A Cost Analysis for a Higher-order Parallel Programming Model

Roopa Rangaswami

Doctor of Philosophy
University of Edinburgh
1996

(Graduation date:December 1996)

To my parents

Sulo and Ranga

and sister Kripa

Abstract

Programming parallel computers remains a difficult task. An ideal programming environment should enable the user to concentrate on the problem solving activity at a convenient level of abstraction, while managing the intricate low-level details without sacrificing performance.

This thesis investigates a model of parallel programming based on the Bird-Meertens Formalism (BMF). This is a set of higher-order functions, many of which are implicitly parallel. Programs are expressed in terms of functions borrowed from BMF. A parallel implementation is defined for each of these functions for a particular topology, and the associated execution costs are derived. The topologies which have been considered include the hypercube, 2-D torus, tree and the linear array. An analyser estimates the costs associated with different implementations of a given program and selects a cost-effective one for a given topology. All the analysis is performed at compile-time which has the advantage of reducing run-time overheads. The cost model's accuracy in choosing a cost-effective implementation and predicting its performance has been studied for three example programs.

The main contribution of the thesis is the cost model which aims to predict realistic performances and which considers several possible parallel implementations for a given program before selecting a cost-effective one.

Acknowledgements

The following people have all helped make this thesis possible and are due many thanks. My supervisor, Murray Cole, was always willing to read and criticise my documents, and this thesis owes a great deal to the many discussions we have had. A number of pertinent points were also raised in conversations with Kevin Mitchell and Peter Thanisch. My parents have always been a source of inspiration - they provided invaluable support during a crucial period. My parents-in-law shared a big responsibility as I started to write up. My husband, Arvind, has supported me in more ways than can be mentioned here. My thanks are also due to him for proof-reading this document and suggesting many improvements.

Parallel machine facilities were provided by the Edinburgh Parallel Computing Centre. This work was partially funded by the Overseas Research Studentship, awarded by the British Council.

Declaration

I declare that this thesis was composed by myself and that the work described in it is my own except where otherwise stated. A part of this work was published in [Ran95].

Roopa Rangaswami

Table of Contents

List of Figures

Chapter 1

Introduction

Developing efficient software for parallel computers is a difficult task, even for the specialist. This is so for reasons which are peculiar to the nature of parallel programs.

- Non-determinism - The order of events in a parallel program can vary at run-time and is difficult to predict. This makes it hard to debug parallel programs and verify their correctness.

- Resource Management - The efficient management of all the computational and communication resources is an onerous task. The computational load has to be optimally divided and the communications managed in a manner which produces programs that are free from deadlock and non-termination.

- Portability - The architectures of parallel machines tend to be different. This means that for a given problem, different algorithms might result in efficient solutions on their respective machines, making parallel programs less portable.

1.1 Approaches to Parallel Programming

The approaches which make parallel programming more manageable can be divided into three main topics.

1. Explicit Parallel Programming Languages - These languages include constructs that allow the programmer to explicitly create processes that can be executed in parallel, and to manage the interactions between them. Languages such as Ada, Modula and Concurrent Pascal, have parallel constructs integrated into them and this makes them fairly architecture-independent. Other sequential languages that use library calls to handle the parallelism may be architecture-dependent. However, the creation of portable message-passing libraries such as MPI [Mes94] and PVM [B$^+$91] makes it possible to write portable parallel programs using sequential languages like *C*.

2. Parallelising Compilers for Sequential Languages - These compilers create parallel code from existing sequential programs. The advantages are the following: programmers need not learn a new language; investment in existing software is not lost; the method is architecture-independent; and efficient code can be generated for many regular problems. However, all the available parallelism may not be detected because these compilers must adopt a conservative approach in generating parallel code, in order to ensure its correctness. Compiler directives may solve this problem to some extent, but these directives may destroy the architecture-independence. Moreover, the code obtained by parallelising the sequential solution to a problem, may not be the most efficient way of solving the problem in parallel.

3. Implicit Parallel Programming Languages - Programs in these languages are inherently parallel and there is no need for explicit parallel constructs. Such languages tend to be *applicative* or *functional* [BW88] in nature, and are architecture-independent. An added advantage of functional languages is that a correct program will produce the same result, irrespective of the order of evaluation. It is possible to write programs at a high level of abstraction

using higher-order functions. Functional languages also possess other attractive features such as a firm mathematical foundation, lazy evaluation, and amenability to transformation [Bac78, AE88, Hug90]. However, functional languages tend to be less efficient than their imperative language counterparts.

This thesis considers a data-parallel model of parallel computation based on Approach 3. The low-level decisions about managing parallelism are transferred from the programmer to the compiler, which results in easier program development and also enhances its portability. However, compilers for these languages may not always detect all the available parallelism nor select the most cost-effective parallel implementation. This thesis addresses the latter problem. A cost model is developed for a particular style of implicit parallel programming, based on the Bird-Meertens Formalism (BMF). The problem associated with the detection of the available parallelism is removed by restricting the expression of data-parallelism to a fixed number of predefined constructs. The model is aimed at executing data-parallel programs on MIMD (Multiple Instructions Multiple Data) machines to produce SPMD (Single Program Multiple Data) programs.

1.2 Motivations

The discussion in Section 1.1 highlights the need for a model of parallel computation that ideally incorporates the following features:

1. Architecture-independence
2. Abstraction from the low-level details of parallel programming
3. Ability to predict performance accurately
4. Ease of learning and use

Several parallel programming models have been proposed and their merits and disadvantages are discussed in Chapter 2. This thesis considers a Higher-order Parallel Programming (**HOPP**) model which possesses all of the features just described. HOPP is based on the Bird-Meertens Formalism [Bir87a]. This formalism was originally developed in a sequential context, with a view to providing a calculus for deriving efficient programs from problem specifications, by using a series of program transformation rules. The theory initially concentrated on list data structures [Bir87b, Spi89], but other data structures such as multidimensional arrays have also been studied [Mil93]. HOPP focuses on the theory that was developed for list data structures. It comprises of a set of useful data-parallel functions whose definitions can be found in Chapter 4. The theory is based on a functional paradigm and automatically inherits features 1 and 2 (For more details, see Section 3.4). Since the model can be embedded in any functional language, there is no learning curve associated with its use (feature 4). The thrust of this thesis lies in the incorporation of feature 3 into the model. The behaviour of the functions in BMF is predetermined and this feature is exploited in building a cost model that aims to accurately predict the costs of programs. The cost model currently focuses on distributed-memory machines in which communication costs tend to be significant. The prediction of the costs associated with the possible parallel implementations of a program would enable a compiler to generate a cost-effective implementation for it. The Bird-Meertens Formalism appears to be an attractive candidate on which to base a model of parallel computation, if the cost model is able to predict realistic program performance.

The observation of the suitability of BMF in parallel computation has been made by a number of other researchers, notably [Ski92, Gor95]. In [Col87], an approach to parallel programming based on *Skeletons* is proposed. Skeletons are abstractions of well-known parallel computational forms for which the compiler can generate and manage efficient parallel implementations. It has been observed [Col88] that skeletons can be elegantly expressed as higher-order functions which are often borrowed from BMF, for which, in turn, parallel implementations can be realised. This idea has been exploited in [D$^+$93].

1.3 Outline of the Thesis

Chapter 2 discusses some of the issues involved in parallel skeletal programming and some of the techniques that have been studied. Other approaches, particularly to parallel functional programming, are surveyed.

Chapter 3 provides an overview of the HOPP model. Some of the advantages and limitations of the HOPP model are discussed, and an outline of the parallelisation scheme is presented.

Chapter 4 defines the functions borrowed from the Bird-Meertens Formalism. Other useful functions which are expressed in terms of these functions and which are included in the HOPP model, are also defined. Three programs which are expressed in terms of these functions are discussed. These programs are used to test the performance of the model and the results of their implementation are presented in Chapter 7.

The cost model is described in Chapter 5. The nature of the problem specification and the compile-time cost analyses of programs are discussed, independent of the characteristics of the target architecture.

In Chapter 6, parallel implementations for the functions defined in Chapter 4 are considered. The algorithms for implementing them on four target topologies, namely, the *hypercube*, *2-D torus*, *tree* and *linear array*, are defined and their associated costs are derived.

The results of implementing the example programs (defined in Chapter 4) are presented in Chapter 7. The programs were implemented on a hypercube topology, since it proved to be suitable for most of the BMF functions and also for the parallelisation scheme. For each program, the predicted cost of the chosen implementation is compared with its actual cost of implementation on a network of transputers.

Chapter 8 highlights the main contributions of the thesis. Some conclusions are drawn and directions for future research are explored.

Chapter 2

Related Work

The difficulties associated with parallel programming have motivated the development of models and techniques to make it a more manageable task. Parallel programming using skeleton functions is one such approach and this chapter focuses primarily on the developments in this area, since it is of direct relevance to this thesis. Techniques based on functional programming constitute a well-researched topic and since the skeleton functions can be elegantly expressed in a functional style, there is a close relationship between the two approaches. The HOPP model is based on the Bird-Meertens Formalism which is functional in nature. The chapter, therefore, also provides a brief introduction to functional programming and surveys some of the approaches to parallel functional programming.

2.1 Parallel Skeletal Programming

A number of recent approaches have focussed on parallel programming using *algorithmic skeletons*. The concept of *algorithmic skeletons* was first presented in [Col87, Col89]. A *skeleton* is an abstraction of some well-known computational form. The idea is to present the programmer with a selection of such skeletons, each of which captures the essence of some particular style of programming, and is parametrised by certain functions and data structures. The programmer selects the skeleton describing the problem to be solved. In order to implement the selected skeleton, the definitions for the functions and data structures on which it is parametrised must be provided. The programmer's responsibility ends at this point and the system then provides an efficient parallel implementation for the skeleton on a chosen architecture. The implementation itself is transparent to the user and can make use of optimisations based on the nature of the target architecture characteristics, in an effort to derive efficient parallel implementations. It was further noted [Col88] that skeletons could be expressed elegantly as higher-order functions (see Section 2.2) in functional languages, thereby establishing a one-to-one correspondence between the two.

Four skeletons were presented in [Col87], together with analyses of their implementation on a 2-D mesh and the corresponding asymptotic performances. Examples of problems that fit into each category were also presented.

- Recursive Divide and Conquer (RDC) - This is a skeleton that describes the well-known technique of solving a problem by recursively dividing it into smaller instances of the same problem. When an instance of the problem is not divisible further, it is solved by some non-recursive method. The skeleton requires that the degree of recursion (i.e. the number of sub-instances generated) be specified.

- Task Queue - The technique is suitable for problems whose instances and solutions may be represented by a shared data structure and whose solutions are obtained by repeatedly executing some task. Tasks are maintained in a queue from which processors pick them up and process them. Any tasks that are created as a result, are placed back on the queue. The procedure is repeated until the queue becomes empty.

- Iterative Combination - This skeleton deals with problems that would be described by a set of objects, together with some rule or criterion for combining pairs of objects. The algorithm performs a series of iterations and in each iteration, pairs of objects are combined according to the rule. The iteration stops when either all the objects have been combined into one or when no more combinations are possible.

- Cluster - This is a general-purpose skeleton, obtained as a result of combining the RDC and Iterative Combination skeletons.

The four skeletons described were recently implemented in paraML [BN93], and the details can be found in [Bai94]. A number of other approaches have been motivated by the work just described. Some of them are described in the following paragraphs.

The work of Darlington *et al* [D$^+$93, DT93, DTG93] is an immediate follow-up of the ideas presented in [Col88]. A set of higher-order functions are used to express skeleton-based parallelism. Program transformation techniques are used at different levels of the development process. At the highest level it is used to transform an existing program specification to one in terms of skeletons. Transformations to convert one skeleton form into another are used with a view to improving portability and efficiency. The set of skeletons available to a programmer include:

- PIPE - for exploiting pipeline parallelism
- FARM - for exploiting data parallelism, where each processor is responsible for a part of the data
- DC - for problems requiring the divide and conquer approach
- RaMP - (Reduce and Map over Pairs) - for problems where interactions between pairs of objects are calculated and the results are combined to produce the final result for each object

- DMPA - (Dynamic Message Passing Architecture) - for exploiting parallelism in programs where the inter-process communications are not predetermined, but are determined at run-time, depending on the values of data.

Performance models are used for each skeleton-machine pair, in order to enable the prediction of its performance on the corresponding machine. [DTG93] describes a prototype implementation on a transputer-based machine and presents the results of implementing a ray-tracing example. The performance of the parallel implementation of the program is shown to be in accordance with that predicted by the performance models.

P^3L (Pisa Parallel Programming Language) [DM+92, BD+93, Pel93] is a parallel language that is aimed at exploiting parallelism in distributed-memory MIMD machines. A set of constructs for expressing parallelism is embedded in the imperative language C++. These constructs include:

- **farm** - to exploit data parallelism
- **pipe** - to exploit pipeline parallelism
- **map** - to apply a given function to all the elements of a vector
- **geometric** - to express general data parallelism in one and two-dimensional arrays
- **reduce** - reduces a vector by applying a binary associative operator to all its elements
- **tree** - to process tree-structured computations
- **loop** - to handle iterative computations

The language incorporates a **sequential** construct for expressing the sequential portions of the code. A program is expressed as a composition of parallel constructs, each of which may contain sequential blocks of code or other parallel constructs. Data communications between constructs is specified by using **in(...)** and **out(...)** parameter lists at the interfaces of the parallel constructs.

P^3L is machine-independent in that it is compiled down to an abstract machine P^3M, which in turn is implemented on top of a real machine. The mapping of the abstract machine to a real machine is achieved by the use of a set of libraries - the *mapping* library and the *optimisations* library, which use the details of the target architecture to determine efficient implementations. The compiler chooses one of several possible implementations for each construct and the actual code for these constructs is maintained in the *process template* library.

SkelML [Bra94, Bra93] is a skeleton-based prototype compiler for ML. Parallelism in programs is implicitly expressed by the use of a small set of predefined skeletons. The skeletons are expressed as higher-order functions which are identified by the compiler. The user is also required to provide representative data sets for the program since the compiler relies on execution profile information in order to decide on the portions of the program that can be efficiently executed in parallel. The compiler performs certain optimising transformations and generates Occam 2 code for the Meiko Computing Surface. Six skeletons are supported: **map**, **filter**, **fold**, **filtermap**, **mapfilter** and **foldmap**, of which the last three are combinations of the first three. A process pipeline which allows for compositions of the six skeletons is also supported. A limitation is that the process pipeline is the only means of nesting skeletons and no other skeletons can be nested within others. Although a limited number of skeletons are catered for, the design of the compiler allows for this set to be extended. The results of applying the compiler to three example programs are presented. The performance of all the programs seems to indicate a satisfactory performance by the compiler.

PUL (Parallel Utilities Library) [BCMT93] is a skeleton-based library that provides utilities which support common parallel programming paradigms such as task farms, divide-and-conquer, spatial domain decomposition and mesh-based problems. Programs can be written in C or FORTRAN. PUL was developed on top of CHIMP (Common High-level Interface to Message Passing), and the different skeleton modules can be combined by the use of the explicit communication primitives in CHIMP.

Parsec (Parallel System for Efficient Compilation) [FSWC92, FW93] is a parallel programming environment based on skeletons. The skeletons supported are processor farms and divide and conquer, which are implemented on a logical tree of processors, which in turn can be mapped onto the real machine. Each skeleton is parametrised on information such as number of processors, topology and granularity, and also has performance models associated with it. Performance information is gathered during test runs and and an analysis of this information, together with the performance model provides the values for the parameters of the skeleton. A graphical interface allows the programmer to tune the application according to requirement.

[DDD95] addresses the issue of performance prediction for skeletons. The execution time of a skeleton is described by a generic *higher-order complexity function*. The time complexity of a particular application is derived from the values of its parameters when the skeleton is instantiated. A measure of the scalability of the application is derived using isoefficiency functions. The method is illustrated on some examples from image processing and the experimental results, both for the time complexity and scalability, are shown to closely match the theoretical predictions.

A number of techniques focus on the development of parallel programs using the transformational approach. A skeleton-based parallel programming environment is augmented with special transformation rules which are used in deriving efficient parallel programs. An example of this approach has already been discussed in [D^+93]. [Gee94] describes a framework for parallel program development using skeletons and transformations. As in [D^+93], the emphasis is on inter-skeleton transformations, in order to enhance portability. In an effort to demonstrate the expressiveness of skeletons and the power of inter-skeleton transformations, the implementations for four example problems on two different architectures are derived using a formal method. A similar approach is emphasised in [BGP93], and a processor array is characterised in terms of skeletons.

The Bird-Meertens Formalism (BMF) [Bir87a] is a calculus for deriving efficient programs from problem specifications. A theory in BMF describes the behaviour of a datatype (e.g. a list), and provides a set of operators on that datatype [Bir89], together with some transformational rules relating the operators. (For details on the set of BMF operators on lists, see Chapter 4). The theory was originally developed in the context of sequential programming, in order to provide a formal basis for step-by-step transformation of a, possibly, inefficient specification of a program into an efficient one. Additionally, the BMF operators are implicitly parallel, which can be exploited in developing a parallel programming model. In particular, the operators on the list datatype are described by higher-order functions and programs written in terms of these operators allow for a high level of abstraction. This means that programs in BMF can be written independent of the architecture at which they are targeted, while the strong theoretical foundation of BMF enables the verification of program correctness and also aids formal parallel program development. These attractive features of BMF have prompted researchers to advocate it as a model of parallel computation [Ski92, Ski91]. Also, since the behaviour of the functions in BMF is predetermined, it is possible to compute their costs of execution [SC93], in an effort to predict the performance of the program.

Many recent approaches to the derivation of efficient parallel programs use the semantics-preserving transformations in BMF to obtain efficient parallel programs from initial abstract specifications. An example of such an approach can be found in [GL93], where the derivation of a parallel implementation for divide-and-conquer applications is described. An initial specification in the form of a mutually recursive functional definition, is subjected to formal refinement using the correctness-preserving transformations in BMF. The first phase of the derivation results in the construction of a parallel functional program scheme, from which an imperative distributed SPMD program is derived. Efficiency figures in the range of 0.6-0.9 are reported on example implementations on a 64-node transputer system. A similar technique is used in [Gor95], in which BMF is used in the derivation of a parallel program for polynomial multiplication. Starting with a mathematical specification, an SPMD program is derived, together with a process topology for the program. This topology must, in turn, be mapped onto a real machine. BMF expressions are used to make decisions regarding data partitioning and interprocessor communications. [Roe94] discusses the derivation of efficient parallel programs for SIMD and MIMD machines using BMF. A process of upward refinement is used, in which successive steps involve the incorporation of more machine-dependent implementation details into the program.

2.2 Functional Programming

Functional languages have their foundations in mathematical logic. Functional programs can be viewed as a set of rules that describe *what* to do as opposed to *how* to do it. Functional languages are based on Church's lambda-calculus [Chu41] and indeed these languages are sometimes referred to as *lambda calculus* *with syntactic sugar.* Expressions in the lambda calculus can be evaluated in any order, (including in parallel), without affecting the final result. Also, in the evaluation of a lambda expression, side effects cannot occur. This means that an expression can be replaced by its value at any point in the program and this property is known as *referential transparency.* Functional languages that do not allow side effects are *pure* and those that do allow side effects are *impure.* ML [MHT89] is an *impure* functional language, while Hope [BDS80] is *pure.* The property of *referential transparency* is lost in impure languages. More details of functional languages and the lambda calculus may be found in [Mic89]. Functional languages can also be classified as *strict* or *lazy.* In the case of *strict* languages such as FP [Bac78], the arguments of functions are evaluated before the function. In the case of *lazy* languages such as Haskell [H^+92], the evaluation of function arguments are delayed until their values are actually required.

Although functional languages have their roots in the lambda calculus, they incorporate a number of features that make them less rigid and more user-friendly. Functions are treated as first class objects and a program in a functional language comprises entirely of a set of functions, each of which has well-defined input parameters and returns a result value. Functions can be written so that they can be applied to a set of different data types. This is made possible by *polymorphic type checking* [Mil78]. Most functional languages have *type inference* facilities whereby the type of an identifier or function is deduced from the type information of other identifiers and functions. Type inference is possible because these languages are *strongly-typed* and provide for *static type checking.* A function can be *partially applied* to only the first few of its arguments. The evaluation of this function returns a new function that is applied to the remainder of the arguments. Such functions are said to be *curried* [Ull94], and the arguments to functions can therefore be represented without the need for any brackets. Functions can be passed as arguments to functions and can also return functions as results. Functions that possess either, or both, of these properties are called *higher-order* functions and provide for a very *high level of abstraction* in expressing programs.

2.3 Parallelism in Functional Languages

Functional programs contain implicit parallelism. This is because of their foundations in the lambda calculus, where expressions can be reduced in parallel without affecting the outcome. The abstraction power in these languages can be used to hide the low level details of parallelism from the programmer, therefore allowing for easier program development. A number of approaches for extracting parallelism from functional languages have been considered. Some of these approaches focus on explicit methods, where the onus of identifying and exploiting parallelism is on the programmer. These approaches tend to incorporate some form of *annotations* in the program, and these annotations indicate the parts of the program that must be evaluated in parallel. Implicit approaches focus on techniques that can identify potential parallelism and parallelise a program with little or no involvement from the programmer. The techniques surveyed in the following sections include those based on *dataflow* and *graph reduction.*

2.3.1 Dataflow Techniques

This section gives a brief introduction to the concept of dataflow computing. Since it is not of direct relevance to this thesis, it is merely meant to serve as a pointer to further reading.

In the dataflow model of parallel computation, a program is represented by a directed graph [DK82], in which the nodes represent instructions and the arcs represent the paths of the data-tokens. A node executes the instruction (or *fires*), when data becomes available on all of its input arcs and places the result on its output arc. Several nodes can *fire* simultaneously, offering scope for very fine-grain parallelism. Features include, absence of side effects, implicit parallelism, and *single assignment*, whereby a variable can be assigned a value only once, which makes the binding unchangeable. Languages based on the dataflow concept include, most notably, Id (Irvine dataflow) and SISAL (Streams and Iteration in a Single-Assignment Language). Id [Eka91] is a functional language which has been implemented on architectures such as the MIT Tagged-token dataflow architecture [AN87] and the MIT/Motorola Monsoon Dataflow System [HCAA93]. SISAL [Ske91] is a functional language with imperative constructs such as **WHILE** and **FORALL** loops, for solving scientific problems. Techniques for the automatic detection and exploitation of parallelism in SISAL programs are discussed in [Sar89]. The technique uses execution profiles of programs and information about the target architecture characteristics in deriving efficient parallel implementations.

2.3.2 Techniques Based on Parallel Graph Reduction

One of the more popular techniques for implementing functional languages is by a method known as *graph reduction.* A program in a functional language can be transformed into some form of the lambda calculus [Jon87]. In evaluating such a program, a lambda expression is represented as a directed graph in which operators, constants and variables are represented as leaves. Lambda abstractions are represented by *lambda* nodes and function applications by binary *apply* nodes. The left child of an *apply* node represents the function to be applied to an argument, which is represented by the right child. The graph reduction proceeds by repeatedly overwriting the *apply* nodes by applying the function on its left branch to its argument on its right branch. Common subexpressions are, therefore, *shared* by means of a pointer. This ensures that a subexpression is evaluated only once and the value can be used by all the other expressions that have a pointer to it. If the body of a lambda abstraction contains free variables, then graph reduction cannot be performed efficiently, since the value of the result will depend on the value of the free variables. *Combinators* are functions (or lambda abstractions) that contain no free variables. It was shown that by using a fixed set of combinators [Tur79], free variables can be abstracted from an expression. This idea was generalised by the introduction of the concept of *supercombinators* [Hug82]. A supercombinator is a lambda-abstraction that contains no free variables and any other lambda-abstractions in its body are also supercombinators. Individual functions in the program can be replaced by supercombinators in order to facilitate efficient graph reduction.

A large number of techniques for exploiting parallelism in functional languages are based on parallel graph reduction [Jon89]. The program graph is reduced in parallel. Tasks comprise of expressions that are ready for evaluation and are stored in a task pool, which may be either central or distributed. They are distributed to processors, which evaluate them. In the evaluation of a task, a new task could be created, and this creates a need for efficient dynamic scheduling and load balancing techniques. A number of parallel graph reduction machines have been built and some of them are briefly discussed. Several of these systems use programmer annotations to identify and exploit useful parallelism and cannot, therefore, be classified as implicitly parallel systems.

- ALICE and Flagship - The ALICE (Applicative Language Idealised Computing Engine) [DR81, HR86] architecture comprises of up to 40 transputers connected by an interconnection network. The tasks are stored in a central pool and processors return any newly-created tasks to the pool. It was one of the first parallel graph reduction machines to be built, but suffered from limitations due to high communication latencies and small grain size

[Kea94]. In order to overcome some of these limitations, Flagship [Kea94] was built. The central task pool was replaced by a distributed program graph, with each processor having its own local memory. In spite of efforts to preserve locality in distributing the program graph, the system did not perform well due to overheads caused by non-local accesses.

- GRIP (Graph Reduction In Parallel) - GRIP [JCSH87] is a machine that is specially designed for the parallel supercombinator reduction of Haskell programs. It comprises of up to 20 Processor Element (PE) boards, with each board comprising of 4 PE's and an Intelligent Memory Unit (IMU), where the program graph is stored. One PE acts as system manager, while the others perform graph reduction. Various strategies for creating (*sparking*) new tasks, such as programmer annotations and dynamic techniques based on current load information [JH92, HJJ94] have been investigated. Reports of speedup on a number of applications coded in Haskell can be found in [Jr93].

- The $< v, G >$-machine [AJ89] - This is a parallel version of the G-machine [Joh84]. It performs the supercombinator reduction of lazy-ML programs. The supercombinators are translated into G-machine instructions. The parallelism is explicitly requested by the programmer, using **SPARK** annotations. Speed-ups have been obtained on example programs, as compared to their corresponding sequential implementations on the G-machine.

- Other Systems - There are several other parallel graph reduction machines that have been built. These include ZAPP (Zero Assignment Parallel Processor) [BS81], MaRS (Machine à Réduction Symbolique) [C$^+$89], HDG (Highly Distributed Graph) Machine [LKB91], etc. Some functional language implementations employ parallel graph reduction techniques. Two such systems are Alfalfa and Buckwheat [Gol89, Gol88], which are parallel implementations for the functional language ALFL on a Distributed Memory Intel iPSC hypercube, and a Shared Memory Encore Multimax, respectively. Concurrent Clean [NSvEP91] is another example of a lazy higher-order language that uses parallel graph reduction based on annotations supplied by the programmer.

2.3.3 Other Approaches

There are several other approaches to data-parallel programming based on functional languages. A few of these are briefly mentioned here, essentially to serve as pointers to further reading. [Jou91] describes a strategy for compiling functional languages onto SIMD architectures. The approach consists of the addition of some primitive data-parallel operators to an enriched form of the lambda calculus. The emphasis is on the construction of all data-parallel operations from this small set of primitive operators. Programs are compiled onto an abstract machine called the *Planar Abstract Machine* (PAM), which is derived from an abstract machine called the *Spineless Tagless G-machine* [JS89]. NESL [Ble93] is a data-parallel strict functional language, which has an ML-like syntax and supports polymorphism. It comprises of data-parallel constructs that can be nested, producing nested parallelism. [PD93] propose a *template*-based approach to parallel programming. An *implementation template* is described as "a parametric process graph that implements a particular parallelism exploitation form onto a given (regular) architecture." Each template has a performance model on a given architecture, thereby enabling the prediction of its performance on that architecture. A slight variant of Backus' FP [Bac78] is chosen as the functional language and the issues and strategies involved in the construction of a template-based compiler for FP are discussed.

2.4 Conclusion and Thesis Objectives

The discussion in the previous sections clearly indicates the suitability of the functional approach in the development of models of parallel computation. Techniques based on Dataflow and Graph Reduction appear to have concentrated on the concept of designing specialised machines to make the corresponding models of parallel computation effective. Also, all problems are handled in an identical manner, as opposed to using specialised approaches for effective implementations of different types of problems. However, the idea in this thesis is to provide a model of parallel computation that could be applied effectively to a variety of problems on different machines. To this end, the method based on algorithmic skeletons seems to be more promising.

In particular, the Bird-Meertens Formalism contains a repertoire of data-parallel, higher-order functions whose performance can be analysed at compile-time. Although it has been pointed out that a model of parallel computation based on BMF is an attractive proposition [Ski92], there has been little research in the development of realistic cost (performance) models for such a model of parallel computation. The most significant contribution in this area has been described in [SC93]. However, the cost model described is at a higher level of abstraction than is desired, in order to produce accurate performance estimates. In particular, communication costs have not been modelled at a level of detail required to reflect practical behaviour. Also, the model only handles parallelism at the level of the outermost higher-order function. This thesis aims to address these issues, with a view to developing a realistic cost model for a model of parallel computation based on the Bird-Meertens Formalism. The main objectives of the thesis can be summarised as follows:

- Investigate the feasibility of a model of parallel computation based on the Bird-Meertens Formalism for distributed-memory MIMD machines.

- Provide an extended set of useful functions based on BMF, in an attempt to make programming in terms of these functions natural.

- Analyse the behaviour of this set of functions on different target topologies. For each target topology, derive cost-effective parallel implementations for all the functions in the set, together with the corresponding cost estimates.

- Develop a cost model that would realistically predict the performance of a program expressed in terms of this set of functions on different topologies. The cost model would use the cost estimates of individual functions together with characteristics of the target architecture and input data structures to select a cost-effective parallel implementation for a given program on a given architecture.

- Study the accuracy of the cost model by testing it on example programs.

Chapter 3

Overview of the HOPP Model

3.1 Introduction

This chapter presents an outline of the HOPP approach to parallel programming. HOPP is based on an implicitly parallel language whose constructs are borrowed from the Bird-Meertens Formalism (BMF) [Bir89] and FP [Bac78]. These constructs are essentially higher-order functions which perform useful operations on lists. Most of these functions are inherently parallel and will henceforth be referred to as *recognised functions*. Since the behaviour of the recognised functions is predetermined, a program which is expressed in terms of these functions can be analysed at compile-time to realise a cost-effective parallel implementation. However, such an analysis is only possible for *regular* problems which are expressed in terms of the recognised functions. The implications of *regularity* in this context are discussed in Section 3.3. The programs are targeted at distributed-memory machines in which the communication costs tend to be significant.

3.2 The Features of the Model

The HOPP model comprises of three parts - the *program* model, the *machine* model and the *cost* model.

- The program model - A program in HOPP is a *composition* of nested instantiations of recognised and user-defined functions. *Composition* in this context refers to functional composition and works from right to left. Each component is referred to as a *phase* of the program.

- The machine model - The programs are targeted at distributed-memory MIMD machines which consist of a set of processors connected by an interconnection network. The topologies of machines considered include the hypercube, 2-D torus, linear array and tree. The data is distributed among the processors of the machine.

- The cost model - For each recognised function, the cost model determines its cost of parallel execution on a given topology. Effectively, the cost model computes the costs of possible implementations of a given program on a given machine topology. The theoretical cost model has been implemented in the form of an *analyser*.

3.3 Language Assumptions for the Model

As already mentioned, the HOPP model is based on a functional paradigm. This section describes the structure of a program in HOPP, along with the implications for parallel implementations which will be considered. The meaning of *regularity* in this context is also discussed.

The only data structure is the list, on which all the recognised functions operate. Lists can be arbitrarily nested and of any type, including standard or user-defined. The current prototype implementation of the analyser and parallel code library only allow lists whose base elements are of standard type or pairs of standard types, but this is only a limitation of the implementation. A further assumption which is made by the analyser is that sublists are of equal length.

This is implicit in the assumptions of *regularity*, which will be discussed shortly.

At the top level, a program is a composition of functions. Each component (phase) could correspond to a nested instantiation of recognised and user-defined functions. A program could consist of one or more phases and this is the *only* allowable structure for a program in HOPP. The following is an example of a program in HOPP. *plus* and *times* correspond to user-defined functions, which add and multiply two integers, respectively. The program comprises of three phases, each with instances of recognised (represented in boldface) and user-defined functions.

prog xs = (**s_fold** plus 0 ∘ **map** (**s_scan** plus 0)
∘ **r_cross_product** times ys) xs;

If the topmost level of a phase corresponds to a user-defined function, then any instances of recognised function(s) that it might have as its argument(s) is(are) not considered for parallel implementation. If the topmost level of a phase comprises of a recognised function, then parallel implementations are considered for its argument recognised function. This procedure is followed up to a maximum of three levels or until a user-defined function is encountered. This implies that parallel implementations are considered for nested recognised functions up to three levels and any recognised functions below it are treated as user-defined functions and implemented sequentially. The reasons for considering parallel implementations for up to only three levels are purely pragmatic, and will be explained more clearly in Section 3.5.

Consider the following examples.

prog_1 = **map** (**s_fold** plus 0) xss;

prog_2 = **map** S xss;

fun S xs = **s_fold** plus 0 xs;

```
prog_3 = map (s_fold plus 0 ∘ map sqr) xss;
prog_4 = map T xss;
         fun T xs = let val ys = map sqr xs in
                            s_fold plus 0 ys
                        end;
```

The recognised functions are depicted in boldface and ML notation is used. The function *plus* simply adds two integers and the function *sqr* produces the square of an integer.

All the four programs consist of a single phase with nested instantiations of recognised and user-defined functions, in the restrictive format allowed by the HOPP model. *prog_1* and *prog_2* are equivalent in a sequential setting. However, **s_fold** is encapsulated within the user-defined function, *S*, in *prog_2*. The current implementation of the HOPP model will only *recognise* **map** as a potential function for parallel execution in *prog_2*, and **s_fold** within *S* will only have a sequential implementation. However, in the case of *prog_1*, **s_fold** is at the top level and will, therefore, be *recognised* as a candidate for parallel evaluation, in addition to **map**. Again, *prog_3* and *prog_4* perform the same functions, but the former has more identifiable parallel implementations than the latter. In *prog_4*, a parallel implementation will be considered only for **map**, but none of the instances of recognised functions within *T* would be treated as such. The limitation arising from cases such as *prog_2* can be easily rectified, by checking for instances of recognised functions in the top level of user-defined functions. However, for cases such as *prog_4*, where instances of recognised functions can be arbitrarily nested within user-defined functions, the problem becomes harder to tackle.

User-defined functions are implemented in a chosen strict functional language, following its syntactic rules. The HOPP model is independent of the base language. However, user-defined functions are not allowed to perform input-output operations. The data is input on a single processor which distributes it to other processors as required, during the intermediate phases of the program. The output is produced at the end of the last phase in the program and could be left distributed across the processors. This is the only allowable form of data flow, which could be violated by allowing sequential functions to perform input-output operations.

The implications of *regularity* in this context restrict polymorphism. The analyser needs type information at compile-time in order to compute the size of the base elements in the input list at every stage in the program. An accurate knowledge of this size is crucial to the computation of communication costs. Consequently, definitions of sequential functions that allow for full or restricted polymorphism are not permitted. The type-checker in the analyser would force the specification of the required type information. The following examples illustrate the point.

```
ex_1 = map g xs;
        fun g (x,y) = ((if (y > 0) then ~y else y), x);
ex_2 = map all_eq xs;
        fun all_eq (x,y,z) = (x = y) andalso (y = z);
```

In *ex_1*, the user-defined function g is polymorphic in x. Consequently, its size cannot be deduced at compile-time and the analyser will not allow such a definition. The programmer will be forced to specify the *type* of x. In *ex_2*, the user-defined function *all_eq* is a valid restricted polymorphic function. However, since the types of x, y and z cannot be deduced at compile-time, such a definition cannot be allowed. It may be noted that both of the above definitions will be allowed if type information is explicitly specified.

