_out is decremented and becomes zero, then the main task
is finished and the reply sent will be SPECIAL. Outline code for carrying out
these operations is shown in figure 5.5.
Making Workers **wait** and Waking Them Up

The manager tells a worker to wait, or sleep, when none of the main tasks on the independent task list needs any more workers. When a worker receives a **wait** it executes a `sigpause()` system call, see [73]. The sleeping workers must be woken up at some stage in the future otherwise the program will never terminate; this is a minimum requirement. The harness must do more than this; it must ensure that whenever new tasks become runnable any sleeping workers are woken up. Waking up sleeping workers is one of the operations a worker will perform when it receives a **special** message from a secretary. There is another situation which requires that sleeping workers be woken up and it occurs when the last main task has finished executing. Here, no new main tasks will become runnable, and if there are any sleeping workers they will remain sleeping unless woken up.

A process wakes up when it receives a `sigcont()` signal. Upon receiving this signal, a process carries on executing the code just after the `sigpause()` command, which is a call to the manager (see figure 5.4). If the worker was woken up because new tasks had become runnable, then it will receive from the manager a main task, and if it was woken up because all the tasks had terminated, it will receive a **stop** from the manager.

It is necessary to keep a record of waiting workers, and for this purpose an integer array `waiting[]` and an integer variable `nwait` are used. The size of the array is the number of workers, one cell for each worker. The values held in the array are the id numbers of the waiting, or sleeping workers.

**Keeping Information of Task Dependency**

A structure is provided for each main task, `T`, declared in the input. There are two fields of the structure for keeping information of task dependencies. The first is an integer value of the number of the tasks on which `T` depends,
call this $L_1$. A task $T$ is independent if this is zero. The second is a pointer to a linked list of the tasks which depend on $T$, call this $L_2$.

Initially all the independent tasks are placed on the independent task list. When an independent task terminates, $L_2$ is inspected. If it is empty, then no other tasks are waiting for $T$ to terminate, but if it is not empty, then for each task, $t$, on the list $L_2$ the following is carried out: let $l_1$ be the number of tasks upon which $t$ depends. The task $T$ must be one of $l_1$, and so $l_1$ is decremented. If the latter becomes zero by this operation then $t$ is placed on the independent task list.

The operations mentioned above are all standard operations for a linked list.

A Data Structure for Main Tasks

In order to facilitate the above operations, a data structure is provided for each main task made up of several fields. Most of these have already been mentioned earlier, the only fields not mentioned are the lock and more. The former, and all the other locks used in the harness, will be mentioned below. The latter is just an integer variable used to denote whether the main task requires more workers or not. Its value can only be changed by the secretary, and it is looked up by the manager when requested for work. The structures are grouped together in an array, $\text{tasks}[\cdot]$, and the index of each entry is the id number of the main task. Figure 5.6 gives the declaration of the structure $\text{main_task}$ used for the main tasks.

Data Returned by the Sub-Managers

This is a structure made up of 11 integer fields. The first is used to indicate STOP and GO. The others are available for the partitioning information, and these fields can be referred to using the macros $\text{W(ARG,1)}$ through to
struct main_task {
    int    depend_ons;
    struct tle *dependents;
    slock_t lock;
    int    more;
    int    removed;
    int    w_out;
    int    active_w[MAXWORKERS + 1];
    void (*task_function)();
    struct sub_task (*sub_man)();
}

where
struct tle {
    int    task;
    struct tle *next;
}

Figure 5.6: Data Structure for Holding Information about the Main Tasks

5.3.5 Initialization

The initialization of the harness is straightforward and involves the following operations:

- Allocating functions and, where necessary, sub-managers to the main tasks.

- Building up the dependency information: the independent task list as well as the two items of dependency information for each main task.

- Initializing the locks.

Allocating functions and sub-managers simply involves assigning to the two fields task_function and sub_man the functions declared in the input.
Where no sub-manager is specified, a default sub-manager, nopart(), is assigned.

Building up the dependency information involves going through the dependency functions, which are derived from the dependency definitions (see the section on processing the input), and carrying out all the necessary operations (memory allocation and insertion) for producing the integer and the linked list which are then assigned to the two fields depend_ons and dependents.

These operations are carried out at the outset and their implementation is given in the appendix. The only feature worthy of note is the way functions are assigned. It will be noticed from figure 5.6 that the fields for the task function and the sub-manager are pointers to functions, thus making it possible to assign the appropriate function name to them.

5.3.6 Locks

As might be expected, several locks are required in order to ensure the correct cooperation of several concurrent processes working on shared data structures and using the same code. These locks are listed below:

ind_tlock This lock is used to control access to the independent task list. Clearly, only one process at any one time can insert, delete, or inspect the independent task list.

wait_lock This lock is used to control access to the shared integer variable nwait which holds the number of waiting workers. This variable is incremented just before a worker goes to sleep, and is decremented when a sleeping worker is woken up. It needs to be protected because it is possible for more than one worker to be woken up, or to go to sleep, at the same time.
lock This is a field in the structure used for holding data on the main tasks, and so there are as many of this lock as there are workers. It is used to control access to the secretary by the workers.

Note that no lock is used to control access to the manager since the manager only inspects the independent task list and the more field mentioned earlier before replying to a worker. Access to the former is controlled by ind_tl.lock. Access to the latter need not be controlled because it can only be changed by the secretary, access to which is controlled.

Note also that the first two locks are used to control access to data, whereas the third controls access to critical code.

5.4 Processing the Input

5.4.1 Introduction

In the previous sections the input to exdata and the operations required for ensuring correct execution of the input program, together with the way in which these operations are implemented, have been discussed. Recall that the input is made up of ordinary program declarations, declarations specific to exdata, and subroutines for the main tasks and the sub-managers. The second part has to be processed and turned into functions to be used in the initialization of the harness. It is possible to dispense with this requirement altogether by requiring the user to supply these functions directly, and earlier versions of exdata worked in this way. The problem here is that the code for the harness is now no longer completely independent from each application. Also, providing these functions imposes a burden on the user that the harness ought to bear. One could say that this is not very important and consider these functions separate from the harness proper because they only serve to define the schedule or task dependencies. Further, that the way the schedule
is specified does not lie at the heart of the idea behind *exdata* and leave it at
that, considering the provision of a user-friendly interface to be a development
of the harness. But there is another, and better reason for providing the
facility for declaring the task dependencies in the way it has been done here,
and it is this. By making the declarations of dependencies similar to other
declarations of the C language, one is already looking ahead to integrating
these with the rest of the language. Please note that the concern here is
not with developing C into some sort of super C for parallel programming.
The aim throughout has been to come up with a way for writing parallel
programs free from the difficulties associated with other methods (see the
introduction to chapter 4). The fact that it is now possible to look at the
result and begin to see how this might lead to new constructs that can be
integrated into an existing language is a bonus. With this in sight, and taking
into consideration the other point raised earlier about the independence of
the harness code from the application code, a way of specifying dependencies
which is similar to declarations in C was provided. These considerations are
also reflected in the way in which these declarations are processed.

To turn the declarations into functions, the following operations are re-
quired:

- They must be lexically scanned to check their correctness.
- They must be parsed and turned into some intermediate form of rep-
  resentation.
- The intermediate form must be turned into C functions.

These operations constitute the actions of a *compiler*. It is being said
here, in other words, that the declarations specific to *exdata* (from now on
they will be called just declarations) must be compiled, albeit the target
language is not machine code, but C. It might be thought that this is going
too far, like using a sledge hammer to crack a nut, and something simpler would do. There are, indeed, tools for transforming strings from one form to another. In Unix (and Dynix) there are at least two such tools, SED and AWK, see [73, 48]. Compiling the declarations can be defended by a general argument to the effect that since these declarations can be made to look like C, and given that they might, therefore, be incorporated into the language at a further stage of development, it is as well to show how this can be achieved. There is, in addition to this general argument, a practical reason for not using tools like AWK for processing the declarations and turning them to C functions. This reason will become apparent after the next subsection.

5.4.2 The Declarations

Recall from section 5.2.1 and figure 5.1 that the declarations are made up of four parts. Declarations of the functions for the main tasks, of the functions for the sub-managers, of the variables used by the sub-managers, and, finally, of the dependency definitions or the schedule. These will be taken one by one.

The Main Tasks Declaration of the main tasks consists of the key word Main..Task followed by a list of the function names. This is identical to the way functions are declared in C up to their type. The name of each function must be followed by a pair of parantheses, the names must be separated by commas, and the list is terminated by a semicolon. When processed the key word is turned into the C type void and is otherwise unchanged. In addition, a table, the symbol table, is built up and the names of the main tasks are entered there to be checked later when processing the dependency definitions. Each task is also given an id number which is used throughout the running of the harness.
**The Sub-Managers** Declarations of the sub-managers consists of the key word Sub_Man followed by a list of the function names, where the list is as described above. When processed, the key word is turned into the type struct sub_task, and is otherwise unchanged. In addition the names of the sub-manager functions are entered into the symbol table to be checked during processing of the dependency definitions.

**The Sub-Manager Variables** Variables used by the sub-managers for data partitioning must be declared separately at the outset. This is because each sub-manager will be expected to run on different processors, and although access to the sub-managers is protected by a lock, the variables they use must be declared as shared. The declaration of these variables consists of a type qualifier, the key word SMVar, followed by the type of the variables. (It turns out that it is usually appropriate to initialize these variables at this stage.) When processed, the key word is changed to the Dynix qualifier shared and the declaration is otherwise unchanged.

**The Schedule** This consists of the keyword DEP followed by a number of definitions, enclosed by a pair of curly brackets, one definition for each main task. The syntax of these definitions was described in section 2.1, and shown in figure 5.1. When processed, the schedule is turned into two C functions that are used by the harness during initialization. The two functions are alloc_functions() and build_lists(). The former just gives each main task an id number and associates with it a sub-manager. Notice that there is a duplication of effort because a symbol table has already been built containing all the main tasks and sub-managers, but this table will disappear once the preprocessing is finished. If and when the preprocessing operations are incorporated into the language's compiler, then there will be no need for this duplication. The latter function builds up the integer depend_ons and
the linked list dependents associated with each main task.

It can now be seen why a simple string processing tool such as AWK — which is based on a text editor — would not be satisfactory, for in addition to changing certain words into others, tables are needed for checking the correctness of the dependency definitions, and for this more powerful tools are needed. The ones used here are lex and yacc.

5.4.3 A Diversion: lex and yacc

Lex and yacc are tools for building compilers. Lex generates a scanner, or lexical analyser, for a given set of regular expressions, and yacc generates a parser for a specified grammar in its input. It is not the intention here to describe lexical and syntactic analysis. This field is quite difficult and is very well covered in the literature [2, 12]. Lex and yacc are also described in [73].

Briefly, lex takes as its input regular expressions together with an action for each regular expression. The action usually is to return a token (to be used by the parser) and to make an entry in a symbol table. The output is a finite state machine with driving tables in the form of a C program; this is the scanner. Lex also has a (limited) capability of recognizing contexts. The resulting scanner then works by taking as its input the source program, here the declarations, making entries in the symbol table for each function it finds, and returning appropriate tokens to the parser.

Yacc (standing for yet another compiler compiler) is, despite its name, a parser generator. It takes a grammar together with an action for each rule (the input is, technically, called a single attribute grammar) and generates a parser in the form of a C program. The resulting parser then takes as input the tokens returned by the scanner, and transforms them into some intermediate representation form. This is entirely up to the user and is decided by the actions associated with the rules of the grammar. If the
grammar contains ambiguities, yacc allows these to be resolved by providing a facility for specifying precedence between items in the input.

There are also several variables that are common to both lex and yacc. These are provided to facilitate the cooperation of these tools.

In addition to the regular expressions and the grammar, the user must also supply a host of supporting routines for building the symbol table, handling errors in the input, building intermediate representations, as well as the main program which calls the various parts of the compiler.

5.4.4 Lexical Analysis of the Declarations

To build a scanner for the declarations, they must be written in the form of regular expressions in the form required by lex. The full input of lex is given in the appendix. Here by way of example, the regular expressions for describing the dependency definitions are given in figure 5.7. What the figure means is that if the scanner comes across a left (or right) curly bracket, it produces the token LBRAC (or RBRAC) to the parser. If it comes across a function name, it looks it up in the symbol table, puts a pointer to its entry there in the variable yylval (which will be used by the parser), and produces the token FNAME to the parser. A colon, comma, or semicolon is simply passed to the parser. A full stop causes the scanner to produce the token DefEnd for the parser. This is rather a cursory description of the input to lex, but there is no way by which to describe everything in full, even for a small application such as the one under consideration here, in a reasonable number of words. There are no parts of lex that can be left out while still keeping the description adequate. To complicate matters further, the variables whose names begin with yy are meant for the cooperation of lex and yacc; while this makes it easier to write a compiler it does not make a good description of how the compiler is built any easier. The important thing to note is that
Figure 5.7: \textit{Lex} regular expressions for recognizing dependency definitions

the declarations are given as regular expressions, and everything else must be supplied by the user.

5.4.5 Syntactic Analysis of the Declarations

As before, it is impossible to give an adequate description of the input to \textit{yacc} without resorting to a full description of \textit{yacc} itself and a good many topics relating to grammars and parsing. The appendix shows the full input and the supporting routines for \textit{yacc} for the declarations. The rules are for a type2 grammar (context sensitive) and the actions associated with the rules build an intermediate representation in the form of linked lists. For each main task $T$ a structure is maintained to hold information about its sub-manager, and the two lists, one for the main tasks that $T$ depends on, and the other for the main tasks that depend on $T$. Here again there is duplication of effort because such lists have to be built up again by the harness during initialization. As an example of the input to \textit{yacc}, figure 5.8 shows the rules and actions for parsing the list of dependent tasks. What the figure means is that a list of dependent tasks is defined to be one of the following. Empty, in this case the attribute of the defined list \textit{depends} is assigned the empty list. A single function, i.e. just one task, and here the attribute is the list containing this task only; the supporting routine \textit{mkdep()} is responsible for
Figure 5.8: Yacc grammar rules for list of dependent tasks

building this attribute. Finally, the list might be made up of more than one
task, in which case the attribute is built by joining the tasks together in a
linked list (note that this is done recursively); the joining is carried out by
another supporting routine, join().

5.4.6 Producing the C Functions

After turning the declarations and dependency definitions into an appropriate
intermediate form, it is straightforward to produce the required functions for
the harness. The routine for performing these operations simply follows the
linked lists associated with each main task transforming them into function
lines. This routine is called printout() and is given in the appendix. This
appendix contains, in addition to the input to lex and yacc, a header file for
the data structures used in the transformation, and the main program for the
whole operation. This is very short and is given in figure 5.9. It consists of a
call to the parser yyparse() and then a call to printout(). Notice that there is
no call to the scanner, this is done by the parser itself, which repeatedly calls
the scanner for successive tokens from the input. When the parser finishes,
it places the final attribute in the variable yylval (the so called left value of
the input). This is defined as a union in order to allow various forms of data
to be used. The routine yywrap() is another of the details that have to be
main()
{
    int yywrap();
    void printout();
        yyparse();
        printout(yylval.par_def);
    return;
}

5.5 Other Considerations

Exdata was built as a tool for parallel programming using explicit function and data partitioning. The previous sections have described in detail the inner workings of the tool, and it has been shown that the objective of separating the operations for scheduling tasks from the underlying algorithm has been achieved. The implementation given here is not the only one possible. Indeed another implementation was made in which no manager was required. In that version, the workers were made to loop continuously around the array task[] looking for runnable tasks that require workers. This version is simpler than the one described, but has the disadvantage that it keeps all the workers busy occupying system time and continuously competing for tasks even when no work is available.

It should also be said that the manager is part of the harness in this implementation and cannot therefore be changed. It follows that the schedule is defined once only at the start and cannot be changed during execution. The schedule for function partitioning can thus only be static. By contrast, the sub-managers are supplied by the user who is free to use static or dy-
dynamic scheduling for data partitioning. There is no reason why \textit{exdata} should not allow the user to supply the manager routine as well, thus allowing for dynamic scheduling in function partitioning, see section 9.2.

One might also consider allowing for recursion where a main task calls itself. This is not possible with this implementation.

Sub-managers return data to workers in the form of a structure which contains integer fields only. It is desirable, though the reason is not necessarily a good one, to allow sub-managers to return any type of data allowed in the language. This would require changes to the C language or extensive preprocessing. The reason why only integers are allowed is that they seem to suffice for data partitioning; any other data can always be passed using shared memory.

There are advantages to \textit{exdata} other than the main reason for it. It can be easily extended and made to behave like a compiler. A true compiler would require modifications to the existing compiler, but it has already been shown that this is possible and, indeed, some headway was made by the use of \textit{lex} and \textit{yacc}.

It is straightforward to use code written in another language such as Fortran. All that is needed is to link the object code of the foreign routines with that of the harness, and linkers already exist for this purpose.

\textit{Exdata} is built using existing parallel library routines. It is, therefore, not difficult to incorporate other routines and facilities (such as vectorization routines) without in any way altering anything fundamental.

The schedule obviates the need for some of the parallel library routines. In particular, there is no need for barrier synchronization.

Finally, routines for giving timing information on the main tasks and sub-tasks have been provided, and these are very useful in evaluating the performance of different strategies, say, of parallel programming.
Chapter 6

Solving a Block Diagonal Bordered System

6.1 Introduction

Following the description of the design and implementation of exdata, it will be used in the implementation of parallel programs. In this and the next chapter two substantial applications will be presented. The aim is to show that the performance is at least acceptable, and its use is convenient.

As an example of the use of exdata in a large program, a linear set of equations, such as equation 6.1, is solved.

\[ A \cdot x = d \]  

(6.1)

where \( A \) is the matrix of coefficients, \( d \) is a vector of known quantities, and \( x \) is the vector of unknown quantities.

\( A \) is a block diagonal bordered matrix, so that equation 6.1 can be written
as in equation 6.2.

\[
\begin{pmatrix}
A_1 & 0 & \cdots & 0 & B_1 \\
0 & A_2 & \cdots & 0 & B_2 \\
\vdots & & \ddots & & \vdots \\
0 & 0 & \cdots & A_N & \ B_N \\
C_1 & C_2 & \cdots & C_N & A_{N+1}
\end{pmatrix}
\begin{pmatrix}
x_1 \\
x_2 \\
\vdots \\
x_N \\
x_{N+1}
\end{pmatrix}
= 
\begin{pmatrix}
d_1 \\
d_2 \\
\vdots \\
d_N \\
d_{N+1}
\end{pmatrix}
\]  
(6.2)

where each block \( A_i \), for \( i = 1, 2, \ldots, N \), is a square matrix with dimensions \( n_i \times n_i \); each block \( B_i \), for \( i = 1, 2, \ldots, N \), is a matrix with dimensions \( n_i \times m \), for some \( m \), with \( n_i \) not necessarily equal to \( m \); each block \( C_i \), for \( i = 1, 2, \ldots, N \), is a matrix with dimensions \( m \times n_i \); and the block \( A_{N+1} \) is a matrix with dimensions \( m \times m \). Each \( x_i \), for \( i = 1, 2, \ldots, N \), is a column vector of \( n_i \) elements; each \( d_i \), for \( i = 1, 2, \ldots, N \), is a column vector of \( n_i \) elements. \( x_{N+1} \) and \( d_{N+1} \) are column vectors with \( m \) elements each. For example, if \( N = 3, m = 8, n_1 = 2 = n_3, \) and \( n_2 = 3, \) then the system will be as shown in equation 6.3 below.

\[
\begin{pmatrix}
A_1(2 \times 2) & B_1(2 \times 8) \\
A_2(3 \times 3) & B_2(3 \times 8) \\
A_3(2 \times 2) & B_3(2 \times 8) \\
C_1(8 \times 2) & C_2(8 \times 3) & C_3(8 \times 2) & A_4(8 \times 8)
\end{pmatrix}
\begin{pmatrix}
x_{1}(2) \\
x_{2}(3) \\
x_{3}(2) \\
x_{4}(8)
\end{pmatrix}
= 
\begin{pmatrix}
d_{1}(2) \\
d_{2}(3) \\
d_{3}(2) \\
d_{4}(8)
\end{pmatrix}
\]  
(6.3)

Where \( A_i(m \times n) \) means that the block \( A \) is of dimension \( m \times n \) etc., and \( x_i(m) \) means that the vector \( x_i(n) \) has \( n \) elements. Such a system arises from applying the finite element method to the analysis of field problems. The sparsity of the matrix of coefficients is caused by the uncoupling between distant nodes which this method achieves, and the particular shape of the matrix (diagonal bordered) is the result of partitioning the nodes into internal and shared nodes. Internal nodes are those that lie within an element, and shared nodes lie on the edges of an element and are thus shared by more than
one element. It is not the intention here to describe in full the finite element method, nonetheless some details will be given below in order to show how the system of equation 6.2 arises.

The matrix of coefficients as described earlier is particularly suitable for parallel solution, since the vectors $x_i, i = 1, 2, \ldots, N,$ can all be determined concurrently. Equation 6.2 will be solved using two direct methods, Gaussian elimination and Choleski factorization. Each method is implemented sequentially and in parallel using {	extit{exdata}} and parallel library subroutines, and the results will be compared.

An important part of the implementation is the way block matrices are represented, and this, too, will be discussed below.

### 6.2 The Finite Element Method

The finite element method is a method widely used by engineers and scientists for the analysis of structures and field problems. These can be represented using partial differential equations whose analytical solution is often impossible. The approach then is to partition the region under consideration into a set of "finite elements" and to approximate the behaviour of the system within each finite element by functions, usually polynomials, that are amenable to analysis. A solution for the overall system is then obtained by summing up the approximate functions over all the elements of the region. The region will be a straight line in one dimensional problems (in which case the partial differential equation is, properly speaking, an ordinary differential equation), an area in two dimensional problems (such as wave propagation on a string where the two dimensions are spatial — along the length of the string — and temporal), a volume in three dimensional problems (such as magnetic flux in a sphere), and so on. The problem is posed in the form of a partial differential equation the solution of which is a function which
describes the behaviour of the system.

Partial differential equations and the finite element method are not our concern. The account that follows is therefore necessarily a mere adumbration of these subjects with the main aim being to show how the set of equations 6.2 is derived. The next section follows closely the exposition in [19].

6.2.1 Structural Analysis

The finite element method was first used and developed by engineers in the analysis of structures before it was used for field problems. A few words on this subject are appropriate for this reason. Moreover, simple structures provide a very clear way of illustrating the ideas behind the finite element method.

Consider structures made from thin rods which obey Hooke's law

\[ F = ku \]  \hspace{1cm} (6.4)

where \( F \) is the force applied along the length, \( u \) is the displacement caused by the force, and \( k \) is the constant of proportionality, called the stiffness of the material.

If \( A \) is the cross-sectional area of the rod, then the stress, \( \sigma \), is defined as

\[ \sigma = F/A \]  \hspace{1cm} (6.5)

and if \( l \) is the natural length of the rod, then the strain, \( \varepsilon \), is defined as

\[ \varepsilon = u/l \]  \hspace{1cm} (6.6)

Young's modulus, \( E \), is defined as the ratio of stress to strain

\[ E = \sigma/\varepsilon \]  \hspace{1cm} (6.7)

and \( E \) is a constant for a given material.
The potential energy per unit volume of the rod is given by

\[
\text{strain energy} = \frac{1}{2} \sigma \varepsilon
\]  
(6.8)

Consider now the spring whose stiffness is \( k \) in the system of three springs shown in figure 6.1. Here, there are three elements in the system. Consider the middle spring:

since the system is in equilibrium \( F_1 + F_2 = 0 \)
and, from Hooke's law \( F_2 = k(u_2 - u_1) \) so that \( F_1 = k(u_1 - u_2) \) or we can write

\[
\begin{pmatrix}
  k & -k \\
  -k & k
\end{pmatrix}
\begin{pmatrix}
  u_1 \\
  u_2
\end{pmatrix}
=
\begin{pmatrix}
  F_1 \\
  F_2
\end{pmatrix}
\]

or

\[ k^e u^e = f^e \]

where \( k^e \) is called the \textit{element stiffness matrix}, \( u^e \) is the \textit{element nodal displacement vector} and \( f^e \) is the \textit{element nodal force vector}.

The equations governing the overall system are derived by combining those obtained for each element in the following way:

Consider figure 6.2 which depicts a system of two springs with stiffnesses \( k_1 \) and \( k_2 \). At equilibrium, we have
Figure 6.2: Two springs with stiffnesses $k_1$ and $k_2$

\[
F_1 + F_2 + F_3 = 0
\]
also
\[
F_1 = k_1(u_1 - u_2)
\]
and
\[
F_3 = k_2(u_3 - u_2)
\]
so that
\[
F_2 = -k_1u_1 + (k_1 + k_2)u_2 - k_2u_3
\]
The last three equations can be written as
\[
\begin{pmatrix}
  k_1 & -k_1 & 0 \\
  -k_1 & k_1 + k_2 & -k_2 \\
  0 & -k_2 & k_2
\end{pmatrix}
\begin{pmatrix}
  u_1 \\
  u_2 \\
  u_3
\end{pmatrix}
= 
\begin{pmatrix}
  F_1 \\
  F_2 \\
  F_3
\end{pmatrix}
\]
or
\[
KU = F \tag{6.9}
\]
where $K$ is the overall stiffness matrix, $U$ is the overall nodal displacement vector, and $F$ is the overall nodal force vector.

The overall stiffness matrix $K$ is related to the element stiffness matrices $k^{(1)}$ and $k^{(2)}$ as follows:

Now associate a Boolean matrix $L^e$ with each element. For the system under consideration, $L^{(1)}$ is
\[
\begin{pmatrix}
  1 & 0 & 0 \\
  0 & 1 & 0
\end{pmatrix}
\]

and \( L^{(2)} \) is
\[
\begin{pmatrix}
0 & 1 & 0 \\
0 & 0 & 1
\end{pmatrix}
\]
Now,
\[
L^{(1)^T}k^{(1)}L^{(1)} = \begin{pmatrix}
1 & 0 & k_1 & -k_1 \\
0 & 1 & -k_1 & k_1 \\
0 & 0 & 0 & 0
\end{pmatrix}
\begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0
\end{pmatrix}
= \begin{pmatrix}
k_1 & -k_1 & 0 \\
-k_1 & k_1 & 0 \\
0 & 0 & 0
\end{pmatrix}
\]
Similarly,
\[
L^{(2)^T}k^{(2)}L^{(2)} = \begin{pmatrix}
0 & 0 & 0 \\
0 & k_2 & -k_2 \\
0 & -k_2 & k_2
\end{pmatrix}
\]
So that
\[
K = L^{(1)^T}k^{(1)}L^{(1)} + L^{(2)^T}k^{(2)}L^{(2)}
\]
Similarly,
\[
F = L^{(1)^T}f^{(1)} + L^{(2)^T}f^{(2)}
\]
Now, \( u^{(1)} = L^{(1)^T}U \) and \( u^{(2)} = L^{(2)^T}U \) so that the overall system can be built up using the element equations and \( L^e \)
\[
(L^{(1)^T}k^{(1)}L^{(1)} + L^{(2)^T}k^{(2)}L^{(2)})U = L^{(1)^T}f^{(1)} + L^{(2)^T}f^{(2)}
\]
or
\[
\sum_e (L^e)^T k^e L^e)U = \sum_e L^e f^e
\]  \( 6.10 \)
The Boolean matrix \( L^e \) is called a selection matrix in [19] and a localization operator in [57]. This idea of assembling equations for individual elements to arrive at the overall system of equations is central to the finite element method and will be used in field problems.
Notice that equation 6.9 cannot be solved because $K$ is singular. The singularity is caused by the fact that the system allows rigid body motion. The singularity can be removed by fixing one of the nodes, say node 1, so that $u_1 = 0$. Note also that $K$ is symmetric, and that $K_{13} = K_{31} = 0$, so that there is no coupling between nodes not connected with the same element.

**Virtual Displacement and Minimum Energy**

**Virtual Displacement** The principle of virtual displacement and the principle of minimum potential energy are both ways of analysing structures which are equivalent. The motivation for mentioning them here is that their equivalence allows the ideas used in structural analysis to be extended to field problems where one can only use potential energy to solve the problems.

The principle of virtual displacement states that for a virtual nodal displacement the total work done is zero. In a virtual displacement, the forces acting on the nodes are considered unchanged.

For the spring of figure 6.1, if the displacements $u_1$ and $u_2$ are considered to be virtual displacements, then the external work done by the forces is just the product of the forces and the extensions, or

$$W_E = F_1 u_1 + F_2 u_2$$

where $W_E$ stands for the external work. The internal work, $W_I$, done by the springs is the product of the elastic force and the extension, or

$$W_I = -k(u_2 - u_1)^2$$

From the principle of virtual displacement $W_E + W_I = 0$, so that

$$F_1 u_1 + F_2 u_2 = k(u_2 - u_1)^2$$

or

$$u^T f = u^T k^e u$$
and since \( u \) is arbitrary, it follows that \( k^e u^e = f^e \) which is the same equation as that obtained for the element using equilibrium analysis.

The strain energy can also be expressed using the stresses and strains. The strain \( \varepsilon \) is (see equation 6.6 above)

\[
\varepsilon = (u_2 - u_1)/l = \begin{pmatrix} -1/l & 1/l \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \end{pmatrix}
\]  
(6.11)

and the stress, \( \sigma \), is (see equation 6.5)

\[
\sigma = k(u_2 - u_1)/A = kl\varepsilon/A
\]  
(6.12)

For virtual displacement, the strain energy will be given by \( f_v \varepsilon \sigma \) (cf. equation 6.8), so that

\[
u^e f^e = \int_V \varepsilon \sigma dV
\]  
(6.13)

In general, for a 3D element, the strain will be a vector of six dimensions, three along the \( x, y, \) and \( z \) axes, and three shear strains along the \( xy, yz, \) and \( zx \) planes. The stress will similarly be a vector of six dimensions. In such a case, the strain energy due to a shear displacement will be \( f_v \varepsilon^T \sigma dV \), so that the principle of virtual displacement, equation 6.13, will be given by

\[
u^e f^e = \int_V \varepsilon^T \sigma dV
\]  
(6.14)

The stress \( \sigma \) and the strain \( \varepsilon \) can be related by a matrix \( K \) which involves Young's modulus for the material and its Poisson's ratio so that \( \sigma = K \varepsilon \).

(The details are not relevant, what is important is the form of the ensuing equations. For details see [69].)

Let \( a \) be the vector of the displacements of a point within the element. \( a \) is related to the strain \( \varepsilon \) by a matrix of differential operators \( D \), so that

\[
\varepsilon = D^T a.
\]

The displacement \( a \) of a point within the element is related to the nodal displacement \( u \) by a matrix \( N^e \) called the matrix function of position. This matrix depends on the element and on the user.
From the aforementioned, equation 6.14 can be rewritten as

\[ u^T f^e = u^T \left( \int_V \alpha^T \kappa \alpha dV \right) u^e \]  
(6.15)

where

\[ \alpha = D^T N^e \]  
(6.16)

Since equation 6.15 holds for an arbitrary displacement \( u \), it follows that

\[ f^e = k^e u^e \]  
(6.17)

where the element stiffness matrix is given by

\[ k^e = \int_V \alpha^T \kappa \alpha dV \]  
(6.18)

**Minimum Potential Energy** The principle of minimum potential energy states that for any system under equilibrium, the potential energy is a minimum.

Both the stress and the strain are related to the nodal displacement. In one dimension, this can be expressed by saying that \( \sigma = \sigma(u) \) and \( \epsilon = \epsilon(u) \).

Under a virtual displacement \( \Delta u \), \( \sigma \) and \( \epsilon \) will become

\[ \epsilon = \epsilon_E + \Delta u \frac{d\epsilon}{du}\bigg|_E + \frac{\Delta u^2}{2} \frac{d^2\epsilon}{du^2}\bigg|_E + \cdots \]

and

\[ \sigma = \sigma_E + \Delta u \frac{d\sigma}{du}\bigg|_E + \frac{\Delta u^2}{2} \frac{d^2\sigma}{du^2}\bigg|_E + \cdots \]

where \( E \) stands for equilibrium.

The internal work, to the first degree, is given by

\[ \int_V \sigma dV = \int_V \epsilon_E \sigma dV + \Delta u \left( \frac{d}{du} \int_V \epsilon dV \right)\bigg|_E \]

and this equals the external work \( \Delta u F \) since \( F \) remains constant under virtual displacement.

It follows from the principle of virtual displacement that \( \Delta u F = \int_V \sigma_E \epsilon_E dV \) and hence, at equilibrium

\[ \frac{d}{du} \int_V \sigma dV = 0 \]
i.e. at equilibrium the strain energy is stationary. This stationary value is a minimum, for if any work is done and the system is no longer in equilibrium then its internal energy must increase, so that

\[
\frac{\Delta u^2}{2} \left( \frac{d^2}{du^2} \int_V \sigma \epsilon \, dV \right) \bigg|_E > 0
\]

For systems in more dimensions, the equivalence of the principle of virtual displacement and that of minimum potential energy leads to equations of the general form

\[
\frac{\partial}{\partial u_i} \int_V \epsilon^T \sigma \, dV = 0, \quad i = 1, \ldots, n \quad (6.19)
\]

and since \( \sigma = \kappa \epsilon \),

\[
\frac{\partial}{\partial u_i} \int_V \epsilon^T \kappa \epsilon \, dV = 0, \quad i = 1, \ldots, n \quad (6.20)
\]

And so the problem becomes one of finding a function \( \epsilon \) of the displacements \( u_1, u_2, \ldots, u_n \) which minimizes the integral \( \int_V \epsilon^T \kappa \epsilon \, dV \).

### 6.2.2 Partial Differential Equations and Variational Methods

As stated earlier, field problems are described using partial differential equations. The solution of these equations turns out, in many cases, to require the minimization of certain integrals just as in the principle of minimum potential energy.

Let \( L \) be a general second-order differential operator. In two dimensions, \( L \) is defined by

\[
L \phi = a \frac{\partial^2 \phi}{\partial x^2} + b \frac{\partial^2 \phi}{\partial x \partial y} + c \frac{\partial^2 \phi}{\partial y^2} + F(x, y, \phi, \frac{\partial \phi}{\partial x}, \frac{\partial \phi}{\partial y})
\]  

(6.21)

The equation

\[
L \phi = f
\]  

(6.22)
is a linear second order partial differential equation iff \( L \) is defined by equation 6.21 and such that \( a = a(x, y) \), \( b = b(x, y) \), and \( c = c(x, y) \) are functions of \( x \) and \( y \) only (if any of them is a function of \( \phi \) itself then \( L \) is non-linear).

