A COMBINATIONAL FRAMEWORK FOR PARALLEL PROGRAMMING USING ALGORITHMIC SKELETONS

by

Mohammad M. Hamdan

Submitted for the degree of Doctor of Philosophy

at Heriot-Watt University

on Completion of Research in the

Department of Computing and Electrical Engineering

January 2000.

I hereby declare that the work presented in this report was carried out by myself at Heriot-Watt University, Edinburgh, except where due acknowledgement is made.

Mohammad M. Hamdan (Candidate)

(Supervisor)

Date

Contents

List of Tables

List of Figures

Acknowledgements

My praises to God for giving me the good health, the strength of determination and support to finish my work successfully.

This thesis could never have been written without the helpful support from a number of peoble to whom I would like to pay tribute.

First and formost I would like to thank my supervisors, Greg Michaelson and Peter King for their strong support and genuine enthusiasm for my work throughout the project. Also, thanks to Norman Scaife for commenting on a draft of this thesis and Jamie Stark for his help in using LaTeX.

I would like to thank the British Council, Yarmouk University, the Arab British Chamber of Commerce and my family for their financial help.

Abstract

This thesis investigates a model of parallel functional programming, where parallelism is identified through the use of higher order functions (HOFs) with equivalent algorithmic skeletons. In this model, parallel applications are naturally constructed using a HOF or, depending on the complexity of the problem, a combination of two or more HOFs.

To fully utilise the benefits of skeleton-oriented programming, it is important to exploit parallelism available in nested HOFs as hierarchical structures of skeletons. While a number of skeleton-based compilers have been constructed, they either provide for unnested HOFs or place restrictions on the depth or structure of skeleton hierarchies.

In the research described in this thesis, the HOF/skeleton model is realised through a novel framework for compiling and executing arbitrary depth HOF nesting and composition in a fully automatic compiler which generates parallel code that does not depend on the target architecture. The framework consists of translation rules (schemes) used at compile-time for the static analysis of the source program and at run-time to execute the combined skeletons in parallel. A static analysis of the source program generates a nesting structure which is used to combine the corresponding HOFs in a process termed **nesting deduction**. The run-time scheduling relies on the compile-time analysis and uses message-passing groups to run combined HOFs in parallel. Processes are allocated to skeletons through the traversal of the HOF nesting structure and processes may be reallocated within a group to support the nested skeletons.

The framework is illustrated through Ektran, a simple functional language for combining HOFs, which acts as a coordination language for skeletons. The implementation of Ektran's compiler is discussed which supports heterogenous pipeline parallelism and the performance of Ektran programs on several distributed memory multiple instruction multiple data platforms is considered.

To my parents

Chapter 1

Introduction

1.1 Motivation and Why Structured Parallel Programming?

There is an ever increasing need for faster computers. However, speed in computers is limited by physical limits as their components cannot exceed the speed of light. Therefore, introducing parallelism into both the application and the computer system is a key technology for solving very large scale problems.

In principle parallel processing is a limitless approach. However, programming parallel machines is a difficult task and varies in difficulty according to the programming model used. Also, the reduced life-cycle of parallel machines due to advances in hardware technology requires new investments to produce software models for the new architectures. Therefore, users require programming models which support:

- **Programmability**, the model should be easy to use and allows the programmer to focus on the problem being solved rather than the target architecture being used.
- **Portability**, the programs should be both code and performance portable across many different parallel machines.
- **Massive Parallelism**, the underlying software components should implicitly exploit the available parallelism in a program to the maximum degree.

Algorithmic Skeletons is a general model [153] for programming parallel machines which was introduced by **Murray Cole** [40] in 1989. This model provides high abstraction, portability across different architectures and high performance. However, algorithmic skeletons prevent the programmer from writing parallel algorithms where it is not possible to capture such an algorithm by combining the current skeletons. Previous work prevented skeleton combinations [20, 40, 45], put restrictions on skeleton nesting for example **Nesl** [15] and **HOPP** [136], asked the programmer to explicitly craft the skeletons together as in **P^{3}L** [7], or provided the programmer with special skeletons to combine skeletons as in **SCL** [171, 49, 47].

The purpose of this thesis is to propose and illustrate a parallel functional programming model based on combining skeletons. Such an **implicit**[1] model and other **explicit**[2] parallel programming models are generally known as **structured parallel programming** [171, 127, 73]. This is similar to structured sequential programming where programs are expressed as the nested composition of a fixed set of control structures. The basic model is introduced in Section 1.3 and is explained and discussed in the remainder of the thesis.

1.2 Background

Some relevant background and related material are reviewed in the following subsections.

1.2.1 Functional Programming Languages

Functional programming is an approach to programming based on function calls as the primary programming construct which is often based on lambda calculus [117]. **Lambda calculus** [38] was developed by **Church** to investigate decidability problems in mathematical logic. Also, it is an attractive vehicle for investigating various programming concepts.

The Turing Award paper by **Backus** [9] has increased interest in functional languages over imperative languages which are closely tied to the von Neumann architecture with its step-by-step model of evaluation. Also, functional programs contain easily detectable implicit parallelism [92]. A well known feature of functional languages is called **referential transparency**. This feature guarantees that the value of an expression only depends on its denotation and not on the order of computations. Therefore, the subexpressions of an expression can be evaluated in any order. Also it allows functions to be combined to form more complex functions [66].

Most functional languages have either strict or non-strict semantics. The former is usually implemented using eager evaluation [3] while the later is often implemented using lazy evaluation [4].

1.2.2 Classification of Parallel Computers

Figure 1.1 (which is a graphical illustration of **Gurd et al's** framework[60]) shows a general classification of parallel computers, which is based on two criteria: (1)

[1]By an implicit model we mean the combination of skeletons is identified automatically by the compiler.

[2]By an explicit model we mean that the programmer has to combine the skeletons together using as a set of combination rules or skeletons.

[3]In eager evaluation, the arguments of a function are evaluated before passing them to the function.

[4]In lazy evaluation, the arguments of a function are evaluated only when they are needed in the function body.

the number of instruction and data streams (mainly SIMD and MIMD) [5] and (2) whether a system contains only distributed memory or a globally addressable shared memory. The first criterion corresponds to the two fundamental activities in a computer system while the second describes the basic entities of a memory system and its relation to the processor(s). For examples of each of these classes and a unifying model for parallel hardware see [60]. However, it is possible to combine the power of SIMD and the generality of MIMD in a single machine. The Rewrite Rule Machine (RRM) [59] is an example.

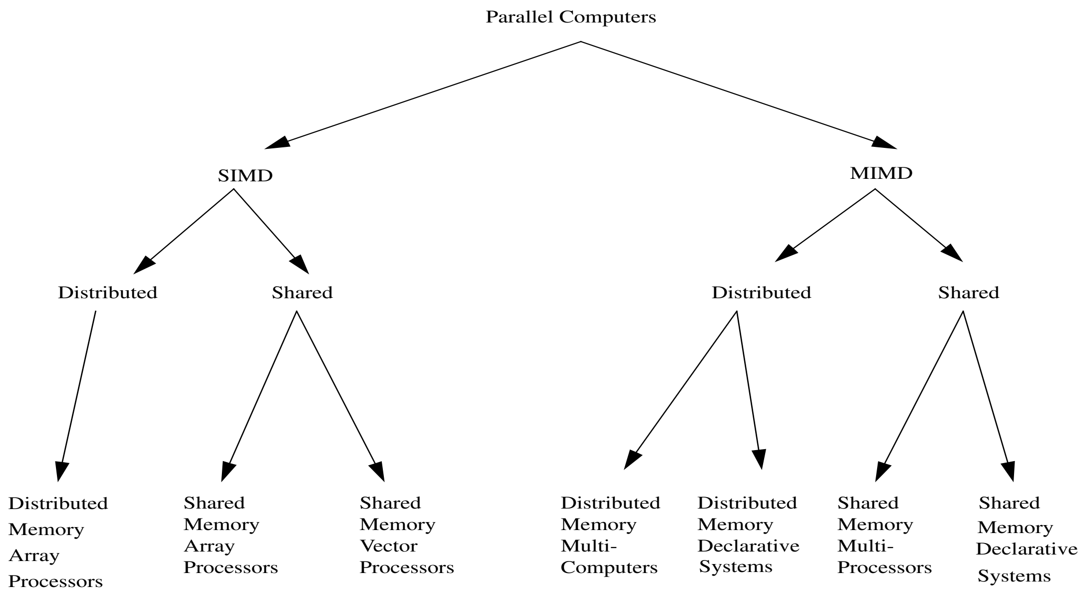

Figure 1.1: Classes of parallel machines.

This thesis investigates a parallel programming model for distributed memory multi-computers which we shall refer to them as DM-MIMD machines. Examples of such machines include the Fujitsu AP1000/AP3000 [70], the Cray T3D [42], the Intel Paragon [114], the IBP SP2 [158], the Thinking Machines CM-5 [106], the Meiko Computing Surface [111] and network of workstations (NOWs). This class of machines was chosen as the communication and data distribution are more explicit than other classes which will make it easier to investigate the behaviour of the proposed programming model on these systems. However, it was possible to have access to a limited number of these machines.

It is common to program the DM-MIMD class of parallel machines using the **Single Program Multiple Data** (SPMD) parallel programming model [154]. This involves a single program executing on different processors and different data sets. Each processor has a copy of the program and works on a subset of the input data. The level of synchronisation in SPMD models is of coarser grain than

[5]SIMD and MIMD characterisation were proposed by Flynn [68] and stand for Single Instruction Multiple Data and Multiple Instruction Multiple Data respectively.

traditional SIMD models [135]. However, the Group SPMD model [67] (a more general model of parallelism than SPMD) is used later in the thesis. It allows the concurrent execution of different SPMD sub-programs on separate groups of processors when resources permit [86].

1.2.3 Parallel Programming Environments

MacDonald and **Norman** suggest that parallel programming environments may be considered according to the facilities they provide for [113]:

- Detecting parallelism,
- Control of partitioning,
- Control of mapping and
- Abstraction from communication.

The above facilities determine the generality, programmability and efficiency of parallel programming environments. If the programmer has to manage them, then parallel programming becomes very hard. Instead, if these are done by one or more of the underlying system components (i.e. the compiler, run-time system, operating system and hardware) then the programmer will concentrate more on designing efficient algorithms for the problem to be solved.

The dominant parallel programming models are: (1) message passing [6] and (2) shared memory [7]. The former is easy to build but very difficult to program, on the other hand, the latter is hard to build but easy to program.

1.2.3.1 Message Passing

In message passing, the communication between two processes is explicit. For example, a process will SEND a message and another process will WAIT for it to arrive. There are many message passing models according to the definition of these SEND and WAIT commands, the two main models [25] are:

Synchronous Message Passing: In this model the process that SENDs a message is delayed until the corresponding WAIT is executed. A well known parallel programming language that uses this model is **Occam** [25] where communication between two processes is done via named and dedicated channels.

Asynchronous Message Passing: This model allows the process that has sent a message to continue execution arbitrarily. In other words, the sending process will not wait until the other process has received the message.

[6] This model is outlined in Section 1.2.3.1 as it is needed as a background material for material presented in Section 5.2.

[7] This model is outlined in Section 1.2.3.2 as it is needed as a background material for the systems described in Section 5.6.

However, the communication in these two models is managed by the programmer which makes parallel programming very hard. Therefore, the programmer can use message passing libraries to avoid dealing with low level communications. For example, the Parallel Virtual Machine (PVM) [57] and the Message Passing Interface (MPI) [121] provide facilities for [128]:

- Creating processes on remote machines,
- Monitoring the state of these remote processes and
- Sending messages from one program to another.

1.2.3.2 Shared Memory

The concept of shared memory as a programming model is preferred by programmers over message passing as it is a natural continuation of sequential programming. **Scott** notes that shared memory is a programming *interface*, not a model of performance [147].

Shared memory systems can be implemented using hardware or software. Hardware-based shared memory systems are often called **Virtual Shared Memory** (VSM) systems and use hardware assistance (similar to hardware cache-coherence mechanisms [8]) to provide a coherent address space [134]. Two well known examples of VSM system are the **DASH** (Distributed Architecture for SHared Memory) [58] and the **MIT Alewife** [56] multiprocessors.

Software-based shared memory systems are often called **Distributed Shared Memory** (DSM) systems. A DSM system [104] gives the illusion of shared memory on top of standard message passing hardware at very low implementation cost, but gives acceptable performance on only a limited class of applications. DSM systems are classified according to the strategy they use for detecting writes to shared data [115]: (Virtual Memory) VM-DSM and (Run-Time) RT-DSM systems. The first approach relies on virtual memory management to provide system-wide page-sharing and to detect and collect writes to shared data. The first DSM system that used this approach was **Ivy** [108] which was implemented on a network of workstations. In the second approach, the compiler and the run-time system are used to detect and collect writes to shared data at user-level [115]. Examples of RT-DSM systems include **Orca** [11] and **Amber** [61].

A key issue in shared memory systems is memory consistency models. That is how to keep memory consistent where a read to a shared memory location must return the most recently written value. This is what programmers intuitively expect and is the strictest memory consistency model. The canonical memory consistency model is **sequential consistency** which was defined by **Lamport** [105] as:

> the result of any execution is the same as if the operations of all processors were executed in some sequential order, and the operations of

[8] Cache-coherence mechanisms maintain a consistent state of multiple copies of the same data item.

each individual processor appear in this sequence in the order specified by its program.

1.2.4 Parallelising Functional Languages

The work presented in this thesis is an approach to parallelise functional languages. Therefore, below we will outline other approaches that exploit parallelism in functional languages. See **Hammond** and **Michaelson** [87] for a recent survey on parallel functional programming.

The work in this area ranges between two extremes: implicit and explicit approaches. Also, in the middle lie "semi-explicit" approaches. **Loogen** [110] has classified the above models by the type of parallelism they exploit which could be data [9] or control [10] parallelism.

1.2.4.1 Purely-Implicit Approaches

In a purely-implicit approach for parallelising functional languages, the programmer has little or no involvement in the parallelisation process. It is completely managed by the underlying system components (compiler, run-time system and hardware). We think that the dataflow model of computation is at the top of this category. Here, programs are represented by a directed graph [51]. The nodes represent machine instructions and the arcs connecting nodes represent the flow of data between the nodes. When all data is available on the arcs of a node, the node executes the instruction or fires and places the result on its output arc. As several nodes can fire simultaneously, the dataflow gives rich opportunities for very fine-grain parallelism.

A number of implementations based on dataflow have been constructed (e.g. the Manchester Dataflow Project [78], and the MIT Tagged-token Project [5]). However, the fine grain parallelism characteristic of dataflow is difficult to exploit successfully and the feasibility of dataflow for practical computing remains to be demonstrated.

Another approach which falls under this category is **evaluation transformers** [26]. They are based on strictness analysis and provide parallelism control mechanisms for specific structured data types. However, the number of evaluation transformers increases as the strictness analysis gets more sophisticated especially when more types are supported.

1.2.4.2 Semi-Implicit Approaches

In semi-implicit approaches, the programmer is required to indicate sites for potential parallelism in the program like the use of annotations. The survey in **Junaidu**'s [95] thesis about the exploitation of implicit parallelism using advisory annotations is useful for further reading.

[9] Data parallelism is exploited by applying an operation to different parts of a data structure simultaneously.

[10] Control (or process) parallelism exploits parallelism by allowing arbitrary parallel execution of the control flow in a program.

Graph reduction is the most common approach for implementing lazy functional languages both sequentially [91] and in parallel [92]. In graph reduction, a program (after transforming it into combinator form) along with its data is represented as a graph. The leaves of the graph hold constant values, built-in functions and variables while nodes hold applications and lambda abstractions. 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. Parallelism can be exploited by performing simultaneous reductions on the program graph. A number of parallel graph reduction machines have been built and they rely on programmer's annotations to identify and exploit useful parallelism like **Applicative Language Idealised Computing Engine** (ALICE) [44], Flagship [97] and **Graph Reduction In Parallel** (GRIP) [93].

However, a major problem with graph reduction systems has to do with graph copying costs as function bodies need to be copied each time they are applied. See **Hammond**'s survey [85] for a historical overview of graph reduction in parallel.

Annotations is a common approach to parallelise functional languages. A recent system is **Naira** [95] which is both a parallel and parallelising compiler for a pure subset of Haskell. The front end runs in parallel and the compiler generates parallel code. The programmer has to annotate the source program to indicate the intention for parallelism explicitly. Here, the programmer assigns explicit parallelism and strictness annotations to components of tuples and arguments to function applications. Two built-in single-argument combinators, **process** and **value**, are used to partition programs statically.

Another approach is to use **parallel combinators** like `par` and `seq`. The `par` function is used to indicate potential parallelism and the `seq` combinator corresponds to sequential composition. They were used to parallelise Haskell programs in Glasgow Parallel Haskell (GPH) [162]. These are identity functions which influence the evaluation of their arguments. However, these combinators are scattered through the program code and tend to obscure the structure of the parallel program.

A restricted semi-implicit approach is to exploit data parallelism by performing an operation on all elements of a large data structure at the same time. **Hill** [90] has worked on exploiting data parallelism in HOFs like map, scan and fold. He proposed a mechanism called POD comprehension for expressing data-parallel computations. It uses an evaluation strategy which maintains a record of the elements of a parallel object that need to be evaluated in order to incorporate data-parallelism and non-strictness in the same language. In fact, exploiting data parallelism is a well known technique to parallelise imperative languages [88].

1.2.4.3 Semi-Explicit Approaches

In semi-explicit approaches, the programmer uses special constructs or operators to control the parallel execution of a program. These constructs are used to describe the parallel behaviour of a program but they hide in their implementations all low level details about parallelism. We also classify this category into controlled- and coordinated-models. The first approach uses high-order functions that manipulate sequential or parallel program components to yield a more complex program. The second approach extends the language with parallel annotations or a coordination language for the definition of parallel interacting processes. The second approach has been used within imperative languages more than functional languages and is described in more detail in Section 1.2.6.2.

Evaluation strategies [161] are a well known example of the first approach. These are functions used by the programmer to influence the dynamic behaviour of an algorithm by changing the degree of evaluation in favour of parallelism. They were built on top of the `par` and `seq` combinators.

Caliban [103] and **Eden** [21] are examples of the second approach. In Caliban, the control language is used to determine a static mapping of parallel tasks to processors. Eden takes a similar approach to Caliban. Process networks are defined declaratively and lazy lists are used to model communication channels.

1.2.4.4 Fully-Explicit Approaches

In fully-explicit approaches, the programmer deals with low level details such as process placement, synchronisation and communication. Therefore, languages which fall under this category provide the programmer with low level constructs for managing parallelism. Examples of such an approach are **MultiLisp** [81], **Simple CAml interface to MPI** (SCAMPI) [148] (which provides some message-passing bindings to a functional language) and **Concurrent ML** (CML) [123].

1.2.5 Algorithmic Skeletons

The previous approaches for parallelising functional languages pose a dilemma for programming parallel machines. They were either highly abstract and portable with poor performance, or with high performance but very difficult to use. Therefore, there is a strong need for an alternative approach to parallelise functional languages which is highly abstract, portable and achieves high performance.

Therefore, **Cole** has proposed to use **algorithmic skeletons** as a technique to parallelise functional languages and to program parallel machines. The idea is to capture common patterns of parallel computation in Higher Order Functions (HOFs). HOFs are a natural part of functional languages and it is straight forward to reason and manage about them and their parallel equivalents, especially where parallelism is restricted to a small set of HOFs. Finally, parallelising compilers are capable of exploiting efficiently their implicit parallelism [18] and they fall under the semi-implicit approach for parallelising functional languages with restrictions to the available parallelism they provide.