A regular program in this context is related to what is termed as a *shapely* one in [Jay95]. The analyser performs operations similar to *shape analysis*, whereby, given the *shape* of the inputs, the *shapes* of all the intermediate values and that of the result can be deduced. For a program that is not regular (*shapely*), these deductions will not, in general, be possible, leading to poor performance prediction. Shapely programs also have predictable communication structures. The latter is imposed by the use of recognised functions which only allow the expression of certain types of computation. This means that it would be difficult to express many irregular problems using the set of recognised functions.

Regularity also implies that the performance of a program does not vary drastically for different data sets. This is a limitation arising due to compile-time analysis, but cannot be enforced by the analyser. Although it is possible to write problems which are not regular in this sense, it cannot be guaranteed that the behaviour predicted by the analyser will be obtained experimentally in such cases. (Refer to Section 5.1.4 for an example).

3.4 The Advantages and Limitations of HOPP

The motivation behind choosing a language based on BMF as a model of parallel computation lies in the advantages that it offers.

- The HOPP model is based on a functional paradigm. It therefore automatically inherits all the advantages of functional programming as described in Chapter 2.

- BMF incorporates functions that perform operations which are characteristic of several common parallel programming paradigms.

- A program which is expressed in terms of the recognised functions is analysed *statically* to realise a cost-effective parallel implementation. This is possible because the behaviour of the recognised functions is predetermined.

Since all the analysis is performed at compile time, it saves on overheads at run time. However, the analyser would require information regarding machine-specific parameters and *shapes* of input lists.

- The recognised functions are either already part of, or can be easily defined in any existing functional language. HOPP does not impose any new programming technique on the programmer. Simplicity of learning and use are naturally inherited by the model.

There are, however, some limitations to the scheme.

- As discussed in Section 3.3, the analysis can be assumed to reflect experimental behaviour only for *regular* problems.

- For programs that do not contain any occurrences of recognised functions, a parallel implementation cannot be realised. This forces the programmer to remain within the fixed repertoire of available functions.

- The current implementation uses the list as the main data structure and this poses problems relating to efficiency. A sequential implementation incurs overheads due to list creation, destruction and garbage collection. Also, list access is linear in the length of the list as opposed to the constant-time access of the array. A parallel implementation using list data structures suffers from an additional overhead which is incurred during the communication of lists between processors. In order to ensure that pointer references are accurate after communication, the pointer addresses must be converted to offsets at the sending end. The processor that receives the list must then compute the real addresses from the offsets. This effectively increases communication costs. However, this limitation is not a direct consideration of the HOPP model.

3.5 The Parallelisation Scheme

A program in HOPP is expressed as a *sequence* of phases. Each phase may contain recognised functions along with instances of user-defined functions. Each recognised function has a predefined parallel implementation on a given target machine topology, along with an associated implementation cost. A knowledge of the target machine topology is essential in order to select a cost-effective implementation associated with that topology for the recognised function. Each implementation attempts to make optimal use of the machine connectivity in an effort to reduce the communication overhead. The performance of the recognised functions on the hypercube, 2-D torus, binary tree and linear array topologies have been studied and will be discussed in Chapter 6. User-defined functions only have a sequential implementation and will henceforth be referred to as *sequential functions*.

In the current scheme, parallelism is only exploited within each phase. The phases themselves are sequential and phase i does not commence until phase $i-1$ is completed. However, future work could consider pipelining as an option, in order to evaluate the phases in parallel. A phase that does not contain any occurrences of recognised functions is implemented sequentially. The parallelisation strategy exploits parallelism in nested recognised functions up to the first three levels. The number of levels exploited for parallelisation is limited to *three* for pragmatic reasons. A phase that has its top three nested functions as recognised ones, will have an input data structure that is at least a list of list of lists and eight possible implementations will be considered for it (see Section 3.5.1). Exploiting more levels for parallelism would further increase the number of implementations to be considered. Parallel implementations are not considered for recognised functions which are nested within sequential functions.

The HOPP model enables architecture-independent parallel programming.

Only *one* program is written, irrespective of the architecture at which it is targeted. However, the decisions which influence the selection of a cost-effective parallel implementation for the program, are dependent on the characteristics of the target architecture. These decisions are now transparent to the programmer. The model allows for portable programs to be written, and at the same time hopes to achieve a realistic reflection of parallel program performance. The model is therefore *parametrised* on the characteristics of the target architecture.

The Bird-Meertens Formalism includes a basic set of higher-order functions. This set has been extended to incorporate a number of *additional* functions. The definitions for all the recognised functions can be found in Chapter 4. It will be shown that all of these functions can be expressed in terms of one or more of the basic set of functions. They have been included as recognised functions in their own right because they are found to be useful in many common problems. The definition, in terms of existing functions, is rather contrived for some of the functions in the extended set. It would therefore save time and effort for the programmer if these functions are already available as recognised functions. It would also make the program itself more readable. More importantly, the scheme attempts to provide a more efficient implementation for programs expressed in terms of these additional functions, in comparison to one based solely on the functions in the basic set.

3.5.1 The Analyser

The outline of the parallelisation scheme is depicted in Figure 3.1. The application program is input to the analyser which first constructs a *program tree*. Each branch in the tree corresponds to a phase of the program. A cost analysis is then carried out on the program.

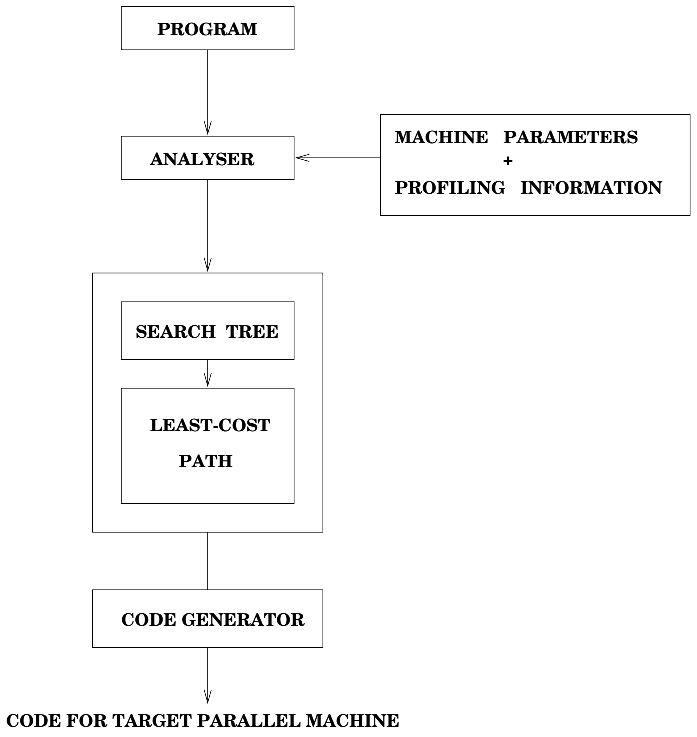


Figure 3.1: The Analysis and Implementation Scheme

The cost of a program comprising of n phases is given by:

$$Cost = \sum_{i=1}^{n} C_{p_i} + \sum_{i=0}^{n-1} C_{i,i+1}$$

where, C_{p_i} is the cost of phase i and $C_{i,i+1}$ represents any communication cost that may be incurred in rearranging the output data of phase i to suit the implementation of phase $i+1$. The cost of a phase depends, among other things, on the nature and the number of recognised functions in that phase and also the parallel implementation selected for that phase, on a given p-processor network. In the present scheme, a phase that has only one occurrence of a recognised function has only one parallel implementation, namely, the parallel implementation for that function on the particular network. For a phase containing two recognised functions, one of which is the argument of the other, three parallel implementations are possible.

- A parallel implementation for the outer function on p processors.
- A parallel implementation for the inner function on p processors.
- A parallel implementation for both functions.

The first two implementations are straightforward. The manner in which the third implementation is handled would depend on the type of the underlying processor network. For example, in the case of a d-dimensional hypercube, the hypercube is divided into 2^k $(0 < k < d)$ smaller hypercubes, with each smaller hypercube containing 2^{d-k} processors. The outer function is evaluated in parallel across 2^k hypercubes, with the inner function being evaluated in parallel across 2^{d-k} processors.

A similar argument can be extended to a phase containing three or more recognised functions, leading to a total of seven possible parallel implementations - three possibilities arising from implementing only one recognised function in parallel, three possibilities arising from implementing any two functions in parallel and the case in which all the three functions are evaluated in parallel. However, parallel implementations are only considered for up to three recognised functions in a phase. Any recognised function(s) below the third recognised function are implemented sequentially. A phase can therefore have at most eight possible implementations including a sequential one. It is important to consider the sequential implementation as well, since it may prove to be the least-cost implementation in some of the cases.

3.5.2 The Search Tree

The costs associated with all possible implementations for each of the phases are estimated by the analyser and a search tree is constructed. The nodes at level i of the tree correspond to the costs associated with the different implementations for that particular phase. These costs include both computation costs and communication costs. The most efficient implementation for the whole program is determined by the least-cost path in the search tree, for which code can then be generated and executed on the parallel machine.

It is important to realise that the analyser does not account for costs arising from low level operations such as memory accesses. This would make the analyser very machine-specific. The selection of the least-cost implementation depends only on cost comparisons and therefore the absolute costs are not crucial. List processing costs are, however, accounted for by the model since this forms a substantial overhead in functional languages. This includes the costs which are incurred in constructing a new list or traversing a list. The analyser estimates these costs based on the nature of the input list and the details are transparent to the user.

It is clear that the size of the search tree will grow exponentially with the number of phases in the program. In this thesis, only programs that comprise of a few phases are discussed, and this keeps the search tractable. However, for problems comprising of a large number of phases, some heuristics for pruning the search tree would have to be considered. This issue is discussed in greater detail in Chapter 5.

3.5.3 Profiling Information

In order to make realistic cost predictions and select an efficient implementation, some estimates of the input data sizes and the costs of sequential functions are required. This information could be obtained by incorporating profiling and type checking capabilities in the analyser. The current implementation of the analyser does not include these capabilities, and the user specifies this information.

In order to estimate communication costs, information such as start-up time, denoted by K_0, the bandwidth of the communication channel, denoted by K_1, and the size of the data to be communicated is required. K_0 and K_1 are machine-specific parameters. A linear model of communication is assumed. The size of the data to be communicated, and therefore the communication cost, also depends on the number of list elements and the size of each element. The size of the input list could be obtained by using profiling information. The size of each base element depends on its type and this could be deduced by a type-checker. This is made possible because, as already discussed, polymorphic functions are not permitted. These issues are discussed more elaborately in Chapter 5.

3.5.4 The Code Generator

The *code generator*, as shown in Figure 3.1 would generate code for the target parallel machine with appropriate communication constructs inserted. However, a fully-fledged code generator has not been implemented. To provide preliminary evidence of the performance of the HOPP model and to assist in program development, some support is available in the form of a library of functions. This library contains the code for the various recognised functions, and also code for performing various types of communications on a particular parallel machine topology. The code in this library is used by all the problems in the performance study. The actual calls to the functions are at present generated by hand. The use of the same code ensures that performance figures for the different examples can be sensibly compared.

3.6 Summary

Many of the recognised functions are also used in the skeletons approach [Col89, Col88, Col87, D$^+$93, HH93]. As in the case of the skeletons, HOPP aims to provide a platform for developing parallel programs where the programmer is not explicitly responsible for parallelism. However, the idea is to be able to express programs that may not entirely match existing skeletons. In that sense, HOPP can be viewed as a more fine-grained approach to parallelising programs. Also, the emphasis is on cost analysis of programs, in an effort to obtain cost-effective parallel implementations.

Chapter 4

The HOPP Model

The Bird-Meertens Formalism includes a set of functions which have useful data-parallel properties. These functions form the basis for the set of recognised functions in the HOPP model. The *extended* set of recognised functions contains additional functions which are commonly encountered. In this chapter, definitions are given for the recognised functions in the basic set, and the additional functions in the extended set. The choice between introducing additional functions with new implementations or as compositions of existing functions, is guided by formal as well as practical considerations. In theory, a function can be made a recognised one, if the following attributes can be provided:

- A definition for the function in terms of one or more of the existing functions.
- A parallel implementation for the function on each of the processor interconnect topologies catered for by HOPP.
- A cost estimate for each parallel implementation of the function.

In practice, the decision to include a function as a recognised one is additionally based on its usefulness. Also, if the cost associated with the parallel implementation of the newly-coined recognised function is much less than that of its composing functions, then it is probably justified to include it as a recognised function in its own right. In some cases the definition of the function in terms of existing functions may be so contrived that it may be sensible to include it as a recognised function in its own right.

4.1 The Basic Set of Functions

The recognised functions which have been borrowed from the Bird-Meertens Formalism are defined below. The ML-style [MHT89, Tof89] notation is used in all the definitions. Many of the functions are defined informally, but some of the functions are clearer if formally defined. However, either notation has no implications for the implementation, which is sequential.

1. **map** - applies some function f to each element of the argument list.

   ```
   fun map f [ ] = [ ]
       map f (x::xs) = (f x) :: map f xs
   ```

2. **fold** - combines the elements of a list using a binary operator.

   fold $\oplus$ a $[x_1, x_2, \ldots, x_n] = (\ldots((a \oplus x_1) \oplus x_2) \ldots) \oplus x_n$

   When a call is made to **fold**, the value of a must be the identity of the f operator. The HOPP model assumes that the argument function f, in the **fold** definition is always associative. If this is not the case, two functions, **foldl** and **foldr**, can be defined, corresponding to operators that are left-associative and right-associative respectively. However, only if the argument function f is associative, can **fold** be implemented in parallel. Different parts of the list are reduced on different processors in parallel. The partial results on the different processors are then combined to produce the final result. If f is not associative, parallel evaluation will produce incorrect results. The assumption of associativity could be removed by making two versions of **fold** available. The onus would then be on the programmer to use the correct version that does not make any assumptions about the binary operator if it is not associative, in which case a sequential implementation would be chosen by the analyser.

The HOPP model provides two versions of **fold** for quite another reason. The following example illustrates the difference between the two versions.

(a) The first example defines a simple function that calculates the sum of a list of integers.

  fun sum xs = fold plus 0 xs

  where, fun plus x y = x + y

  If xs = $[1, 2, 3]$ then a step-by-step sequential execution of **fold** results in the following:

  i. fold plus 0 $[1, 2, 3]$
  ii. fold plus (plus 0 1) $[2, 3]$
  iii. fold plus (plus 1 2) $[3]$
  iv. fold plus (plus 3 3) [ ]
  v. 6

  It should be noted here that the size of the emerging result remains constant in every step of the **fold** operation.

(b) The second example defines a function that flattens a list.

  fun flat xs = fold app [ ] xs

  where, fun app xs ys = xs @ ys

  Let xs = $[[1, 2], [3, 4]]$.

i. fold app [ ] [[1, 2], [3, 4]]

ii. fold app (app [ ][1, 2]) [[3, 4]]

iii. fold app (app [1, 2] [3, 4]) [ ]

iv. [1, 2, 3, 4]

In this case, it is clear that the size of the emerging result *grows* by an amount equal to the size of each list element, after every step of the **fold** operation.

The function **s_fold** (*static*) is introduced to express computations typified by the example in 2a, and similarly the function **g_fold** (*growing*) expresses computations of the type in example 2b. Syntactically, there is no need to distinguish between the two types of computations. However, the execution cost for a parallel implementation of **fold** also includes communication costs and this would depend on the size of the data being communicated. Given a list with n elements, each of size m, then computations in 2b result in a list of size nm elements. At each step of the **fold** operation, the size of the result grows by m. Example 2a results in a list of size m. It may be noted that in the case of **g_fold**, the argument data structure is at least a list of lists.

In a parallel implementation of **fold**, partial results are communicated to neighbouring processors. The size of the data being communicated at each step has a significant effect on the execution cost and is crucial in determining the choice of parallel implementation. By distinguishing between the two versions of the **fold** function, the programmer is invited to provide further cost information to the analyser which enables a more accurate prediction of execution costs. If the version of the **fold** function is not indicated, then the default assumption is **s_fold**.

3. **scan** - similar to **fold**, but the partial results after each application of the binary operator are also retained in the resulting list.

   scan f a [ ] = [a]

   scan f a (x::xs) = [a] @ scan f (f a x) xs

   Again, f is assumed to be associative. Otherwise two functions, *scanl* and *scanr*, must be defined. A **scan** can be implemented in parallel only if the argument operator is associative. As in the case of **fold**, two versions of the **scan** function are introduced - **s_scan** and **g_scan**.

4. **filter** - removes the elements that do not satisfy a property p, from its argument list.

   ```
   filter p [ ] = [ ]
   filter p (x::xs) = if (p x) then x :: filter p xs
                      else filter p xs
   ```

   Only those elements that satisfy the property p are retained in the result list. The result of **filter** is, possibly, a shrunken list.

5. **inits** - results in a list of lists, in which each sublist contains the initial segments of its argument list.

   inits [ ] = [[ ]]

   inits $[x_1, x_2, \ldots, x_n] = [[\,], [x_1], [x_1, x_2], \ldots, [x_1, x_2, \ldots, x_n]]$

6. **tails** - results in a list of lists, in which each sublist contains the final segments of the list.

   tails [ ] = [[ ]]

   tails $[x_1, x_2, \ldots, x_n] = [[x_1, x_2, \ldots, x_n], [x_2, \ldots, x_n], \ldots, [x_n], [\,]]$

7. **cross_product**

   Two functions are defined for performing the cross-product operation, based on their order of evaluation. The basic operation is the same in both the cases, but the evaluation determines the different ordering of elements in the resulting list.

   - **r_cross_product** - specifies that the cross-product is to be performed in a row-major order.

     r_cross_product f [a_1,a_2,. . .,a_m] [b_1,b_2, . . .,b_n]

     $$\begin{aligned} = [&[(\mathrm{f}\ a_1\ b_1),\ (\mathrm{f}\ a_1\ b_2),\ldots,\ (\mathrm{f}\ a_1\ b_n)],\\ &[(\mathrm{f}\ a_2\ b_1),(\mathrm{f}\ a_2\ b_2),\ldots,\ (\mathrm{f}\ a_2\ b_n)],\\ &\qquad\vdots\\ &[(\mathrm{f}\ a_m\ b_1),\ (\mathrm{f}\ a_m\ b_2),\ldots,\ (\mathrm{f}\ a_m\ b_n)]] \end{aligned}$$

   - **c_cross_product** - specifies that the cross-product is to be performed in a column-major order.

     c_cross_product f [a_1,a_2,. . .,a_m] [b_1,b_2, . . .,b_n]

     $$\begin{aligned} = [&[(\mathrm{f}\ a_1\ b_1),\ (\mathrm{f}\ a_2\ b_1),\ldots,\ (\mathrm{f}\ a_m\ b_1)],\\ &[(\mathrm{f}\ a_1\ b_2),(\mathrm{f}\ a_2\ b_2),\ldots,\ (\mathrm{f}\ a_m\ b_2)],\\ &\qquad\vdots\\ &[(\mathrm{f}\ a_1\ b_n),\ (\mathrm{f}\ a_2\ b_n),\ldots,\ (\mathrm{f}\ a_m\ b_n)]] \end{aligned}$$

   Both these functions apply a binary operator f to pairs of elements taken from each of the two argument lists.

8. **composition** - the composition operation is represented by $\circ$.

   (f $\circ$ g) x = f (g x)

4.2 The Extended Set of Recognised Functions

The functions in the set defined by BMF operate on the *list* type. In the context of the HOPP model, it is sometimes difficult to express certain operations using just these functions. The resulting expressions appear to be rather contrived. This is probably due to two reasons. Firstly, the operations that need to be expressed may not be ones that would typically be performed on list types, but rather on arrays. Secondly, some of these operations may have been devised in the context of parallel programming. The Bird-Meertens Formalism is based on lists and in the context of sequential programming. These difficulties arise when the HOPP model attempts to use the BMF functions to solve array-based problems in the context of parallel programming.

The set of functions in BMF has been extended to incorporate some additional functions which provide *off-the-shelf* recognised functions that can be used effectively for a larger class of problems. These additional functions can be expressed in terms of one or more of the existing recognised functions. This imposes some restrictions on the kind of functions that can be made *recognised functions.* More importantly, it is hoped that the property of amenability to transformations will also be retained by the recognised functions. The possibility of new transformations on the extended set of functions can be investigated and more efficient programs can be derived in the context of parallel programming and the HOPP model. In most of these cases, the newly-coined function has a more cost-effective parallel implementation when compared to the naive implementation corresponding to its composing functions.

The extended set of recognised functions is now described. Each function is first defined using informal ML-style notation, which is followed by its definition in terms of the functions in the basic set of recognised functions. It is to be emphasised that the latter definitions merely serve as illustrations, and there may be other, possibly better ways of expressing the same.

The functions in the extended set are divided into two categories - those which are intuitive and for which their inclusion can be motivated, and some which are not so intuitive, but could be useful in writing programs.

The more important functions are as follows:

1. **map2** - similar to **map**. This function is sometimes referred to as *zipwith*.

   ```
   map2 f [ ] [ ] = [ ]
   map2 f (x::xs) (y::ys) = (f x y) :: map2 f xs ys
   ```

   This is just an extension of **map** to cater for two argument lists. The argument lists must be of the same size.

2. **zip** - pairs up corresponding elements from two input lists, resulting in a list of pairs.

   ```
   zip [ ] [ ] = [ ]
   zip (x::xs) (y::ys) = (x,y) :: zip xs ys
   ```

   zip can be defined in terms of **map2**.

   ```
   zip xs ys = map2 pairup xs ys
   where,
           pairup x y = (x,y)
   ```

3. The iterative functions - this is a set of functions that apply a function *f* to a list, for a specified number of times. After each stage of the iteration, *f* is applied to the result of the previous stage. The number of times that *f* is applied could either be predetermined or conditional. It may be

necessary in some cases, to perform operations on the resulting list before passing it on to the next stage of the iteration, or before the final result is output. This operation may also involve the original list. The function g in the following definitions allows for any transformation on the resulting list. If no such transformation is necessary, then g can be defined to be the identity function. There are three iterative functions which are available as recognised functions.

- **iterate_up** - an iterative function that applies a function f to a list for a specified number of times. After each stage of the iteration, f is applied to the result of the previous stage. A *start* index and a *finish* index are specified, and it iterates until *start* is greater than *finish*. At each iteration, the value of *start* is incremented by one, so that the number of iterations is (*finish* - *start* + *1*).

```
iterate_up finish start f g xs = if (start > finish) then (g xs)
                                 else
                                     iterate_up finish (start + 1) f g
                                                   (f finish start (g xs) )
```

- **iterate_down** - similar to **iterate_up**, but the iteration counter is a down-counter. After each iteration, the value of the start index *start* is decremented by one. The iteration stops when *start* is less than *finish*. The number of iterations is (*start* - *finish* + *1*).

```
iterate_down finish start f g xs = if (start < finish) then (g xs)
                                   else
                                       iterate_down finish (start - 1) f g
                                       (f finish start (g xs))
```

- **iterate_cond** - provides for conditional iteration. The function f is applied to the input list. The resulting list is tested to check whether it satisfies some condition defined by the function *cond.* The iteration is stopped if the condition is satisfied, otherwise f is applied to the result. Sometimes it may be necessary to compare the resulting list with the original list in order to check whether the condition is satisfied. Therefore, the original list is a parameter to the function *cond* which operates on two input lists.

```
iterate_cond cond f g xs = let val result = f xs
                           in if (cond result xs) then (g result xs)
                              else
                                 iterate_cond cond f g (g result xs)
                           end
```

The three iterative functions are not themselves implemented in parallel, although they belong to the set of recognised functions. This is because iteration is inherently sequential - step i uses the results of step $(i-1)$. The functions f and g in the definitions of the iterative functions may be implemented in parallel if they are composed of recognised functions. However, if f and g happen to be sequential functions, then the scheme would only select a sequential implementation for the iterative functions. The details of the parallel implementations are given in Chapter 6.

Iteration on a list can be expressed in terms of function **composition**, which belongs to the set of functions in BMF. An illustration is given for the case of **iterate_up** and the definitions for the other iterative functions are similar.

iterate_up finish start f g xs $\equiv \underbrace{f1 \circ f1 \circ \ldots \circ f1}_{n}$ (f, g, finish, start, xs)

where,

n = finish - start + 1

f_1 (f, g, finish, start, xs) = let val result = f finish start (g xs)

```
in
   (f, g, finish, start+1, result)
end
```

This is a rather informal definition, but it demonstrates that the iterative functions can be expressed in terms of an existing BMF function. The function **iterate_cond** cannot be expressed in the same way as the other two iterative functions since the exact number of iterations is not known. However, a similar idea is applicable.

4. **split** - splits a given list into a specified number of sublists.

split 1 $[x_1, x_2, \ldots, x_n] = [[x_1, x_2, \ldots, x_n]]$

split k $[x_1, x_2, \ldots, x_n] = [[x_1, x_2, \ldots, x_{\lceil \frac{n}{k} \rceil}], \ldots, [x_{(k-1)\lceil \frac{n}{k} \rceil + 1}, \ldots, x_n]]$

The introduction of **split** as a recognised function was motivated by the need to express divide-and-conquer in applications. The function **split** takes two arguments, an integer $0 < k \leq n$ and a list. The result of applying the function to the list is to split the list into k sublists. A list of size n is rearranged to be a list of lists, with $(k-1)$ sublists of size $\lceil \frac{n}{k} \rceil$ and the last sublist being of length $n - (k-1)\lceil \frac{n}{k} \rceil$.

split can be expressed in terms of the basic set of recognised functions. The resulting composition of functions is quite complex.

```
split k xs = let val part = (len xs)/k
             in
                 (fold append [ ] ∘ map (filter (eq part)) ∘ map tails ∘
                  filter (multiple part) ∘ inits) xs
             end
where,
       append xs ys = xs @ ys
       eq n xs = (len xs) = n
       multiple n xs = (len xs > 0) and (len xs) mod part = 0
```

The code for **split** in terms of the existing functions is quite contrived. The cost of implementing it as a composition of functions would be much higher than the cost of the version given in Chapter 6. Given that it is a useful function which is applicable in a variety of problems, the incorporation of **split** in the extended set of recognised functions can be justified.

5. $\mathcal{R}^k$ - A **composition** of functions allows for an input list to be piped through several phases of operation. In each phase, some operation (defined by the function(s) in that phase) is performed on the list. The resulting list is passed on to the next phase of the **composition**. This appears to be the only form of control flow in programs that can be expressed using the functions in BMF.

   It is clear, however, that not all the recognised functions operate on a single input list. Functions such as **cross_product** operate on two input lists. It may be the case that two copies of the input list arriving from the previous phase are its arguments. More generally, it may be necessary to perform some operation on each of the copies before they serve as inputs to the recognised function. The combinator $\mathcal{R}^k$ caters for precisely these situations.

The definition is provided for $\mathcal{R}^2$, the case where a recognised function operates on two input lists.

$\mathcal{R}^2$ F f_1 f_2 … f_n g h xs = F f_1 f_2 … f_k (g xs) (h xs)

where,

F is a recognised function with 2 input lists

f_1, f_2, …, f_n are parameters to F

g and h are functions that operate on lists

Depending on which of the input lists is to be passed unchanged to F, g or h or both would correspond to the identity function. In general, g and h could be any complex functions including other recognised functions.

In general, $\mathcal{R}^k$ is defined for a recognised function operating on k input lists. Obviously, a maximum value must be defined for k and this would depend on the maximum number of input lists to a recognised function in the set.

It is difficult to provide a *formal* justification for including $\mathcal{R}^k$ in the extended set. An intuitive justification is based on the nature of control flow that may be required for some of the functions in BMF. **Composition** only provides for one form of control flow in the program, in which a *single* list is passed from one phase to the next. However, the nature of certain recognised functions suggests the need for a second form of control flow which is provided by $\mathcal{R}^k$.

6. **get_neigh** - obtains the left and right neighbours for every element in the argument list. The result is a list of lists, with each sublist containing three elements - the particular list element, and its left and right neighbours. An informal definition for **get_neigh** is as follows.

get_neigh l-∞ r-∞ $[x_1, x_2, x_3, \ldots, x_{n-1}, x_n]$ =

$[[x_1,$ l-∞, $x_2], [x_2, x_1, x_3], \ldots, [x_n, x_{n-1},$ r-$\infty]]$

In the definition, ∞ indicates no neighbour in that particular direction, implying an end element of the list. l-∞ and r-∞ represent the left and right neighbours for the first and last elements of the list respectively. The first and last elements do not have neighbours, so the definition allows for some boundary values to be specified. In order to define **get_neigh** in terms of the existing set of functions, a function *sequence* which applies a set of functions to its input list is first defined. *sequence* can be defined in terms of **map2**.

sequence $[f_1, f_2, \ldots, f_n]$ xs = [(f_1 xs), (f_2 xs),...,(f_n xs)]

sequence $[f_1, f_2, \ldots, f_k]$ xs = map2 g $[f_1, f_2, \ldots, f_k]$ [xs, xs,..., k copies]

where, g f_i xs = f_i xs

Then, **get_neigh** can be defined in the following manner.

get_neigh l-∞ r-∞ xs = (stage_3 $\circ$ stage_2 $\circ$ stage_1) xs

where ,

stage_1 xs = sequence[($\mathcal{R}^2$ zip id (scan pick_sec l-∞)),
(reverse $\circ$ $\mathcal{R}^2$ id (zip $\circ$ reverse)
((scan pick_sec r-∞) $\circ$ reverse))] xs

pick_sec x y = y

id xs = xs

stage_2 xs = ($\mathcal{R}^2$ id (zip $\circ$ (fold append [ ])) (select 1)) xs

stage_3 xs = map left_right xs

left_right ((x,y),(w,z)) = [x,y,z]

Essentially, *stage_1* performs one left shift and one right shift on the original list. This is achieved by the use of the **scan** function and it obtains the left and right neighbours for each element. The shifted lists are then *zipped* up with the original list. *stage_2* and *stage_3* just produce the output in the required format. The function **select** is defined below.

The code for **get_neigh** in terms of the basic set of functions is rather contrived. This again, is due to the linear nature of operations on the list structure. **get_neigh** is a function that requires an operation which is not typical of list structures - going backwards rather than forwards.

The function **get_neigh** has been introduced in the list of recognised functions both for expressiveness and efficient implementation.

The less intuitive functions are discussed next. These functions also serve to illustrate that in some of the cases, there is no natural way of expressing them in the BMF-style of programming.

1. **len** - returns the length of a list.

   len [ ] = 0

   len (x::xs) = 1 + len xs

   len can be defined in terms of both **map** and **fold**.

   len xs = fold plus 0 ∘ map subst_1

   where,

   ```
   subst_1 x = 1
   plus x y = x + y
   ```

2. **select** - selects the jth element in a list.

```
select $j$ [ ] = error
select $j$ xs = if ($j$ <= 0) then error
                 else if ($j$ > len xs) then error
                      else get $j$ xs
where,
    get 1 (x::xs) = x
    get $j$ (x::xs) = get ($j - 1$) xs
```

Assuming the absence of error conditions, **select** can be defined using a **fold** in the following manner.

```
f ($k, j$,x) $x_j$ = if ($j = k$) then ($k + 1, j, x_j$)
                   else ($k + 1, j$, x)

select $i$ xs = third $\circ$ foldl f (1,$i$,y) xs
where,
    third (x,y,z) = z
```

select is defined in terms of *foldl* which is not associative. However, a suitable parallel implementation for **select** has been defined. This is possible because the behaviour of **select** is predetermined.

3. **apply_select** - applies a function f to a specified set of elements in the input list. The list of indices of the elements to which f is to be applied must be sorted in *ascending* order.

apply_select f $[i_j, i_k, \ldots, i_m]$ $[x_1, x_2, \ldots, x_j, x_k, \ldots, x_m, \ldots, x_n]$ =

$[x_1, x_2, \ldots,$ (f x_j), (f x_k), $\ldots$, (f x_m), $\ldots, x_n]$

apply_select can be defined in terms of **map**.

```
apply_select f $[i_j, i_k, \ldots, i_n]$ xs = map (f 1 $[i_j, i_k, \ldots, i_n]$) xs
where,
      f m (ind::inds) [ ] = [ ]
      f m (ind::inds) (x::xs) = if (m = ind) then
                                      f x :: f (m+1) inds xs
                                   else
                                      x :: f (m+1) (ind::inds) xs
```

4. **copy** - distributes a particular element of the list to the other elements in the list.

   copy i $[x_1, x_2, \ldots, x_n] = [(x_i, x_1), (x_i, x_2), \ldots, (x_i, x_n)]$

   copy can be expressed in terms of **select** and **map**.

```
copy $i$ xs = let val x = select i $[x_1, x_2, \ldots, x_n]$
              in
                map (pairup x) $[x_1, x_2, \ldots, x_n]$
              end
where,
      pairup x y = (x,y)
```

   The index of the element which is to be distributed is an argument to the function **copy**. The result of **copy** is a list of pairs with the first of the pair being the distributed element.

5. **reverse** - reverses a list, as the name suggests.

   reverse $[x_1, x_2, x_3, \ldots, x_n] = [x_n, \ldots, x_3, x_2, x_1]$

reverse can be expressed in terms of **fold**.

reverse xs = g_fold rev [ ] xs

where,

rev xs x = [x] @ xs

4.3 Example Programs

In this section, some well-known problems are expressed in the style advocated by the HOPP model, using the functions from the extended set of recognised functions. The same examples will be later used to test the accuracy of the HOPP model, comparing the execution on a parallel machine with the prediction made by the analyser. Details on the parallel implementations and the results can be found in Chapter 7. The examples have been coded in ML.

4.3.1 Matrix Multiplication

The problem of multiplying two matrices $A_{m \times n}$ and $B_{n \times k}$, which results in the matrix $C_{m \times k}$, is considered here.

The problem can be expressed as a composition of two phases. The recognised functions are depicted in bold face and an informal ML-style notation is used.

fun mat_mult times plus A B = **map** (**map** (**s_fold** plus 0)) $\circ$
r_cross_product (**map2** times) A B^T;

fun plus a b = a + b;

fun times a b = a $*$ b;

B^T represents the transpose of B

Each matrix is represented as a list of lists. Each sublist represents a row/column of the matrix. *Phase one* - multiplies the corresponding pairs of elements from pairs of sublists. *Phase two* - performs the addition to obtain the inner products.

The function **s_fold** is used because the size of the emerging result after each stage of the *fold* operation is a constant, and is equal to the size of the base type of the matrix. The transpose of the second matrix is performed sequentially in this case, and therefore increases the costs of all the implementations by the same amount.

4.3.2 Merge Sort

The *merge sort* example is considered next. A list of integers is to be sorted in ascending order. The strategy adopted is to sub-divide the list into a list of lists, with each sublist containing two elements. These two elements are sorted and the sorted sublists are then merged, two at a time, to obtain a fully sorted list. This is similar to the *divide-and-conquer* approach, except that *divide* is not performed recursively. The list is transformed into a list of lists in a single step, using the **split** function. Each sublist then represents the base case which can be solved. For the sake of simplicity, the number of elements in the input list is assumed to be a power of 2.

The problem can be expressed as a sequence of three phases.

```
fun sort n xs = g_fold (merge [ ]) ∘ map msort ∘
                                   split (n/2) xs;
where, n is the length of the list
        fun msort [x,y] = if(x ≥ y) then [y,x] else [x,y];
        fun merge [ ] [ ] = [ ]
            merge xs [ ] = xs
            merge [ ] ys = ys
            merge (x::xs) (y::ys) = if (x ≤ y) then
                                        x :: merge xs (y::ys)
                                      else y :: merge (x::xs) ys;
```

Phase one performs the *divide* step, in which the list is split into sublists, each of size two; *Phase two* sorts each sublist; and *Phase three* performs the merge operation. The function **g_fold** is used since at each step of the **fold** operation, the size of the resulting list is the sum of the sizes of the two lists that were merged. It may be noted that *phase two* may be removed by splitting the list into singleton sublists. It is retained in this example for illustrative purposes only.