Partial differential equations (p.d.e.'s) are classified according to the coefficients of their second order derivatives. Equation 6.22 is said to be

**elliptic** if \( b^2 < 4ac \). An example is Poisson's equation, \(-\nabla^2 \phi = \epsilon/\rho\) which describes the electrostatic potential \( \phi \) in a material whose permittivity is \( \epsilon \) and where the charge density is \( \rho \). (The operator \( \nabla^2 \) is defined as \( \partial^2/\partial x^2 + \partial^2/\partial y^2 \).)

**parabolic** if \( b^2 = 4ac \). An example is the heat equation \( \partial^2 \phi/\partial x^2 = (1/k)(\partial \phi/\partial t) \) which describes the temperature \( \phi \) along a thin rod whose thermal diffusivity is \( k \), and where \( t \) stands for time.

**hyperbolic** if \( b^2 > 4ac \). An example is the wave equation \( \partial^2 \phi/\partial x^2 = (1/c)(\partial^2 \phi/\partial t^2) \) which describes the transverse displacement \( \phi \) of a vibrating string, where \( c \) is the speed of the wave and \( t \) stands for time.

The study of p.d.e.'s is not the main concern here. It turns out, however, that there is a close analogy between elasticity problems, such as those considered earlier, and problems modelled by (the elliptic) Poisson's equation. For this reason, and in order to demonstrate the finite element method, Poisson's equation only will be considered. It should be added that the finite element method can also be used to solve other p.d.e.'s not just Poisson's equation.

To describe a problem, the p.d.e. must be defined on a particular region \( R \) in two dimensions with a boundary \( C \). (In three dimensions the p.d.e. will be defined on a volume \( V \) within a surface \( S \) and so on.) Associated with the p.d.e. then, there will be the **boundary conditions**. These are of the following three types.
Dirichlet

\[ \phi = g(s) \quad (6.23) \]

Neumann

\[ \frac{\partial \phi}{\partial n} = j(s) \quad (6.24) \]

Mixed

\[ \frac{\partial \phi}{\partial n} + \sigma(s)\phi = h(s) \quad (6.25) \]

If the right hand side of any of the equations 6.23–6.25 is zero, then that boundary condition is said to be \textit{homogeneous}, and \textit{non-homogeneous} otherwise. The letter \( n \) in these equations stands for the normal, and \( s \) is the boundary.

A differential operator \( L \) is said to be \textit{self-adjoint} iff the expression

\[ \iint_{\mathcal{R}} \psi L\phi \, dx\,dy - \iint_{\mathcal{R}} \phi L\psi \, dx\,dy \quad (6.26) \]

is a function of \( \phi, \psi \) and their derivatives on the boundary. In particular, for homogeneous boundary conditions, \( L \) is self-adjoint iff

\[ \iint_{\mathcal{R}} \psi L\phi \, dx\,dy - \iint_{\mathcal{R}} \phi L\psi \, dx\,dy = 0 \quad (6.27) \]

The operator \( L \) is said to be \textit{positive-definite} iff

\[ \iint_{\mathcal{R}} \phi L\phi \, dx\,dy \geq 0 \quad (6.28) \]

equality holding iff \( \phi \equiv 0 \).

For example, \(-\nabla^2\) is self-adjoint since

\[ \iint_{\mathcal{R}} \psi(-\nabla^2)\phi \, dx\,dy - \iint_{\mathcal{R}} \phi(-\nabla^2)\psi \, dx\,dy = \int_{\mathcal{C}} \left( \frac{\partial \psi}{\partial n} - \frac{\partial \phi}{\partial n} \right) \, ds \]

using Green's theorem. Further,

\[ \iint_{\mathcal{R}} \phi(-\nabla^2)\phi \, dx\,dy = \iint_{\mathcal{R}} \nabla\phi \cdot \nabla\phi \, dx\,dy - \int_{\mathcal{C}} \phi \frac{\partial \phi}{\partial n} \, ds \]
using $\phi \nabla^2 \phi = \nabla.(\phi \nabla \phi) - \nabla \phi \cdot \nabla \phi$ and the divergence theorem. Where $\nabla \phi$ is the gradient (grad) of $\phi$, and $\nabla \cdot a$ is the divergence (div) of the vector function $a$.

So that if $\phi$ satisfies either of the homogeneous Dirichlet or Neumann boundary conditions, then $-\nabla^2$ is positive-definite.

If $\phi$ satisfies the homogeneous mixed boundary condition, then

$$- \int_c \phi \frac{\partial \phi}{\partial n} \, ds = \int_c \sigma \phi^2 \, ds$$

and it follows that $-\nabla^2$ is positive-definite provided $\sigma > 0$.

**Uniqueness of Solution** If $L$ is linear and positive-definite, then the solution of $L\phi = f$ is unique. This is proved by the following argument. Let $\phi_1$ and $\phi_2$ be two solutions, and let $\psi = \phi_1 - \phi_2$. Since $L$ is linear, it follows that $L\psi = L\phi_1 - L\phi_2 = 0$, and so $\iint_R \psi L \psi \, dx \, dy = 0$. Since $L$ is positive-definite, it follows that $\psi \equiv 0$, so that $\phi_1 = \phi_2$.

**Solution of $L\phi = f$ and Minimization of Energy**

The equation

$$- \nabla^2 w = p/T$$

(6.29)

describes the transverse displacement $w$ of a membrane stretched across a frame with boundary $C$, subjected to pressure $p(x,y)$, and with constant tension $T$. The surface of the membrane over which equation 6.29 holds is denoted by $R$.

Consider the behaviour of the system when it is given a small displacement $\Delta w$.

The total work done by the force is

$$\Delta \iint_R p \Delta w \, dx \, dy = \iint_R p \Delta w \, dx \, dy = \iint_R -T \nabla^2 w \Delta w \, dx \, dy$$
and using $\psi \nabla^2 \phi = \nabla.(\psi \nabla \phi) - \nabla \psi \cdot \nabla \phi$ and the divergence theorem, the expression for the work done becomes

$$\int \int_{\mathbb{R}} T \nabla(\Delta w).\nabla w \, dxdy - \int_{\mathbb{C}} T \Delta w \frac{\partial w}{\partial n} \, ds$$

It can be shown by algebraic manipulation that $\nabla(\Delta w).\nabla w = \frac{1}{2} \Delta |\nabla w|^2$ so the expression for the work, which is equal to the energy in the membrane, is given by

$$\int \int_{\mathbb{R}} \frac{1}{2} T \Delta |\nabla w|^2 \, dxdy - \int_{\mathbb{C}} T \Delta w \frac{\partial w}{\partial n} \, ds \quad (6.30)$$

Boundary conditions, both homogeneous and non-homogeneous will now be considered.

**Homogeneous boundary conditions** Let the boundary conditions be homogeneous Dirichlet on a part $C_1$ of $C$, then $w = 0$ on $C_1$, so that $w$ is fixed and $\Delta w = 0$ consequently.

Let the boundary conditions be homogeneous Neumann on another part $C_2$ of $C$, then $\frac{\partial w}{\partial \phi} = 0$ on $C_2$.

It follows that for either case, the line integral (the integral along the boundary) which is the second term in equation 6.30 vanishes, and the work done is given by

$$\Delta \int \int_{\mathbb{R}} pw \, dxdy = \Delta \int \int_{\mathbb{R}} \frac{1}{2} T |\nabla w|^2 \, dxdy$$

so that

$$\Delta \int \int_{\mathbb{R}} \left(|\nabla w|^2 - \frac{2p}{T} w \right) \, dxdy = 0$$

Let $I[w]$ be the functional given by

$$I[w] = \int \int_{\mathbb{R}} \left(|\nabla w|^2 - \frac{2p}{T} w \right) \, dxdy \quad (6.31)$$

then the solution, $w_0$, of equation 6.29 subject to homogeneous Dirichlet and Neumann boundary conditions is such that $\Delta I = 0$. 

154
In other words, at equilibrium, the potential energy is stationary, since \( \frac{1}{2} T |\nabla w|^2 \) is the energy per unit area of the membrane, and \( pw \) is the energy per unit area of the applied force. (Compare this with the equations for internal and external work in section 6.2.1 above.)

If the boundary conditions are homogeneous mixed on another part \( C_3 \) of \( C \), then

\[
\partial w/\partial n + \sigma w = 0 \quad \text{on} \quad C_3,
\]

and

\[
\int_k \left[ w \right] dxdy + \int_{C_3} \sigma w^2 ds = 0.
\]  

(6.32)

After some algebra, equation 6.32 can rewritten in terms of \( \nabla^2 \), to give

\[
I[w] = \int_{\mathcal{R}} \left\{ w (-\nabla^2) w - \frac{2p}{T} w \right\} dxdy
\]  

(6.33)

Equation 6.33 which is derived using \(-\nabla^2 w = p/T\) applies to all p.d.e.'s \( L\phi = f \) provided \( L \) is self-adjoint and positive definite, and provided the boundary conditions are homogeneous. Indeed:

the solution of \( L\phi = f \) where \( L \) is self-adjoint and positive definite, with homogeneous boundary conditions, occurs at a minimum value of \( I[\phi] \) where

\[
I[\phi] = \int_{\mathcal{R}} (\phi L\phi - 2\phi f) dxdy
\]  

(6.34)

Proof To prove this, let \( \phi_0 \) be the solution of \( L\phi = f \). Then

\[
I[\phi] = \int_{\mathcal{R}} (\phi L\phi - 2\phi L\phi_0) dxdy
\]

\[
= \int_{\mathcal{R}} \phi L(\phi - \phi_0) dxdy
\]

\[
- \int_{\mathcal{R}} (\phi - \phi_0) L\phi_0 dxdy - \int_{\mathcal{R}} \phi_0 L\phi_0 dxdy
\]

and using the facts that \( L \) is self-adjoint and the boundary conditions are homogeneous,

\[
I[\phi] = \int_{\mathcal{R}} (\phi - \phi_0) L(\phi - \phi_0) dxdy - \int_{\mathcal{R}} \phi_0 L\phi_0 dxdy
\]

155
For $L$ positive definite implies that $\int\int_{\mathcal{R}} \phi_0 L \phi_0 \, dx\,dy > 0$ since $\phi_0$ is not trivial, also that

$$\int\int_{\mathcal{R}} (\phi - \phi_0) L (\phi - \phi_0) \, dx\,dy \geq 0$$

with equality holding iff $\phi = \phi_0$. Thus $I[\phi]$ has its minimum value with $\phi = \phi_0$, the exact solution of $L\phi = f$. \hfill \Box

Non-homogeneous boundary conditions Let the boundary conditions, equations 6.23–6.25, be non-homogeneous. These conditions can be written in the form

$$B\phi = b(s) \quad (6.35)$$

where $B$ is a linear differential operator.

To find the solution $\phi_0$ of $L\phi = f$ subject to equation 6.35, let $v$ be a function which satisfies equation 6.35, so that $Bv = b$, and let $u$ be $u = \phi - v$.

Then, $Bu = B\phi - bv = 0$ by the linearity of $B$ and provided $\phi$ satisfies the boundary conditions.

Similarly, let $u_0 = \phi_0 - v$, then $Lu_0 = f - Lv = F$, say.

It can now be seen that $u_0$ is the unique function which minimizes $I[u]$ defined by

$$I[u] = \int\int_{\mathcal{R}} (uLu - 2uF) \, dx\,dy$$

from equation 6.34 above. Then

$$I[\phi] = \int\int_{\mathcal{R}} (\phi L\phi - 2\phi f + \phi Lv - vL\phi) \, dx\,dy + \int\int_{\mathcal{R}} (2vf - vLv) \, dx\,dy$$

and since the second term is fixed as far as the minimization is concerned

$$I[\phi] = \int\int_{\mathcal{R}} (\phi L\phi - 2\phi f + \phi Lv - vL\phi) \, dx\,dy \quad (6.36)$$

which is minimized by $\phi_0$.  

156
Minimization of an Integral

To minimize the integrals of the previous section, a method known as the Rayleigh-Ritz method is used. The idea is to choose a set of linearly independent basis functions \( \psi_i(x, y) \), \( i = 1, 2, \ldots \). Then the exact solution \( \phi_0 \) of \( L\phi = f \) is approximated by

\[
\phi_n = \sum_{i=1}^{n} c_i \psi_i
\]  

with \( c_i \) chosen to minimize \( I[\phi_n] \) at each stage. The procedure is said to converge if \( \phi_n \to \phi_0 \) as \( n \to \infty \). In fact, provided that \( L \) is positive definite, the method converges.

The functions chosen must satisfy the essential boundary conditions. (The Dirichlet boundary conditions are essential, meaning that they must be enforced in the solution, whereas the Neumann and mixed boundary conditions are natural boundary conditions, meaning that a solution of \( L\phi = f \) will always satisfy them.)

Now, the functional of equation 6.36 above must be minimized. \( I[\phi] \) can be written in terms of the coefficients \( c_i \) of the basis functions \( \psi_i \) and the basis functions themselves, thus

\[
I[c_1, \ldots, c_n] = \iint_{\mathcal{R}} \left\{ \left( \sum_{i=1}^{n} c_i \psi_i \right) L \left( \sum_{i=1}^{n} c_i \psi_i \right) - 2 \left( \sum_{i=1}^{n} c_i \psi_i \right) f ight. \\
+ \left. \left( \sum_{i=1}^{n} c_i \psi_i \right) L u - v L \left( \sum_{i=1}^{n} c_i \psi_i \right) \right\} dx dy \\
= c_i^2 \iint_{\mathcal{R}} \psi_i L \psi_i \ dx dy + \sum_{j \neq i} \left\{ c_i c_j \iint_{\mathcal{R}} (\psi_i L \psi_i + \psi_j L \psi_j) \ dx dy \right\} \\
+ c_i \iint_{\mathcal{R}} (-2 \psi_i f + \psi_i L u - v L \psi_i) \ dx dy \\
+ \text{terms independent of } c_i, \text{ for } i = 1, \ldots, n
\]

and

\[
\frac{\partial I}{\partial c_i} = 2 c_i \iint_{\mathcal{R}} \psi_i L \psi_i \ dx dy + \sum_{j \neq i} \left\{ c_j \iint_{\mathcal{R}} \mathcal{R}(\psi_i L \psi_i + \psi_j L \psi_j) \ dx dy \right\}
\]
\[ + \int_{\mathcal{R}} (-2\psi_if + \psi_i Lv - vL\psi_i) \, dx \, dy, \text{ for } i = 1, \ldots, n \]

To make \( I \) stationary, \( \partial I / \partial c_i = 0 \), so we have
\[ \sum_{j=1}^n \{ c_j \int_{\mathcal{R}} (\psi_i L\psi_i + \psi_j L\psi_j) \, dx \, dy \} = \int_{\mathcal{R}} (2\psi_i f - \psi_i Lv + vL\psi_i) \, dx \, dy \]
for \( i = 1, \ldots, n \)

or
\[ \sum_{j=1}^n A_{ij} c_j = h_i, \text{ for } i = 1, \ldots, n \]  \( (6.38) \)

where
\[ A_{ij} = \frac{1}{2} \int_{\mathcal{R}} (\psi_i L\psi_i + \psi_j L\psi_j) \, dx \, dy \]  \( (6.39) \)

and
\[ h_i = \frac{1}{2} \int_{\mathcal{R}} \left\{ \psi_i f + \frac{1}{2} (vL\psi_i - \psi_i Lv) \right\} \, dx \, dy \]  \( (6.40) \)

or, in matrix notation:
\[ A \cdot c = h \]  \( (6.41) \)

from which values of \( c_i \) can be worked out since \( A \) is non-singular provided that \( L \) is positive definite. This is true provided that the basis functions chosen satisfy the essential boundary conditions.

### 6.2.3 From Variational Methods to Finite Elements

Several difficulties arise in trying to apply the methods of the previous section to field problems:

- It is not generally possible to find a series of basis functions that satisfy the essential boundary conditions. Even when such functions can be found, they are likely to be polynomials of a very high degree.

- No continuous function, as all polynomials are, can model discontinuous regions and boundaries.
- It is not possible to concentrate on one part of the region which might be of more interest than the rest.

- The matrix equation that results from variational methods will have a dense matrix of coefficients which will be costly computationally. The reason for the denseness is that distant points in the region are coupled.

To overcome these difficulties the finite element method is used. Just as in structural analysis, the structure is considered to be a set of elements connected with each other at the nodes, and each element is considered individually before the overall system is assembled, so in the finite element method the region over which the problem is defined is divided into a set of finite elements connected at nodes. It has been seen that the virtual displacement principle used for structural analysis is equivalent to the principle of minimum potential energy, and that the latter leads to a functional to be minimized. In the finite element method, the Rayleigh-Ritz method is applied piecewise to the elements. In this way it is possible to approximate any boundary by the arcs of a polygon (in two dimensions). When choosing an approximate function for an element, it is assumed that the value of this function is zero everywhere outside the element so that discontinuities can be handled. Further, if it is desired to concentrate on a particular part of the region, this can be done by dividing the part into many small elements. Finally, since the elements are considered to be connected only at the nodes, then nodes not actually connected to each other are uncoupled, and this leads to a sparse matrix of coefficients in the final equations.

An example in one dimension Consider a two point boundary value problem

\[-\frac{d^2 \phi}{dx^2} = 2 \text{ in the "region" } 0 < x < 1\]  

(6.42)
Consider a 2 element discretization with 3 nodes as shown in figure 6.3. There are two nodal variables with each element, so a linear interpolating polynomial is chosen for each element.

Figure 6.4 shows an element $e$ of length $h$ and midpoint $x_m$. Use a local coordinate $\zeta$ where

$$\zeta = \frac{2}{h} (x - x_m)$$

and the shape function matrix $N^e(\zeta)$, which relates the value of $\phi$ within the element to its values at the nodes will be just

$$N^e(\zeta) = \begin{pmatrix} N_1^e(\zeta) & N_2^e(\zeta) \end{pmatrix}$$

where

$$N_1^e(\zeta) = \frac{1}{2} (1 - \zeta)$$

and

$$\phi(0) = 0 \text{ at } x = 0$$

(6.43) and

$$\frac{d\phi}{dx} = 0 \text{ at } x = 1$$

(6.44)
and
\[ N_\theta^e(\zeta) = \frac{1}{2}(1 + \zeta) \] (6.45)

so that
\[ \phi^e = N^e u^e \] (6.46)

where
\[ u^e = \begin{pmatrix} \phi_A \\ \phi_B \end{pmatrix} \] (6.47)

The solution is obtained by summing up the approximations over all the elements
\[ \Phi = \sum_{e=1}^{2} \phi^e \] (6.48)

Now, the variational formulation of equation 6.42 is obtained using equation 6.33 above as
\[ \text{minimize } I[\phi] = \int_0^1 \left\{ \left( \frac{d\phi}{dx} \right)^2 - 2\phi.2 \right\} dx \] (6.49)

From equations 6.48 and 6.49
\[ I[\phi] = \int_0^1 \left\{ \left( \sum_e \frac{d\phi^e}{dx} \right)^2 - s \sum_e \phi^e.2 \right\} dx \]

Now, in element 1, \( \phi^{(2)} = 0 \) and in element 2, \( \phi^{(1)} = 0 \) (this is a basic assumption in the finite element method, that the value of the approximate function for an element \( e \) is zero everywhere outside the element), so that
\[ I[\phi] = \int_0^{1/2} \left\{ \left( \frac{d\phi^{(1)}}{dx} \right)^2 - 2\phi^{(1)}.2 \right\} dx + \int_{1/2}^1 \left\{ \left( \frac{d\phi^{(2)}}{dx} \right)^2 - 2\phi^{(2)}.2 \right\} dx \]
or
\[ I[\phi] = \sum_e \int_e \left\{ \left( \frac{d\phi^e}{dx} \right)^2 - 2\phi^e.2 \right\} dx \] (6.50)

To obtain the nodal variables
\[ \frac{\partial I}{\partial \phi_i} = 0, \quad i = 1, 2, 3 \]
or
\[ \sum_e \int_e \left\{ 2 \frac{d\phi^e}{dx} \frac{\partial}{\partial \phi_i} \left( \frac{d\phi^e}{dx} \right) - 2 \frac{\partial \phi^e}{\partial \phi_i} \right\} \, \frac{dx}{dx} = 0, \text{ for } i = 1, 2, 3 \]

Now,
\[ \frac{\partial}{\partial \phi_i} \left( \frac{d\phi^e}{dx} \right) = \frac{d}{dx} \left( \frac{\partial \phi^e}{\partial \phi_i} \right) = \frac{dN_i^e}{dx} \]

so that to obtain the nodal variables, the following are solved
\[ \sum_e \int_e \left( \frac{d\phi^e}{dx} \frac{dN_i^e}{dx} - 2N_i^e \right) \, dx = 0, \text{ for } i = 1, 2, 3 \]

for \( i = 1 \):
\[ \int_0^{1/2} \left\{ \left( \frac{dN_A}{dx} \phi_1 \right) \frac{dN_A}{dx} + \left( \frac{dN_B}{dx} \phi_2 \right) \frac{dN_A}{dx} - 2N_A \right\} = 0 \]

for \( i = 2 \):
\[ \int_0^{1/2} \left( \left( \frac{dN_A}{dx} \phi_1 \right) \frac{dN_A}{dx} + \left( \frac{dN_B}{dx} \phi_2 \right) \frac{dN_B}{dx} - 2N_B \right) \]
\[ + \int_{1/2}^1 \left( \left( \frac{dN_A}{dx} \phi_1 \right) \frac{dN_A}{dx} + \left( \frac{dN_B}{dx} \phi_3 \right) \frac{dN_A}{dx} - 2N_A \right) = 0 \]

and for \( i = 3 \):
\[ \int_{1/2}^1 \left( \left( \frac{dN_A}{dx} \phi_2 \right) \frac{dN_B}{dx} + \left( \frac{dN_B}{dx} \phi_3 \right) \frac{dN_B}{dx} - 2N_B \right) = 0 \]

These three equations can be written as
\[
\begin{align*}
  k_{11}^{(1)} \phi_1 & + k_{12}^{(1)} \phi_2 & = f_1^{(1)} \\
  k_{21}^{(1)} \phi_1 & + (k_{22}^{(1)} + k_{11}^{(1)}) \phi_2 & + k_{12}^{(1)} \phi_3 & = f_2^{(1)} + f_1^{(2)} \\
  k_{21}^{(2)} \phi_2 & + k_{22}^{(2)} \phi_3 & = f_2^{(2)}
\end{align*}
\]

or
\[ K\phi = F \] \tag{6.52}

\( K \) and \( F \) are obtained from the element stiffness matrices \( k^e \) and the element forces vectors \( f^e \) where
\[ k_{AB}^e = \int_e \frac{dN_A}{dx} \frac{dN_B}{dx} \, dx, \text{ for } A, B = 1, 2 \text{ and } e = 1, 2 \] \tag{6.53}
and
\[ f_A^e = \int_e 2N_A \, dx, \text{ for } A = 1, 2 \text{ and } e = 1, 2 \quad (6.54) \]

Notice the similarity with equation 6.9 earlier which was obtained using structural analysis. Here again, \( K \) is symmetric and \( K_{13} = K_{31} = 0 \) showing that there is no coupling between nodes not on the same element. Further, \( K \) is singular with the singularity caused by the fact that the essential Dirichlet boundary condition has not yet been enforced.

Using local coordinates, equation 6.53 leads to
\[ k_{AB} = \int_{-1}^{1} \frac{2 \, dN_A}{h} \frac{2 \, dN_B}{h} \frac{h}{2} \, d\zeta \text{ for } A, B = 1, 2 \]
so that from equation 6.45
\[ k_{11} = k_{22} = 1/h \]
and
\[ k_{21} = k_{12} = -1/h \]
Further,
\[ f_A = \int_{-1}^{1} 2N_A \frac{h}{2} \, d\zeta, \text{ for } A = 1, 2 \]
so that \( f_1 = f_2 = h \).

The element stiffness matrices and the force vectors can now be written
\[
k^{(1)} = \frac{1}{h} \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}
\]
\[
k^{(2)} = \frac{1}{h} \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}
\]
\[
f^{(1)} = h \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}
\]
\[
f^{(2)} = h \begin{pmatrix} 1 \\ 1 \end{pmatrix}
\]
The overall equations can now be assembled with \( h = 1/2 \) as
\[
\frac{1}{1/2} \begin{pmatrix} 1 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} \phi_1 \\ \phi_2 \\ \phi_3 \end{pmatrix} = \frac{1}{2} \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix}
\]

The singularity of the matrix of coefficients is removed by enforcing the essential Dirichlet boundary condition \( \phi(0) = 0 \), i.e. by setting \( \phi_1 \) to 0, to leave
\[
4\phi_2 - 2\phi_3 = 1
\]
\[
-2\phi_2 + 2\phi_3 = 1/2
\]
whence \( \phi_2 = 3/4 \) and \( \phi_3 = 1 \).

Now,
\[
\phi^{(1)}(x) = \frac{1}{2} \begin{pmatrix} 1 - \zeta & 1 + \zeta \\ 3/4 \end{pmatrix} = \frac{3x}{2}
\]
since, in element (1), \( \zeta = 2(x - 1/4)/2 \), and
\[
\phi^{(2)}(x) = \frac{1}{2} \begin{pmatrix} 1 - \zeta & 1 + \zeta \\ 3/4 \end{pmatrix} = \frac{x + 1}{2}
\]
since, in element (2), \( \zeta = 2(x - 3/4)/2 \), so that the approximate solution of equation 6.42 subject to equations 6.43 and 6.44 is
\[
\Phi(x) = \begin{cases} 
\frac{3x}{2}, & 0 \leq x \leq 1/2 \\
\frac{x+1}{2}, & 1/2 \leq x \leq 1 
\end{cases}
\]

### 6.2.4 Internal and Shared Nodes

The example of the previous section was merely to show how the finite element method worked in principle. (In practice the example above can be worked out analytically.) The approximate functions chosen, i.e. the linear interpolating polynomials, were, following the simplicity of the problem, simple. If the elements are such that they contain internal nodes, then higher
order polynomials are needed. Since these internal nodes are not coupled with any other nodes from another element, it turns out that it is possible to eliminate these nodes independently of the other nodes.

For an element with stiffness matrix \( k^e \), force vector \( f^e \) and displacement vector \( u^e \), partition \( u^e \) into a vector for the displacement of internal nodes \( u^e_I \) and another for the shared nodes \( u^e_S \), or

\[
\begin{bmatrix}
  u^e_I \\
  u^e_S 
\end{bmatrix}
\]

Similarly,

\[
k^e = \begin{bmatrix}
  k^e_I & k^e_{IS} \\
  k^e_{SI} & k^e_S
\end{bmatrix}
\]

and

\[
f^e = \begin{bmatrix}
  f^e_I \\
  f^e_S
\end{bmatrix}
\]

so that from the equilibrium equation of element \( e \), \( k^e u^e = f^e \), we get

\[
k^e_I u^e_I + k^e_{IS} u^e_S = f^e_I
\]

In a similar way, the equilibrium equations for the overall system, \( KU = F \), can be partitioned by partitioning \( U \) into \( U_I \) and \( U_S \), the internal and shared nodes of all the elements.

\[
U = \begin{bmatrix}
  U_I \\
  U_S
\end{bmatrix}
\]

where \( U_I \) is simply the vector \( \{ u^{(1)}_I, u^{(2)}_I, \ldots \} \) made up of all the vectors of nodal displacements of all the elements. \( U_S \) and \( U_S^e \) can be related by a Boolean selection matrix such those used earlier in section 6.2.1, so that

\[
u^e_S = L^e U_S
\]
In a similar way $F$ is partitioned so that $F = \{F_I, F_S\}$ where $F_I$ is just $\{f_I^{(1)}, f_I^{(2)}, \ldots\}$ and $F_S$ is given by
\[ F_S = \sum_c L^c f_S^c \quad (6.59) \]

Now, the overall equilibrium equation, $KU = F$, assuming that there are $N$ elements, can be written as
\[
\begin{pmatrix}
  k_I^{(1)} & k_{IS}^{(1)} & \cdots & k_{IS}^{(N)} \\
  k_I^{(2)} & k_{IS}^{(2)} & \cdots & k_{IS}^{(N)} \\
  \vdots & \vdots & \ddots & \vdots \\
  L(I)^T k_I^{(1)} & L(IS)^T k_{IS}^{(1)} & \cdots & L(IS)^T k_{IS}^{(N)}
\end{pmatrix}
\begin{pmatrix}
  u_I^{(1)} \\
  u_I^{(2)} \\
  \vdots \\
  U_S
\end{pmatrix}
= \begin{pmatrix}
  f_I^{(1)} \\
  f_I^{(2)} \\
  \vdots \\
  F_S
\end{pmatrix}
\quad (6.60)
\]

and this is how equation 6.2 is obtained.

### 6.3 Two Methods of Solving Linear Equations

As mentioned in the introduction to this chapter the system of linear equations 6.2 will be solved using two direct methods (as opposed to iterative methods): Gaussian elimination and Choleski decomposition or factorization. These two methods are very widely known (see [68, 45] for example) and only a brief description of them will be given below.
6.3.1 Gaussian Elimination

Consider the system of linear equations 6.61 below.

\[
\begin{pmatrix}
    a_{11} & a_{12} & \cdots & a_{1n} \\
    a_{21} & a_{22} & \cdots & a_{2n} \\
    \vdots & \vdots & \ddots & \vdots \\
    a_{n1} & a_{n2} & \cdots & a_{nn}
\end{pmatrix}
\begin{pmatrix}
    x_1 \\
    x_2 \\
    \vdots \\
    x_n
\end{pmatrix}
= 
\begin{pmatrix}
    b_1 \\
    b_2 \\
    \vdots \\
    b_n
\end{pmatrix}
\tag{6.61}
\]

The solution of the unknown variables \(x_i, i = 1, \ldots, n\), by Gaussian elimination proceeds in two steps:

Step (1): Elimination Let \(r_i = \begin{pmatrix} a_{i1} & a_{i2} & \cdots & a_{in} \end{pmatrix}\) be row \(i\) of the coefficient matrix. Then for \(i = 2, 3, \ldots, n\), replace \(r_i\) by \(r'_i\) and \(b_i\) by \(b'_i\), where

\[
\begin{align*}
    r'_i &= r_i - (a_{i1}/a_{11})r_1 \\
    b'_i &= b_i - (a_{i1}/a_{11})b_1
\end{align*}
\]

for \(i = 2, 3, \ldots, n\)

The result of this step is to transform equations 6.61 into equations 6.62 below.

\[
\begin{pmatrix}
    a_{11} & a_{12} & \cdots & a_{1n} \\
    0 & a'_{22} & \cdots & a'_{2n} \\
    \vdots & \vdots & \ddots & \vdots \\
    0 & a'_{nn} & \cdots & a'_{nn}
\end{pmatrix}
\begin{pmatrix}
    x_1 \\
    x_2 \\
    \vdots \\
    x_n
\end{pmatrix}
= 
\begin{pmatrix}
    b_1 \\
    b'_2 \\
    \vdots \\
    b'_n
\end{pmatrix}
\tag{6.62}
\]

Following this, \(r'_i\), for \(i = 3, \ldots, n\), is replaced by \(r''_i\) and \(b'_i\) by \(b''_i\) where

\[
\begin{align*}
    r''_i &= r'_i - (a'_{i2}/a'_{22})r'_2 \\
    b''_i &= b'_i - (a'_{i2}/a'_{22})b'_2
\end{align*}
\]

for \(i = 3, 4, \ldots, n\)

This is repeated until the system of equations is transformed into the set.
of equations 6.63

\[
\begin{pmatrix}
a_{11} & a_{12} & \cdots & a_{1n} \\
0 & a'_{22} & \cdots & a'_{2n} \\
\vdots & \vdots & \ddots & \vdots \\
0 & 0 & \cdots & a_{nn}^{(n-2)}
\end{pmatrix}
\begin{pmatrix}
x_1 \\
x_2 \\
\vdots \\
x_n
\end{pmatrix}
= 
\begin{pmatrix}
b_1 \\
b_2 \\
\vdots \\
b_n^{(n-2)}
\end{pmatrix}
\tag{6.63}
\]

This procedure can be summarised by the following algorithm

(1) for \( i = 1 \) to \((n - 1)\)
(2) for \( j = (i + 1) \) to \( n \)
(3) \( m = a_{ij}/a_{ii} \)
(4) for \( k = (j + 1) \) to \( n \)
(5) \( a_{jk} = a_{jk} - m \times a_{ik} \)
(6) endfor
(7) \( b_j = b_j - m \times b_i \)
(8) endfor
(9) endfor

Algorithm 1: Elimination in Gaussian Elimination

It should be noted that it is not always possible to carry out the above procedure even when the matrix of coefficients is not singular. The reason is that division by \( a_{ii} \) is not always possible because it might be very small leading to unacceptable rounding errors, or, worse still, it might be zero. To overcome this problem a technique called pivoting must be used. For the purpose of solving equations 6.2, however, this situation does not arise and so the algorithm above will always work.

Step 2: Back-Substitution  After transforming the equations to the form given in equation 6.63 where the matrix of coefficients is an upper triangular matrix, the unknown values can now be found, one by one, from \( x_n \) down to \( x_1 \) by back-substitution:
Algorithm 2: Back-Substitution in Gaussian Elimination

notice that for the first loop of the iteration, when \( i = n \), the summation is taken over \( (n+1) \) to \( n \) which is taken to be zero.