Also, **Skillicorn** and **Talia** have classified algorithmic skeletons as a nothing explicit and parallelism implicit parallel computation model. They also noted that the skeleton model is a static structure where an abstract program is based on fundamental units whose implementations are predefined.

The major advantage of **algorithmic skeletons** is the portability of parallel programs written using this approach. This results from the separation of meaning and behaviour for each skeleton, as identified by **Darlington et al** [46]. A HOF in a functional language can be used to express the declarative meaning for a skeleton which could have more than one implementation. Therefore, the declarative meaning is independent of any implementation and hence any behavioural constraints [48].

1.2.6 Programming Parallel Computers

The work presented in thesis is also relevant to non-functional models for programming parallel machines. In the next subsections, we will outline briefly a number of these models.

1.2.6.1 Programming using Parallel Library Routines

One of the common approaches to parallelise programming languages is to extend them with predefined parallel library routines. They correspond to operations that are frequently used by an application programmer. Two successful models are message passing and parallel scientific libraries.

The first model provides the programmer with a machine-independent interface to low level details about communication between processes. Therefore, the main goal of such models is the portability of programs and achieving acceptable performance. Two well known examples are MPI [121] and PVM [57].

The second model, which is a higher-level approach, resulted from parallelising scientific programming libraries such as BLAS and Linpack [37, 2]. **Brewer** [22] has developed a high-level library for stencil computations which are a technique for iteratively solving partial differential equations. Each routine has more than one parallel implementation and the compiler selects the best one for a given target architecture using a profiler and statistical modeling.

1.2.6.2 Coordination Models and Languages

A coordination model [4] consists of entities (such as processes), media (used for communication between entities) and laws or rules (control how entities communicate through media) [11].

The basic idea is to separate computation (which is the basic action that solve a problem) from coordination (which is the way in which computations are combined into a whole). Therefore, a parallel program consists of two parts: a computation model and a coordination model [72]. In his thesis, **Yang** [171] has suggested that coordination can be achieved using compiler directives as in Vienna Fortran [12], shared data pool as in Linda [31], channels as in Fortran M [69] and skeletons as in P^3L [62, 125].

There is plenty of work on coordination models, languages, theory, architecture and applications and it is beyond the scope of this thesis. In general, this

[11] An example from real life is coordination between airplanes. Here airplanes are entities, ether (radio waves) are media and the tower control is the law.

is an explicit structured approach which provides the programmer with special constructs to control the parallel behaviour of a program. However, the work presented in this thesis automates the process of finding coordination between skeletons as it is handled by the underlying system components (the compiler and the run-time system). In Section 2.3.13 the **Structured Coordination Language** SCL [171] is outlined which integrates the coordination language approach with the skeleton approach.

1.3 Thesis Proposal

The thesis proposes a contribution to the parallel programming model where parallelism is identified through the use of higher order functions (HOFs) with equivalent algorithmic skeletons. In this model, parallel applications are naturally constructed using a HOF or, depending on the complexity of the problem, a combination of two or more HOFs. The model exploits the available parallelism in nested HOFs as hierarchical structures of skeletons making a contribution to achieving massive parallelism.

The HOF/skeleton model is realised through a novel framework for compiling and executing arbitrary depth HOF nesting and composition in a fully automatic compiler that generates parallel code that does not depend on the target architecture. It consists of translation rules (schemes) used at compile-time for the static analysis of the source program and at run-time to execute the combined skeletons in parallel. The static-analysis of the source program generates a nesting structure which is used to combine the corresponding HOFs in a process we call nesting deduction. The run-time scheduling relies on the compile-time analysis and uses message-passing groups to run combined HOFs in parallel. Processes are allocated to skeletons through the traversal of the HOF nesting structure and processes may be reallocated within a group to support the nested skeletons. This involves dividing the group into sub-groups for each of its immediately nested skeletons, and creating a top group which connects all sub-groups. The process then continues recursively for inner skeletons.

The work presented in this thesis can be seen as the framework that combines separate activities into an ensemble to form a coordination model. Hence, it adds weight to the argument that skeletons are well suited as building blocks for high-level parallel programming provided that there is an automated mechanism which connects the components together.

1.4 Previous Publications

Some of the ideas in this thesis have been presented or published elsewhere.

More discussion of the cost models presented in Chapter 2 can be found in:

M. Hamdan. A Survey of Cost Models for Algorithmic Skeletons. Technical Report RM/99/11, Department of Computing and Electrical Engineering, Heriot-Watt University, UK, September 1999.

An overview of the compiler structure and the parallel implementations has been presented in:

M. Hamdan, G. Michaelson, and P. King. A Scheme for Nesting Algorithmic Skeletons. In Tony Davie Kevin Hammond and Chris Clack, editors, *Draft Proceedings of the 10th International Workshop on Implementation of Functional Languages, IFL '98*, pages 195–212, September 1998.

A fuller account of the complete system has been accepted to be published in:

M. Hamdan, G. Michaelson, and P. King. A Framework for Nesting Algorithmic Skeletons. In *Parallel Computing '99*, Delft, The Netherlands, August 1999. to be published by IC Press.

1.5 Thesis Outline

The rest of the thesis is organised as follows.

Chapter 2 gives a comprehensive survey of skeleton-oriented parallel programming, in particular within the functional programming community. The chapter gives a detailed comparison of skeleton models and systems. It also points at the advantages of exploiting parallelism available in combined skeletons and outlines the problems associated with implementing nested skeletons in parallel.

Chapter 3 presents the design, organisation and implementation of the parallel compiler. The components of the compiler are described and how they were integrated to provide an architecture-independent parallelising compiler. The parallel functional programming language **Ektran** is introduced along with its parallel operators which serves as a coordination language for skeletons.

Chapter 4 discusses in detail the compile-time analysis of an **Ektran** program and how the nesting structure used to combine the skeletons is generated. The analysis is presented in the form of translation rules (schemes) which work over **Ektran**'s grammar. It consists of two stages: preprocessing and translation. The first stage consists of four main parts: symbolic beta reduction, pipeline normalisation, nesting deduction and free variable detection. The second stage translates the program into an intermediate format where the annotations placed on the abstract syntax tree by the first stage have been converted to arguments to the parallel implementations of the skeletons.

Chapter 5 presents the run-time framework used to execute the combined skeletons in parallel. It consists of an automated mechanism that operates at run-time based on the compile-time analysis. It uses the concept of message-passing groups to run nested skeletons in parallel. Also the algorithms used for the parallel implementations of the supported skeletons are described.

Chapter 6 presents the research apparatus used to evaluate the **Ektran** model for parallel programming. The target architectures are described along with the benchmark programs used to evaluate the methodology. The performance of the programs are presented for different DM MIMD computers.

Finally Chapter 7 summarises the thesis, gives a conclusion of the work described and outlines future work. The contributions of the thesis are discussed and the limitations of the work are described.

Chapter 2

Skeleton-Oriented Parallel Programming

2.1 Introduction

Skeleton-oriented parallel programming is a promising model for architecture-independent parallel programming on general-purpose computers. The term **algorithmic skeletons** was coined by **Cole** in 1989 [40]. Since then, various researchers have looked into exploiting algorithmic skeleton parallelism inside and outside the functional programming community. This chapter surveys the major work done in this area in particular within the functional programming community. Section 2.2 first part looks at the initial work of algorithmic skeletons. A number of researchers have worked on similar skeletons and these are discussed from three different perspectives: 1) the HOFs used to represent them, 2) their parallel implementations and the target architectures used and 3) the cost models that were developed for them. Note that the primary operators for the parallel programming model which will be described in Chapter 3 were chosen from this initial set of skeletons.

Section 2.3 then looks at complete systems for skeleton-oriented parallel programming. Section 2.4 presents a detailed comparison of the work in this area. In Section 2.5 the importance of combining skeletons and the problems associated with their implementations are given. Finally the chapter is summarised in Section 2.6.

2.2 Skeleton Programming Environments

A common way to parallelise sequential languages is to extend the host language with parallel constructs using predefined library routines. Initial work on skeletons looked at parallelising functional languages by interfacing them with a parallel library. The interface came by replacing a given set of HOFs (which inherently have parallel properties) with their equivalent parallel implementations. A number of researchers looked at this technique to parallelise their systems. However, many of their skeletons were similar or even identical. This initial work on skeletons inspired other researchers to build more substantial systems using the concept of skeletons. This section surveys the major initial work done on skeletons.

Before presenting the survey we need to give our definition of **algorithmic skeletons**. A **skeleton** is the combination of source code HOF, an equivalent parallel construct and a cost (performance) model which allows the parallel performance to be predicted from analysis of the HOF.

A function can be called a higher-order function (HOF) [168] if it has one or both of the following properties: 1) it can take another function as an argument or 2) it is a result of function application. These features make HOFs highly abstract and they are often used over and over again. In particular, HOFs are used because they have implicit parallel properties.

A cost model is a mathematical formula used to estimate the execution time or complexity of a skeleton's performance. To derive a cost model for a skeleton, the skeleton's performance must be broken down into its principal components with respect to communication and computation times involved. These components are combined in a formula describing the sought performance characteristics. However, the cost model needs to be parameterised with the problem and machine characteristics to return an accurate prediction of the performance. Normally cost models are used to decide if the parallelism available in a HOF is worth exploiting. See **Hamdan** [82] for more detail about the cost models for algorithmic skeletons and [94, 112] for performance prediction of parallel programs.

It is important to note that a HOF can have more than one parallel implementation and one parallel implementation can correspond to a number of HOFs. It is the role of the performance model to identify the best parallel implementation for a skeleton.

Skeletons have been labeled by other researchers as templates, paradigms, patterns, forms and parallel constructs [20]. Besides, we can call the HOF as a high-level skeleton and the parallel implementation as a low-level skeleton. Normally, researchers refer to this approach of parallel programming as the HOF/skeleton model where a skeleton is the parallel implementation for a given HOF. However, in this thesis the terms HOF and skeleton are used interchangeably and refer to the same definition of algorithmic skeletons given previously.

Campbell's [30] general classification of algorithmic skeletons as recursively partitioned, task queue or systolic is used to present the skeletons. The categories in his classification are discussed in the following subsections. In each subsection, the first skeleton described was found to be the most general case of all similar skeletons corresponding to his classification. Most of the skeletons are described according to their HOFs, parallel implementations and cost models.

2.2.1 Recursively Partitioned

A **recursively partitioned** skeleton works by generating subordinate problems as a result of dividing the original problem. These generated problems will also be divided further in an attempt to solve them. Two well known algorithms that belong to this category are quick-sort and least-cost search. **Rabhi** [131] gave a

HOF describing this skeleton:

```
rp$ ind solve divide combine prob
    | ind prob = solve prob
    | otherwise = combine prob
      (map (rp$ ind solve divide combine) (divide prob))
```

where:

- ind prob: returns true if the problem *prob* is indivisible.
- solve prob: solves an indivisible instance of a problem *prob*.
- divide prob: divides the problem *prob* into sub-problems.
- combine prob sols: combines the solutions *sols*.

Rabhi's recursively partitioned skeleton was implemented using **distributed graph reduction** [131]. However, no performance model was developed for this skeleton.

Darlington et al [45] have presented a HOF for the divide and conquer (DC) skeleton. Similar to the recursively partitioned skeleton, the idea is to split larger tasks into sub-tasks and combine the results obtained by solving the sub-tasks independently. Their HOF for DC is as follows:

$$DC :: (\alpha \rightarrow Bool) \rightarrow (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow [\alpha]) \rightarrow ([\beta] \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$$

```
DC test solve divide combine x
    | test x = solve x
    | not (test x) = (combine .  map (DC test solve divide combine)
                   .  divide) x
```

where:

- test x: tests if a problem x is trivial.
- solve x: solves a problem x.
- divide x: divides a larger task x into sub-tasks.
- combine ((map DC ...) x): combines the sub-results to produce the main result.

The DC skeleton was implemented using the functional language $Hope^+$ as the source language and using C as the target language. The CS Tools library was linked to C code to provide the communication layer on two platforms: a Meiko Transputer Surface and a Fujitsu AP1000. They also developed a cost model for their DC skeleton. Their general model needs to be instantiated with average communication and computation costs measured from the target implementation. The DC performance model was not developed further by instantiating it with measurements taken from the target implementation.

The performance model for DC assumes the processors are organised into a balanced binary tree and all processors will eventually be used as leaves (i.e. intermediate nodes also perform work). The execution time for DC can be predicted using this formula:

$$t_{sol_x} = \sum_{i=0}^{\log(p)-1} (t_{div_{x/2^i}} + t_{setup_{x/2^i}} + t_{comb_{x/2^i}} + t_{comm_{x/2^i}}) + t_{seq_{x/2^{\log p}}}$$

where t_{sol_x} is the time to solve a problem of size x, t_{div_x} is the time to divide a problem of size x, t_{comb_x} is the time to combine the two results, t_{setup_x} and t_{comm_x} are setup and transmission time for communication and t_{seq_x} is the time to solve a problem of size x sequentially.

Cole [40] has also presented a HOF for his fixed degree divide and conquer (FDDC) skeleton. This is a special form of the general divide and conquer skeleton where the number of sub-problems to be divided are fixed in advance.

$$D_C : (prob \rightarrow bool) \rightarrow (prob \rightarrow [prob]) \rightarrow ([sol] \rightarrow sol) \rightarrow (prob \rightarrow sol) \rightarrow (prob \rightarrow sol)$$

```
D_C indivisible split join f = F
         where F P = f P, if indivisible P
                 = join_k (map F (split_k P)), otherwise
```

Cole [40] did not have a parallel implementation for the FDDC HOF shown in Section 2.2.1 but discussed an idealised implementation. In this case, a k-ary tree of processes is mapped onto a k- ary tree of processors. The root processor will divide the original problem into k sub-divisions. This sub-division process will carry on recursively until no more sub-divisions are possible. When this happens, the leaf processors will compute a base function f. Finally, the results will be passed up the tree to the root processor which will compute the result for the whole problem.

Asymptotic performance analysis was the technique used to generate a formulae to predict the execution time this skeleton by describing the time complexity of the H-tree [1] implementation of the binary FDDC skeleton:

$$T_{P2}(n) = 4(\sqrt{n} - 1) + f(1) - 2\log n + \sum_{i=1}^{\log n} [s(2^i) + j(2^i) + p(2^{i-1}) + r(2^{i-1})]$$

When the depth $\log n$ is odd the first term in the above formulae is replaced by $3\sqrt{(2n)} - 4$. T_{P2} is the parallel run-time for a binary tree implementation, $s(n)$ is the time to split a problem of grain n into two sub-problems of grain $\frac{n}{2}$, $p(n)$, $r(n)$ are the number of constant size packets of information required to describe a problem instance and result of a problem of grain n respectively, $j(n)$ is the time to join two sub-results of grain $\frac{n}{2}$ to form one result of grain n and $f(1)$ is the time needed to evaluate the base case function on an indivisible instance. The above model is the summation of three phases: 1) splitting and distributing the problem, 2) computing the base case at the leaves and 3) returning and combining

[1] The H-tree implementation is a well known layout of a complete binary tree on a grid of processors [1].

the results. While the above performance model was not verified on any target architecture but it inspired other researchers in the field to work on performance models for skeletons.

Feldcamp, Sreekantaswamy, Wagner and Chanson [64] did not present a HOF for the DC skeleton they developed. Their skeleton was integrated in the program development environment Parsec [65] and implemented as a binary tree on a Transputer network. This version of DC is similar to **Cole's** FDDC skeleton except that internal processors of the tree topology also perform work.

The cost model for the DC skeleton which was developed by Sreekantaswamy [156] for speedup with startup cost taken into account is as follows:

$$SP(N) = \frac{MT_e}{T_{startup} + \frac{M-1}{S_N}}$$

where steady state throughput is

$$S_i = S_{i-1}\left(\frac{T_e(i) + \beta_e - T_s(i) - T_j(i) - \beta_f}{T_e(i) + \beta_e}\right) + \frac{1}{T_e(i) + \beta_e}$$

and startup time is

$$T_{startup} = \sum_{i=2}^{N}(2T_\tau(i) + T_s(i) + T_j(i) + \beta_f) + T_e(1) + \beta_e$$

M is the number of tasks, $T_s(i)$ is the CPU time for splitting a task at level i, $T_j(i)$ is CPU time for joining tasks at level i, $T_e(i)$ is CPU time for executing a task at level i, $T_\tau(i)$ is the transfer time of a task at level i, β_e and β_f are the CPU overhead for processing a task locally and forwarding a task, respectively. For example, β_f on a Transputer could be the CPU overhead resulting from the link setup time which cannot be overlapped with computation. The startup time is the time needed to return the first result which is affected by the scheduling strategy. The startup time can be neglected in cases where there are large number of tasks and in turn large execution time, otherwise it is necessary to consider it. The steady state throughput S_N of the complete tree can be calculated by recursively evaluating the S_i's up to level N. This performance model differs from other models as it includes the CPU overhead resulting from non-overlapped start-up cost of communication. The cost model was validated on a 74 node T800 Transputer based multicomputer system. The difference between the measured speedup and predicted speedup by the model was claimed not to exceed 5%.

Campbell [30] has developed a model for parallel computation called CLUMPS. CLUMPS stands for **Campbell's Lenient, Unified Model for Parallel Systems** which is composed of an abstract machine with an associated cost model. The modeling of DC under CLUMPS which is based on asymptotic analysis:

$$O(\left(\frac{N}{P}SOLVE\right) + \left(\log P + 2\frac{N}{P} - 1\right)TEST + \left(\log P + \frac{N}{P} - 1\right)(ORDIVIDE$$
$$+ANDCOMBINE) + f(\log P, d(N)) + f(logP, c(N)))$$

N is the problem size, P is the number of processors, $SOLVE$ is the cost of solving an indivisible problem of size 1, $TEST$ is the cost of testing whether a problem is indivisible, $ORDIVIDE$ is the cost of dividing a problem into 2 sub-problems, $ANDCOMBINE$ is the cost of combining 2 sub-solutions to sub-solutions, $d(N)$ is the number of communication packets required to represent a problem of size N, $c(N)$ is the number of communication packets required to represent a solution of size N, $f(\log P, d(N))$ is the communication cost and $f(\log P, c(N))$ is the communication cost for re-combination.

2.2.2 Task Queue

Algorithms that belong to the **task queue** skeleton have workers in a work pool where each worker will check if tasks are available and then request the next available one. Next, the workers will execute the tasks which might result in creating other tasks. The generated tasks will be added to the queue. The algorithm terminates when no more tasks are available and all processors are inactive. When tasks are executed they access a shared data structure that represents the problem to be solved. The task queue can have a number of queuing disciplines; stack, FIFO queue, unordered heap and strictly ordered queue. However, no parallel implementation was reported, as far as we know, for the task queue skeleton.

Rabhi [131] gave a HOF which describes his **Distributed Independent** skeleton which fits in this category. The HOF is:

```
di$ f n divide combine p = combine (map f (divide n p))
```

where f is the function executed by the slaves, n is the number of slaves, divide is a function that specifies how work is divided among the slaves and combine is a function that combines the results on the host. Note that the number of subdivisions is defined by the programmer. It was implemented in a similar way to the **recursively partitioned** skeleton discussed previously and no performance model was reported.