4.3.3 Solving Partial Differential Equations

Elliptic partial differential equations are commonly encountered in equilibrium or steady-state problems. One of the best-known elliptic equations is Poisson's equation, represented by Equation 4.1

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x, y) \tag{4.1}$$

where, f is the source term. The domain of integration of such a two-dimensional elliptic equation is always an area S bounded by a closed curve C. Only a limited number of special types of elliptic equations have been solved analytically. There are several numerical approximation methods available for solving differential equations, of which those employing finite-differences are more frequently used because of their wider applicability. In these methods, the area of integration S bounded by the closed curve C, is overlayed by a system of rectangular meshes. The meshes are formed by two sets of equally spaced lines, each set parallel to the X-axis and Y-axis respectively. An approximate solution is found at the n points of intersection which are called mesh points. The approximation consists of replacing each derivative of the partial differential equation at a mesh point, $P_{i,j}$, by a finite-difference approximation in terms of the values of u at $P_{i,j}$ and the neighbouring mesh points and boundary points. For each of the n mesh points, an algebraic equation approximating the differential equation is written, giving a set of n algebraic equations in n unknowns. Sets of linear algebraic equations can be solved by *direct* or *iterative* methods. Direct methods are based on Gaussian elimination with pivoting or triangular decomposition of the matrix of coefficients. Descriptions and C programs for these methods can be found in [P$^+$88]. The method considered in this section is an iterative method due to Jacobi.

The area S is assumed to be rectangular, with sides of length ph and qh, and to have known values b on the perimeter of S. If S is sub-divided into a network of squares of side h, then the mesh points are defined by:

$$\begin{aligned} x &= ih, \ (i = 0, 1, \ldots, p) \\ y &= jh, \ (j = 0, 1, \ldots, q). \end{aligned} \tag{4.2}$$

Approximating the equation by the five-point difference scheme [Smi65, Ame69] yields:

$$u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{i,j} - h^2 f_{i,j} = 0. \tag{4.3}$$

$u_{i,j}$ is then given by:

$$u_{i,j} = \frac{1}{4}(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - h^2 f_{i,j}). \tag{4.4}$$

If the nth iterative value of $u_{i,j}$ is denoted by $u^n_{i,j}$, then an iterative procedure for solving Equation 4.4 is defined by:

$$u^{(n+1)}_{i,j} = \begin{cases} b_{i,j}; & \text{if } (i = 0, p), \ (j = 0, q) \\ \frac{1}{4}(u^n_{i-1,j} + u^n_{i+1,j} + u^n_{i,j-1} + u^n_{i,j+1} - h^2 f_{i,j}); & \text{otherwise} \end{cases} \tag{4.5}$$

Successive iteration causes the approximate solution to converge to the exact solution.

The program for implementing the Jacobi method is given below. The region S is assumed to be a square, with $p = q =$ JMAX, where JMAX is some predefined constant, $h = 1$ and $b_{i,j} = 0$; $(i = 0, p; \ j = 0, q)$.

```
fun gen_zeros 0 = [ ]
  | gen_zeros n = 0.0 :: gen_zeros (n-1);

val inf1 = gen_zeros JMAX;
val inf2 = gen_zeros JMAX;


fun get_rest [ ] [ ] [ ] = [ ]
  | get_rest [ ] ts [ ] = [ ]
  | get_rest [ ] [ ] bs = [ ]
  | get_rest es ts [ ] = [ ]
  | get_rest es [ ] bs = [ ]
  | get_rest [ ] ts bs = [ ]
  | get_rest (es::ess) (t::ts) (b::bs) =
           (es @ [t,b]) :: get_rest ess ts bs;


fun rearrange [xs, ys, zs] =
             let val res = get_neigh 0.0 0.0 xs
             in get_rest res ys zs
             end;
fun less_than (eps:real) (value:real) = (value < eps);
fun maxi (x:real) y = if (x > y) then x else y;
fun diff (x:real) (y:real) = abs (x-y);
fun update f [u0,u1,u2,u3,u4] = 0.25 * (u4 + u3 + u2 + u1) - f/4.0;
fun id x y = x;


fun test (eps:real) xs ys = ((less_than eps) ∘ (s_fold maxi 0.0) ∘
                            (map (s_fold maxi 0.0)) ∘
                            (map2 (map2 diff) xs) ) ys;
```

```
fun jac fss uss = ((**map2** (**map2** update) fss) ∘
                  (**map** rearrange) ∘
                  (**get_neigh** inf1 inf2) ) uss;


fun pde (eps:real) fss uss =
                   **iterate_cond** (test eps) (jac fss) id uss;
```

The function, *pde*, is the main function that iteratively applies the function, *jac*, to the input matrix *uss*. The source term is represented by *fss*. Both *uss* and *fss* are represented as lists of lists. The function *jac* obtains the neighbouring values for each mesh point and updates its value according to Equation 4.5. The function *test* checks whether convergence has been achieved. After each iteration, the value of a mesh point is subtracted from the corresponding value at the previous iteration. If the absolute value of the maximum of such differences is smaller than some predefined constant, *eps*, then iteration is stopped and the result is the required solution.

4.4 Conclusion

This chapter introduced the set of recognised functions and illustrated their use with three example programs. The programs for matrix multiplication and merge sort were more natural to express in the BMF-style, compared to the program for jacobi iteration. In all the three cases the difficulty of natural expression was caused by the use of the list data structure. The programs are best-suited to array data structures, but the recognised functions only work on lists. A *theory of arrays* [Mil93] with data-parallel operations defined on it would probably result in a more natural data-parallel model of programming.

Chapter 5

The Cost Model

A style of programming is advocated which is based on the set of recognised functions as presented in Chapter 4. An analytical cost model is employed to predict the execution costs for these programs at compile-time. Based on these costs, a cost-effective parallel implementation is selected. The analyser requires some information about machine-dependent parameters, input data sizes and sequential functions, which is discussed in more detail in the following section.

The scheme for obtaining a parallel implementation for a program is shown in Figure 3.1. Chapter 3 gives an outline of the analysis scheme. This chapter discusses the analyser in greater detail.

5.1 The Problem Specification

In principle, the analyser should abstract away from low-level details to the extent possible. However, the costs computed by the analyser should reflect the practical costs accurately. The input to the analyser is in the form of the following tuple:

$$\text{program} = (P, M, D, I_s, F_t, S, C_f, F_s).$$

Symbol	*Meaning*
P	Program Tree
M	Parallel Machine Characteristics
D	Input List Nesting
I_s	Input List Size
F_t	Input List Type
S	Set of Relationships
C_f	Cost of Sequential Function
F_s	Output Type of Sequential Function

The meaning of each symbol will be discussed in the following subsections. Much of the information required could be obtained with minimum programmer interference if profiling [ADM87, Bus93, San93, Sar91] and type-checking capabilities were built into the analyser. However, the current prototype implementation of the analyser does not include these capabilities and the programmer is required to supply the necessary information.

In all future discussions, it is assumed that:

$$\mathcal{H} = \text{the set of recognised functions,}$$
$$\mathcal{F} = \text{the set of sequential functions.}$$

5.1.1 The Program Tree (P)

P is the program tree representing the program. The program tree is indirectly supplied by the programmer; the analyser constructs it from the specification of the program code. Each node in P corresponds to either a recognised function or a user-defined one. Each phase in the program is represented by a branch in the tree. A function G, which is an argument of another function F, is represented as a child node of that representing F, on the branch corresponding to that phase. Since the program is typically a **composition** of phases, each phase is an argument of the **composition** function. The branch corresponding to each phase is therefore represented as a child of the **composition** node. The input lists are not represented on the program tree. However, the number of input lists required

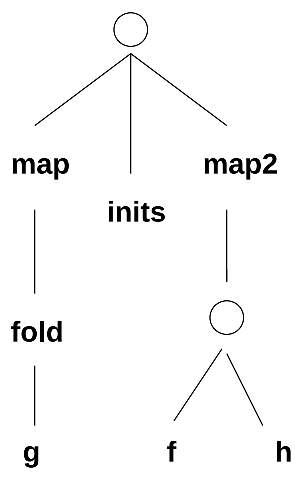


Figure 5.1: An Example of a Program Tree

by each phase is determined by the nature of the recognised functions present in the phase. Henceforth, the terms *branch* and *phase* will be used interchangeably. Therefore, the program,

$$\mathbf{map}\ (\mathbf{fold}\ \mathrm{g}) \circ \mathbf{inits} \circ \mathbf{map2}\ (\mathrm{f} \circ \mathrm{h})$$

would have a program tree as shown in Figure 5.1.

5.1.2 The Parallel Machine Characteristics (M)

M is a 4-tuple which describes the characteristics of the parallel machine on which the program is to be executed and has to be supplied by the programmer. The parallel machine is assumed to have p identical processors, each with the same amount of local memory.

$M : string \times Z_0^+ \times Q_0^+ \times Q_0^+$
$M =$ (topology, number-of-processors, K_0, K_1).

Although the program itself is independent of the architecture on which it is executed, the implementation that will be selected would depend on it. The cost model has therefore been *parameterised* on the characteristics of the parallel machine.

- topology - The choice of parallel implementations for the recognised functions is dependent on the interconnection topologies between the processors. Cost models have been studied for some well-known interconnection topologies, such as the *hypercube*, *2-D torus* and *tree.*

- number-of-processors - This specifies the maximum number of processors available on the parallel machine.

- K_0 and K_1 - In a message-passing system, data communication involves two costs - the start-up cost to initiate the communication and the actual cost of transferring the data. The start-up cost, K_0, is usually a significant portion of the communication cost (and can dominate it, especially when the data size is small), and cannot be ignored. The cost for the actual communication of data will depend on the bandwidth of the communication link, K_1, between the two processors. K_0 is expressed in some time unit (e.g. ms) and K_1 is expressed in bytes/time unit (e.g. bytes/ms). Both of these parameters are specific to a given architecture and can be obtained from the machine manufacturer. A linear model is used to compute the communication costs [SS89] between nearest neighbours. Therefore, the cost of communicating n bytes of data between neighbours is given by the following expression:

$$T_c = K_0 + \frac{1}{K_1}n. \tag{5.1}$$

  It is assumed that only data is being communicated, and that each processor possesses a copy of the source code. It is also assumed that processors cannot communicate with more than one neighbour at a time. Therefore, initiating communications with m different neighbours would incur a start-up cost of mK_0 (as opposed to K_0). The algorithms and costs for data communication routines are discussed in Chapter 6.

5.1.3 The Input List Characteristics (D, I_s, F_t, S)

- D represents the level of nesting in each of the input lists.

  D : string $\times Z_0^+$

  string is the variable that represents the list. For example, for a list called *matrix_A* which is a list of lists, the D would be specified as ("matrix_A", 2).

- I_s is a D-tuple which represents the list sizes at levels, $0, 1, \ldots, (D-1)$.

  I_s : string $\rightarrow \underbrace{(Z_0^+ \times \ldots \times Z_0^+)}_{D-tuple}$

  For example, for the list *matrix_A* which is a list of lists of size (32×64), I_s would be specified by ("matrix_A", (32,64)). I_s could be estimated by profiling. In the absence of a profiler in the prototype implementation, if the input list size is not known at compile-time, I_s is computed after the specification for S is obtained. It is important to note that the size(s) of the input list(s) is(are) required to be specified only at the beginning of the first phase in the program. For subsequent phases, the analyser deduces the size of the input list, based on the transformations applied to it by the functions in the previous phase. However, for a recognised function with two argument lists, if only one input list arrives from the previous phase, then an estimate of the size of the other list will be required.

- F_t is a function that computes the size of each element in level $(D-1)$ of the input list(s).

  F_t : string $\times$ type $\rightarrow Q_0^+$

  In other words, F_t represents the size of the base element of each of the input lists in the program, where *string* is the variable representing the input list.

This is computed from the *type* of each input list, which could be deduced by a type-checker. For example, for the list *matrix_A* which is a list of lists of integers, F_t would be specified by ("matrix_A", int). The *sizeof(int)* function then computes the required size. Programs in functional languages can be written at high levels of abstraction, where a single program could apply to a whole group of list types. The analyser will, however, force the programmer to specify the type of the base element of each input list. In that sense, some of the power of *abstraction* in functional languages is lost. This information, however, is necessary for the analyser to predict communication costs. In the absence of type information, the data size is not known, making the true communication costs difficult to predict.

- S is a $(\binom{D}{2} + D)$ -tuple, expressing the relationship between sizes in different levels (i.e. $0, 1, \ldots, (D-1)$) of the input list(s) and the number of processors.

$S = (R_0, R_1, \ldots, R_{D-1})$

$R_i \in [\ll, <, \approx, >, \gg],\ 0 \leq i < D$

where,

$n \ll k \Rightarrow 0 < \frac{n}{k} < 0.1$

$n < k \Rightarrow 0.1 < \frac{n}{k} < 1$

$n \approx k \Rightarrow \frac{n}{k} \approx 1$

$n > k \Rightarrow 1 < \frac{n}{k} < 10$

$n \gg k \Rightarrow \frac{n}{k} \geq 10$

The set of relationships in S, represents a simple method for expressing constraints on the size of the inputs. The choice of 10 as the factor for expressing the estimates is based purely on pragmatic considerations. For compile-time analysis, some knowledge of the *shape* of the input data is required to guide the selection of a parallel implementation. This might even be different for the same program for different input list sizes. In the absence of a profiler, the specifications in S provide some estimates for the analyser regarding the relative sizes of the input list and the number of available processors. In the presence of profiling information, the specification for S would not be required.

Since the number of processors is known (from the specification in M), and the specifications in S estimate size(s) of the input list(s) in relation to the number of processors, I_s can now be estimated for cost calculations.

5.1.4 Specifications for Sequential Functions (C_f, F_s)

- Costs of Sequential Functions.

  C_f is the cost function for sequential functions in the program.

  $C_f : G_c \cup G_f$

  $G_c : \mathcal{F} \rightarrow Q_0^+$

  $G_f : \mathcal{F} \rightarrow (G \rightarrow Q_0^+)$

  G_f represents the case where the cost of a sequential function is proportional to the size(s) of the input argument(s) and G_c represents the case where the cost is a constant. The cost is expressed in some decided unit, e.g. ms. The cost of a recognised function is a function of the cost of its argument function(s) which could ultimately be a sequential function whose cost is not predetermined. It is therefore necessary to determine the cost of the sequential function(s) in the argument, in order to compute the cost of the recognised function. The costs of sequential functions could be estimated by profiling, in which case the specification for C_f would not be required of the programmer.

An assumption that is automatically made by the analyser is that the sequential functions take the same time to operate on different input data. This assumption arises as a result of the *regularity* restriction that is imposed on problems which can be handled by the scheme (See Section 3.3). For example, in **map** f xs, the cost of f is assumed to be the same for every element of the list xs. This automatically means that sequential functions in this scheme, should not arbitrarily alter the size of the input data. Consider the following example.

```
fun f 0 = [ ]
fun f i = i :: f (i-1)
```

This would produce a list whose size depends on the value of i. However, in **map** f xs, f no longer takes the same amount of time to operate on all the elements of the list xs. The cost of f now depends on the actual value of each list element. This violates the assumption of *regularity* in the context of this scheme. In such cases, the specification of C_f would be difficult and one based on the worst or average-case cost would produce predictions that cannot always be guaranteed to reflect practical costs. Profiling would probably remove this restriction to the extent of allowing sequential functions with costs that depend on the actual input data. However, problems would still be required to have a predictable (*regular*) communication structure in order to be expressible in terms of the recognised functions.

- Outputs from Sequential Functions

  F_s computes the size of the output which is produced by the sequential functions in the program, from the specification of the type of the output of a function. This could, again, be deduced by a type-checker.

$F_s : \mathcal{F} \times \text{type} \rightarrow Q_0^+$

Again, F_s imposes some limitation on the full abstraction power of functional languages. However, this information is also important to the analyser for predicting communication costs. In each phase of the program, the input list is transformed by the functions in that phase. Although the nature of the transformation of the input list(s) by a recognised function is predetermined, the manner in which a sequential function would transform its argument list(s) cannot be determined, if the program is written at a high level of abstraction. A sequential function could transform a base element of one *type* into a another *type*. This would result in a change in the size of the input list for the subsequent phase and the analyser must account for this change. The specification of F_s is a means of providing the analyser with the information necessary to account for such changes in the size of the input list between subsequent phases.

Three types of transformation that can be applied on input lists by sequential functions are of interest to the cost analysis. Not all of them can be accounted for by the analyser in its present form.

1. The output list could be transformed to just the extent that the size of the base element is different from that in the input list. e.g. a list of integers into a list of reals. In this case, the sequential function does not alter the D or I_s specifications and these can be used as such to estimate the costs for the subsequent phase. The only change is in the specification of F_t and this is determined simply by the specification of F_s for the sequential function. If $F_{s'}$ specifies the output size for the sequential function in the phase, then $F_t = F_{s'}$ for the list input to the next phase.

2. The sequential function may transform the list in a manner that alters its D-value, e.g. a list of lists of integers could be transformed into a list of integers. In that case, $D = 2$ for the input list to the sequential function and $D = 1$ for the output list. An example of such a function is *update*, as defined in the Jacobi Iteration example in Section 4.3.3. For the subsequent phase, the analyser must account for the change in the D-value of the input list, if it is to make realistic cost predictions. Such transformations can be deduced by a compiler, but the analyser does not incorporate such facilities at present, and is left for future work.

3. The sequential function could transform the input list so as to alter its I_s specification, e.g. a list of lists of size $(m \times n)$ could be altered to a list of lists of size $(m \times k)$. An example of such a function is the *rearrange* function as defined in Section 4.3.3. It transforms an input list of size $(3 \times n)$ into one of size $(n \times 5)$. Such transformations cannot, in general, be deduced at compile-time. In such cases, an inaccurate size of the input list will be deduced for subsequent phases. This in turn might lead to poor cost predictions and in the worst case may lead to the selection of an inefficient parallel implementation for the program. This is a disadvantage of the scheme, at present. Again, a profiler could rectify the situation to some extent.

Sequential functions could cause transformations on input lists, so that some combinations of 1, 2, 3 are produced. The arguments remain the same.

5.2 The Compile-time Analysis

In the HOPP model, an application program is a **composition** of phases. The **composition** itself is performed sequentially, i.e. phase i does not commence until phase $(i-1)$ is completed. This need not be the case and future work could consider pipelining the phases. The results from phase $(i-1)$ can then be input to phase i as they become available. In the absence of pipelining, the total cost of the program comprising of k phases is given by:

$$C_p = \sum_{i=1}^{k} C_{p_i} + \sum_{i=0}^{k-1} C_{i,i+1}$$

where, C_{p_i} represents the cost of phase i, and $C_{i,i+1}$ represents the cost of phase transition between phases i and $i+1$, respectively. The cost of a phase depends on the nature and cost of its composing functions. The cost of phase transition represents the communication cost incurred in rearranging the output data of one phase to be suitable for the subsequent one. Costs for communication routines such as *scatter*, *gather*, *total exchange* and *broadcast*, on the different processor topologies have been derived (see Chapter 6 for details). The rearrangement of data between consecutive phases would typically involve one or more of these communication operations. The analyser uses the costs of these routines in computing the costs of phase transition. If no data movement is required between two consecutive phases, then the cost of phase transition for those two phases is zero.

The following points should be noted.

1. Each recognised function operates on a list data structure.

2. Every recognised function has an associated parallel implementation on each target machine topology included in the model.

3. The cost of implementing a recognised function, F, in parallel on p processors, on an input list of size n is represented by:

$$C = F(n, p, C_a)$$

where, C_a, is the cost of the argument function (if any). In general, if the recognised function, F, operates on k input lists of sizes $n_1, n_2, \ldots, n_k$ respectively, then the cost is represented by:

$$C = F((n_1, n_2, \ldots, n_k), p, C_a).$$

The analyser performs a cost analysis for each branch in the program tree. A branch in a program tree might comprise of instances of both recognised and sequential functions. If a branch (phase) comprises entirely of sequential functions, then the analyser would be forced to select a sequential implementation for it.

Let p be the number of processors and C_s be the cost of the sequential argument function (if any). In the following sections, the representation of cost expressions assume that the recognised functions operate on a single input list.

- The branch could contain only one occurrence of a recognised function, F, in which case only one parallel implementation is possible for a particular topology. If the input list is assumed to be of size m, then the cost of the branch is given by:

$$C_1 = F(m, p, C_s). \tag{5.2}$$

- The branch could contain a recognised function, F, that has another recognised function, G, as its argument. The input data structure would be a list of lists ($D \geq 2$). If the data structure is assumed to comprise of m lists, each of which contains n elements, then three different parallel implementations are considered.

1. One in which the function F is implemented in parallel on p processors, and the function G sequentially. The cost in this case is given by:

$$C_2^1 = F(m, p, G(n, 1, C_s)). \tag{5.3}$$

2. One in which the function F is implemented sequentially, and the function G in parallel on the p processors. The cost in this case is given by:

$$C_2^2 = F(m, 1, G(n, p, C_s)). \tag{5.4}$$

3. The case where both the functions, F and G, are implemented in parallel. The p-processor machine is divided into p_1 parts, each of which comprises of p_2 processors, where, $p_1 \times p_2 \leq p$. The function F is implemented in parallel on p_1 processors, and the function G is implemented in parallel on p_2 processors. Each of these parts retains the same logical topology as the original machine. However, on certain topologies, logical neighbours may no longer be physical neighbours after such a division. In such a case, it would take more than one hop to communicate between logically neighbouring processors, thereby increasing the communication cost. In order that these increased communication costs are accounted for, the cost function is parametrised on the maximum number of hops to be traversed between logical neighbours, and is represented by l. The value of l would depend on the nature of the topology. The cost in this case is given by:

$$C_2^3 = F(m, (p_1, l_1), G(n, (p_2, l_2), C_s)). \tag{5.5}$$

If $l_1 = l_2 = 1$, Equation 5.5 can be reduced to:

$$C_2^3 = F(m, p_1, G(n, p_2, C_s)). \tag{5.6}$$

As an illustration of such an implementation, a p-processor hypercube of dimension, $d = \log_2 p$, is divided into 2^k $(0 < k < d)$ smaller hypercubes (subcubes). Each subcube comprises of 2^{d-k} processors. The m lists are scattered across the 2^k subcubes, $\lceil \frac{m}{2^k} \rceil$ per subcube. The $\lceil \frac{m}{2^k} \rceil$ elements all go to a single processor in each of the subcubes, namely, to processors numbered by

$$0, 2^{d-k}, 2.2^{d-k}, 3.2^{d-k}, \ldots, (2^k - 1).2^{d-k}.$$

This implies a parallel implementation for F on 2^k processors, with $\lceil \frac{m}{2^k} \rceil$ elements per processor. Then the function G can be evaluated in parallel on 2^{d-k} processors, with $\lceil \frac{n}{2^{d-k}} \rceil$ elements per processor. Since all the subcubes are connected, and each subcube retains the same connectivity as the original hypercube, $l_1 = l_2 = 1$. The cost in this case is given by:

$$C_2^h = F(m, 2^k, G(n, 2^{d-k}, C_s)). \qquad (5.7)$$

In all the three cases, the cost of an inner-level function is passed as a parameter to the outer-level function. At the innermost level this might correspond to a sequential function for which no known parallel solution exists. Information regarding the number of processors available and the input data sizes, is passed down from the outer function to the inner function. This information would depend on the implementation that is selected. The inner function then evaluates its own cost, based on the information it receives. Information about the cost of the inner function is then passed back up to the outer function, which can then evaluate its cost. This argument could be applied to any number of levels in a branch in the program tree.

- The branch could contain three or more occurrences of recognised functions. First, the case when the branch contains exactly three recognised functions, F, G and H, is considered. G is the argument of F, and H is the argument of G. The input data structure is a list of list of lists ($D \geq 3$). Seven possibilities arise, corresponding to the implementations of each of the three functions in parallel, any two in parallel and all three in parallel. For the first six cases, arguments similar to the previous case can be applied and the costs can be similarly derived. If the input data structure is assumed to comprise of m lists, each of which comprises of n lists, each of which in turn contains k elements, then the cost of the branch corresponding to implementing all three functions in parallel is given by:

  $$C_3 = F(m, (p_1, l_1), G(n, (p_2, l_2), H(k, (p_3, l_3), Cs))) \tag{5.8}$$

  where, $p_1 \times p_2 \times p_3 \leq p$, and l_1, l_2, l_3 are defined as before.

  Theoretically, the occurrence of more than three recognised functions on a branch could be considered. This would mean a data structure that is a list of lists of lists ($D \geq 4$). If such a case does arise, the HOPP model exploits potential parallelism only up to the first three levels and the discussion in the previous paragraph applies.

It is clear that for every phase in the program, there may be several possible parallel implementations, depending on the number of recognised functions in that phase. Under the current scheme, the maximum number of implementations that can be considered for a phase is *eight* (including a sequential implementation). If all the possibilities are considered at every phase, the analysis results in the construction of a search tree. The weights on the nodes of the search tree at each level represent the costs of the different possible implementations for each phase. This cost includes computation as well as any communication costs that may be incurred in the implementation of the recognised function(s) in that phase. The weights on the edges of the search tree represent the costs of phase transition and comprise only of communication costs. A traversal of each path in the tree yields a cost of implementation for the whole program. Each such cost represents the cost of implementing the program when some particular sequence of implementations is chosen for the phases in the program. Each path in the search tree represents a unique and complete implementation for the entire program. The least-cost path in the search tree corresponds to the most cost-effective implementation for the program. The analyser first performs a cost analysis on the program and constructs the search tree at compile-time. The search tree also contains information about the functions in each phase, the number of processors to be used for the implementation of each recognised function in the phase, the nature of the data distribution at the end of each phase, and so on. Using the information on the branches in the path that is selected, code for the implementation can be generated and a copy distributed to all the processors. This would contain code for the recognised and sequential functions in each branch as well as for performing any communications required in the implementation of the program.

It is clear that the size of the search tree grows exponentially with the number of phases in the program. It is difficult to provide a threshold value for the number of phases in the program, after which the search space could be termed as *unacceptably large*. However, even for the Jacobi Iteration example comprising of seven phases (see Section 4.3.3), the search space was quite large and it would have been helpful to reduce it. Heuristics to prune the exponential growth of the search tree have to be investigated. Rather than considering all possible implementations, those that are likely to result in high execution costs could be identified and discarded at the analysis stage itself.

It is important to note that low-level costs such as memory access times are not accounted for. This would make the model very machine-specific and is not desirable. List processing costs, which include the costs incurred in constructing or traversing a list, are however, estimated by the analyser since these costs can prove to be significant. The analyser estimates these costs based on the nature of the input list (obtained from I_s and F_t) and the details are transparent to the programmer. During the analysis of the program, the analyser actually performs list construction or traversal operations on a dummy list that is of the same type as the specified input list and estimates the cost, which is then used as the list processing cost. Since all the low-level costs associated with a practical implementation are not accounted for by the analyser, the actual cost of implementation is expected to be greater than that predicted. However, computing the *absolute* cost is not necessary, since the strategy is to select an implementation based on cost *comparison*. There are four different costs involved.

- The theoretically predicted cost of sequential execution for the program, represented by T^s_{theor}.
- The theoretically predicted cost of the selected parallel implementation on p processors, represented by T^p_{theor}.
- The actual cost of sequential execution for the program when executed on a single processor on the parallel machine, represented by T^s_{prac}.
- The actual cost of the selected implementation for the program when executed on p processors on the parallel machine, represented by T^p_{prac}.

If the model reflects the practical performance, then

$$\frac{T^p_{theor}}{T^s_{theor}} \approx \frac{T^p_{prac}}{T^s_{prac}}. \tag{5.9}$$

5.2.1 The Algorithm for the Analyser

Algorithm **ANALYSE** presents a high-level description of the analyser. The function GET_SPEC obtains the problem specification as discussed in Section 5.1. The function, CONSTRUCT_PROGRAM_TREE, constructs the program tree from the specification of the program code. If a branch in the program tree contains only one recognised function, then in the following algorithm the input list is represented by n_1 and the cost of the recognised function is represented by R_1. If the branch contains two recognised functions, then the input list is represented by (n_1, n_2) and the costs of recognised functions are represented by R_1 and R_2, respectively. For a branch containing three recognised functions, the input list is represented by (n_1, n_2, n_3) and the costs of the recognised functions are represented by R_1, R_2 and R_3, respectively. C_s represents the cost of a sequential argument function. Phase$_i$ of the program is assumed to be implemented on p_i processors; $p_0 = 1$, implying that the data is initially resident on a single node of the parallel machine. N_i represents the number of recognised functions in a phase, $0 \leq N_i \leq 3$. The following rule is used in the derivation of cost expressions:

$$\text{If } k = N_i + 1, \text{ then } R_k(n_k, p, C_a) \equiv C_s^k$$

It means that R_k is not a recognised function and its cost must, therefore, be replaced by the cost of the sequential function at the k^{th} level.

The function ADD_TO_S_TREE inserts a child node to a specified node in the search tree. In the implementation, the weight on the inserted branch is added to the weight on the destination node. In other words, the inter-phase transition cost is added to the cost of implementation of the subsequent phase. C^i_{seq} represents the cost of implementing phase$_i$ sequentially. $C^i_{par\_x}$ represents the cost of implementing function R_x in phase i in parallel $(1 \leq x \leq 3)$, $C^i_{par\_xy}$ represents the cost of implementing both the functions R_x and R_y in phase i in parallel on p_i^x and p_i^y processors respectively $(1 \leq x \leq 2, 2 \leq y \leq 3)$, $(p_i^x p_i^y \leq p_i)$ and $C^i_{par\_123}$ represents the cost of implementing functions R_1, R_2 and R_3 in phase i in parallel on p_i^1, p_i^2 and p_i^3 processors respectively $(p_i^1 p_i^2 p_i^3 \leq p)$.

The function REARR computes the cost for rearranging the output data of one phase to suit the implementation of the subsequent phase. The information about the current data distribution is obtained from the current parent node in the search tree. If the output list of size n_1 from phase $i-1$ is distributed across p_{i-1} processors, and phase i requires the data list to be distributed across p_i processors, $(p_{i-1}, p_i \leq p)$, then REARR (n_1, p_{i-1}, p_i) computes the required data rearrangement cost. Similarly, for the list (n_1, n_2) to be distributed such that the sublists are distributed across p_i^1 processors and each sublist across p_i^2 processors $(p_i^1 p_i^2 \leq p)$, REARR $((n_1, n_2), p_{i-1}, (p_i^1, p_i^2))$ computes the required data rearrangement cost.

S_tree represents the search tree, and *S_lev* represents the current level in the search tree. Initially, *S_tree* simply comprises of a root node which is at level zero and has zero cost. The analysis of each branch in the program tree could insert up to eight child nodes (corresponding to the maximum number of implementations considered by the analyser), to each node of the search tree at the current level. In the algorithm, the inter-phase transition cost is added to that of each such implementation in order to compute its total cost.

The function ADD_TO_S_TREE inserts a child node to a specified node in the search tree. In the implementation, the weight on the inserted branch is added to the weight on the destination node. In other words, the inter-phase transition cost is added to the cost of implementation of the subsequent phase. C^i_{seq} represents the cost of implementing phase$_i$ sequentially. $C^i_{par\_x}$ represents the cost of implementing function R_x in phase i in parallel $(1 \leq x \leq 3)$, $C^i_{par\_xy}$ represents the cost of implementing both the functions R_x and R_y in phase i in parallel on p_i^x and p_i^y processors respectively $(1 \leq x \leq 2, 2 \leq y \leq 3)$, $(p_i^x p_i^y \leq p_i)$ and $C^i_{par\_123}$ represents the cost of implementing functions R_1, R_2 and R_3 in phase i in parallel on p_i^1, p_i^2 and p_i^3 processors respectively $(p_i^1 p_i^2 p_i^3 \leq p)$.

The function REARR computes the cost for rearranging the output data of one phase to suit the implementation of the subsequent one. The information about the current data distribution is obtained from the current parent node in the search tree. If the output list of size n_1 from phase $i-1$ is distributed across p_{i-1} processors, and phase i requires the data list to be distributed across p_i processors, $(p_{i-1}, p_i \leq p)$, then REARR (n_1, p_{i-1}, p_i) computes the required data rearrangement cost. Similarly, for the list (n_1, n_2) to be distributed such that the sublists are distributed across p_i^1 processors and each sublist across p_i^2 processors $(p_i^1 p_i^2 \leq p)$, REARR $((n_1, n_2), p_{i-1}, (p_i^1, p_i^2))$ computes the required cost.

S_tree represents the search tree, and *S_lev* represents the current level in the search tree. Initially, *S_tree* simply comprises of a root node which is at level zero and has zero cost. The analysis of each branch in the program tree could insert up to eight child nodes (corresponding to the maximum number of implementations considered by the analyser), to each node of the search tree at the current level. In the algorithm, the inter-phase transition cost is added to that of each such implementation in order to compute its total cost.

```
Algorithm ANALYSE
BEGIN
    GET_SPEC            /* obtain problem specification */
    CONSTRUCT_PROGRAM_TREE (P)
    S_tree ⟵ root;    S_lev ⟵ 0        /* initialise search tree */
    FOR $i \longleftarrow 1$ to number-of-branches in P DO
        $N_i \longleftarrow$ number of recognised functions in branch$_i$     /* $\leq 3$ */
        $j \longleftarrow 1$
        FOR each node$_j$ in level S_lev of S_tree DO
            Compute $C^i_{seq}$       /* sequential implementation */
            ADD_TO_S_TREE (S_tree, S_lev, $j$, $C^i_{seq}$)
            FOR $x \longleftarrow 1$ to $N_i$ DO
                FOR $k \longleftarrow 1$ to $N_i$ DO
                    IF $(k = x)$ THEN $p_k \longleftarrow p_i$ ELSE $p_k \longleftarrow 1$
                $C^i_{par\_x} \longleftarrow$ REARR $(n_x, p_{i-1}, p_i)$ +
                                $R_1(n_1, p_1, R_2(n_2, p_2, R_3(n_3, p_3, C_s)))$
                ADD_TO_S_TREE (S_tree, S_lev, $j$, $C^i_{par\_x}$)
            FOR $x \longleftarrow 1$ to $N_i$ - 1 DO
                FOR $y \longleftarrow 2$ to $N_i$ DO
                    IF $(x \neq y)$ THEN
                        FOR $k \longleftarrow 1$ to $N_i$ DO
                            IF $(k = x)$ THEN $p^k_i \longleftarrow p^x_i$
                            ELSE IF $(k = y)$ THEN $p^k_i \longleftarrow p^y_i$ ELSE $p^k_i \longleftarrow 1$
                        $C^i_{par\_xy} \longleftarrow$ REARR $((n_x, n_y), p_{i-1}, (p^x_i, p^y_i))$ +
                                $R_1(n_1, (p^1_i, l_1), R_2(n_2, (p^2_i, l_2), R_3(n_3, (p^3_i, l_3), C_s)))$
                        ADD_TO_S_TREE (S_tree, S_lev, $j$, $C^i_{par\_xy}$)
            IF $(N_i = 3)$ THEN
                $C^i_{par\_123} \longleftarrow$ REARR $((n_1, n_2, n_3), p_{i-1}, (p^1_i, p^2_i, p^3_i))$ +
                                $R_1(n_1, (p^1_i, l_1), R_2(n_2, (p^2_i, l_2), R_3(n_3, (p^3_i, l_3), C_s)))$
                ADD_TO_S_TREE (S_tree, S_lev, $j$, $C^i_{par\_123}$)
        S_lev ⟵ S_lev + 1;
END.
```

The algorithm in the actual implementation also checks that the number of processors available is sufficient to implement more than one recognised function in parallel, before further analysis is carried out. If there are insufficient processors for any implementation, then the corresponding branches will not be added to the search tree. Also, if two recognised functions are to be implemented in parallel, then the most cost-effective partition of the processor network is first determined and the costs corresponding to this partition are inserted in the search tree. For the sake of brevity, these and a few other low-level details of the analysis are not shown in the algorithm.

