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.