Darlington et al. [45] have presented a HOF for the **process farm** skeleton which is a special form of the **task queue** skeleton. The process farm skeleton represents simple data-parallelism. A function is applied to a list of data. Parallelism arises by utilising multiple processors to apply the function to different parts of the list. Their HOF is as follows:

$$FARM :: (\alpha \to \beta \to \gamma) \to \alpha \to ([\beta] \to [\gamma])$$

```
FARM f env = map (f env)
```

Darlington et al's [45] **FARM** skeleton was implemented in a similar way to their DC skeleton discussed in Section 2.2.1. The FARM consists of two major parts: 1) a master processor and 2) a worker processor(s). The master processor has a pool of work packets that need to be processed. Therefore, it distributes them to workers. Once the worker receives a work packet it will process it to compute the result. Next, it sends the result back to the master processor. The work finishes when the master processor receives back all the results. In summary, the FARM work using a "distribute-work-collect" phase. The number of steps needed to complete the work depends on the number of work packet and number of workers.

In the beginning, a general cost model was developed for the FARM skeleton:

$$t_{farm} = t_s + R(t_e + 2t_c)$$

where t_s is the skeleton startup overhead, R is the number of steps required to process the work pool which is simply $\lceil \frac{N}{P} \rceil$, P is the number of workers and N is the number of work packets, t_e is the required time to solve a work packet and t_c is the communication time required to receive a result or send the data.

The above model was developed further by instantiating it with measurements taken from a Transputer implementation. The machine parameters used to instantiate the general cost model for the FARM skeleton were: 1) average execution time of instructions on each of the processors, 2) number of processors used by the skeleton and 3) communication costs of the machine.

The communication time t_c can be refined in terms of the communication startup time to send or receive one byte of data t_b. In this case $t_c = t_o + t_b b$ where b is the number of bytes to transfer. The final performance model for the process farm was as follows:

$$\begin{aligned}
T_{farm} &= T_{setup} + R(T_{comms} + T_{slave}) + T_{gc} \\
\\
\textit{where} & \\
T_{setup} &= I_o + I_1 P \\
R &= \left\lceil \frac{N_w}{g(P-1)} \right\rceil \\
T_{comms} &= 2T_o + g(d+r)T_b + g(2T_{fo} + T_f(r+d)) \\
T_{slave} &= gT_w \\
T_{gc} &= \left\lfloor W_s g G \left\lceil \frac{N_w}{g(P-1)} \right\rceil \right\rfloor T_g
\end{aligned}$$

T_{setup}, is the setup time taken to allocate processors to a skeleton can be determined by benchmarking once for each skeleton. The constants I_o and I_1 depend on the initialisation protocol used. The number of rounds, R, depends on the number of work packets N_w, the clustering unit g and number of processors P. The communication time, T_{comms}, can be refined in terms of the communication startup overhead and the time taken to send one byte on the network between two processors denoted by T_o and T_b respectively. The process of *flattening*[2] and *unflattening*[3] the data structures used by $Hope^+$ imposed further overheads on communication. The two parameters for this overhead are T_{fo}, the flatten startup overhead, and T_f, the time to flatten, which are measured by benchmarking. The variables d and r represent the size of the distributed work packets and the

[2] Converting data structures to byte streams.
[3] Converting byte streams to data structures.

corresponding results, respectively. The time to process a work packet on any of the slaves, T_{slave} is the result of multiplying the clustering unit, g, by the time required to process a work packet, T_w. An additional overhead is added to the above formulae which results from garbage collection, T_{gc}. This overhead depends on the amount of work done for each packet, W_s, the frequency at which garbage collection occurs, G, and the average time taken for a call to the collector T_g. Experimental results showed very good accuracy of the performance model in predicting the FARM behaviour for the Transputer implementation.

Sreekantaswamy [156] has also developed a cost model for the processor farm skeleton. In the model it was assumed that tasks do not split, processors choose to execute the task or pass it to a child processor and for all processors i, $T_e(i) = T_e$, where T_e is the processing time of the task. The cost model for the speedup with startup costs taken into account is as follows:

$$SP(N) = \frac{MT_e(1 - a^N)}{(M-1)(T_e + \beta_e) - k(T_e + \beta_e - \beta_f) + T_{startup}(1 - a^N)}$$

where

$$a = k(T_e + \beta_e - \beta_f)/(T_e + \beta_e),$$

steady-state throughput:

$$S(N) = \frac{1}{(T_e + \beta_e) - k(T_e + \beta_e - \beta_f)} \left[1 - \left(\frac{k(T_e + \beta_e - \beta_f}{T_e + \beta_e} \right)^N \right]$$

and start-up time:

$$T_{startup} = (N-1)(2T_\tau + \beta_f) + T_e + \beta_e$$

The variables in the above formulae have the same meaning as the performance model for the divide and conquer skeleton discussed in Section 2.2.1. The model assumes the processor farm structured as an N level balanced k-ary tree executing M tasks with average processing time T_e. The startup time for this skeleton is simply the time it takes to forward the first task to the Nth level of the tree and return the computed result. Note that the startup overhead for forwarding a task, β_f, was subtracted from the $(T_e + \beta_e)$ in the steady state throughput formulae as it was already added in the startup formulae for (N-1) processors. The performance model was validated on a 74 node T800 Transputer based multicomputer system. The difference between the measured speedup and the predicted speedup by the model was claimed to not exceed 5%.

Bratvold's [20] **FARM** skeleton was implemented as farming on a linear pipeline of processors. He also has developed a cost model for it. The completion time for the skeleton was given as follows:

$$T_C = T_{start} + T_{run} + T_{close}$$

$$where$$

$$
\begin{aligned}
T_{start} + T_{close} &= P_w * Farm_{startclose\_add} + Farm_{startclose\_base} \\
P_w &= P - 1 \\
T_{run} &= max(T_{run1}, T_{runP_w}) \\
T_{run1} &= (n - d) * (t_f + t_k) + (N - (n - d)) * t_p \\
T_{runP_w} &= (n + d) * (t_f + t_k) \\
d &= \frac{t_p * (N - n)}{2 * t_f + 2 * t_k - t_p} \\
t_p &= t_{ep} + t_{cp} \\
t_k &= t_{ek} + t_{ck}
\end{aligned}
$$

T_{start} and T_{close} are the time when not all workers are busy, T_{run} is the time when all workers are busy working, P_w is the number of workers $(P - 1)$, $Farm_{startclose\_base}$ is a fixed cost when some workers are idle, $Farm_{startclose\_add}$ is the additional overhead cost for all workers in the $FARM$, T_{run1} and T_{runP_w} are the total computation times for the first and last workers respectively, n is the average number of elements that each worker gets which is equal to $\frac{N}{P_w}$, d is a constant, t_f is the execution time for function f, t_p is the time for passing the data elements to other workers and results towards the farmer, t_k is the time for receiving elements and keeping them in local memory and sending the results from local memory to the farmer, t_{ep}, t_{cp}, t_{ek} and t_{ck} are the times resulting from communication overhead. The validation of the performance model for the Transputer implementation had an accuracy of around 90%.

The model was designed specifically for the Transputer implementation and may not work on other parallel architectures. Also, the cost model was for a linear list process farm.

Busvine's [29, 28] process farm was implemented as a tree with the farmer as root. It is based on extracting farm parallelism from linearly recursive functions as linear recursion is a principal and frequently used control structure in functional languages. The cost model for the FARM skeleton is as follows:

$$
\begin{gathered}
T_{Farm} = max(N * (T_{Rec} + K_{Rec} + T_{Comms}) + T_{Base}, \\
(N/W) * (T_{Task} + K_{Worker} + K_{Tree} * T_{Comms})) \\
+ N * (T_{Combine} + K_{Combine}) + K^{W}_{Setup}
\end{gathered}
$$

where T_{Rec} is the average time for the farmer to generate a task for each level in the recursion, K_{Rec} is the time to store tasks for later forwarding to workers and saving the results of evaluated tasks received from the workers, T_{Comms} is the time for communicating one task and its result, T_{Base} is the time for evaluating a single base case, T_{Task} is the average time for task evaluation, K_{Worker} is a constant time for unpacking tasks and packing results ready for communication, K_{Tree} is a constant that depends on the shape of the worker pool, $T_{Combine}$ is the time for combining the evaluated tasks, $K_{Combine}$ is the time for extracting an evaluated task, K^{W}_{Setup} is the time for starting up and shutting down the worker pool, N is the depth of recursion and W is the number of workers.

Values for the constants K_{Rec}, K_{Worker}, K_{Tree} and $K_{Combine}$ were tuned by fitting this equation for T_{Farm} against measured running times.

From the parameters used in the model like T_{Rec} and N, we can note that the model was used mainly to determine whether a linear recursive function should be implemented as a process farm. Therefore, the model has a limitation in predicting the performance for general uses of the process farm skeleton.

2.2.3 Systolic

The **systolic** skeleton (which is the general form of pipeline[4] like skeletons) consists of a number of stages where parallelism is exploited by operating on a flow of data that passes through the various stages of the pipeline.

Rabhi [131] has presented a HOF for the **Process Networks** skeleton which is:

```
pipe$ fs = (foldr (.)  id fs)
```

where the (.) operator is the composition function and id is the identity function and fs is a list of functions. Its implementation is similar to other skeletons developed by the same researcher. However, no performance model was reported for this skeleton.

Darlington et al [46] have presented a HOF for the pipeline skeleton which is:

$$PIPE :: [[\alpha] \to [\alpha]] \to [\alpha] \to [\alpha]$$

```
PIPE = foldr1 ( o )
```

It was developed as linear pipeline of processes specified as a list of functions. The cost model for the pipeline skeleton is:

$$t_{pipe} = t_s + (t_e v + t_c)(p + n - 1)$$

This resembles a general model for the pipe skeleton which predicts the total execution time of a particular instance of the pipe skeleton. For this model, t_s is the startup time, t_c is the execution time of a stage for a single element, t_c is the communication time between stages, p is the number of stages and n is the number of elements in the list.

Similar to the process farm skeleton, the pipe cost model was further instantiated with measurements taken from the Transputer implementation.

$$T_{pipe} = T_{setup} + T_{fill} + (R-1)(T_{prc} + T_{comms}) + T_{gc}$$

$$\textit{where} \quad R = \left\lceil \frac{N_w}{g} \right\rceil$$

[4]Sometimes called pipe.

$$
\begin{aligned}
T_{prc} &= gT_w \left\lceil \frac{N_p}{P-1} \right\rceil \\
T_{fill} &= P(T_o + dT_b) + (P-1)(2(T_{fo} + dT_f) + T_{prc}) \\
T_{comms} &= T_o + dT_b + 2(T_{fo} + dT_f)
\end{aligned}
$$

For this model, N_w is the total number of work packets, T_w is the time to process one packet, g is the clustering unit, d is the number of bytes transferred between stages for each packet and N_p is the total number of virtual stages [5] which is independent of the number of processors. T_{prc} is the processing time, T_{comms} is the communication time, T_{fill} is the time to fill the pipe, T_{gc} is the time for garbage collection and T_{setup} is the setup time. The experimental results showed reasonable predictions for the pipe skeleton. In fact, they were less accurate compared with the FARM skeleton. This was due to the poor modeling of the garbage collector.

Bratvold [20] has also developed a cost model for the pipeline skeleton. However, he assumed a pipeline of s stages where each stage is a skeleton. The total latency[6] of the skeleton was given by:

$$L = \sum_{i=1}^{s}(L_i + c_i)$$

where L_i is the latency of stage i, and c_i is the time to communicate one data element from stage i to stage $i+1$ assuming $c_s = 0$ as s is the last stage. The completion time is given by:

$$T_C = (max_{i=1}^{s} T_{C_i}) + L - L_{max}$$

where each stage i has a completion time of T_{C_i} and L_{max} is latency of the stage with highest T_{C_i}.

2.2.4 Application-based Skeletons

Skeletons in this category were primarily developed for use in solving particular problems. Normally, program developers who use such skeletons are experts in the application area and domain specific knowledge is required for using such skeletons. Some of them are outlined next.

Deldarie et al [53, 52] have developed special cases of skeletons related to image processing. One skeleton called **local window** (LW) which transforms an image into another image. Each pixel in the resulting output is derived from pixels in a window surrounding the corresponding pixel in the input image. Another skeleton called **split and merge** (SAM) where an image is partitioned into slices, an operation is applied to each slice then the results are merged. This skeleton resembles the divide and conquer skeleton.

[5] In an ideal pipe it is assumed that one processor was allocated to each process. However, this is not always possible due to limited resources. Therefore, a number of virtual stages, need to be allocated to a real stage.

[6] The latency of a skeleton is the time taken from the first of the input list is fed into a skeleton until the first result element appears.

The overall time complexity for the LW skeleton is:

$$O(\log^2 P + t_w n^2/P)$$

Where $O(\log^2 P)$ is the cost for forking and joining P processes, t_w is the cost for a single window function, n^2 is the number of pixels in image and P is the number of processors. For the SAM skeleton the overall time complexity is:

$$O(\log^3 P + n^2/P + t_s(n^2/P) + \sum_{i=1}^{\log P} (I_i + t_m(I_i) + J_i))$$

where $\log^3 P$ is the result of multiplying $\log P$ merge states with the cost for barrier synchronisation costing $\log^2 P$, t_s is the cost for solving a slice of n^2/P pixels, t_m is the cost of merge, I_i is the number of bytes copied from global memory and J_i is the number of resulting bytes from the merge to be copied to global memory.

The execution time for the SAM and LW skeletons was presented as a generic higher order complexity function. Also, the models were extended to cover scalability analysis using the **isoefficiency function** [77].

Further work by **Davy et al** [50] resulted in developing skeletons for **Constructive Solid Geometry** (CSG). The skeleton was integrated in a prototype library called **Geometric Evaluation Library** (GEL).

Schwarz and **Rabhi** [145, 146] have developed a skeleton for coarse grained data parallel algorithms with iterative control structure. It is called the **Static Iterative Transformation** (SIT) skeleton. They used a specialised form of distributed graph reduction for the parallel implementation. The skeleton can be used to solve problems in numerical analysis, image processing, parallel branch and bound, and graph theory. An example on how to produce an executable specification of Laplace's problem was given.

Wallace, **Michaelson**, **Scaife** and **Austin** [165, 144] have developed four skeletons for low-level image processing and intermediate-level computer vision. The four skeletons were related to a common divide and conquer HOF. The skeletons were **geometric decomposition** (GD), **data farming** (DF), **task farming** and **geometric decomposition with interworker communication** (GDIW). The skeletons differ in how the image structure is partitioned and the amount of communication between the processors. Each skeleton is used for different cases like equal processing load, irregular data, resolving boundary conflicts and processing the received results again.

2.2.5 Other Skeletons

There are other skeletons that were used within skeleton programming systems and are briefly described here. Their description is taken from [6, 20, 76].

The **fold** skeleton, which is sometime called **reduce**, combines the elements of a list using a binary associative operator. In fact this is one of the skeletons used in the parallel programming model presented in this thesis and our description of this skeleton is presented in Section 3.2.2.1.

The **scan** skeleton, which is sometimes called **parallel prefix**, computes the list of running totals with associative operator.

The **map** skeleton, which is we think is just another name for the process farm skeletons, applies a function to every element in its list data structure. This skeleton is described in more detail in Section 3.2.2.1.

The **filter** skeleton, which is similar to the map skeleton, applies a given function to an element of its list data structure if this element satisfies a certain condition. It is normally implemented in a similar way to the map skeleton.

2.3 Skeleton Programming Models and Systems

This is the second part of the survey which looks at abstract models and concrete systems for skeleton-oriented parallel programming.

2.3.1 BMF

The Bird-Meertens Formalism **BMF** [13] s a notation for describing functions with a calculus for manipulating function descriptions. The idea of BMF is to derive programs from problem specifications. Most of the work on BMF has concentrated on lists. A theory in BMF describes the behaviour of a data type, and provides a set of operators on that type and transformational laws for the operators. For example, operators for the list type include: map (written $\star$), filter (written $\triangleleft$) and reduce (written $/$). They can be used in the following way

$$\begin{aligned} f \star [a, b, c, ..., z] &= [f a, f b, f c, ..., f z] && \text{(map with f)} \\ \oplus/[a, b, c, ..., z] &= a \oplus b \oplus c \oplus ... \oplus z && \text{(reduce with } \oplus\text{)} \end{aligned}$$

while

$$p \triangleleft [a, b, c, ..., z] \qquad\qquad \text{(filter with } p\text{)}$$

where f is a unary function and $\oplus$ is a binary function. Some of the transformation laws which can be applied to generate efficient programs include:

$$\begin{aligned} (f \circ g)\star &= (f\star) \circ (g\star) && \text{(map distributivity)} \\ (p\triangleleft) \circ (q\triangleleft) &= (q\triangleleft) \circ (p\triangleleft) && \text{(commutativity of filters)} \end{aligned}$$

Skillicorn et al [151, 152] have supported the use of BMF for parallel programming as it abstracts from parallel architectures which allows portability of programs, and its operators like map and reduce contain implicit parallelism. They have defined a cost calculus to be a cost system with two properties: (1) the cost of a program depends on the cost of its pieces, i.e. compositional and (2) the cost of transformation can be associated with its rule.

The cost calculus that they have developed for BMF integrates cost information with equational rules. This feature adds the possibility of deciding the direction in which an equational substitution is cost-reducing. The cost calculus will be used to guide the application of rules, in this case for BMF transformational rules for transforming one parallel structure into another in an attempt to find a cost-effective implementation.

The cost of implementing the **map** function in parallel is:

$$\begin{aligned} t_n(f\star_n) &= t_n(PAR_n f) \\ &= t_1(f) \end{aligned}$$

where $t_n(f)$ is the cost of executing f on n processors and $PAR_n f$ denoted n parallel applications of f.

The cost of a parallel reduction with constant-sized partial results is:

$$\begin{aligned} t_n(\oplus/_n) &= t_n(PARIMPL_n(\oplus/)) \\ &= \log n \end{aligned}$$

where $PARIMPL_n(\oplus/)$ is the parallel implementation of $\oplus/$ using n processors.

After finding the costs of the basic operations, the cost information is used to label BMF identities. For example, we have $(f \circ g)\star = f\star \circ g\star$ as a BMF identity. To label it we need to find the costs of the left hand side and the right hand side. For this identity both sides do have equal costs (cost-neutral). Therefore, we can label it as:

$$(f \circ g)\star \rightleftharpoons f\star \circ g\star$$

Other identities were labeled in the same way and used to guide transformations for cost-effective implementations.

2.3.2 HOPP

R. Rangaswami's HOPP [136] is a parallel programming model for skeleton-oriented programming. HOPP stands for Higher-Order Parallel Programming. The model consists of three parts: the **program** model, the **machine** model and the **cost** model. The program model is a composition of nested instantiations of recognised and user-defined functions [7]. Each stage in the pipeline resulting from the composition is referred to a **phase** of the program. The machine model targets architectures for the programs which include hypercube, 2-D torus, linear array and tree. The cost model determines the cost of executing a recognised function on a given architecture.

A HOPP program could comprise of one or more composed **phase(s)** where parallelism is exploited only within each phase. The parallelism exploited in each phase depends on the number of recognised functions in the phase itself.

[7] All of the recognised functions work on lists.

If there are n recognised functions in a given phase then it is possible to have $2^n - 1$ different parallel implementations for that phase. However, parallelism was exploited only for the first three levels of recognised functions and any recognised functions below the first three was implemented sequentially. Code generation was done manually using the support of a library of functions. The code for the parallel implementations was written in C, with CSTools [111] being used as the communication harness. The target architecture is the Meiko Computing Surface.