5.3 An Example of Compile-time Analysis

To provide a flavour for the manner in which the analyser handles the input specifications, a simple example is considered.

fun ex xs = (**s_fold** mult ∘ **map** (**s_fold** plus)) xs;

The recognised functions are represented in boldface. The functions *mult* and *plus* are defined to be functions that multiply and add two integers respectively.

The program comprises of two phases. The first phase contains two recognised functions, **map** and **s_fold**, and the second phase contains one recognised function, **s_fold**. The analyser first requires the problem specification, as discussed in Section 5.1. In this simple example, the input sizes are assumed to be known at compile-time.

The program tree, P, representing the program is depicted by Figure 5.2.

The parallel machine is assumed to comprise of p processors connected as a hypercube. The dimension of the hypercube is represented by d. K_0 and K_1 represent the communication start-up cost and communication channel bandwidth on the hypercube, respectively. The specification for M for both the phases is:

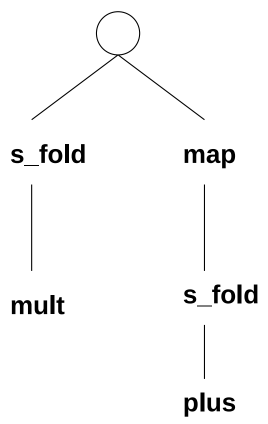


Figure 5.2: Program Tree for the Example

$M \equiv (p,$ "hypercube", K_0, $K_1)$.

Since the input data sizes are assumed to be known at the start of the program and the analyser deduces it for the second phase, $S \equiv \emptyset$, for both the phases.

First, the specifications for *phase one* are considered.

- $C_f \equiv$ Cost of *plus*, represented by C_p.

- $D \equiv 2$, since the input data structure is a list of lists, in which each sublist is a list of integers. (This could be deduced from the type of the function *plus*).

- I_{sa} is specified since it is assumed that the input data sizes are known at compile-time. Let the size of the input list be represented by $(m \times n)$ - i.e. the list comprises of m sublists, each of which contains n integers.

- $F_t \equiv$ (size of integer).

- $F_s \equiv$ (size of integer), since the function *plus* adds two integers and outputs an integer.

In this example, D, F_t and F_s could be deduced from the *type* of the function *plus*, if the analyser incorporated a type-checker. C_f and I_{sa} could be estimated if the analyser incorporated a profiler.

The input size specifications are made available (either by the programmer or by profiling techniques) at the beginning of the first phase. The analyser computes the data sizes for the subsequent phases from the nature of the recognised functions which are currently being analysed. In the example program, the input to the first phase is a list of lists of size $(m \times n)$. The **s_fold** in *phase one* reduces this to a list of size m, in which each element is an integer. The analyser deduces the following attributes for *phase two*.

- $D \equiv 1$, since a list of lists of integers has been reduced to a list of integers.
- $I_{sa} \equiv m$.

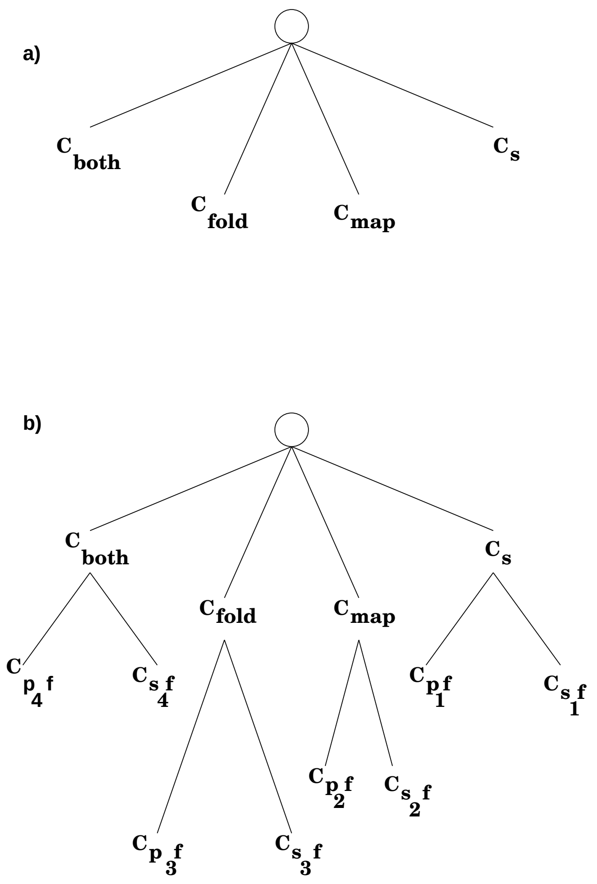


Figure 5.3: Search Tree for the Example

A type-checker and a profiler (or the programmer) provide the following information:

- $C_f \equiv$ Cost of *mult*, represented by C_m.
- $F_t \equiv$ (size of integer). This again, could be deduced from the *type* of the function *mult*.
- $F_s \equiv$ (size of integer), also deducible from the *type* of the function *mult*.

Both functions can be implemented in parallel only if the dimension, d, of the hypercube ≥ 2. Figure 5.3a represents the search tree after *phase one*. (Since **composition** works right to left, the rightmost branch on the program tree is executed first). C_s represents the cost of sequential implementation for the phase, C_{map} represents the cost of implementing **map** in parallel and **s_fold** sequentially, C_{fold} represents the cost of implementing **s_fold** in parallel and **map** sequentially and C_{both} represents the cost of implementing both **map** and **s_fold** in parallel.

Figure 5.3b represents the search tree after *phase two*. In *phase two*, the analyser computes the costs for two possible implementations for **s_fold** - the sequential and parallel implementation, respectively, and the corresponding costs are represented by $C_{s_i f}$ and $C_{p_i f}$, $1 \leq i \leq 4$. For each branch in *level two* of the search tree, the costs of implementation include the costs incurred in re-distributing the data from *phase one* to *phase two*. For some branches, (e.g. $C_{p_2 f}$), no data re-distribution is required.

Chapter 6

Parallel Implementations and Costs for Recognised Functions

A program in the HOPP model is written independently of the architecture on which it is executed, the implication being that only *one* program is written. However, the nature of the parallel implementation which is selected does depend, among other parameters, on the characteristics of the target architecture as discussed in Section 5.1.2. The hypercube, 2-D torus, s-ary tree and linear array have been studied as potential target topologies. The hypercube proved to be the most suitable topology for the set of recognised functions and the tree and the linear array proved to be inefficient. In this chapter, implementations for the recognised functions on the previously mentioned topologies are discussed and their costs are derived.

6.1 Data Communications on the Topologies

Communication costs constitute a significant part of the total cost of execution of parallel programs on distributed memory machines. For the purposes of the implementation scheme, five types of communication patterns have been identified. The communication algorithms for the hypercube topology are based on [SS89]. Some of the mesh algorithms are based on [YM89]. However, the algorithms in

[SS89] and [YM89] assume that processors can communicate with more than one neighbour simultaneously. As mentioned in Section 5.1.2, the HOPP model does not assume such a capability. The algorithms and the derivation of the associated costs reflect this assumption.

The following patterns of communications recur in programs.

1. Nearest Neighbour Communication

   This type of communication involves sending a message (or a data packet) to a neighbouring processor. Two processors are considered to be neighbours if they are connected by a direct communication link.

2. Broadcast Operation

   This operation involves moving the same data packet from one processor to all the other processors in the network.

3. Scatter Operation

   This operation involves moving different data packets (one to each processor), from one processor to all the other processors in the network.

4. Gather Operation

   This operation involves collecting the data packets distributed across the processors in the network, onto a single processor.

5. Total Exchange Operation

   This operation involves moving data packets from every processor in the network to every other processor.

As pointed out in [SS89], the gather operation is the dual of the scatter operation, i.e. the algorithm for the gather operation can be obtained by reversing the data paths in the scatter algorithm and vice-versa. Hence, the costs for both the operations are identical. In the following discussion, only the scatter operation is considered.

The total number of processors in each case is assumed to be p.

- Hypercube - The number of processors in a binary hypercube of dimension d is given by: $p = 2^d$

- Torus - The number of processors in a 2-dimensional torus, where,

  p_1 represents the number of processors in each row
  p_2 represents the number of processors in each column
  is given by: $p = p_1 p_2$, $p_1 = 2^{k_1}$, $p_2 = 2^{k_2}$, $p_1 \leq p_2$.

  The number of processors in each dimension is made a power of two in order to allow a direct comparison of the performance of the torus with that of the hypercube. The cost expressions would remain unchanged even with this restriction removed.

- Tree - The number of processors in a tree of arity s and depth d is given by: $p = \frac{s^{d+1}-1}{s-1}$

Figure 6.1 illustrates the processor-numbering schemes for examples of the four topologies.

The algorithms for performing communications on the hypercube topology are based on the corresponding ones in [SS89], are therefore not described here, with only the cost expressions being presented. The size of the data to be communicated is assumed to be n bytes and all the data is assumed to be initially resident on processor$_0$. This does not matter in the case of the hypercube; the algorithms would remain unchanged irrespective of the processor from which the data is broadcast, scattered or gathered. It must also be emphasised that the

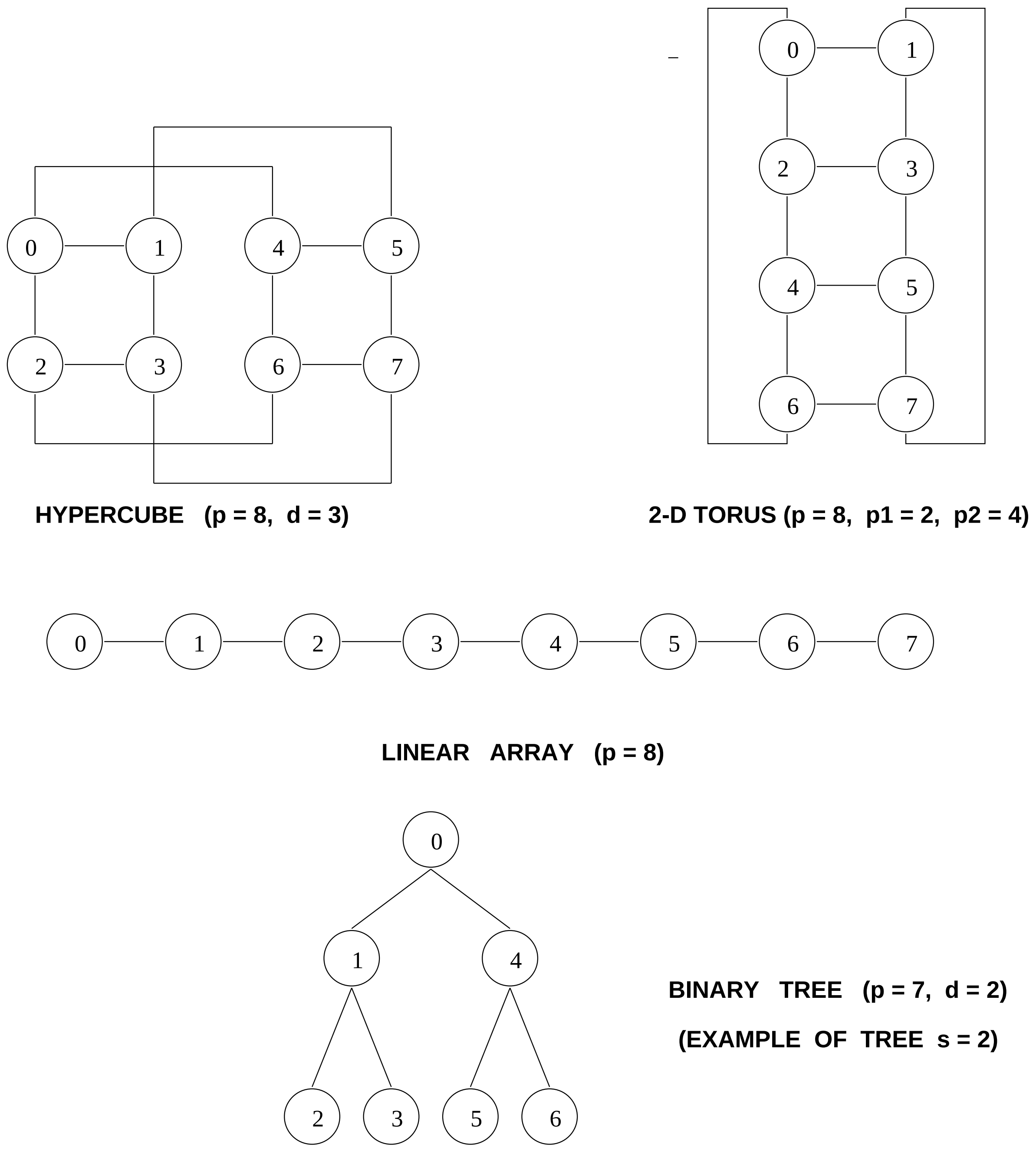


Figure 6.1: Processor-numbering Schemes for the Four Topologies

algorithms are not necessarily *optimal*, since the derivation of optimal communication algorithms is not central to this thesis. However, this will not affect the results of the thesis; a more cost-effective data distribution algorithm would probably result in a lower cost of implementation and could be incorporated in the scheme even at a later stage.

In the ensuing discussion, it is assumed that n, the number of elements in the input list, is divisible by the number of processors, p. If not, the cost computations would have to consider the expression $\lceil \frac{n}{p} \rceil$ and this would make it difficult to simplify cost expressions. If n is not divisible by p, then without loss of generality, it can be replaced by n', where $n' = (n+k)$, $0 < k < p$ and $(n+k) \bmod p = 0$. The costs are computed with the number of elements equal to n'. In all the cost expressions, it is assumed that the necessary substitution has already been made.

6.1.1 Nearest-neighbour Communication

The cost expression for this operation is identical on all the topologies. The operation itself is very simple - processor A sends a data packet to a neighbouring processor B which receives it. Using the linear model of communication, the cost for communicating a data packet of size n bytes to a nearest neighbour is given by:

$$C_{nn} = K_0 + \frac{1}{K_1} n \tag{6.1}$$

6.1.2 The Broadcast Operation

1. Hypercube - The cost of broadcasting a data packet of size n bytes on a hypercube of dimension d is given by:

$$C^h_{broad} = d(K_0 + \frac{1}{K_1} n). \tag{6.2}$$

2. Linear Array - The cost of communication on the linear array is more expensive because of its poor connectivity. In order to exploit parallelism in

the links, a data packet of size n bytes is split into r equal-sized packets and the communication of these packets are overlapped on different links. The overlap is possible only when $p > 3$, otherwise, the case is trivial and the data reaches the last processor in the linear array in two steps. For $(p > 3)$, the first data packet reaches the last processor in $(p-1)$ steps. Thereafter, since a *send* and a *receive* cannot be performed simultaneously in the assumed model, the last processor receives a data packet every alternate step. Since there are $(r-1)$ packets still to be received, a total of $2(r-1)$ steps will be required for the last processor to receive all the data. The other processors would have received all the data by this time. Since the amount of data transferred in each step is $\frac{n}{r}$, the cost of communication is given by:

$$C^a_{broad} = ((p-1) + 2(r-1))(K_0 + \frac{1}{K_1}\frac{n}{r}). \tag{6.3}$$

The expression simplifies to:

$$C^a_{broad} = (p + 2r - 3)(K_0 + \frac{1}{K_1}\frac{n}{r}). \tag{6.4}$$

The optimal value of r is given by: $r = \lceil\sqrt{\frac{(p-3)n}{2K_0K_1}}\rceil$.

3. 2-D Torus/Mesh - The data is first pipelined vertically and in the second stage of the broadcast, each row of processors pipelines the data horizontally in parallel. The wrap-around connections can be used to reduce the cost of broadcast in a 2-D torus. Instead of pipelining the data down the first column, processor$_0$, in the first step sends the data to the last processor in the column, using the wrap-around link (See Figure 6.1). In the second step, both the processors pipeline the data in opposite directions until all the processors in the first column receive the data. This reduces the length of the pipeline by half. A similar strategy is repeated along each of the rows in parallel in the next step. The total cost is then obtained by adding the

costs for each of the three stages.

$$\begin{aligned} C_{broad}^{m\_ideal} &= (K_0 + \frac{1}{K_1}n) + (\frac{p_2}{2} - 1)(K_0 + \frac{1}{K_1}n) + (K_0 + \frac{1}{K_1}n) + \\ &\quad (\frac{p_1}{2} - 1)(K_0 + \frac{1}{K_1}n) \end{aligned}$$

The total cost of broadcast in a mesh is given by:

$$C_{broad}^{m\_ideal} = \frac{p_1 + p_2}{2}(K_0 + \frac{1}{K_1}n). \qquad (6.5)$$

There is a slight non-uniformity in the 2-D torus considered for implementation. The example programs are to be implemented on a transputer-based machine. The transputers are configured as a 2-D torus for the purposes of implementation. Each transputer has only four links and each processor in a 2-D torus has at most four neighbours, so the logical neighbours on the torus can be made physical neighbours on the transputer network. However, an extra link on one processor is required to communicate with the operating system. This requirement causes a slight non-uniformity in the configured torus architecture and, as a result, processor$_0$ is physically connected to only three neighbours. This increases the cost of the broadcast operation since the data communication to one of processor$_0$'s neighbours (processor$_{p_1-1}$, in this case), cannot be performed in one step. This implies that the wrap-around connection cannot be used in the first row, to reduce the length of the pipeline by half. The cost expression is therefore modified to the following.

$$C_{broad}^{m} = (p_1 + \frac{p_2}{2} - 1)(K_0 + \frac{1}{K_1}n) \qquad (6.6)$$

4. s-ary Tree - The broadcast operation in the case of the tree is straightforward. The data to be broadcast is initially at the root of the tree. The root sends the data to one child at a time, starting from the left to the right. Each child node receives the data from its parent and communicates

it to its children, starting with the leftmost child. The communications performed by the nodes in each level are overlapped, resulting in s such communications. For a tree of depth d, the cost is given by:

$$C^t_{broad} = sd(K_0 + \frac{1}{K_1}n). \tag{6.7}$$

6.1.3 The Scatter Operation

The data is divided into p equal parts which are numbered 0,1,2,..., $(p-1)$. After the scatter operation, each processor has $\frac{n}{p}$ bytes of the data. It is important that each processor receives the correct portion of the data. In particular, $processor_i$ should receive the i-th part of the data. (For processor numbering patterns, see Figure 6.1). This is because the algorithms for the parallel implementations of the recognised functions are based on a particular node numbering and assume the correct placement of data. Since the arguments of none of the recognised functions are assumed to be commutative, incorrect results may be obtained if the data is not scattered in the required order.

1. Hypercube - The cost of scattering data on a hypercube of dimension d is given by:
$$C^h_{scatter} = dK_0 + \frac{1}{K_1}\frac{n}{p}(p-1). \tag{6.8}$$

2. Linear Array - The data is just sent down the array and each processor retains the first $\frac{n}{p}$ bytes of the data, sending the rest to its neighbour. There is no parallelism exploited, but it is unlikely to improve the performance if communication is overlapped on the different links. The communication involves $(p-1)$ steps, and in each step, the amount of data reduces by $\frac{n}{p}$ bytes. The total cost is therefore given by:
$$C^a_{scatter} = \sum_{i=1}^{p-1}(K_0 + \frac{1}{K_1}(n - i\frac{n}{p})).$$

The cost expression simplifies to:

$$C^{a}_{scatter} = (p-1)(K_0 + \frac{1}{K_1}\frac{n}{2}). \tag{6.9}$$

3. 2-D Torus/Mesh - In the first step, the data is divided into two equal (or nearly equal) parts, with one half being sent from processor$_0$ to the last processor in the first column, using the wrap-around link. Each processor has $\frac{n}{2}$ bytes of the data after the first step. Each retains the appropriate amount of data ($\frac{n}{p_1}$ bytes), and in the second step, both the processors pipeline the data in opposite directions. At the end of step two, each processor in the first column has the data necessary ($\frac{n}{p_1}$ bytes) for its respective row. In the third step, all the processors in the first column repeat a similar procedure by sending data along the rows in parallel. The amount of data transferred along the links decreases with each transfer, by $(\frac{n}{p_1})$ bytes in step two, and by $(\frac{n}{p})$ bytes in step three. At the end of step three, each processor has the required portion of the data. Using the results for a linear array for the communications in steps two and three, the cost is given by:

$$\begin{aligned} C^{m\_ideal}_{scatter} &= (K_0 + \frac{1}{K_1}\frac{n}{2}) + (\frac{p_2}{2} - 1)(K_0 + \frac{1}{K_1}\frac{n}{4}) + \\ &\quad (K_0 + \frac{1}{K_1}\frac{n}{2p_2}) + (\frac{p_1}{2} - 1)(K_0 + \frac{1}{K_1}\frac{n}{4p_2}). \end{aligned}$$

On simplification, the cost is given by:

$$C^{m\_ideal}_{scatter} = K_0(\frac{p_1+p_2}{2}) + \frac{1}{K_1}\frac{n}{4}((\frac{p_2}{2}+1) + \frac{1}{p_2}(\frac{p_1}{2}+1)). \tag{6.10}$$

If the non-uniformity in the wrap-around connections is accounted for, then in step three, the entire length of the pipeline must be considered. So, the cost expression modifies to the following.

$$\begin{aligned} C^{m}_{scatter} &= (K_0 + \frac{1}{K_1}\frac{n}{2}) + (\frac{p_2}{2} - 1)(K_0 + \frac{1}{K_1}\frac{n}{4}) + \\ &\quad (p_1 - 1)(K_0 + \frac{1}{K_1}\frac{n}{2p_2}) \end{aligned}$$

The final cost expression for the non-uniform mesh simplifies to:

$$C^m_{scatter} = K_0(p_1 + \frac{p_2}{2} - 1) + \frac{1}{K_1}(\frac{n}{4}(\frac{p_2}{2} + 1) + \frac{n}{2p_2}(p_1 - 1)). \qquad (6.11)$$

4. Tree - A parent node communicates data to each of its s children in turn, starting with the leftmost child. Each child node retains the first $\frac{n}{p}$ bytes of the data, being the data that is meant to reach that processor. The rest of the data is divided into s parts and communicated to its children. This scheme ensures that the data reaches the correct processor, since the trees are numbered by pre-order traversal (refer to Figure 6.1).

   At level j in the tree, each processor has to communicate the data it received from its parent, to k processors below it, where

$$k = s\frac{s^{d-j} - 1}{s - 1}. \qquad (6.12)$$

   Each processor must possess $\frac{n}{p}$ bytes of the data after the scatter operation. So the amount of data still to be communicated at level j in the tree is $k\frac{n}{p}$. This data is divided into s parts and sent to each of the children. Each such communication involves the transfer of $\frac{k}{s}\frac{n}{p}$ bytes of data. Since there are s such communications in each level and there are d levels in the tree, the total cost is given by:

$$C^t_{scatter} = \sum_{j=0}^{d-1} s(K_0 + \frac{1}{K_1}(\frac{k}{s}\frac{n}{p})). \qquad (6.13)$$

   Substituting for k from Equation 6.12 and simplifying the summation yields:

$$C^t_{scatter} = sdK_0 + \frac{1}{K_1}\frac{n}{p}\frac{s}{s-1}(p - 1 - d). \qquad (6.14)$$

6.1.4 Total-Exchange

It is assumed that all the processors exchange equal amounts of data, i.e. $\frac{n}{p}$ bytes.

1. Hypercube - The cost of the total-exchange operation on a hypercube of dimension d is given by:

$$C^h_{exchange} = 2dK_0 + 2\frac{1}{K_1}\frac{n}{p}(p-1). \tag{6.15}$$

2. Linear Array - The total exchange operation on the linear array is very expensive due to its poor connectivity. The procedure involves dividing the linear array into two sub-arrays. In the first step, the processors in the right sub-array move their data to the left, and the processors in the left sub-array move their data to the right, in parallel. Hence, data from the two sub-arrays is gathered in the first processor of the right sub-array and the last processor of the left sub-array, respectively. In the second step, these two processors exchange the data. In the third step, the data is passed back in opposite directions along both the sub-arrays.

   The first step is just a gather operation involving half the number of processors in the original linear array and half the original data. The second step will therefore involve an exchange of $\frac{n}{2}$ bytes of data. In the third step, each processor also appends its own data to the data it receives and sends it on to its neighbour. With each communication, the amount of data which is transferred progressively increases by an amount equal to $\frac{n}{p}$ bytes. The total cost is obtained by adding the costs for the three steps. The number of processors, p, is assumed to be even. The cost is given by:

$$\begin{aligned} C^a_{exchange} &= (\frac{p}{2}-1)(K_0 + \frac{1}{K_1}\frac{n}{p}) + 2(K_0 + \frac{1}{K_1}\frac{n}{2}) + \\ &\quad \sum_{i=1}^{\frac{p}{2}-1}(K_0 + \frac{1}{K_1}(\frac{n}{2} + i\frac{n}{p})). \end{aligned}$$

On simplification, the cost expression is given by:

$$C^{a}_{exchange} = p(K_0 + \frac{1}{K_1}\frac{n}{2}). \tag{6.16}$$

3. Mesh - The method adopted is based on the one described in [YM89]. Each column and row of the mesh forms a ring because of wrap-around. In the first step of the total exchange operation, all the nodes exchange data with their column neighbours, by moving the data round the vertical rings. In the second step, a similar procedure is repeated along the rings in the rows. The communication in the columns (rows) are performed in parallel. Since $\frac{n}{p}$ bytes of data are involved in each transfer, the cost is given by:

$$C^{m\_ideal}_{exchange} = (K_0 + \frac{1}{K_1}\frac{n}{p})(p_2 - 1) + (K_0 + \frac{1}{K_1}\frac{n}{p_1})(p_1 - 1). \tag{6.17}$$

which on simplification yields:

$$C^{m\_ideal}_{exchange} = K_0(p_1 + p_2 - 2) + \frac{1}{K_1}n(1 - \frac{1}{p}). \tag{6.18}$$

However, for the case of the non-uniform mesh, the wrap-around connection in the first row cannot be used. So, two traversals of the first row are required. Since the number of processors along the rows (p_1), may be less than the number of processors along the columns (p_2), it proves to be more cost-effective if the data is exchanged along the rows in the first step, and along the columns, in the second step. The cost is given by:

$$C^{m}_{exchange} = (K_0 + \frac{1}{K_1}\frac{n}{p})(2p_1 - 2) + (K_0 + \frac{1}{K_1}\frac{n}{p_2})(p_2 - 1).$$

On simplification,

$$Cost^{m}_{exchange} = K_0(2p_1 + p_2 - 3) + \frac{1}{K_1}\frac{n}{p}(p_1 + p - 2).$$

Equation 6.16 could be used in the first step to further reduce the cost.

4. Tree - The method adopted involves gathering the results to the root processor and then broadcasting the results to all the processors. Since the cost of the gather operation is equal to the cost of the scatter operation, the cost is given by (refer to Equations 6.14, 6.7):

$$C^t_{exchange} = C^t_{scatter} + C^t_{broad} \tag{6.19}$$

6.2 Algorithms and Costs for the Parallel Implementations of Recognised Functions

Although the functions in the extended set can be expressed in terms of one or more functions in the basic set, their parallel implementation will often be different from that of the constituent functions. This is because the definitions for these functions in terms of the basic set of functions is often quite contrived and the corresponding implementations may not be cost-effective. In computing the costs of the recognised functions, the following assumptions are made.

1. Initial data distribution costs are ignored. These costs, which comprise of the costs for operations such as *scatter* or *broadcast*, are accounted for when the costs are computed for the entire problem.

2. The number of processors in the parallel machine is represented by p.

3. The input list is represented by xs, where $\mid xs \mid = n$

4. The input list, xs, is assumed to be distributed across the p processors in the parallel machine. The following assumption is applicable to all the algorithms, unless otherwise stated.

   $xs = xs_0 @ xs_1 @ \ldots @ xs_{p-1}$

   $xs_i \in \text{processor}_i$

   $\mid xs_i \mid = \frac{n}{p}; \qquad ; i = 0, \ldots, (p-1)$

5. T^m_{com} represents the cost of communicating m elements of the list to a neighbour.

$$T^m_{com} = K_0 + \frac{1}{K_1} ms \tag{6.20}$$

where s is the size (in bytes) of each element of the list.

6. The sequential argument function of a recognised function is represented by f and its cost by C_f.

7. The analyser makes the following assumption for all the sequential functions in the program.

$\forall x_i \in xs; \qquad i \in 0, 1, \ldots, (n-2)$

Cost $(f\ x_i) \approx$ Cost $(f\ x_{i+1})$,

where, f is any sequential function in the program.

This is an important assumption implying the *regularity* of the application that is considered for cost analysis.

For each of the recognised functions, the sequential cost is first computed. The algorithm for the parallel evaluation of the function on each of the four topologies is then presented, along with their associated costs. For the purposes of presentation, the same ordering of recognised functions is followed as in Chapter 4.

6.2.1 map

The sequential cost of **map** is given by:

$$C^s_{map} = nC_f \tag{6.21}$$

The parallel implementation for **map** on all the topologies is straightforward. All the processors perform the **map** operation in parallel on their respective local data. The result of the operation remains distributed across the network.

Result $= ys = ys_0 @ ys_1 @ \ldots @ ys_{n-1}$

where, $ys_i \in$ processor$_i$

$i = 0, 1, \ldots, (n-1)$

The algorithm for the parallel implementation of **map** on all the four processor topologies is as follows.

```
Algorithm PAR_MAP (all topologies)
BEGIN
        FOR every processor$_i$ DO in PARALLEL
              map $f$ $xs_i$
END.
```

There is no communication involved in the implementation. The cost expression for the parallel implementation of **map** on any p-processor topology is identical and is given by:

$$C_{map} = \frac{n}{p} C_f \tag{6.22}$$

6.2.2 fold

In a sequential implementation of **fold**, there are $(n-1)$ applications of the operator f. (For the purposes of implementation and cost computations, $f\ a\ x_0$ is not considered, since the result is x_0). The sequential cost of **s_fold** is:

$$C^s_{s\_fold} = (n-1) C_f \tag{6.23}$$

In the case of **g_fold**, the input list is at least a list of lists, i.e. $D \geq 2$. If the list is assumed to comprise of n sublists, each of size m, then two cases need to be considered for the cost of the sequential function f.

- The case where the cost of f is independent of the sizes of the input lists on which it operates. The cost function can then be represented by C_f, and the cost expression for **g_fold** is identical to that of **s_fold**.

$$C^{sc}_{g\_fold} = (n-1)C_f \tag{6.24}$$

- The case where the cost of f depends on the sizes of the input lists on which it operates. The cost function in this case is represented by a function g, as follows:

  $$C_g = g(l_1, l_2)$$

  where l_1 and l_2 represent the lengths of the first and second argument lists, respectively. It may be noted that since each sublist is assumed to comprise of m elements, assumption 7 is still valid. Therefore, $l_1 = l_2 = m$. The cost expression is given by:

  $$C^{sf}_{g\_fold} = \sum_{i=1}^{n-1} g(im, m) \tag{6.25}$$

For all the topologies, after a parallel implementation of **fold**, the result is left on processor$_0$ (or the root processor). The algorithms for the two versions of **fold**, viz. **s_fold** and **g_fold**, are the same. The costs are different, because in the case of **g_fold**, the size of the data communicated increases at each step and must be accounted for in the computation of T_{com}.

6.2.2.1 fold on the Linear Array

The algorithm is as described in PAR_FOLD (linear array). The algorithm assumes that the number of processors is a power of 2, otherwise Step 2 of the algorithm will need slight modification. The cost expressions are unchanged.

```
Algorithm PAR_FOLD (linear array)
BEGIN
        1. FOR each processor$_i$ DO in PARALLEL
                /* evaluate partial results */
                result $\longleftarrow$ fold $f$ a $xs_i$
        2. $j \longleftarrow 1$
           WHILE $j \leq \log p$
                  combine partial results between pairs of
                  processors which are $j$ hops away
                  /* combination from right to left */
                  $j \longleftarrow 2 * j$
END.
```

- **s_fold**

$$C^{a}_{s\_fold} = \underbrace{C_f(\frac{n}{p} - 1)}_{step\_1} + \underbrace{(T^{1}_{com}(1 + 2 + \ldots + 2^{\log p - 1}) + C_f \log p)}_{step\_2}$$

On simplification, the cost of **s_fold** is given by:

$$C^{a}_{s\_fold} = C_f(\frac{n}{p} - 1 + \log p) + T^{1}_{com}(p-1). \qquad (6.26)$$

- **g_fold** - After performing Step 1 of the algorithm, each processor is left with $\frac{n}{p}m$ elements. In each iteration of Step 2, the number of elements to be communicated increases.

For the case where the cost of f is a function of its input list sizes:

$$C^{a_f}_{g\_fold} = \underbrace{\sum_{i=1}^{\frac{n}{p}-1} g(im, m)}_{step\_1} + \underbrace{\sum_{i=0}^{\log p - 1} (2^i T_{com}^{2^i \frac{n}{p} m} + g(2^i \frac{n}{p} m, 2^i \frac{n}{p} m))}_{step\_2}.$$

$$C^{a_f}_{g\_fold} = \sum_{i=1}^{\frac{n}{p}-1} g(im, m) + \sum_{i=0}^{\log p - 1} g(2^i \frac{n}{p} m, 2^i \frac{n}{p} m) + \sum_{i=0}^{\log p - 1} 2^i T_{com}^{2^i \frac{n}{p} m}. \qquad (6.27)$$

For the case where the cost of f is independent of the input sizes, the cost expression simplifies to the following:

$$C^{a_c}_{g\_fold} = \underbrace{C_f(\frac{n}{p} - 1)}_{step\_1} + \underbrace{\sum_{i=0}^{\log p-1} 2^i T_{com}^{2^i \frac{n}{p} m} + C_f \log p}_{step\_2}.$$

Grouping together the computation and communication terms, the cost is:

$$C^{a_c}_{g\_fold} = C_f(\frac{n}{p} - 1 + \log p) + \sum_{i=0}^{\log p-1} 2^i T_{com}^{2^i \frac{n}{p} m}. \qquad (6.28)$$

6.2.2.2 fold on the Hypercube

The **fold** operation on a hypercube of dimension d involves the following two steps.

1. $(\frac{n}{p} - 1)$ local applications of the argument function on each processor, in parallel.

2. d nearest neighbour communications of partial results to the root processor, and d applications of the argument function to partial results from pairs of processors. (The root processor is the smallest-numbered processor in the cube or subcube implementing the **fold**).