Gaussian Elimination for Block Matrices

Gaussian elimination can be readily applied to systems of linear equations such as equation 6.2 where the matrix of coefficients is a block diagonal bordered matrix. Consider equation 6.64

\[
\begin{pmatrix}
A_1 & B_1 \\
A_2 & B_2 \\
\vdots & \vdots \\
A_N & B_N \\
C_1 & C_2 & \cdots & C_N & A_{N+1}
\end{pmatrix}
\begin{pmatrix}
x_1 \\
x_2 \\
\vdots \\
x_N \\
x_{N+1}
\end{pmatrix}
= 
\begin{pmatrix}
d_1 \\
d_2 \\
\vdots \\
d_N \\
d_{N+1}
\end{pmatrix}
\]

(6.64)

where the matrix of coefficients is symmetric so that \( C_i = B_i^T \) for \( i = 1, 2, \ldots, N \) and each of the blocks \( A_i, i = 1, 2, \ldots, N, N+1 \) is also symmetric. Further, since such a matrix results from the finite element method, it is also assumed that it is positive definite. The other properties of this system of equations have been fully described in the introduction.

Step 1: Elimination  Whereas for a point system elimination involves dividing each row by the value of each diagonal element, in the block system the analogous operation is multiplication by the inverse of each diagonal block. The following algorithm describes the elimination process for equation 6.64.
Algorithm 3: Elimination Step of the Block Gaussian Elimination

This transforms the system into that given by equation 6.65

\[
\begin{pmatrix}
I & B'_1 \\
I & B'_2 \\
& \vdots \\
I & B'_N \\
0 & 0 & \ldots & 0 & B'_{N+1}
\end{pmatrix}
\begin{pmatrix}
x_1 \\
x_2 \\
\vdots \\
x_N \\
x_{N+1}
\end{pmatrix}
= 
\begin{pmatrix}
d'_1 \\
d'_2 \\
\vdots \\
d'_N \\
d'_{N+1}
\end{pmatrix}
\tag{6.65}
\]

Step 2: Back-Substitution

This is done by

Algorithm 4: Back-Substitution Step of the Block Gaussian Elimination

Concurrent Gaussian Elimination

Step 1 of Gaussian elimination for the block system, algorithm 3, consists of a single loop. The first two statements in the body of the loop, statements (2) and (3), are each (data) independent for each iteration. Statements (4) and (5) involve assignment to the same variables, \(A_{N+1}\) and \(d_{N+1}\), in all iterations, and so, although the order of the execution of each iteration is...
not important, only one iteration can be executed at any one time. This problem can be partially overcome by using the extra variables shown in algorithm 5.

(1) for $i = 1$ to $N$
(2) $D_i = A_i^{-1}$
(3) $B_i = D_i B_i$
(4) $d_i = D_i d_i$
(5) $D_i = C_i B_i$
(6) $e_i = C_i d_i$
(7) endfor
(8) for $i = 1$ to $N$
(9) $A_{N+1} = A_{N+1} - D_i$
(10) $d_{N+1} = d_{N+1} - e_i$
(11) endfor

Algorithm 5: Block Gaussian Elimination

The iterations of the first loop are now independent of each other so that they can all be executed concurrently. The iterations of the second loop, on the other hand, have to be executed one at a time, though the order of execution is not important. Note however that the second loop involves subtractions only so that the effect of it being sequential will not significantly affect the speedup.

Step 2 of the process, algorithm 4, is made up of a matrix inversion and multiplication, step (1), followed by a loop. The loop requires the result of step (1), and so cannot start until step (1) is finished. The iterations of the loop are independent of each other and can therefore all be executed concurrently. Algorithm 6 shows the parallel Gaussian elimination process as discussed so far.
Matrix inversion and multiplication are well-known and extensively studied operations for parallel execution [3, 70]. It is possible to go further with the parallelization of Gaussian elimination and perform steps (2), (3), (5), and (12) of algorithm 6 in parallel. This, however, is unnecessary and indeed counterproductive since it is already possible to utilize all the available processors, so that further parallelization would require more operations which would have to be performed by the same processors. The situation would be different if vectorization were available, but since it is not, no more will be said about it.

The only exception to the above is step (12), which is a costly step in terms of the number of operations it involves, and which, according to algorithm 6, is to be executed by one processor with all the other processors idle.

In the following the number of operations considered is the number of multiplication (or division) operations only, i.e. addition and subtraction operations are ignored. Let $\alpha_i$ be the number of operations required to invert
the matrix $A_i$, step (2); $\beta_i$ be the number of operations required to multiply $D_i$ by $B_i$, step (3); $\gamma_i$ be the number of operations required for step (4); $\delta_i$ be the number of operations required for step (5); and $\epsilon_i$ be the number of operations required for step (6). For example, in equation 6.3, $\alpha_1$ is the number of operations required to invert a $(2 \times 2)$ matrix, $\beta_1$ is the number of operations required to multiply a $(2 \times 2)$ matrix by a $(2 \times 8)$ matrix, $\gamma_1$ is the number of operations required to multiply a $(2 \times 8)$ matrix by a two element vector, $\delta_1$ is the number of operations required to multiply a $(2 \times 8)$ matrix by a $(2 \times 8)$ matrix, and $\epsilon_1$ is the number of operations required to multiply an $(8 \times 2)$ matrix by a two element vector.

The second loop of the algorithm, steps (8) to (11), is made up entirely of subtraction operations only. Step (12) requires $a_{N+1} + b_{N+1}$ operations. Finally, let step (14) require $\eta_i$ operations.

The time taken for the execution of the algorithm is, of course, proportional to the number of operations that make up the algorithm. Therefore, let the time be represented by the number of operations, and the total sequential time for algorithm 6 is $T_S$ which is given by

$$T_S = \sum_{i=1}^{N} (\alpha_i + \beta_i + \gamma_i + \delta_i + \epsilon_i) + \alpha_{N+1} + \gamma_{N+1} + \sum_{i=1}^{N} \eta_i$$

(6.66)

Now, $A_i$, $i = 1, \ldots, N$, is a square matrix of dimension $n_i \times n_i$, and $A_{N+1}$ is a square matrix of dimension $m \times m$. $B_i$, $i = 1, \ldots, N$, is a matrix of dimension $n_i \times m$, and $C_i$, $i = 1, \ldots, N$, is a matrix of dimension $m \times n_i$. $d_i$ and $x_i$, $i = 1, \ldots, N$, are vectors of $n_i$ elements each, and $d_{N+1}$ and $x_{N+1}$ are vectors with $m$ elements each. It follows that $\alpha_i$ is $O(n_i^3)$, $\beta_i$ is $O(n_i^2 m)$, $\gamma_i$ is $O(n_i^2)$, $\delta_i$ is $O(n_i^2 m)$, $\epsilon_i$ is $O(n_i m)$, and $\eta_i$ is $O(n_i m)$, for $i = 1, \ldots, N$; $\alpha_{N+1}$ is $O(m^3)$, and $\gamma_{N+1}$ is $O(m^2)$.

In order to facilitate the analysis, assume that $n_i = n$ for $i = 1, 2, \ldots, N$, and $m = kn$ for some positive number $k$. Then $T_S$ becomes

$$T_S = Nn^2\{(2k+1)n+k+1\} + n^2k^2(kn+1) + Nn^2k$$

(6.67)

173
Let \( p \) be the number of available processors, and let \( N = pq + r \), \( r < q \), i.e. \( q \) is the quotient of \( N/p \) and \( r \) is the remainder. Then the first loop of algorithm 6 can now be executed in \( \lceil N/p \rceil \) (where \( \lceil x \rceil \) is the ceiling of \( x \)) the time for one iteration. Similarly, the final loop, steps (13) to (15), can be executed in \( \lceil N/p \rceil \) the time for one iteration. Thus, with \( p \) processors, the time taken for parallel execution, \( T_p \), is given by
\[
T_p = tn^2\{(2k+1)n + k + 1\} + n^2k^2(k + 1) + tn^2k
\] (6.68)
where \( t = \lceil N/p \rceil \).

If step (12) is also executed in parallel, then the fastest time in which algorithm 6 can be executed, \( T'_p \), is
\[
T'_p = tn^2\{(2k+1)n + k + 1\} + n^2k^2(k + 1)/p + tn^2k
\] (6.69)

The improvement in speedup as a result of parallelizing step (12), \( I \), is given by
\[
I = \frac{(S'_p - S_p)}{S_p}
\] (6.70)
where \( S_p = T_s/T_p \) and \( S'_p = T_s/T'_p \). Thus
\[
I = \frac{T_p - T'_p}{T'_p} = \frac{n^2k^2(k + 1)(1 - 1/p)}{tn^2(2k+1)(n + 1) + (1/p)n^2k^2(k + 1)}
\]
or
\[
I = \frac{k^2(k + 1)(p - 1)}{pt(2k+1)(n + 1) + k^2(k + 1)}
\] (6.71)

Tables 6.1, 6.2, and 6.3 show values of \( S_p \), \( S'_p \) and \( I \) for several values of \( N, k, \) and \( n \), for \( p = 2, 5, \) and 9.

The improvement in speedup resulting from the parallelization of step (12) of algorithm 6 is not significant for large values of \( N \). Furthermore, equation 6.71 and tables 6.1, 6.2, and 6.3 are obtained by ignoring the time
### Table 6.1: Speedup with \( p = 2 \)

<table>
<thead>
<tr>
<th>( N )</th>
<th>( k )</th>
<th>( n )</th>
<th>( S_p )</th>
<th>( S'_p )</th>
<th>( I(%) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>3</td>
<td>20</td>
<td>1.6</td>
<td>1.7</td>
<td>3.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>50</td>
<td>1.6</td>
<td>1.7</td>
<td>1.6</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>1.6</td>
<td>1.7</td>
<td>9.7</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>50</td>
<td>1.6</td>
<td>1.7</td>
<td>4.3</td>
</tr>
<tr>
<td>25</td>
<td>3</td>
<td>20</td>
<td>1.9</td>
<td>1.9</td>
<td>0.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>50</td>
<td>1.9</td>
<td>1.9</td>
<td>0.4</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>1.9</td>
<td>1.9</td>
<td>2.4</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>50</td>
<td>1.9</td>
<td>1.9</td>
<td>1.0</td>
</tr>
<tr>
<td>100</td>
<td>3</td>
<td>20</td>
<td>2.0</td>
<td>2.0</td>
<td>0.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>50</td>
<td>2.0</td>
<td>2.0</td>
<td>0.1</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>2.0</td>
<td>2.0</td>
<td>0.7</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>50</td>
<td>2.0</td>
<td>2.0</td>
<td>0.3</td>
</tr>
</tbody>
</table>

### Table 6.2: Speedup with \( p = 5 \)

<table>
<thead>
<tr>
<th>( N )</th>
<th>( k )</th>
<th>( n )</th>
<th>( S_p )</th>
<th>( S'_p )</th>
<th>( I(%) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>3</td>
<td>20</td>
<td>4.2</td>
<td>5.0</td>
<td>18.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>50</td>
<td>4.6</td>
<td>5.0</td>
<td>8.0</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>3.4</td>
<td>5.0</td>
<td>46.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>50</td>
<td>4.2</td>
<td>5.0</td>
<td>20.3</td>
</tr>
<tr>
<td>25</td>
<td>3</td>
<td>20</td>
<td>4.8</td>
<td>5.0</td>
<td>3.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>50</td>
<td>4.9</td>
<td>5.0</td>
<td>1.6</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>4.5</td>
<td>5.0</td>
<td>10.1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>50</td>
<td>4.8</td>
<td>5.0</td>
<td>4.2</td>
</tr>
<tr>
<td>100</td>
<td>3</td>
<td>20</td>
<td>5.0</td>
<td>5.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>50</td>
<td>5.0</td>
<td>5.0</td>
<td>0.4</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>4.9</td>
<td>5.0</td>
<td>2.6</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>50</td>
<td>4.9</td>
<td>5.0</td>
<td>1.1</td>
</tr>
</tbody>
</table>

175
Table 6.3: Speedup with $p = 9$

taken for addition and subtraction and also by assuming that the parallelization of step (12) achieves perfect speedup for this step. In addition, matrix inversion and multiplication have been widely studied (see [3], for example). For these reasons, step (12) of algorithm 6 was not parallelized, and Gaussian elimination was implemented using algorithm 6. In chapter 8 a new concept called partial dependence will be defined and used to achieve a better speedup without resorting to the fine grain parallelism needed to parallelize step (12).

### 6.3.2 Choleski Factorization

Given a set of linear equations

$$A \cdot x = b$$ (6.72)

where the matrix of coefficients, $A$ is symmetric, the Choleski factorization of $A$ is

$$A = L \cdot L^T$$ (6.73)

where $L$ is a lower triangular matrix and $L^T$, the transpose of $L$, is upper triangular.

<table>
<thead>
<tr>
<th>$N$</th>
<th>$k$</th>
<th>$n$</th>
<th>$S_P$</th>
<th>$S'_P$</th>
<th>$I(%)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>3</td>
<td>20</td>
<td>4.2</td>
<td>5.1</td>
<td>21.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>50</td>
<td>4.6</td>
<td>5.0</td>
<td>8.9</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>3.4</td>
<td>5.3</td>
<td>54.3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>4.2</td>
<td>5.1</td>
<td>23.2</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>3</td>
<td>20</td>
<td>7.8</td>
<td>8.3</td>
<td>7.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>50</td>
<td>8.1</td>
<td>8.3</td>
<td>3.0</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>20</td>
<td>7.0</td>
<td>8.4</td>
<td>18.9</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>7.7</td>
<td>8.3</td>
<td>7.9</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>3</td>
<td>20</td>
<td>8.2</td>
<td>8.3</td>
<td>1.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>50</td>
<td>8.3</td>
<td>8.3</td>
<td>0.7</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>20</td>
<td>8.0</td>
<td>8.3</td>
<td>4.8</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>8.2</td>
<td>8.3</td>
<td>2.0</td>
<td></td>
</tr>
</tbody>
</table>
From equations 6.72 and 6.73

\[ L L^T x = b \]  \hspace{1cm} (6.74)

Let \( L^T x = y \), then

\[ L y = b \]  \hspace{1cm} (6.75)

and the vector \( y \) can be solved by forward substitution. Then \( x \) can be found using back substitution from \( L^T x = y \).

From equation 6.73

\[
\begin{pmatrix}
a_{11} & a_{12} & \cdots & a_{1n} \\
a_{21} & a_{22} & \cdots & a_{2n} \\
\vdots & \vdots & \ddots & \vdots \\
a_{n1} & a_{n2} & \cdots & a_{nn}
\end{pmatrix}
\begin{pmatrix}
l_{11} \\
l_{21} \\
\vdots \\
l_{n1}
\end{pmatrix}
= 
\begin{pmatrix}
l_{11} & l_{21} & \cdots & l_{n1} \\
l_{21} & l_{22} & \cdots & l_{n2} \\
\vdots & \vdots & \ddots & \vdots \\
l_{n1} & l_{n2} & \cdots & l_{nn}
\end{pmatrix}
\begin{pmatrix}
l_{11} \\
l_{21} \\
\vdots \\
l_{n1}
\end{pmatrix}
\]

or

\[
\begin{pmatrix}
a_{11} & a_{12} & \cdots & a_{1n} \\
a_{21} & a_{22} & \cdots & a_{2n} \\
\vdots & \vdots & \ddots & \vdots \\
a_{n1} & a_{n2} & \cdots & a_{nn}
\end{pmatrix}
\begin{pmatrix}
l_{11}^2 \\
l_{21}l_{11} \\
\vdots \\
l_{n1}l_{11}
\end{pmatrix}
= 
\begin{pmatrix}
l_{11}^2 & l_{11}l_{21} & \cdots & l_{11}l_{n1} \\
l_{21}l_{11} & l_{21}^2 + l_{22}^2 & \cdots & l_{21}l_{n1} + l_{22}l_{n2} \\
\vdots & \vdots & \ddots & \vdots \\
l_{n1}l_{11} & l_{n1}l_{21} + l_{n2}l_{22} & \cdots & l_{n1}^2 + l_{n2}^2 + \cdots + l_{nn}^2
\end{pmatrix}
\begin{pmatrix}
l_{11} \\
l_{21} \\
\vdots \\
l_{n1}
\end{pmatrix}
\]

so that \( L \) can be found using algorithm 7 below

\begin{align*}
(1) & \text{ for } i = 1 \text{ to } n \\
(2) & \quad l_{ii} = \sqrt{a_{ii} - \sum_{j=1}^{i-1} l_{ij}^2} \\
(3) & \quad \text{ for } k = i + 1 \text{ to } n \\
(4) & \quad l_{ki} = (a_{ki} - \sum_{j=1}^{i-1} l_{kj} l_{ij}) / l_{ii} \\
(5) & \text{ endfor} \\
(6) & \text{ endfor}
\end{align*}

Algorithm 7: Choleski Factorization

where it is assumed that \( \sum_{j=1}^{n} z = 0 \) if \( x > y \).

This procedure will now be applied to the block diagonal bordered matrix of equation 6.2.
Choleski Factorization for Block Matrices

It is possible to apply algorithm 7 directly to block matrices by replacing each element there by a block (see [23]). This, however, results in an algorithm which involves finding the square roots of matrices which are not easy to find. To get around this problem, an alternative form of Choleski factorization called the root-free Choleski factorization, is used. This involves decomposing a symmetric matrix $A$ into the product $LDL^T$ where $L$ is a lower triangular matrix the elements of whose leading diameter are all 1, $D$ is a diagonal matrix, and $L^T$ is the transpose of $L$, (see [23]).

There is another way of applying Choleski factorization to a block matrix and this is given here.

Given a symmetric block diagonal bordered matrix $A$, decompose it into a product of a lower and an upper triangular matrix as shown in equation 6.76 below.

\[
\begin{pmatrix}
A_1 & B_1^T \\
A_2 & B_2^T \\
\vdots & \vdots \\
A_N & B_N^T \\
B_1 & B_2 & \cdots & B_N & A_{N+1}
\end{pmatrix}
= 
\begin{pmatrix}
\Delta_1 & \\
\Delta_2 & \\
\vdots & \\
\Delta_N & \Delta_{N+1}
\end{pmatrix}
\begin{pmatrix}
\Gamma_1^T & \\
\Gamma_2^T & \\
\vdots & \\
\Gamma_N^T & \Delta_{N+1}^T
\end{pmatrix}
\]

(6.76)
where $\Delta_i, i = 1, \ldots, N+1$ is a lower triangular matrix. We now have

$$
\begin{pmatrix}
A_1 & B_1^T \\
A_2 & B_2^T \\
\vdots & \vdots \\
A_N & B_N^T \\
B_1 & B_2 & \cdots & B_N & A_{N+1}
\end{pmatrix}
= 
\begin{pmatrix}
\Delta_1 \Delta_1^T & \Delta_1 \Gamma_i^T \\
\Delta_2 \Delta_2^T & \Delta_2 \Gamma_i^T \\
\vdots & \vdots \\
\Delta_N \Delta_N^T & \Delta_N \Gamma_i^T \\
\Gamma_1 \Delta_1^T & \Gamma_2 \Delta_2^T & \cdots & \Gamma_N \Delta_N^T & \sum_{i=1}^{N-1} \Gamma_i \Gamma_i^T + \Delta_{N+1} \Delta_{N+1}^T
\end{pmatrix}
$$

from which it can be seen that $\Delta_i, i = 1, \ldots, N$, is just the Choleski factorization of $A_i, i = 1, \ldots, N$; and $\Gamma_i, i = 1, \ldots, N$, is just $B_i.(\Delta_i^T)^{-1}$. Note also that the inversion of $\Delta_i^T$ simply involves solving for each column of the unit matrix using back-substitution since $\Delta_i^T$ is an upper triangular matrix. Note further that the solution of equation 6.2, $A.x = b$, can proceed using forward and back-substitution processes only involving the inverses of the diagonal elements already worked out.

Algorithm 8 shows the procedure for performing the Choleski factorization on a block matrix.

---

Algorithm 8: Choleski Factorization for a Block Matrix

1. for $i = 1$ to $N$
2. \quad factorize $A_i$ into $\Delta_i \Delta_i^T$ using algorithm 7
3. \quad $\Gamma_i = B_i.(\Delta_i^T)^{-1}$
4. \quad $A_{N+1} = A_{N+1} - \Gamma_i \Gamma_i^T$
5. endfor
6. factorize $A_{N+1}$ into $\Delta_{N+1} \Delta_{N+1}^T$ using algorithm 7

---

179
Analysis of Choleski Factorization

Let $m$ denote a multiplication operation, and $q$ a square-root operation. Algorithm 7 consists of an outer loop, steps (1) to (6), which is repeated $n$ times, where $n$ is the order of the matrix whose Choleski factorization is being performed. In order to facilitate the analysis of the algorithm, all the loops are opened up as in algorithm 9 below.

```
(1) for $i = 1$ to $n$
(2)     for $j = 1$ to $i - 1$
(3)         $a_{ij} = a_{ii} - i_j^2$
(4)     endfor
(5)     $l_{ii} = \sqrt{a_{ii}}$
(6)     for $k = i + 1$ to $n$
(7)         for $j = 1$ to $i - 1$
(8)             $a_{kj} = a_{kj} - l_kj l_{ij}$
(9)         endfor
(10)      $l_{ki} = a_{ki}/l_{ii}$
(11)     endfor
(12) endfor
```

Algorithm 9: Choleski Factorization with all the loops opened up

Consider now the body of the main loop, step by step

Steps (2) to (4) This loop consists of just one multiplication, and repeated $i - 1$ times so the total number of operations is $(i - 1)m$.

Step (5) This is just a square root operation and the number of operations is $q$.

Steps (6) to (11) This loop consists of an inner loop, steps (7) to (9), and a multiplication, step (10). The body of the inner loop consists of one multiplication repeated $i - 1$ times. So the number of operations of loop (6) to (11) is given by $(n - i)(i - 1)m + (n - 1)m$, or $(n - i)m$. 

180
The total number of operations that make up iteration $i$ of the main loop is just the sum of the operations for the steps making up the body of the loop, or

$$(i - 1)m + q + (N - i)m$$

$$= \{-i^2 + i(n + 1) - 1\}m + q \quad (6.77)$$

and the total number of operations for the whole algorithm is the sum of equation 6.77 over all the iterations of the main loop, from $i = 1$ to $i = n$, or

$$
\sum_{i=1}^{n} \{-i^2 + i(n + 1) - 1\}m + q
$$

$$= -m \sum_{i=1}^{n} i^2 + (n + 1)m \sum_{i=1}^{n} i - m \sum_{i=1}^{n} 1 + q \sum_{i=1}^{n} 1
$$

$$= -m \sum_{i=1}^{n} \left\{2 \left( \begin{array}{c} i \\ 2 \end{array} \right) + \left( \begin{array}{c} i \\ 1 \end{array} \right) \right\} + (n + 1)m \sum_{i=1}^{n} i - m \sum_{i=1}^{n} 1 + q \sum_{i=1}^{n} 1
$$

$$= -m \left\{2 \left( \begin{array}{c} n + 1 \\ 3 \end{array} \right) + \left( \begin{array}{c} n + 1 \\ 2 \end{array} \right) \right\} + (n + 1)m \frac{1}{2} n(n + 1) - mn + qn
$$

$$= \frac{1}{6} n(n + 4)(n - 1)m + nq \quad (6.78)$$

Concurrent Block Choleski Factorization

Algorithm 8 gives the procedure for block Choleski factorization. It consists of a loop, steps (1) to (5), followed by (point) Choleski factorization of a block, step (6). The body of the loop, steps (2) to (4) is made up of three steps which must be performed sequentially since step (3) requires the result of step (2), and step (4) requires the result of step (3). Steps (2) and (3) together for any one iteration of the loop are independent of all other iterations, but step (4) of the loop involves assignment to the same variable in all the iterations, and so at any one time, only one process can execute step (4), although the order of execution is not important. In order to overcome this dependency locks can be used to ensure that only one process executes step (4) at any
one time, but this solution puts severe restrictions on the possible speedup. Another solution is to introduce another variable to hold the product \( \Gamma_i \Gamma_i^T \) thus allowing all the loop iterations to be executed concurrently, as shown in algorithm 10.

#### Algorithm 10: Modified Choleski Factorization for a Block Matrix

Let \( \alpha_i \) be the number of operations required for factorizing block \( A_i \), \( \beta_i \) be the number of operations required for inverting \( \Delta_i \), \( \gamma_i \) be the number of operations required for multiplying \( B_i \) by \( (\Delta_i^T)^{-1} \), and \( \delta_i \) be the number of operations required for multiplying \( \Gamma_i \) by \( \Gamma_i^T \).

\( \alpha_i \) is given by equation 6.78 above as \( n_i(n_i+4)(n_i-1)m/6 \) multiplications and \( n_i \) square root operations. For the sake of simplifying the analysis, assume that a multiplication operation takes as long as a square root operation, so that

\[
\alpha_i = \frac{1}{6}n_i(n_i^2 + 3n_i - 4) \quad (6.79)
\]

\( \beta_i \) is the number of operations required for inverting a triangular matrix.

As mentioned earlier, this involves substitution (forward in the case of a lower triangular matrix, and backwards in the case of an upper triangular matrix) only. Let \( U \) be an upper triangular matrix of dimension \( n_U \), and let \( V \) be its inverse. Denote by \( \text{I}_{ci} \) the \( i \)'th column of the unit matrix of dimension \( n_U \).
Then $V$ is found by solving the equations

$$UV_{Ci} = I_{Ci} \quad \text{for } i = 1, 2, \ldots, n_U$$  \hfill (6.80)

Note that the inverse of an upper (a lower) triangular matrix is also an upper (a lower) triangular matrix. So that $V_{Ci}$ will have zeros as the value of all its elements below the leading diagonal.

The solution of equation 6.80 for $i = j$ is then produced by algorithm 11 below.

\begin{align*}
(1) & \quad v_{Cj_j} \\
(2) & \quad \text{for } k = j - 1 \text{ downto } 1 \\
(3) & \quad ac = 0.0 \\
(4) & \quad \text{for } l = k + 1 \text{ to } j \\
(5) & \quad ac = ac - u_{kl} \times v_{Cj_l} \\
(6) & \quad \text{endfor} \\
(7) & \quad v_{Cj_k} = ac/u_{kk} \\
(8) & \quad \text{endfor}
\end{align*}

Algorithm 11: Inverting an upper triangular matrix Matrix

Although this algorithm is still $O(n^3)$, it requires considerably less operations than the inversion of a full matrix. In fact

$$\beta_i = \frac{1}{12}n_i(n_i^2 + 3n_i + 8)$$  \hfill (6.81)

$\gamma_i$ is the number of operations required for multiplying an $m \times n_i$ matrix by an upper triangular matrix of dimension $n_i \times n_i$, and

$$\gamma_i = \frac{1}{2}mn_i(n_i + 1)$$  \hfill (6.82)

$\delta_i$ is the number of operations required for multiplying an $n_i \times n_i$ lower triangular matrix by an upper triangular matrix of the same dimension, and

$$\begin{align*}
\delta_i & = \sum_{c=1}^{n_i} \left\{ \sum_{j=1}^{c-1} j + c(n_i - c + 1) \right\} \\
& = \frac{1}{6}n_i(n_i + 1)(2n_i + 1) \quad \hfill (6.83)
\end{align*}$$
As before, assume that \( n_1 = n_2 = \cdots = n_N = n \), and \( m = kn \) for some positive number \( k \). Using the number of operations required as the basis for the time taken for the execution of the algorithm, the sequential time, \( T_s \), for algorithm 10 is obtained by adding \( \alpha_i, \beta_i, \gamma_i, \) and \( \delta_i \) over all the iterations, to \( \alpha_{N+1} \) for step (9), to get

\[
T_s = \frac{nN}{12} \left( n^2(7 + 6k) + n(15 + 6k) + 4 \right) + \frac{nk}{6} (n^2k^2 + 3nk - 3) \quad (6.84)
\]

As before (section 6.3.1), let \( p \) be the number of available processors, and let \( N = pq + r, r < q \). Then the first loop of algorithm 10 can be executed, using \( p \) processors, in \( \lfloor N/p \rfloor \) the time for one iteration, so that the parallel execution time, \( T_P \), is given by

\[
T_P = \frac{nt}{12} \left( n^2(7 + 6k) + n(15 + 6k) + 4 \right) + \frac{nk}{6} (n^2k^2 + 3nk - 3) \quad (6.85)
\]

where \( t = \lfloor N/p \rfloor \). The speedup, \( S_P \), is given by

\[
S_P = \frac{T_s}{T_P} = \frac{Nx + y}{tx + y} \quad (6.86)
\]

where \( x = \frac{n}{12} \left( n^2(7 + 6k) + n(15 + 6k) + 4 \right) \) and \( y = \frac{nk}{6} (n^2k^2 + 3nk - 3) \).

### 6.4 Data Structures for Block Matrices

A block matrix can be thought of as a four dimensional array, \( A(i, j, u, v) \), where the first index, \( i \), is the block row number, the second index, \( j \), is the block column number, so that the first two indices specify the block; the third index, \( u \), is the element column number within the block \( A_{ij} \), and the fourth index, \( v \), is the element column number within the block. A declaration of a variable with dimensions \((I, J, U, V)\) is then a block matrix made up of \( I \times J \) blocks, each of which is a \( U \times V \) matrix.

For the block matrices considered here (diagonal bordered) however, it is not necessary to use a four dimensional array because the matrix is sparse.
The matrix is held as three three dimensional arrays, \( A(i, u, v) \), \( B(i, u, v) \), \( C(i, u, v) \), say, one for the diagonal blocks, and one for each of the border blocks. The first index, \( i \), now is the number of the block, and the other two indices, \((u, v)\), is an element within that block.

For each of the vectors of unknown vectors and the right hand side vectors, a two dimensional array can be used, \( b(i, u) \), say, which is to be thought of as a vector of sub-vectors, so that \( i \) is the number of a sub-vector, and \( u \) is an element within the sub-vector.

In C, arrays can have at most two dimensions, so that it is not possible to implement block matrices in the way suggested above, but this shortcoming can be overcome, and with interest. Recall that the blocks and sub-vectors are not all the same size (see, for example, equation 6.3), so that to use the memory efficiently sufficient memory must be allocated according to the size of the block; moreover it is necessary to know the size of each block during the solution by either of the two methods mentioned, so that it is necessary not only to hold the values within the matrices themselves, but also to associate with each block two variables to hold the dimensions of that block. For vectors of sub-vectors, one variable is needed with each sub-vector to hold its number of elements. The way this is done is by declaring a block as a structure of three fields, the first field is a two dimensional array, and the other two are integers to hold the block's dimensions, thus:

```c
struct block {
    float ** blk;
    int nrows;
    int ncols;
};
```

then the diagonal blocks and borders are just arrays of struct block. Similarly,
each sub-vector is

```c
struct sub_vec{
    float *svec;
    int nelems;
};
```

and the vectors of unknowns and the vector of the right hand side are just arrays of `struct sub_vec`.

The requirement for efficiency in the usage of memory dictate that the memory allocation be performed dynamically. Notice that the matrix of coefficients is symmetric, so that each diagonal block is itself a symmetric matrix, and each lower border block is the transpose of the corresponding right border block. This symmetry is not made use of by the Gaussian elimination method, but the Choleski factorization method utilizes this symmetry. An advantage is derived from implementing the matrices in the way explained above and from dynamic memory allocation, for now it becomes possible to store only one half each of the diagonal blocks of the original matrix of coefficients and only its lower (or upper) triangular factor in a straightforward manner. If the lower half is stored, then the first row of each block is allocated one element, and the next row is allocated two elements, and so on down to a full row for the last row. The same idea is used for allocating memory to vectors of sub-vectors. This not only reduces the memory requirement to almost half, which is a good thing in itself, but also makes fetching and writing blocks faster.

**Three Dimensional Arrays in C** Despite the above, there is a way of implementing three dimensional arrays in C, albeit a way that depends on the indulgence of the compiler! Define a type, `Matrix`, say, to be a two dimensional array, then declare a variable as an array of `Matrix`. Say we have
typedef float **Matrix;
Matrix *diagonal;

then diagonal is a three dimensional array. It is, of course, still not possible to use diagonal[i][j][k] because it would be a syntax error. The way around this problem is to declare another array, say

Matrix diag_block;

and then, to get the element diagonal[i][j][k], assign diagonal[i] to diag_block

    diag_block = diagonal[i]

and then the required element is diag_block[j][k]. Such an under-hand technique is not to be recommended, and need not work with every compiler. In fact the Dynix compiler used gives a warning but the program behaves correctly.

6.5 Timing Results

In this section timing results are given for several implementations of the two methods of solution presented above, i.e. Gaussian elimination and Choleski factorization. Each method is implemented as a sequential program, and in parallel with the parallel library routines using static and dynamic scheduling, as well as with exdata with task allocation corresponding to static scheduling and another kind of scheduling which will be called “mixed”. This will be explained below. Please note that the terms static and dynamic scheduling are used here to mean, respectively, the allocation of tasks one by one to the processors regardless of how many processors there are, and dividing the available tasks by the number of processors and handing out blocks of tasks at a time depending on the number of tasks and the available
number of processors as described in section 3.4. What is sometimes meant by static scheduling will be called here deterministic scheduling, and what is sometimes meant by dynamic scheduling will be called here nondeterministic scheduling.

Let there be $N$ iterations of a loop that are to be executed concurrently on $p$ processors. In static scheduling, each processor carries out one loop iteration until all the iterations are done. Let $pid$ be the unique integer associated with each process, $1 \leq pid \leq p$, then each processor carries out the following:

\[
\text{for } (i = pid; i \leq N; i += p) \\
\text{do iteration } i;
\]

In dynamic scheduling, the number of iterations $N$ is divided into blocks according with the number of processors available, which are then allocated to the processors. Thus:

\[
\text{size} = \frac{N}{p}; \\
\text{while } ((\text{start} = (m\_next() - 1) * \text{size} + 1) \leq N) \{ \\
\text{finish} = \text{start} + \text{size} - 1; \\
\text{if } (\text{finish} > N) \\
\text{finish} = N; \\
\text{for } (i = 1; i \leq \text{finish}; i + +) \\
\text{do iteration } i; \\
\}
\]

Although this involves more processing than static scheduling, it has the advantage that the number of iterations can be controlled; in the above this is determined by $N$ and $p$, but any criteria can be used. (See chapter 3.)
Sub_Man staticm(TASK)
smarg
{ismarg
  if (MORE_W) {
    STOP_GO = 1;
    W_ARG_1 = loop.index;
    W_ARG_2 = loop.index;
    if (++loop.index > N)
      MORE_W = 0;
  }
  else
    STOP_GO = 0;

  return JOB;
}

Figure 6.5: A Sub_Manager for Static Scheduling with exdata

Static scheduling as implemented using exdata is carried out by the sub-managers. The simplest sub-managers for scheduling N iterations hands out the next iteration to be carried out to each successive request by the workers. The full code for such a sub-manager is given here (figure 6.5).