The cost models for the recognised functions were used to statically analyse a given HOPP program to determine the best cost-effective parallel implementation for a given architecture. The theoretical cost models were implemented as an analyser. In general, the cost of a program comprising of n *phases* is given by:

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

where C_{pi} is the cost of phase i and $C_{i,i+1}$ is the communication cost in case there is a need for rearranging the output of phase i to suit the implementation of phase $i+1$.

In the implementation scheme, five modes of communication were identified for rearranging the data between the phases:

- Nearest Neighbor Communication: in this operation a processor sends a message to another processor which has a direct link with it.
- Broadcast Operation: in this operation a processor sends a message to all the other processors in the network.
- Scatter Operation: in this operation a processor will move different messages to all the processors in the network.
- Gather Operation: in this operation a processor will collect the distributed data packets across the processors in the network.
- Total Exchange Operation: in this operation data packets from every processor will be moved to every other processor in the network.

For the skeletons **fold** and **scan** there were static and growing versions. For the static version, the emerging result remains constant in every step. However, for the growing version the size of the emerging result grows by an amount equal to the size of each list element.

2.3.3 To's Approach

H. W. To [160] has looked into optimising combinations of algorithmic skeletons. A language for combining skeletons was proposed and a set of **primitive skeletons** was chosen which is based on the operators of parallel abstract data types

(PADTs)[8]. This choice was based on the observation that many highly parallel applications exploit parallelism from a large data structure. Two PADTs were described: restricted lists (**rlist**s) and **arrays**. A number of combining skeletons were proposed where each skeleton captures a common pattern of control flow. For example, the compose ($\circ$) function [9] takes the output of one skeleton and pass it as the input of another skeleton. Another two combining skeletons are **iterateFor** and **iterateUntil** which actually extend composition where the same skeleton might iterate over a data structure. Also it is possible to combine binary skeletons using the **compose-2** ($\circ\circ$) function. In cases where it is necessary to apply more than one skeleton to an instance of a data structure the **split** ($\triangle$) skeleton could be used.

For each primitive skeleton two kinds of data-level parallelism were exploited: data distributed and data pipelined.

For data distributed array primitives two distribution patterns were studied: **blocked** and **cyclic**. An example will explain these techniques. Suppose the distribution of an array on p processing elements (PEs). In a **blocked** distribution each processor will get a contiguous part of the original array. In a **cyclic** distribution, elements of the array will be distributed in a round robin style across the PEs. The idea of having two distribution is to a give trade-of between load balancing and communication overhead.

For data pipelined array primitives an implementation of a general pipeline stage instantiated for each skeleton was also discussed. However, the methodology involves transforming the flat data structure into a nested one. This causes an additional overhead which is a disadvantage of the methodology. In principle, each stage can be instantiated with a skeleton.

2.3.4 P3L

P^{3}L (Pisa Parallel Programming Language) [62, 7, 125, 126, 127] is an explicit parallel programming language developed the University of Pisa and the Pisa Science Center/Hewlett-Packard Laboratories. The project aims at exploiting massive parallelism for distributed memory MIMD machines. The compiler of P^3L consists of three parts. The first part, front-end, translates the source code to the intermediate form. The second part, middle-end, performs a number of optimisations on the construct tree and generates an architecture independent data structure representing the architecture dependent implementation of the source program. The last part, back-end, generates the actual code for the target machine using libraries of macros modeling the typical processes used for the implementation of the constructs. Also the compiler consults performance models to optimise the parallel program for a given target architecture.

P^{3}L has two components: a sequential part used for writing algorithmic code in an imperative language (C++), and a set of constructs for structuring parallelism. The set of constructs supported includes pipeline parallelism, process

[8] A PADT is an aggregate data structure and a set of parallel operators with which to manipulate it.

[9] Note that we are talking about composing skeletons here and not functions.

farm, tree reduction and geometric parallelism. Programs are expressed as hierarchical composition of constructs, each of which may again contain further constructs. However, it is the responsibility of the programmer to explicitly craft the skeletons together and specify the flow of data between the constructs.

2.3.5 PUFF

Busvine's PUFF (Parallelisation Using Farmed Functions) [29] is a compiler that exploits processor farm parallelism from linearly recursive functions in SML programs. The processor farm skeleton used in this system was previously described along with its performance model in Section 2.2.2. The PUFF compiler relies on profiling information and performance modeling to determine which linear recursive functions should be implemented as processor farms. The compiler can generate Occam2 code from a pure functional subset of SML targeting the Meiko Computing Surface using T800. No support for nesting the processor farm skeleton is allowed in this system.

2.3.6 SkelML

Bratvold's SkelML [19, 20] is a skeleton-based prototype compiler for ML. The compiler exploits the parallelism available in the program through a set of predefined skeletons. The skeletons are expressed as HOFs over lists in the source program. The skeletons are: **map**, **filter**, **fold**, **filtermap**, **mapfilter** and **foldmap**, where the last three being combinations of the first three. Also pipeline parallelism is supported by composing the skeletons. However, skeletons cannot nest themselves which is a major limitation in this work.

The compiler supports a profiler based on structural operational semantics, and uses performance models to predict execution times of program components. However, the programmer is responsible for providing the system with sample data sets. Also, the system is capable of optimising the program by performing a number of transformations. The compiler generates Occam2 at the back-end (as it uses the PUFF compiler) and targets the Meiko Computing Surface with T800. The novelty in the SkelML compiler is in integrating all of the above components in one system.

2.3.7 Gorlatch et al's Approach

Gorlatch et al [75, 76, 74] have looked at transformation rules for HOFs taken from BMF. A general schema was given for transforming the functional specification into a parallel functional implementation. Nesting of HOFs was handled by special transformations. For example, `map (map h) =` map^2 `h` where map^2 is called double map and maps a function over a list of lists. Also, a function called **flat** was provided which flattens a list of lists, i.e., it eliminates all inner brackets.

However, nesting is handled only if there exists a transformation rule which transforms nested HOFs into a flat implementation. This would restrict the expressivenes of the model and it is very difficult to have transformations for all HOFs combinations up to arbitrary depth.

2.3.8 Skil

Skil (Skeleton Imperative Language) [17, 16] is an imperative language enhanced with higher-order functions and currying. It aims at high level programming with an efficient implementation. Therefore, it integrates algorithmic skeletons which work with distributed arrays. Therefore, the system targets numerical algorithms like multi-grid algorithms. However, Skil does not support nested parallelism as the emphasis was placed on the efficiency of the implementations and on the integration of the functional and imperative features, rather than on the composition of skeletons.

2.3.9 ALWAN

ALWAN (A Language With A Name) [24, 79] is a parallel language and programming environment developed at the Parallel laboratory of the University of Basel (PUB). It is based on Modula-2 and (C or Fortran 77) programs are built using (sequential) calculational and (parallel) coordination parts. Therefore, the language provides the programmer with high level constructs for the description of parallel coordination aspects such as data partitioning and distribution, process topology management and communication aspects. The compiler, with support from the libraries, processes the coordination skeletons and sequential building blocks and automatically generates programs for various parallel architectures. The libraries were implemented using MPI, PVM, CMMD and NX and the system targets CM5, SP1, Paragon and a workstation cluster of NeXT and Sun workstations.

It is not clear if the system allows nesting of skeletons. However, if it supports nesting then the programmer has to use the coordination skeletons to coordinate the parallel computation, which is very explicit.

2.3.10 HDC

HDC [89] is a subset of Haskell which integrates skeletons with a predefined efficient parallel implementation into a functional program. HDC focuses on the divide-and-conquer paradigm. Also a number of skeletons are supported like **map**, **reduce**, **scan** and **filter**. Other skeletons are introduced for improved efficiency but we think that introducing specialised skeletons make the programming model more difficult to use.

Although skeleton composition is claimed, it is not clear how this is done and no example programs are given to demonstrate composition of skeletons. Also, only the map skeleton is implemented and the work is still under construction.

2.3.11 POPE and SITSS

POPE (a "Paradigm-Oriented" parallel Programming Environment) was developed by **Rabhi** and **Schwarz** [132, 133]. The system uses the (Static Iterative Transformation) SIT skeleton, which exploits coarse grain data parallelism. With the aid of the programmer, the system transforms an instantiated skeleton into a Haskell program with annotations for controlling parallel graph reduction.

The work of **Parsons** and **Rabhi** [124] resulted in developing a portable parallel programming system called the **Static Iterative Transformation Specification System** (SITSS) which is based on the SIT skeleton. The system is capable of generating code for a number of target parallel systems and uses a communication cost model to select the best data partitioning strategy. However the system can solve problems that only fit the SIT skeleton.

2.3.12 TPascal

TPascal [23] is a programming language designed for programming MIMD computers with distributed memory in a task parallel way using explicit message passing. The language consists of two parts. The first one, the host language which is similar to Pascal, is the sequential programming language. The second one is the coordination language. The coordination language consists of constructs to set up parallel processes and to exchange data between them.

TPascal uses the concept of **topologies** which are sets of processes arranged in a specific way according to the needs of program. The use of topologies is quite similar to the use of skeletons in functional languages. Among the available topologies are pipes, rings, farms and trees of processes. A key feature in this system is the nesting of topologies to construct large collection of processes. However, the programmer is responsible to explicitly craft the topologies together and how topologies exchange data between them. We think this model of parallel programming is very similar to P^3L.

2.3.13 SCL

Darlington el al's Structured Coordination Language (SCL) [171, 49, 47] has two components: the computational model (which is a single threaded computation) and the coordination model (the glue that binds separate activities into a union). This model is between the coordination and the skeleton approaches for parallel programming.

SCL introduces three kinds of skeleton: configuration (which abstract commonly used patterns of basic data structures for programming), elementary (which define the basic data parallel operations upon which the parallelism in the current implementation is founded such as map, fold, scan, etc) and computational (which abstract commonly occurring patterns of computation such as iteration) skeletons. It is important to note that SCL is a base language and can be integrated with other host languages. For example, **Fortran-S** [47] was designed by taking the traditional sequential language Fortran as the base language for describing local sequential computation, while functional skeletons in SCL are used for coordinating these local sequential codes.

SCL supports nested parallelism by flattening it using special transformations. This is an explicit approach for parallel programming as it provides the programmer with special constructs to combine basic components.

2.3.14 Michaelson et al's Approach

Michaelson et al [118, 119, 143] have developed a parallelising compiler for Standard ML using algorithmic skeletons. It supports only the **map** and **fold** skeletons. Arbitrary depth nesting of HOFs can be implemented in parallel using a static approach for generating parallel code. However, the experimental results reported for a matrix multiplication program do not show any improvement for the nested implementation compared to the unnested implementation. Therefore, it is difficult to say if their methodology is successful without presenting more examples to explain how nested skeletons behave.

2.3.15 Miscellaneous

There are other systems which included skeletons (or similar ideas) in them such as the work of **Roe** [142], **Preto** [129, 130], **Yamazaki** and **Ando** [170] (they developed a case-based parallel programming system using skeleton in order to reduce the burden of parallel programming by utilising relevant parallel programs), **SkIE** [8] (a commercial analogue of P^3L developed by QSW Ltd.) and **Nesl** [15] (which is described in Section 5.6.2).

2.4 Comparison of Skeleton Programming Models and Systems

The main work done on exploiting skeleton parallelism is summarised in Table 2.4. The notations used in the table for describing the cost models are: CM for cost model, AA for asymptotic analysis and MA for measurement analysis. Asymptotic analysis is a technique used to provide a theoretical study about skeletons which need not to have a parallel implementation. Measurement analysis is used to give a prediction of the execution time for an implemented skeleton. Some of the cost models were verified in order to have a comparison between the real and predicted performance.

Most of the early parallel implementations used CSTools while more recent implementations used more abstract communication models like MPI or PVM. The main target architecture was the Meiko Computing Surface. Other implementations targeted MIMD machines like the Cray T3D, Fujitsu AP1000, Thinking Machines CM-5, IBM SP1 and Intel Paragon. The code generation process was automated for most of the implementations.

Most of the programming models disallowed the nesting of skeletons and where allowed it was restricted. Also the process of detecting skeleton combinations was done manually.

We can conclude that there were a number of approaches for skeleton parallelism where the programmer is provided with operators that have parallel properties to build parallel programs. However, the use of functional languages provide an environment for programming which is more abstract but less efficient than imperative languages. The use of message passing libraries like MPI or PVM increases portability across many parallel machines.

A major point to note is that the exploitation of nested skeleton parallelism is usually either disallowed or restricted. For example, P3L relies on the programmer to craft skeletons manually which is a big burden on the programmer. In HOPP, nesting of skeletons was limited to the first three levels and code had to be generated manually. For Wing To's looked at optimal combinations of skeletons, the skeletons need to be annotated with labels to specify the behavior of the nested skeletons. However, it is not clear if this has to be done by the programmer or automated by the compiler. The work does report the design of a compiler based on behaviour inference [10] but relies on the programmer to annotate the data structures to determine the behavior for each skeleton. Therefore, this annotation process is a burden on the programmer. Also, there is no implementation work to validate the proposed methodology.

2.5 The Importance and Difficulties of Skeleton Combinations

2.5.1 The Importance of Skeleton Combinations

A number of researchers have pointed to the importance of nested skeletons.

Bratvold [20, page 189] notes:

> *Nesting allows parallelism to be exploited at several levels simultaneously, which is certainly an important contributer to exploitation of massive parallelism. By disallowing nested parallelism, we diminish the degree of parallelism exploitable.*

We think that to fully utilise the benefits of skeleton-oriented programming, it is important to exploit parallelism available in nested HOFs as hierarchical structures of skeletons. To the main benefit of nesting skeletons is the exploitation of massive parallelism and achieving good speedups on large number of processors. Also we think the nesting might improve the scalability of skeleton programs, for example, if parallel implementation does not scale well because of a centralised bottleneck like having one process to transmit tasks to other processes.

Many parallel programming models have a restricted number of operators that have implicit parallel properties. If the complexity of the problem to be solved requires more skeletons then it might be possible to combine the available operators according to the needs of the problem.

[10] Behavioral inference is similar to type inference but uses behavior variables rather than type variables

Name	Year	Host Language	CM	CM Verified?	Parallel Implementation	Target Architecture(s)	Code Generation	Skeleton Nesting	Detection of Nesting
Cole	1989	Functional	AA	N/A	Analysis	2D-mesh	No	Disallowed	N/A
P3L	1991	Imperative	MA	Yes	C + CS Tools	Meiko CS T3D & CM-5	Automatic	Restricted	Manual
Darlington	1993	Functional	MA	Yes	C + CS Tools	Meiko CS & AP1000	Automatic	Disallowed	N/A
Parsec	1993	Imperative	MA	Yes	Logical Systems C	Meiko CS	Automatic	Disallowed	N/A
PUFF	1993	Functional	MA	Yes	Occam	Meiko CS	Automatic	Disallowed	N/A
Nesl	1993	Functional	AA	N/A	VCODE	CM-2/CM-5	Automatic	Restricted	Automatic
POPE	1994	Functional	None	N/A	C + PVM	Distributed Graph Reduction	Automatic	Disallowed	N/A
SkelML	1994	Functional	MA	Yes	Occam	Meiko CS	Automatic	Disallowed	N/A
CLUMPS	1994	???	AA	N/A	???	Abstract Machine	???	Disallowed	N/A
Deldarie	1995	Imperative	AA	N/A	C	WPRAM Simulator	Automatic	Disallowed	N/A
Wing To	1995	Functional	MA	Yes	N/A	N/A	N/A	Restricted	Manual
SCL	1995	Imperative	MA	Yes	Fortran77 + MPI	AP1000	Automatic	Restricted	Manual
Gorlatch	1995	Functional	AA	N/A	CS Tools	Parsytec T805	Automatic	Restricted	Automatic
HOPP	1996	Functional	MA	Yes	C + CS Tools	Meiko CS	Manual	Restricted	Manual
Skil	1996	Imperative	AA	N/A	Parallel C	Parsytec with T800	Automatic	Disallowed	N/A
ALWAN	1996	Imperative	MA	Yes	C + Fortran MPI + PVM	CM-5, SP1, Paragon Sun Workstations	Automatic	Disallowed	N/A
TPascal	1996	Imperative	None	N/A	???	MIMD	Automatic	Restricted	Manual
HDC	1999	Functional	None	N/A	Parallel C	Sun Workstations	Automatic	Restricted	???
Michaelson	1999	Functional	None	N/A	C + MPI	AP3000	Automatic	Allowed	Automatic

Table 2.1: A comparison between different skeleton-oriented parallel programming models.

2.5.2 Difficulties in Parallelising Combinations of Skeletons

A compile-time problem for not implementing nested skeleton is how to identify skeletons when they are hidden in user defined functions. Coordination system provide explicit constructs to combine skeletons at the language level which we think complicates parallel programming. Therefore, it is important to provide an automated mechanism (possibly through static analysis of the source program) to identify nested skeletons.

Also free variable bindings need to be identified which may result from partially applied HOFs within HOFs. A mechanism needs to be found to enable the run-time system to transmit the free-variables to where they are actually used.

Bratvold [20, page 189] has noted that:

> *Our main reason for not supporting nested parallelism is that it complicates the generation of code for skeleton harnesses.*

Also, the run-time system needs to know which free variables that need to be transmitted. We think they can be achieved by integrating compile-time and run-time techniques.

2.6 Summary

This chapter gave a survey of skeleton-oriented parallel programming. It looked at initial skeletons and complete skeleton models and systems. A detailed comparison of skeletons was given. The importance of combining skeletons and difficulties that arise in their parallel implementation were presented.

The next chapter presents a complete parallel programming model based on algorithmic skeletons which provides a novel framework for compiling and executing arbitrary depth skeleton nesting and composition. Here skeletons can be combined (composed and/or nested) to build parallel applications. The exploitation of skeleton combination is automated by the compiler without any interference from the programmer. Also, the run-time system for this model handles the execution of combined skeletons. The work can be seen as making several contributions to solving the problem of exploiting parallelism in arbitrarily nested and composed skeletons.

Chapter 3

The Compile-Time Framework

3.1 Introduction

This chapter presents the design, organisation and implementation of the parallelising compiler and its environment [83]. The purpose of the compiler is to provide an experimental environment with which to develop a parallel programming framework based on nested algorithmic skeletons. The next section describes the high level languages chosen as the source and intermediate languages for the compiler and Section 3.3 gives an overview of the parallel compiler.

3.2 High Level Languages

At an early stage in the project, it was decided that a high-level language had to be chosen which would provide a parallel programming framework with the following features:

- Gives the programmer a fixed set of primitives that perform general-purpose operation on lists. These primitives are inherently parallel and the programmer uses them to express programs in terms of compositions of these sets of functions along with user-defined functions.

- Abstract away from the target architecture to ensure portability. In other words, the framework should present a level of abstraction which allows focusing on the problem to be solved instead of the hardware platform to be used.

- Hides low-level issues like load balancing, process mapping and inter-process communication from the programmer.

The next sub-sections describes the language **FP** which influenced this work, the source language **Ektran** used to demonstrate the compile-time framework and the intermediate language **Caml** used by the compiler.

3.2.1 FP

FP [9] (a **F**unctional **P**rogramming language) was introduced by Backus in the late 1970s. It was proposed as an alternative functional style of programming to conventional programming languages as they suffered from their step-at-a-time style of programming which was inherited from the von Neumann computer.