In the following algorithm, p_{no} represents the processor number, and p^j_{no} represents the number of the neighbouring processor in dimension j of the hypercube; $0 \leq j < d$, $0 \leq p_{no}, p^j_{no} < p$.

```
Algorithm PAR_FOLD (Hypercube)
BEGIN
        1. FOR every processor$_i$ DO in PARALLEL
                **result** $\longleftarrow$ **fold** $f$ a $xs_i$
        /* combine partial results on all processors */
        2. FOR every processor$_i$ DO
                FOR $j \longleftarrow 0$ TO $d-1$ DO
                    IF $p^j_{no} < p_{no}$ THEN
                        send **result** to processor $p^j_{no}$
                        EXIT
                    ELSE
                        receive **data** from $p^j_{no}$
                        **result** $\longleftarrow$ $f$ **result data**
END.
```

- **s_fold** - The cost of **s_fold** is given by:

$$C^h_{s\_fold} = \underbrace{C_f(\frac{n}{p} - 1)}_{step\_1} + \underbrace{d(T^1_{com} + C_f)}_{step\_2}$$

  which, on rearrangement gives:

$$C^h_{s\_fold} = C_f(\frac{n}{p} + d - 1) + T^1_{com}d. \tag{6.29}$$

- **g_fold** - If the cost of the sequential argument, f, is a function of input sizes, the cost is given by:

$$C^{h\_f}_{g\_fold} = \underbrace{\sum_{i=1}^{\frac{n}{p}-1} g(im, m)}_{step\_1} + \underbrace{\sum_{i=0}^{d-1} T^{2^i\frac{n}{p}m}_{com} + \sum_{i=0}^{d-1} g(2^i\frac{n}{p}m, 2^i\frac{n}{p}m)}_{step\_2}.$$

  On rearrangement, the cost expression is given by:

$$C^{h\_f}_{g\_fold} = \sum_{i=1}^{\frac{n}{p}-1} g(im, m) + \sum_{i=0}^{d-1}(T^{2^i\frac{n}{p}m}_{com} + g(2^i\frac{n}{p}m, 2^i\frac{n}{p}m)). \tag{6.30}$$

If the cost of the argument function is a constant given by C_f,

$$C_{g\_fold}^{h\_c} = C_f(\frac{n}{p} + d - 1) + \sum_{i=0}^{d-1} T_{com}^{2^i \frac{n}{p} m}. \qquad (6.31)$$

6.2.2.3 fold on the 2-D Torus/Mesh

The algorithm is based on an ideal torus where each processor has four physical neighbours. It exploits the wrap-around links in order to reduce the computation cost by half. The right and left halves of each row are treated as linear arrays and an algorithm similar to PAR_FOLD(linear array) is executed in each half. The partial results in each row are combined using the wrap-around link, so that the first column processors now hold the new set of partial results. A similar procedure is repeated along the first column processors, to obtain the final result on processor$_0$.

- **s_fold** -

$$\begin{aligned} C_{s\_fold}^{m\_ideal} &= \underbrace{C_f(\frac{n}{p} - 1)}_{step\_1} + \underbrace{T_{com}^1(\frac{p_1}{2} - 1) + C_f \log \frac{p_1}{2}}_{step\_2} + \underbrace{(T_{com}^1 + C_f)}_{step\_3} \\ &+ \underbrace{T_{com}^1(\frac{p_2}{2} - 1) + C_f \log \frac{p_2}{2}}_{step\_4a} + \underbrace{T_{com}^1 + C_f}_{step\_4b} \end{aligned}$$

Grouping like terms together, the cost is given by:

$$C_{s\_fold}^{m\_ideal} = C_f(\frac{n}{p} + 1 + \log \frac{p_1}{2} + \log \frac{p_2}{2}) + T_{com}^1(\frac{p_1 + p_2}{2}). \qquad (6.32)$$

```
Algorithm PAR_FOLD (ideal-mesh)
BEGIN
        1. FOR every processor$_i$ DO in PARALLEL
                 result $\longleftarrow$ fold $f$ a $xs_i$
        2. $j \longleftarrow 1$
           WHILE $j \leq \log \frac{p_1}{2}$
                Right (Left) half processors DO
                       combine partial results between pairs of
                       processors $j$ hops away
                       /* from left (right) to right (left) */
                $j \longleftarrow 2 * j$
        3. Last and first-column processors DO
                IF last-column processor THEN
                    send result to first processor in row
                ELSE
                    receive data from last processor in row
                    result $\longleftarrow$ $f$ result data
        4. IF first-column processor THEN
           a. $j \longleftarrow 1$
              WHILE $j \leq \log \frac{p_2}{2}$
                    Bottom (Top) half processors DO
                        combine partial results between pairs of
                        processors $j$ hops away
                    /* from top (bottom) to bottom (top) */
                    $j \longleftarrow 2 * j$
           b. IF last processor THEN
                  send result to processor$_0$
              processor$_0$ DO
                    receive data from last processor
                    result $\longleftarrow$ $f$ result data
END.
```

- **g_fold** -

$$\begin{aligned} C_{g\_fold}^{mf\_ideal} &= \underbrace{\sum_{i=1}^{\frac{n}{p}-1} g(im,m) +}_{step\_1} \underbrace{\sum_{i=0}^{\log\frac{p_1}{2}-1} (2^i T_{com}^{2^i fracnpm} + g(2^i \frac{n}{p}m, 2^i \frac{n}{p}m)) +}_{step\_2} \\ & \underbrace{T_{com}^{\frac{p_1}{2}\frac{n}{p}m} + g(\frac{p_1}{2}\frac{n}{p}m, \frac{p_1}{2}\frac{n}{p}m) +}_{step\_3} \\ & \underbrace{\sum_{i=0}^{\frac{p_2}{2}-1} (2^i T_{com}^{2^i \frac{n}{p_2}m} + g(2^i \frac{n}{p_2}m, 2^i \frac{n}{p_2}m)) +}_{step\_4a} \\ & \underbrace{T_{com}^{\frac{n}{2}m} + g(\frac{n}{2}m, \frac{n}{2}m)}_{step\_4b}. \end{aligned} \tag{6.33}$$

$$\begin{aligned} C_{g\_fold}^{mc\_ideal} &= C_f(\frac{n}{p} + 1 + \log\frac{p_1}{2} + \log\frac{p_2}{2}) + \sum_{i=0}^{\log\frac{p_1}{2}-1} (2^i T_{com}^{2^i \frac{n}{p}m}) + T_{com}^{\frac{p_1}{2}\frac{n}{p}m} + \\ & \sum_{i=0}^{\log\frac{p_2}{2}-1} (2^i T_{com}^{2^i \frac{n}{p_2}m}) + T_{com}^{\frac{n}{2}m}. \end{aligned} \tag{6.34}$$

If the mesh does not have a wrap-around connection on all the rows, then all the links will have to be traversed in the horizontal direction. However, in the vertical direction, the wrap-around link can still be used. Steps 2 and 3 of the algorithm would require minor modification. Then, it is easily shown that the costs are modified as follows:

$$C_{s\_fold}^{m} = C_f(\frac{n}{p} + \log p_1 + \log\frac{p_2}{2}) + T_{com}^{1}(p_1 + \frac{p_2}{2} - 1). \tag{6.35}$$

$$\begin{aligned} C_{g\_fold}^{m\_f} &= \sum_{i=1}^{\frac{n}{p}-1} g(im,m) + \sum_{i=0}^{\log p_1 - 1} (2^i T_{com}^{2^i \frac{n}{p}m} + g(2^i \frac{n}{p}m, 2^i \frac{n}{p}m)) + \\ & \sum_{i=0}^{\log\frac{p_2}{2}-1} (2^i T_{com}^{2^i \frac{n}{p_2}m} + g(2^i \frac{n}{p_2}m, 2^i \frac{n}{p_2}m)) + \\ & T_{com}^{\frac{n}{2}m} + g(\frac{n}{2}m, \frac{n}{2}m). \end{aligned} \tag{6.36}$$

$$\begin{aligned} C^{m\_c}_{g\_fold} &= C_f(\frac{n}{p} + \log p_1 + \log\frac{p_2}{2}) + \sum_{i=0}^{\log p_1 - 1} 2^i T_{com}^{2^i \frac{n}{p} m} + \\ &\sum_{i=0}^{\log\frac{p_2}{2}-1} 2^i T_{com}^{2^i \frac{n}{p_2} m} + T_{com}^{\frac{n}{2}m}. \end{aligned} \quad (6.37)$$

6.2.2.4 fold on the Tree

```
Algorithm PAR_FOLD ($s$-ary tree)
BEGIN
        1. FOR every processor$_i$ DO in PARALLEL
                **result** $\longleftarrow$ **fold** $f$ a $xs_i$
        2. FOR every leaf processor DO
                send **result** to parent
           FOR every non-leaf processor DO
                FOR $i \longleftarrow 1$ to $s$ DO
                        receive **data** from child$_i$
                        **result** $\longleftarrow$ $f$ **result data**
                        IF (not root processor) THEN
                            send **result** to parent
END.
```

For a tree of depth d, the partial results take d steps to reach the root processor. At each level, there are s communications and s applications of the function f in parallel.

- **s_fold** - The cost expression is given by:

$$C^t_{s\_fold} = C_f(\frac{n}{p} - 1 + ds) + ds(T^1_{com}). \quad (6.38)$$

- **g_fold** - The cost expression is given by:

$$\begin{aligned} C^{t\_f}_{g\_fold} &= \sum_{i=1}^{\frac{n}{p}-1} g(im, m) + s\sum_{j=0}^{d-1} T_{com}^{\frac{s^{d-j}-1}{s-1}\frac{n}{p}m} + \\ &\sum_{j=0}^{d-1}\sum_{i=0}^{s-1} g(\frac{n}{p}m + i\frac{s^{d-j}-1}{s-1}\frac{n}{p}m, \frac{s^{d-j}-1}{s-1}\frac{n}{p}m) \end{aligned} \quad (6.39)$$

If the cost of f is independent of the size of its arguments, then

$$C^{t\_c}_{g\_fold} = C_f(\frac{n}{p} + ds - 1) + s\sum_{j=0}^{d-1} T_{com}^{\frac{s^{d-j}-1}{s-1}\frac{n}{p}m}. \tag{6.40}$$

The first term in Equation 6.39 is obtained from Step 1 of the algorithm. At level j in the tree (the root is at level 0, the leaves are at level $d-1$), each processor would have to send data that has been gathered from $\frac{s^{d-j}-1}{s-1}$ processors. At the end of Step 1, each processor has $\frac{n}{p}m$ data items. Since there are s communications at each level, and there are d levels in the tree, the total communication cost is given by the second term in Equation 6.39. At each level in the tree, a processor makes s applications of the function f. Each processor makes the first application of f on its local data and the data gathered from the leftmost child. For the second application of f, the size of the first argument is the sum of the sizes of the arguments in the first call to f. The same reasoning is valid for all the s applications of f at any level in the tree. For a non-leaf node at level j, each child node would have gathered data from $\frac{s^{d-j}-1}{s-1}$ nodes. Since there are d levels in the tree, the total cost of the function applications is derived to be the third term in Equation 6.39.

6.2.3 scan

The sequential implementation of **scan** has $(n-1)$ applications of the operator f. The sequential cost of **scan** is, therefore, the same as the sequential cost of **fold**. As in the case of **fold**, $g(l_1, l_2)$ represents the cost of the argument of **g_scan**, when it is a function of the sizes of its input lists. Once again, $l_1 = l_2 = m$.

$$C^{s}_{s\_scan} = (n-1)C_f \tag{6.41}$$

$$C^{s_f}_{g\_scan} = \sum_{i=1}^{n} g(im, m) \tag{6.42}$$

$$C^{s_c}_{g\_scan} = (n-1)C_f \tag{6.43}$$

The algorithms for the parallel implementations of **s_scan** and **g_scan** are identical, although the cost expressions are different. The result of the parallel **scan** operation is distributed across the network as follows:

Result $= ys = ys_0@ys_1@\ldots@ys_{n-1}$

where, $ys_i \in$ processor$_i$.

In the following algorithms, *last* is a function that returns the last element of a list.

6.2.3.1 scan on the Linear Array

The partial results reach the last processor in the array in (p-1) steps. The total cost is once again obtained by adding the costs for both the steps in order.

- **s_scan**

$$C^a_{s\_scan} = \underbrace{C_f(\frac{n}{p} - 1)}_{step\_1} + \underbrace{((p-1)(T^1_{com} + C_f) + C_f(\frac{n}{p} - 2))}_{step\_2}$$

On simplification, the total cost is given by:

$$C^a_{s\_scan} = C_f(2\frac{n}{p} + p - 4) + T^1_{com}(p-1). \qquad (6.44)$$

```
Algorithm PAR_SCAN (linear array)
BEGIN
        1. FOR every processor$_i$ DO in PARALLEL
                **result** $\longleftarrow$ **scan** $f$ a $xs_i$
                **last_result** $\longleftarrow$ *last* **result**
        2. processor$_0$ DO
              send **last_result** to processor$_1$
           FOR every processor except processor$_0$ DO
                receive **data** from left neighbour
                **last_result** $\longleftarrow$ $f$ **data last_result**
                send **last_result** to right neighbour
                FOR $j \longleftarrow 1$ TO $(\frac{n}{p} - 1)$ DO
                      **result**$_j$ $\longleftarrow$ $f$ **data result**$_j$
                **result**$_{\frac{n}{p}}$ $\longleftarrow$ **last_result**
END.
```

- **g_scan**

$$C^{af}_{g\_scan} = \underbrace{\sum_{i=1}^{\frac{n}{p}-1} g(m, im)}_{step\_1} + \underbrace{\sum_{i=1}^{p-1}(T_{com}^{i\frac{n}{p}m} + g(i\frac{n}{p}m, \frac{n}{p}m))}_{step\_2} + \underbrace{\sum_{i=1}^{\frac{n}{p}-2} g((p-1)\frac{n}{p}m, im)}_{step\_2} \qquad (6.45)$$

For the input-independent case, the cost of **g_scan** is given by:

$$C^{ac}_{g\_scan} = C_f(2\frac{n}{p} + p - 4) + \sum_{i=1}^{p-1} T_{com}^{i\frac{n}{p}m}. \qquad (6.46)$$

6.2.3.2 scan on the Hypercube

```
Algorithm PAR_SCAN (hypercube)
BEGIN
        1. FOR every processor$_i$ DO in PARALLEL
                **result** ⟵ **scan** $f$ a $xs_i$
                **last_result** ⟵ *last* **result**
        2. FOR all processors DO
                FOR $j$ ⟵ 0 to $(d-1)$ DO
                        IF (bit$_j$ of processor_number = 0) THEN
                            send **last_result** to neighbour in dimension $j$
                        ELSE
                            receive **data** from neighbour in dimension $j$
                            **last_result** ⟵ $f$ **data last_result**
                            store **data** in **data_list**
        3. FOR each subcube$_i$ of dimension $> 0$ DO
                communicate partial results from lower-numbered
                subcubes to lower-numbered processors in subcube$_i$
        4. FOR all processors DO
                **par_result** ⟵ **fold** $f$ a **data_list**
                /* data_list has the partial results
                   from lower-numbered processors */
                FOR $j$ ⟵ 1 to $(\frac{n}{p}-1)$ DO
                        **result**$_j$ ⟵ f **par_result result**$_j$
                **result**$_{\frac{n}{p}}$ ⟵ **last_result**
END.
```

Note that step 3 in the algorithm would involve at most $(d-1)$ communications. The cost expressions are derived in the following manner.

- **s_scan** - Adding the costs for each of the steps, in order,

$$C^h_{s\_scan} = \underbrace{C_f(\frac{n}{p}-1)}_{step\_1} + \underbrace{d(C_f + T^1_{com})}_{step\_2} + \underbrace{T^1_{com}(d-1)}_{step\_3} + \underbrace{C_f(d-1) + C_f(\frac{n}{p}-2)}_{step\_4}.$$

Grouping like terms provides the following:

$$C^h_{s\_scan} = C_f(2\frac{n}{p} + 2d - 4) + T^1_{com}(2d-1). \qquad (6.47)$$

- **g_scan** - Representing the cost function by $g(l_1, l_2)$ gives:

$$C^{hf}_{g\_scan} = \sum_{i=1}^{\frac{n}{p}-1} g(m, im) + \sum_{i=0}^{d-1}(T_{com}^{2^i\frac{n}{p}m} + g(2^i\frac{n}{p}m, 2^i\frac{n}{p}m)) + (d-1)T_{com}^{2^{d-1}\frac{n}{p}m} + \sum_{i=1}^{\frac{n}{p}-2} g((p-1)\frac{n}{p}m, im). \qquad (6.48)$$

The cost for the input-independent case is given by:

$$C^{hc}_{g\_scan} = C_f(2\frac{n}{p} + 2d - 4) + \sum_{i=0}^{d-1} T_{com}^{2^i\frac{n}{p}m} + (d-1)T_{com}^{2^{(d-1)}\frac{n}{p}m}. \qquad (6.49)$$

6.2.3.3 scan on the 2-D Torus/Mesh

The algorithm is based on the one described in [Akl89].

- **s_scan** - If the wrap-around links are not used, then the cost is given by:

$$C1^m_{s\_scan} = \underbrace{C_f(\frac{n}{p}-1)}_{step\_1} + \underbrace{(T^1_{com} + C_f)(p_1 - 1)}_{step\_2} + \underbrace{(C_f(p_2-1) + T^1_{com}(p_2-1) + T^1_{com}(p_1-1))}_{step\_3} + \underbrace{C_f(1 + \frac{n}{p} - 1)}_{step\_4}.$$

The final cost expression is given by:

$$C1^m_{s\_scan} = C_f(2\frac{n}{p} + p_1 + p_2 - 3) + T^1_{com}(2p_1 + p_2 - 3). \qquad (6.50)$$

```
Algorithm PAR_SCAN (mesh)
BEGIN
        1. FOR every processor$_i$ DO in PARALLEL
                **result** $\longleftarrow$ **scan** $f$ a $xs_i$
                **last_result** $\longleftarrow$ *last* **result**
        2. /* update along the rows in parallel */
           FOR all processors DO
                IF (not first-column processor) DO
                    receive **data$_1$** from left neighbour
                    **last_result** $\longleftarrow$ $f$ **data$_1$** **last_result**
                forward **last_result** to right neighbour
        3. FOR processors in last column DO
                /* send partial results of each row to subsequent row */
                IF (not first-row processor) DO
                    receive **data$_2$** from neighbour in previous row
                    **last_result** $\longleftarrow$ $f$ **data$_2$** **last_result**
                forward **last_result** to neighbour in next row
                IF (not first-row processor) DO
                    forward **data$_2$** to left neighbour
           FOR all other processors DO
                IF (not first-row processor) DO
                    receive **data$_2$** from right neighbour
                    forward **data$_2$** to left neighbour
                                /* except first-column processors */
        4. FOR all processors DO
                **data** $\longleftarrow$ $f$ data$_2$ data$_1$
                FOR $j$ $\longleftarrow$ 1 to $(\frac{n}{p} - 1)$ DO
                        **result$_j$** $\longleftarrow$ $f$ **data result$_j$**
END.
```

- In step 3, instead of sending the partial results from the previous rows all along the length of the row, the wrap-around could be used to communicate it to the neighbour in the first column. Then, both processors could communicate the partial results in opposite directions along the row, thereby reducing the communication costs. Note that processors in the first row do not have to communicate partial results to neighbours in that row. So, even in the case of the non-uniform mesh, the cost can be improved to:

$$C^{m}_{s\_scan} = C_f(2\frac{n}{p} + p_1 + p_2 - 4) + T^1_{com}(\frac{3}{2}p_1 + p_2 - 1). \tag{6.51}$$

- **g_scan** - With wrap-around links and a cost function $g(l_1, l_2)$:

$$\begin{aligned} C^{mf}_{g\_scan} = & \sum_{i=1}^{\frac{n}{p}-1} g(im, m) + \sum_{i=1}^{p_1-1} (T_{com}^{i\frac{n}{p}m} + g(i\frac{n}{p}m, \frac{n}{p}m)) + \\ & \sum_{i=1}^{p_2-1} (T_{com}^{i\frac{n}{p}mp_1} + g(i\frac{n}{p}mp_1, \frac{n}{p}mp_1)) + \\ & \frac{p_1}{2}T_{com}^{\frac{n}{p}mp_1(p_2-1)} + g(\frac{n}{p}mp_1(p_2-1), \frac{n}{p}m(p_1-1)) + \\ & \sum_{i=1}^{\frac{n}{p}-1} g(\frac{n}{p}m(p-1), im) \end{aligned} \tag{6.52}$$

For the input-independent argument function the cost simplifies to:

$$\begin{aligned} C^{mc}_{g\_scan} = & C_f(2\frac{n}{p} + p_1 + p_2 - 3) + \sum_{i=1}^{p_1-1} T_{com}^{i\frac{n}{p}m} + \\ & \sum_{i=1}^{p_2-1} T_{com}^{i\frac{n}{p}mp_1} + \frac{p_1}{2}T_{com}^{\frac{n}{p}mp_1(p_2-1)}. \end{aligned} \tag{6.53}$$

6.2.3.4 scan on the s-ary Tree

The parallel implementation on an s-ary tree is a modification of the version described in [Ble89], for binary trees. Assumption 4 requires modification for this algorithm. The elements of the input list are initially at the leaves of the tree. Any communication costs incurred in achieving this distribution will also be accounted for when the cost for the entire problem is computed.

$xs = xs_0@xs_1@\ldots@xs_{\frac{n}{s^d}-1}$

where, $xs_i \in leaf_i$; $\qquad i \in 0, 1, \ldots, (n-1)$

Leaves are numbered from left to right. A tree of arity s and a depth d has s^d leaves. Therefore, $\mid xs_i \mid = \frac{n}{s^d}$.

```
Algorithm PAR_SCAN (s-ary tree)
BEGIN
    1. FOR every leaf processor DO in PARALLEL
            $\mathbf{result} \longleftarrow \mathbf{scan}\ f$ a $xs_i$
            $\mathbf{last\_result} \longleftarrow$ last $\mathbf{result}$
            send $\mathbf{last\_result}$ to parent
    2. FOR all non-leaf processors DO in PARALLEL
            FOR $j \longleftarrow 0$ to $(s-1)$ DO
                receive $\mathbf{data}$ from child$_j$
                $\mathbf{temp}_j \longleftarrow \mathbf{data}$
            $\mathbf{result} \longleftarrow \mathbf{scan}\ f$ a $\mathbf{temp}$
            IF (root processor) THEN
              FOR $j \longleftarrow 0$ to (s-1) DO
              send $\mathbf{result}_j$ to child$_j$
            ELSE
                send (last $\mathbf{result}$) to parent
    3. FOR all processors, except root and leaves DO
            receive $\mathbf{data}$ from parent
            send $\mathbf{data}$ to child$_0$
            FOR $j \longleftarrow 1$ to $(s-1)$ DO
                $\mathbf{temp}_j \longleftarrow f\ \mathbf{data}\ \mathbf{result}_{j-1}$
                send $\mathbf{result}_j$ to child$_j$
    4. Leaf processors DO
           receive $\mathbf{data}$ from parent
           $\mathbf{result}_0 \longleftarrow \mathbf{data}$
           FOR $j \longleftarrow 1$ to $(\frac{n}{s^d} - 1)$ DO
               $\mathbf{result}_j \longleftarrow f\ \mathbf{data}\ \mathbf{result}_j$
    5. Root processor$_0$ DO
            send (last $\mathbf{result}$) to rightmost leaf
END.
```

Step 5 in the algorithm is essential since the last element of the resulting list will remain at the root. In an actual implementation, the last element can be communicated along with $\mathbf{temp}_{s-1}$ in step 3 of the algorithm. This will not increase the communication cost very significantly. In the following cost computations, Step 5 is assumed to be combined with Step 3, and the cost of communicating the additional element is ignored, to simplify the cost expression. After the **scan** operation the result is left distributed across the leaves of the tree.

Given a tree of depth d, the results take d steps to reach the root in step 2, and a further d steps for the results to reach the leaves again in step 4, with each step involving s communications.

- **s_scan**

$$\begin{aligned} C^t_{s\_scan} &= \underbrace{C_f(\frac{n}{s^d} - 1)}_{step\_1} + \underbrace{d(sT^1_{com} + C_f(s-1))}_{step\_2} + \\ & \underbrace{((d-1)(s-1)C_f + dsT^1_{com})}_{step\_3} + \underbrace{C_f(\frac{n}{s^d} - 1)}_{step\_4} \end{aligned}$$

$$C^t_{s\_scan} = C_f(2\frac{n}{s^d} + (s-1)(2d+1)) + T^1_{com}2ds. \qquad (6.54)$$

- **g_scan** - Different data sizes will be communicated in parallel on different branches of the tree during the down-sweep (i.e. steps 3 and 4). The largest data size is communicated along the rightmost branch of the tree, since the partial results from all the nodes to its left must reach the nodes on this branch. Also, the size of the data communicated increases with each level down the tree. The non-uniformity in data transfers results in a more complicated cost expression.

If $k = (s-1)s^{d-1}\frac{n}{s^d}m$ then

$$\begin{aligned}
C^{tf}_{g\_scan} \;=\; & \underbrace{\sum_{i=1}^{\frac{n}{s^d}-1} g(im, m)}_{step\_1} + \underbrace{\sum_{i=0}^{d-1} sT_{com}^{s^i\frac{n}{s^d}m} + \sum_{i=0}^{d-1}\sum_{j=1}^{s-1} g(js^i\frac{n}{s^d}m, s^i\frac{n}{s^d}m) +}_{step\_2} \\
& \underbrace{\sum_{j=0}^{s-1} T_{com}^{js^{d-1}\frac{n}{s^d}m} + (s-1)\sum_{i=0}^{d-2} g(k, s^i\frac{n}{s^d}m) + T^{k}_{com} +}_{step\_3} \\
& \underbrace{s(T_{com}^{k+\frac{n}{s^d}ms^{d-2}} + T_{com}^{k+\frac{n}{s^d}m(s^{d-2}+s^{d-3})} + \ldots + T_{com}^{k+\frac{n}{s^d}m\sum_{i=1}^{d-2}s^i})}_{step\_3} + \\
& \underbrace{(s-1)T_{com}^{k+\frac{n}{s^d}m\sum_{i=0}^{d-2}s^i}}_{step\_3} + \underbrace{\sum_{i=1}^{\frac{n}{s^d}-1} g(k+\frac{n}{s^d}m\sum_{j=0}^{d-2}s^j, im)}_{step\_4}.
\end{aligned}$$

The first three terms constitute the cost for the up-sweep and are easily deduced. The cost for the down-sweep is deduced as follows. At the root, the identity element is communicated to the leftmost child, data of size $s^{d-1}\frac{n}{s^d}m$ (which is the size of the first partial result) is communicated to child$_1$, ..., data of size $(s-1)s^{d-1}\frac{n}{s^d}m$ is communicated to child$_{s-1}$. The cost of communicating the identity element can be assumed to be zero, since it does not actually have to be communicated. These communications account for the fourth term in the cost expression. At each level in the tree, there are $(s-1)$ applications of f performed in parallel. However, since the maximum data size is handled by the rightmost node at each level, the cost for that level will be determined by the cost of this node. Since there are $(d-1)$ non-leaf nodes (excluding root) on each branch in a tree of depth d, the cost for the $(s-1)$ applications of f is given by the fifth term in the expression. The terms that follow represent the communication costs. They account for the cost of communicating the increasing data sizes to the child nodes, from the rightmost node at each level, since largest sizes are communicated from the rightmost node. The last expression accounts for the update performed at the leaves of the tree, after the down-sweep. Once again, the cost is based on the rightmost leaf, since it would handle the largest data size. In practice, the implementation cost could be reduced if the parent at each level started by sending the partial results to the rightmost child.

6.2.4 filter

From the definition of **filter**, it is clear that the size of the output list is data-dependent. After this operation, the sizes of the sublists on different processors may be different. The exact nature of the output distribution is not predictable. However, since all the subsequent phases of the problem would assume balanced load, it appears that a load balancing operation needs to be performed after a phase containing the **filter** function. This would involve gathering the results from all the processors and redistributing them (if necessary), so that the load is balanced for the next phase. This may prove to be unnecessary if the **filter** operation results in balanced load for some data set. On the other hand, there may be extreme cases in which the output sublists on some processors are of the same size as the corresponding input sublists, (i.e. all the elements retained), but others in which the output sublists are empty. In the latter case, data re-distribution is crucial to obtain good performance.

The sizes involved in gathering and redistributing the data are also unknown. The analyser, therefore, adopts a pessimistic approach in computing the cost of **filter** and assumes that the size of the output list is equal to the size of the input list. If there are several phases following the one involving the **filter** function, then the predicted costs may be much worse than the actual costs, and this in turn is dependent on the nature of the data set.

However, since the pessimistic approach assumes the availability of the entire data set after the **filter** operation, it may, in the worst case, result in the selection of a parallel implementation when a sequential one would have proved to be more cost-effective. A pessimistic approach to cost computation may result in an overly-optimistic approach to the selection of a parallel implementation. The incorporation of a profiler would alleviate this problem.

The sequential cost of **filter** is given by:

$$C^s_{filter} = C_f n. \tag{6.55}$$

The parallel implementation of **filter** on all the topologies is the same, except that each uses its own *gather* algorithm.

```
Algorithm PAR_FILTER (all topologies)
BEGIN
        1. FOR every processor_i DO in PARALLEL
                result ⟵ filter $f$ $xs_i$
        2. gather result from all processors
END.
```

The cost of the parallel implementation is given by:

$$C^p_{filter} = C_f \frac{n}{p} + G \tag{6.56}$$

where G is the cost of gathering n elements distributed across p processors on the particular topology. The cost of redistributing the result of the **filter** operation is not accounted for in its cost, but will be, by the subsequent phase. This will be zero if the latter is sequential.

6.2.5 inits and tails

For an input list of size n, the function **inits** produces $(n + 1)$ sublists that successively increase in size. The size of sublist$_i$ is *one* more than the size of sublist$_{i-1}$. The total number of elements in all the sublists is given by:

$$\begin{aligned} total &= 0 + 1 + 2 + \cdots + n \\ total &= \frac{n(n+1)}{2}. \end{aligned}$$

The computation of the function **inits** only involves list processing costs. In a parallel implementation of **inits**, the communication cost would make a significant contribution to the total cost. As discussed in Section 4.1, **inits** can be expressed in terms of the function **scan** and can, therefore, be implemented in parallel by using the parallel **scan** algorithm. However, this is not desirable for two reasons.

- Since **inits** produces sublists of increasing sizes, the load on the processors would be imbalanced after the **inits** operation, even if there were the same number of elements on all the processors initially. This would cause non-uniform processor utilisation for subsequent phases of the problem.

- During the implementation of the **scan** algorithm, the partial results exchanged between processors would be of unequal sizes. The size of the data communicated increases with the progress of the algorithm. This is undesirable, and it can be shown that it leads to higher communication costs compared to the alternative suggested below.

In order to obtain balanced load after the **inits** operation, ideally, the total number of elements from all the sublists on each processor should be $\frac{n(n+1)}{2p}$. This is achieved by allowing different processors to have different numbers of sublists. Some processors have a larger number of sublists of smaller sizes, while others have a smaller number of sublists of larger sizes. Each processor calculates the number of sublists that will be resident on it after the **inits** operation. In the following discussion, the *nil* list which is the first element of the result of **inits** is ignored.

After the **inits** operation, the number of sublists on processor$_i$ is represented by n_i. In particular, on processor$_0$ the number of sublists is n_0, and the length of sublist$_i$ is i. The condition which has to be satisfied in order to obtain balanced load is as follows:

Total number of elements in all sublists on processor$_0$ $= \frac{n(n+1)}{2p}$.
$\Rightarrow 1 + 2 + 3 + \cdots + n_0 = \frac{n(n+1)}{2p}$
$\Rightarrow \frac{n_0(n_0+1)}{2} = \frac{n(n+1)}{2p}$
Solving for n_0 gives,

$$n_0 = \frac{-1 + \sqrt{1 + \frac{4n(n+1)}{p}}}{2}. \qquad (6.57)$$

The last sublist on processor$_0$ is of length n_0 and consequently, the first sublist on processor$_1$ is of length $n_0 + 1$. Again, the condition which has to be satisfied to obtain a balanced load results in the following deduction.

Total number of elements in all sublists on processor$_1$ $= \frac{n(n+1)}{2p}$.
$\Rightarrow (n_0 + 1) + (n_0 + 2) + \cdots + (n_0 + x) = \frac{n(n+1)}{2p}$
$\Rightarrow x = \frac{\sqrt{(2n_0+1)^2 + \frac{4n(n+1)}{p}} - (2n_0+1)}{2}$

This could be generalised to the following.

For every processor$_i$, where $i = 1, 2, \ldots, (p-1)$
length of first sublist $= n_{i-1} + 1$
length of last sublist $= n_{i-1} + x$
where, $x = \frac{\sqrt{(2n_{i-1}+1)^2 + \frac{4n(n+1)}{p}} - (2n_{i-1}+1)}{2}$
and
length of first sublist on processor$_0$ $= 1$
length of last sublist on processor$_0$ $= n_0$ (Equation 6.57)

Each processor first determines the number and sizes of the sublists that it should generate, which is done locally. This implies that each processor should possess a copy of the input list. If the input list is not already resident on all the processors, then it should be broadcast to them. If it is already scattered across the network, then a total exchange operation would have to be performed, so that each processor obtains a copy of the entire list. If C_{be} represents the cost of Broadcast or Total Exchange, the cost of **inits** is given by:

$$C^p_{inits} = C_{be} \tag{6.58}$$

List processing costs involved in generating the sublists are not accounted for in the cost expression since the analyser accounts for such costs for every output list that is constructed. A similar strategy is applied to **tails**.

6.2.6 cross_product

The costs for both **r_cross_product** and **c_cross_product** are identical and their costs are therefore discussed jointly. Since **cross_product** has two argument lists, an additional assumption for cost calculations is required.

The second input list is represented by ys

where, $\mid ys \mid = m$ and $ys \in$ processor$_0$

The second list is assumed to be resident on a single processor, so the cost for a parallel implementation must account for the cost of distributing this list to the other processors. The sequential cost of **cross_product** is given by:

$$C^s_{cross\_product} = C_f nm \tag{6.59}$$

```
Algorithm PAR_CROSS_PRODUCT (all topologies)
BEGIN
        Broadcast ys to all the processors
        FOR every processor_i DO in PARALLEL
              result ⟵ cross_product f $xs_i$ ys
END.
```

From the definitions of **r_cross_product** and **c_cross_product**, it is clear that if the first input list is distributed across p processors in the network, then the second list would need to be broadcast to all the processors in order to evaluate the **cross_product** in parallel. The parallel implementations are identical on all the topologies, except for the different algorithms and the corresponding costs involved in the broadcast operation. The results are left distributed across the network.

The cost is given by:

$$C^{p}_{cross\_product} = B + C_f \frac{n}{p} m \tag{6.60}$$

where, B is the cost of broadcasting m elements to p processors, on the particular topology.

6.2.7 composition

It is clear from the definition of **composition** that each phase uses the results of the previous phase. A possible way of implementing **composition** in parallel would be to pipeline the intermediate results, so that the implementation of the next phase could be overlapped with that of the current phase [Kel89]. However, in the current scheme, **composition** is implemented sequentially. Any potential parallelism is exploited within each phase of the **composition**.

The cost of a **composition** involving k phases is, therefore, given by:

$$C_{composition} = \sum_{i=1}^{k} C_i \tag{6.61}$$

where, C_i is the cost of phase$_i$, together with any data rearrangement costs that would be incurred in its implementation.

6.2.8 map2 and zip

map2 is just an extension of **map** to cater for two argument lists. If both the input lists are already distributed across the network, then the cost of **map2** is identical to the cost of **map**. If not, then the additional cost of scattering the second list on the relevant processor network must be added to the cost of **map**. A similar strategy is followed for the parallel implementation of **zip**.

6.2.9 The iterative functions

As pointed out earlier, none of the iterative functions are themselves implemented in parallel. If the argument of an iterative function comprises of a composition of recognised functions, then the composition could be implemented in parallel using the analysis described in Section 5.2. The cost of an iterative function is taken to be the cost of a *single* application of its argument function *f*. An implicit assumption here is that the costs of all the iterations are equal. Since the strategy for the selection of a parallel implementation relies only on cost comparisons, the absolute cost of all the iterations is not required. Moreover, in many cases, e.g. **iterate_cond**, it may not be possible to determine the number of iterations at compile-time.