In figure 6.5, the names Sub_Man, smarg, ismarg, TASK, JOB, STOP_GO, W_ARG_1 and W_ARG_2 are pre-defined. The code for carrying out the loop iterations is just

\[
\text{start} = W\_\text{ARG}_1; \\
\text{finish} = W\_\text{ARG}_2; \\
\text{for}(i = \text{start}; i \leq \text{finish}; i++) \\
\quad \text{do iteration } i;
\]

The allocation of the iterations here is different from that in static scheduling using the parallel library subroutines, but the result is the same, that is, the
processors carry out one loop iteration at a time as opposed to carrying out a block of iterations.

Such static scheduling has the merits of being easy to implement and general enough to allow the scheduling of any loop whose iterations are independent of each other. Specifically for exdata, the sub-manager of figure 6.5 is the only one that need be written; all loops whose iterations are to be parallelized using static scheduling can then use this sub-manager provided, of course, that it is passed the values at which the iterations are to begin and end.

Static scheduling, however, does have a disadvantage. If there are many processors available, and the number of iterations is also large, then a lot of time will be wasted by processors waiting for access to the sub-manager. This cost will be incurred every time a processor has executed one iteration and so has to go back to the sub-manager for the next iteration. This cost is reduced by making the sub-manager hand out a block of iterations for each request, i.e. by using dynamic scheduling.

There is a serious drawback with dynamic scheduling, which is best illustrated by an example. Let \( N = 20 \) and \( p = 6 \). Then the size of each block of iterations is \( 3(= \lfloor 20/6 \rfloor) \). Now, each processor will carry out 3 iterations, so 6 processors will work concurrently to carry out 18 iterations of the loop. Using dynamic scheduling, the remaining 2 iterations will be handed out to the processor whose request for work is dealt with first, leaving the remaining 5 processors idle. Thus if \( T \) is the time for one loop iteration, the time taken to carry out all the iterations, using 6 processors and dynamic scheduling, is \( 5T \). This means that the speedup is, at best, \( 4 \) which is less than the speedup that could be achieved using 5 processors, and equal to that with 4 processors. In fact, the speedup figure of \( 4 \) ignores the cost, in time, of the requests themselves, and since there will be 7 requests for work when \( p = 6 \) and only four requests for \( p = 4 \), the speedup for \( p = 4 \) should be higher.
p_count = m_get_numprocs();
size = N/p_count;
...
Sub_Man mixed_m(TASK)
smarg
{
ismarg
if(MORE_W) {
    STOP_GO = 1;
    W_ARG_1 = loop_index;
    if(p_count > 0) {
        loop_index += size;
        W_ARG_2 = loop_index - 1;
        p_count--;
    } else
        W_ARG_2 = loop_index++;
    if(loop_index > N)
        MORE_W = 0;
} else
    STOP_GO = 0;
return JOB;
}

Figure 6.6: A Sub-Manager for Mixed Scheduling using \textit{exdata}

than for $p = 6$. This anomalous and undesirable situation can be overcome by a scheduling technique which will be called \textit{mixed}; it is an attempt to get the best of static scheduling and dynamic scheduling. A submanager for carrying out mixed scheduling is given in figure 6.6.

The sub-manager of figure 6.6 schedules the loop iterations as follows: Let $N = pq + r$, where $N$ is the number of loop iterations and $p$ is the number of processors; then $q$ is the quotient of $N/p$, $q = \lfloor N/p \rfloor$, and $r$ is the remainder. In figure 6.6, size $= q$, and the value of $p$ is stored in the variable p_count. The

191
sub_manager then hands out size iterations for each of the first \( p \) requests, after which it starts handing out one iteration for each subsequent request until all the iterations have been exhausted. In this way load balancing which is the best possible (subject to the performance of \( exdata \)) is achieved while the overhead of calls to the sub_manager is reduced to a minimum subject to load balancing requirements. Further, this sub-manager is the only one that need be written and it can be used for all loops since it is separate from the loop itself.

### 6.5.1 Timing Results of the Gaussian Elimination Routines

Table 6.4 shows the times taken for the execution of Gaussian elimination as described in section 6.3.1 above using: A sequential program, SEQ_GAUSS. Two parallel programs with the parallel library subroutines using static scheduling with one and dynamic scheduling with the other, LIB_ST_GAUSS and LIB_DYN_GAUSS, respectively. Two parallel programs written for \( exdata \) using static scheduling with one and mixed scheduling with the other, EX_ST_GAUSS and EX_MIX_GAUSS, respectively. In table 6.4 the system of equations solved is one where all the blocks are of the same size and of dimension \( n = 10 \), and where \( N = 20 \).

Table 6.5 shows the speedup figures corresponding to the times of table 6.4 as well as the expected ideal speedup from equations 6.65 and 6.66 above. The speedup is calculated as the ratio of the time for SEQ_GAUSS to that of the parallel program. Figure 6.7 is a plot of the speedup figures of table 6.5.

The first thing to say about these figures is that the times given are "wall clock" times. They were obtained by reading the system's clock just before the start of the execution of the algorithm and again just after the end of the algorithm. The units of the system's clock are hundredths of a
<table>
<thead>
<tr>
<th>NProcs.</th>
<th>SEQ</th>
<th>LIB_ST</th>
<th>LIB_DYN</th>
<th>EX_ST</th>
<th>EX_MIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5.05</td>
<td>5.10</td>
<td>5.10</td>
<td>5.18</td>
<td>5.06</td>
</tr>
<tr>
<td>2</td>
<td>2.79</td>
<td>2.80</td>
<td>2.74</td>
<td>2.72</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2.16</td>
<td>2.41</td>
<td>2.03</td>
<td>2.00</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1.76</td>
<td>1.77</td>
<td>1.57</td>
<td>1.55</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1.61</td>
<td>1.64</td>
<td>1.35</td>
<td>1.33</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1.72</td>
<td>1.92</td>
<td>1.36</td>
<td>1.35</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1.57</td>
<td>1.80</td>
<td>1.15</td>
<td>1.15</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1.69</td>
<td>1.88</td>
<td>1.17</td>
<td>1.17</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1.79</td>
<td>1.96</td>
<td>1.17</td>
<td>1.16</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1.68</td>
<td>1.64</td>
<td>0.98</td>
<td>0.96</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.4: Timing Results for Gaussian Elimination with $N = 20, n = 10$, in seconds

<table>
<thead>
<tr>
<th>NProcs.</th>
<th>Expected Speedup</th>
<th>LIB_ST</th>
<th>LIB_DYN</th>
<th>EX_ST</th>
<th>EX_MIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.00</td>
<td>0.99</td>
<td>0.99</td>
<td>0.97</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>1.97</td>
<td>1.81</td>
<td>1.84</td>
<td>1.84</td>
<td>1.86</td>
</tr>
<tr>
<td>3</td>
<td>2.77</td>
<td>2.34</td>
<td>2.10</td>
<td>2.49</td>
<td>2.53</td>
</tr>
<tr>
<td>4</td>
<td>3.81</td>
<td>2.87</td>
<td>2.85</td>
<td>3.22</td>
<td>3.26</td>
</tr>
<tr>
<td>5</td>
<td>4.69</td>
<td>3.14</td>
<td>3.08</td>
<td>3.74</td>
<td>3.80</td>
</tr>
<tr>
<td>6</td>
<td>4.69</td>
<td>2.94</td>
<td>2.63</td>
<td>3.71</td>
<td>3.74</td>
</tr>
<tr>
<td>7</td>
<td>6.10</td>
<td>3.22</td>
<td>2.81</td>
<td>4.39</td>
<td>4.39</td>
</tr>
<tr>
<td>8</td>
<td>6.10</td>
<td>2.99</td>
<td>2.69</td>
<td>4.32</td>
<td>4.32</td>
</tr>
<tr>
<td>9</td>
<td>6.10</td>
<td>2.82</td>
<td>2.58</td>
<td>4.32</td>
<td>4.32</td>
</tr>
<tr>
<td>10</td>
<td>8.71</td>
<td>3.01</td>
<td>3.08</td>
<td>5.15</td>
<td>5.26</td>
</tr>
</tbody>
</table>

Table 6.5: Speedup figures corresponding to the timing figures of table 6.4
Figure 6.7: Plot of Speedup v. number of processors from table 6.5
second. It follows that these times represent the duration of the running of the programs, which includes not only the periods when the programs themselves are being executed (which are what we would like to measure) but also periods of system activity such as swapping processes from one processor to another and memory management activities such as fetching from and writing pages to the disk, and so on. In order to reduce the effect of the system's activities on the timing figures, the programs were run when the computer was free, but it is true to say that the times given represent an upper limit, or, in other words, the worst possible performance. However, since all the programs were timed in the same way, it is possible to compare them.

The second thing to say is that the expected ideal speedup figures in table 6.5 and figure 6.7 were obtained using equations 6.67 and 6.68, which do not take into account addition and subtraction operations, or communication and scheduling costs. They are, indeed, ideal.

The salient features of figure 6.7 are:

1. All five curves have the same shape.

2. The curves can be grouped into three groups, the first containing the ideal speedup curve, the second containing the two exdata curves, and the third containing the two library subroutines curves.

3. The two exdata curves are almost identical, though mixed scheduling shows very slightly better speedups than static scheduling does, as expected. Where the ideal speedup curve remains constant for $p = 6, 7$ and 8, both the exdata curves dip slightly. This, too, is expected since the increased number of processors means that the processors have to wait longer for access to the manager and the sub-managers.

The very close similarity between the two exdata curves, and given
the comments earlier about the timing of the routines, lead one to ask whether it is worthwhile to use mixed scheduling at all. The answer to this question will be postponed until results for Gaussian elimination with \( n = 25 \), i.e. when the sub-tasks are considerably bigger, and also when \( N = 120 \) with \( n = 10 \), i.e. when the number of sub-tasks is increased, are given.

4. The two library curves show a generally worse performance than the \textit{exdata} curves, with the disparity between the two groups getting larger as the number of processors is increased. It is clear that the overheads of setting up the processors overwhelms the performance of the algorithms. Again this point will be taken up later as more results are given.

Table 6.6 gives the timing results for Gaussian elimination with \( N = 20 \) and \( n = 25 \). The number of operations per sub-task is increased form \( n = 10 \) by a ratio of \((25^3/10^3 \approx)16\). Table 6.7 gives the speedup figures and figure 6.8 is a plot of the speedup figures.

Table 6.8 gives the timing results for Gaussian elimination with \( N = 120 \) and \( n = 10 \). Table 6.7 gives the speedup figures and figure 6.9 is a plot of the speedup figures.

In figure 6.8, with the size of each sub-task increased significantly from that in figure 6.7, the overhead costs of setting up the processes and communication become less significant, and the performance of all four programs is closer to the ideal. In particular, static scheduling with the parallel library subroutines, and the two \textit{exdata} programs perform very similarly. For \( p \geq 5 \), the best performance is obtained with mixed scheduling using \textit{exdata}, followed very closely by static scheduling using \textit{exdata}. For \( p < 5 \), the best performance is that of static scheduling using the parallel library subroutines.

As in figure 6.7, the performances of the two \textit{exdata} programs are so very
<table>
<thead>
<tr>
<th>NProcs.</th>
<th>SEQ</th>
<th>LIB_ST</th>
<th>LIB_DYN</th>
<th>EX_ST</th>
<th>EX_MIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>70.45</td>
<td>70.87</td>
<td>71.02</td>
<td>71.09</td>
<td>70.32</td>
</tr>
<tr>
<td>2</td>
<td>36.44</td>
<td>36.15</td>
<td>36.75</td>
<td>36.34</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>26.12</td>
<td>29.31</td>
<td>26.19</td>
<td>26.19</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>19.12</td>
<td>19.34</td>
<td>19.42</td>
<td>19.37</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>16.02</td>
<td>15.96</td>
<td>16.15</td>
<td>16.01</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>16.09</td>
<td>19.25</td>
<td>16.06</td>
<td>16.04</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>12.83</td>
<td>16.05</td>
<td>12.84</td>
<td>12.81</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>12.97</td>
<td>16.08</td>
<td>12.97</td>
<td>12.92</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>13.28</td>
<td>16.22</td>
<td>13.01</td>
<td>12.98</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>10.07</td>
<td>10.30</td>
<td>9.70</td>
<td>9.68</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.6: Timing Results for Gaussian Elimination with $N = 20, n = 25$, in seconds

<table>
<thead>
<tr>
<th>NProcs.</th>
<th>Expected Speedup</th>
<th>LIB_ST</th>
<th>LIB_DYN</th>
<th>EX_ST</th>
<th>EX_MIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.00</td>
<td>1.00</td>
<td>0.99</td>
<td>0.99</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>1.97</td>
<td>1.93</td>
<td>1.95</td>
<td>1.92</td>
<td>1.94</td>
</tr>
<tr>
<td>3</td>
<td>2.77</td>
<td>2.70</td>
<td>2.40</td>
<td>2.69</td>
<td>2.69</td>
</tr>
<tr>
<td>4</td>
<td>3.81</td>
<td>3.67</td>
<td>3.64</td>
<td>3.63</td>
<td>3.64</td>
</tr>
<tr>
<td>5</td>
<td>4.69</td>
<td>4.40</td>
<td>4.41</td>
<td>4.36</td>
<td>4.40</td>
</tr>
<tr>
<td>6</td>
<td>4.69</td>
<td>4.38</td>
<td>3.66</td>
<td>4.39</td>
<td>4.39</td>
</tr>
<tr>
<td>7</td>
<td>6.10</td>
<td>5.49</td>
<td>4.39</td>
<td>5.49</td>
<td>5.50</td>
</tr>
<tr>
<td>8</td>
<td>6.10</td>
<td>5.43</td>
<td>4.38</td>
<td>5.43</td>
<td>5.45</td>
</tr>
<tr>
<td>9</td>
<td>6.10</td>
<td>5.30</td>
<td>4.34</td>
<td>5.42</td>
<td>5.43</td>
</tr>
<tr>
<td>10</td>
<td>6.10</td>
<td>7.00</td>
<td>6.84</td>
<td>7.26</td>
<td>7.28</td>
</tr>
</tbody>
</table>

Table 6.7: Speedup figures corresponding to the timing figures of table 6.6

197
Table 6.8: Timing Results for Gaussian Elimination with $N = 120, n = 10$, in seconds
<table>
<thead>
<tr>
<th>NProcs.</th>
<th>Expected Speedup</th>
<th>LIB_ST</th>
<th>LIB_DYN</th>
<th>EX_ST</th>
<th>EX_MIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>0.99</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>1.99</td>
<td>1.96</td>
<td>1.97</td>
<td>1.90</td>
<td>1.93</td>
</tr>
<tr>
<td>3</td>
<td>2.98</td>
<td>2.84</td>
<td>2.85</td>
<td>2.75</td>
<td>2.79</td>
</tr>
<tr>
<td>4</td>
<td>3.97</td>
<td>3.69</td>
<td>3.68</td>
<td>3.51</td>
<td>3.57</td>
</tr>
<tr>
<td>5</td>
<td>4.95</td>
<td>4.42</td>
<td>4.45</td>
<td>4.22</td>
<td>4.31</td>
</tr>
<tr>
<td>6</td>
<td>5.92</td>
<td>5.02</td>
<td>5.11</td>
<td>4.91</td>
<td>4.93</td>
</tr>
<tr>
<td>7</td>
<td>6.56</td>
<td>5.56</td>
<td>5.59</td>
<td>5.36</td>
<td>5.38</td>
</tr>
<tr>
<td>8</td>
<td>7.85</td>
<td>6.13</td>
<td>6.16</td>
<td>6.04</td>
<td>6.12</td>
</tr>
<tr>
<td>9</td>
<td>8.40</td>
<td>6.29</td>
<td>5.88</td>
<td>6.36</td>
<td>6.42</td>
</tr>
<tr>
<td>10</td>
<td>9.76</td>
<td>6.86</td>
<td>6.93</td>
<td>6.98</td>
<td>7.08</td>
</tr>
</tbody>
</table>

Table 6.9: Speedup figures corresponding to the timing figures of table 6.8

Figure 6.9: Plot of Speedup v. number of processors from table 6.9
close as to suggest that it is not worthwhile to use mixed scheduling at all. What happens in mixed scheduling is that the number of calls to the sub-managers is reduced, but the work done by the sub-managers for each call is increased. For the problem under consideration, with \( N = 20 \), the number of loop iterations is quite small, and so there is not a large number of calls to the sub-managers in the first place.

In figure 6.9, \( n \) is the same as in figure 6.7 (\( n = 10 \)), but the number of sub-tasks, \( N \), is increased to 120. The performances of all four programs are, as in figure 6.8, closer to the ideal performance than in figure 6.7, and for the same reasons. Here also, static scheduling with the library routines yields best performance for up to five processors, and thereafter it is mixed scheduling with \( exdata \) that comes out on top. However, the performances represented by the top three curves are very close to each other.

With \( N = 120 \), one would have expected mixed scheduling to outperform static scheduling significantly. Although mixed scheduling does perform better than static scheduling, the improvement is not significant. In fact, the improvement is between 1 and 2% only. One concludes that mixed scheduling is not worth using in cases where the number of loop iterations is not large, in the hundreds, say, and where each iteration involves an amount of work such that it makes the time spent in calls to the sub-managers insignificant. In the next chapter an example will be given where mixed scheduling becomes necessary.

Comparing figure 6.7 on the one hand, and figures 6.8 and 6.9 on the other, the most striking difference is in the improvement of the poor performance of the library programs for a small number of small sub-tasks (figure 6.7) which is far behind those of \( exdata \). With the size of the sub-tasks or the number of the sub-tasks increased, the performance becomes comparable with \( exdata \) and indeed better for a small number of processors. The explanation for this is not known. One can guess the overheads of the library
routines are high, but then it would follow that this would always make a difference; however, it would still be possible that once the processes are set up they run more efficiently than under *exdata*. To find out the reason for this behaviour one needs to look at the way the library routines are implemented and this is not our concern here. Our concern is to show that explicit parallel programming can be generalized so as to make it possible to devise tools for facilitating it and discovering, possibly, paradigms that may be used in future development of general parallel processing, and this has been achieved. Even had the performance of the programs under *exdata* been consistently worse than using the parallel library subroutines, provided reasonable results were achieved, this would have been enough.

In describing the system of equations which have been solved using Gaussian elimination, it was stated that the blocks making up the matrix of coefficients and the sub-vectors making up the vector of unknown quantities and the right hand side vector need not all be the same size, but the examples given above have all been for systems with equal size blocks and sub-vectors. The reason is that it was desired to compare the various parallel programs and for this equal size blocks are perfectly adequate. Variable size blocks require considerations of scheduling that are best left to a separate discussion and this will be included in chapter 8.

### 6.5.2 Load Balancing in *exdata* for Gaussian Elimination

The distribution of loop iterations among concurrent processes using the parallel library routines is deterministic in the sense that each process carries out iterations that are determined by the loop control statement:

\[(for \ i = \text{pid}; \ i \leq N; \ i++; =p)\]
so that the process whose id number is \( j \) will carry out iterations \( j, j + p, j + 2p, \ldots \) every time the program is run. In \textit{exdata} however, the distribution is non-deterministic, and it it not known in advance which process, or worker, will carry out which iteration. The sub-manager of a particular loop hands out the first iteration or block of iterations to the first request it receives, and the second iteration or block of iterations to the second request it receives, and so on. This non-determinism is a consequence of the fundamental principle upon which \textit{exdata} was built, namely that all the workers are identical to each other and perform instructions received from the manager and sub-managers who alone are responsible for scheduling the main tasks and sub-tasks.

The deterministic distribution of the loop iterations carried out by the loop control statement ensures that the iterations are distributed evenly among the processes. There is no such guarantee with \textit{exdata}; it is possible that the loop iterations will be distributed unevenly among the processes. The only thing that will prevent this happening is the quick and efficient performance of the tool \textit{exdata} itself. But even if \textit{exdata} were as efficient as possible, its operations must take up some time, and if the amount of work in each sub-task takes a period of time which is comparable to that taken up by securing access to the manager and sub-managers, then uneven distribution will very probably occur. Again, this situation arises out of the very idea behind \textit{exdata} mentioned in the previous paragraph. It is clear, however, that when sub-tasks are that small, parallelism of the sort considered here will not be beneficial anyway.

The two observations above make it important to look and see how \textit{exdata} distributed the sub-tasks for the programs actually run, namely \texttt{EX.ST.GAUSS} and \texttt{EX.MIX.GAUSS}. There are two version of \textit{exdata} one of which produces a "log" file for the workers after each run, giving, for each worker, each main task it received from the manager, and the sub-tasks it received from the
Table 6.10: Load Distribution using ex_ST_GAUSS with $N = 20, n = 10, p = 6$

<table>
<thead>
<tr>
<th>Worker</th>
<th>Iterations of Elimination</th>
<th>Iterations of Back-Substitution</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2,7,13,19</td>
<td>4,10,16</td>
</tr>
<tr>
<td>1</td>
<td>3,12,18</td>
<td>3,9,15</td>
</tr>
<tr>
<td>2</td>
<td>5,11,17</td>
<td>6,12,18</td>
</tr>
<tr>
<td>3</td>
<td>4,9,14,20</td>
<td>5,11,17</td>
</tr>
<tr>
<td>4</td>
<td>1,8,16</td>
<td>1,7,13,19</td>
</tr>
<tr>
<td>5</td>
<td>6,10,15</td>
<td>2,8,14,20</td>
</tr>
</tbody>
</table>

Figure 6.10: The activity of 6 workers in a run of EX_ST_GAUSS

sub-managers, along with the times at which these were received. From this information, the distribution of the sub-tasks can be seen. Table 6.10 gives the distribution for EX_ST_GAUSS with $N = 20, n = 10$ and using 6 workers.

From the workers' log file the activity of the workers can also be inspected. Figure 6.10 is a summary of the action of the 6 workers in the run from which table 6.10 was obtained.

All the runs of the programs under exdata produced even load distribution, figure 6.10 is one example. It will be noted here that information of
the workers' activities is given without requiring the user to provide anything other than the input routines and the dependency relations, and that this information can be very useful to a programmer during the development stage of a program.

6.5.3 Timing Results of the Choleski Factorization Routines

The programs of the implementations of the Choleski factorization method are SEQ_CHOL, sequential program, LIB_STAT_CHOL, parallel program with the library routines and static scheduling; LIB_DYN_CHOL, parallel program with the library routines and dynamic scheduling; EX_ST_CHOL, parallel program in exdata using static scheduling; and EX_MIX_CHOL, parallel program in exdata using mixed scheduling.

Table 6.11 shows the times taken for the execution of all these programs with $N = 20$ and $n = 10$. Table 6.12 gives the speedup figures corresponding to these times, and figure 6.11 is a plot of the speedup figures.

Table 6.13 gives the execution times with $N = 20$ and $n = 25$; table 6.14 gives the speedup figures, and figure 6.12 is a plot of table 6.14.

Table 6.15 gives the execution times with $N = 120$ and $n = 10$; table 6.16 gives the speedup figures, and figure 6.13 is a plot of table 6.16.

The expected ideal speedup figures in tables 6.12, 6.14, and 6.16, are worked out using equation 6.86.

The comments made on the timing results of the Gaussian elimination programs, on the accuracy of the times, and on the expected ideal speedups, apply here as well.

Indeed, the plots of the speedup figures, figures 6.11, 6.12, and 6.13, are very similar to those for Gaussian elimination (figures 6.7, 6.8, and 6.9), and so all the comments made earlier on these also apply here.
### Table 6.11: Timing results for Choleski factorization with $N = 20, n = 10$. All times are in seconds.

<table>
<thead>
<tr>
<th>NProcs</th>
<th>SEQ</th>
<th>LIB_ST</th>
<th>LIB_DYN</th>
<th>EX_ST</th>
<th>EX_MIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.27</td>
<td>2.32</td>
<td>2.30</td>
<td>2.33</td>
<td>2.28</td>
</tr>
<tr>
<td>2</td>
<td>1.27</td>
<td>1.26</td>
<td>1.27</td>
<td>1.26</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1.01</td>
<td>1.10</td>
<td>0.97</td>
<td>1.26</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.86</td>
<td>0.86</td>
<td>0.74</td>
<td>0.92</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0.83</td>
<td>0.83</td>
<td>0.65</td>
<td>0.73</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0.90</td>
<td>1.00</td>
<td>0.66</td>
<td>0.64</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0.87</td>
<td>0.96</td>
<td>0.57</td>
<td>0.56</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0.95</td>
<td>1.04</td>
<td>0.58</td>
<td>0.59</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1.00</td>
<td>1.10</td>
<td>0.60</td>
<td>0.60</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1.00</td>
<td>1.00</td>
<td>0.52</td>
<td>0.51</td>
<td></td>
</tr>
</tbody>
</table>

It is worth repeating that the Choleski factorization programs work on only one half of the diagonal blocks since Choleski factorization preserves the symmetry of symmetric matrices, whereas the Gaussian elimination programs work on the full diagonal blocks. It follows that the size of the sub-tasks in Choleski factorization, for a given $n$, are about half the size in Gaussian elimination.

The same considerations for load distribution apply here as well, and figure 6.14 shows the activity of the workers in factorizing a system with $N = 20, n = 10$, using 7 processors.
Table 6.12: Speedup figures corresponding to table 6.11

<table>
<thead>
<tr>
<th>NProcs</th>
<th>Expected</th>
<th>LIB_ST</th>
<th>LIB_DYN</th>
<th>EX_ST</th>
<th>EX_MIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.00</td>
<td>0.98</td>
<td>0.99</td>
<td>0.97</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>1.98</td>
<td>1.79</td>
<td>1.80</td>
<td>1.79</td>
<td>1.80</td>
</tr>
<tr>
<td>3</td>
<td>2.81</td>
<td>2.25</td>
<td>2.06</td>
<td>2.34</td>
<td>2.47</td>
</tr>
<tr>
<td>4</td>
<td>3.90</td>
<td>2.64</td>
<td>2.64</td>
<td>3.07</td>
<td>3.11</td>
</tr>
<tr>
<td>5</td>
<td>4.84</td>
<td>2.73</td>
<td>2.73</td>
<td>3.49</td>
<td>3.55</td>
</tr>
<tr>
<td>6</td>
<td>4.84</td>
<td>2.52</td>
<td>2.27</td>
<td>3.44</td>
<td>3.49</td>
</tr>
<tr>
<td>7</td>
<td>6.34</td>
<td>2.61</td>
<td>2.36</td>
<td>3.98</td>
<td>4.05</td>
</tr>
<tr>
<td>8</td>
<td>6.34</td>
<td>2.39</td>
<td>2.18</td>
<td>3.91</td>
<td>3.85</td>
</tr>
<tr>
<td>9</td>
<td>6.34</td>
<td>2.27</td>
<td>2.06</td>
<td>3.78</td>
<td>3.78</td>
</tr>
<tr>
<td>10</td>
<td>9.30</td>
<td>2.27</td>
<td>2.27</td>
<td>4.37</td>
<td>4.45</td>
</tr>
</tbody>
</table>

Figure 6.11: Plot of speedup vs. number of processors from table 6.12
<table>
<thead>
<tr>
<th>NProcs</th>
<th>SEQ</th>
<th>LIB_ST</th>
<th>LIB_DYN</th>
<th>EX_ST</th>
<th>EX_MIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>30.92</td>
<td>30.87</td>
<td>30.87</td>
<td>30.87</td>
<td>31.03</td>
</tr>
<tr>
<td>2</td>
<td>15.83</td>
<td>15.72</td>
<td>15.09</td>
<td>15.93</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>11.44</td>
<td>12.79</td>
<td>11.37</td>
<td>11.34</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>8.47</td>
<td>8.39</td>
<td>8.37</td>
<td>8.41</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6.99</td>
<td>6.99</td>
<td>6.94</td>
<td>6.87</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>7.05</td>
<td>8.45</td>
<td>6.91</td>
<td>6.89</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>5.74</td>
<td>7.10</td>
<td>5.45</td>
<td>5.44</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>5.77</td>
<td>7.18</td>
<td>5.48</td>
<td>5.48</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>5.84</td>
<td>7.31</td>
<td>5.51</td>
<td>5.50</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>4.60</td>
<td>4.58</td>
<td>4.05</td>
<td>4.05</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.13: Timing results for Choleski factorization with \( N = 20, n = 25 \).
All times are in seconds.

<table>
<thead>
<tr>
<th>NProcs</th>
<th>Expected</th>
<th>LIB_ST</th>
<th>LIB_DYN</th>
<th>EX_ST</th>
<th>EX_MIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>1.98</td>
<td>1.95</td>
<td>1.97</td>
<td>1.94</td>
<td>1.94</td>
</tr>
<tr>
<td>3</td>
<td>2.82</td>
<td>2.70</td>
<td>2.42</td>
<td>2.72</td>
<td>2.73</td>
</tr>
<tr>
<td>4</td>
<td>3.91</td>
<td>3.65</td>
<td>3.69</td>
<td>3.69</td>
<td>3.68</td>
</tr>
<tr>
<td>5</td>
<td>4.85</td>
<td>4.42</td>
<td>4.42</td>
<td>4.46</td>
<td>4.50</td>
</tr>
<tr>
<td>6</td>
<td>4.85</td>
<td>4.38</td>
<td>3.66</td>
<td>4.47</td>
<td>4.49</td>
</tr>
<tr>
<td>7</td>
<td>6.38</td>
<td>5.39</td>
<td>4.35</td>
<td>5.67</td>
<td>5.68</td>
</tr>
<tr>
<td>8</td>
<td>6.38</td>
<td>5.36</td>
<td>4.31</td>
<td>5.64</td>
<td>5.64</td>
</tr>
<tr>
<td>9</td>
<td>6.38</td>
<td>5.29</td>
<td>4.23</td>
<td>5.61</td>
<td>5.62</td>
</tr>
<tr>
<td>10</td>
<td>9.33</td>
<td>6.72</td>
<td>6.75</td>
<td>7.63</td>
<td>7.63</td>
</tr>
</tbody>
</table>

Table 6.14: Speedup figures corresponding to table 6.13
Table 6.15: Timing results for Choleski factorization with $N = 120, n = 10$. All times are in seconds.

<table>
<thead>
<tr>
<th>NProcs</th>
<th>SEQ</th>
<th>LIB_ST</th>
<th>LIB_DYN</th>
<th>EX_ST</th>
<th>EX_MIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>6.99</td>
<td>6.95</td>
<td>7.30</td>
<td>7.18</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4.86</td>
<td>4.81</td>
<td>5.10</td>
<td>4.96</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3.78</td>
<td>3.72</td>
<td>4.01</td>
<td>3.91</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>3.21</td>
<td>3.14</td>
<td>3.38</td>
<td>3.26</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2.81</td>
<td>2.79</td>
<td>2.95</td>
<td>2.86</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>2.58</td>
<td>2.60</td>
<td>2.64</td>
<td>2.64</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>2.43</td>
<td>2.40</td>
<td>2.44</td>
<td>2.37</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>2.41</td>
<td>2.51</td>
<td>2.32</td>
<td>2.27</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>2.26</td>
<td>2.22</td>
<td>2.15</td>
<td>2.08</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.16: Speedup figures corresponding to table 6.15
Figure 6.12: Plot of speedup figures vs. number of processors from table 6.14

Figure 6.13: Plot of speedup figures vs. number of processors from table 6.16
Figure 6.14: The activity of the workers in carrying out Choleski factorization, with $N = 20, n = 10$ with exdata using 7 workers. The time is in seconds.
Chapter 7

Parallel Romberg Integration

7.1 Introduction

Numerical integration, or quadrature, is a large field of knowledge. One of the simplest ways of carrying out quadrature is the trapezoidal rule, which is one of a family of rules called the closed Newton-Cotes formulae. This rule, however, does not give results whose accuracy can always be assessed. Romberg integration is a way of combining results obtained from the trapezoidal rule, or another closed Newton-Cotes formula, to obtain more accurate values of the integral as well as better error assessment.

In this chapter single integrals and double integrals will be evaluated in parallel. Numerical evaluation of single integrals is far less difficult than that of double and higher integrals. There are two reasons for this [78]. The first is that one-dimensional space is much simpler geometrically than two- or higher dimensional spaces. In one-dimensional space all regions are equivalent to each other under a transformation, but in higher dimensional spaces the number of closed connected regions is infinite. It follows then that a different integration formula is required for each of these regions. The double integrals evaluated in this chapter are taken over a rectangular
region. The second reason is that the integration formulae are related to polynomial approximations of functions, and these are very well understood in one dimension, but general polynomial approximations for three and higher dimensions are not known.

The parallelization of the Romberg integration procedure, as implemented here, will be seen to require data partitioning which is quite different from that encountered in the examples of the previous chapters. It will be required here to parallelize a double nested loop, in the case of single integrals, and a triple nested loop in the case of double integrals. The problem encountered is that the amount of work in the body of the outer loop varies from one iteration to the next, and so it will be necessary to partition the loops at two levels simultaneously, which is not possible using DOACROSS constructs, and the use of a sub-manager becomes essential, as will be seen below.

7.2 Romberg Integration

7.2.1 The Trapezoidal Rule

To evaluate the integral $I$ given by equation 7.1 below

$$ I = \int_a^b f(x) \, dx $$

(7.1)

where $f$ is continuous and differentiable in the interval $(a, b)$, we divide the interval $(a - b)$ into $n$ divisions each of length $h = (a - b)/n$. Let $f_0$ denote the value of the function $f$ at $a$, i.e. $f_0 = f(a)$, and let $f_i$, $1 \leq i \leq n - 1$, denote the value of $f(a + ih)$, and $f_n$ denote the value of $f$ at $b$, or $f(b)$. Clearly, $a + nh = b$.

Figure 7.1 shows a curve representing the function $f(x)$ between $x = a$ and $x = b$, with the interval divided into five divisions. In the trapezoidal method the area between the curve $f(x)$ and the $x$-axis is approximated as a sum
Figure 7.1: The interval $a$ to $b$ divided into 5 divisions.

of the areas of trapezoids, each of which is obtained by approximating the function $f(x)$ within a division, that is between $(a + ih)$ and $(a + (i+1)h)$ by a straight line between $f(a + ih)$ and $f(a + (i + 1)h)$ as shown in figure 7.1.