The **FP** program system consists of a set of objects, a set of functions from objects to objects, a set of functional forms used to combine existing functions or objects to form new functions, the function application operation (:) and a set of function definitions. **FP** is founded on the use of combining (functional) forms for creating programs. Combining forms are used within user-defined functions to build more complex functions from the primitive functions provided. Some of these functional forms are map, fold and compose HOFs but in a different terminology.

Daneluto and **Pelagatti** [43] have noted that it is possible to have parallel implementations for the functional forms available in **FP**. They proposed a skeletal version of FP where skeletons are called templates in their terminology. A compiler is described and a library of templates. No implementation work was reported for their design.

3.2.2 Ektran

Ektran [1] is a simple functional programming language with polymorphic types and it draws on Backus' **FP** which was found suitable as:

- It is a simple functional programming language, and is needed only as a programming framework which allows combining of basic primitives,
- It provides a set of basic functions which can be combined to build other functions and
- It is an implicit language with no control over process mapping, data distribution and inter-process communication.

The use of functional forms in **FP** has influenced the design of **Ektran**. We can think of the functional forms as coordination functions for composing sequential functions into large sequential functions. Therefore, it would be useful to define coordination skeletons to define and manipulate skeletons. However, as we want to avoid special constructs to be used by the programmer to combine the skeletons, we prefer to automate this process using compile-time analysis. The analysis will analyse the coordination information between the naturally combined skeletons as we shall see in the next chapter.

An **Ektran** program consists of constant bindings, function definitions and an expression to be evaluated. We give below an overview of Ektran's essential characteristics. Table 3.1 shows the Ektran grammar in BNF (Backus Normal Form). The brackets [ ] enclose optional phrases and if they are in bold this

[1] Ektran is an Arabic word which means a function.

<table>
<tr><td colspan="3">Constant bindings and declarations:</td></tr>
<tr><td><prog></td><td>::=</td><td>[<constant_bindings> ;] [<defs>;] <exp></td></tr>
<tr><td><constant_bindings></td><td>::=</td><td>val <name> = <const> | val <name> = <list> |
val <name> = <tuple></td></tr>
<tr><td><defs></td><td>::=</td><td><def> | <def>; <defs></td></tr>
<tr><td><def></td><td>::=</td><td>def <name> [<args>] = <exp></td></tr>
<tr><td><args></td><td>::=</td><td><name>[:<type_sig>] | <name>[:<type_sig>]
<args> | <tuple_arg> | <tuple_arg> <args></td></tr>
<tr><td><tuple_arg></td><td>::=</td><td><tmp_val> | <tmp_val> , <tuple_arg></td></tr>
<tr><td><tmp_val></td><td>::=</td><td><name> [: <type_sig>]</td></tr>
<tr><td><type_sig></td><td>::=</td><td>i | r | b | s | l <type_sig></td></tr>
<tr><td colspan="3">Expressions:</td></tr>
<tr><td><exp></td><td>::=</td><td><value> | <operator> | (<exp> <exp>) |
(<exp>)| if <exp> then <exp> else <exp> |
<binary_operation> | <unary_operation></td></tr>
<tr><td><value></td><td>::=</td><td><const> | <list> | <tuple></td></tr>
<tr><td><unary_operation></td><td>::=</td><td><unary_operator> <exp></td></tr>
<tr><td><binary_operation></td><td>::=</td><td><binary_operator> <exp> <exp></td></tr>
<tr><td><unary_operator></td><td>::=</td><td>not | hd | tl | sin | cos | tan | sqrt</td></tr>
<tr><td><binary_operator></td><td>::=</td><td>< | <= | == | <> | > | >= | + | − | % |
+ | − | / | * | || | && | ++ | ^</td></tr>
<tr><td><operator></td><td>::=</td><td>map | fold | Fold | SFold | @</td></tr>
<tr><td><const></td><td>::=</td><td><name> | <number> | <string> | <bool></td></tr>
<tr><td><list></td><td>::=</td><td>[]| [<list>[,<list>]* | <value> [, <value>]*]</td></tr>
<tr><td><tuple></td><td>::=</td><td>{ <tuple_seq> }</td></tr>
<tr><td><tuple_seq></td><td>::=</td><td><value> | <value> , <tuple_seq></td></tr>
<tr><td><name></td><td>::=</td><td>letter[letter|digit]*</td></tr>
<tr><td><number></td><td>::=</td><td>[~][digit]*[.][digit]*</td></tr>
<tr><td><letter></td><td>::=</td><td>A - Z | a - z</td></tr>
<tr><td><bool></td><td>::=</td><td>true | false</td></tr>
<tr><td><digit></td><td>::=</td><td>0 - 9</td></tr>
<tr><td><string></td><td>::=</td><td>' [letter|digit]* '</td></tr>
</table>

Table 3.1: Ektran grammar.

means they are part of the language, the symbol * means repeat the previous expression any number of times.

In constant bindings, an identifier is associated with a constant value for more readable and structured programs. Constant values are atomic expressions which can be integers, reals, booleans or strings, for instance:

```
val IntegerConst = 1234; -- an integer constant
val RealConst = 34.45; -- a real constant
val BoolConst = true; -- a boolean constant
val StringConst = 'AbC12'; -- a string constant
val IntConst = IntegerConst; -- an integer constant
```

More complex constants like tuples and lists can be built using atomic expressions, for instance:

```
val TupleConst = {'ABC',{21, true}}; -- a tuple constant
val ListConst = [[1,2,3],[4,5,6]]; -- a nested list constant
```

In function definitions, a single or multiple function(s) can be defined. To define a function an identifier has to be given after the **def** keyword which might be followed by a number of arguments. Each argument has to be a valid name identifier and it is possible to give it a type value.

Types are abbreviated by single letter rather than full type names. Subsequently, i stands for integer, r for real, b for boolean, s for string and l for list. For example, to express a bound variable of type real it has to be followed by r in a function definition. For example, giving the type signature llr to an argument of a function will declare this argument to be a list of lists of type real. While the compile time system has a polymorphic type checker, the purpose of type variables is to identify monomorphic types instead of polymorphic types for function definitions. There are two reasons for this provision: 1) in cases where the type checker might fail in deducing the type of a function then the programmer will assist the type checker by providing types for the arguments of a function. For example, if the type checker could not resolve the types of arguments for an overloaded arithmetic operator, 2) in some cases a given function has to be of monomorphic type in order to find the communication costs (which includes transmission and flattening costs) for profiling purposes. For example, if the function argument for a given operator has a polymorphic type then it is difficult to estimate the exact communication costs for the processed tasks.

The body of a function is an expression. Expressions are the final part of the language and can be a value, an operator, a unary operation or a prefix binary operation, a function application of two expressions, an expression enclosed by parentheses, or a conditional expression, Values can be either atomic or complex constants. The unary and binary operations include arithmetic, relational, list, string, and logical operators where unary operators take one argument and binary operators take two. All operators are free from side-effects. An application in Ektran is written by separating function and argument by a space and enclosing them within parentheses. For instance:

```
def IncrementOne x = + x 1; -- function definition
(IncrementOne 5); -- function application, applying IncrementOne
```

```
to 5
    def MultiplyTwo x = * 2 x; -- function definition
```

Conditional evaluation can be introduced using the **if** .. **then** .. **else** .. construct:

if Exp1 **then** Exp2 **else** Exp3

Here expression *Exp1* must be boolean-valued. If it evaluates to true then the conditional expression has the value of *Exp2*, otherwise the expression's value is *Exp3*. For instance:

```
def apply b ll =
      If b     -- function definition
      Then ((map IncrementOne) ll)
      Else ((map MultiplyTwo) ll);
((apply true) [1,2,3,4,5,6,7,8,9,10]); -- function application
```

3.2.2.1 The Parallel Operators

Ektran provides the following skeleton-HOFs for parallel programming and we call them parallel operators. The chosen operators are a number of the functional forms provided in **FP** and we think they are common HOFs used in skeleton programming systems. All of them operate on the list data structure. They can be nested to an arbitrary depth and the size of the sublists can be of an arbitrary length. There are no limitations on the type of the base elements of lists. However, complex data types will make it difficult to estimate communication costs. The ML-style [168] notation is used in all definitions below and has no implications for the implementation.

- **map** applies its function argument f to each element of its list argument. Its definition is given below.

  ```
  fun map f [] = [] |
      map f (h::t) = (f h) ::  map f t;

  val map = fn :  ('a → 'b) → 'a list → 'b list
  ```

  Parallelism arises by applying the function f to each element of the list in parallel. This is a commonly used HOF and is useful in generating data-parallelism. An example of using the **map** HOF is given below using Ektran's syntax:

  ```
  def sq x:i = * x x;
  ((map (map sq)) [[1,2,3,4],[5,6,7,8],[9,10]]);
  ```

- **fold** applies its function argument f across a list. Its definition is given below.

```
fun fold f b [] = b |
   fold f b (h::t) = f h (fold f b t);

val fold = fn :  ('a → 'b → 'b) → 'b → 'a list → 'b
```

The function argument `f` is assumed to be associative (that is `f a (f b c) = f (f a b) c`), commutative (that is `f a b = f b a`) and there is some base value `b` (which has to be identified by the programmer) such that `f a b = a`. An example of using the **fold** HOF is given below using Ektran's syntax:

```
def add x:i y = + x y;
(((fold add) 0) [1,2,3,4,5,6,7,8,9]);
```

The absence of associativity would result in failing to parallelise the **fold** operator as the parallel evaluation will produce incorrect results. This means that the results will be different from the results obtained using a sequential implementation. The reason is related to the parallel implementation used to parallelise the **fold** HOF. Note that the subelements of the list will be distributed across a number of processors. Therefore, it is difficult to maintain an order in applying **fold**'s function argument to the distributed elements. However, the absence of commutativity would not fail the parallelisation process but may affect the final result if care is not taken in the parallel implementation. In other words, if the function argument to the fold HOF is not commutative then the sequential and parallel evaluations will produce the same result as the run-time system will distribute the subelements of the list data structure in a specific order. The availability of commutativity allows the list elements to be permuted without affecting the final result. Note that it is the programmer's responsibility to supply a function that is both commutative and associative as the compiler does not check for these two properties and will not report an error.

An example will show how the absence of commutativity will not affect the final result if data is distributed to processors in a given order. Below we give the definition of function `append` which is associative but not commutative.

```
fun append [] l = l |
     append (h::t) l = h::(append t l);
```

The function `append` takes two arguments (lists) and returns a list. We can use this function as an argument to **fold** over a list of lists.

```
fold append [] [[1,2],[3,4],[5,6],[7,8],[9,10],[11,12],[13,14],[15,6]];
```

In the above program **fold** with `append` as its function argument is applied to a list that has 8 sublists. The expected result is [1,2,3,4,5,6,7,8,9,10,11,12, 13,14,15,16]. However, if the sublists are not distributed in a specific order then incorrect results will be returned. This order is mentioned in the discussion of the parallel implementation for the **fold** HOF in Section 5.5.2. A possible incorrect result could be [1,2,3,4,5,6,7,8,13,14,15,16,9,10,11,12].

Note that **fold** cannot have any other parallel operator as its function argument because all of them are not associative. Also, using the **fold** HOF requires care from the programmer as the function argument could be of the form `fn h ⇒ fn t ⇒ f1 (f2 h) t`. Therefore, we have introduced two special operators for dealing with this special case. These are **Fold** and **SFold** and they are described next.

- **Fold** is a special case of **fold** where the function argument is of the form `fn h ⇒ fn t ⇒ f1 (f2 h) t`. Here we are looking for associativity/commutativity in `f1` not in `f2`. In this case, **Fold** works by folding a function over the list resulting from mapping some other function over the original list. Note that the `f1` will be used to combine (merge) the received sub-results in the parallel implementation. Below we give two possible definitions for **Fold**.

```
fun Fold f1 f2 b [] = b |
   Fold f1 f2 b (h::t) = f1 (f2 h) (Fold f1 f2 b t);

fun foldr f b [] = b |
   foldr f b (h::t) = f h (foldr f b t);
fun Fold f1 f2 b l = foldr f1 b (map f2 l);

val Fold = fn :  ('a → 'b → 'b) → ('c → 'a)
       → 'b → 'c list → 'b
```

  An example (which is explained in more detail in Section 6.3.5) of using the **Fold** HOF is given below using Ektran's syntax:

```
def merge ...
def insert ...
def sort ...
def mergesort l = ((((Fold merge) sort) []) l);
```

- **SFold** is a special case of **Fold**. In case the first function argument for Fold is not associative it might be possible to replace it with two functions where one of them is associative. For example, if we have `fn h ⇒ fn t ⇒ (f h) :: t` [2] where `f` is a function which does not necessarily have to be associative or commutative and `::` is the list constructor (`cons`) which would create a new list from an element and a list. The `cons` operator is neither associative nor commutative and cannot be used to combine (merge) the received sub-results in the parallel implementation. We can add the `append (@)` operator which is an associative operator and would give correct results. By introducing the following definition `fn h ⇒ fn t ⇒ [(f h)] @ t`, where `@` would append two lists into one list of the same type, we can replace the definition which used the `cons` operator. Here, `@` replaced `cons` only in the parallel implementation, and a list `[f h]` is

[2]Note that this is the common definition used to implement map as fold.

created from the application `(f h)` and the empty list `[]`. Therefore, `f1` is the `append @` operator which is associative but not commutative, `f2` is the `cons` operator and `f3` is `f`. Below we give its formal definition. Note that `f1` is not used in the sequential implementation. However, it is passed as an argument to the parallel implementation where it is used to combine (merge) the received sub-results. Each sublist will be reduced using the functions `f2`and `f3`.

```
    fun SFold f1 f2 f3 b [] = b |
      SFold f1 f2 f3 b (h::t) = f2 (f3 h) (SFold f1 f2 f3 b t);

val SFold = fn :  'a → ('b → 'c → 'c) → ('d → 'b) → 'c → 'd
list → 'c
```

An example (which is explained in more detail in Section 6.3.4.1) of using the **SFold** HOF is given below using Ektran's syntax:

```
    def append ...
    def cons ...
    def MatrixMul ...
    def listarrange ...

(((((SFold append) cons) MatrixMul) []) (listarrange 8));
```

- **composition** in Ektran uses the operator @. Here the output of one function is the input to the next and so on. The composed functions can be grouped into one data structure which is normally a list. The list of functions is called a pipeline. Applying the pipeline to a stream of input data would give rise to parallelism assuming that the pipeline functions are distributed over several processors. Therefore, the compose pipeline will only work effectively when nested within another HOF which generates a sequence of data values. In the current Ektran implementation, the compose operator has to be nested within the **map** operator to exploit pipeline parallelism. Section 4.3 explains how the pipeline is built from composed functions in a process we call **pipeline normalisation**.

  Also, our pipeline is unrestricted. The functions in the pipeline can be of heterogeneous types where the domain type of each function is the same as the range type of its argument function. Pipeline parallelism introduced by other researchers was homogeneous such as [20] and [45].

  Below we give a possible definition for a monomorphic pipeline:

```
fun compose f g x = f (g x);
fun id z = z;
fun pipeline lfs = foldr compose id lfs;
fun mappipeline data = map pipeline data;
val mappipeline = fn :  ('a → 'a) list list → ('a → 'a) list
```

However, the generated list of functions (lfs) which is the argument for the pipeline function determines if the pipeline is of monomorphic or polymorphic type. The list of functions may contain functions of different types. The above definition is a simple illustration of how the pipeline works and has no implications for the polymorphic parallel implementation which will be explained in more detail in Section 5.5.5. An example is given below which shows how to use the compose operator in Ektran's syntax:

```
def f1 x:i = * x x;
def f2 y = + y 4;
def f3 w = - w 2;
def f4 z = / z 2;
def fall x = (((@ f1) ((@ f2) f3)) x);
((map (@ f4) fall) [1,2,3,4,5,6,7,8,9,10]);
```

3.2.2.2 Ektran Assumptions

There are a number of restrictions that the programmer has to be aware of:

- Nested HOFs should not be used in the body of recursive functions as nesting cannot be deduced at the current stage for such cases. This is a limitation of the compile-time analysis as recursive functions are not analysed.

- Recursive functions are restricted to a simple form where a function can call itself only within its own body. In other words no mutual recursion is allowed as the type checker has no forward references.

- It is the programmer's responsibilty to provide an associative function to the **fold** HOF or transform a non-associative function (if possible) into a different form in order to use **Fold** or **SFold**.

3.2.2.3 An Example Program

An example of an Ektran program is given in Figures 3.1 and 3.2. The example is a simple merge sort program, which uses the map, fold and compose parallel operators. The algorithm works as follows. The first step is to sort the sublists using a map HOF which applies an insert sort to its sublists. The next step is to merge the subresults using a fold HOF. These two steps are the two stages of the heterogeneous pipeline. Note that the stages of the pipeline are parallel stages as well. The final step is to merge the sublists to produce the final list.

The pipeline for this example nests two HOFs where the first is a **map** and the second is a **fold**. This illustrates the expressiveness of the Ektran model where pipeline parallelism is exploited with heterogeneous stages. In Chapter 4 we use this example program to demonstrate how the static analysis works and the results for this sample program are shown in Section 6.3.3.

```
def length l =
    if == l []
    then 0
    else + 1 (length (tl l));
def less1 n1 n2 =
    if && (== n1 []) (== n2 [])
    then false
    else || (< (hd n1) (hd n2)) (&& (== (hd n1) (hd n2)) ((less1
(tl n1)) (tl n2)));
def less n1 n2 =
    if < (length n1) (length n2)
    then true
    else
       if > (length n1) (length n2)
       then false
       else ((less1 n1) n2);
def merge l1 l2 =
    if == l2 []
    then l1
    else
       if == l1 []
       then l2
       else
          if ((less (hd l1)) (hd l2))
          then ++ (hd l1) ((merge (tl l1)) l2)
          else ++ (hd l2) ((merge l1) (tl l2));
def insert v l1 =
    if == l1 []
    then [v]
    else
       if ((less v) (hd l1))
       then ++ v l1
       else ++ (hd l1) ((insert v) (tl l1));
def rfold f b l =
    if == l []
    then b
    else ((f (hd l)) (((rfold f) b) (tl l)));
```

Figure 3.1: An Ektran example program I.

```
def sort l1 = (((rfold insert) []) l1);
def merge1 l2:llli = (((fold merge) []) l2);
def sortmap l3:llli = ((map sort) l3);
def mod1 x y = - x (* y (% x y));
def gen1 n m =
    if == n m
    then []
    else ++ ((mod1 n) 10) ((gen1 (- n 1)) m);
def gen2 i n s =
    if < i 1
    then []
    else ++ ((gen1 (+ n s)) n) (((gen2 (- i 1)) (+ n s)) s);
def gen3 j i n s =
    if < j 1
    then []
    else ++ (((gen2 i) n) s) ((((gen3 (- j 1)) i) n ) s);
def gen4 k j i n s =
    if < k 1
    then []
    else ++ ((((gen3 j) i) n ) s) (((((gen4 (- k 1)) j) i) (+ n
(* i s))) s);
def sortll ll = (((@ merge1) (map ((@ merge1) sortmap))) ll);
    (sortll (((((gen4 10) 10) 10) 9) 40));
```

Figure 3.2: An Ektran example program II.