At the end of each iteration, the results are left distributed. If the implementation of the next iteration requires the data to be distributed in the same way as left by the previous phase, then no data rearrangement cost is incurred. If not, the cost of data rearrangement is added to the cost of the iterative function.

In particular, in the case of a parallel implementation of **iterate_cond**, the result is left distributed across the processor network for the implementation of the next iteration. The *condition* is only evaluated on a *copy* of the result. This could save on potential communications overheads.

Again, there are limitations that arise as a result of assuming identical cost for every iteration. The model attempts to minimise potential discrepancies in the estimated costs, e.g. initial distribution costs at the start of the first iteration are not added to the cost of the iterative function if these costs will not be incurred by subsequent iterations. The idea is that such a cost will be negligible when averaged over a large number of iterations (assuming there are a fairly large number of iterations). On the other hand, if this initial distribution cost is added, then it will substantially increase the estimated implementation cost and may lead to the selection of a poor parallel implementation.

6.2.10 split

The introduction of **split** as a recognised function was motivated by the need to express divide-and-conquer type of applications. The function **split** takes two arguments, an integer $k > 0$ and a list of size n. The result of applying the function to the list is to split the list into k sublists, with $(k-1)$ sublists of size $\lceil \frac{n}{k} \rceil$ and the last sublist of length $n - (k-1)\lceil \frac{n}{k} \rceil$. **split** only involves list rearrangement costs which are computed by the analyser, based on the nature of the input list. The cost is represented by:

$$C^s_{split} = S_{n,k}. \tag{6.62}$$

For a parallel implementation of **split** with the list distributed across p processors, each processor splits the part of the list resident on it into $\lceil \frac{k}{p} \rceil$ parts. If k is not divisible by p, then the parallel implementation would result in the list being split into a slightly different number of sublists as compared to a sequential implementation for the same list. If it is the case that this would not affect the outcome of the final result itself, then this difference may be acceptable. An example is the merge sort program described in Section 4.3.2. Since the combining function *merge* is associative, slight differences in the manner in which the list is split would not affect the result. However, there may be cases where this might produce erroneous results. In such cases, the programmer is invited to use a variant of the function **split**, called **seq_split**. Its definition is identical to that of **split**, but it would force the analyser (and in turn the code generator) to consider only a sequential implementation for **split**.

```
Algorithm PAR_SPLIT (all topologies)
BEGIN
        FOR every processor$_i$ DO
              result ⟵ split $\frac{k}{p}$ $xs_i$
END.
```

Adopting the terminology mentioned earlier, the cost of the parallel implementation of **split** can be represented by:

$$C^{p}_{split} = S_{\frac{n}{p}, \frac{k}{p}} \tag{6.63}$$

If the value of k is unknown at compile-time, then the analyser assumes that it is equal to n. In other words, each sublist is assumed to be of size *one*. This again, is a pessimistic approach to cost computation, and could result in the selection of a poor parallel implementation. Again, profiling techniques could help in dealing with the problem.

6.2.11 The Combinator $\mathcal{R}^k$

The definition for $\mathcal{R}^2$, the case where a recognised function operates on two input lists, is as follows (refer to Section 4.2):

$\mathcal{R}^2$ F f_1 f_2 ... f_k g h xs = F f_1 f_2 ... f_k (g xs) (h xs)

where, F is a recognised function with 2 input lists

f_1, f_2, ..., f_k are parameters to F

g and h are functions that operate on lists.

In general, g and h could be any complex functions including other recognised functions. However, some situations may just require two copies of the same list, in which case both of these functions would be the identity function.

The present set of recognised functions have, at most, two argument lists, so the following discussion focuses on $\mathcal{R}^2$. The number of possibilities that arise in the selection of a parallel implementation of $\mathcal{R}^k$ makes it a complex scheduling problem in itself and will not be discussed here. In the case of $\mathcal{R}^2$, the discussed heuristic is used to prune the search-space, while attempting not to eliminate implementations that may eventually result in the least cost. This cannot, however, be guaranteed and in some cases the least-cost implementation may be eliminated, since the heuristic performs cost optimisation that is local to the phase under consideration. The heuristic computes the costs for the various possible implementations which are considered. The least-cost one is then selected.

C_g^s represents the sequential cost of g

$C_g^{p'}$ represents the cost of the parallel implementation of g on p' processors

$MAX(x, y)$ = if $(x > y)$ then x else y

$MIN(x, y)$ = if $(x < y)$ then x else y

$C_B^{p'}$ is the cost of broadcasting the list to p' processors in the topology

$C_{GS}^{p'}$ is the cost of gathering/scattering the list from/to p' processors in the topology.

```
HEURISTIC PAR_$\mathcal{R}^2$
BEGIN
        1. IF $g$ and $h$ are sequential functions THEN
              Gather list from $p$ processors
              a. Implement $g$ and $h$ on one processor
                    $C_1 \longleftarrow C^p_{GS} + C^s_g + C^s_h$
              b. Send copy of list to Processor$_1$
                 Implement $g$ on Processor$_0$
                 Implement $h$ on Processor$_1$
                    $C_2 \longleftarrow C^p_{GS} + MAX(C^s_g, C^s_h) + C^2_B$
              Implement F in parallel on $p$ processors
              $C_{R1} \longleftarrow C^p_F + C_1 +$ **data-rearrangement-costs**
              $C_{R2} \longleftarrow C^p_F + C_2 +$ **data-rearrangement-costs**
              $C_{\mathcal{R}} \longleftarrow MIN(C_{R1}, C_{R2})$
        2. IF $g$ and $h$ are recognised functions THEN
              a. Compute $C_{R1}$ and $C_{R2}$ as in step 1
                 $C_{Rp1} \longleftarrow MIN(C_{R1}, C_{R2})$
              b. /* assuming two copies of list available */
                 Implement $g$ in parallel on $p$ processors
                 Implement $h$ in parallel on $p$ processors
                    $C_2 \longleftarrow C^p_g + C^p_h$
              Implement F in parallel on $p$ processors
              $C_{Rp2} \longleftarrow C^p_F + C_2$
              $C_{\mathcal{R}} \longleftarrow MIN(C_{Rp1}, C_{Rp2})$
        3. IF $g$ is sequential and $h$ is recognised THEN
              $C_{\mathcal{R}} \longleftarrow C^p_{GS} + C^s_g + C^p_h + C^p_{GS} + C^p_F$
        4. OUTPUT $C_R$
END
```

The implementation heuristic for $\mathcal{R}^2$ follows the usual rules. If g and h are sequential functions and the input list is distributed, then the list has to be gathered before g and h can be evaluated. The two transformed lists can then be redistributed, depending on whether or not a parallel implementation is selected for F. If g and/or h involve recognised functions, then a distributed implementation could be selected for g and/or h, so that it results in the output data being distributed as required by F. This would save data rearrangement overheads. In any case, the resulting distribution of the two lists must be such that it is suitable for the implementation selected for F and any cost that is incurred in achieving this distribution is represented by the term, **data-rearrangement-costs**, in the heuristic.

6.2.12 get_neigh

The sequential implementation of **get_neigh** only involves list rearrangement and so the only cost of implementation comprises of list processing costs. If this cost is represented by G_n, then

$$C^s_{get\_neigh} = G_n. \tag{6.64}$$

The algorithms for the parallel implementation of **get_neigh** on the various topologies do not correspond to its definition in terms of the basic set of recognised functions. It may be observed that only the first and the last elements of (the part of) the list resident on each processor have non-local neighbours. Each processor is, therefore, involved in a communication routine that obtains these values. Then all the processors can execute a local **get_neigh** operation in parallel to obtain the final result, which is distributed across the network. In the following algorithms the functions *first* and *last* return the first and last elements of a list, respectively.

6.2.12.1 get_neigh on the Linear Array

```
Algorithm PAR_GET_NEIGH (Linear Array)
BEGIN
    1. FOR every processor DO
            result_left ⟵ first $xs_i$
            result_right ⟵ last $xs_i$
    2. Odd-numbered processors DO
            send result_left to left neighbour
            receive data_left from left neighbour
            /* this is the left neighbour for first element */
            send result_right to right neighbour
            receive data_right from right neighbour
            /* this is the right neighbour for last element */
       Even-numbered processors DO
            receive data_right from right neighbour
            send result_right to right neighbour
            Except processor$_0$ DO
                receive data_left from left neighbour
                send result_left to left neighbour
    3. FOR every processor DO
            /* perform get_neigh locally */
            result ⟵ get_neigh data_left data_right $xs_i$
END.
```

The algorithm comprises of two communication steps, and in each of them one list element is exchanged between pairs of processors. The total cost is:

$$C^a_{get\_neigh} = 4T^1_{com} + G_{\frac{n}{p}}. \tag{6.65}$$

6.2.12.2 get_neigh on the Hypercube

The described algorithm implements **get_neigh** in parallel on a hypercube of dimension d with p processors. It comprises of three stages. The first stage comprises of $(d-1)$ communication steps in which lower-numbered processors forward their first and last elements, i.e. the boundary values, to higher-numbered ones. This happens in parallel in the two subcubes of dimension $(d-1)$. The second stage is the exchange step in which the boundary values in the two subcubes are exchanged. The third stage again comprises of $(d-1)$ communication steps in which the higher-numbered processors communicate the boundary values to the lower-numbered ones. At the end of $2d$ steps, every processor will have obtained the neighbouring values for all its local elements. All the processors can then execute **get_neigh** locally to produce the result. The following notation is used in the algorithm.

p_no is the processor-number in binary representation; $0 \leq p\_no < p$

$neigh\_no_i$ is the number of the neighbouring processor in dimension i,
in binary representation

$is\_one(i, p\_no)$ returns TRUE if the last i bits of p_no are 1, FALSE otherwise

T is a 2-element array on each processor

If the cost of the initialisation step is ignored, then the cost of the algorithm for **get_neigh** is given by:

$$C^h_{get\_neigh} = T^2_{com}(d-1) + 2T^2_{com} + T^2_{com}(d-1) + G_{\frac{n}{p}}.$$

On simplification, the cost is given by:

$$C^h_{get\_neigh} = T^2_{com}2d + G_{\frac{n}{p}}. \tag{6.66}$$

The described algorithm results in a better implementation when compared to the one corresponding to gathering the list, executing **get_neigh** on a single processor and then redistributing the resulting list. The latter procedure would involve a *gather* operation, followed by an implementation of **get_neigh** on a single processor, followed by a *scatter* operation. The total cost in that case would be given by:

$$Cost^{n}_{get\_neigh} = 2(dK_0 + \frac{1}{K_1}\frac{n}{p}(p-1)) + G_n. \qquad (6.67)$$

```
Algorithm PAR_GET_NEIGH (hypercube)
BEGIN
        1. FOR every processor DO
                /* initialisation step */
                IF $p\_no$ is even THEN $M = 0$
                ELSE $M \longleftarrow$ largest $i$ s.t., $is\_one\ (i, p\_no)$ = TRUE;
                                        $0 < i < d$
                let $q_0, q_1, \ldots, q_M$ be 2-element queues
                insert first $xs_i$ into $q_0$
                insert last $xs_i$ into $q_M$
        2. FOR $i \longleftarrow 0$ TO $(d-2)$ DO
                /* stage 1 - forward communication */
                IF $(is\_one\ (i, p\_no))$ THEN
                    IF $(p\_no < neigh\_no_i)$ THEN
                        Send $q_i$ to $neigh\_no_i$
                        $q_i \longleftarrow$ NIL
                    ELSE
                        $T \longleftarrow$ data received from $neigh\_no_i$
                        IF $(p\_no - neigh\_no_i = 1)$ THEN
                            /* neighbour for first element received */
                            insert $T[1]$ into ln list
                            insert $T[0]$ into $q_{i+1}$
                        ELSE insert $T[0]$ into $q_{i-1}$, $T[1]$ into $q_{i+1}$
        /* CONTD ... */
```

```
Algorithm PAR_GET_NEIGH (hypercube)
CONTD
      3. IF ($is\_one$ ($i(= d-1), p\_no$)) THEN
            /* Stage 2 - Exchange step */
            IF ($p\_no < neigh\_no_i$) THEN
               Send $q_i$ to $neigh\_no_i$
               $T \longleftarrow$ data received from $neigh\_no_i$
            ELSE
               $T \longleftarrow$ data received from $neigh\_no_i$
               Send $q_i$ to $neigh\_no_i$
            IF ($i = 0$) THEN        /* 2-D hypercube */
               insert $T[0]$ and $T[1]$ into ln list
            ELSE
               insert $T[0]$ into $q_{i-1}$
               insert $T[1]$ into ln list
      4. FOR $i \longleftarrow (d-2)$ DOWNTO 0 DO
               /* stage 3 - Backward Communications */
               IF ($is\_one$ ($i, p\_no$)) THEN
                  IF ($p\_no < neigh\_no_i$) THEN
                     $T \longleftarrow$ data received from $neigh\_no_i$
                     insert $T[0]$ into ln list
                     IF ($i > 0$) THEN
                        insert $T[1]$ into $q_{i-1}$
                     ELSE insert $T[1]$ into ln list
                  ELSE
                     Send $q_i$ to $neigh\_no_i$
                     $q_i \longleftarrow$ NIL
      5. FOR every processor DO
               result $\longleftarrow$ get_neigh ln[0] ln[1] $xs_i$
END.
```

In Equation 6.66, $T^2_{com} = K_0 + 2\frac{1}{K_1}$ (See Equation 6.20). Clearly, for any value of $n > 2$, Equation 6.66 corresponds to a lower implementation cost as compared to Equation 6.67. The algorithm does not result in an order of magnitude decrease in cost. However, the implementation corresponding to Equation 6.67 would also suffer from much higher list processing costs, since $G_n > G_{\frac{n}{p}}$. The communication costs in that case would also be greater, since the size of the resulting list involved in the scatter operation is of a larger size than the input list.

6.2.12.3 get_neigh on the 2-D Torus/Mesh

```
Algorithm PAR_GET_NEIGH (2-D Torus/Mesh)
BEGIN
        1. /* identical to PAR_GET_NEIGH (Linear Array) */
        2. /* identical to PAR_GET_NEIGH (Linear Array) */
        3. First Column Processors DO
                Send data_left to south neighbour
           Last Column Processors DO
                Send data_right to north neighbour
        4. FOR every processor DO
                result ⟵ get_neigh data_left data_right $xs_i$
END.
```

The first two steps of the algorithm are identical to that of the linear array. In other words, each row of the torus is treated as a linear array, except for the wrap-around link. The processors in the first column treat the ones in the last column (and the same row) as their left neighbours and vice-versa. After the second step, each processor in the first column still requires the left neighbour for its first element and each processor in the last column requires the right neighbour for its last element. In the third step, processors in the first column obtain this value from their neighbour above and processors in the last column obtain it from their neighbour below. Steps 1 and 2 in the algorithm are identical to the ones in PAR_GET_NEIGH (Linear Array). The communication cost in Step 3 can be reduced by nearly half, by overlapping the *sends* in the two halves of the columns. The cost of the algorithm is then given by:

$$C^m_{get\_neigh} = T^1_{com}(\frac{p_2}{2} + 4) + G_{\frac{n}{p}}. \tag{6.68}$$

6.2.12.4 get_neigh on the Tree

```
Algorithm PAR_GET_NEIGH (s-ary Tree)
BEGIN
        1. FOR every Processor DO
                Create Queues $q_{up}$ and $q_{d_1}, q_{d_2}, \ldots, q_{d_s}$
        2. Leaf Processors DO
                insert *first* $xs_i$ and *last* $xs_i$ into $q_{up}$
           Non-leaf Processors DO
                insert *first* $xs_i$ into $q_{up}$; *last* $xs_i$ into $q_{d_1}$
        3. /* *Up-sweep* */
           Leaf Processors DO
                Send $q_{up}$ to parent
           Non-Leaf Processors DO
                FOR $i \longleftarrow 1$ to $s$ DO
                      Receive $q$ from child$_i$
                      IF $(i = 1)$ THEN
                         /* *received right neigh for last elm* */
                         Insert $q[1]$ into **local_neigh** list
                         $q_{d_{i+1}} \longleftarrow q[2]$
                      ELSE
                         $q_{d_{i-1}} \longleftarrow q[1]$
                         IF $(i+1) \leq s$ THEN
                            $q_{d_{i+1}} \longleftarrow q[2]$
                         ELSE
                            $q_{up} \longleftarrow q[2]$
        /* CONTD ... */
```

```
Algorithm PAR_GET_NEIGH (s-ary Tree)
CONTD
      4. /* Down-sweep */
         Root Processor DO
               Send $q_{d_i}$ to child$_i$
         Non-Leaf Processors DO
              Receive $q$ from parent
              /* received left neighbour for first elm */
              Insert $q[1]$ into local_neigh list
              IF two elements received THEN
                  /* processors in rightmost branch receive 1 value */
                  $q_{d_s} \longleftarrow q[2]$
                  Send $q_{d_i}$ to child$_i$
         Leaf Processors DO
              Receive $q$ from parent
              /* left neigh for first elm is $q[1]$ */
              /* right neigh for last elm is $q[2]$ */
              local_neigh $\longleftarrow q$
      5. FOR all processors DO
               result $\longleftarrow$ get_neigh local_neigh[1]
                              local_neigh[2] $xs_i$
END.
```

The algorithm comprises of two sweeps - an up-sweep in which processors send the elements which are meant for processors in the sub-trees to their left or right, to their respective parents; and a down-sweep in which the parents forward the acquired data to their respective children. Queues, q_{up}, to send data to parent and $q_{d_1}, q_{d_2}, \ldots, q_{d_s}$ to send data to child$_i$ ($1 \leq i \leq s$) respectively, are maintained by each processor. Obviously, the root processor does not need to maintain q_{up} and similarly, the leaf processors do not require queues q_{d_i}, but these are not explicitly mentioned in the algorithm. Steps 3 and 4 of the algorithm each involve s communications of two elements. For a tree of depth d, there are d such communications in each of steps 3 and 4, respectively. The cost is given by:

$$C^t_{get\_neigh} = T^2_{com} 2ds + G_{\frac{n}{p}}. \tag{6.69}$$

6.2.13 len

The sequential cost of **len** is straightforward.

$$C^s_{len} = C_f(n-1) \tag{6.70}$$

where, C_f in this case is the cost of a '+' operation.

The parallel implementation of **len** corresponds to those of its composing functions, viz. **map** and **fold** (refer to Section 4.2). In the following algorithm, PAR_FOLD represents the parallel **fold** algorithm on the relevant processor topology.

```
Algorithm PAR_LEN (all topologies)
BEGIN
        1. FOR every processor$_i$ DO in PARALLEL
               temp ⟵ map subst_1 $xs_i$
        2. result ⟵ PAR_FOLD plus 0 temp
END.
```

The cost of implementing **len** in parallel is given by:

$$C^p_{len} = C_{subst\_1}(\frac{n}{p} - 1) + C^{plus}_{PAR\_FOLD} \tag{6.71}$$

where, $C^{plus}_{PAR\_FOLD}$ is the cost of the parallel **fold** operation on the specific architecture with the sequential function *plus* as its argument.

It may prove to be expensive to implement **len** in parallel, since the arguments of the **map** and **fold** functions are trivial operations. The communication cost which is incurred in scattering the input list and in the implementation of **fold**, would probably make it more expensive to implement it in parallel than sequentially. However, if the elements of the input list are already scattered across the network, and the **len** function is encountered, then it may prove to be more expensive to gather the elements of the list for the purpose of merely computing its length. In such cases, the parallel implementation will be chosen by the analyser.

6.2.14 select

The **select** function does not involve any computation. For a list data structure the sequential cost of **select** is proportional to the length of the list that needs to be traversed to locate the required element. In the worst case, this length would be equal to the size of the list. Since the index of the element to be selected may not be known until run-time, the analyser adopts a pessimistic approach in computing the cost of **select**. The sequential cost of **select** is, therefore:

$$C^s_{select} \propto n \tag{6.72}$$

The constant of proportionality is the cost of list processing that is estimated by the analyser.

select has been defined in terms of a **fold** operation (refer to Section 4.2). However, from the definition, it is clear that the argument of **fold** is not an associative function. Hence, the usual parallel implementation for **fold** cannot be applied. In a distributed implementation, the element to be selected would be resident on some processor. If the index of this element is unknown at compile-time, then this processor can only be identified at run-time, depending on the size of the input list and the index of the element. In the following algorithm, it is assumed that every processor holds the value of $\frac{n}{p}$ and that p_no represents the (unique) number of a processor.

```
Algorithm PAR_SELECT (all topologies)
    /* selects the $j^{th}$ element of a distributed list */
BEGIN
    1. FOR every $processor_i$ DO in PARALLEL
           IF $(\frac{n}{p}p\_no \leq j < \frac{n}{p}p\_no + \frac{n}{p})$ THEN
               /* this processor holds the required element*/
               select $(j - \frac{n}{p}p\_no)$
END.
```

The algorithm is very simple. It just accounts for the fact that the original input list is distributed across p processors and computes the offset from the actual value of j, to obtain the index value within a processor. If the index falls within the range of the indices of the elements that a particular processor holds, then it selects the element corresponding to that index. The selected element is left on the processor on which it was found. The cost of the parallel implementation is similar to the sequential one, but since each processor only holds a part of the entire list, the cost is given by:

$$C^p_{select} \propto \frac{n}{p} \tag{6.73}$$

6.2.15 apply_select

The cost of **apply_select** depends on the number of elements of the list to which the argument function f is applied. If this number is assumed to be m, then the sequential cost is given by:

$$\begin{aligned} C^s_{apply\_select} &= mC_f \\ & m \leq n \end{aligned} \tag{6.74}$$

If the value of m is unknown, then it is assumed that $m = n$. In other words, the worst-case cost is assumed.

The parallel implementation for **apply_select** on any topology is described by the following algorithm.

```
Algorithm PAR_APPLY_SELECT (all topologies)
      /* indices to be selected are $i_j, i_k, \ldots, i_m$ */
BEGIN
      1. FOR every processor$_i$ DO in PARALLEL
              FOR every index $b$ DO
                    IF ($\frac{n}{p}p\_no \leq b < \frac{n}{p}p\_no + \frac{n}{p}$) THEN
                        /* element$_b$ $\in$ processor$_i$ */
                        element$_b$ $\longleftarrow$ $f$ element$_b$
END.
```

The worst-case situation occurs if all the elements to which the function f is to be applied are resident on the same processor.

$$\begin{aligned} C^p_{apply\_select} &= mC_f \\ & m \leq \frac{n}{p} \end{aligned} \tag{6.75}$$

6.2.16 copy

copy is a communication construct. The cost of implementing **copy** sequentially follows from its definition and is proportional to the length of the list. It comprises of the list processing costs involved in selecting the appropriate element and then constructing a new list of pairs with the selected element and each of the other list elements. Since the index of the element to be copied may not be known until run-time, a worst-case situation is assumed, and the cost of **copy** is given by:

$$C^s_{copy} \propto 2n. \tag{6.76}$$

```
Algorithm PAR_COPY (all topologies)
    /* copies the $j^{th}$ list element to all other list elements */
BEGIN
        1. Call PAR_SELECT to select the $j^{th}$ element
        2. Processor on which the element was found DO
                Broadcast the element to all processors
        3. Every Processor DO
                Pair up the received element with each list element
END.
```

The costs of steps 1 and 3 involve only list processing costs and are proportional to the length of the list that is resident on each processor, viz. $\frac{n}{p}$. If this cost is represented by C_{lproc} and the cost of broadcasting a single element of the list to all the processors is represented by B_1, then the cost of the parallel implementation of **copy** is given by:

$$C^p_{copy} = C_{lproc} + B_1. \tag{6.77}$$

6.2.17 reverse

The sequential implementation of **reverse** only involves list processing costs incurred in constructing the new (reversed) list. This cost is proportional to the length of the list:

$$C^s_{reverse} \propto n \tag{6.78}$$

The parallel implementation for **reverse** does not always follow its definition. If the entire list is resident on a single processor, then the usual sequential implementation would apply. If the list is distributed across several processors, then the following algorithms are adopted on the different topologies.

6.2.17.1 reverse on the Linear Array

There seems to be no efficient way of reversing a list that is distributed across a linear array. The parallel implementation in this case just follows the definition for **reverse** in terms of **g_fold**. So, the algorithm for **reverse** on the linear array is the same as that for **fold**, and the cost is given by the cost of **g_fold** on the linear array (Equation 6.27), with the sequential function being *rev.* (refer to Section 4.2). The resulting list would be left on processor$_0$.

6.2.17.2 reverse on the Hypercube

```
Algorithm PAR_REVERSE (hypercube)
BEGIN
        1. FOR every processor DO
                 result ⟵ reverse $xs_i$
        2. FOR $i \longleftarrow (d-1)$ DOWNTO 0 DO
                 exchange result with neighbour in dimension $i$
END.
```

The algorithm results in the reversed list being distributed across the processor network in a manner identical to the original list. The cost of reversing the portion of the list on each processor is, again, proportional to the length of the list that is resident on it. If this cost is represented by C_r, then the cost of the parallel implementation of **reverse** is given by:

$$C^h_{reverse} = C_r + 2T^{\frac{n}{p}}_{com} \qquad (6.79)$$

The described algorithm provides a more economical implementation as compared to just gathering the list, reversing it and then re-distributing the reversed list. The number of communications required may not be very different in the two cases, but the amount of data involved in each communication step is much less in the recommended algorithm. Also, since the processors perform the **reverse** operation in parallel on different parts of the list, C_r would be less than $C^s_{reverse}$.

6.2.17.3 reverse on the 2-D Torus/Mesh

As in the case of the linear array, the algorithm for the parallel implementation of **reverse** on the 2-D torus follows its definition in terms of the basic set of recognised functions. The algorithm is the same as that for **fold** on the 2-D torus and the cost is given by Equation 6.33. The result is left on processor$_0$ and an additional scattering cost will be incurred if it needs to be redistributed for the subsequent phase.

6.2.17.4 reverse on the s-ary Tree

It appears to be difficult to design an efficient algorithm for reversing a list of elements distributed across a tree. The algorithm, therefore, simply follows the definition of **reverse** in terms of the basic set of recognised functions. The cost is given by Equation 6.39.

6.3 The 2-D Torus Topology

The cost expressions clearly indicate (not surprisingly!) that most of the recognised functions can be implemented most efficiently on the hypercube topology, with the 2-D torus proving to be the next-best candidate. The tree and the linear array prove to be inefficient and are not studied further. Chapter 7 presents the results of implementing three example programs on the hypercube topology. Although this thesis does not provide a practical implementation on the torus, a possible strategy is discussed in this section.

The hypercube is likely to have a much better performance than the 2-D torus in the context of the implementation strategy, since it is divisible into subcubes, in which the communication latencies are identical to that of the original hypercube. Two problems could arise as a result of dividing the torus into smaller parts:

- Logical neighbours in the smaller tori will not necessarily be physical neighbours. This would lead to an increase in communication costs, since a logical neighbour may no longer be just one hop away.
- Also, the numbering on the nodes in the smaller tori may not be as required for the parallel implementation of the particular phase. A rearrangement of the data may be necessary in order to obtain correct data placement on processors in the smaller tori. Not only would this contribute to the inter-phase data rearrangement costs, but it may also complicate the communication algorithms required.

In view of the problems just discussed, an alternative parallel implementation strategy could be considered for the 2-D torus. A hypercube embedding could be defined on the 2-D torus and the recognised functions could be implemented on this logical hypercube, using the corresponding hypercube algorithms. One example of an embedding of a hypercube in a 2-D torus is described in [Bre83]. In the following discussion, this logical hypercube is referred to as the *embedded hypercube*. The following points should be noted.

1. A hypercube of dimension d with $p = 2^d$ processors would be embedded in a 2-D torus with $p = p_1 p_2$ processors, where $p_1 = 2^{\lfloor \frac{d}{2} \rfloor}$ and $p_2 = 2^{\lceil \frac{d}{2} \rceil}$. p_1 represents the number of processors in each row and p_2 represents the number of processors in each column.

2. Logically neighbouring processors will not necessarily be physical neighbours in the embedded hypercube of dimension greater than four, since each processor in a 2-D torus has only four nearest neighbours. Nearest-neighbour preserving embeddings should be possible for hypercubes with $d \leq 4$.

3. The 2-dimensional hypercube is identical to the torus with four processors, including the numbering on the nodes. It could form the base case for an algorithm which would work by combining two $(d-1)$-dimensional embedded hypercubes, to construct the d-dimensional one.

Although the physical connectivity of the embedded hypercube is not as good as that of the corresponding hypercube, it may still prove to be more cost-effective than the corresponding 2-D torus, especially for small values of d. New data distribution algorithms, such as *scatter*, must be devised for the embedded hypercube and their costs computed. The costs of the recognised functions on the embedded hypercube must be computed, accounting for the additional hops required to communicate with logically neighbouring processors that are not physical neighbours.

A hypercube embedding appears to overcome some of the problems associated with the torus. However, rigorous cost computations both for the algorithms implementing the recognised functions and the communication routines on the embedded hypercube are required before a final conclusion can be reached. Other constraints which would affect the choice are the following:

- The number of processors in the network (p) - For small values of p, it may be more cost-effective to use the embedded hypercube. However, for larger values of p, the increase in the number of hops to communicate between logical neighbours on the embedded hypercube may make it more cost-effective to use the torus itself.

- The nature of the application program - If the phases of the application program comprise only of one recognised function each, or if the selected parallel implementation implements only one recognised function in parallel in a phase, then no sub-division of the topology is required. In such cases, the 2-D torus topology may prove to be more cost-effective than the hypercube embedded in it.

A possible approach would involve computing two sets of costs - one for the 2-D torus itself and another for the hypercube embedded in it - and selecting the more cost-effective implementation for the particular problem.

Chapter 7

The Implementation Scheme and Example Programs

The accuracy of the performance predictions made by analyser for a hypercube topology is studied in this chapter. Three programs are expressed in terms of the recognised functions and input to the analyser. For each program, the analyser determines a cost-effective parallel implementation, with the corresponding code being generated and executed on a hypercube network. The computation times are measured and compared with the predictions of the analyser. As mentioned in Section 3.5.4, a fully-fledged parallel code generator has not been implemented. However, support is available in the form of a library of functions for developing and testing programs. This chapter describes the environment for program development and discusses the results of implementing the three example programs.

The implementations are performed on the Meiko Computing Surface, which is a transputer-based distributed-memory parallel machine. Each transputer has four communication links and represents a single node on the binary hypercube. It is ensured that *logical* neighbours on the hypercube are also *physical* neighbours on the transputer network, which is assumed to be the case in the analyser's cost model. Therefore with only four physical links, it is not possible to simulate a hypercube of dimension greater than four. The communication harness, CSTools

[Mei], does allow for more than four neighbours per processor, but some *logical* neighbours will no longer be *physically* only a hop away. Worse still, it is not possible to predict the number of hops involved in the routing of a message, and indeed this value may vary for different executions of the program. This in turn would make it almost impossible to accurately predict the cost of such communications. Consequently, all the experiments have been performed on a hypercube with a maximum of 16 processors.

7.1 The Analyser

The algorithm for the analyser, as given in Section 5.2.1, has been implemented as a *C* program. The input is an 8-tuple, as described in Section 5.1. The analyser in its current form lacks type-inference and profiling capabilities; therefore, the types of functions and data, and estimates of data sizes have been explicitly provided. The cost expressions for the recognised functions have been coded into the analyser, which constructs the program tree for the application and computes the costs for the different possible implementations. The cost computation builds a search tree from which the least-cost path is selected. The path comprises of a set of nodes, one for each phase in the program. Each node contains the necessary information for its implementation: the number of processors for a particular phase, the nature of the current list distribution (as a result of the previous phase), the type of list distribution required, and which of the recognised functions in the phase are to be implemented in parallel and on how many processors. A communication routine computes the costs associated with inter-phase data rearrangements. It also keeps track of the nature of the input list distribution at any point in the execution of the program. The information contained in the set of nodes returned by the analyser should enable the automatic generation of code for a target parallel machine.

7.2 The Parallel Library

In the prototype implementation, the code for each call to a recognised function is generated manually. Some support is available in the form of a library of functions, which contains the code for the recognised functions on the hypercube topology. This enables the performance figures for the different test examples to be compared on a uniform basis. The code for the parallel implementation of the recognised functions is written in *C*, with CSTools [Mei] being used as the communication harness.

The support available to develop and test programs on the Meiko Computing Surface comprises of the following:

- A *definitions* guide which contains the *C* prototypes for the recognised functions and the various communication routines, and the list data structure on which the recognised functions operate.
- A library of routines containing the code for the implementation of the recognised functions on the hypercube.
- A library of routines which implements the definitions for the communication operations on the hypercube topology.
- A library of utilities containing list manipulation functions such as the creation and copying of lists.

Several optimisations have been adopted in order to reduce the cost of the parallel implementations. These are not program-specific, which ensures uniformity in performance comparisons. The analyser assumes that the following optimisations will be performed, which, in turn, is reflected in the cost estimates.

7.2.1 Memory Allocation

By its very nature, the list structure is dynamic. An implementation based on lists therefore involves a number of list creation and destruction operations, making it important to handle memory allocation efficiently. The implementation scheme allocates space for a list in a single operation, as opposed to allocating space individually for each list element, resulting in a significant reduction in memory allocation overheads. Also, since these costs are not accounted for by the analyser, it helps to bridge any disparity between the predicted and practical results. The current implementation does not incorporate a garbage collector.

This *one-step* memory allocation is possible because of the predefined behaviour of the recognised functions. For every recognised function applied to a list of a particular size, it is possible to compute the size of the resulting list (with the exception of the **filter** function, in which case the worst-case size of output list, i.e. the size of input list, can be assumed).

7.2.2 In-line Expansion of Function Calls

Function calls are expensive on the Computing Surface. Quite often, the cost of calling the function may be much larger than executing it. The model does not account for this cost which could lead to a disparity between the predicted and actual values. This, in turn, could lead to the selection of a less efficient implementation as the optimal one.

The problem could be resolved by designing the analyser such that it accounts for the costs of function calls. However, this introduces a machine-specific detail in the analyser. The problem is overcome by adopting an in-line expansion of the called function. A language such as *C++* would handle such expansions automatically. In its absence, these expansions are done manually.

A potential problem with in-line expansion would arise in the case of recursive functions. If a sequential function is recursively defined, then it has to be first converted into an equivalent iterative one before expansion.

7.2.3 Distribution of Lists among Processors

This is an optimisation that is made with a view to reducing the communication overhead. It involves the case where a list whose D-value ≥ 1 (i.e. a list of lists), is to be scattered across the processor network such that each sublist is distributed across p processors. Such a situation would arise when, in a branch with two nested recognised functions, the innermost one is to be implemented in parallel on p processors.

Let Xss represent a list of lists as follows.

$$\begin{aligned} Xss &= [Xs_1, Xs_2, \ldots, Xs_m] \\ Xs_i &= [x^i_1, x^i_2, \ldots, x^i_n] \\ & \quad 1 \leq i \leq m \end{aligned} \tag{7.1}$$

The size of Xss is $(m \times n)$. For the sake of simplicity in cost computations, it is assumed that $(n \bmod p = 0)$. The recognised function **split** can be used to divide the list, Xs_i, in Equation 7.1, into p parts which results in the following:

$$\begin{aligned} Xs_i &= [[x^i_1, x^i_2, \ldots, x^i_{\frac{n}{p}}] \\ & \quad [x^i_{\frac{n}{p}+1}, \ldots, x^i_{2\frac{n}{p}}] \\ & \quad \vdots \\ & \quad [x^i_{(n-1)\frac{n}{p}+1}, \ldots, x^i_n]] \\ &\equiv [Y^i_1, Y^i_2, \ldots, Y^i_p] \\ & \quad 1 \leq i \leq m \end{aligned}$$

In order to scatter each Xs_i across the p processors, the operation that needs to be performed on Xss is: **map** *scatter* Xss.