The area of each trapezoid is given by (see figure 7.2)

$$t_A = h f_{i+1} + \frac{1}{2} h (f_i - f_{i+1})$$

$$= h \left( \frac{1}{2} f_i + \frac{1}{2} f_{i+1} \right)$$

This is true also if, unlike figure 7.2, $f_{i+1} > f_i$.

An approximate value of the integral is then obtained by summing up the areas of all the trapezoids,

$$T(h) = \sum t_A = h \left\{ \frac{1}{2} f_0 + \frac{1}{2} f_1 + \frac{1}{2} f_2 + \cdots + \frac{1}{2} f_{n-1} + \frac{1}{2} f_n \right\}$$

$$= h \left( \frac{1}{2} f_0 + f_1 + f_2 + \cdots + \frac{1}{2} f_n \right)$$

(7.2)

Equation 7.2 is the well known trapezoidal rule.

A more rigorous way of obtaining the trapezoidal rule is as follows. Consider the function $f(x)$ whose values at $x = x_0$ is $f_0$ and at $x = x_1$ is $f_1$. The first degree Lagrange polynomial approximation is given by

$$P_1(x) = \frac{x - x_1}{x_0 - x_1} f_0 + \frac{x - x_0}{x_1 - x_0} f_1$$

(7.3)
Let $x_1 - x_0 = h$, then

$$P_1(x) = \frac{1}{h} \{x(f_1 - f_0) + x_1f_0 - x_0f_1\} \quad (7.4)$$

An approximate value of $\int_{x_0}^{x_1} f(x) \, dx$ is obtained by integrating $P_1(x)$ to obtain

$$\int_{x_0}^{x_1} f(x) \, dx \approx \frac{1}{h} \int_{x_0}^{x_1} \{x(f_1 - f_0) + x_1f_0 - x_0f_1\} \, dx$$

$$\approx \frac{1}{2} h(f_0 + f_1) \quad (7.5)$$

which is the same as the area of the strip $t_A$ above.

It can be shown that for integrands with finite values and finite derivatives of all orders within the interval, and for a sufficiently small division, $h$, $T(h)$ is related to $I$ by equation (7.6)

$$I = T(h) + Ah^2 + Bh^4 + Ch^6 + \cdots \quad (7.6)$$

where $A, B, C, \text{etc.}$ are constants whose values are functions of the derivatives of $f(x)$ at the terminal points $a$ and $b$ [29]. This equation is known as the Euler-Maclaurin formula. The following derivation is based on the
use of operator calculus and comes from [18]. It is important to note that
derivation using operator calculus is not the same as a formal proof. First we
introduce some introductory operator calculus, followed by a consideration
of the function \( F(x) \) given by

\[
F(x) = \frac{x}{2} \cdot \frac{e^x + 1}{e^x - 1}
\]

(7.7)

which is important in the derivation that follows.

Operator Calculus

It is possible to carry out calculations treating operators as though they
were variables in algebraic expressions. Two operators will be used in the
derivation of the Euler-Mac1aurin formula: the differentiation operator \( \mathcal{D} \);
and the forward difference operator \( \mathcal{E} \). These are defined as follows:

\[
\mathcal{D} f(x) = f'(x)
\]

(7.8)

\[
\mathcal{E} f(x) = f(x + h)
\]

(7.9)

An operator \( \mathcal{Q} \) is said to be linear if

\[
\mathcal{Q}(\alpha f + \beta g) = \alpha Qf + \beta Qg
\]

where \( \alpha \) and \( \beta \) are arbitrary constants, and \( f \) and \( g \) are arbitrary functions.
The operators \( \mathcal{D} \) and \( \mathcal{E} \) are both linear.

Algebraic relations between two operators \( \mathcal{P} \) and \( \mathcal{Q} \), are defined by equa-
tions 7.10 below for an arbitrary function \( f \) and an arbitrary constant \( \alpha \):

\[
\begin{align*}
(P + Q)f &= Pf + Qf \\
(P - Q)f &= Pf - Qf \\
(PQ)f &= P(Qf) \\
(\alpha P)f &= \alpha Pf \\
P^n f &= P \cdot P \ldots Pf \quad n \text{ factors}
\end{align*}
\]

(7.10)
For example, the Taylor series:

\[ f(x + h) = f(x) + hf'(x) + \frac{h^2}{2!}f''(x) + \frac{h^3}{3!}f'''(x) + \ldots \]  

(7.11)

can be written using operators as

\[ \mathcal{E}f(x) = \left(1 + h\mathcal{D} + \frac{(h\mathcal{D})^2}{2!} + \frac{(h\mathcal{D})^3}{3!} + \ldots\right)f(x) \]  

(7.12)

The Taylor expansion for the exponential function \( e^h \) is

\[ e^h = 1 + h + \frac{h^2}{2!} + \frac{h^3}{3!} + \ldots \]  

(7.13)

so that,

\[ \mathcal{E} = e^{h\mathcal{D}} \]  

(7.14)

Equation 7.14 will be needed in the derivation.

Finally we note that the inverse of the differential operator \( \mathcal{D} \) is the indefinite integral, i.e.

\[ \frac{1}{\mathcal{D}}f(x) = \int f(x) \, dx \]

Expansion of the function \( F(x) \)

Set \( F(x) \) as the series

\[ F(x) = c_0 + c_1x + c_2x^2 + c_3x^3 + \ldots \]  

(7.15)

Observe first that \( F(x) = F(-x) \), i.e. the function is even, so that the odd terms in the series of equation 7.15 are all zero, for

\[ F(-x) = \frac{x}{2} \cdot \frac{e^{-x} + 1}{e^{-x} - 1} = -\frac{x}{2} \cdot \frac{1 + e^x}{1 - e^x} \]

by multiplying the numerator and the denominator by \( e^x \), so that

\[ F(x) = F(-x) \]

It follows then that

\[ F(x) = c_0 + c_2x^2 + c_4x^4 + \ldots \]  

(7.16)
Now, equation 7.7 can be rewritten as

\[(e^x - 1)F(x) = \frac{1}{2}x(e^x + 1)\]

and, by equations 7.13 and 7.16

\[
\left( x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots \right) (c_0 + c_2x^2 + c_4x^4 + \ldots) = \frac{1}{2}x \left( 2 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots \right)
\]

(7.17)

by comparing coefficients, \(c_0 = 1\).

From equation 7.17 we obtain

\[
\left( x + \frac{x^2}{2} + \frac{x^3}{2.3!} + \frac{x^5}{2.4!} + \ldots \right) = c_0x + c_2\frac{x^2}{2} + c_4x^3 + x^4 \left( \frac{c_2}{2!} + \frac{c_4}{4!} \right) + x^5 \left( \frac{c_4}{3!} + \frac{c_2}{5!} \right) + \ldots
\]

and taking the coefficients of \(x^{2n+1}\)

\[
\frac{c_0}{(2n+1)!} + \frac{c_2}{(2n-1)!} + \frac{c_4}{(2n-3)!} + \ldots + \frac{c_{2n}}{1!} = \frac{1}{2} \frac{1}{(2n)!} \quad \text{for } n \geq 1 \quad (7.18)
\]

and so the coefficients are found as functions

\[
n = 1 \Rightarrow c_2 = \frac{1}{2} \cdot \frac{1}{2} - \frac{29}{3!} = \frac{1}{12}
\]

\[
n = 2 \Rightarrow c_4 = \frac{1}{2} \cdot \frac{1}{4} - \frac{29}{5!} - \frac{32}{3!} = -\frac{1}{720}
\]

etc.

The Euler-Maclaurin formula can now be derived.

With operator notation the trapezoidal rule becomes

\[
T(h) = h \left( \frac{1}{2} + e^{hD} + e^{2hD} + e^{3hD} + \ldots + e^{(n-1)hD} + e^{nhD} \right)
\]

\[
= h \left( \frac{1}{2} + \sum_{i=1}^{n-1} e^{ihD} + \frac{1}{2} e^{nhD} \right) f(x_0)
\]

\[
= h \left( -\frac{1}{2} + \sum_{i=0}^{n-1} e^{ihD} + \frac{1}{2} e^{nhD} \right) f(x_0)
\]

(7.19)
but \( \sum_{i=0}^{n-1} e^{ihD} \) is just a geometric series whose sum is \( (e^{nhD} - 1)/(e^{hD} - 1) \) and so equation 7.19 becomes

\[
T(h) = \frac{h}{2} \cdot \frac{e^{hD} + 1}{e^{hD} - 1} \cdot (e^{nhD} - 1)f(x_0)
\]

\[
= \frac{hD}{2} \cdot \frac{e^{hD} + 1}{e^{hD} - 1} \cdot \frac{e^{nhD} - 1}{D} f(x_0)
\]

(7.20)

But

\[
\frac{hD}{2} \cdot \frac{e^{hD} + 1}{e^{hD} - 1} = 1 + \sum_{i=1}^{\infty} c_i(hD)^i
\]

where all terms \( c_{2r+1} \) are zero, so that

\[
T(h) = \left( 1 + \sum_{i=1}^{\infty} c_i(hD)^i \right) \cdot \frac{e^{nhD} - 1}{hD} \cdot f(x_0)
\]

\[
= \frac{e^{nhD} - 1}{D} f(x_0) + \sum_{i=1}^{\infty} c_i(hD)^{i-1}(e^{nhD} - 1)hf(x_0)
\]

(7.21)

Now,

\[
\frac{e^{nhD} - 1}{D} f(x_0) = \frac{1}{D} \{ f(x_0 + nhD) - f(x_0) \} = \int_{x_0}^{x_n} f(x) \, dx
\]

and

\[
h(hD)^{i-1}(e^{nhD} - 1)f(x_0) = h^i \left\{ f^{(i-1)}(x_n) - f^{(i-1)}(x_0) \right\}
\]

and so

\[
T(h) = \int_{x_0}^{x_n} f(x) \, dx + \sum_{i=1}^{\infty} c_i h^i \left\{ f^{(i-1)}(x_n) - f^{(i-1)}(x_0) \right\}
\]

where all the odd terms \( c_{2r+1} \) are zero. This is the Euler-Maclaurin formula.

### 7.2.2 Romberg Integration

Consider now two approximations for \( I \) by the trapezoidal rule using two different values of \( h, h_1 \) and \( h_2 \), so that

\[
I = T(h_1) + Ah_1^2 + Bh_1^4 + Ch_1^6 + \cdots
\]

(7.22)

and

\[
I = T(h_2) + Ah_2^2 + Bh_2^4 + Ch_2^6 + \cdots
\]

(7.23)
Now, $A$ can be eliminated from equations 7.22 and 7.23

$$\frac{1}{h_1^2} \{ I - T(h_1) - Bh_1^4 - Ch_1^6 \} = \frac{1}{h_2^4} \{ I - T(h_2) - Bh_2^4 - Ch_2^6 \}$$

or

$$I(h_1^2 - h_2^2) = h_1^2T(h_2) - h_2^2T(h_1) + B(h_1^2h_2^4 - h_2^2h_1^4) + \cdots$$

so that

$$I = \frac{h_1^2T(h_2) - h_2^2T(h_1)}{h_1^2 - h_2^2} - Bh_1^2h_2^2 - \cdots \quad (7.24)$$

Let

$$T(h_1, h_2) = \frac{h_1^2T(h_2) - h_2^2T(h_1)}{h_1^2 - h_2^2} \quad (7.25)$$

then

$$T(h_1, h_2) = \frac{T(h_2)(h_1^2 - h_2^2 + h_2^2) - h_2^2T(h_1)}{h_1^2 - h_2^2}$$

$$= \frac{T(h_2)(h_1^2 - h_2^2) + h_2^2T(h_2) - h_2^2T(h_1)}{h_1^2 - h_2^2}$$

$$= \frac{h_2^2}{h_1^2 - h_2^2} \{ T(h_2) - T(h_1) \} \quad (7.26)$$

and

$$I = T(h_1, h_2) - Bh_1^2h_2^2 - \cdots \quad (7.27)$$

Now, the error in $T(h_1)$ and $T(h_2)$ is dominated by $h^2$, whereas the error in $T(h_1, h_2)$ is dominated by $h^4$, and this improvement has been achieved at the cost of very little extra computation on top of the computation required for $T(h_1)$ and $T(h_2)$.

This process can be continued as follows:

Derive $T(h_i)$ using the trapezoidal rule, equation 7.2, for different values of the division $h_i, 1 \leq i \leq m$. Compute $T(h_i, h_{i+1})$ from each pair $T(h_i)$ and $T(h_{i+1}), 1 \leq i \leq (m-1)$, using equation 7.26. Then compute $T(h_i, h_{i+1}, h_{i+2})$ for each pair $T(h_i, h_{i+1})$ and $T(h_{i+1}, h_{i+2}), 1 \leq i \leq (m - 2)$, and so on. Romberg integration is just the procedure described above and tabulating the results as in figure 7.3.
Figure 7.3: Tabulation of values for Romberg integration

The values along a diagonal in the tableau get closer together, regardless of whether or not the values are correct, by virtue of equation 7.26. To assess the accuracy of the results, one needs therefore, to look at the values along a column.

Note that equation 7.26 requires very little computation apart from that required for evaluating \( T(h_1) \) and \( T(h_2) \). It follows that little computation is required to work out the values of the tableau after the first column. The values of column 1 are, of course, obtained using the trapezoidal rule.

By the trapezoidal rule we have:

\[
T(h_4) = h_4 \left\{ \frac{f(a)}{2} + f(a + h_4) + f(a + 2h_4) + \cdots + f(b)/2 \right\}
\]
\[
T(h_{i+1}) = h_i \left\{ \frac{f(a)}{2} + f(a + h_{i+1}) + f(a + 2h_{i+1}) + \cdots + f(b)/2 \right\}
\]

By making \( h_{i+1} = (1/2)h_i \), for \( i = 1, 2, \ldots, (m - 1) \), the amount of computation necessary to fill column 1 of the tableau is reduced considerably since all the values of \( f(x) \) obtained in the course of evaluating \( T(h_4) \) can now be used in evaluating \( T(h_{i+1}) \). In fact, the usual procedure is to choose \( h_1 = (b - a) \). The values in column 1 will then be:

\[
\begin{array}{cccc}
T(h_1) & T(h_1, h_2) & T(h_1, h_2, h_3) & T(h_1, h_2, h_3, h_4) \\
T(h_2) & T(h_2, h_3) & T(h_2, h_3, h_4) & \cdots \\
T(h_3) & T(h_3, h_4) & \cdots & \cdots \\
T(h_4) & \cdots & \cdots & \cdots \\
\vdots & \vdots & \vdots & \vdots \\
\end{array}
\]
\[ T(h_1) = T(\pi/2) = 0.7853982 \]
\[ 1.0022799 \]
\[ T(h_2) = T(\pi/4) = 0.9480595 \]
\[ 0.9999916 \]
\[ 1.0001346 \]
\[ 1.0000000 \]
\[ T(h_3) = T(\pi/8) = 0.9871158 \]
\[ 0.9999999 \]
\[ 1.0000083 \]
\[ 1.0000000 \]
\[ T(h_4) = T(\pi/16) = 0.9967852 \]
\[ 1.0000000 \]
\[ 1.0000005 \]
\[ T(h_1) = T(\pi/32) = 0.99919668 \]

Figure 7.4: Tableau for the function \( \sin x \) of the example

\[ T(h_1) = h_1 \{1/2(f(a) + f(b))\} \]
\[ = h_1 F_1, \]
where \( F_1 = 1/2(f(a) + f(b)) \)
\[ T(h_2) = (1/2)h_1 \{F_1 + f(a + h_1/2)\} \]
\[ = (1/2)h_1 F_2, \]
where \( F_2 = F_1 + f(a + h_1/2) \)

etc.

This point will be referred to later in the section on the implementation of Romberg integration.

Example As an example consider the integral

\[ I = \int_{0}^{\pi/2} \sin x \, dx \]

The result is \( I = 1 \), of course. Romberg integration with \( h_1 = \pi/2 \) and each subsequent \( h_i = h_{i-1}/2 \) leads to the tableau in figure 7.4

7.2.3 Double Integrals

As stated in the introduction, double and higher integrals are a lot more difficult than single integrals, and only proper double integrals over a rect-
The angular region will be evaluated here. The integrals considered are then of the form shown in equation 7.28

$$I = \int_A \int f(x, y) \, dA = \int_{y_0}^{y_m} \int_{x_0}^{x_n} f(x, y) \, dx \, dy$$  \hspace{1cm} (7.28)

so that the rectangular region $A$ is given by the lines $x = x_0, x = x_n, y = y_0$ and $y = y_m$. The integral can be evaluated as an *iterated* integral; that is, by integrating $f(x, y)$ with respect to $x$ treating $y$ as a constant, and then integrating the resulting function in $y$ with respect to $y$. See [55].

Figure 7.5 shows the region $A$ over which the integral is taken divided into a grid. The grid is formed by the intersection of the lines $x = x_0, x_1, \ldots, x_n$ and the lines $y = y_0, y_1, \ldots, y_m$. The $x = constant$ lines are equidistant, with the distance between them given by $(x_n - x_0)/n = h$. Similarly, the $y = constant$ lines are equidistant with $(y_m - y_0)/m = k$. (cf. figure 7.1.)

A formula for the numerical evaluation of $I$ can be obtained by applying the trapezoidal rule. In evaluating the single integral, the formula used was a weighted sum of the function values at certain points. In an analogous way to that of evaluating double integrals analytically, the inner integral is evaluated as weighted sums of function values, with each sum obtained by
keeping one variable constant. The double integral is then the weighted sum of these sums. Using the trapezoidal rule in both directions, and writing \( f_{i,j} \) for \( f(x + hi, y + kj) \), then

\[
I = \int_{y_0}^{y_m} \int_{x_0}^{x_n} f(x, y) \, dx \, dy = \frac{1}{2} k \left\{ \frac{1}{2} (f_{0,0} + 2f_{1,0} + \cdots + f_{n,0}) + 2 \times \frac{1}{2} (f_{0,1} + 2f_{1,1} + \cdots + f_{n,1}) + \cdots + 2 \times \frac{1}{2} (f_{0,m-1} + 2f_{1,m-1} + \cdots + f_{n,m-1}) + \frac{1}{2} (f_{0,m} + 2f_{1,m} + \cdots + f_{n,m}) \right\}
\]

or

\[
I = \frac{1}{4} hk (F_C + 2F_B + 4F_I)
\]

(7.29)

where

\[
\begin{align*}
F_C &= f_{0,0} + f_{n,0} + f_{0,m} + f_{n,m} \\
F_B &= \sum_{i=1}^{n-1} (f_{i,0} + f_{i,m}) + \sum_{j=1}^{m-1} (f_{0,j} + f_{n,j}) \\
F_I &= \sum_{i=1}^{n-1} \sum_{j=1}^{m-1} f_{i,j}
\end{align*}
\]

(7.30)

Equation 7.29 can also be obtained by writing a Lagrange polynomial for \( f(x, y) \) using the four values of \( f(x, y) \) on the corners of a square of the grid of figure 7.5, integrating the polynomial, and then summing up over the entire area of the grid.

Given the values of \( f \) at \((x_0, y_0), (x_0, y_1), (x_1, y_0), \) and \((x_1, y_1)\), the two dimensional Lagrange polynomial is given by (see [Gerald])

\[
P(x, y) = \frac{y - y_1}{y_0 - y_1} \left\{ \frac{x - x_1}{x_0 - x_1} f_{0,0} + \frac{x - x_0}{x_1 - x_0} f_{1,0} \right\} + \frac{y - y_0}{y_1 - y_0} \left\{ \frac{x - x_1}{x_0 - x_1} f_{0,1} + \frac{x - x_0}{x_1 - x_0} f_{1,1} \right\}
\]

So that

\[
\int_{y_0}^{y_m} \int_{x_0}^{x_n} f(x, y) \, dx \, dy \approx \int_{y_0}^{y_m} \int_{x_0}^{x_n} P(x, y) \, dx \, dy
\]

223
\[ \frac{1}{4}hk(f_{0,0} + f_{0,1} + f_{1,0} + f_{1,1}) \quad (7.31) \]

which is the analogue in two dimensions of equation 7.5. Summing up over the entire grid gives the formula of equation 7.29.

### 7.2.4 Romberg Integration for the Double Integral

We now proceed to derive the Romberg integration procedure for double integrals.

From equation 7.6

\[
\int_{x_0}^{x_n} dx \bigg|_{y=y_j} = \frac{1}{2}h(f_{0,j} + 2f_{1,j} + \ldots + 2f_{n-1,j} + f_{n,j}) + A_jh^2 + B_jh^4 + \ldots \quad (7.32)
\]

so that

\[
\int_{y_0}^{y_m} \int_{x_0}^{x_n} f(x, y) \, dx \, dy = \frac{1}{4}hk \sum_{i=0}^{n} \sum_{j=0}^{m} w_i w_j f_{i,j} + \frac{1}{2}k(A_0 + 2A_1 + \ldots + 2A_{m-1} + A_m)h^2 + \frac{1}{2}k(B_0 + 2B_1 + \ldots + 2B_{m-1} + B_m)h^4 + \bar{A}k^2 + \bar{B}k^4 + \ldots \quad (7.33)
\]

where \(w_i\) and \(w_j\) are the weights, see equation 7.29, \(\bar{A}\) and \(\bar{B}\) are the coefficients of the error term in the \(y\) direction. Each of the sums of \(A\)'s and \(B\)'s can be written as \(2n\) times some average value of \(A\) and \(B\), \(A_{av}\) and \(B_{av}\), say (see [34]), so that the error term is given by

\[
\text{error} = \frac{k}{2}(nA_{av})h^2 + \frac{k}{2}(nB_{av})h^4 + \bar{A}k^2 + \bar{B}k^4 + \ldots \quad (7.34)
\]

Let \(k = \alpha h\), then

\[
\text{error} = \left( \frac{x_n - x_0}{2} A_{av} \alpha \right) h^2 + \left( \frac{x_n - x_0}{2} B_{av} \right) h^4 + \bar{A} \alpha^2 h^2 + \bar{B} \alpha^4 h^4 + \ldots = K_1 h^2 + K_2 h^4 + \ldots \quad (7.35)
\]

224
and this is of the same form as equation 7.6, which shows the error in the numerical estimate of a single integral by the trapezoidal rule. The Romberg scheme can therefore be applied here as well, and two estimates can be combined to obtain another estimate with an error term of the order of $h^4$ as described in section 7.2.2 above. Note that whenever $h$ is halved, $k$ must also be halved. Quite clearly, the number of points at which the function must be evaluated increases rapidly. The function evaluations do not depend on each other and they can all proceed in parallel. In the following section, equations will be derived for the number of function evaluations required at each level.

### 7.3 Parallel Implementation of Romberg Integration

The algorithm is made up of three parts: finding the initial estimate, finding the estimates as the interval is successively halved, and producing the tableau. The first two parts require finding the function values at the appropriate points and producing the weighted sums.

#### 7.3.1 The Single Integral

Let $h$ be the initial interval, and $n$ the initial number of divisions. The first two parts of the algorithm are given in figures 7.6 and 7.7, for the integral

$$ \int_a^b f(x) \, dx $$

and the tabulation algorithm, being very straightforward, is omitted here.

Quite clearly, by far the largest proportion of the computation is taken up by the calculation of the function values at the various points along the interval of integration.
(1) \( \text{sum} = f(a) + f(b); \)
(2) \( x = a; \)
(3) \( \text{temp} = 0; \)
(4) for \( i=1 \) to \( n-1 \) {
(5) \( x = x + h; \)
(6) \( \text{temp} = \text{temp} + f(x); \)
(7) }
(8) \( \text{sum} = \text{sum} + 2 \times \text{temp}; \)
(9) \( \text{tab}[1][1] = 0.5 \times h \times \text{sum}; \)

Figure 7.6: Producing the initial estimate.

(10) for \( i=2 \) to \( l \) {
(11) \( h = h/2; \)
(12) \( n = n + n; \)
(13) \( \text{temp} = 0; \)
(14) \( x = a + h; \)
(15) for \( j=2 \) to \( n \), step 2 {
(16) \( \text{temp} = \text{temp} + f(x); \)
(17) \( x = x + h + h; \)
(18) }
(19) \( \text{sum} = \text{sum} + 2 \times \text{temp}; \)
(20) \( \text{tab}[j][1] = 0.5 \times h \times \text{sum}; \)
(21) }

Figure 7.7: Producing the successive estimates.
Let $e_i$ be the number of times the function $f$ has to be evaluated to produce the estimate $T(h_i)$. Then $e_1 = n + 1$, $e_2 = e_1 + n$, ..., and in general

$$
\begin{align*}
  e_1 &= n + 1 \\
  e_i &= e_{i-1} + 2^{i-2}n, \quad i = 2, \ldots, \ell
\end{align*}
$$

(7.36)

Since the pairs $T(h_i)$ and $T(h_{i+1})$ are chosen such that $h_{i+1} = h_i/2$, and letting $e'_i$ stand for $e_i - e_{i-1}$, we have

$$
e'_i = 2^{i-2}n, \quad i = 2, \ldots, \ell
$$

(7.37)

so that at each successive halving of the interval, the number of iterations of the inner loop, lines (15)-(18), is doubled.

In this algorithm, every function evaluation is independent of every other function evaluation, and the algorithm therefore has a very high degree of parallelism. The difficulty is in keeping track of where to add the function values. For, observe that the first loop, lines (4)-(7), can be re-written so that its iterations are not dependent on $x$, thus

```
sum = f(a) + f(b);
temp = 0;
for i=1 to n-1 {
  x = a + (i-1) * h;
  temp = temp + f(x);
}
```

Similarly, the second loop, lines (10)-(21), including the inner loop, lines (15)-(18), can be written so that not one of the iterations depends on $h$ or $n$, as follows:

```
for i=2 to l {
  temp = 0;
  for j=2 to n*2^{(i-1)}, step 2 {
    x = a + (2j-3)*h/2^{(i-1)};
  }
  sum = sum + 2 * temp;
  tab[i][j] = 0.5 * sum * h/2^{(i-1)};
}
```
Whilst this certainly removes the dependency on $h$ and $n$, it adds quite significantly to the calculations required, and it is not proposed to implement the algorithm in this way. This is only to show that all the function evaluations can be carried out independently of each other.

Before considering how to partition the loops, it is helpful to get a clearer picture of the way the function evaluations proceed. Consider the case when $h_1 = b - a$ (and $n = 1$). From equations 7.36 and 7.37, $T(h_1)$ requires two function evaluations, $f(a)$ and $f(b)$. Consider then how many extra points the function has to be evaluated as one goes from $T(h_i)$ to $T(h_{i+1})$. To evaluate $T(h_2)$, where $h_2 = h_1/2$, $f(x)$ has to be evaluated in addition at the point $x = a + h_1/2$. To obtain $T(h_3)$, $f(x)$ must be further evaluated at the two points $x = a + h_1/4$ and $x = a + 3h_1/4$, and so on. Let the set $X_i$ be the ordered set of all points $x$ that are needed to evaluate $T(h_i)$. Then $X_1 = \{a, b\}$. Denote by $y_i$ the set of all pairs of adjacent points in $X_i$, e.g. $y_1 = \{(a, b)\}$. It can be seen that $X_i, i \geq 2$, is the set $\{x : x$ lies midway between all pairs in $y_{i-1}\} \cup X_{i-1}$. Let $X_i = X_i - X_{i-1}$, then $X_i$ is the set of all values of $x$ at which the function $f$ has to be evaluated to obtain $T(h_i)$ in addition to the values of $f$ required for evaluating $T(h_{i-1})$. This is shown in figure 7.8.
If one looks at figure 7.8 from \( X_2 \) down and ignores, for now, \( X_1 \), then the figure has the same structure as a binary tree. At each level, the extra number of points at which \( f(x) \) has to be evaluated is double that of the previous level. And this structure agrees with equation 7.36 and 7.37.

In order to allow the partitioning of the algorithm, the variable temp can be turned into a two dimensional array, part\_sums of order \( \ell \times w \), where \( \ell \) is the number of levels, that is the number of estimates in the first column of the tableau, and \( w \) is the number of processors used. The array is then shared by all the processors, with each processor writing its result, or partial sum, for each level it operates on, in its own column of the array, so that there is no conflict when more than one processor is writing to the array.

The whole of the tabulation process, including the production of the first column of the tableau, lines (8), (9), (19) and (20) of the algorithm can be done after all the function evaluations have been carried out.

Consider now the partitioning of the algorithm. Note first that the two loops, that is the two parts of the algorithm shown in figures 7.6 and 7.7, are independent of each other, so that both can be executed concurrently. Second, the amount of work in the second loop is far greater than that in the first. Consider then the partitioning of the second loop. It is possible to partition this nested loop at either the outer or the inner level. However, partitioning the outer loop would lead to uneven load balancing, because for iteration \( i \) of the outer loop, the inner loop has \( 2^{(i-2)}n \) iterations. Let the number of iterations of the inner loop for iteration \( i \) of the outer loop be \( j_i \), then it can be seen that

\[
j_i = \sum_{k=2}^{i-1} j_k + n \tag{7.38}
\]

which is obtained by summing terms given by equation 7.37 above, where, clearly, \( c'_i = j_i \). This implies that partitioning the outer loop would produce, at best, a speedup of 2 no matter how many processors were used.
Clearly then, the inner loop should be considered. Partitioning the inner loop is a straightforward matter. The iterations \( j_i \) are distributed amongst the available processors. In such a scheme, the outer loop would be executed sequentially, and the inner loop concurrently. Given DOACROSS type structures for parallel loop execution, this scheme is the obvious one to choose. It has however two drawbacks. The first is that as the iterations of the outer loop are successively executed the number of iterations of the second loop handed out to each processor is successively doubled. If, for example, \( n = 10 \) and \( \ell = 7 \), then \( j_2 = 10, j_3 = 20, j_4 = 40, j_5 = 80, j_6 = 160, \) and \( j_7 = 320 \). It follows that the granularity might be too small for the first few iterations of the outer loop to make the parallelization of the inner loop worthwhile.

The second drawback is this: both parts of the algorithm, loop (4)–(7) and loop (10)–(21), can be executed concurrently. If function partitioning is used to achieve this, then the processor that executes the first part should, after it finishes this task, join the other processors in executing the second part. This requires that the process which partitions the inner loop be made aware of when the first task has finished and to start allocating the iterations taking into account the extra available processor. Alternatively, the two main parts of the algorithm can be executed one after the other, with the first loop partitioned, and the second loop as described above. This would again lead to the problem of the granularity being too fine in the first task as well as in the first few iterations of the second.

These drawbacks can be avoided altogether using exdata by turning the two parts of the algorithm into two independent main tasks. The second loop can be partitioned at both the outer and the inner level using a suitable sub-manager. Given a number of iterations, \( g \), say, as a suitable grain size, the sub-manager keeps track of \( i \) and can thus work out \( j_i \). When a request for a sub-task is received, the sub-manager hands back \( i \) and the range of iterations of the inner loop to be performed; if \( j_i > g \), then \( g \) iterations are
returned. In this way the loop is partitioned at both levels simultaneously. Further, the processor that is allocated the first task will, upon completion of this, be given the second task and will thus join the other processors in the execution of the second loop. The sub-manager need not be aware of when this takes place, for it is the processor itself that makes the request.

The extra calculations required to make each iteration of the original algorithm independent of every other iteration are now reduced considerably. The first loop is not parallelized, and it is only when a request is made to the sub-manager that the expensive operation of raising 2 to the power \((i - 1)\) need be made, and this can be controlled by the grain size \(g\).

An outline of the sub-manager for partitioning the second task is shown in figure 7.9. Figure 7.10 shows an outline of the main task for implementing the second task of the integration (figure 7.7). The full program is given in the appendix. Note the use of the array variable `part_sums[][]` for holding the intermediate results.

Producing the tableau requires the adding up of the values in the array `part_sums[][]` and multiplying by half the size of the interval to give the first column of the tableau; the rest of the tableau is obtained by repeatedly applying equation 7.26.

The number of elements in each successive column of the tableau is one less than that of the preceding column. The tableau itself is implemented in the same way as upper triangular matrices were in chapter 6.

The amount of work required for producing the tableau is very small, as can be readily seen by considering that the number of elements in the first column, \(\ell\), is very small. If \(n\) is initially 10, say, then with \(\ell = 16\), \(h_{16} = (b-a)/327,680\). The columns of the tableau have to be produced sequentially because each column depends on the previous column. Parallelization of the tabulation procedure need not therefore be considered. Table 7.1 shows the time taken producing the partial sums and for producing the tableau. The
Sub_Man rom_man() {
    if (i > l) stop;
    else {
        \[ W_{\text{ARG}.4} = 2^{(i-1)}; \]
        \[ nl = n \times W_{\text{ARG}.4}; \]
        \[ W_{\text{ARG}.1} = i; \]
        \[ W_{\text{ARG}.2} = j; \]
        \[ j = j + g; \]
        if (j \geq nl) {
            \[ W_{\text{ARG}.3} = nl; \]
            \[ j = 2; \]
            \[ l = l + 1; \]
        }
        else {
            \[ W_{\text{ARG}.3} = j; \]
            \[ j = j + 2; \]
        }
    }
}

Figure 7.9: Outline of the sub-manager for partitioning the nested loop of figure 7.7.

Main_Task sec_loop() {
    temp = 0;
    hl = h/W_{\text{ARG}.4};
    nl = n \times W_{\text{ARG}.4};
    x = a + (W_{\text{ARG}.2} - 1) \times hl;
    for i = W_{\text{ARG}.2} to W_{\text{ARG}.4}, step 2 {
        temp = temp + f(x);
        x = x + hl + hl;
    }
    part_sums[W_{\text{ARG}.1}][m.get_myid() + 1] += temp + temp;
}

Figure 7.10: Outline of the main task for implementing figure 7.7.
<table>
<thead>
<tr>
<th>$\ell$</th>
<th>Partial Sums</th>
<th>Tabulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.02</td>
<td>0.00</td>
</tr>
<tr>
<td>5</td>
<td>0.15</td>
<td>0.01</td>
</tr>
<tr>
<td>10</td>
<td>4.78</td>
<td>0.01</td>
</tr>
<tr>
<td>15</td>
<td>152.29</td>
<td>0.01</td>
</tr>
</tbody>
</table>

Table 7.1: Times, in seconds, for the two parts of Romberg integration, for the integral \( \int_0^1 (0.92 \cosh x - \cos x) \, dx \) with \( n = 10 \).