3.2.2.4 The Advantages and Limitations of Ektran

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

- Ektran inherits all the advantages of functional programming as it is based on a functional paradigm. Therefore, it is relatively easy to learn and use Ektran parallel operators as they are naturally inherited in the model.
- The parallel operators provided by Ektran are characteristic of several parallel programming paradigms [136, 20].
- It is possible to extend the current model to incorporate other parallel operators.

There are, however, some limitations to the model.

- Parallelism is only exploited if the programmer uses the fixed repertoire of the given operators. This forces the programmer to remain within a strict model.
- The assumptions imposed by the Ektran model restrict the programmer from expressing programs where parallel operators are used within recursive functions.

3.2.3 Caml Light

Caml is a strongly-typed functional programming language [116, 41] and it was chosen as the first stage target language. The name **Caml** is an acronym which stands for Categorical Abstract Machine Language. The Categorical Abstract Machine is an abstract machine to define and execute functions. It resulted from theoretical considerations about the relationship between category theory and lambda-calculus. The language can be considered as a dialect of ML. Caml was chosen as an intermediate language for the following reasons:

- It is possible to generate C code from Caml programs using the **camlot** compiler. Camlot is a high performance Caml Light to C compiler. This has a useful advantage as C code can be ported easily to many different architectures.

- Caml Light provides a simple and usable C interface where Caml programs can call C functions and C routines can make *call backs* to Caml functions. This is a useful feature where HOFs in the source code can be replaced by C calls to a C skeleton library. Also, the C routines can make use of Caml functions to traverse complex data structures rather than writing C code to access Caml data structures.

- The Caml system was recommended by other researchers [10] due to its lightweight implementation and respectable performance. Other compilers like New Jersey ML [3] and sml2c [159] were discarded as being too resource hungry.

- Caml Light grammar is close to Ektran and therefore it is easy to translate Ektran to Caml code.

3.3 Engineering the Parallelising Compiler

The underlying philosophy of the Ektran compiler is to shift as much as possible of the burden of dealing with parallelism from the programmer to the compiler. This implies an implicit approach, where parallelism is exploited by the compiler. The programmer's responsibility is to use a fixed set of skeleton-HOFs which have an underlying parallel implementation. The compiler will automatically identify skeletons by the occurrence of skeleton-HOFs in the source program and generate the parallel code.

A parallelising compiler for Ektran has been engineered by plugging different components together. Each component corresponds to a main stage during the compilation process. Our approach in building the compiler was influenced by the work of **Michaelson et al** [118] and **Scaife et al** [143].

Figure 3.3 illustrates the various stages involved in compiling an Ektran source program. The purpose of the first stage (Front End 1) is to transform an Ektran program into a CAML source program. This stage was written completely using SML. The next stage (Front End 2) will generate a C program from the CAML source program. Finally, target code can be generated for different target architectures by the back end. The skeleton library is linked at compile-time to the generated code. The skeleton library contains the parallel implementations for each supported skeleton-HOFs. Chapter 5 explains in detail the skeletons and the run-time system used to run nested skeletons in parallel.

Below we describe each stage in more detail.

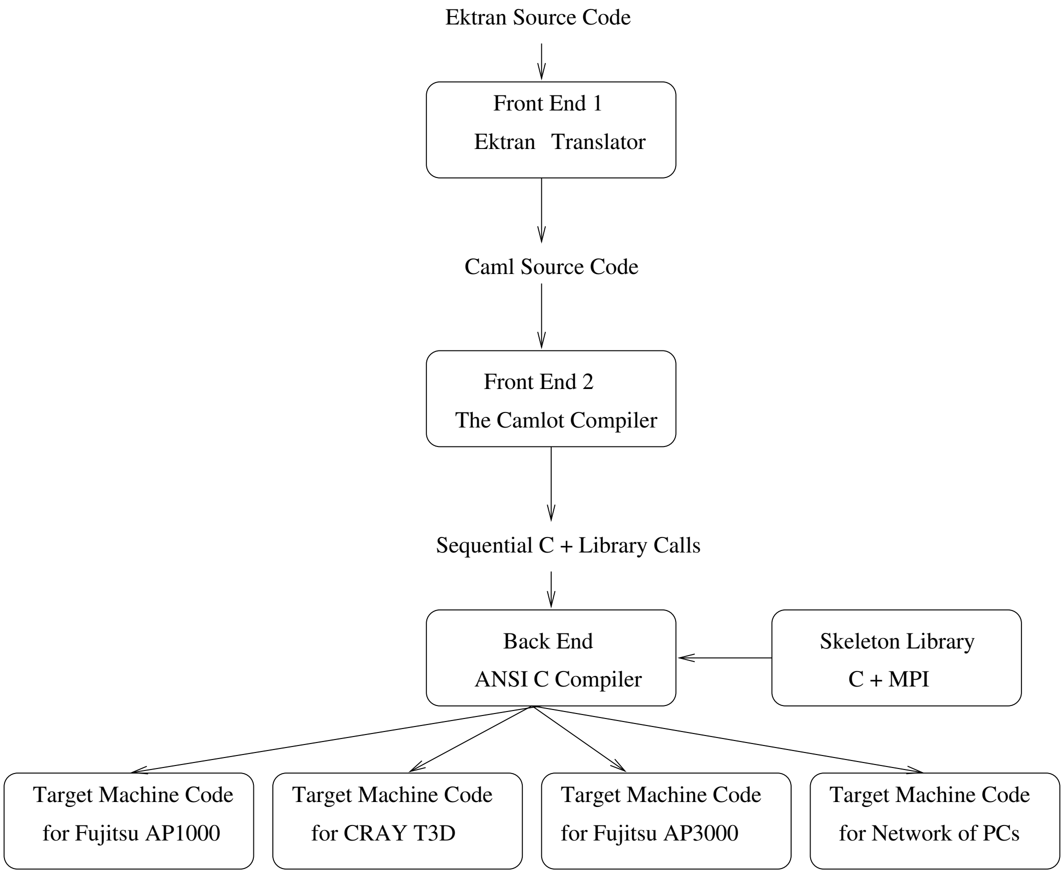

Figure 3.3: The compilation environment.

3.3.1 Front End 1

In this stage an **Ektran** program is scanned, parsed, type-checked (based on [155] and [139]), preprocessed and translated into a Caml Light source program with calls to the skeleton library using the language C interface. The parser generates an abstract syntax tree (**AST**) which is used by the preprocessor for the static-analysis of the source program. Also, in this stage we generate the pipeline wrappers to allow heterogeneous stages for pipeline parallelism. This stage can be modified to add other tasks like analysing the program to exploit useful parallelism. The preprocessing and translation stages are the topics of investigation in the next chapter.

3.3.2 Front End 2

The purpose of this stage is to transform the resulting program from the previous stage into a portable source program which can be compiled easily to generate the target machine code. Here, the resulting Caml light source code is compiled to a C program using the **Camlot** compiler. However, to successfully compile a Caml light program, a module interface file is required for the parallel operators. The type declarations in this file are shown in Appendix A for each parallel operator. A file which contains the wrappers for the pipeline stages is also required. This file is generated during the translation process which is described in Section 4.6.2.

3.3.3 Back End

The generated C code can be compiled using an ANSI C compiler and linked with the skeleton library to generate target code for a given machine. For each supported machine there is a skeleton library. The current compiler supports the following machines: the Fujitsu AP3000, the Cray T3D, the Fujitsu AP1000 and a network of Linux workstations (NOWs). The library can be recompiled for other architectures as long as they support **MPI**. Note that only the skeleton library has to be compiled for each different target architecture as the generated C code does not depend on the target machine.

3.3.4 The Skeleton Library

The skeleton library contains the parallel implementations for each supported skeleton. The skeletons were developed using C + MPI and are linked at compile time to the generated C code. For each skeleton, there is a simple wrapper around the skeleton's C interface which is often called the "stub code". The purpose of the stub code is to decode the arguments to extract C values from the given Caml Light values, and to encode the return value as a Caml Light value. Here the wrappers are for the skeleton C interface. However, it is also possible to write skeletons directly in the source language, an approach advocated by **Serot** [148]. However, in **Serot's SCAMPI** the wrappers are for MPI routines rather than skeletons which is over explicit according to Ektran's approach for parallel programming. The arguments used in the C interface were generated by the translator by interpreting the annotations for each HOF. The run-time system uses these arguments to decide how to schedule the nested skeletons in parallel. The wrappers for the parallel implementations are outlined in the sub-sections of Appendix A.

3.4 Summary

This chapter presented the design, organisation and implementation of the compile-time framework of our approach to structured parallel programming. We have outlined Ektran and how it is used to demonstrate our framework for structured parallel programming using nested skeletons. The components of the compiler and their integration to provide an architecture-independent parallelising compiler were described. In the next chapter we will explain in detail the preprocessing and translation stages of the **front end 1** and show how they are used to statically analyse the source program in order to generate a nesting structure which can be used by the run-time system to run nested skeletons in parallel.

Chapter 4

Preprocessing and Translation

4.1 Introduction

In this chapter the techniques used for the compile-time analysis of an **Ektran** program are described. The discussion details the preprocessing stages of the source code which are based on translation schemes that generate a nesting structure which is used to combine the corresponding skeletons. The result is an abstract syntax tree (**AST**) with details about nesting.

The preprocessing consists of four stages: symbolic beta-reduction, pipeline normalisation, nesting deduction and free variable detection. Finally the fully processed program is translated into an equivalent source program with calls to the skeleton library.

The stages are illustrated in Figure 4.1 and described in the following sections as translation schemes. The schemes are presented in a general format where the name of the scheme is in bold, followed by an **Ektran** syntactic object enclosed by double square brackets $[\![ ]\!]$ and possibly a number of related arguments. The $\{\}$ enclose optional expressions and ; is used to separate different expressions of an **Ektran** program. Note that each scheme has different cases according to the syntactic object it is applied to. To simplify, a translation scheme can be regarded as a function, which takes an **Ektran** syntactic object and produces an equivalent syntactic object as a result.

The schemes are outlined in an ML-style notation.

4.2 Symbolic Beta-Reduction

The **SymbolicBetaReduction** scheme is the first stage of the preprocessing process and can be described as partial beta reduction [91] of a functional program. The purpose of this stage is to lift **HOFs** to the top level of the source program to simplify the nesting deduction stage. The basic idea is to replace the name of a function with its body. It can be called in-line expansion of the called function.

The scheme is illustrated in Figure 4.2 where it takes an **Ektran** program and generates a partially beta reduced program where **HOFs** have been lifted to

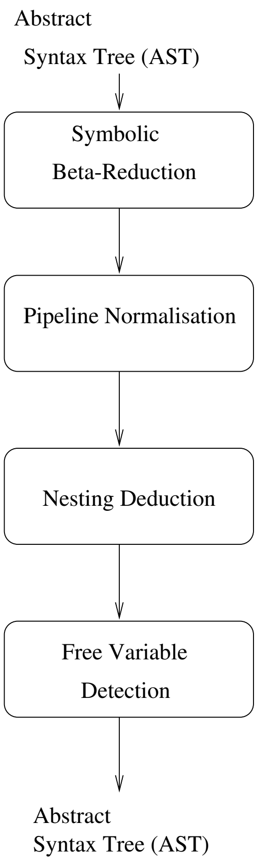

Figure 4.1: The preprocessing stages.

the top level of the program. The scheme calls another scheme called **Reduce** explained below, which does the work.

SymbolicBetaReduction ⟦ constant_bindings ; defs ; exp ⟧ =
let val env1 = **make_constants_env** ⟦ constant_bindings ⟧
    val env2 = **make_defs_env** ⟦ defs ⟧
    val env = env1 @ env2
    in ⟦ constant_bindings ; defs ;
        Reduce ⟦ exp ⟧ (**make_rec_list** ⟦ defs ⟧) [] env ⟧

Figure 4.2: The symbolic beta reduction scheme.

The remaining schemes **make_rec_list**, **make_constants_env** and **make_defs_env** generate a list of recursive functions, an environment which associates names of constants with their values and a list of tuples which associates each function name with its body respectively.

The **Reduce** scheme partially beta reduces an **Ektran** expression. It takes four parameters. The first is an expression, the second is a list of recursive function names found in the program, the third is an empty list to start with and the last parameter is a list of tuples which associates each constant identifier with its value and each function name with its body. Figure 4.3 presents the scheme with the main cases it has to account for. If it is applied to a name then it first checks if this identifier is found in the list of recursive functions. If it is found then no reduction is done. This is a current restriction of the scheme as recursive functions are not beta reduced. If it is not found then the **Find** scheme is applied to the identifier. The associated value or function body corresponding to the identifier is returned after an exhaustive search in the relevant environments.

In case **Reduce** is applied to a function application then the function expression is beta reduced first. If the reduced function expression is found to be function definition then it is reduced with the identifier bound to the reduced argument expression added to the environment. If not then the application expression is reduced. If **Reduce** is applied to a function definition then the function body is reduced with the identifier added to the environment. The rest of the cases are straight forward and not explained for brevity.

```
Reduce ⟦ name ⟧ env0 env1 env2 =
    if FindRecursive ⟦ name ⟧ env0
    then ⟦ name ⟧
    else Find ⟦ name ⟧ env2 env1
Reduce ⟦ (e1 e2) ⟧ env0 env1 env2 =
    let val re1 = Reduce ⟦ e1 ⟧ env0 env1 env2
    in case re1 of
    ⟦ (fn n ⇒ b) ⟧ =
   Reduce ⟦ b ⟧ env0 ((n, Reduce ⟦ e1 ⟧ env0 env1 env2)::env1) env2
    _ =
    ⟦ (re1 (Reduce ⟦ e2 ⟧ env0 env1 env2)) ⟧
Reduce ⟦ fn name ⇒ exp ⟧ env0 env1 env2 =
    ⟦ fn name ⇒ Reduce ⟦ exp ⟧ env0 ((name,name)::(env1)) env2) ⟧
Reduce ⟦ if exp1 then exp2 else exp3 ⟧ =
    ⟦ if Reduce ⟦ exp1 ⟧ env0 env1 env2
    then Reduce ⟦ exp2 ⟧ env0 env1 env2
    else Reduce ⟦ exp3 ⟧ env0 env1 env2 ⟧
Reduce ⟦ [ exp1,..., expn ] ⟧ env0 env1 env2 =
    ⟦ [ Reduce ⟦exp1⟧ env0 env1 env2,.., Reduce ⟦expn⟧ env0 env1 env2 ] ⟧
Reduce ⟦ exp ⟧ env0 env1 env2 = ⟦ exp ⟧
```

Figure 4.3: The reduce scheme.

The scheme **FindRecursive** will return true if its first argument is found in its second argument which is a list of recursive functions and the scheme **Find** will search its environment arguments for the body of a given function name.

Figure 4.4 shows the main expression of the example program from Figure 3.1 in Chapter 3 after the beta reduction stage. Note that, only the reduced expression of the program is reported. Hence, the functions **merge**, **insert** and **gen4** are not beta reduced as they are recursive functions. The resulting expression will be processed by the rest of the schemes as a demonstration in the following sub-sections.

```
(sortll (((((gen4 10) 10) 10) 9) 40));     ⟹


(((@ (fn l2:llli ⇒ (((fold merge) []) l2)))
     (map ((@ (fn l2:llli ⇒ (((fold merge) []) l2)))
          (fn l3:llli ⇒
          ((map (fn l1 ⇒ (((rfold insert) []) l1))) l3)))))
          (((((gen4 10) 10) 10) 9) 40))
```

Figure 4.4: An example program after the symbolic beta reduction stage.

4.3 Pipeline Normalisation

It is common to use the function composition operator (@ for **Ektran**) to compose functions where the output of one will be the input to another function and so on. The purpose of this stage is to implicitly exploit pipeline parallelism which may arise from using this operator. The problem is simply how to generate a list of functions which corresponds to the composed functions in the program?

```
PipelineNormalisation ⟦ constant_bindings ; defs ; exp ⟧ =
    let val exp_normalised = FlattenPipe (Normalise ⟦ exp ⟧)
    in ⟦ constant_bindings ; defs ; exp_normalised ⟧
```

Figure 4.5: The pipeline normalisation scheme.

To do so, a number of steps are involved in this stage. The **PipelineNormalisation** scheme presented in Figure 4.5 takes the output of the previous stage as input and applies the schemes **Normalise** and **FlattenPipe** to it in the given order. The **Normalise** scheme takes an expression as an argument. Figure 4.6 illustrates its main case. On its first encounter with the operator @ in a function application it generates a pipeline and calls the **Extract** scheme in order to build a list of functions from the function application.

The **Extract** scheme takes two parameters. The first is an expression and the second is a list of functions. Figure 4.7 demonstrates the scheme with its main cases. When it is applied to a function application where the function argument is the compose operator (@) then the **Normalise** scheme is applied to the application argument. The idea is to drop the compose operator from the list and generate a list of functions found in the function application.

Normalise ⟦ (map ((@ e1) e2)) ⟧ =
  ⟦ mappipeline **Extract** ⟦ ((@ e1) e2) ⟧ [] ⟧
Normalise ⟦ (e1 e2) ⟧ = ((**Normalise** ⟦ e1 ⟧) (**Normalise** ⟦ e2 ⟧))
Normalise ⟦ (fn arg ⇒ exp) ⟧ =
  ⟦ fn arg ⇒ **Normalise** ⟦ exp ⟧ ⟧
Normalise ⟦ if exp1 then exp2 else exp3 ⟧ =
  ⟦ if **Normalise** ⟦ exp1 ⟧ env0 env1 env2
  then **Normalise** ⟦ exp2 ⟧ env0 env1 env2
  else **Normalise** ⟦ exp3 ⟧ env0 env1 env2 ⟧
Normalise ⟦ exp ⟧ = ⟦ exp ⟧

Figure 4.6: The normalisation scheme.

Extract ⟦ (@ f) ⟧ L = ⟦ (**Normalise** ⟦ f ⟧)::L ⟧
Extract ⟦ (<operator> e) ⟧ L = ⟦ ((<operator> **Normalise** ⟦ e ⟧))::L ⟧
Extract ⟦ (e1 e2) ⟧ L= **Extract** ⟦ e2 ⟧ (**Extract** ⟦ e1 ⟧ L)
Extract ⟦ (fn arg ⇒ exp) ⟧ L = ⟦ (fn arg ⇒ **Normalise** ⟦ exp ⟧)::L ⟧
Extract ⟦ exp ⟧ L = ⟦ exp::L ⟧

Figure 4.7: The extract scheme.

The **FlattenPipe** scheme takes one parameter which is an expression. Figure 4.8 presents the scheme which search the expression for the pipeline construct. If a pipeline is found then its list of functions is flattened using the **FlattenList** scheme.

Figure 4.9 illustrates the **FlattenList** scheme rules. If it encounters an expression of the form (`fn arg ⇒ exp`) and exp is a pipeline then it would delete it and keep its list of functions for the outer pipeline. In case the expression is a function application then nothing is performed if the function argument is a **HOF**, otherwise both arguments are flattened. The last case has the pipeline construct as an argument where its list of functions is flattened and then inserted into the outer list.

The schemes **DropPipe**, **FindPipe** and **InsertList** delete the pipeline construct from an expression of the form (`fn arg ⇒ exp`), checks if the argument is a pipeline and insert the elements of a list into another list, respectively.

The **DropPipe** scheme will search for multiple occurrences of function compositions and delete them. For example, consider a pipeline where one of its stages is another pipeline. This stage could be flattened and its stages inserted in the outer pipeline. The **FindPipe** scheme will return true if it finds a pipeline in the body of a function and otherwise false.