If d represents the dimension of the hypercube, from Equation 6.8 it can be deduced that the cost of distributing each sublist of the list of lists Xss is given by:

$$C_1 = m(dK_0 + \frac{1}{K_1}\frac{n}{p}(p-1)) \tag{7.2}$$

However, a more cost-effective distribution is possible and is discussed below. Consider the following definitions.

re_order p xss = **g_fold** *append* [ ] $\circ$ *shuffle* $\circ$ **map** (**split** p) xss

where,

shuffle ([ ]::ys) = [ ]

shuffle yss = (**map** *head* yss) :: *shuffle* (**map** *tail* yss)

The functions *head* and *tail* obtain the head and tail of a list, respectively. Now consider the function *re_order*, as applied to the list Xss defined in Equation 7.1.

If the result of the first stage of the **composition**, viz. (**map** (**split** p) Xss), is represented by Yss, then

$$\begin{aligned} Yss &= [[Y_1^1, Y_2^1, \ldots, Y_p^1], \\ &\quad [Y_1^2, Y_2^2, \ldots, Y_p^2], \\ &\quad \vdots \\ &\quad [Y_1^m, Y_2^m, \ldots, Y_p^m]] \\ Y_i^j &= [x^j_{(i-1)\frac{n}{p}+1}, x^j_{(i-1)\frac{n}{p}+2}, \ldots, x^j_{i\frac{n}{p}}] \\ &\quad 1 \le j \le m \\ &\quad 1 \le i \le p \end{aligned} \tag{7.3}$$

The size of Yss is given by $(m \times p \times \frac{n}{p})$. Next, the second stage of the composition is applied to the result of the first stage. Let Zss = shuffle Yss.

$$\begin{aligned} Zss &= [[Y_1^1, Y_1^2, \ldots, Y_1^m], \\ &\quad [Y_2^1, Y_2^2, \ldots, Y_2^m], \\ &\quad \vdots \\ &\quad [Y_p^1, Y_p^2, \ldots, Y_p^m]] \end{aligned} \tag{7.4}$$

The function *shuffle* effectively produces the transpose of its input list. The size of Zss is given by $(p \times m \times \frac{n}{p})$. Applying the final stage of the composition, viz. **g_fold** *append* [ ], to flatten the list, the result of *re_order* is as follows:

$$Rss = [Y_1^1, Y_1^2, \ldots Y_1^m, \ldots, Y_p^1, Y_p^2, \ldots, Y_p^m]. \tag{7.5}$$

The size of Rss is given by $(mp \times \frac{n}{p})$. Once the list Xss is transformed into a form as given by Rss in Equation 7.5, a *scatter* operation on Rss would place the elements on the correct processors. This method of distributing the list amounts to performing the following operation:

scatter $\circ$ (*re_order* p) Xss.

A *scatter* operation on a list of size $(a \times b)$ would involve the distribution of $a(b+1)$ list nodes. (For each sublist, there are b nodes representing each element of the sublist and one node that represents the entire sublist. There are a such sublists, hence the total number of nodes is $a(b+1)$). Since the size of Rss is $(mp \times \frac{n}{p})$, the total number of list nodes to be scattered is $(mp(\frac{n}{p}+1))$.

Again, referring to Equation 6.8, the cost of the entire list distribution operation is deduced to be:

$$C_2 = dK_0 + \frac{1}{K_1}\frac{mp(\frac{n}{p}+1)}{p}(p-1) + C_{re\_order}. \tag{7.6}$$

C_{re_order} represents the cost of the function *re_order* and essentially comprises of list processing costs.

Consider the following equation (Equation 7.2 - Equation 7.6):

$$C = dK_0(m-1) - \frac{1}{K_1}(p-1)m - C_{re\_order}.$$

$C > 0$ implies that:

$$1 + dK_0K_1\frac{m-1}{m} - \frac{K_1}{m}C_{re\_order} > p. \tag{7.7}$$

The cost of the *re_order* function could be computed by the analyser from its estimates of the list processing costs and the sizes of the input and output lists. If the left-hand-side of Equation 7.7 is greater than p, then the second option could be selected for distributing the input list across the p processors.

In practice, the costs of list processing will probably be much less than the communication costs and the second option can be safely selected as the more cost-effective one. As an example, the cost of processing a single list element is approximately $\frac{1}{100}K_0$ on the Computing Surface (by experiment). In the current implementation of the analyser, the second method of list distribution is assumed by the analyser in computing data distribution costs.

7.2.4 Parallel Implementations of Recognised Functions Involving Communication

This optimisation is used in the implementation of those recognised functions that occur in situations satisfying the following conditions:

1. The function is an argument of the recognised function **map**
2. The function is being implemented in parallel on p processors
3. The computation of the function in parallel on the p processors involves communication.

Some possible examples of such functions are **fold**, **scan** and **copy**.

Let $\mathcal{R}$ be a recognised function that occurs in such a situation. Typically, the distributed implementation for $\mathcal{R}$, represented by $\mathcal{R}_{dis}$, can be defined in the following manner.

$\mathcal{R}_{dis} = \mathcal{U} \circ \mathcal{C} \circ \mathcal{R}_{seq}$

where,

$\mathcal{R}_{seq}$ is the function describing the sequential implementation for $\mathcal{R}$

$\mathcal{C}$ is a communication function that involves the communication of partial results to neighbouring processors

$\mathcal{U}$ is a function that updates a processor's local partial result with the one that is received from a neighbour

According to condition 1, $\mathcal{R}$ must be an argument of **map**. Consider the following equation:

$$\text{map}\ \mathcal{R}_{dis} \equiv \text{map}\ (\mathcal{U} \circ \mathcal{C} \circ \mathcal{R}_{seq}). \tag{7.8}$$

Using the **map**-promotion rule, 7.8 can be expanded to:

$$\text{map}\ \mathcal{R}_{dis} \equiv \text{map}\ \mathcal{U} \circ \text{map}\ \mathcal{C} \circ \text{map}\ \mathcal{R}_{seq}. \tag{7.9}$$

The expression in Equation 7.9 has less communication overhead associated with it, in comparison to the one in Equation7.8. The implementation corresponding to Equation 7.8 involves the application of the sequential definition of the function, followed by the communication of the partial result for *each* element of the list in sequence. This implies that the communication start-up cost is multiplied by the length of the list. Also, the entire bandwidth of the channel will probably not be utilised, since only one list element is communicated at each step. In Equation 7.9, the sequential definition of the function is first applied to all the elements of the list resident on the particular processor. The resulting partial values are then communicated in one step. The communication start-up cost is incurred only once and the bandwidth of the communication channel is used more effectively. Both of these factors lead to a significant reduction in the communication cost if the implementation corresponding to Equation 7.9 is chosen.

7.3 Communication on the Hypercube

The algorithms for the different types of communication on the hypercube topology, as described in [SS89], have been implemented on the Meiko Computing Surface. In the communication of list data structures between processors, the one that *sends* the list must convert pointer addresses to relative ones and the one that *receives* the list must compute the real addresses from the offsets. These operations are necessary to obtain correct pointer references across processor boundaries. This effectively increases the communication overhead and is incurred in the form of list processing costs. The analyser accounts for these costs in the same way as it does for other list processing overheads, since the data to be communicated at any step in the program is predetermined.

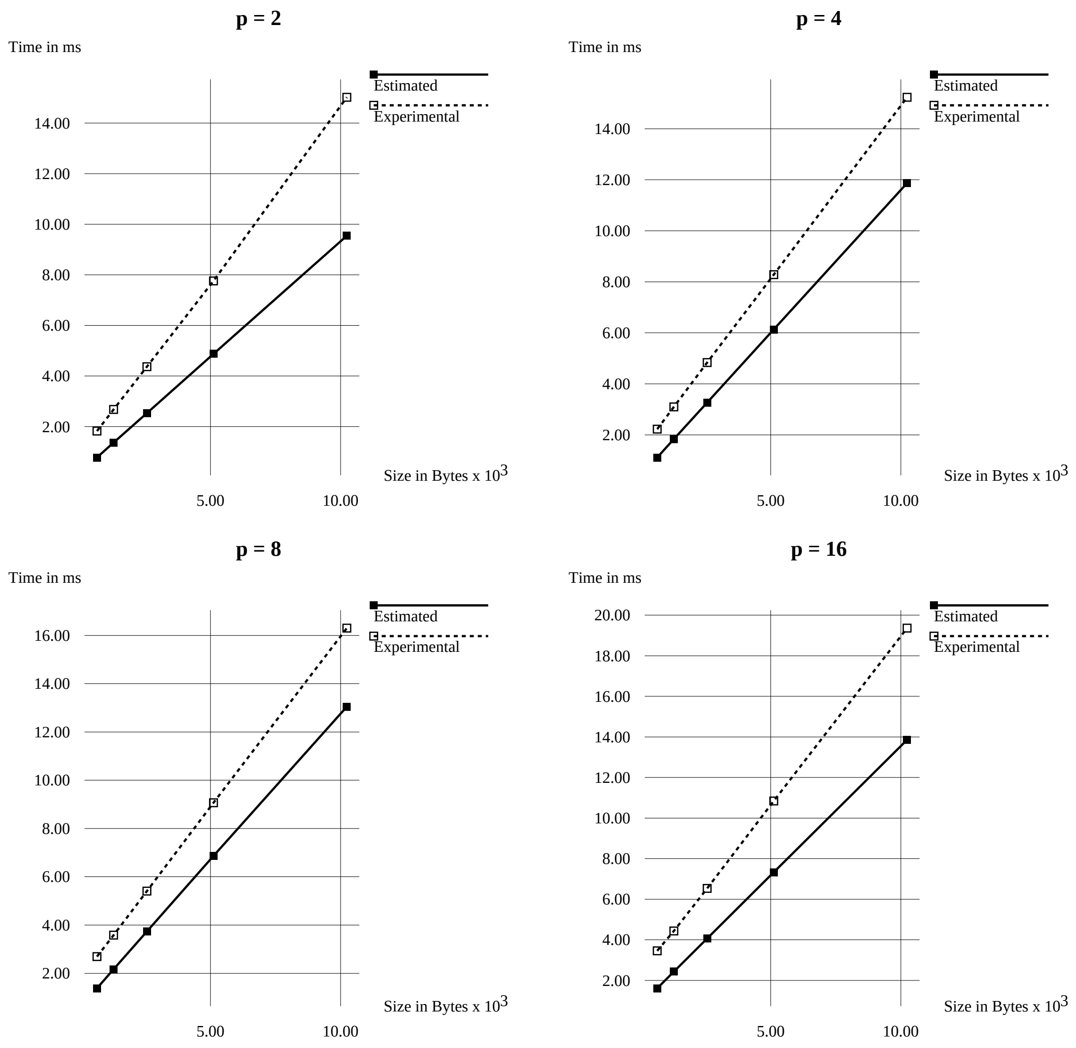
Graphs depicting the predicted and experimental results for some of the communication routines (ones which are used in the examples) on the hypercube topology are shown in Figures 7.1 and 7.2, respectively. The following points regarding the method used to obtain the graphs may be noted.

- Each processor executes an *initialisation* routine in which it opens its communication links and identifies its neighbours. Due to the lack of global synchronisation on a distributed-memory machine like the Computing Surface, it is unlikely that all the processors would operate in lock-step. This means that all the processors may not complete their initialisation and start on their communication routines at the same time. This implies that data transfers which are assumed to overlap, may not do so in a practical implementation. Consequently, the predicted communication costs may not always be obtained experimentally. In order to minimise this discrepancy, the experimental costs of communication routines are averaged over a large number of runs.

- The *scatter* operation is the dual of the *gather* operation. i.e. the *gather* algorithm is obtained by reversing the steps of the *scatter* algorithm. The costs of the two routines are, therefore, identical and have been plotted as a single graph. They are determined by performing a number of *scatter-gather* operations and halving the average cost. In the case of the *broadcast* operation, the processor that receives the data last, performs a broadcast in the reverse direction, so that the original broadcaster is the last one to receive it. This introduces a form of synchronisation and the cost of the *broadcast* operation is half the cost, averaged over a number of runs.

A study of the graphs reveals that the experimental curves follow closely with the theoretical predictions. However, as the size of the data to be communicated increases, the experimental curves seem to diverge from the predicted ones. A possible explanation for this feature may lie in the manner in which the Computing Surface handles communication. Beyond a particular size, the data may have to be divided into smaller-sized chunks and transferred in more than one step. This would increase the communication cost, which is not predicted by the model. Also, with increase in the number of processors, the match between the predicted and observed behaviour appears to improve, more so in the broadcast routine. A similar feature is to be expected in programs using these routines.

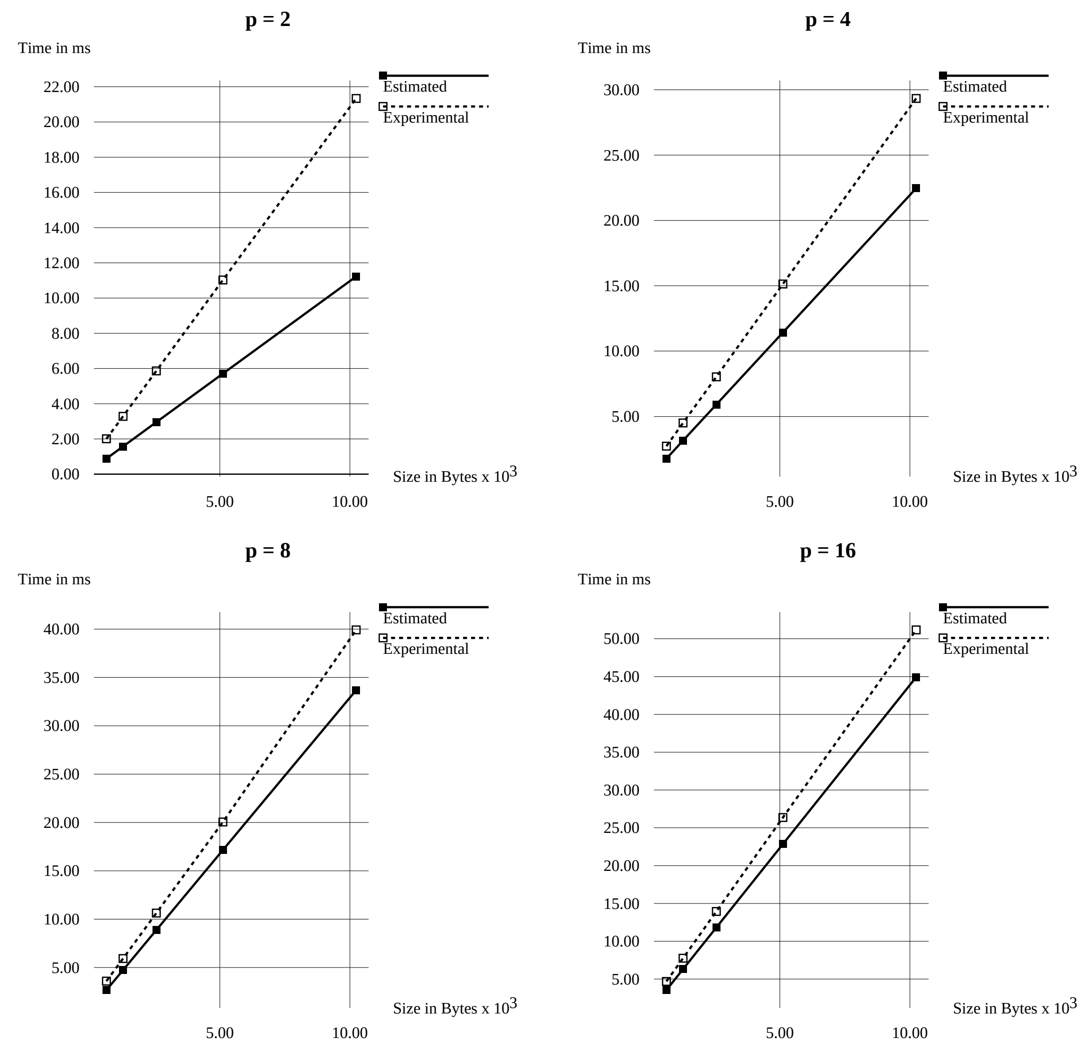
Figure 7.1: Scatter/Gather on a Hypercube



7.4 Example Programs

The three programs chosen for implementation are *Matrix Multiplication*, *Merge Sort*, and *Jacobi Iteration*, as discussed in Chapter 4. They are well-known problems in parallel programming and illustrate the use of different recognised functions in the HOPP model. The chosen programs are also intended to highlight different features of the analyser. The program for Matrix Multiplication is the straightforward case where the costs of all sequential functions are independent of input size, and where the analyser is capable of making correct inferences

Figure 7.2: Broadcast on a Hypercube



regarding problem size. The program for Merge Sort tests the performance of the analyser in the case where the cost of a sequential function (*merge*) is not a constant, but proportional to the sizes of its input lists. The program for Jacobi Iteration considers the performance of the analyser in a situation where it is incapable of making correct inferences regarding problem size. This, in turn, serves to demonstrate some of the limitations of the analyser in its current form.

For each of the three example programs, parallel implementations are considered and the corresponding costs as computed by the analyser are examined.

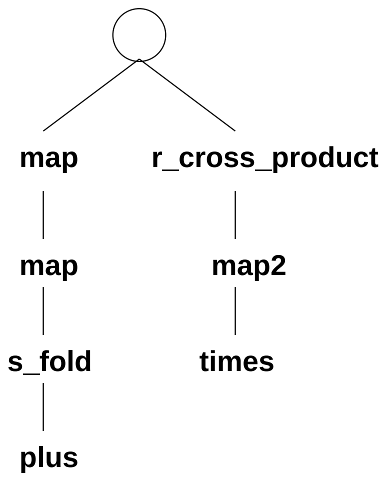


Figure 7.3: The Program Tree for Matrix Multiplication

The results of implementing the most efficient choice of the analyser on the Meiko Computing Surface are presented. The calls to the recognised functions and the communication routines are generated by hand, and the C-code in the parallel library is used. The *second-best* choice is also implemented, to add confidence in the performance of the analyser. The parallel implementations are considered on hypercubes with 2,4,8 and 16 processors, respectively.

The results of implementing the matrix multiplication and merge sort programs which are discussed next, can also be found in [Ran95].

7.4.1 Matrix Multiplication

This section discusses the well-known problem of multiplying two matrices $A_{m\times n}$ and $B_{n\times k}$, resulting in the matrix $C_{m\times k}$. The code for performing the multiplication is discussed in Section 4.3.1. Each matrix is represented as list of lists.

7.4.1.1 The Analysis

The program tree constructed by the analyser is shown in Figure 7.3. The root node represents **composition**. In *phase one*, the analyser computes the costs for a sequential implementation and three parallel implementations, corresponding to the two recognised functions in the phase, viz. **r_cross_product** and **map2**. On a p-processor hypercube, if **r_cross_product** is implemented on p_1^1 processors and **map2** is implemented on p_2^1 processors, ($p_1^1 \leq p$, $p_2^1 \leq p$ and $p_1^1 p_2^1 = p$), then the cost of *phase one* can be deduced as follows: (see Sections 6.2.6, 6.2.8)

$$C_{p1} = C_{com} + \lceil \frac{m}{p_1^1} \rceil \lceil \frac{n}{p_2^1} \rceil k C_{times}.$$

C_{com} represents the costs incurred in distributing the initial input lists across the p processors. C_{times} represents the cost of the sequential function *times*. It should be noted that the result of *phase one* is a list of lists of size $(m \times n \times k)$. *Phase two* contains three recognised functions. The analysis results in the costs for a sequential implementation and seven different parallel ones. If the outer **map** is implemented on p_1^2 processors, the inner **map** on p_2^2 processors and the **fold** is implemented on p_3^2 processors, respectively ($p_1^2 \leq p$, $p_2^2 \leq p$, $p_3^2 \leq p$ and $p_1^2 p_2^2 p_3^2 = p$), then the cost of *phase two* can be deduced to be as follows: (also refer to Sections 6.2.1, 6.2.2.2)

$$C_{p2} = R_{com} + \lceil \frac{m}{p_1^2} \rceil \lceil \frac{n}{p_2^2} \rceil C_{plus}(\lceil \frac{k}{p_3^2} \rceil - 1) + d(T_{com} + C_{plus}).$$

R_{com} represents the cost incurred in re-arranging the results of *phase one* to suit the implementation of *phase two*, $(d = \log_2 p_3^2)$, T_{com} is the cost of communicating the partial result of **fold** on one processor to a neighbouring one.

Two sets of matrices, $A^1_{32 \times 32}$, $B^1_{32 \times 32}$ and $A^2_{64 \times 32}$, $B^2_{32 \times 16}$, are considered for analysis. In both the cases, the following implementations are selected as the *best* (i.e. least-cost) and *second-best* parallel implementations, respectively, for $p = 2, 4, 8, 16$.

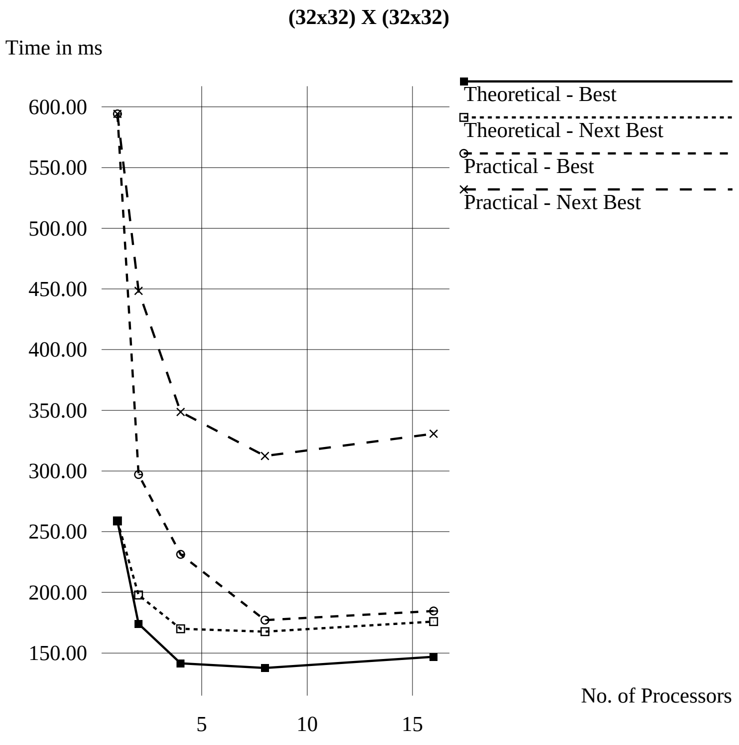
1. *best* - for the two phases, $p_1^1 = p$, $p_2^1 = 1$ and $p_1^2 = p$, $p_2^2 = 1$. This corresponds to parallel implementations for **r_cross_product** and the outer **map** in the first and second phases, respectively.

2. *second-best* - for $p = 2$, it is clear that only one function in a phase can be implemented in parallel. For the *second-best* implementation, **map2** and **fold** are implemented in parallel in the first and second phases, respectively. For $p = 4, 8, 16$, the following implementation was selected as the *second-best*. In *phase one*, $p_1^1 = 2$, $p_2^1 = \frac{p}{p_1^1}$. In *phase two*, $p_1^2 = 2$, $p_2^2 = 1$ and $p_3^2 = \frac{p}{p_1^2}$. This corresponds to a parallel implementation for **r_cross_product** on p_1^1 processors and **map2** on p_2^1 processors in the first phase. In the second phase, the outer **map** and the **fold** are implemented in parallel on p_1^2 and p_3^2 processors, respectively. The inner **map** is implemented sequentially.

7.4.1.2 The Results

The theoretically predicted values are plotted along with the experimental ones in Figure 7.4, for the two chosen implementations. In each of the graphs, the *trend* predicted by the analyser is also obtained experimentally. The experimental costs are greater than the predicted ones. This is because the analyser does not account for low-level costs which are incurred by a practical implementation. For both sets of matrices, the number of processors that is predicted to be the optimum, is confirmed experimentally. The experimental curves tend to have much steeper gradients compared to the predicted ones, especially in the case of smaller numbers of processors. This feature is less easy to explain. A probable explanation may lie in the observed behaviour of the communication routines (refer to Figure 7.1, Figure 7.2), where the match between predicted and observed behaviour improves with an increase in the number of processors.

It may be noted that the theoretical and experimental curves in implementation 1 are much closer than the corresponding ones in implementation 2. The sequential code that is executed in both the cases is identical. However, implementation 2 involves more communication than implementation 1, both for initial



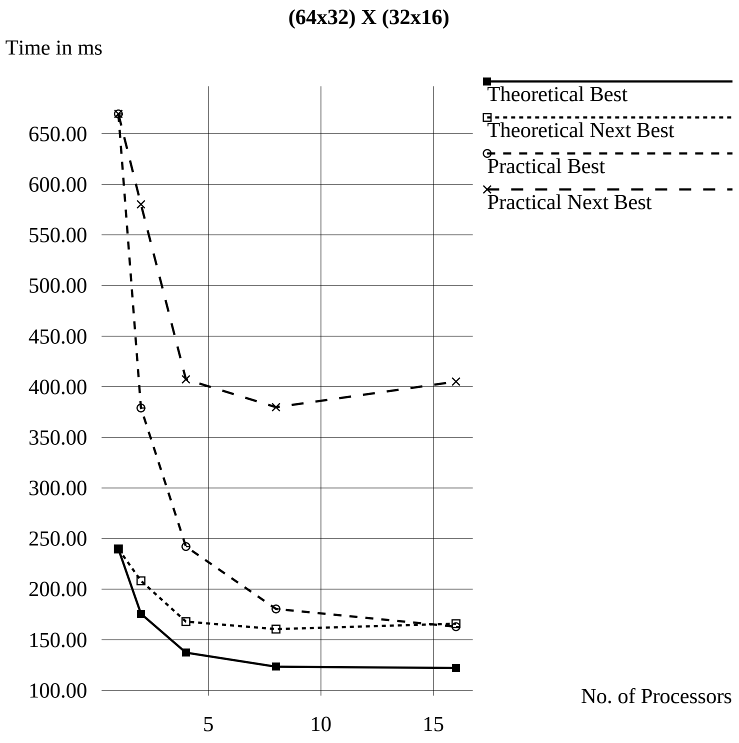


Figure 7.4: The Results for Matrix Multiplication

data distribution and for the parallel implementation of **s_fold** in *phase two*. Implementation 1 involves a *scatter* and a *broadcast* operation on matrices A and B, respectively, at the beginning of *phase one*. There is no communication in *phase two*. *Phase one* of implementation 2 involves a *scatter* followed by a *broadcast* operation on matrices A and B, respectively, to place them on p_1^1 processors for the parallel implementation of **r_cross_product**. Two further *scatter* operations are required to distribute each of these on p_2^1 processors, for the parallel implementation of **map2**. More communication is required in *phase two* for processors to communicate the partial results of **s_fold**, since it is implemented in parallel. There seems to be a greater discrepancy between the predicted and experimental results when an implementation involves more communication. This is in a large part due to the lack of global synchronisation between processors.

The costs for communication routines assumed overlapping data paths which may not always happen in practice. This effect gets more pronounced with increase in communication, with the processors getting more staggered and leading to a greater discrepancy between predicted and observed behaviour. Care was taken in the cases of the communication routines such as *scatter* and *broadcast*, to minimise these discrepancies by averaging the costs over several executions and forcing some synchronisation by performing the dual of the corresponding routine and considering only half the total cost. However, in the implementation of a program, the corresponding routine is executed only once and this leads to a staggered operation of processors which are otherwise assumed to operate in lock-step. The discrepancy observed in the performance of the individual communication routines is also probably a contributing factor.

7.4.2 Merge Sort

The code for the program is discussed in Section 4.3.2. For the sake of simplicity, the number of elements in the input list is assumed to be a power of 2 and is represented by n.

7.4.2.1 The Analysis

Each of the three phases have only one recognised function. Hence, only two implementations are possible for each phase - a parallel implementation and a sequential one. The cost of the function *msort* is a constant, but the cost of the function *merge* is proportional to the sizes of the two lists which are being merged. In this case, as discussed in Section 5.1.4, the scheme allows the user to specify a cost function for the sequential function, *merge*. The analyser uses this function to estimate implementation costs.

The analyser first constructs the program tree. For *phase one*, the cost of **split** is given by $S_{\frac{n}{p},\frac{n}{2p}}$, (refer to Section 6.2.10). The cost for the sequential implementation is obtained by setting p to 1. At the end of *phase one*, each processor contains a list of lists of size $(\frac{n}{2p} \times 2)$. *Phase two* only involves a **map** and its cost is given by $\frac{n}{2p}C_{ms}$, where C_{ms} represents the cost of the function *msort*. In *phase three*, after each *merge* operation, the size of the resulting list is the sum of the sizes of the two lists being merged. As the **fold** progresses, the size of the intermediate result increases. After each processor executes the **fold** locally, the number of elements on each processor is $\frac{n}{p}$. In the first step of the combination of partial results, two lists each of size $\frac{n}{p}$ are merged to produce a list of size $\frac{2n}{p}$. The second step involves the merging of two lists each of size $\frac{2n}{p}$ to produce a list of size $\frac{4n}{p}$, and so on. Let the cost of merging two lists of sizes m and n be represented by $f(m,n)$. Using the formula for the cost of **g_fold** in

Section 6.2.2.2, the cost of *phase three* is as follows :

$$\sum_{i=1}^{\frac{n}{2p}-1} f(2i,2) + \sum_{i=0}^{d-1}(T_{com}^{2^i\frac{n}{p}} + f(2^i\frac{n}{p}, 2^i\frac{n}{p})).$$

In order that the analyser accounts for the increasing size of the emerging result while computing communications costs, the user is required to specify the **fold** operation using **g_fold**.

Two lists of sizes 512 and 1024 elements are considered for analysis. In both the cases, the following implementations are selected as the *best* (i.e. least-cost) and *second-best* implementations respectively, for $p = 2, 4, 8, 16$.

1. *best* - implement each recognised function in parallel on p processors for all the phases.

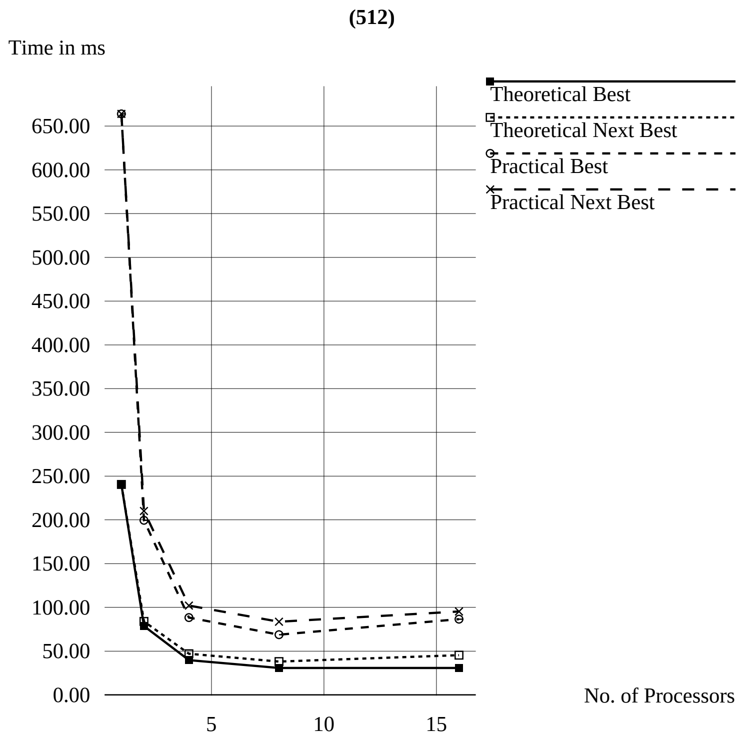
2. *second-best* - in *phase one* implement the function **split** sequentially. Then *scatter* the result to implement the two recognised functions in the next two phases in parallel on p processors.

7.4.2.2 <u>The Results</u>

The results for merge sort are plotted in Figure 7.5. Once again the *trend* predicted by the analyser is obtained experimentally.

For the input list of size 512, the optimal number of processors is predicted to be 8 for both implementations and this is confirmed experimentally. For the list of size 1024, the analyser predicted a more cost-effective implementation with 16 processors, which is again confirmed experimentally.

In the case of merge sort, fewer communications are involved in the initial distribution of data, as compared to both implementations of matrix multiplication. The initial data distribution comprises of a single *scatter* routine, either at the beginning of *phase one* or *phase two*, depending on whether the *best* or *second-best* implementations are considered. This probably causes a smaller stagger in the



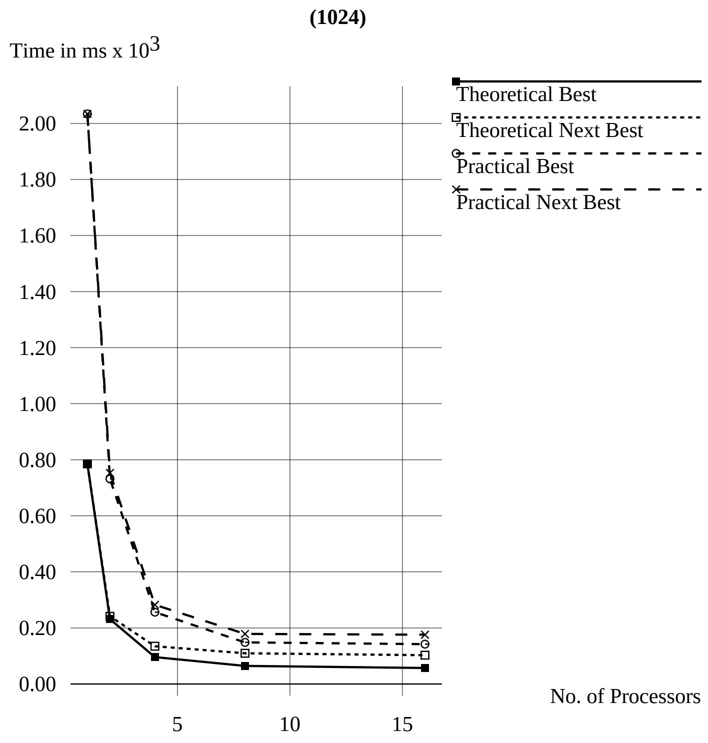


Figure 7.5: The Results for Merge Sort

processors. Also, the parallel implementation of **g_fold** in the last phase forces some form of synchronisation between the processors, since the partial results are communicated to processor$_0$, which computes the time of execution for the program. Both of these factors seem to produce a closer match between the predicted and experimental results in the case of merge sort. However, the gradients of the curves in a practical implementation are still steeper than those of the predicted curves and the same explanation as given in the case of matrix multiplication applies.

7.4.3 Jacobi Iteration

The code for the program has been discussed in Section 4.3.3. As explained in Section 5.1.4, the analyser in its current form is incapable of inferring that the function *rearrange*, which is the argument of **map** in *phase two* of the function *jac*, produces an output list whose size is different from the size of its input list. Consequently, the costs computed by the analyser for *phase three* will be inaccurate. Also, the function *update* in *phase three* changes the D-value of its argument list. This again, cannot be determined by the analyser. The change in D-value will not influence the cost of the function that tests for convergence. This is because the function *test* assumes that the output of the function that operates on the input list at each iteration, is of the same size as its input list. This assumption is justified because the analyser assumes that the cost of iteration$_i$ is the same as the cost of iteration$_{i-1}$. The inability of the analyser to detect the change in the D-value of the output of *update* will only influence the cost of the subsequent iteration. However, in the case of iterative functions, since the cost of a single iteration is considered for the purposes of determining a cost-effective parallel implementation, this will not lead to inaccuracies in cost predictions for this particular program.