<table>
<thead>
<tr>
<th>No. Procs.</th>
<th>Sequential Time</th>
<th>Parallel Times</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>47.60</td>
<td>47.99</td>
<td>0.99</td>
</tr>
<tr>
<td>2</td>
<td>24.02</td>
<td>1.98</td>
<td>12.07</td>
</tr>
<tr>
<td>3</td>
<td>16.13</td>
<td>2.95</td>
<td>5.45</td>
</tr>
<tr>
<td>4</td>
<td>12.15</td>
<td>3.92</td>
<td>3.09</td>
</tr>
<tr>
<td>5</td>
<td>9.71</td>
<td>4.90</td>
<td>1.96</td>
</tr>
<tr>
<td>6</td>
<td>8.14</td>
<td>5.85</td>
<td>1.39</td>
</tr>
<tr>
<td>7</td>
<td>6.99</td>
<td>6.81</td>
<td>1.00</td>
</tr>
<tr>
<td>8</td>
<td>6.13</td>
<td>7.77</td>
<td>1.27</td>
</tr>
<tr>
<td>9</td>
<td>5.47</td>
<td>8.71</td>
<td>1.57</td>
</tr>
<tr>
<td>10</td>
<td>4.95</td>
<td>9.62</td>
<td>1.94</td>
</tr>
<tr>
<td>11</td>
<td>4.52</td>
<td>10.53</td>
<td>2.33</td>
</tr>
</tbody>
</table>

Table 7.2: Times, in seconds, and speedups for \( \ell = 10 \) and \( n = 10 \).

latter is, quite clearly, insignificant.

The integral
\[
\int_0^1 (0.92 \cosh x - \cos x) \, dx
\]
whose value is 0.2397141... (this is used as an example in [Fox]) was evaluated using a parallel program partitioned in the way described above. Table 7.2 gives the times and the speedup obtained. The speedup is plotted in figure 7.11. The number of levels used was 10, and the number of iterations of the inner loop was chosen, arbitrarily, to be 200. As might be expected from such a potentially highly parallel algorithm, the speedups achieved are very close to the ideal.
7.3.2 Algorithm for the Double Integral

Let $n$ and $m$ stand for the number of divisions in the $x$ and $y$ directions respectively, and let $h_1$ and $k_1$ be the corresponding intervals, i.e. $h_1 = (x_n - x_0)/n_1$ and $k_1 = (y_m - y_0)/m_1$. Then the algorithm for evaluating

$$\int_{x_0}^{x_n} \int_{y_0}^{y_m} f(x, y) \, dx \, dy$$

is made up, as before, of three parts. Finding the initial estimate, successively halving $h_1$ and $k_1$, and doubling $n_1$ and $m_1$, and adding the weighted sum of the function values at the extra points thus created to get successive estimates of the integral. And finally combining these estimates and tabulating the results. Each of the first two parts will be considered separately because they are more complicated than for the single integral. The tabulation, on the other hand, is the same as before.

From equations 7.29 and 7.30, the initial estimate $T_1$ is given by

$$T_1 = \frac{1}{4}hk\{f_{0,0} + f_{n,0} + f_{0,m} + f_{n,m}\}$$

234
where \( f_{i,j} \) stands for \( f(x_0 + ih_1, y_0 + jk_1) \). This can be broken down neatly into four functions, one for finding the sum of the function values at the four corners of the grid; two for finding the sum of the function values at the four borders of the grid, one for the top and bottom borders, and the other for the left and right borders; and finally one for finding the sum of the function values at the internal points. Let the number of function evaluations required to produce \( T_1 \) be \( d_1 \), then

\[
d_1 = n_1m_1 + n_1 + m_1 + 1
\]  

(7.40)

Figure 7.12 shows a \( 5 \times 5 \) grid further subdivided to become a \( 10 \times 10 \) grid. (The \( 5 \times 5 \) grid is shown in solid lines, and the subdivided grid is shown using dashed lines.) The new points created by the subdivision are marked out with small circles. The situation with regard to the new points...
on the borders is similar to that when the interval is subdivided in the single integral. With regard to the internal points the situation is different. Observe that along each of the lines \( x = x_1, \ x = x_3, \ldots, \ x = x_8 \), there are 9 new points at the intersection with the lines \( y = y_1, \ y = y_2, \ldots, \ y = y_9 \); but that along each of the lines \( x = x_2, \ x = x_4, \ldots, \ x = x_8 \), there are just 5 new points at the intersection with the lines \( y = y_1, \ y = y_2, \ldots, \ y = y_9 \). It can be seen then that

\[
T_q = T_{q-1} + \frac{1}{4} h_1 k_q \left\{ 2 \sum_{i=1}^{n_q/2} (f_{2i-1,0} + f_{2i-1,m_q}) + 2 \sum_{j=1}^{m_q} (f_{0,2j-1} + f_{n_q,2j-1}) + 4 \sum_{i=1}^{n_q/2} \sum_{j=1}^{m_q-1} f_{2i-1,j} + 4 \sum_{i=1}^{n_q/2} \sum_{j=1}^{m_q/2} f_{2i,2j-1} \right\} \tag{7.41}
\]

This, too, can be broken down into four functions. Two for evaluating the function on the borders of the region and two for the internal points. The number of function evaluations required for \( T_q \) is then

\[
d_q = d_{q-1} + \frac{3}{4} m_q n_q + \frac{1}{2} n_q + \frac{1}{2} m_q \tag{7.42}
\]

Let \( d'_q \) stand for \( d_q - d_{q-1} \). Since \( n_q = 2n_{q-1} \) and \( m_q = 2m_{q-1} \), it follows that

\[
d'_q = 3m_{q-1}n_{q-1} + n_{q-1} + m_{q-1} = 3d_{q-1} - 2(n_{q-1} + m_{q-1}) - 3 \tag{7.43}
\]

This is analogous to equation 7.37 of the single integral. A binary tree was used to illustrate the spread of the points along the interval as the number of divisions was successively doubled for the single integral. The author does not know of an analogous structure to illustrate the spread of the points in the grid for the double integral, but one presumes that equation 7.43 gives the relationship between the number of leaves and the number of all their
find the initial estimate

(1) \[ \text{sum} = f(x_0, y_0) + f(x_n, y_0) + f(x_0, y_m) + f(x_n, y_m); \]

(2) \[ \text{sum} = \text{sum} + 2 \cdot \sum_{i=1}^{n-1} \{f(x_i + ih, y_0) + f(x_i + ih, y_m)}; \]

(3) \[ \text{sum} = \text{sum} + 2 \cdot \sum_{j=1}^{m-1} \{f(x_0, y_0 + jk) + f(x_n, y_0 + jk)}; \]

(4) \[ \text{sum} = \text{sum} + 4 \cdot \sum_{i=1}^{n-1} \sum_{j=1}^{m-1} f(x_0 + ih, y_0 + jk); \]

(5) \[ \text{tab}[1][1] = .25 \cdot h \cdot k \cdot \text{sum}; \]

Figure 7.13: Producing the initial estimate for a double integral.

predecessors (cf. a binary tree where the number of leaves is equal to the total number of all the other nodes plus 1).

The parallelization of the algorithm is carried out by partitioning the algorithm into concurrent functions and further partitioning the data for some of these functions. The problem to be solved, as in the case of the single integral, is one of partitioning nested loops at different levels. From equations 7.39 and 7.41, the algorithm can be written as in figures 7.13 and 7.14.

The independent functions into which the algorithm is partitioned correspond to lines (1) and (2), line (4), line (9), line (10), line (11) and line (12). The nested loop of line (4) is partitioned at the outer level. Each of the double nested loops of lines (11) and (12) are partitioned at two levels, the outermost and the middle. The algorithm requires numerous function evaluations, all of which, with a suitable choice of data structures, can be carried out concurrently. The above partitioning is rather arbitrary, and no particular attention is given to load balancing; for example, the innermost loop of line (11) contains twice as many iterations as the innermost loop of line (12). However, the sheer amount of work required should alone produce enough tasks to keep the processors busy, and, indeed, the results given below
successively halve the intervals and produce the appropriate estimate

(6) for $q=2$ to $l$

(7) $h = h/2; k = k/2$;

(8) $n = n+n; m = m+m$;

(9) $\text{sum} = \text{sum} + 2 \times \sum_{i=1}^{n/2} f(x_0+(2i-1)h, y_0) + f(x_0+(2i-1)h, y_m)$;

(10) $\text{sum} = \text{sum} + 2 \times \sum_{j=1}^{m/2} f(x_0, y_0+(2j-1)k) + f(x_n+(2i-1)h, y_0+(2j-1)k)$;

(11) $\text{sum} = \text{sum} + 4 \times \sum_{i=1}^{n/2} \sum_{j=1}^{m-1} f(x_0+(2i-1)h, y_0+jk)$;

(12) $\text{sum} = \text{sum} + 4 \times \sum_{i=1}^{(n/2)-1} \sum_{j=1}^{m/2} f(x_0+2ih, y_0+(2j-1)k)$;

(13) $\text{tab}[q][l] = 0.25 \times h \times k \times \text{sum}$;

(14) }

Figure 7.14: Producing successive estimates of a double integral.

show speedups that are close to the ideal.

Before giving the results, the data structures used are briefly mentioned. The same strategy is used here as for the single integral, whereby a two dimensional array is used. The number of rows is the number of levels used in the tableau, and the number of columns is the number of processors used in the running of the program. Each processor is designated a column in the array wherein to place the sums it has accumulated.

As a sub-manager for partitioning a double loop at both levels has already been outlined in the previous section, the sub-managers used here will not be given. The full program is included in the appendix.

Table 7.3 gives the times for evaluating the double integral

$$\int_0^1 \int_0^1 \sin x \cos x \, dx \, dy$$

with $n_1 = m_1 = 10$ and $\ell = 6$, and the speedup figures. These figures are plotted in figure 7.15.

There are two aspects of the parallelization of the algorithm that are
<table>
<thead>
<tr>
<th>No. Procs.</th>
<th>Sequential Time</th>
<th>Parallel Times</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>59.73</td>
<td>60.24</td>
<td>0.99</td>
</tr>
<tr>
<td>2</td>
<td>30.31</td>
<td>1.97</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>20.16</td>
<td>2.96</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>15.17</td>
<td>3.94</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>12.17</td>
<td>4.91</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>10.16</td>
<td>5.88</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8.73</td>
<td>6.84</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>7.68</td>
<td>7.78</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>6.81</td>
<td>8.77</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>6.17</td>
<td>9.68</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>5.64</td>
<td>10.59</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.3: Times, in seconds, and speedup for evaluating a double integral.

Figure 7.15: Plot of the speedup figures from table 7.3.
remarkable here. The first is that all the functions into which the algorithm is partitioned are run concurrently and at the same time some of these are further parallelized using data partitioning. The second is that some loops are partitioned at two levels simultaneously, and, indeed, the technique can be extended to an arbitrarily nested loop. This will be taken up and discussed further in chapter 8.
Chapter 8

Data and Function
Partitioning, and Scheduling

8.1 Introduction

In the previous two chapters examples of parallel programs were given and implemented for the purpose of illustrating how exdata can be used to facilitate parallel programming, and to compare the performance achieved with exdata with that of similar programs written using the parallel library routines.

It was stated in chapter 2 that it would be profitable not to assume any particular paradigms for parallel programming when designing a tool for facilitating it; rather, to allow the programmer complete freedom in choosing how to partition a program, provided it was possible to dissociate the partitioning process from the logic of the underlying algorithm. If any paradigms are found, then they can be easily recognized and isolated, and implemented separately, so that they can be used in different programs. This approach also allows the programmer to experiment with different strategies without the need for writing a new program for each strategy.
In chapter 3 a parallel program was defined as a set of tasks with a schedule.

The discussion that followed in chapter 4 then led to exdata as a tool for writing parallel programs which conform with the two notions mentioned above, namely explicit partitioning and the definition of a parallel program of chapter 3.

The programs presented in chapter 6 were given, as mentioned above, to show that the tool works, and its performance is at least comparable with that of programs written using the available parallel library subroutines on the Sequent computer. The scheduling employed then was quite straightforward. The Romberg integration program of chapter 7 required scheduling that was less straightforward, and the aim was to show the usefulness of explicit data assignment for partitioning nested loops.

It must be stressed here that it is not claimed that exdata can do what cannot be done by other means. Indeed, the idea that no paradigms should be presumed is consistent with the philosophy behind Sequent's parallel library subroutines, where, again, no paradigms are presumed, and a collection of routines for the most basic operations required for parallel programming only is provided. Further, exdata itself is written with the help of the parallel library subroutines, but, as was pointed out in chapter 4, it might have been written without them. What is being claimed is that explicit assignment of functions and data provides a way for paradigms to evolve naturally out of the requirements of the programmer instead of being, as it were, forced. Also, that having a definition of a parallel program from the software point of view enables the design and implementation of tools for writing parallel programs for any machine with the same user interface.

In this chapter, the solution of the block diagonal bordered system of linear equations will be considered again, but this time it will be implemented for systems where the block sizes are not all the same. For such a system,
the scheduling used in chapter 6, static and mixed, will be seen to be inadequate, and priority scheduling will be used and implemented. Next, to improve the speedup, a new scheduling strategy with partial dependence will be introduced and implemented, and it will be shown how \textit{exdata} and the two basic ideas behind it lead to a natural implementation of this scheduling. This implementation will be compared with another using the parallel library routines.

Finally in this chapter a general method for partitioning arbitrary nested loops will be described and implemented. Some consideration will also be given to the rôle of the sub-managers as explicit loop control statements and their possible further development.

### 8.2 Scheduling Different Sized Tasks

The problem of scheduling independent tasks (or sub-tasks) to run concurrently over a number of processors such that the execution time is minimized is an \textit{NP}-complete problem \cite{32}. For our purpose here, it is sufficient to say that this means the only "solution" to the scheduling problem is by exhaustion, i.e. to try out all possible combinations of the tasks and processors in order to see which of them minimizes the execution time. (The reference above can be consulted for formal definitions and proofs.) To solve the scheduling problem heuristics have to be used.

In chapter 6 the terms \textit{static} and \textit{dynamic} scheduling were used to describe the ways in which the sub-tasks were allocated to concurrent processors. These terms are widely used, including, for example, in the Sequent manual \cite{73}, and their usage here is in accordance with their usage there. So that static scheduling has meant that the sub-tasks are handed out in blocks whose size is independent of the number of processors being used. And dynamic scheduling has meant that the sub-tasks are grouped into blocks whose size
depends on the number of processors being used and handed out in blocks. Mixed scheduling was then devised from these two ideas to improve the speedup.

Also in chapter 6 (section 6.5.2) the distribution of the sub-tasks in the programs using the parallel library was described as deterministic because the sub-tasks executed by each processor is determined by the processor’s id number. Using exdata the distribution of the sub-tasks was nondeterministic because it depended on the order in which requests from the workers reached the sub-managers, with all the workers identical to each other. Note that in the literature, the term static scheduling is sometimes used to refer to what has been called here deterministic scheduling, for example in [70].

When the sub-tasks are all of the same size, it does not matter in which order they are executed or which processor executes which sub-tasks, and nondeterministic scheduling satisfies load balancing requirements better than deterministic scheduling because the latter does not allow for the effects on the execution time of interrupts, swaps, etc. When the sub-tasks are not all the same size, however, nondeterministic scheduling can lead to a load distribution that is far from the optimum. For example, consider a list $\mathcal{S}$ of independent tasks, $\mathcal{S} = \{T_1, T_2, \ldots, T_5\}$, with sizes $s_{T_1} = 1$, $s_{T_2} = 5$, $s_{T_3} = 2$, $s_{T_4} = 3$, $s_{T_5} = 6$, and two processors, $p_1$ and $p_2$. Say $p_1$ is allocated $T_1$, then $p_2$ will get $T_2$. $p_1$ will then execute $T_3$ and start executing $T_4$ while $p_2$ is still executing $T_2$. When $p_2$ finishes $T_2$, $p_1$ will still be executing $T_4$, and so $p_1$ gets $T_5$. The result is that $p_1$ will execute $T_1$, $T_3$ and $T_4$, whose total size is 6, and $p_2$ will execute $T_2$ and $T_5$, whose total size is 11. Clearly some order must be imposed on the allocation of the tasks, if sensible load balancing is to be achieved.

While the allocation of sub-tasks to the workers, or processors, in exdata is nondeterministic in the sense that it is not known before the program executes which processor will receive which sub-task, the order in which the
sub-tasks are handed out is determined by the main task's sub-manager, and this order is therefore entirely in the hands of the user who writes the sub-managers. The rest of this section will be concerned with how to schedule sub-tasks of various sizes to achieve load balancing under \textit{exdata} with the system of block diagonal bordered equations. But before going on, it is important to point out that the concern here is with how the scheduling can be implemented rather than with the scheduling strategy itself. As pointed out earlier, the problem to be solved is \textit{NP}-complete; the subject of \textit{NP}- completeness is not an easy subject and the whole field of its study depends on an assertion which to this day has not been proved correct, nor has it been proved wrong. A simple heuristic will be used for the scheduling strategy as will be seen below.

### 8.2.1 Priority Scheduling

Consider the first loop of algorithm 6, chapter 6, which performs part of the elimination process for the Gaussian elimination for the system of block diagonal bordered equations. This loop is given here as Algorithm 1.

(1) for \(i = 1, 2, \ldots, N\) in parallel  
(2) \(D_i = A_i^{-1}\)  
(3) \(B_i = D_i \times B_i\)  
(4) \(d_i = D_i \times d_i\)  
(5) \(D_i = C_i \times B_i\)  
(6) \(e_i = C_i \times d_i\)  
(7) endfor

Algorithm 1: First loop of Gaussian elimination algorithm

As seen in chapter 6, the execution time for each iteration is \(O(n_i^2m)\) where \(n_i \times n_i\) is the order of the square diagonal block \(A_i\), and \(n_i \times m\) is the
order of the right border block $B_i$, with $m \times n_i$ the order of the lower border block $C_i$. With variable size blocks then, the time taken for each iteration depends on $n_i$ since $m$ is the same for all iterations.

The iterations must be scheduled so that the execution time is minimized for any number of processors used. A simple scheduling strategy will be used. The iterations will be handed out to the processors according to the time it takes for each of them to be executed, with the iteration that takes the longest time handed out first, and so on in descending order of duration. This, of course, does not guarantee optimum load distribution, but it does tend to make the discrepancy between the loads on different processors comparable to the size of the smallest iterations (those with the shortest durations). An advantage of this strategy is that it does not depend on the number of processors, so that it is possible to work out the order of allocation of the sub-tasks once only. (In practice, this would require that this be done at the compiling stage of the program.)

In exdata such scheduling is straightforward to implement. All that is needed is for the $n_i$'s, $i = 1, \ldots, N$, to be sorted in descending order, and then for the sub-managers to hand out the iterations in the sorted order to the workers. For this purpose an array $\text{index.array}[]$ can be used to store the values of $n_i$, and the array is then sorted. The sub-manager will be identical to that used earlier (figure 6.5) with the only difference being that when requested for work, it hands out the iteration $\text{index.array}[\text{loop.index}]$ instead of $\text{loop.index}$ as before. The sub-manager thus becomes as shown in figure 8.1.

The use of a data structure, in this case an array, to hold the iteration number, effectively the sub-tasks, thus allows priority in scheduling. Any criteria can be used to establish the priority, and the sub-manager would still be the same. What is more, the sub-manager is completely separate from the code of the main task, which therefore need not change at all regardless of the
Sub_Man var_size.m (TASK)
smarg
{
  ismarg
    if (MORE_W) {
      STOP_GO = 1;
      W_ARG_1 = W_ARG_2 = index_array[loop_index];
      if (++loop_index > N)
        MORE_W = 0;
    }
    else
      STOP_GO = 0;
    return JOB;
}

Figure 8.1: Sub-Manager for allocating tasks from an ordered array of iterations.

sub-manager and its activities. This is true for whatever scheduling strategy
used, and this is the point to be emphasized here. Even had the scheduling
strategy demanded that the allocation of the sub-tasks be deterministic, for
whatever reason, then this can be accommodated easily. For, recall that a
sub-manager is just a subroutine called by a worker to request data before
executing a main task handed to it by the manager; it follows that the id
of a sub-manager, as seen by the system, is the same as that of its worker,
since, as far as the system is concerned, manager, sub-manager, worker, etc.
are all part of the system process, so that it is perfectly legitimate to use
the id number of a sub-manager to determine which sub-task to allocate. It
has to be added, though, that on a shared memory machine with identical
processors, it is very unlikely that such a requirement will arise.

The same can be done for the implementation of Gaussian elimination with
the parallel library routines. The iteration numbers are stored in an array
and sorted prior to the execution of the loop, and the loop control statement
picks out the next available iteration from the array. Effectively, the loop
shared loop_index = 0;
...
void iterate(...);
...
{
  int i;
  lock;
  i = ++loop_index;
  unlock;
  while (i <= N) {
    do iteration index.array[i];
    lock;
    i = ++loop_index;
    unlock;
  }
  return;
}

Figure 8.2: Allocating tasks from an ordered array of iterations using the parallel library routines.

The parallel Gaussian elimination algorithm (algorithm 6, chapter 6) contains two loops that can be processed concurrently; the first of these two was
given earlier as algorithm 1, and the second is given below as algorithm 2.

(1) for $i = 1, 2, \ldots, N$ in parallel
(2) \[ x_i = d_i - B_i \times x_{N+1} \]
(3) endfor

Algorithm 2: Second loop of Gaussian elimination algorithm

As with the first loop, the amount of work in each iteration depends on $n_i$, where $n_i \times m$ is the order of the block $B_i$, and to partition this loop and allocate the iterations the various processors, exactly the same procedure is followed as that for the first loop. Namely, the sorted array $\text{index\_loop}[\cdot]$ is used to hand out the iterations to the processors.

Example To solve the block diagonal bordered system shown in equation 8.1

\[
\begin{pmatrix}
A_1 & 0 & B_1 \\
A_2 & B_2 \\
A_3 & B_3 \\
0 & \ddots & \vdots \\
& A_N & B_N \\
C_1 & C_2 & C_3 & \cdots & C_N & A_{N+1}
\end{pmatrix}
\begin{pmatrix}
x_1 \\
x_2 \\
x_3 \\
\vdots \\
x_N
\end{pmatrix}
= 
\begin{pmatrix}
d_1 \\
d_2 \\
d_3 \\
\vdots \\
d_N
\end{pmatrix}
\]

(8.1)

where $A_i$, $1 \leq i \leq N$, is an $n_i \times n_i$ square matrix; $B_i$, $1 \leq i \leq N$, is an $n_i \times m$ matrix; and $C_i$, $1 \leq i \leq N$, is an $m \times n_i$ matrix; $d_i$ and $x_i$, $1 \leq i \leq N$, are vectors of order $n_i$; $A_{N+1}$ is an $m \times m$ matrix, and $x_{N+1}$ and $d_{N+1}$ are vectors of order $m$.

Let $N = 20$, $m = 30$, and $n_i$, be as given in equations 8.2 below:

\[
\begin{align*}
 n_1 &= n_5 = n_9 = n_{13} = n_{17} = 10 \\
n_2 &= n_6 = n_{10} = n_{14} = n_{18} = 15 \\
n_3 &= n_7 = n_{11} = n_{15} = n_{19} = 20 \\
n_4 &= n_8 = n_{12} = n_{16} = n_{20} = 25
\end{align*}
\]

(8.2)
Table 8.1: Timing results for Gaussian elimination for the system of equations 8.2. Times are in seconds.

The optimum scheduling of the blocks for the two parallel loops would require a different schedule according to the number of processors to be used. Here, no such scheduling is attempted; the blocks are handed out simply in the order of their size. This is implemented using \textit{exdata}, as \textit{EX..PRS}, and the parallel library routines, as \textit{LIB..PRS}, and, for comparison, the system is also solved using the library routines with static scheduling (\textit{LIB..ST} of chapter 6) and \textit{exdata} also with static scheduling (\textit{EX..ST}, chapter 6). The timing figures for these programs are given in table 8.1, and the speedup figures corresponding to them are given in table 8.2. The speedup figures are plotted, and their graph is shown in figure 8.3.

The erratic path followed by the speedup curve for the \textit{LIB..ST} program is expected. Using \( n_i^3 \) as a guide to the time taken for the execution of the body of the loop shown in figure 8.1 above, table 8.3 shows the distribution of the iterations among the processors, when \( p \), the number of processors, is 5, and when \( p \) is 8. At \( p = 5 \), the speedup achieved with \textit{EX..ST} is the best, because, as seen in table 8.3, the distribution of the iterations is optimal, due purely to the way in which the numbers in this example were chosen. At \( p = 8 \) the
Table 8.2: Speedup figures corresponding to the timing figures of table 8.1.

<table>
<thead>
<tr>
<th>N Procs</th>
<th>LIB_ST</th>
<th>EX_ST</th>
<th>LIBPRS</th>
<th>EX_PRS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>1.59</td>
<td>1.80</td>
<td>1.84</td>
<td>1.85</td>
</tr>
<tr>
<td>3</td>
<td>2.51</td>
<td>2.37</td>
<td>2.44</td>
<td>2.58</td>
</tr>
<tr>
<td>4</td>
<td>2.18</td>
<td>2.93</td>
<td>3.06</td>
<td>3.21</td>
</tr>
<tr>
<td>5</td>
<td>4.02</td>
<td>3.28</td>
<td>3.42</td>
<td>3.82</td>
</tr>
<tr>
<td>6</td>
<td>3.47</td>
<td>3.72</td>
<td>3.82</td>
<td>4.23</td>
</tr>
<tr>
<td>7</td>
<td>4.37</td>
<td>4.04</td>
<td>4.12</td>
<td>4.68</td>
</tr>
<tr>
<td>8</td>
<td>3.28</td>
<td>4.31</td>
<td>4.45</td>
<td>5.08</td>
</tr>
<tr>
<td>9</td>
<td>4.20</td>
<td>4.49</td>
<td>4.53</td>
<td>5.48</td>
</tr>
<tr>
<td>10</td>
<td>5.41</td>
<td>4.80</td>
<td>4.72</td>
<td>5.78</td>
</tr>
</tbody>
</table>

Figure 8.3: Plot of Speedup vs. Number of Processors from table 8.2
speedup is very poor because, as seen in table 8.3, the distribution of the iterations is very poor (again, purely due to the numbers of this example).

The curve for the speedup with static scheduling using `exdata` is smooth and rises steadily as the number of processors is increased. The smoothness of the curve is a result of the nondeterministic allocation of the iterations to the processors; a processor is given the next iteration when it requests work. While this does not necessarily produce an optimum schedule, the resulting speedup is acceptable.

The speedup curve for `LIBPRS` is disappointing. It is only slightly better than that for `EX.ST` which does not schedule the sub-tasks with priority. Looking at figure 6.8, one sees there that the performance of `EX.ST` and that of `LIB.ST` were almost identical. There are two reasons for the drop in performance of `LIBPRS` here relative to `EX.PRS`. Recall figure 6.7, which showed that the performance of the programs with the library routines was considerably worse than of those with `exdata` when the size of the sub-tasks was small. In figure 8.3 the programs solve a system where some of the sub-
tasks are also small, so that the comments made on figure 6.7 apply here. The second reason is technical. To enforce priority in scheduling the sub-tasks, extra synchronization must be used. Although the technical details of how the library routines implement the parallelization operations they perform are not our concern here, one can say, from experience, that some of them, and in particular barriers, are not very efficient. Locks can also sometimes produce unexpectedly long execution times.

Finally, the speedup curve for the **EX.PRS** is the best of the four programs. It is better at all points than the other three, except at \( p = 5 \), where **LIB.ST** produces a better speedup. This needs to be considered. It has already been said that the scheduling of the iterations at \( p = 5 \) is optimal for **LIB.ST** (table 8.3). However, with priority scheduling, **EX.PRS** should also produce an optimum schedule when \( p = 5 \). Consider equation 8.2 above. \( N = 20 \), and there are five blocks with \( n_i = 10 \), five with \( n_i = 15 \), five with \( n_i = 20 \), and five with \( n_i = 25 \). After the array of loop indices is ordered, the five largest blocks will be at the head of the array, followed by the five blocks where \( n_i = 20 \), and so on to the five smallest blocks. One would expect then that the first request from each of the five processors would cause each of them to receive one of the five largest blocks. Since all processors should spend the same amount of time on the first block they receive, they should all make their second request at the same time with the result that each will receive one of the second five largest blocks. This operation should be repeated until all the blocks have been processed. Indeed, this can be verified experimentally, and table 8.4 gives the iterations of the two parallel loops performed by each processor for one run of the program, in the order in which they were performed. It can be seen that the scheduling was optimum. The fact that **EX.PRS** performs worse than **LIB.ST** when \( p = 5 \) is caused by the extra overhead borne by the former due to the extra work involved in ordering the sub-tasks and the locks required for controlling access to the
In this section, exdata was used to implement scheduling sub-tasks of different sizes. This was done by modifying the sub-managers of chapter 6 and introducing an array for holding the loop indices. The code for the main tasks, however, did not change. Timing results for one example were given, and these have shown that the performance is at least comparable with that achieved with the parallel library subroutines.

### 8.3 Scheduling with Partial Dependence

In this section a new way of scheduling independent tasks, or sub-tasks, will be introduced, which leads to improved speedups in general.

Let $S$ be a set of independent tasks, $S = \{T_1, T_2, \ldots, T_N\}$. Let $\sigma(T_i)$ be the size of task $T_i$, and assume that $\sigma(T_1) = \sigma(T_2) = \cdots = \sigma(T_N)$. For example, $S$ might be the iterations of the loop of algorithm 1 when all the blocks are the same size. Let $p$ be the number of available processors. Then the speedup, $S$, achieved is as given in equation 8.3.

$$S = \frac{N}{\lceil N/p \rceil}$$  \hspace{1cm} (8.3)

where, as before, $\lceil \rceil$ is the ceiling function, defined by

$$\lceil x \rceil = \text{the smallest integer} \geq x$$  \hspace{1cm} (8.4)
and since, for our purposes, $N$ and $p$ are always positive integers, $[x] \geq 1$ always.

If $N \mod p = 0$, i.e. if $p$ divides $N$, then the speedup is $p$ and this is the ideal limit. But if $N \mod p > 0$ then the speedup is less than the ideal. Scheduling with partial dependence improves the speedup when $N$ is not divisible by $p$. Consider an order of execution (the order does not matter) of $N$ independent tasks by $p$ processors for $N = 6$ and $P = 4$, see figure 8.4.

If each task is made up of just one statement, then nothing further can be done about the speedup. In any case, if each task is just one statement one would not consider them for parallel processing of the sort considered here because the granularity would be too fine for any benefit to be gained. Assume then that each task is made up of several statements. Divide each task into two equal sized demi-tasks, i.e. divide each $T_i$ into $T_{i,1}$ and $T_{i,2}$, $1 \leq i \leq N$. The set of tasks to be scheduled is now

$$S' = \{T_{1,1}, T_{2,1}, \ldots, T_{N,1}, T_{1,2}, T_{2,2}, \ldots, T_{N,2}\}$$

but whereas the set $S$ was made up entirely of independent tasks, half the tasks in $S'$ are dependent on tasks from the other half. Specifically, each $T_{i,2}$ depends on $T_{i,1}$.
Looking at figure 8.4 and thinking of the tasks there as the demi-tasks $T_{i,1}$'s, it is immediately clear that processors 3 and 4 can start executing $T_{i,2}$ and $T_{2,2}$ at the same time that processors 1 and 2 are executing $T_{5,1}$ and $T_{6,1}$. If this were done, the order of execution of the tasks in $S'$ would be as shown in figure 8.5.

In fact, if $N \geq p$, and the tasks $T_{i,1}$ are scheduled first, then the dependency requirements are satisfied provided the tasks $T_{i,2}$ are scheduled in the same order as the tasks $T_{i,1}$.

Call such a way of dividing tasks and running them scheduling with partial dependence. This will be defined properly later.

The speedup corresponding to figure 8.4 is 3, whereas that corresponding to figure 8.5 is 4, which is equal to the number of processors. Of course, the numbers used in this example were deliberately chosen so that the speedup would become ideal. It should be obvious, however, that dividing each task into two and scheduling the demi-tasks in the way indicated above produces, at worst, the same speedup as would be achieved without the division. This will be proved later. For now, consider two examples, one where no improvement in speedup is achieved, and the other where an improvement is achieved.
even though in both examples \( N \mod p > 0 \) and \( 2N \mod p > 0 \).

For the first example let \( N = 10 \) and \( p = 3 \), then the speedup would normally be \( 10/[10/3] = 2.5 \). With partial dependence, the speedup becomes \( 20/[20/3] = 2.86 \) which is an improvement on 2.5.

For the second example let \( N = 5 \) and \( p = 3 \), then the speedup would normally be \( 5/[5/3] = 2.5 \). With partial dependence the speedup remains 2.5 (\( = 10/[10/3] \)).

Three points should be stressed here:

- No assumption is made about the independence of the statements within the tasks. Indeed, all the statements in each of the original tasks in the set \( S \) are assumed to be completely dependent on each other.

- Provided \( N \geq p \) then

- No synchronization is necessary.

The process can be continued further. The tasks \( T_i \) in \( S \) can each be divided into three (dependent) equal sized semi-tasks, \( T_{i,1}, T_{i,2}, \) and \( T_{i,3} \), which can then be executed such that the dependency requirements are satisfied by first executing all the \( T_{i,1} \), in any order, then, when the last \( T_{i,1} \) has been allocated (and not finished) to a processor, the tasks \( T_{i,2} \) are executed in the same order of execution as the \( T_{i,1} \)'s, and when the last \( T_{i,2} \) has been allocated, the tasks \( T_{i,3} \) are executed also in the same order. Provided \( N \geq p \) no synchronization is necessary and the speedup will be at least the same as that when the tasks are not divided into sub-tasks.