Figure 4.10 shows the example previously shown in Figure 4.4 after the pipeline

FlattenPipe ⟦ (e1 e2) ⟧ = ⟦ ((**FlattenPipe** ⟦ e1 ⟧) (**FlattenPipe** ⟦ e2 ⟧)))⟧
FlattenPipe ⟦ (mappipeline l) ⟧ = ⟦ mappipeline (**FlattenList** ⟦ l ⟧) ⟧
FlattenPipe ⟦ (fn arg ⇒ ⟦ exp ⟧) ⟧ =
    ⟦ (fn arg ⇒ **FlattenPipe** ⟦ exp ⟧) ⟧
FlattenPipe ⟦ if exp1 then exp2 else exp3 ⟧ =
    ⟦ if **FlattenPipe** ⟦ exp1 ⟧ env0 env1 env2
    then **FlattenPipe** ⟦ exp2 ⟧ env0 env1 env2
    else **FlattenPipe** ⟦ exp3 ⟧ env0 env1 env2 ⟧
FlattenPipe ⟦ exp ⟧ = ⟦ exp ⟧

Figure 4.8: The flatten pipe scheme.

FlattenList ⟦ [] ⟧ = ⟦ [] ⟧
FlattenList ⟦ (name::l) ⟧ = ⟦ (name)::(**FlattenList** ⟦ l ⟧) ⟧
FlattenList ⟦ ((fn arg ⇒ exp)::l) ⟧ =
    if **FindPipe** ⟦ exp ⟧
    then (**InsertList** (rev (**DropPipe** (fn arg → exp))) (**FlattenList** l))
    else (fn arg → exp)::(**FlattenList** l)
FlattenList ⟦ ((e1 e2)::l) ⟧ =
    if **IsHOF** ⟦ e1 ⟧
    then ⟦ ((e1 e2)::(**Flattenlist** ⟦ l ⟧)) ⟧
    else ⟦ ((**FlattenPipe** ⟦ e1 ⟧) (**FlattenPipe** ⟦ e2 ⟧))::(**FlattenList** ⟦ l ⟧)⟧
FlattenList ⟦ ((mappipeline l)::l2) ⟧ =
    Insertlist (rev (**FlattenList** ⟦ l ⟧)) (**FlattenList** ⟦ l2 ⟧)
FlattenList ⟦ (z::l) ⟧ = ⟦ (z::(**FlattenList** ⟦ l ⟧)) ⟧

Figure 4.9: The flatten list scheme.

normalisation stage. Note that the second compose operator has disappeared from the example program as it was replaced with the **mappipeline** skeleton which was introduced by the **Normalise** scheme. The composed functions were used to generate a list of functions and supplied as an argument for the skeleton.

4.4 Nesting Deduction

The purpose of this stage is to generate a nesting structure that corresponds to the nested **HOFs** in the source program. The nesting structure is the abstract syntax tree with annotations about nesting. For example, if HOF_1 is nesting HOF_2 then the scheme will annotate the first HOF as HOF_1^N where N means that the function argument is a HOF. Figure 4.11 presents the **NestingDeduction** scheme which takes one parameter. When applied to an Ektran program it applies the **Deduce** scheme to the main expression.

```
(((@ (fn l2:llli → (((fold merge) []) l2)))
   (mappipeline [(fn l2:llli ⇒ (((fold merge) []) l2))),
     (fn l3:llli ⇒ ((map (fn l1 ⇒
                    (((rfold insert)    []) l1))) l3))])
     (((((gen4 10) 10) 10) 9) 40))
```

Figure 4.10: The example program after the pipeline normalisation stage.

$$\begin{array}{l}\textbf{NestingDeduction}\ [\![\ \text{constant\_bindings; defs; exp}\ ]\!] = \\ \quad [\![\ \text{constant\_bindings; defs; }\textbf{Deduce}\ [\![\ \text{exp}\ ]\!]\ ]\!]\end{array}$$

Figure 4.11: The nesting deduction scheme.

The **Deduce** scheme takes one parameter which is the main expression of the source program. The deduction itself is eased after the first stage of the processing where all HOFs have been lifted to the top level of the program. Figure 4.12 illustrates the scheme rules. The main case is function application. The scheme **IsHOF** is applied to the function argument which checks for HOFs. If a HOF is found then the application argument is checked for a HOF as well. If a HOF is found then the function argument is annotated. The **IsHOF** scheme is defined in Figure 4.13. The **AnnotateHOF** scheme takes a **HOF** and annotates it with a flag to indicate that the **HOF** is in a nesting position.

In case the expression is a pipeline then the **DeducePipeline** scheme illustrated in Figure 4.14 is applied. It takes a pipeline as an argument and applies the **Reduce** and **IsHOF** schemes to each element in its list argument. In case one of its stages found to be another **HOFs** then the pipeline is annotated about nesting.

Figure 4.15 is the example program shown after applying the **NestingDeduction** scheme to it. Note that **mappipeline** skeleton has been annotated with an N indicating that one or more of its stages contain HOFs.

4.5 Free Variable Detection

In implementing functional languages we are faced with the problem of handling free variables which occur inside local functions. In such cases, the result of applying a function does not only depend on its arguments but also on the values of its free variables.

Since Ektran's source code is translated into Caml Light and the latter is subsequently compiled to generate the target code, the treatment of free variables in Ektran depends on the compilation technique used to produce the machine

```
Deduce ⟦ (exp1 exp2) ⟧ =
    if (IsHOF exp1)
    then
        if (IsHOF exp2)
        then ⟦ ((AnnotateHOF exp1) (Deduce exp2)) ⟧
        else ⟦ ((Deduce exp1) (Deduce exp2)) ⟧
    else ⟦ ((Deduce exp1) (Deduce exp2)) ⟧
Deduce ⟦ fn arg ⇒ exp ⟧ = ⟦ fn arg ⇒ Deduce exp ⟧
Deduce ⟦ (mappipeline l) ⟧ = ⟦ DeducePipeline (mappipeline l) ⟧
Deduce ⟦ if exp1 then exp2 else exp3 ⟧ =
    ⟦ if Deduce ⟦ exp1 ⟧ env0 env1 env2
    then Deduce ⟦ exp2 ⟧ env0 env1 env2
    else Deduce ⟦ exp3 ⟧ env0 env1 env2 ⟧
Deduce ⟦ exp ⟧ = ⟦ exp ⟧
```

Figure 4.12: The deduce scheme.

```
IsHOF ⟦ map ⟧ = true
IsHOF ⟦ fold ⟧ = true
IsHOF ⟦ Fold ⟧ = true
IsHOF ⟦ SFold ⟧ = true
IsHOF ⟦ (mappipeline lf) ⟧ = true
IsHOF ⟦ (exp1 exp2) ⟧ = or (IsHOF ⟦ exp1 ⟧) (IsHOF ⟦ exp2 ⟧)
IsHOF ⟦ (fn arg ⇒ exp) ⟧ = IsHOF ⟦ exp ⟧
IsHOF ⟦ if exp1 then exp2 else exp3 ⟧ =
or (or (IsHOF ⟦ exp1 ⟧) (IsHOF ⟦ exp2 ⟧)) (IsHOF ⟦ exp3 ⟧))
IsHOF _ = false
```

Figure 4.13: The is hof scheme.

code. In Caml expressions are evaluated using a model based on environments. An environment may be regarded as a look-up table in which certain variables are bound to values. However, functional values pose a special problem because they may contain free variables that have no meaning except in an environment. The value of a functional expression in an environment will thus be a pair: the expression and this environment. Such a pair is often known as a closure.

To enter a closure for a given HOF at run-time its partial application has to be applied to a single argument. If this HOF happens to be nested within another HOF then we need to check for free variables in the inner HOF. Therefore, the argument that was used to construct the closure for the inner HOF has the possibility of being a free variable. If a free variable is found then it has to be transmitted in order to build consistent closures across all processors. Note that only one argument in needed to build a closure which implies that only one free

DeducePipeline ⟦ (mappipeline l) ⟧ =
  let val $l2$ = map **Deduce** l
  val bool2 = map **IsHOF** $l2$
  val bool3 = foldl or false bool2
  in
  if bool3
  then ⟦ (mappipeline^N l2) ⟧
  else ⟦ (mappipeline l2) ⟧
  end

Figure 4.14: The deduce pipeline scheme.

```
(((@ (fn l2:llli ⇒ (((fold merge) []) l2)))
    (mappipeline^N [(fn l2:llli ⇒ (((fold merge) []) l2))) ,
          (fn l3:llli ⇒
             ((map (fn l1 ⇒ (((rfold insert) []) l1))) l3)) ])
    (((((gen4 10) 10) 10) 9) 40))
```

Figure 4.15: The example program after the nesting deduction stage.

variable needs to be transmitted as skeletons can be entered at run-time only when applied to their last argument.

However, if the Ektran grammar is to be extended or other skeletons are added to the language then it is possible to have multiple free variables in nested HOFs. In such case, all free variables have to be detected and grouped in one tuple. At run time this tuple will be sent to the processors that need to have a copy of all free variables.

DetectFreeVariables ⟦ constant_bindings ; defs ; exp ⟧ =
  ⟦ constant_bindings ; defs ; **FindFree** ⟦ exp ⟧ ⟧

Figure 4.16: The detect free variables scheme.

In Figure 4.16 the **DetectFree** scheme is outlined. When applied to an Ektran program it traverses the AST to find all free variables which are within nested HOFs. It calls the scheme **FindFree** which actually does the detection. Figure 4.17 outlines the **FindFree** scheme which takes an Ektran syntactic object and produces another as a result. It works in a similar manner to the **Deduce** scheme. However, the main case for the scheme is function application. If the

```
FindFree ⟦ (fn arg ⇒ exp) ⟧ = ⟦ (fn arg ⇒ FindFree ⟦ e2 ⟧ ) ⟧
FindFree ⟦ (e1 e2) ⟧ =
    if IsHOFNested ⟦ e1 ⟧
    then
      if IsFree ⟦ e2 ⟧
      then ⟦ (AnnotateFree ⟦ e1 ⟧ FindFree ⟦ e2 ⟧ ) ⟧
      else ⟦ (FindFree ⟦ e1 ⟧ FindFree ⟦ e2 ⟧) ⟧
    else ⟦ (FindFree ⟦ e1 ⟧ Findfree ⟦ e2 ⟧ ) ⟧
FindFree ⟦ if exp1 then exp2 else exp3 ⟧ =
    ⟦ if FindFree ⟦ exp1 ⟧ env0 env1 env2
    then FindFree ⟦ exp2 ⟧ env0 env1 env2
    else FindFree ⟦ exp3 ⟧ env0 env1 env2 ⟧
FindFree ⟦ (mappipeline l) ⟧ =
    ⟦ (mappipeline (map FindFree ⟦ l ⟧)) ⟧
FindFree ⟦ exp ⟧ = ⟦ exp ⟧
```

Figure 4.17: The find free scheme.

```
IsFree ⟦ fn arg ⇒ exp ⟧ = IsFreeArg ⟦ arg ⟧ ⟦ exp ⟧
IsFree ⟦ (e1 e2) ⟧ = or (IsFree ⟦ e1 ⟧ ) (IsFree ⟦ e2 ⟧)
IsFree ⟦ if exp1 then exp2 else exp3 ⟧ =
      or (IsFree ⟦ exp1 ⟧ ) (or (IsFree ⟦ exp2 ⟧) (IsFree ⟦ exp3 ⟧))
IsFree _ = false
```

Figure 4.18: The is free scheme.

function argument is found to be a HOF nesting another HOF then the application argument has to be analysed to detect free variables. In this case the scheme **IsFree** is applied to the application argument to detect free variables. In Figure 4.18 we outline the **IsFree** scheme which takes one argument and returns a boolean value when applied to Ektran's syntactic objects. The main case is expressions of the form `fn arg ⇒ exp` where it applies the scheme **IsFreeArg** to it, otherwise it simply traverses function application and conditional expressions. The **IsFreeArg** scheme shown in Figure 4.19 takes two arguments. The first is an identifier which has the possibility of being a free variable and the second is an expression. The main case of the scheme is when it is applied to a lambda function. In such case, the **Scope** scheme outlined in Figure 4.20 is called to check that a given identifier is free or bound in an expression. The scheme needs to be supplied with a list of arguments which has already been passed as an argument to the **IsFreeArg** scheme itself.

The remaining schemes **get_args**, **IsArgNotInList**, **IsHOFNested** and **AnnotateFree**, build a list of arguments for a given function, check that an argument is not in a list, check that a given HOF is nesting another HOF and

IsFreeArg arg (e1 e2) =
  or (**IsFreeArg** ⟦ arg ⟧ ⟦ e1 ⟧) (**IsFreeArg** ⟦ arg ⟧ ⟦ e2 ⟧)
IsFreeArg arg ⟦ fn arg ⇒ exp ⟧ =
  let val (list_of_args, e2) = **get_args** [] ⟦ fn arg ⇒ exp ⟧
  in **Scope** arg list_of_args ⟦ e2 ⟧
IsFreeArg arg ⟦ if exp1 then exp2 else exp3 ⟧ =
  or (**IsFreeArg** arg ⟦ exp1 ⟧)
    (or (**IsFreeArg** arg ⟦ exp2 ⟧) (**IsFreeArg** arg ⟦ exp3 ⟧))
IsFreeArg arg _ = false

Figure 4.19: The IsFreeArg scheme.

Scope arg list_of_args ⟦ name ⟧ =
  (and (arg = ⟦ name ⟧) (**IsArgNotInList** ⟦ name ⟧ list_of_args))
Scope arg list_of_args ⟦ fn arg ⇒ exp ⟧ =
  Scope arg list_of_args ⟦ exp ⟧
Scope arg list_of_args ⟦ (e1 e2) ⟧ =
  or (**Scope** arg list_of_args ⟦ e1 ⟧) (**Scope** arg list_of_args ⟦ e2 ⟧)
Scope arg list_of_args ⟦ if exp1 then exp2 else exp3 ⟧ =
  or (**Scope** arg list_of_args ⟦ exp1 ⟧)
    (or (**Scope** arg list_of_args ⟦ exp2 ⟧) (**Scope** arg list_of_args ⟦ exp3 ⟧))
Scope arg list_of_args ⟦ exp ⟧ = false

Figure 4.20: The scope scheme.

annotate a HOF with a flag (F) to indicate that a free variable has been detected, respectively.

The example below illustrates a simple program where the variable `e` is free in the inner map HOF which is an argument to the fold HOF in the function `mem`. Note that `e` is free in the subexpression `((map (equal e)) l)` and not in the function `mem`. The problem happens because of the partial application of function `mem` and it is a function argument for the map HOF in function `contains` where it requires the argument `e` to evaluate `mem l e`.

```
def or1 x y = || x y;
def and1 x y = && x y;
def equal e x = == x e;
def mem l e = (((fold or1) false) ((map (equal e)) l));
def contains l1 l2 = (((fold and1) true) ((map (mem l1)) l2));
```

Therefore, the outer HOF which is a `map` in the function `contains` has to be annotated to indicate that its inner HOF has a free variable. Applying the **DetectFree** scheme in conjunction with the previous schemes to the example program will annotate the map HOF in the function `contains` in the following way:

```
def mem l e = (((fold or1) false) ((map (equal e)) l));
def contains l1 l2 = (((fold and1) true) ((map^NF (mem l1))
l2));
```

This annotation will be used at run-time where the outer map will send to its inner HOF the needed argument for consistent evaluation.

An unimplemented solution is presented below to solve the problem of multiple free variables found in an expression. The following steps have to be done in the following order to ensure consistent results:

- for the inner HOF find out how many arguments it has and call this variable num_args. The number of free variables (num_free) in such an expression could range from $0 \leq$ num_free $\leq$ num_args,
- find all free variables in inner HOF and group them in one tuple and
- add an extra argument to the inner function which is the tuple for all free variables. At run-time send this tuple to the inner skeleton.

Lambda lifting [91] is a well known solution to cope with this problem. It is a program transformation technique where the free variables of each local definition are passed as extra parameters to the function. A number of other solutions were summarised in [95].

4.6 Translation

In this final stage, a preprocessed program is translated into Caml. A simple pretty printer has been developed which takes an AST and generates the Caml code and not shown for brevity. The generated Caml code for the example program shown in Figure 3.1 can be found in Appendix B.

4.6.1 Generating Skeleton Library Calls

Whenever the translator encounters a HOF in the AST it converts it to a call to the skeleton C library through the Caml C interface. The annotations added to the HOFs are used to generate arguments [1] to the skeleton C interface which help the run-time system in making its decisions about how to handle the nesting of skeletons.

There are two annotations that a HOF can have: `N` and `F` . The default is to set the arguments to false. However, if the first annotations was found on a HOF then its `Nest` argument will be set to true indicating that the HOF is nesting another HOF. If the second annotation was also found then the `Free` argument is also set to true indicating that the HOF has free variable in its inner HOF.

[1] The arguments are explained in detail in Section 5.5.

4.6.2 Pipeline Wrappers

Different stages in a chain of compositions may have different types where only the range of a function in a given stage has to be the same as the domain of the next stage. The Camlot compiler and possibly other Hindly-Milner polymorphic functional language compilers require all elements of a list to be of the same type. Therefore, we use an ad-hoc approach to overcome this limitation. We use the Ektran type checker to find the type of each stage in the pipeline. The type of each stage is used as an argument to the generated type constructor. The constructor itself is used as a wrapper for the corresponding stage in the pipeline. In Figure 4.21 we show the wrappers generated for the example shown in Figure 4.15 in Caml syntax.

```
type wrapper =
      pipe11 of (int list list list → int list list list) |
      pipe12 of (int list list list → int list list);;
```

Figure 4.21: Wrappers generated for a program.

Here, the constructor `pipe11` corresponds to the first stage in the pipeline which has a **fold** HOF while the constructor `pipe12` corresponds to the second stage which has a **map** HOF. Both constructors will precede each function in the pipeline. Note that each stage has a different type which makes our pipeline polymorphic and allows heterogeneous stages. Previous work on pipeline parallelism such as [45, 161, 127] is monomorphic where all stages have to be of the same type.

4.7 Summary

We have presented the compile time framework in the format of translation rules (schemes). It consisted of four main stages: symbolic beta reduction, pipeline normalisation, nesting deduction and free variables detection. After successful processing of the source code, it is translated into Caml code. The annotations on the syntax tree get interpreted to generate arguments to the skeleton C interface. Complete examples were presented which shows how the schemes work. The next chapter presents in detail the run-time framework which is used to execute nested skeletons in parallel.

Chapter 5

The Run-time Framework

5.1 Introduction

In Chapter 4, the static analysis performed on the source program has generated a nesting structure which corresponds to the nested skeletons. The scheduler for each parallel implementation uses the nesting information to make decisions about how to manage combined skeletons. The run-time framework provides an abstract level for the software infrastructure that is needed to execute nested skeletons in parallel. The mechanism operates at run-time but relies on the static-analysis done at compile-time.

This chapter discusses the run-time framework used to run nested skeletons in parallel. First, we give an overview of the proposed skeleton nesting scheme in Section 5.2. Second, we present the implementation strategy for each supported skeleton in Section 5.3. A discussion of the implementation issues of skeletons is outlined in Section 5.4. The parallel implementation for each supported skeleton is discussed in detail in Section 5.5. Finally we summarise related work in Section 5.6.