7.4.3.1 The Analysis

The input matrix is represented by a list of lists of size $(m \times n)$. The analyser first constructs the program tree. The first and second phases have only one recognised function each, while *phase three* has two recognised functions. Consequently, there is only one possible parallel implementation for the first and second phases. For *phase three*, there are three possible parallel implementations. The costs for the first and second phases are straightforward, being the costs of **get_neigh** and **map**, respectively. Again, the cost of the function *rearrange* in *phase two*, is a function of the size of its input list and the analyser determines its cost from the specification of a cost function for *rearrange*. Consider the costs for *phase one* and *phase two*, represented by C_{p1} and C_{p2}, respectively. Referring to Equations 6.66 and 6.22, the costs are given by:

$$\begin{aligned} C_{p1} &= C_{com} + T^2_{com} 2d + G_{\frac{m}{p}} \\ C_{p2} &= R^1_{com} + \lceil \frac{m}{p} \rceil C_{rearr} \end{aligned} \tag{7.10}$$

where, C_{com} is the communication cost incurred in scattering the input list across p processors, T^2_{com} represents the cost of communicating two sublists each of size n to a neighbouring processor, $d = \log_2 p$, C_{rearr} represents the cost function specifying the cost of the function *rearrange*, and R^1_{com} is the communication cost incurred in rearranging the output list of *phase one*, to suit the implementation of *phase two*. It should be noted that $R^1_{com} = 0$, if both *phase one* and *phase two* are implemented in parallel on p processors, since each of the phases has only one recognised function. The result of *phase one* is a list of size $(m \times 3 \times n)$. The function *rearrange* in *phase two*, transforms an input list of size $(3 \times n)$ into a list of size $(n \times 5)$. So, the output of *phase two* is a list of size $(m \times n \times 5)$. However, the analyser cannot deduce this change in the size of the output list and instead assumes it to still be $(m \times 3 \times n)$, since the recognised function in *phase two* is

map, which is not assumed to change the size of its input list. Hence, the cost computed by the analyser for *phase three* will be inaccurate. If the outer **map2** is implemented on p_1 processors and the inner **map2** on p_2 processors, ($p_1p_2 = p$), the cost of *phase three* is as follows: (refer to Section 6.2.8)

$$C_{p3} = R^2_{com} + \lceil \frac{m}{p_1} \rceil \lceil \frac{n}{p_2} \rceil C_{update} \tag{7.11}$$

where, R^2_{com} again represents the cost incurred in rearranging the output of *phase two* to suit the implementation of *phase three*. However, the analyser will compute a different cost (since the list size it deduces is different), and probably select an implementation that is not the most cost-effective one.

The function *test* (see Section 4.3.3) comprises of four phases, and its cost is the sum of the costs of these phases. The first three phases in the function *test* could be implemented in parallel, since each of them has instances of recognised functions. The cost of *test* has to be added to the costs C_{p1}, C_{p2} and C_{p3} to obtain the total cost of **iterate_cond**.

As mentioned in Section 6.2.9, the cost of only one iteration is considered for the purposes of determining a cost-effective parallel implementation. In an actual implementation, this is the average cost of an iteration.

7.4.3.2 The Results

Two lists of sizes (32×32) and (64×64), respectively, are considered for analysis. In each case, four costs are considered:

- The costs of the *best* and *second-best* implementations, as predicted by the analyser, which will not be accurate.

- The correct costs for the *best* and *second-best* implementations. These are computed manually, in order to determine the percentage error in the costs computed by the analyser.

In each case, the following implementations are selected as the *best* and *second-best* implementations, respectively.

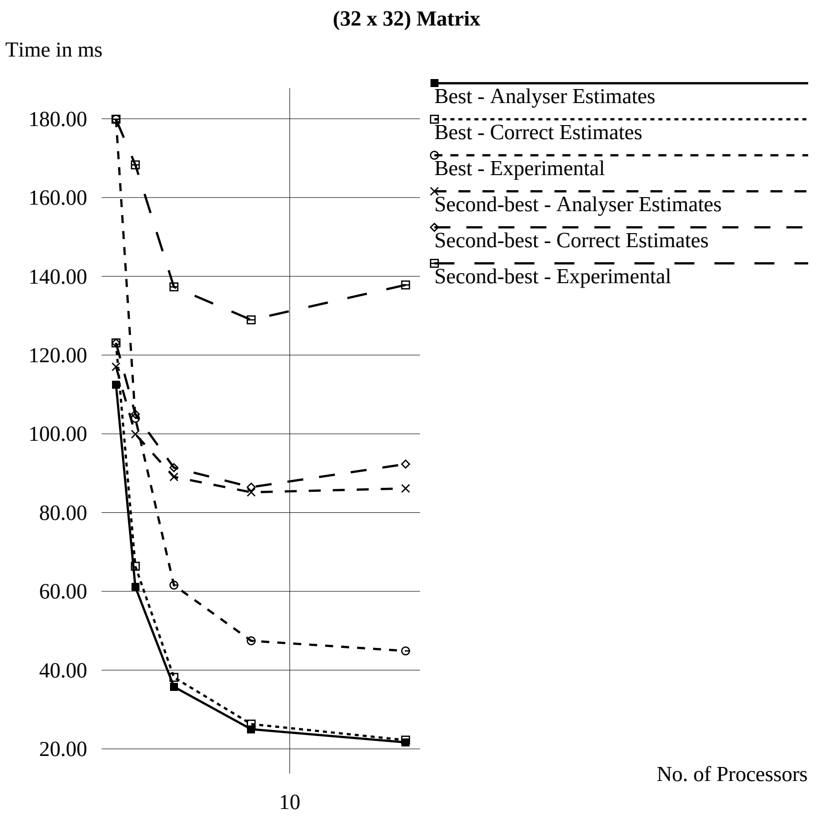
1. *best* - implement **get_neigh**, **map** and the outer **map2** in parallel on p processors for the three phases. Implement the first three phases of the function *test* in parallel, implementing the outer **map2** in *phase one*, and the **map** and **s_fold** in phases *two* and *three*, respectively, in parallel on p processors. The function *less_than* is implemented sequentially. Note that the result of **s_fold** in *phase three* of *test* would leave its result on processor$_0$ and the function *less_than* can then be implemented sequentially on processor$_0$.

2. *second-best* - implement **get_neigh**, **map** and the outer **map2** in parallel on p processors for the three phases. Then *gather* the results and implement the function *test* on a single processor (processor$_0$). As noted in Section 6.2.9, only a *copy* of the result is gathered back to processor$_0$ for evaluating the function *test*. Each processor still possesses its copy of the result to serve as the data for a possible next iteration.

It is to be noted here, that in both implementations the result of the function *test* is left on processor$_0$. Processor$_0$ must *broadcast* this result to all the other processors, so that they can determine whether the iteration is to be terminated. This *broadcast* is performed after every step of the iteration and the analyser includes its cost in the computation of the cost for the program. The cost of an iteration is, therefore, given by (refer to Equations 7.10, 7.11):

$$C_{jac} = C_{p1} + C_{p2} + C_{p3} + C_{test} + C_{broad} \tag{7.12}$$

where, C_{test} is the cost of the function *test*, and would depend on whether a sequential or parallel implementation is selected for it. C_{broad} is the cost of broadcasting the result of the function *test* to all the processors in the network.



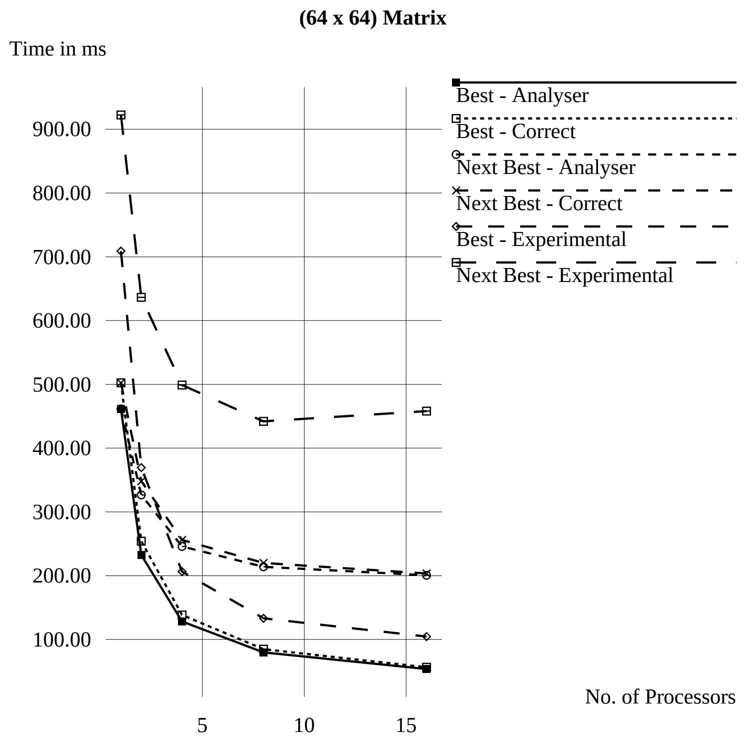


Figure 7.6: The Results for Jacobi Iteration

The three sets of costs for the *best* and *second-best* implementations are plotted in Figure 7.6. Again, the predicted *trend* is also obtained experimentally. In the case of the (32×32) matrix, for the *best* implementation a speed-up is obtained with 16 processors, but for the *second-best* implementation, the optimal number of processors is predicted to be 8, and this is also obtained experimentally. The analyser under-estimates the costs slightly, but still (in this case), the most cost-effective implementation is the one that is selected. (This has been verified by computing the costs manually). The similar feature of the gradient of the experimental curves being steeper than that of the predicted ones is also observed in this example. The distance between the predicted and experimental curves in the *second-best* implementation is larger than that between the corresponding curves in the *best* implementation. This again, is probably because the *second-best* implementation involves a *gather* operation before the function *test* can be implemented and this involves more communication than in the case of the *best* implementation. The distance between the corresponding curves in this example is much larger than that in the *Merge Sort* example. This again is probably due to the fact that the *Jacobi Iteration* example involves more communication than the *Merge Sort* example.

7.5 Conclusions

The performance of three programs expressed in terms of the recognised functions has been studied on a hypercube topology. The analyser computed the most cost-effective implementation for each of the programs. These, along with the *second-best* parallel implementations have been executed on the hypercube, and the predictions made by the analyser have been confirmed experimentally. The three programs are *regular* in the context of the HOPP model, and it is, therefore, possible to express them in terms of the existing constructs. It may not be possible to effectively parallelise irregular problems such as quicksort that depend on the actual input data values, by using this scheme. This is a limitation of the scheme and arises due to the use of compile-time techniques to predict program performance, as well as the restrictive nature of the recognised functions.

Chapter 8

Conclusions and Directions for Future Research

8.1 Thesis Summary

A model of parallel computation (HOPP) based on the functional style and the Bird-Meertens Formalism has been investigated in this thesis. In particular, the thesis has focussed on developing a realistic cost model for HOPP. The model has been targeted at distributed-memory MIMD machines in which the costs of interprocessor communications tend to be significant. Consequently, sufficient effort has been directed at minimising these costs. The motivations for choosing BMF as the basis for the model of parallel computation lie in the advantages it offers and includes the following:

- It provides a high level of abstraction, therefore relieving the programmer of handling the low-level details involved in parallel programming.
- Only one program is written irrespective of the target architecture, which enhances the portability of the program.
- The model comprises of a fixed repertoire of constructs, most of which are implicitly parallel. Any potential parallelism in the program can only be expressed through these constructs. It should be possible to predict the cost

of execution of such a program on a given architecture. Also, once such an estimate has been obtained for a given architecture, it can be re-used for different programs expressed in terms of these constructs.

- The set of implicitly parallel constructs are either already defined, or are easily definable in most functional languages, thereby removing the need for learning any new programming technique in order to use the model.

The implicitly parallel constructs are higher-order functions which are based on the functions in BMF. The basic set of higher-order functions in BMF has been extended by incorporating some additional useful functions. These additional functions are not *new*, but are well-known functions for which definitions in terms of the functions in BMF have been provided, along with cost-effective parallel implementations on four target topologies. This extended set of implicitly parallel functions is referred to as the set of *recognised* functions. A program is expressed in terms of these constructs, typically as a composition of instances of recognised functions. For each recognised function, a parallel implementation is defined on a given target parallel machine topology, and the cost corresponding to that parallel implementation is derived. Associated with each target topology is a cost model which describes the possible methods for implementing a given program in parallel, together with the corresponding cost estimates. A cost analysis is performed on the program, using the analytical cost model that has been developed for the chosen target architecture. The analyser considers the costs associated with the various possible implementations and selects the one that results in the least cost. Parallel code can then be generated for the selected implementation, which, in turn, can be executed on the parallel machine.

The performance of the recognised functions has been studied on four target architectures, viz. hypercube, 2-D torus, tree and linear array. The performance of a majority of these functions was found to be most efficient on the hypercube and hence an analyser has been implemented for this topology. Three example programs were analysed and the chosen parallel implementations were executed on a transputer-based machine configured as a hypercube. In all the cases, the predictions of the analyser have been confirmed by the actual implementations. The 2-D torus follows the hypercube as the one on which the functions can be efficiently implemented. However, due to the poor connectivity of the torus, several possible parallel implementations for a program would probably be inefficient. A hypercube embedding in the torus was proposed as a possible approach in overcoming this inefficiency. A possibility then would be to compute two sets of costs associated with implementing a program on the torus:

- With the torus configured as a hypercube - this would use the defined hypercube embedding and analyse the cost of the program with respect to the embedded hypercube topology.

- With the torus retained as such - this would analyse the cost of the program by considering the possible implementations on the torus itself.

The least-cost implementation could then be chosen.

The tree and the linear array proved to be quite inefficient and were not investigated further. It is possible to develop cost models for other target topologies, thereby extending the range of architectures catered for by the HOPP model. For each such architecture, a parallel implementation must be provided for all the recognised functions, together with the corresponding costs. The cost model must describe the techniques for handling the implementation of nested recognised functions and must provide cost estimates for any data distribution functions which may be involved in the implementation of programs.

8.2 Contributions of Thesis

The main contributions of this thesis can be summarised as follows.

- The development of the HOPP model which provides an extended set of predefined implicitly parallel constructs called *recognised functions*, in an effort to make the task of writing a wider class of programs more feasible.

- Cost-effective parallel implementations have been devised for the functions in this extended set on four different interconnect topologies. The cost associated with each such implementation has also been derived.

- A hierarchical cost model has been developed for these topologies. The cost model for the hypercube topology has been implemented in the form of an analysis program. The analyser considers the costs of the parallel implementations for the recognised functions and their arguments, in the selection of a cost-effective parallel implementation for the program. It also considers the costs associated with a number of possible parallel implementations for the program before selecting a cost-effective one.

- The performance of the model has been illustrated by implementing three example programs. The predictions of the analyser have been confirmed by the experimental results, in almost all the cases.

8.3 Limitations

There are some limitations to the analyser in its current form. Most of these can be overcome by the incorporation of some additional features in the analyser. However, there are some limitations to the model arising from the nature of the chosen constructs and cost analysis, and these cannot be wished away. They constitute the trade-offs involved in obtaining a simple parallel programming model that provides a high level of abstraction. These limitations are discussed below.

- The main limitation is that only *regular* problems can be effectively parallelised using this approach. In this context, a *regular* problem is one whose performance does not depend on the nature of the input data and one which also has a predictable communication structure. For problems which are not *regular* in this sense, the analyser cannot be guaranteed to make performance predictions that will reflect practical behaviour. This limitation arises partly due to the use of compile-time techniques to predict performance and also due to the nature of the constructs available for expressing parallelism.

- The fixed repertoire of constructs for expressing programs does limit the programmer. Although this set comprises of functions which should allow the expression of a number of types of data-parallel operations, situations could arise when this becomes difficult. For example, it may be difficult to express a program based on an array data structure with array operations.

- The analyser in its current form requires more assistance from the programmer than is actually necessary, e.g. it requires estimates of the sizes of input data structures and the costs of sequential functions. These demands on the programmer could be removed by extending the analyser so that it incorporates profiling capabilities. The analyser also lacks type inference and requires the programmer to explicitly supply the base types of input lists and also the input and output types of sequential functions. This could also be automated.

- The analyser cannot detect certain types of transformations of the input lists by sequential functions (See Section 5.1). Consequently, the estimates of the sizes of the corresponding output lists would be inaccurate. This in turn would result in inaccuracies in the cost estimates for subsequent phases of the program. The analyser could be extended so that some of these transformations are detected. Again, this would involve incorporating type inference in the analyser. However, due to the compile-time techniques which are used, a transformation of an input list resulting in an output list of different size, cannot be detected by the analyser.

8.4 Avenues for Further Research

The research reported in this thesis opens several avenues for further investigation.

- An immediate possibility would be the construction of a compiler to automatically generate parallel code for the implementation selected by the analyser. Such a compiler would be targeted at a parallel machine like the Meiko Computing Surface or the Cray-T3D. The design of the compiler could be such that it generates code in a high-level language such as *C*, with the code for the appropriate communication constructs being generated. Standard communication harnesses such as MPI could be used. This would have a double advantage in that the issues involved in generating efficient machine code would be transferred to standard *C* compilers and the portability of the program would also be enhanced. The onus of generating efficient *C* code is, however, still on the compiler. Another option is to design the compiler such that it generates code in some intermediate form, which in turn could be executed on the parallel machine. Some method of inserting the communication constructs would have to be devised.

- The cost model was studied on four different target topologies, of which only two were found to be cost-effective. The model could be extended to cater for other interconnect topologies, such as, fat-trees [Lei85].

- The hypercube embedding in the 2-D torus can be implemented. An analyser to predict execution costs and select cost-effective implementations on the torus can then be realised.

- The scheme only considers parallel implementations for recognised functions that do not occur within sequential functions. Any occurrences of recognised functions within sequential functions are implemented sequentially. A possible extension would be to consider the exploitation of potential parallelism for recognised functions nested within sequential functions. This would possibly involve the formulation of rules that describe when such exploitation of parallelism is allowed, so that semantic inconsistencies are not introduced into the program.

- The cost model is hierarchical but it considers parallel implementations for nested recognised functions only up to the first three levels. This could be extended to handle more levels, if the situation arises. Alternatively, the number of recognised functions implemented in parallel could still be three, but instead of simply choosing the first three recognised functions for analysis, parallel implementations involving recognised functions in other levels could be analysed.

- The search tree constructed by the analyser is exhaustive in the sense that it considers all the possible parallel implementations described by the model, for every phase in the program, before selecting the least-cost implementation. This causes an exponential increase in the size of the search tree with increase in the number of phases in the program. However, this is probably not required since many of the implementations would perform too poorly to merit consideration. Heuristics could be designed to eliminate such poor implementations at the analysis stage itself, in order to reduce the search space. This would probably involve a combination of local minimisation and inter-phase elimination techniques.

- The limitations of the analyser discussed in Section 8.3 could be overcome by extending the analyser to incorporate the required capabilities.

- The analyser cannot handle problems where the performance of sequential functions depends on the actual *value* of the input data. In other words, the cases where the cost of a sequential function varies for different data values, cannot be handled. This arises due to the limitation of *regularity* that is imposed on the nature of the problem that can be analysed. However, the introduction of profiling techniques in the analyser could be used to overcome the problem to some extent. Profiling could be used to obtain estimates of the values of each data set. This could, in turn, be used by the analyser to predict execution costs and select a cost-effective parallel implementation for that data set. The implementation of a phase containing a sequential function whose cost is a function of the input value, would probably involve placing different numbers of data items on processors, in order to achieve load balance. The analyser could be extended to handle such cases, thereby making it less restrictive.

- Program transformations can be employed to generate efficient parallel implementations for programs. Programs written in the functional style naturally lend themselves to transformations. The cost model could be used to predict the costs of different versions of the program, obtained by using semantics-preserving transformations. The parallel implementation corresponding to the least-cost version could then be selected for execution.

- More example programs from different scientific disciplines could be tested on the scheme. This could assist in identifying more useful functions which could be incorporated in the set of recognised functions.

- The model is currently based on distributed-memory machines. It would be interesting to develop a cost model for shared-memory machines.

Bibliography

[ADM87] Andrew W. Appel, Bruce F. Duba, and David B. MacQueen. Profiling in the presence of optimization and garbage collection. Technical Report CS-TR-197-88, Princeton University, Dept. Computer Science, Princeton, NJ, USA, November 1987.

[AE88] Arvind and Kattamuri Ekhanadham. Future scientific programming on parallel machines. *J. Par. and Dist. Computing*, 5:460–493, 1988.

[AJ89] Leonart Augustsson and Thomas Johnsson. Parallel graph reduction with the $< v, g >$-machine. In *Conference on Functional Programming Languages and Computer Architecture*, pages 203–213, 1989.

[Akl89] Selim G Akl. *The Design and Analysis of Parallel Algorithms.* Prentice-Hall, 1989.

[Ame69] William Ames. *Numerical Methods for Partial Differential Equations.* Thomas Nelson and Sons Ltd., 1969.

[AN87] Arvind and Rishiyur S Nikhil. Executing a program on the MIT tagged-token dataflow architecture. In *Parallel Architectures and Languages Europe*, volume LNCS 259, pages 1–29, Eidhoven, The Netherlands, 1987. Springer-Verlag.

[B$^+$91] Beguelin et al. *A User's Guide to PVM Parallel Virtual Machine.* Oak Ridge National Laboratory, USA, July 1991.

[Bac78] John Backus. Can programming be liberated from the von Neumann stye : a functional style and its algebra of programs. *Comm. ACM*, 21(8):613–41, August 1978.

[Bai94] Peter Bailey. Algorithmic skeletons in paraML. Research Report: TRACS-funded Visit to EPCC, July 1994. Edinburgh Parallel Computing Centre.

[BCMT93] Alasdair R A Bruce, Simon R Chapple, Neil B MacDonald, and Arthur S Trew. CHIMP and PUL: Support for portable parallel computing. Technical Report EPCC-TR93-07, Edinburgh Parallel Computing Centre, March 1993.

[BD$^+$93] Bruno Bacci, Marco Danelutto, et al. P^3L: A structured high-level parallel language and its support. Technical Report HPL-PSC-93-55, Pisa Science Centre, Hewlett-Packard Laboratories, Pisa, Italy, May 1993.

[BDS80] Rod M Burstall, John Darlington, and Don Sanella. Hope: An experimental applicative language. Technical Report CSR-62-80, Department of Computer Science, University of Edinburgh, 1980.

[BGP93] Eerke A Boiten, A Max Geerling, and Helmut A Partsch. Transformational derivation of (parallel) programs using skeletons. Technical Report 93-20, Computing Science Institute, Katholieke Universiteit Nijmegen, September 1993.

[Bir87a] Richard Bird. A calculus of functions for program derivation. Technical Report Technical Monograph PRG-64, Oxford University Computing Laboratory, United Kingdom, December 1987.

[Bir87b] Richard Bird. An introduction to the theory of lists. *Logic of Programming and Calculi of Discrete Design*, F36:5–42, 1987.

[Bir89] Richard Bird. Algebraic identities for program calculation. *The Computer Journal*, 32(2):122–126, 1989.

[Ble89] Guy Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, 11:1526–1538, November 1989.

[Ble93] Guy Blelloch. NESL: A nested data-parallel language. Technical Report CMU-CS-93-129, School of Computer Science, Carnegie Mellon University, April 1993. Updated 1994-version available.

[BN93] Peter Bailey and Malcolm Newey. Implementing ML on distributed memory multicomputers. *ACM SIGPLAN Notices*, 28(1):59–63, 1993.

[Bra93] Tore A Bratvold. A skeleton-based parallelising compiler for ML. In *Proceedings of the 4th International Workshop on Implementation of Functional Languages*, pages 23–24, Nijmegen, The Netherlands, September 1993.

[Bra94] Tore A Bratvold. *Skeleton-based Parallelisation of Functional Programs*. PhD thesis, Herriot-Watt University, 1994.

[Bre83] Gordon Brebner. *Parallel Computation on Sparse Networks of Processors*. PhD thesis, University of Edinburgh, 1983. CST-25-83.

[BS81] F W Burton and M R Sleep. Executing functional programs on a virtual tree of processors. In *ACM Conference on Functional Programming Languages and Computer Architecture*, pages 187–194. ACM, 1981.

[Bus93] David Busvine. *Detecting Parallel Structures in Functional Programs.* PhD thesis, Dept of Computing and Electrical Engineering, Heriot-Watt University, October 1993.

[BW88] Richard Bird and Phil Wadler. *Introduction to Functional Programming.* Prentice Hall, 1988.

[C$^+$89] A Contessa et al. MaRS: A combinator graph reduction multiprocessor. In E Odijk, J-C Syre, and M Rem, editors, *PARLE'89*, number 365, Vol. 1 in LNCS, pages 176–192. Springer-Verlag, 1989.

[Chu41] Alonzo Church. *The Calculi of λ-conversion.* Princeton University Press, 1941.

[Col87] Murray Cole. *Algorithmic Skeletons : Structured Management of Parallel Computations.* PhD thesis, University of Edinburgh, 1987.

[Col88] Murray Cole. Higher-order functions for parallel evaluation. In *Proceedings of the 1988 Glasgow Workshop on Functional Programming*, pages 8–20, August 1988.

[Col89] Murray Cole. *Algorithmic Skeletons : Structured Management of Parallel Computations.* Research Monographs in Parallel and Distributed Computing. Pitman/MIT Press, 1989.

[D$^+$93] John Darlington et al. Parallel programming using skeleton functions. In *PARLE '93 Parallel Architectures and Languages Europe*, 5th International PARLE Conference, Munich, Germany, pages 146–160, June 1993.

[DDD95] H Deldarie, J R Davy, and P M Dew. The performance of parallel algorithmic skeletons. Research Report Series 95.6, School of Computer Studies, University of Leeds, March 1995.

[DK82] Alan L Davis and Robert M Keller. Data flow program graphs. *IEEE Computer*, pages 26–41, February 1982.

[DM$^+$92] Marco Danelutto, Roberto Di Meglio, et al. A methodology for the development and the support of massively parallel programs. *Future Generation Computer Systems*, 8:205–220, August 1992.

[DR81] John Darlington and M J Reeve. ALICE - a multi-processor reduction machine for the parallel evaluation of applicative languages. In *Proc. ACM Conference on Functional Programming Languages and Computer Architecture*, pages 65–75, 1981.

[DT93] John Darlington and Hing Wing To. Building parallel applications without programming. Presented at the Second Workshop on Abstract Machine Models for Highly Parallel Computers, Leeds, April 1993.

[DTG93] John Darlington, Hing Wing To, and M Ghanem. Structured parallel programming. In *Working Conference on Massively Parallel Programming Models: Suitability, Realisation and Performance*, Berlin, Germany, 1993.

[Eka91] Kattamuri Ekanadham. A perspective on Id. *Parallel Functional Languages and Compilers*, 1991.

[FSWC92] David Feldcamp, H V Sreekantaswamy, Alan Wagner, and S Chanson. Towards a skeleton-based parallel programming environment. *Transputer Research and Applications*, 5:104–115, 1992.

[FW93] David Feldcamp and Alan Wagner. Parsec - a software development for performance oriented parallel programming. *Transputer Research and Applications*, 6:247–262, 1993.

[Gee94] A Max Geerling. Program transformations and skeletons: formal derivation of parallel programs. Technical Report CSI-R9411, Computing Science Institute, Katholieke Universiteit Nijmegen, October 1994.

[GL93] Sergei Gorlatch and Christian Lengauer. Parallelisation of divide-and-conquer in the Bird-Meertens formalism. Technical Report MIP-9315, Department of Mathematics and Informatics, University of Passau, December 1993.

[Gol88] Benjamin Goldberg. Multiprocessor execution of functional programs. *International Journal of Parallel Programming*, 17(5):425–473, 1988.

[Gol89] Benjamin Goldberg. *Multiprocessor Execution of Functional Programs*. PhD thesis, Yale University, 1989.

[Gor95] Sergei Gorlatch. From transformations to methodology in parallel program development: A case study. Technical Report MIP-9508, Department of Mathematics and Informatics, University of Passau, May 1995.

[H$^+$92] Paul Hudak et al. Report on the programming language Haskell, a non-strict, purely functional language, version 1.2. *ACM SIGPLAN Notices*, 27(5), May 1992.

[HCAA93] James Hicks, Derek Chiou, Boon Seong Ang, and Arvind. Performance studies of Id on the monsoon dataflow system. *Journal of Parallel and Distributed Computing*, 18:273–300, 1993.

[HH93] H.Stolze and H.Kuchen. Parallel functional programming using algorithmic skeletons. In *Parallel Computing : Trends and Applications*, Proceedings of the International Conference ParCo93, Grenoble, France, pages 651–654, September 1993.

[HJJ94] Kevin Hammond, Jim S Mattson Jr, and Simon L Peyton Jones. Automatic spark strategies and granularity for a parallel functional language reducer. In *CONPAR*, September 1994.

[HR86] Peter Harrison and M J Reeve. The parallel graph reduction machine ALICE. In Joseph H Fasel and Robert M Keller, editors, *Workshop on Graph Reduction*, volume LNCS 279, pages 181–202, Santa Fe, New Mexico, USA, 1986. Springer-Verlag.

[Hug82] John Hughes. Graph reduction with supercombinators. Technical Report Technical Monograph PRG-28, Oxford University Computing Laboratory, United Kingdom, 1982.

[Hug90] John Hughes. Why functional programming matters. In David Turner, editor, *Research Topics in Functional Programming*, pages 17–42. Addison-Wesley, 1990.

[Jay95] C.Barry Jay. Shape analysis for parallel computing. In John Darlington, editor, *Proceedings of the Fourth International Parallel Computing Workshop: Imperial College London, 25–26*, pages 287–298. Imperial College/Fujitsu Parallel Computing Research Centre, September 1995.

[JCSH87] Simon L Peyton Jones, Chris Clack, Jon Salkild, and Mark Hardie. GRIP - a high performance architecture for parallel graph reduction. In *Functional Programming Languages and Computer Architecture*, pages 98–112, Berlin, September 1987. Springer-Verlag.

[JH92] Simon L Peyton Jones and Kevin Hammond. Profiling scheduling strategies on the GRIP parallel reducer. In *4th International Workshop on the Parallel Implementation of Functional Languages*, Aachen Germany, September 1992. Aachener Informatik-Berichte Nr 92-19.

[Joh84] Thomas Johnsson. Efficient compilation of lazy evaluation. In *SIGPLAN 84 Symposium on Compiler Construction*, pages 58–69, Montreal, Canada, 1984.

[Jon87] Simon L Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[Jon89] Simon L Peyton Jones. Parallel implementations of functional programming languages. *The Computer Journal*, 32(2):175–186, 1989.

[Jou91] Guido K Jouret. Compiling functional languages for SIMD architectures. In *3rd. IEEE Symposium on Parallel and Distributed Languages*, December 1991.

[Jr93] Jim S Mattson Jr. Performance of parallel schedulers for distributed graph reduction. In *Nijmegen Workshop on the Implementation Functional Languages*, 1993.

[JS89] Simon L Peyton Jones and Jon Salkild. The spineless tagless G-machine. In *Functional Programming Languages and Computer Architecture*, pages 184–201. ACM, September 1989.

[Kea94] John A Keane. An overview of the Flagship system. *Journal of Functional Programming*, 4(1), January 1994.

[Kel89] Paul Kelly. *Functional Programming for Loosely-coupled Multiprocessors.* Research Monographs in Parallel and Distributed Computing. Pitman/MIT Press, 1989.

[Lei85] Charles E Leiscerson. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, C-34(10), October 1985.

[LKB91] Hugh Lester, David R Kingdon, and Geoffrey L Burn. The HDG-machine: A highly distributed graph reducer for a transputer network. *The Computer Journal*, 34(4):290–301, 1991.

[Mei] Meiko. *Computing Surface - CSTools Documentation Guide.*

[Mes94] Message Passing Interface Forum. *MPI:A Message-Passing Interface Standard*, January 1994.

[MHT89] Robin Milner, Robert Harper, and Mads Tofte. The definition of standard ML. Technical Report ECS-LFCS-89-81, LFCS, Department of Computer Science, University of Edinburgh, 1989.

[Mic89] Greg Michaelson. *An Introduction to Functional Programming through Lambda Calculus.* Addison-Wesley, 1989.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[Mil93] Richard Miller. A constructive theory of multidimensional arrays. Programming Research Group, University of Oxford, February 1993. MSc to DPhil transfer.

[NSvEP91] E G J M H Nocker, J E W Smetsers, M C J D van Ekelen, and M J Plasmeijer. Concurrent Clean. In E H L Aarts, J van Leeuwen, and M Rem, editors, *PARLE'91*, number 506, Vol. 2 in LNCS, pages 202–219. Springer-Verlag, 1991.

[P$^+$88] William Press et al. *Numerical Recipes in C - The Art of Scientific Computing.* Cambridge University Press, 1988.

[PD93] Susanna Pelagatti and Marco Danelutto. Structuring parallelism in a functional framework. Technical Report TR-29/93, Dipartimento di informatica, Università di Pisa, 1993.

[Pel93] Susanna Pelagatti. *A Methodology for the Development and the Support of Massively Parallel Programs.* PhD thesis, Università di Pisa-Genova-Udine, Italy, 1993. TD 11/93.

[Ran95] Roopa Rangaswami. HOPP - a higher-order parallel programming model. In Marc Moonen, editor, *Algorithms and Parallel VLSI Architectures.* Elsevier, 1995.

[Roe94] Paul Roe. Derivation of efficient data parallel programs. In *17th Australasian Computer Science Conference*, pages 621–628, 1994.

[San93] Patrick M. Sansom. Time profiling a lazy functional compiler. In *Glasgow Workshop on Functional Programming*, pages XXIII–1 – XXIII–6, 1993.

[Sar89] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors*. Research Monographs in Parallel and Distributed Computing. Pitman/MIT Press, 1989.

[Sar91] Vivek Sarkar. PTRAN - The IBM Parallel Translation System. In *[Szy91]*, 1991.

[SC93] David B Skillicorn and Wentong Cai. A cost calculus for parallel functional programming. Technical report, Department of Computing and Information Sciences, Queens University, Kingston, Canada, March 1993.

[Ske91] Stephen K Skedzielewski. SISAL. *Parallel Functional Languages and Compilers*, 1991.

[Ski91] David B Skillicorn. Models for practical parallel computation. *International Journal of Parallel Programming*, 2(20):133–158, April 1991.

[Ski92] David B Skillicorn. The Bird-Meertens formalism as a parallel model. *NATO ARW, Software for Parallel Computation*, June 1992.

[Smi65] Gordon Smith. *Numerical Solution of Partial Differential Equations.* Oxford University Press, 1965.

[Spi89] Mike Spivey. A categorical approach to theory of lists. *Mathematics of Program Construction*, LNCS 375:399–408, June 1989.

[SS89] Youcef Saad and Martin Schultz. Data communications in hypercube. *Journal of Parallel and Distributed Computing*, 6:115–135, 1989.

[Szy91] Boleslaw K Szymanski, editor. *Parallel Functional Languages and Compilers.* Frontier Series. ACM Press, New York, 1991.

[Tof89] Mads Tofte. Four lectures in standard ML. Technical Report ECS-LFCS-89-73, LFCS, Department of Computer Science, University of Edinburgh, March 1989.

[Tur79] David A Turner. A new implementation technique for applicative languages. *Software - Practice and Experience*, 9:31–49, 1979.

[Ull94] Jeffrey D Ullman. *Elements of ML Programming.* Prentice Hall, 1994.

[YM89] Y.Saad and M.H.Schultz. Data communications in parallel architectures. *Parallel Computing*, 11:131–150, 1989.