Clearly, the process can be generalized so that each task is divided into, say, \( d \) semi-tasks:

Given a set \( S \) of independent tasks, \( S = \{T_1, \ldots, T_N\} \), all of the same size, and a number \( p \) of processors, divide each task, \( T_i \), into \( d \) sub-tasks,
$T_{1,1}, T_{1,2}, \ldots, T_{1,d}$, where $T_{i,j}$ depends on $T_{i,j-1}$, $2 \leq j \leq d$. The set $S'$ of sub-tasks given by

$$S' = \{T_{1,1}, T_{2,1}, \ldots, T_{N,1}, T_{1,2}, T_{2,2}, \ldots, T_{N,2}, \ldots, T_{1,d}, T_{2,d}, \ldots, T_{N,d}\} \quad (8.5)$$

can be split into the sets $S'_1, S'_2, \ldots, S'_d$, where $S'_j$, $1 \leq j \leq d$, is given by

$$S'_j = \{T_{1,j}, T_{2,j}, \ldots, T_{N,j}\}, \quad 1 \leq j \leq d \quad (8.6)$$

and the tasks of the sets $S'$ can be scheduled with partial dependence which is defined here:

**Definition 1** *Scheduling with Partial Dependence.*

Given the sets $S'_j$ of equation 8.6 above, execute all the tasks of $S'_1$ first, in any order; the tasks of $S'_2$ can be executed as soon as one or more processors become idle, regardless of whether or not all the tasks of $S'_1$ have finished, provided that for each $T_{i,2}$ in $S'_2$, $T_{i,1}$ has finished executing.

In general, the tasks of $S'_j$ are scheduled after $S'_{j-1}$, as soon as a processor becomes idle, or, to put it another way, as soon as all the tasks of $S'_{j-1}$ have been allocated.

The reason for calling this scheduling with partial dependence is clear. For while it is true that $S'_j$ depends on $S'_{j-1}$, this dependence is partial because not every task in $S'_j$ depends on all the tasks in $S'_{j-1}$.

Notice that in this definition no assumption was made about the sizes of the tasks or the sub-tasks. As observed earlier, the dependency relations are satisfied provided $N \geq p$, the tasks in $S'_j$ are scheduled in the same order (but not necessarily on the same processors) as those in $S'_{j-1}$, and all the sub-tasks are of the same size (but see lemma 2 below).

**Lemma 1** *If $N < p$, then no improvement in speedup can be achieved by dividing the tasks and scheduling the sub-tasks with partial dependence.*

**Proof**
This is immediately clear. Let the sub-tasks of $S_j'$ be allocated to $N$ of the available processors. Then none of the tasks of $S_{j+1}'$ can be allocated to any of the remaining $p - N$ processors until all the sub-tasks in $S_j'$ have finished executing.

Let $\sigma(T)$ be the size of task $T$, then

**Lemma 2** If $N \geq p$, and $\sigma(T_{i,j}) \geq \sigma(T_{i,j-1})$, then scheduling the sub-tasks of $S_j'$ in the same order as those of $S_{j-1}'$ ensures that the dependency relations are satisfied.

**Lemma 3** The speedup achieved by dividing the tasks into equal sized sub-tasks and scheduling these with partial dependence is at worst that achieved by ordinary scheduling of the tasks.

**Proof**

If $p > N$, then the speedup does not change (lemma 1).

Now let $p \leq N$, then the speedup $S$ is given by equation 8.3, i.e.

$$S = \frac{N}{\lfloor N/p \rfloor}$$

Let the speedup achieved by dividing the tasks into $d$ sub-tasks each and scheduling these with partial dependence be $S'$, then $S'$ is given by

$$S' = \frac{dN}{\lfloor dN/p \rfloor} \quad (8.7)$$

Proving this lemma is the same as proving that

$$S' - S \geq 0 \quad (8.8)$$

Now,

$$S' - S = \frac{dN}{\lfloor dN/p \rfloor} - \frac{N}{\lfloor N/p \rfloor}$$

$$= A([N/p]d - [dN/p]) \quad (8.9)$$
where

\[ A = \frac{N}{\lfloor dN/p \rfloor \lfloor N/p \rfloor} \]

We now need to prove that

\[ ([N/p]d - \lfloor dN/p \rfloor) \geq 0 \]

**Case 1:** \( N \mod p = 0. \)

Here \([N/p] = N/p\) and \([dN/p] = dN/p\) so that

\[ ([N/p]d - \lfloor dN/p \rfloor) = 0 \]

**Case 2:** \( N \mod p > 0, \) and \( dN \mod p = 0. \)

Here

\[ [N/p]d - \lfloor dN/p \rfloor = [N/p]d - dN/p = d([N/p] - N/p) > 0 \]

**Case 3:** \( N \mod p > 0, \) and \( dN \mod p > 0. \)

Since

\[ [N/p] > N/p \]

it follows that

\[ d[N/p] > dN/p \]

Now, for an integer \( i \) and a non integer \( x, i \geq \lfloor x \rfloor \) if and only if \( i > x, \) so that

\[ d[N/p] \geq \lfloor dN/p \rfloor \]

\[ d[N/p] - \lfloor dN/p \rfloor \geq 0 \]

In general, the larger \( d \) is, the better the speedup. This can be very conveniently seen by looking at figure 8.6, which is made up of four plots of
speedup vs. \( N \) and \( p \) for \( 10 \leq N \leq 40 \) and \( 1 \leq p \leq 10 \), for various values of \( d \). Figure 8.6(a) shows the speedup when \( d = 1 \) (i.e., when no division of tasks is involved). The speedup surface is seen to be very uneven, going up when \( N \mod p = 0 \) and down otherwise. As \( d \) is increased, the speedup surface gets smoother, indicating that the speedup in general improves with increasing \( d \).

The above is only true in general, but the words *in general* must be emphasized. Lemma 3.3 tells us that at worst there is no improvement in speedup in going from \( d = 1 \) to \( d > 1 \). It is not, however, true that there is always an improvement in speedup in going from \( d = n \) to \( d > n \) when \( n > 1 \); indeed, the speedup when \( d = n \) can be better than the speedup when \( d > n \). For example, it has already been seen that with \( N = 6 \) and \( p = 4 \), the speedup is 3 with \( d = 1 \); with \( d = 2 \) the speedup is 4. If \( d \) is now made 3, the speedup falls to \( 3 \times 6/[3 \times 6/4] = 3.6 \).

### 8.3.1 Practical Considerations

After defining scheduling with partial dependence from a theoretical point of view, it must now be implemented. Two cases have to be considered. The first is when the tasks \( T_i \) making up the set \( S \) are different functions, and the second is when they are the same function acting on different data such as when they are different iterations of the same loop. The implementation will be considered both in *exdata* and using the parallel library routines.

For the first case, it will be seen that the implementation using *exdata* is straightforward and does not require any modification or addition to the original program other than the splitting of the tasks. Using the parallel library routines, however, does require some considerable effort to implement partial dependence.

The second case will require more effort when implemented with *exdata*,

261
Figure 8.6: Part 1 (of 2) Speedup for various values of $n, p,$ and $d$. 

(a) Speedup with $d=1$

(b) Speedup with $d=2$
Figure 8.6: Part 2 (of 2) Speedup for various values of $n, p,$ and $d$
but here again it will be seen that this is easier than using the library routines.

Before going on to the implementations, two observations are made:

1. The speedup is expected to be better, in general, the larger the number of sub-tasks, \( d \), into which the original tasks are divided. In practice, a limit on \( d \) is set by the tasks themselves, i.e. to what extent they can be divided, and due consideration must be given to the overhead required for scheduling a large number of small sub-tasks.

2. In the theoretical discussion, it was assumed that a task can be divided into equal sized sub-tasks, and it was shown that, provided \( N \geq p \), no synchronization is necessary. In practice one cannot assume that such division is possible, and even if it were, then one cannot assume that equal sized tasks take the same time for their execution, and the difference between the actual processors as well as extraneous factors such as the system's activities must be allowed for. Further, the code of the program itself must be correct for all situations and not just when \( N \geq p \). It follows that synchronization must be provided in order to ensure correct execution.

The number of sub-tasks into which each task is to be divided depends entirely on the tasks themselves, and no general rule can be made about it. Synchronization, on the other hand, can be treated in a general way.

Various forms of synchronization were mentioned in chapter 3; using locks, barriers, and semaphores, as well as software interrupts (signals) which were used for building \textit{exdata} (see chapter 5). None of these can provide satisfactory synchronization for partial dependence, and another technique will be used.
8.3.2 Partial Dependence with Exdata

Case 1: The tasks $T_i$ are different functions A key idea behind scheduling with partial dependence can be said to be function partitioning with nondeterministic function allocation. This will become clear during the discussion of the implementation of partial dependence using the parallel library routines. For now it is observed that with $exdata$ not only is the allocation of data to sub-tasks, when data is partitioned, nondeterministic, as has already been pointed out on several occasions, but the allocation of functions to workers in function partitioning is also nondeterministic. For it is not known in advance which processor, or worker, will perform which task. This nondeterminism makes implementing partial dependence particularly easy. (Indeed, it was the use of $exdata$ that stimulated the idea of partial dependence.)

Given a program made up of five main tasks, $U, T_1, T_2, T_3, V$, such that $T_1, T_2,$ and $T_3$ are independent of each other, and all three depend on $U$, and $V$ depends on $T_1, T_2,$ and $T_3$. Then this can be declared in $exdata$ as follows:

\[
\text{DEP } \{ \\
U : T_1, T_2, T_3 ; U_m. \quad U_m \text{ is } U\text{'s sub-manager etc.} \\
T_1 : U ; V ; T_{1.m}. \\
T_2 : U ; V ; T_{2.m}. \\
T_3 : U ; V ; T_{3.m}. \\
V : T_1, T_2, T_3 ; V_m. \\
\}
\]

and this will be the same regardless of the number of processors. If the tasks $T_i, i = 1, 2, 3$ are now each split into, say, two semi-tasks each, $T_{i,1}$ and $T_{i,2}$, then the resulting eight tasks would be declared as follows in $exdata$

\[
\text{DEP } \{ \\
\}
\]
Apart from writing the tasks themselves so that each $T_i$ becomes $T_{i,1}$ and $T_{i,2}$, this is all that has to be done, for any number of processors, and regardless of how the tasks are split, i.e. regardless of the sizes of the semi-tasks, the operation of the program would be correct.

The ease with which partial dependence is achieved in this case hides difficulties which will become apparent once the same is attempted using the parallel library routines.

Case 2: The tasks $T_i$ are the same function. The problem here is different, and its implementation in `exdata` is not straightforward. For here it is required that a task, $U$ say, which is dependent (albeit partially) on another, $V$ say, should start execution before $V$ has finished.

Given a task $T$ which is to be executed on several processors concurrently, each processor to execute $T$ with different data. Without loss of generality it can be assumed that $T$ is made up of a loop, and each processor is to execute some of the iterations of the loop. This assumption can be made because it is always possible to hold pointers to the different data in an array and then the action of $T$ becomes a loop ranging over the indices of the array. Denote by $T_i$ the $i$'th iteration of the loop, for $i = 1, 2, \ldots, N$, where $N$ is
the total number of iterations. It is desired to split $T$ into two semi-tasks, $T^{(1)}$ and $T^{(2)}$, and each iteration $T_i$ becomes $T_{i,1}$ and $T_{i,2}$, and the resulting tasks to be scheduled with partial dependence, i.e. by first scheduling all the $T_{i,1}$'s, and the $T_{i,2}$'s scheduled to run as soon as any processor becomes free, as described earlier. It follows that at some point the (semi-)task $T^{(2)}$ will have to start running while $T^{(1)}$ is still running. The problem here is that the dependency declarations in $exdata$, on which the order of execution of the main tasks depends, allow only for scheduling tasks that are either completely independent from or else completely dependent on other tasks; they do not, in other words, allow for partial dependence. (This is to be expected, since partial dependence is something new.) Notwithstanding this, it is possible to use $exdata$ to implement partial dependence. Before describing how this is achieved in general, it is noted that data partitioning, which is usually implemented by the use of sub-managers, can also be regarded as function partitioning and implemented as such. It is simply a matter of definition. If a function is defined as the code and the data on which it is to operate, then the problem at hand becomes the same as the one considered previously (case 1). This however, would make nonsense of the claim that explicit parallel programming can lead to, or at least point the way to, a general purpose parallel language. For such an approach would require that a new program be written every time the data is changed! (incidentally, it is also possible to regard function partitioning as data partitioning, and this point will be returned to briefly later in this chapter, and in a little more detail in the final chapter.) Another solution is presented here below.

Recall from chapter 5 that at the preprocessing stage of an input program to $exdata$ the dependency declarations are used, among other things, to build a list of independent tasks which is used by the manager to allocate tasks to workers requesting work. Any task on the list is either ready to be executed or already being executed but hasn’t finished (the number of workers exe-
cuting the same task is, of course, determined by the task's sub-manager). It is a trivial matter to arrange the functioning of \textit{exdata} such that the list of independent tasks becomes a queue. Initially the queue is made up of all independent tasks \textit{in the order in which these are declared} in the dependency declaration. During the running of the program, as other tasks become independent, they are added at the end of the queue. When the manager receives a request for work, it hands out the first task on the queue (after checking with its sub-manager that it requires more workers). Even without partial dependence in mind, this scheme is eminently sensible, and this is the way \textit{exdata} already functions.

Consider now a program made up of three main tasks, $U$, $T$, and $V$, where $T$ depends on $U$, and $V$ depends on $T$. Data partitioning is to be used with $T$ which can be considered to be a loop whose iterations range over $i = 1, 2, \ldots, N$. It is desired to split $T$ into two semi-tasks $T^{(1)}$ and $T^{(2)}$ and to schedule these with partial dependence. The dependency declaration in \textit{exdata} is as follows:

\begin{verbatim}
DEP {
 U : ; T^{(1)}, T^{(2)} ; U.m.  U.m is U's sub-manager etc.
 T^{(1)} : U ; V ; T^{(1)}.m.
 T^{(2)} : U ; V ; T^{(2)}.m.
 V : T^{(1)}, T^{(2)} ; ; V.m.
}
\end{verbatim}

It is important to stress here that the list of independent tasks operated by \textit{exdata} is a queue, and that entering tasks in the queue is determined by the order in which the tasks appear in the declarations. If this is not the case then $T^{(2)}$ might be scheduled before $T^{(1)}$, which would be wrong. As it is, $T^{(2)}$ will not be handed out until $T^{(1)}$ requires no more workers. It follows that if $N \geq p$ and the task $T^{(2)}$ is at least the same size as $T^{(1)}$, and $T^{(2)}$'s
sub-manager hands out iterations in the same order as \( T^{(i)} \)'s sub-manager, then the execution of the program would be correct and there would be no need for anything else to be added. But, as mentioned above, it cannot be assumed that the first two of these conditions will always be satisfied, and synchronization has to be provided.

All dependencies between two halves of an iteration, \( T_{i,1} \) and \( T_{i,2} \), are data dependencies, i.e. the values of some variables are changed by \( T_{i,1} \), and these variables are then used by \( T_{i,2} \). Let \( X_i \) be the set of these variables. One approach to synchronization is provided by regarding \( T_{i,1} \) as the producer of \( X_i \) and \( T_{i,2} \) as the consumer of \( X_i \). To ensure that no variable is consumed before it has been produced, a flag, \( f_x \), is associated with each variable, \( x \in X_i \). The flag is initially set to zero, say. When \( x \) is produced by \( T_{i,1} \), \( f_x \) is set to 1. Before \( T_{i,2} \) consumes \( x \), it checks the value of \( f_x \). If this is zero, it waits until it becomes 1, and only then does it consume \( x \), at which point it resets \( f_x \) to zero. \( T_{i,1} \), on the other hand, only writes to \( x \), or produces \( x \), when \( f_x = 0 \). It is clear that with this arrangement there is no need for locks, because at any one point in time, there is at most one instance of \( T_{i,1} \) and one instance of \( T_{i,2} \), and these two are the only tasks operating on the variables in \( X_i \). This technique is used to implement synchronization based on shared variables by, for example, the Force (see [58] and references there).

Whilst associating a flag with each variable and regarding the processes that operate on the variables as producers and consumers of the variables would, clearly, solve the problem of synchronization for scheduling with partial dependence, its implementation here would be overdoing things a bit. For it is not necessary to make \( T_{i,2} \) check that each and every variable \( x \in X_i \) has been produced; all that is required is that \( T_{i,2} \) wait until \( T_{i,1} \) has finished before it starts. The reason for mentioning the producer/consumer paradigm and their synchronization is that the same idea will be adopted, but the flag
will now be associated with the whole task rather than with each variable.

Associate a flag, $f_i$, with each pair of semi-tasks $T_{i,1}$ and $T_{i,2}$. The flag is initially set to zero. When $T_{i,1}$ finishes execution, it sets $f_i$ to 1. This is most conveniently done by $T_{i,1}$ itself. Now, $T_{i,2}$ is one iteration, iteration $i$, of $T^{(2)}$; it is, in the terminology used in *exdata*, a sub-task, or one instance of a main task. It follows that $T_{i,2}$ is created only after $T^{(2)}$ has been handed by the manager to a worker, which was subsequently handed the data $i = 2$ by $T^{(2)}$'s sub-manager. If $T_{i,2}$ itself is made to check the flag $f_i$ and wait until it is 1 before it starts executing, then this can cause unnecessary delay, because it might be the case that $T_{i,1}$ has not yet finished but another instance of $T^{(1)}$, $T_{j,1}$, say, ($i \neq j$), has finished and $T_{j,2}$ can therefore run. It is better then to make $T^{(2)}$'s sub-manager check the flags $f_i$ and hand out the first $i$ for which $f_i = 1$. In this case, delays are minimized if $T^{(2)}$'s sub-manager checks $f_i$ in the order in which the iterations of $T^{(1)}$ were handed out.

The importance of regarding and implementing the list of independent tasks as a queue in ensuring the correct order of execution was stressed earlier. When synchronization is added, this becomes even more important. For if $T^{(2)}$ is handed out to workers before $T^{(1)}$, not only might the results be wrong, there won't be any results at all: there will, instead, be deadlock, with all the workers waiting for an event that will never happen. Namely, the setting of the flags $f_i$ to 1.

To recap: in order to implement scheduling with partial dependence when the set of tasks is a set of instances of the same function, two things must happen. The order in which tasks are entered in and handed out from the list of independent tasks must be controlled. The other is that an array of flags must be used for synchronization.

In the discussion above, the task $T$ was divided into just two semi-tasks, $T^{(1)}$ and $T^{(2)}$. But what was said applies also when $T$ is divided into an arbitrary number, $d$, of semi-tasks. The queue remains a queue and the tasks
$T^{(1)}, T^{(2)}, \ldots, T^{(d)}$ must be declared in the correct order. As far as synchronization is concerned, a flag $f_{i,j}^{(j+1)}$ is associated with each pair $T_{i,j}$ and $T_{i,j+1}$, $(1 \leq j < d)$, which is initialized to zero, set to 1 upon completion of $T_{i,j}$, and used by $T^{(j+1)}$'s sub-manager in allocating sub-tasks. It is possible, however, to reduce the number of variables used for synchronization by changing the flags to integer variables. Associate a variable, $pd_i$, with each iteration $i$, which is initially set to zero. Upon completion of $T_{i,1}$, $pd_i$ is incremented by 1. The sub-manager of $T^{(2)}$ only hands out iteration $i$ if $pd_i = 1$, and upon completion of $T_{i,2}$, $pd_i$ is again incremented by 1. In general, the sub-manager of $T^{(d)}$ only hands out iteration $i$ if $pd_i = j - 1$, and $pd_i$ is incremented upon completion of each $T_{i,j}$.

### 8.3.3 Partial Dependence with the Parallel Library Routines

**Case 1: The tasks $T_i$ are different functions** Functional partitioning is not as straightforward as data partitioning using the parallel library routines. Consider, for example, three independent tasks, $T_1$, $T_2$, and $T_3$. It is easy enough, using the routines, to spawn two children, thus having three processes including the parent process, and to make each of the processes perform one of the three tasks. The code below shows how this might be done:

```c
... 
m_set_procs(3);
m_fork(ff);
...
void ff()
{
    if (m_get_myid() == 0)
        do task $T_1$;
    else if (m_get_myid() == 1)
        do task $T_2$;
        ...
```
else
    do task T3;

return;
}

All the processes spawned by an m.fork() call are identical to each other and to the parent process. They are only distinguished from each other by their id number, and so this is used for function partitioning. The few lines of code above are correct provided three processes are always created. This, however, need not always happen. It could be that the system is busy and so only two processes, say, are created. If this happens, then the program would not be correct and task T3 would never be executed. It follows that something must be done by the programmer to ensure that the tasks are allocated to the available number of processors. One way of doing this might be using code such as

...  
m_set_procs(3);  
m_fork(ff);  
...
void ff()
{
    ...
    i = m_get_nprocs();  /* this is the actual number of processors */
    if (i == 1) { do T1; do T2; do T3; }
    else if (i == 2) {
        if (m_get_myid() == 0) { do T1; do T2; }
        else { do T3; }
    }
    else {
        if (m_get_myid() == 0) { do T1; }
        else if (m_get_myid() == 1) { do T2; }
        else { do T3; }
    }
    return;
}
While this is correct, it is clearly not suitable for a large number of independent tasks and a large number of processors. Even with, say, ten tasks and ten processors, a few pages of code would be required to test every possibility.

The situation becomes even more unwieldy if the tasks are now split into two each and the resulting semi-tasks are scheduled with partial dependence, because now it has to be decided in advance how the first half task, \( T_{i,1}, i = 1, 2, \ldots \), is to be allocated, as well as the second half, \( T_{i,2} \). In general, a processor which performs \( T_{i,1} \) will not perform \( T_{i,2} \), otherwise there would be no need for splitting the tasks in the first place. Clearly, it would not make sense to write code that would go through every possibility of allocation of functions.

The reason why data partitioning is so much easier than function partitioning is that data is easily manipulated and assigned. So, to solve the problem of allocating functions the functions must be manipulated as though they were data. In C this can be achieved by the use of pointers to functions. An array of such pointers can be used, with each element of the array pointing to one of the tasks. Now, if element \( k \) points to \( T_{i,1} \) then \( k \) becomes effectively a label for \( T_{i,1} \). Function partitioning can now be achieved by partitioning the array, just as in data partitioning. No more details will be given here because this has already been used in building \textit{exdata} and explained in chapter 5.

Synchronization too has to be provided and this can be done in the same way as that described in the previous section, i.e. by associating a flag, \( f_i \), for each pair \( T_{i,1} \) and \( T_{i,2} \), which is set by \( T_{i,1} \) and inspected by \( T_{i,2} \). It was mentioned in the previous section that to avoid unnecessary waiting, the sub-manager should inspect the flag rather than the worker itself (through code in the task), and it would be beneficial in this case, too, to provide a separate function for handing out the function labels. Such a function would
only hand out a task $T_{i,2}$ if $f_i = 1$.

**Case 2: The tasks $T_i$ are the same function**  The problem here is a lot easier than that in case 1 because there is no function partitioning involved even though each task $T$ is split into two tasks $T^{(1)}$ and $T^{(2)}$.

It was observed earlier that, without loss of generality, $T$ can be thought of as a loop whose iterations range over $i$. If $T$ is run over several processors, data partitioning is used to allocate some of the iterations to each processor. If $T$ is now to be split into the two semi-tasks $T^{(1)}$ and $T^{(2)}$, then it would be very convenient not to split the function that implements $T$ into two functions, as was done previously in *exdata*, but rather to keep it as one function but split the loop within into two loops. Each of the two loops range over $i$, but the body of the first loop performs $T^{(1)}$ and that of the second performs $T^{(2)}$. Thus the need for function partitioning is avoided.

Let $T$ be the following function:

```c
void ff0
{
    for (i = 1; i <= n; i++) { s1; s2; }
    return;
}
```

After splitting $T$, the function becomes

```c
void ff0
{
    for (i = 1; i <= n; i++) { s1; }
    for (i = 1; i <= n; i++) { s2; }
    return;
}
```

When this is run concurrently on several processors, the first loop is always executed first. A process will start executing the second loop only after all the iterations of the first loop have been allocated (not necessarily when they...
have all finished), which is just what is required for scheduling with partial
dependence.

The first loop can be partitioned in any suitable way, for example using
static scheduling. Care must be taken however that the second loop is not
partitioned in the same way, for otherwise the two loops of the split function
would behave exactly the same as the loop of the original function. Some form
of explicit data assignment is required therefore at least for the allocation of
the iterations of the second loop.

This can be extended to cases where $T$ is split into more than two semi-
tasks.

Here again, synchronization has to be provided and the same technique
as before can be used. This provides another reason for using explicit data
assignment.

8.3.4 The Rôle of Synchronization in Partial Depen-
dence

If the tasks are split into suitably sized semi-tasks, then as has been observed
and proved, there is no need for synchronization. Synchronization is provided
to ensure that situations which are beyond the programmer's control do not
affect the correct execution of the program. The intention behind schedul-
ing with partial dependence is to keep as many processors busy for as long
as there is work that still has to be done; it follows that synchronization
should not be thought of, in this case, as forming part of the idea of partial
dependence.

It is, of course, possible to look at the situation differently. Since, by
providing synchronization, any division of the tasks into semi-tasks would
yield a correct program, one can divide all the tasks that make up a program
into semi-tasks of one statement each and end up with a parallel program
whose mode of execution is akin to one written in a data flow language. This is not at all the intention behind providing synchronization and the possibility is mentioned only because it would appear to be a natural extension of the idea behind scheduling with partial dependence.

8.3.5 Scheduling with Partial Dependence and Loop Spreading

In this section loop spreading is explained and compared with scheduling with partial dependence. As will be seen shortly, there are similarities between the two but also important differences.

Loop spreading is a recent technique [53] which applies specifically to loops whose iterations are run concurrently on several processors. Let $N$ be the number of loop iterations and let $p$ be the available number of processors. The intention with loop spreading is, as in scheduling with partial dependence, to improve load balancing, and hence the speedup, in cases where $N \mod p > 0$. It has already been noted that when $N \mod p > 0$ then in the final round of iterations some processors (at least one) are idle (see figure 8.4). The idea of loop spreading is to remedy the situation by spreading the body of the iterations in the final round across all the processors. (The following example is from [53].) For example, consider the loop whose iterations range over $i = 1, 2, 3, 4$, and the body of iteration $i$ is made up of the three statements:

$$s_1; s_2; s_3;$$

and let $p = 3$. Then the order of execution of the statements would proceed something like that in figure 8.7.

With loop spreading, the statements of loop 4 are spread out across all three processors with the resulting order of execution shown in figure 8.8.

Clearly such an arrangement would only work if the statements mak-
<table>
<thead>
<tr>
<th>Processor</th>
<th>Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$s_1$; $s_2$; $s_3$; (loop1) $s_4$; $s_2$; $s_3$; (loop4)</td>
</tr>
<tr>
<td>2</td>
<td>$s_1$; $s_2$; $s_3$; (loop2) $s_2$; (loop2) $x$</td>
</tr>
<tr>
<td>3</td>
<td>$s_1$; $s_2$; $s_3$; (loop3) $s_3$; (loop3) $x$</td>
</tr>
</tbody>
</table>

Figure 8.7: An order of execution of 4 iterations on 3 processors

<table>
<thead>
<tr>
<th>Processor</th>
<th>Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$s_1$; $s_2$; $s_3$; (loop1) $s_4$; (loop4)</td>
</tr>
<tr>
<td>2</td>
<td>$s_1$; $s_2$; $s_3$; (loop2) $s_2$; (loop4)</td>
</tr>
<tr>
<td>3</td>
<td>$s_1$; $s_2$; $s_3$; (loop3) $s_3$; (loop3) $s_4$; (loop4)</td>
</tr>
</tbody>
</table>

Figure 8.8: An order of execution of 4 iterations on 3 processors with loop spreading

The statements within the loop are independent of each other. To apply the technique to all cases and not just when the statements within the loop are independent of each other, the idea of "spreading" is extended to the whole loop body. All the statements of all the iterations are spread across as many processors as are available using some (deterministic) spreading scheme (see [53]) which calculates which statements are to be executed on which processor and in which order. To ensure that the dependency requirements are satisfied, synchronization is provided in the form of SYNC/WAIT pairs of statements which operate as follows: a statement, $s_2$, that wishes to wait for another statement, $s_1$, to finish before it starts has to execute the command WAIT($s_1$). There has to be a statement $s_1$ which has the command SYNC($s_2$), so that synchronization is done on a statement by statement basis.

It is at once clear that loop spreading is the same as scheduling with
partial dependence when the latter is applied to data partitioned tasks, and where each task is split into as many semi-tasks as possible.

Scheduling with partial dependence can be applied to all tasks in all situations where load balancing can be improved by changing the sizes of the tasks even when no independent tasks are created. Further, although speedup improves, in general, with the number of semi-tasks, no benefit can be expected once the granularity of the semi-tasks becomes too small. Scheduling with partial dependence allows the programmer to split the tasks into exactly the size that best suits the programmer's requirements.

8.3.6 Gaussian Elimination with Partial Dependence

The discussion of scheduling with partial dependence is now concluded with an example of its use in the solution of the block diagonal bordered linear system of chapter 6 using Gaussian elimination. For this example, the elimination step, reproduced as algorithm 1 in this chapter, is split into two semi-tasks. The first of these carries out the inversion of the diagonal element, statement (2) of algorithm 1, and the other carries out the rest of the elimination procedure.

Table 8.5 gives the execution times, and the corresponding speedup figures, of two programs when used to solve a system of 20 blocks each of which has $25 \times 25$ elements, EX_ST_GAUSS of chapter 6, and the other is EX_PD_GAUSS where the elimination step is split into two semi-tasks as explained above.

Figure 8.9 is a plot of the speedup figures. As can be seen, the speedup improves when the number of processors does not divide the number of iterations and remains the same otherwise, as expected.
Table 8.5: Timings and speedup figures for block Gaussian elimination with $N = 20$ and $n = 25$ with and without scheduling with partial dependence.

<table>
<thead>
<tr>
<th>No. Procs</th>
<th>SEQ_GAUSS</th>
<th>EX_ST_GAUSS</th>
<th>S</th>
<th>EX_PD_GAUSS</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>70.45</td>
<td>71.09</td>
<td>0.99</td>
<td>71.47</td>
<td>0.99</td>
</tr>
<tr>
<td>2</td>
<td>36.75</td>
<td>36.83</td>
<td>1.92</td>
<td>36.83</td>
<td>1.93</td>
</tr>
<tr>
<td>3</td>
<td>26.19</td>
<td>25.51</td>
<td>2.69</td>
<td>25.51</td>
<td>2.76</td>
</tr>
<tr>
<td>4</td>
<td>19.42</td>
<td>19.68</td>
<td>3.63</td>
<td>19.68</td>
<td>3.58</td>
</tr>
<tr>
<td>5</td>
<td>16.15</td>
<td>16.28</td>
<td>4.36</td>
<td>16.28</td>
<td>4.33</td>
</tr>
<tr>
<td>6</td>
<td>16.06</td>
<td>14.09</td>
<td>4.39</td>
<td>14.09</td>
<td>5.00</td>
</tr>
<tr>
<td>7</td>
<td>12.84</td>
<td>12.92</td>
<td>5.49</td>
<td>12.92</td>
<td>5.45</td>
</tr>
<tr>
<td>8</td>
<td>12.97</td>
<td>11.62</td>
<td>5.43</td>
<td>11.62</td>
<td>6.06</td>
</tr>
<tr>
<td>9</td>
<td>13.01</td>
<td>10.58</td>
<td>5.42</td>
<td>10.58</td>
<td>6.66</td>
</tr>
<tr>
<td>10</td>
<td>9.70</td>
<td>9.67</td>
<td>7.92</td>
<td>9.67</td>
<td>7.29</td>
</tr>
</tbody>
</table>

Figure 8.9: Plot of the speedup figures from table 8.5.
8.4 Nested Loops and Sub-Managers

It can be said that all data partitioning is the partitioning of a loop for the very definition of data partitioning is the application of the same instructions to different data. Since the data can always be considered as elements of a set, and since the set can always be indexed, the operations can then be represented as a loop iterating over the index set. The ubiquitous DO loops which iterate over array elements are a case in point. Let us distinguish between loops that iterate over elements of a data structure and those that do not. The distinction is that in the former, the loop index is just that: an index for selecting elements from a set. The loop is merely a convenient way of expressing that an instruction, or a group of instructions, is to be applied to different data. Whereas a loop such as

\[
\begin{align*}
y &= 1; \\
&\text{for } (i = 1; i \leq n; i + +) \\
y &= y \times x;
\end{align*}
\]

is a very different affair. The loop works out \( y = x^n \), and the loop index is part and parcel of the algorithm. The concern here is with loops where the index acts as a selector of data elements to be operated on; call this a counting loop. The loop control statement can be thought of here as a selector function.

In sequential programming it does not matter whether one thinks of the control statement as a mere counter for determining how many times a group of statements is to be repeated, or as a selector function. In parallel programming with data partitioning, however, it is helpful to think of it as the latter. The sub-managers, where they have been used here, behave in this way. Conversely, they can be thought of as revealed loop control structures.

Partitioning a single loop, when the iterations are independent of each other and each iteration requires the same amount of work as each of the oth-
ers, is a straightforward matter of dividing the range of the loop index by the number of processors available to do the work, and requiring each processor to iterate over its allocated range of iterations. When the amount of work is not the same in all the iterations the scheduling problem is encountered. This was touched on earlier in this chapter.

Nested loops present difficulties of a different nature. The problem is how to partition the iterations to ensure good load balancing? For example, in the loop of figure 8.10

\[
\text{for } i = 1 \text{ to } m \{
    \text{for } j = 1 \text{ to } n \{
        \text{...}
    \}
\}
\]

Figure 8.10: A double nested loop.

there are, altogether, $m \times n$ iterations of the body of the loop. If 3 processors are available, and $m = 5$, $n = 6$, there are 30 iterations and these should be partitioned so that each processor performs 10 iterations. Figure 8.11 shows one such possible partitioning.

Such partitioning cannot be achieved using a DOACROSS type structure. One way of achieving it would be by using explicit partitioning through a sub-manager. Before discussing this, however, a general procedure for partitioning nested loops will be given.

