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.