Figure 8.11: An ideal partitioning of the iterations of figure 8.10 among 3 processors when $m = 5$ and $n = 6$. 
One of the transformations used in automatic parallelizers for the handling of nested loops is called loop coalescing [64]. The idea here is based on an earlier technique called loop collapsing [86]. It is similar to the way in which multi-dimensional arrays are stored in memory. Since memory is one-dimensional, a mapping function is used to transform the subscripts of the array \((i_1, i_2, \ldots, i_n)\) into a universal index, \(u\). For an \(n\)-dimensional array where the subscript ranges are \(i_{1,b}\) to \(i_{1,e}\), \(i_{2,b}\) to \(i_{2,e}\), \ldots, \(i_{n,b}\) to \(i_{n,e}\), the mapping function \(f(i_1, \ldots, i_n)\) is given by equation 8.10

\[
u = f(i_1, \ldots, i_n) = (i_1 - i_{1,b}) \prod_{j=2}^{n} (i_{j,e} - i_{j,b} + 1) + \\
(i_2 - i_{2,b}) \prod_{j=3}^{n} (i_{j,e} - i_{j,b} + 1) + \ldots + (i_{n-1} - i_{n-1,b})(i_{n,e} - i_{n,b} + 1) + (i_n - i_{n,b} + 1)
\]

This can be written as

\[
f(i_1, \ldots, i_n) = \sum_{j=1}^{n} \left\{(i_j - i_{j,b}) \prod_{k=j+1}^{n} (i_{k,e} - i_{k,b} + 1)\right\}
\]

In loop coalescing, the mapping function is just the product of the ranges of each loop iteration. If a nested loop is represented by the ranges of its constituent loops, such that, for example, the loop of figure 8.10 is represented by \(L(m, n)\), then for a general nested loop

\[
L(n_1, n_2, \ldots, n_m)
\]

the mapping function \(f_L\) to produce a universal index is just

\[
u_L = f_L = \prod_{i=1}^{m} n_i
\]

and the loop is partitioned by giving each of the processors \([u_L/p]\) iterations to perform, where \(p\) is the available number of processors.

It is now necessary to have a reverse mapping, \(g_L\), from the universal index \(u_L\) back to the loop indices. In loop coalescing, it is assumed that the
loop indices are used as subscripts of arrays in the body of the loop. The back transformation function is then used in place of these array subscripts. An example will serve to illustrate. The loop (this is based on an example in [65]):

\[
\begin{align*}
\text{for } i &= 1 \text{ to } n_1 \\
& \quad \text{for } j = 1 \text{ to } n_2 \\
& \quad \quad \text{for } j = 1 \text{ to } n_3 \\
& \quad \quad \quad a_{i,j,k} = \cdots;
\end{align*}
\]

becomes

\[
\begin{align*}
\text{for } u &= 1 \text{ to } n_1 n_2 n_3 \\
& \quad a(\lfloor u/n_2 n_3 \rfloor, \lfloor u/n_1 n_2 \rfloor - n_2 \lfloor (u-1)/n_2 n_3 \rfloor, u - n_3 \lfloor (u-1)/n_3 \rfloor) = \cdots;
\end{align*}
\]

In order to avoid the expensive evaluations of the reverse mapping function for every iteration, the partial products obtained in calculating the mapping, equation 8.11 above, are stored in an array. Let 

\[
r = \lfloor n_1 n_2 n_3 / p \rfloor
\]

which is the number of iterations each processor performs. Each processor therefore performs the iterations \((p - 1)r + 1\) to \(pr\) of the coalesced loop. It is now possible to determine the range, of each index, that each processor performs. For a loop \(L(n_1, n_2, \ldots, n_m)\), the range of each index is given by

\[
r_j = \left[ \frac{(p - 1)r + 1}{\prod_{k=1}^{j-1} n_k} \right] - n_j \left[ \frac{(p - 1)r}{\prod_{k=1}^{j-1} n_k} \right] \text{ to } \left[ \frac{pr}{\prod_{k=1}^{j-1} n_k} \right] - i_j \left[ \frac{pr - 1}{\prod_{k=1}^{j-1} n_k} \right] \quad (8.12)
\]

This, in fact, produces ranges that are outside those of the original loop, but the original loop limits can be used to pick out only those values of \(r_j\) that lie within them.

An imperfectly nested loop (the term non-perfectly nested loop is used in the literature) is a loop of the form:
\begin{verbatim}
for i = 1 to n {
  s1;
  for j = 1 to n {
    s2;
  }
}
\end{verbatim}

where s1 and s2 each represents a block of statements. If s2 does not depend on s1, loop coalescing can be used, but it must be ensured that s1 is not executed except when the universal index is one of the values 1, \(n+1\), \(2n+1\), etc., so that the coalesced loop becomes

\begin{verbatim}
t = 0;
for u = 1 to \(n^2\) {
  if ([u/n] \(!=\) t) {
    s1;
    t = [u/n];
  }
  s2;
}
\end{verbatim}

If s2 depends on s1, then loop coalescing cannot be applied even if the outer iterations are independent of each other.

Loop coalescing has three shortcomings. The first is that it only applies to nested loops where all the index ranges are constant, so it cannot deal with cases where the index ranges are functions of other indices (such as the loops encountered in the Romberg integration). The second is that it assumes that the loop indices are always used as subscripts of arrays, whether directly or through functions. The third is that it requires the manipulation of all the statements where the loop indices appear.

In the following, a mapping function will be introduced for general nested loops, and its reverse mapping, in the form of an algorithm, will be derived. A loop transformation will be given such that statements involving the loop indices as well as the statements making up the body of the loop will not be altered.
An alternative strategy using explicit data partitioning will also be given.

8.4.1 A universal Mapping for Nested Loops

Consider the triple nested loop

\[
\text{for } i = 1 \text{ to } n \{ \\
\text{for } j = 1 \text{ to } 2i \{ \\
\text{for } k = 1 \text{ to } 2j \{ \\
\ldots \\
\} \\
\} \\
\} \\
\]

and assume that the iterations are independent of each other. To work out the total number of iterations, multiplication of the index ranges cannot be used because the number of iterations varies with \(i\) and \(j\); instead, we have to form the sum:

\[
\sum_{i=1}^{n} \sum_{j=1}^{2i} \sum_{k=1}^{2j} 1 = \frac{1}{3}(4n^3 + 9n^2 + 5n) \tag{8.13}
\]

The mapping function, \(f(x, y, z)\), from \((x, y, z)\), \(1 \leq x \leq n, 1 \leq y \leq 2x, 1 \leq z \leq 2y\), to the universal index, \(u\), which is in the range

\(1\) to \((4n^3 + 9n^2 + 5n)/3\)

is given by (cf. equation 8.10)

\[
u = f(x, y, z) = \sum_{i=1}^{x-1} \sum_{j=1}^{2i} \sum_{k=1}^{2j} 1 + \sum_{j=1}^{y-1} \sum_{k=1}^{2j} 1 + z \tag{8.14}\]

The partitioning of the loops amongst \(p\) processors requires handing each processor a range of iterations from \(u\). Let \([u/p] = r\), and let processor \(p_a\) receive the couple \((u_a, r)\) meaning that it is to start at iteration \(u_a\), of the universal index, and carry out \(r\) consecutive iterations.

Each processor, \(p_a\), must convert \(u_a\) into the triple \((x_a, y_a, z_a)\) and start executing at \(i = x_a, j = y_a, \) and \(k = z_a\). It has been observed that in
loop coalescing, a range is worked out for each index, but that this requires a reverse mapping function which produces more iterations than in the original loop. A much simpler solution is for each processor to keep a count, \( c \), say, which is incremented every time it performs an inner loop iteration; it stops when the count is equal to \( r \). Consider also what happens if \( r \) is such that the processor performs the iterations \( i = x_0, x_0 + 1 \). At \( i = x_0, \) \( j = y_0 \) to \( 2i \); but at \( i = x_0 + 1, \) \( j \) must start at 1. For this reason, and for similar considerations of the indices \( j \) and \( k \), the loop must be transformed into

\[
c = 0;
i = x; j = y; \text{for} (k = x; k \leq 2j \&\& c < r; k++, c++) \{ \\
\ldots \\
\} \\
\text{for} (j = y + 1; j \leq 2i \&\& c < r; j++) \{ \\
\text{for} (k = 1; k \leq 2j \&\& c < r; k++, c++) \{ \\
\ldots \\
\} \\
\} \\
\text{for} (i = x + 1; c < r; i++) \{ \\
\text{for} (j = 1; j \leq 2i \&\& c < r; j++) \{ \\
\text{for} (k = 1; k \leq 2j \&\& c < r; k++, c++) \{ \\
\ldots \\
\} \\
\} \\
\} \\
\}
\]

The transformation simply separates the first iteration of each level from the rest of the iterations, and adds, as a condition for repeating the iterations, the check \( c < r \) (note that \( c \) is initialized to zero). The body of the loop does not make any difference.

The above procedure can clearly be generalized for any nested loop. For the loop

\[
\text{for } i_1 = b_1 \text{ to } e_1 \{ \\
\text{for } i_2 = b_2 \text{ to } e_2 \{ \\
\ldots \\
\text{for } i_n = b_n \text{ to } e_n \{ \\
\}
\]

286
where the ranges $b_j$ to $e_j$, $2 \leq j \leq n$, are functions of $i_1, i_2, \ldots, i_{j-1}$, the universal mapping function $f_u(x_1, \ldots, x_n)$ is given by

$$f_u(x_1, \ldots, x_n) = \sum_{i_1=b_1}^{x_1-1} \sum_{i_2=b_2}^{x_2-1} \ldots \sum_{i_n=b_n}^{x_n-1} 1$$

$$+ \sum_{i_{n-1}=b_{n-1}}^{x_{n-1}-1} \sum_{i_n=b_n}^{x_n-1} 1 + \sum_{i_n=b_n}^{x_n-1} 1$$

(8.15)

this can be written as

$$f_u(x_1, \ldots, x_n) = \sum_{j=1}^{n} \left( \sum_{i_j=b_j}^{x_j-1} \sum_{i_{j+1}=b_{j+1}}^{x_{j+1}-1} \ldots \sum_{i_n=b_n}^{x_n-1} 1 \right) + 1$$

$$= P_1(x_1) + P_2(x_1, x_2) + \ldots + P_n(x_1, x_2, \ldots, x_n)$$

(8.16)

where $P_i(x_1, \ldots, x_i)$ is a polynomial in $x_1, \ldots, x_n$. In fact, $P_n$ will always be $(e_n - b_n + 1)$. Clearly, if the ranges, $b_j$ to $e_j$, $1 \leq j \leq n$, are all constants, then the polynomial becomes the products

$$P_j = (x_j - b_j + 1) \prod_{k=j+1}^{n} (e_k - b_k + 1)$$

(8.17)

and this is the special case dealt with by loop coalescing. The total number of iterations can be obtained from $f_u(e_1, e_2, \ldots, e_n)$, which is also

$$f_u(e_1, e_2, \ldots, e_n) = \sum_{i_1=b_1}^{e_1} \sum_{i_2=b_2}^{e_2} \ldots \sum_{i_n=b_n}^{e_n}$$

(8.18)

For the nested loop $L(n, 2i, 2j)$ above, the mapping function, equation 8.14, yields

$$f(x, y, z) = \frac{1}{3}x(x - 1)(4x + 1) + y(y - 1) + z$$

(8.19)

so that

$$f(n, 2n, 4n) = (4n^3 + 9n^2 + 5n)/3$$
to map \( u \rightarrow (x_1, x_2, \ldots, x_n) \)
(1) \( r = u; \)
(2) for \( (q = 1; q < n; q++) \) {
(3) \quad for \( (c = b_q + 1; r > 0; c++) \) {
(4) \quad \quad pr = r; \)
(5) \quad \quad \quad r = u - P_q(c); \)
(6) \quad \}
(7) \quad u = pr; \)
(8) \quad x_q = c - 2; \)
(9) \quad \quad r = u; \)
(10) \}
(11) \quad x_n = pr - 1 + b_n; \)

Figure 8.12: The universal reverse mapping algorithm.

as in equation 8.13.

We still need to find the reverse mapping. When the loop ranges are constant, this requires no more than dividing by the partial sums, but when the mapping function is as in equation 8.16, successive subtraction must be used to peel off the iterations. This is shown in figure 8.12.

Notice that in line (5), the polynomial \( P_q(c) \) means

\[
P_q(x_1, x_2, \ldots, x_{n-1}, c)
\]

Each polynomial \( P_j(x_1, \ldots, x_n) \) in equation 8.16 above gives the total number of iterations for \( i_1 = x_1, \ldots, i_j = x_j \) of all loops whose ranges are \( b_k \) to \( e_k \), for \( k = i + 1 \) to \( n \). This fact is used to derive the algorithm of figure 8.12.

The algorithm represents a general mapping function from the universal index \( u \) to the \( n \)-tuple \( (x_1, \ldots, x_n) \). It is not, however, efficient. It can be made more efficient in four ways. The first is analogous to the technique used in loop coalescing whereby the partial sums are held in an array at compile time. Here, the partial sums produced when \( f_u \), equation 8.18, is calculated can be held; so that these values can be substituted for \( P_q(c) \) in line (5) of
the algorithm. In general there will be
\[ \prod_{j=1}^{n} (e_j - b_j + 1) \]
such values. Not all of them need be stored, but values at the higher levels of the loop would reduce considerably the amount of calculations required.
The second is to take advantage of any polynomials \( P_j(x_1, \ldots, x_j) \) that are linear in \( x_j \) and use division instead of the loop (3)–(6). If
\[ P_j(x_1, \ldots, x_j) = K(x_j + a) \]
then
\[ x_j = \left\lfloor \frac{u'}{K} \right\rfloor - a - 2 \]
where \( u' = u - \sum_{k=1}^{j-1} P_k(x_1, \ldots, x_{j-1}) \), which is the value of the variable \( u \) at line (3) of the algorithm. The third is to use a technique similar to one widely used in searching a sorted list called binary search. To find a key \( k \), from among \( n \) records, look up record \( n/2 \), if its key is less than \( k \), look up record \( 3n/4 \), and if it is greater than \( k \), look up record \( n/4 \), and so on until the desired record is found. For the algorithm at hand, the loop (3)–(6) need not start at \( c = b_q + 1 \), but at a higher value, and if \( r \) is negative, then \( c \) is decremented, and so on. The choice of the starting value of \( c \) will depend on the polynomial \( P_q \). The fourth is to use, where possible, loop interchange so that a loop whose polynomial is difficult to evaluate becomes the innermost loop. From figure 8.12 it is clear that the last index requires a subtraction operation, so this would improve the efficiency of the algorithm.

8.4.2 An Explicit Partitioning Approach

A less sophisticated, more pedestrian, but cheerful approach to the problem of partitioning nested loops is provided by the use of sub-managers. We still require a mapping function to produce a universal index, but the reverse mapping will now be carried out by the sub-manager.
The reverse mapping function of figure 8.12 is used by each worker or processor to work out the ranges of iterations it is to perform. In this approach, the same functions are evaluated but using a simple count. The sub-manager uses a universal index count \( c_u \), and mimics the nested loop by going through its iterations and executing in the body of the innermost loop, a single statement which increments its universal index counter. If the total number of iterations, \( u \), is partitioned into groups of iterations each of length \( r \), then the sub-manager goes through the iterations \( r \) times at each request for work from a worker, after which it hands to the worker the values of its indices at that point. The execution resumes from these values at the next request. Thus the sub-manager is as shown in figure 8.13.

For example, for the triple nested loop considered earlier, its sub-manager would be as shown in figure 8.14.

As an example of its action, consider the case when \( n = 2 \); then \( u \) is given by equation 8.13 as 26; with \( r = 4 \), the (start, end) pairs produced would be

<table>
<thead>
<tr>
<th>start</th>
<th>1,1,1</th>
<th>1,2,3</th>
<th>2,2,1</th>
<th>2,3,1</th>
<th>2,3,5</th>
<th>2,4,3</th>
<th>2,4,7</th>
</tr>
</thead>
<tbody>
<tr>
<td>end</td>
<td>1,2,2</td>
<td>2,1,2</td>
<td>2,2,4</td>
<td>2,3,4</td>
<td>2,3,2</td>
<td>2,4,6</td>
<td>2,4,8</td>
</tr>
</tbody>
</table>

The loop itself must be modified so that each individual loop is separated into the first iteration and the rest, as described earlier, but now the count is not required; and instead, the last iteration of each individual loop is also separated from the rest.

The use of a sub-manager in this way has one serious drawback; because now the loop partitioning is effectively adding the cost of going through all the loop iterations, albeit without executing the body of the loop, *serially.* For, at any one time, only one processor has access to the sub-manager; indeed, it is because of the restricted access that it becomes unnecessary to go through the universal mapping algorithm of figure 8.12 and a simple count suffices. However, if the reverse mapping function is used, then it can
sub-manager for partitioning a general nested loop
initially: \( x_1 = b + 1, x_2 = b_2, \ldots, x_n = b_n; \)
\( c_u = 0; \)
at each request:
while \((u > 0)\) {
    if \((r > u)\) \(r = u;\)
    \(i_1 = x_1, \ldots, i_{n-1} = x_{n-1};\)
    for \((i_n = b_n; i_n \leq e_n \land c_u < r; i_n + +, c_u + +)\)
        for \((i_{n-1} = x_{n-1} + 1; i_{n-1} \leq e_{n-1} \land c_u < r; i_{n-1} + +)\)
            for \((i_n = b_n; i_n \leq e_n \land c_u < r; i_n + +, c_u + +)\)

... \(\)
    for \((i_1 = x_1 + 1; c_u < r; i_1 + +)\)
        for \((i_2 = b_2; i_2 \leq c_u \land c_u < r; i_2 + +)\)

... \(\)
    for \((i_n = b_n; i_n \leq e_n \land c_u < r; i_n + +)\);
start = \(\langle x_1, x_2, \ldots, x_n \rangle;\)
end = \(\langle i_1 - 1, i_2 - 1, \ldots, i_n - 1 \rangle;\)
\(u = u - r;\)
\(c_u = 0;\)
\(x_1 = i_1 - 1; x_2 = i_2 - 1; \ldots; x_n = i_n;\)
if \((x_n > e_n)\) \{
    \(x_{n-1} = x_{n-1} + 1;\)
    \(x_n = b_n;\)
    if \((x_{n-1} > e_{n-1})\) \{
        \(x_{n-2} = x_{n-2} + 1;\)
        \(x_{n-1} = b_{n-1};\)
        if \((\cdots)\) \{
            \(\cdots\)
            if \((x_2 > e_2)\) \{
                \(x_1 = x_1 + 1;\)
                \(x_2 = b_2;\)
            \}
        \}
    \}
\}
\}

Figure 8.13: A sub-manager for partitioning a general nested loop.
sub-manager for the triple nested loop $i = 1,n; j = 1,2i; k = 1,2j$
initially: $x_1 = 1, x_2 = 1, x_3 = 1, c = 0;$
while ($u > 0$) {
  if ($r > u$) $r = u$;
  $i = x_1; j = x_2;$ for ($k = x_3; k \leq 2j \&\& c < r; k +=, c +=$);
  for ($j = x_2 + 1; j \leq 2i \&\& c < r; j +=$)
    for ($k = 1; k \leq 2j \&\& c < r; k +=, c +=$);
  for ($i = x_1 + 1; c < r; i +=$)
    for ($j = 1; j \leq 2i \&\& c < r; j +=$)
      for ($k = 1; k \leq 2j \&\& c < r; k +=, c +=$);

  start = ($x_1, x_2, x_3$);
  end = ($i - 1, j - 1, k - 1$);
  $u = u - r$
  $c = 0$
  $x_1 = i - 1; x_2 = j - 1; x_3 = k$
  if ($x_3 > 2x_2$) {
    $x_2 = x_2 + 1$
    $x_3 = 1$
    if ($x_2 > 2x_1$) {
      $x_1 = x_1 + 1$
      $x_2 = 1$
    }
  }
  return (start, end);
}

Figure 8.14: A sub-manager for partitioning the nested loop used as an example.
be executed *concurrently* by all the processors.

Fortunately, it is not common to have nested loops such that each individual loop has an index which is a function of all the indices of the loops that contain it. One can therefore usually produce more efficient reverse mapping functions and better sub-managers. Two further examples will now be given of nested loops. The first of which has indices that are all functions of the indices of the loops that contain them, and the second is more amenable to shortcuts. After these examples, this section will conclude with a discussion of the rôle and meaning of the sub-managers.

### 8.4.3 Example 1

Consider the multiplication of two lower triangular matrices,

\[
\begin{pmatrix}
  a_{11} & a_{12} & \cdots & a_{1n} \\
  a_{21} & a_{22} & \cdots & a_{2n} \\
  \vdots & \vdots & \ddots & \vdots \\
  a_{n1} & a_{n2} & \cdots & a_{nn}
\end{pmatrix}
\begin{pmatrix}
  b_{11} \\
  b_{12} \\
  \vdots \\
  b_{n1} & b_{n2} & \cdots & b_{nn}
\end{pmatrix}
= 
\begin{pmatrix}
  c_{11} \\
  c_{12} \\
  c_{13} \\
  \vdots \\
  c_{n1} & c_{n2} & \cdots & c_{nn}
\end{pmatrix}
\]

(8.20)

To perform the multiplication, a triple nested loop is required:

\[
\text{for } i = 1 \text{ to } n \{
\text{for } j = 1 \text{ to } i \{
\text{for } k = j \text{ to } i \{
  c_{i,j} = c_{i,j} + a_{i,k} \times b_{k,j};
\}
\}
\}
\]

This is an imperfectly nested loop with the statement \( c_{i,j} = 0 \) intervening between the second and the third loops. To make the loops entirely parallel, \( c_{i,j} \) is replace by \( c_{i,j,p} \) where \( p \) is the id number of the processor. At the end of the loop these values are summed over all the processors for each element.
The mapping function is given by

\[ f(x, y, z) = \sum_{i=1}^{x-1} \sum_{j=1}^{y-1} \sum_{k=1}^{z-1} 1 + \sum_{i=1}^{y-1} \sum_{j=1}^{z-1} 1 + \sum_{i=1}^{z-1} 1 \]  \hspace{1cm} (8.21)

so that

\[ f(x, y, z) = \frac{1}{6} x(x-1)(x+1) + \frac{1}{2} (y-1)(2x-y+2) + (z-y+1) \]  \hspace{1cm} (8.22)

The total number of iterations is given by

\[ f(n, n, n) = n(n + 1)(n + 2)/6 \]  \hspace{1cm} (8.23)

The table below shows how the indices proceed for \( n = 3 \)

<table>
<thead>
<tr>
<th>( i )</th>
<th>1</th>
<th>2</th>
<th>2</th>
<th>2</th>
<th>3</th>
<th>3</th>
<th>3</th>
<th>3</th>
<th>3</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>( j )</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>( k )</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

From equation 8.22, it can be seen that \( P_1(x) \) is a cubic in \( x_1 \), \( P_2(x, y) \) is a quadratic in \( y \), and \( P_3(x, y, z) \) is linear in \( z \). It follows that the reverse mapping function has to evaluate \( P_1(x) \) and \( P_2(x, y) \) and successively peel off their values. From this and the general reverse mapping function of figure 8.12, the reverse mapping function for the loop in hand becomes as shown in figure 8.15.

In order to parallelize the loop then, the value of \( f(n, n, n) \) is divided by the number of processors and the appropriate range is given to each processor which then performs the reverse mapping and executes the loop, suitably modified by separating the iteration of each individual loop as described above. The full program is given in the appendix.

**8.4.4 Example 2**

For this example one of the loops encountered in the double Romberg integration will be considered. The loop is that of lines (6)–(12) in figure 7.14; it is given here below slightly modified to allow for its parallelization.
to map $u = x(x - 1)(x + 1)/6 + (y - 1)(2x + 2)/2 + (z - y + 1)$ to $(x, y, z)$

```c
    r = u;
    for (c = 2; r > 0; c++)
        pr = r;
        r = u - c(c - 1)(c + 1)/6;
    }
    u = pr;
    x = c - 2;
    r = u;
    for (c = 2; r > 0; c++)
        pr = r;
        r = u - (c - 1)(2x - c + 2)/2;
    }
    y = c - 2;
    z = pr - 1 + y;
```

Figure 8.15: Reverse mapping for the matrix multiplication example.

1) for $q = 2$ to $\ell$

2) $g = 2^{(q-1)}$

3) $m_q = m \times g; \; n_q = n \times g$

4) $h_q = h/g; \; k_q = k/g$

5) sum = sum + $4 \sum_{i=1}^{r_q/2-1} \sum_{j=1}^{n_q/2} f(x_0 + 2ih_q, y_0 + (2j - 1)k_q)$

6) }

The mapping function is thus given by

$$f(x, y, z) = \sum_{q=2}^{x-1} \sum_{i=1}^{n_2^{(q-2)}-1} \sum_{j=1}^{m_2^{(q-2)}} 1 + \sum_{i=1}^{y-1} \sum_{j=1}^{n_2^{(q-2)}} 1 + \sum_{j=1}^{z} 1$$  \hspace{1cm} (8.24)

so that

$$f(x, y, z) = \frac{1}{3}mn(2^x - 1) - m(2^x - 1) + m(y - 1)2^{(x-2)} + z$$

$$= P_1(x) + P_2(x, y) + P_3(x, y, z)$$  \hspace{1cm} (8.25)

The total number of iterations is obtained from

$$f(\ell, n2^{(\ell-2)} - 1, m2^{(\ell-2)}) = \frac{1}{3}mn(2^{(\ell-1)} - 1) - m(2^{(\ell-1)} - 1)$$  \hspace{1cm} (8.26)
to map $u = mn(2^{2(z-2)} - 1)/3 - m(2^{(z-2)} - 1) + m(y - 1)2^{(z-2)} - z$ to $(x, y, z)$

$r = u; a = 1; b = 1;
for (c = 3; r > 0; c++) {
    pr = r; a = a \times 4; b = b \times 2;
    r = u - mn(a - 1) + m(b - 1);
}

u = pr;
x = c - 2;
b = b/2;
y = \frac{x}{mb} + 1;
z = u - \frac{y - 1}{mb}$

Figure 8.16: Reverse mapping for the Romberg loop of example 2.

Note that $P_2(x, y)$ is linear in $y$, and this fact can be used to simplify the reverse mapping function, which now becomes as shown in figure 8.16.

### 8.4.5 Sub-Managers as Loop Control Statements

At the beginning of this section loop control statements of certain types of loops, called counting loops, were likened to selection functions. From the discussion that followed it is clear that loop control statements, in their rôle as selection functions, act precisely in the way that mapping functions work; indeed, they may be regarded as implementations of mapping functions, albeit primitive implementations. Primitive in the sense that they employ primitive operations, viz. incrementing a variable and jumping on zero. The sub-manager given in figure 8.13 acts, in this respect, as the extension of a loop control statement in parallel programming. This points the way to their future development beyond what they are now, mere subroutines for data partitioning, to become, perhaps, a new control structure that provides a general extension of DO type loops in parallel programming.
Chapter 9

Conclusions

9.1 Conclusions

The work presented in this thesis can be described in three stages. The first is the implementation of a harness for writing parallel programs using explicit partitioning of the functions and data. The second stage, which follows on from this, is the use of the harness in applications in order to show its performance and the advantages that its use can bring. The third stage then follows on from both the first and the second stages. The adaptability of the harness was illustrated when the applications were modified, and it was shown how the underlying model of parallel processing leads naturally to new paradigms in parallel programming. These stages will now be described.

The Harness The main thrust of the work arose from the observation made in chapters 2 and 3 that the parallelization of an algorithm cannot and does not change the algorithm. It follows that the parallelization adds statements and data structures that are, properly speaking, extraneous to the algorithm. Facilities for parallel programming, in the form of extensions to existing languages and tools, however, are provided either to take advantage of specific hardware or to implement particular paradigms of parallel
programming. In the light of this, explicit partitioning was proposed as a technique for separating the parallelization process from the implementation of an algorithm. This was defined in chapter 4 and its advantages were enumerated. In particular, it allows different partitioning strategies without requiring extensive rewriting of the algorithm; it allows the development of parallel control structures and paradigms because it does not presume any to begin with; and as a technique it can be generalized so that it can be implemented separately from any particular application. In addition, just as a sequential program contains control statements for controlling the execution flow of the program, so a parallel program can be regarded as a collection of tasks along with statements or routines that determine the order of their execution. Further generalization was carried out, and chapter 5 described in detail the design and implementation of a prototype harness, called *exdata*, for running parallel programs regarded as a collection of tasks and a schedule for determining the order of their execution. An immediate benefit was that the harness obviated the need for synchronization from within the program.

**Applications** The second part was the use of the harness in implementing two substantial applications. The aim was twofold: to show that the harness gives satisfactory results and to highlight the advantages of the explicit partitioning technique advocated.

The first application, chapter 6, was the solution of a system of linear equations that arises from the finite element method by two direct methods, Gaussian elimination and Choleski factorization. The solutions were implemented using the parallel library routines provided on the Sequent computer and the harness. Two scheduling strategies were used, static and dynamic. (The use of these two terms is here taken to mean, respectively, the allocation of tasks one by one to the processors, and allocation of tasks in blocks whose size depends on the number of tasks and the number of available pro-
cessors. In the literature the terms static and dynamic scheduling are usually taken to mean what is called in this thesis deterministic and nondeterministic scheduling.) In addition, a scheduling strategy which is a mixture of the two was developed which is useful when the number of tasks is not large. This showed that the performance from the harness was indeed satisfactory and also that its use allowed different scheduling strategies to be employed without the need to alter the program. The only changes made were to the sub-manager responsible for allocating the tasks, which, as a result of the explicit partitioning used, is separate from the rest of the program.

The second application, chapter 7, was the numerical evaluation of single and double integrals using the Romberg method. Here again the performance of the harness was satisfactory; and, perhaps more importantly, the use of the harness made two things possible that would otherwise have been very difficult or impossible. It allowed the scheduling of the tasks under both function and data partitioning to be performed in a very easy and convenient way to the programmer, and the nondeterministic allocation of the tasks and data meant that processors would be kept busy regardless of how many processors there are for as long as work was available. Secondly, it allowed the partitioning of nested loops. This, in turn, served to point out the advantage of explicit partitioning over specific parallel structures.

**Development** The flexibility and convenience of using explicit partitioning was further emphasized when scheduling with priority was implemented to solve the system of equations of chapter 6 when the tasks were of differing sizes. In particular, the only changes necessary were to the sub-managers.

Scheduling with partial dependence was then introduced. This required the modification of the basic model of parallel processing so that independent tasks were given different priorities. This did not, however, necessitate changing the harness, but only a reinterpretation of its action. It did, however,
necessitate imposing further synchronization on the tasks. The technique of partial dependence is new, and it allows better load distribution when the number of tasks is not a multiple of the number of available processors. It was pointed out that a recent technique, called loop spreading, which has the same reasoning behind it as scheduling with partial dependence, has two disadvantages: the first is that it requires rewriting the loops involved depending on the number of iterations and the number of processors; and that it requires synchronization on individual statements. Neither is true for partial dependence under *exdata*; indeed the technique was a direct result of the flexibility the harness offers in function and data partitioning. One might also add that loop spreading applies only to loops, but partial dependence can be applied to any tasks, and this was described in detail.

Partitioning of nested loops was then presented. This was first done, ad hoc, in chapter 7. This was developed by generalizing the idea and formalizing it. The partitioning of arbitrary loops is a new development in the field; in particular the universal mapping function for mapping loop indices into a universal index, the reverse mapping algorithm, and the necessary transformation of a nested loop were formalized. In this connexion, it was observed that explicit partitioning leads implicitly to the generalization and formalization produced. In addition to the parallelization of nested loops by the use of a universal mapping, the problem was also solved using a general sub-manager, whence the idea of sub-managers as a generalization of loop control statements in parallel programming. Not only is the treatment of general nested loops new, but the transformation of the loops is remarkable for it requires no more than separating the iterations into the first iteration and the rest, without in any way affecting the action of the loop and the statements making up the body.

One may add, at the end, the remark that the harness has opened an avenue for exploiting function partitioning. Although function partitioning is
always mentioned in the literature, apart from pipelining (and some authors
do not even consider pipelining to be a form of parallelism at all) one is
hard pressed to find a single example where function partitioning is actually
used in parallelizing an algorithm. In the work presented here, function
partitioning has come up twice: in the scheduling with partial dependence,
and in the integration application.

9.2 Suggestions for Further Work

Three kinds of development come to mind: the improvement of the harness,
the extension of the harness to handle a more powerful model of parallel
processing, and the use the harness in a wider range of applications.

The harness is, by necessity, a prototype. It can be improved by adding to
it features for detecting incorrect or inconsistent dependency declarations; by
allowing data types other than integers to be returned by the sub-managers;
it can be made more efficient. A most desirable improvement would be to
include it in a compiler. At the moment, source to source transformations
are carried out on the input program, and it is only at run time that the
harness produces the dependency lists and other bookkeeping information.
These actions can all be done at compile time.

The harness can also be extended. Two developments, in particular,
should be suggested. The first is to allow the writing of the manager in the
same way that sub-managers can at present be written by the user. The key
to this would be to make the names of main tasks and their sub-managers
like a data type that can be returned by a subroutine. It would then be
possible for the programmer to determine the partitioning of a program in
response to the data and number of processors available. Such a development
would lead to a very strong model of parallel programming.

The second kind of development requires rather more work. At the mo-
ment sub-managers, and the manager if written as described above, are self-contained subroutines; yet they are shared by all the processors. It follows that they must, when called, perform operations to determine, as it were, where they are, usually by looking up certain variables. Consequently the code for the sub-managers looks awkward for the prevalence of IF statements. It would be an enormous improvement if they were to made similar to coroutines used in, say, SIMULA [17].

Another suggestion for exdata is to incorporate the synchronization necessary for partial dependence in the harness.

Finally, the mapping function and the reverse mapping algorithm used for partitioning nested loops can be studied and formalized more rigorously, and ways of finding efficient implementations sought.
Appendix A

The main programs relating to the work presented here are put on the floppy disk which is in the pocket of the back cover. The floppy can be read on any Apple computer. It includes a README file which lists the programs included.

The programs themselves were all written for the Sequent Balance computer under Dynix.
Bibliography


305


[80] A. Takeuchi. How to solve it in concurrent prolog, 1983. This is an unpublished note; the reference to it is taken from the Foster and Taylor book on Strand, and is included here for completeness.