5.2 Skeleton Nesting

The run-time schemes for arbitrary depth nesting of algorithmic skeletons have been designed and implemented as part of the scheduler for each parallel implementation of the supported skeletons. They were built on top of MPI's [121, 63] routines for managing message-passing groups which has the advantage of making our system dynamic rather than static. We think that for irregular problems, run-time decisions have to be made by the scheduler regarding nesting and how to divide a group. However, this would involve run-time monitoring of the parallel program which might cause an extra overhead to the execution of the program.

The schemes to be presented are loosely defined in MPI's terminology as an indication that they can be independent of any message passing library with groups. Therefore, it is possible to implement them using a primitive communication library by building routines to manage groups and then building the run-time schemes on top. This might be useful for implementing the skeletons using a parallel machine basic communication library which might reduce the overhead associated with communication that is caused by the multiple-layers on which more generic message passing libraries are built on.

The terminology used in the run-time schemes is defined below:

- A **group** is an ordered set of process identifiers (processes).
- Each process in a group has an **ID** (identifier) which is an integer constant starting from zero.
- The number of processes in a group is known as **group size** which has to be at least one process.

Also, there are a number of features associated with the schemes for the nesting system to work:

- All processes in a group can communicate with each other (i.e. they can send and receive messages to/from other processes). In other words, the system has to allow **intra-group** communication.
- Each group can be recursively sub-divided into a number of sub-groups, forming a tree-like hierarchy of groups.
- A new group can be created from selected processes in different groups. This group will be used to manage communication between the sub-groups. We can say the created group is used for **inter-group** communication.
- Processes with **ID**s zero (which might be called the root process of the group) may belong to two groups. In this case, the process has a distinct **ID** for each group it belongs to. All processes belong only to a single group when active.
- Each skeleton instance runs with access to a single group where all the processes of the group are available as its resources.

The schemes allow skeletons to be nested within themselves and within other skeletons. The basic idea is to allocate processes to skeleton groups top-down through traversal of the HOF nesting structure.

Initially, at the beginning of program execution, all processes in the cluster form a single group, the **root-group**. Their number will remain constant through program execution (SPMD-style of program execution). The root-group might be dynamically divided during program execution, following the nesting structure of the nested HOFs.

If a skeleton nests other skeletons then it reallocates processes in its group in order to support the nested skeletons. This involves dividing the group into sub-groups for each of its immediately nested skeletons, and creating a top group which connects all sub-groups. The process then continues recursively forming a tree-like group hierarchy. It should be noted that nested skeletons are executed by groups of processes; rather than by individual processes.

We define the **SplitGroup** scheme in *pseudo-code* in Figure 5.1. The scheme divides or splits a given group (which is a set of processors) into a fixed number of sub-groups. It takes two parameters. The first is the original group and the second is the split value or key which determines how many sub-groups to be created out of the original group. The scheme **SizeofGroup** returns the number of processes in a group and will be used in the parallel implementations as well.

```
SplitGroup OriginalGroup Key =
        Size ← SizeofGroup OriginalGroup
        Create Key SubGroups
        SubGroup_i = {} where 1 ≤ i ≤ key
        for i ← 0 to (Size - 1)
            Add process i to the (i mod Key)th Sub-Group
    Return SubGroups
```

Figure 5.1: The split (divide) group scheme.

The **CreateTopCommGroup** scheme is defined in Figure 5.2. The scheme creates a new group which consists of the first element in each subgroup that was created by the **SplitGroup** scheme. It takes two parameters. The first is the original group which was divided by the **SplitGroup** scheme, and the second is the same split value (key) used for the **SplitGroup** scheme.

```
CreateTopCommGroup OriginalGroup Key =
        Size ← SizeofGroup OriginalGroup
        CommGroup ← {}
        for i ← 0 to (Size - 1) step Key
            Add process i to CommGroup
    Return CommGroup
```

Figure 5.2: The create top comm group scheme.

Figure 5.3 gives a simple visualisation of how a group can be divided into two sub-groups and how a new group gets created from two sub-groups by applying the above schemes to a group. For this example, the value of Key is 2.

In the current implementation of the scheduler, for all skeletons the value of Key is decided at run-time to find out how to split or divide a given group in a number of subgroups. We use a simple heuristic which is the square root of group size rounded down.

Table 5.1 shows how a given number of processes can be divided into sub-groups at run time using this heuristic. An example will explain the division process further. Suppose the run-time system wants to divide 99 processes using the previous heuristic. The integer square root of 99 is 9. Therefore 9 subgroups

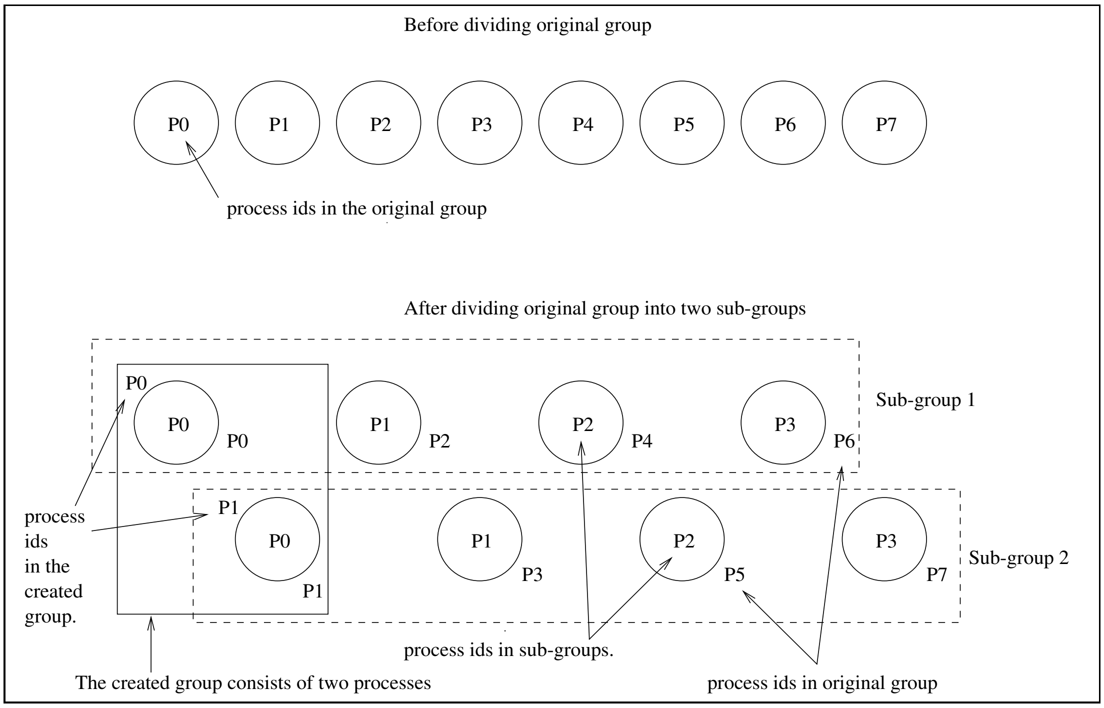

Figure 5.3: Dividing an existing group into sub-groups and creating a top group which connects the sub-groups.

will be generated where each has 11 processes in it. This is an even splitting of the group and it is expected that work load of each sub-group will be load balanced as long as the computation is regular. However, splitting 104 processes will generate 6 sub-groups of size 10 and 4 sub-groups of size 11.

5.3 Implementation Strategy for Supported Skeletons

This sections aims to outline the implementation strategy used to implement each HOF. Below, we explain the terminology used in the following subsections and later in the thesis.

- **Root group** is the original set of processes before any division into subgroups which is equal to the number of available processors when the program started its execution.

- **Root process** is the process with ID zero for a given group or subgroup. Note that this process normally belongs to two subgroups (SubGroup and CommGroup).

- **SubGroup** is a set of process IDs that has resulted from dividing a group. The union of the generated subgroups is the original group that has been divided.

no. of Processes (P)	$\lfloor\sqrt{P}\rfloor$	no. of sub-groups	size	no. of other sub-groups	size	Load Balanced
1	1	1	1	None	-	Yes
2	1	1	2	None	-	Yes
3	1	1	2	1	1	No
4	2	2	2	None	-	Yes
5	2	1	3	1	2	No
6	2	2	3	None	-	Yes
. . .						
96	9	6	11	3	10	No
97	9	7	11	2	10	No
98	9	8	9	1	10	No
99	9	9	11	None	-	Yes
100	10	10	10	None	-	Yes
101	10	9	10	1	11	No
102	10	8	10	2	11	No
103	10	7	10	3	11	No
104	10	6	10	4	11	No
105	10	5	10	5	11	No

Table 5.1: Dividing a given number of processes using the square root heuristic.

- **CommGroup** is a set of processes with that has IDs zero in the generated subgroups. This is the groups that is used for communication between the subgroups.

5.3.1 Implementation Strategy for map

The **map** HOF which applies a function to each element of a list, is realised as a broadcast process-farm skeleton. The process farm is a widely used construct for data parallelism [39]. The farmer process has access to a pool of worker processes, each of which runs the same function. The farmer distributes data to workers and collects the results back. The effect is to apply the function to every data item.

There are a variety of topologies for process farms including linear lists and trees of workers. However these require relatively low level intercommunication and incur overheads for intermediate message passing between workers [18, 164]. Instead, we have taken advantage of MPI's flexibility to avoid commitment to a concrete topology, and have implemented a broadcast farm. The farmer communicates directly with each worker through virtual connections and we rely upon the MPI implementation on each system to allocate processes to processors.

The reason why a process farm implementation was chosen for the parallel implementation of the **map** HOF as it is believed that it is more suitable for irregular problems. Other implementations are possible such as dividing the original list data structure into chunks according to the number of available processes or subgroups. The later implementation might give greater granularity and reduce the communication overhead as it has a static nature. Also it is not good for irregular problems where heavy load might be transmitted to only one process or subgroup.

Figure 5.4 shows the implementation strategy for the **map** HOF where it nests an anonymous HOF. The purpose of the illustration is to show the different processes involved in the parallel implementation.

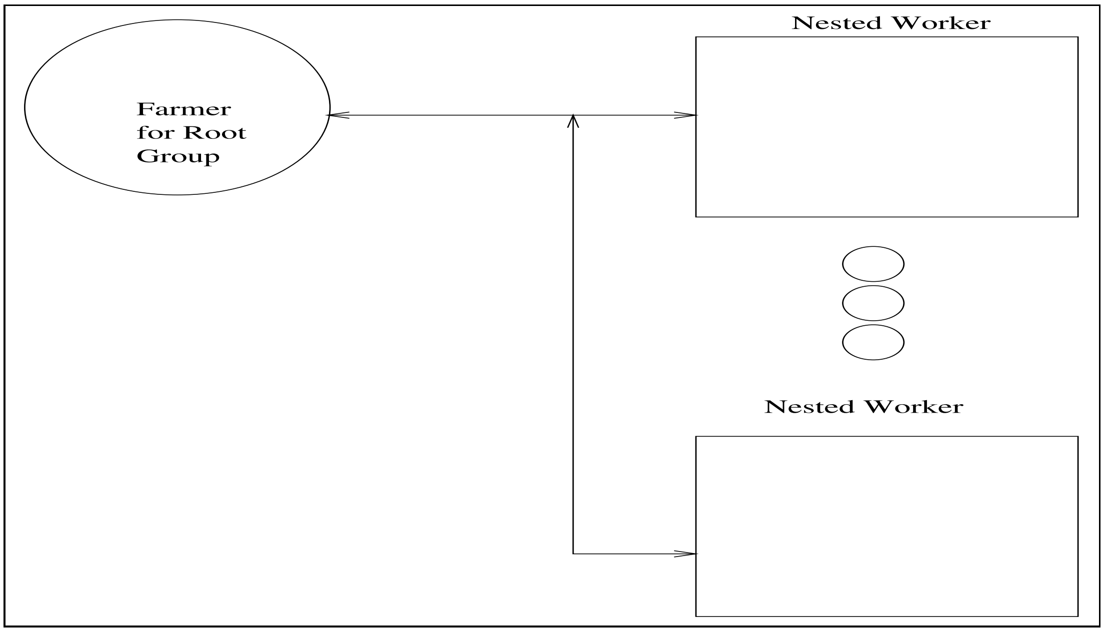

Figure 5.4: Implementation strategy for the map HOF.

From the discussion above, we can notice that the current parallel implementation for the **map** HOF has dynamic load management as the farmer keeps sending tasks to the free workers or subgroups. It might be possible to introduce more dynamic parallelism by reallocating the available processes in a subgroup to the other subgroups once a subgroup has finished its work. This reallocation will increase the number of processes in each subgroup and hopefully reduce the total execution time. However, this reallocation might cause an overhead which needs to be estimated before such a reallocation is done.

5.3.2 Implementation Strategy for fold

The **fold** HOF which applies a function between each element of a list, is realised as a broadcast binary divide and conquer skeleton. The top level node (root) divides the original list into a fixed number of sub-lists and broadcasts the sub-lists to all nodes (which include intermediate and leaf nodes) below it in the tree and keeps one sub-list for local processing. Next, the root will apply the function on its sub-list and receive sub-results from its children. The leaf nodes in the tree will receive sub-lists from root, apply the function on its local sub-lists and then send sub-results to parents. The intermediate nodes will receive sub-lists from root, apply the function on its local sub-lists, receive sub-results from children and then send the final result to parents. Note that this strategy is used with minor differences to implement the other HOFs: **Fold** and **SFold**.

Figure 5.5 shows the implementation strategy for the divide and conquer skeleton (where it is nesting itself) which can be used to implement the **fold**, **Fold** and **SFold** HOFs.

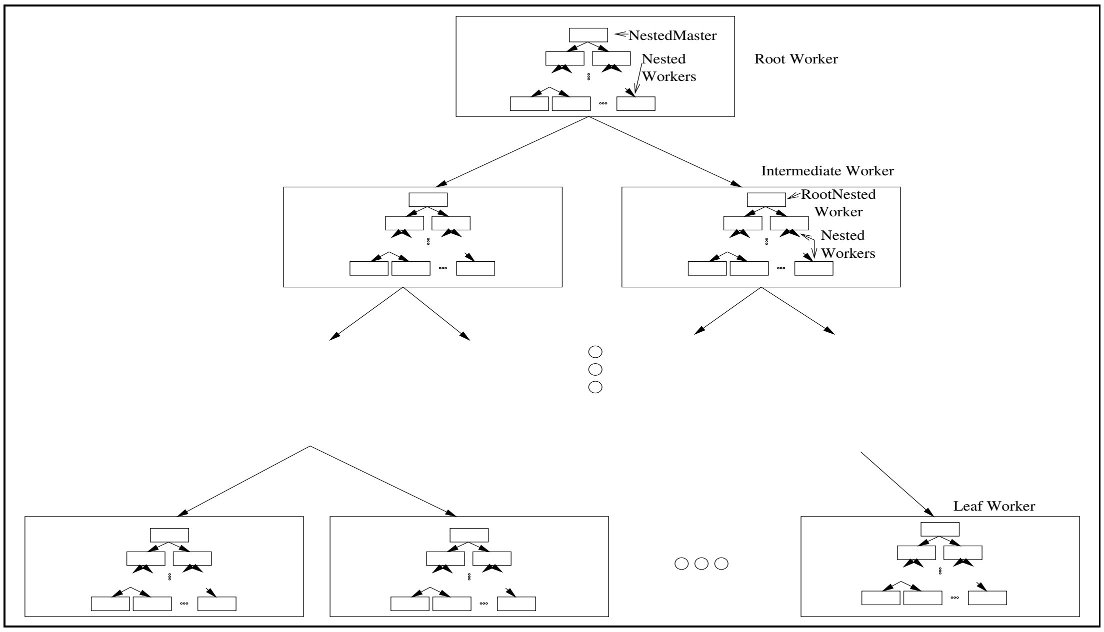

Figure 5.5: Implemenation Strategy for the fold, Fold and SFold HOFs.

5.3.3 Implementation Strategy for compose

The **compose** HOF, where the result of one function is the argument for the next, is realised as a pipeline. It is usual to express an arbitrary depth composition as folding binary compose over a list of functions, but this restricts the composed functions to having the same domain and range type. In contrast, as discussed in Section 3.2.2.1 we have implemented an unrestricted compose, where the domain type of each function must only be the same as the range type of its argument function. However, the compose pipeline will only work effectively when nested within another skeleton which generates a sequence of data values.

Figure 5.6 shows the implementation strategy for the **compose** HOF when it has two stages where each stage has a HOF as well.

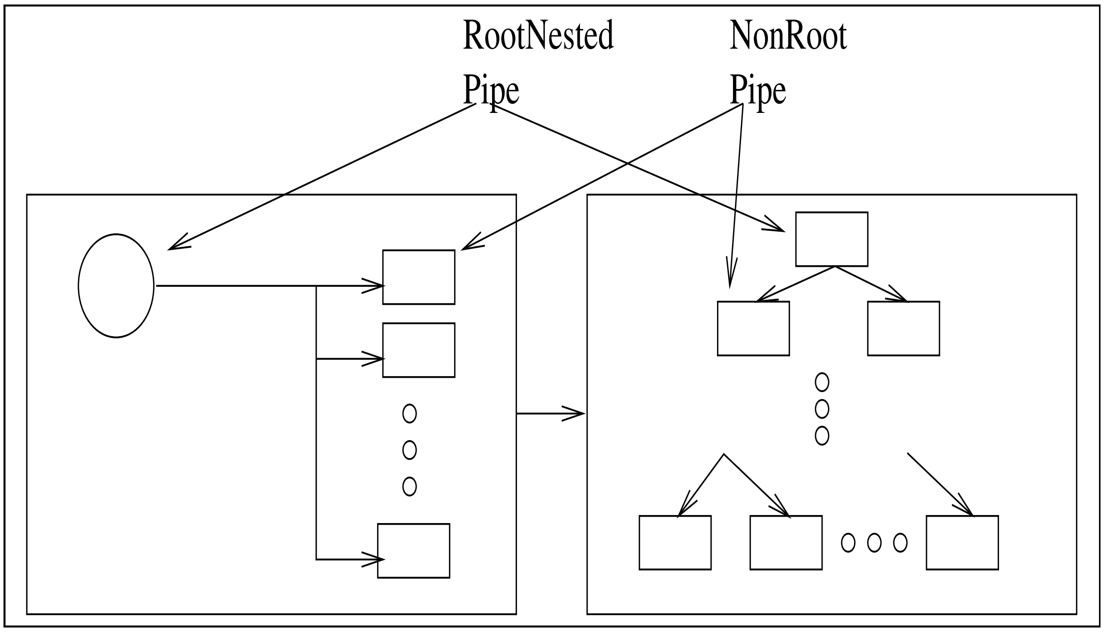

Figure 5.6: Implementation strategy for the compose HOF.

5.4 Implementation Issues for Nested Skeletons

This section aims to clarify how a number of implementation issues for nested skeletons that can have an effect on performance. These are data distribution, group synchronisation and load balancing.

5.4.1 Data Distribution

It can be seen that data is distributed according to the communication model of the parallel implementation for each supported HOF. In general, data is located on the root process of the root group just before making a call to an operator. Data will be passed through the arguments of the wrapper until it reaches the parallel implementation for the given operator. Subsequently, the root process of the root group will start transmitting the elements of its list data structure to its workers or nested workers. This depends on whether the functions being applied are skeletons or not. After the root process distributes the tasks it will wait for the sub-results to come back. It will gather the sub-results into one data structure and return the final result to the calling function. Note that all root processes in sub-groups will perform a similar activity. Also, a process could be a root process of a sub-group but it is the non-root process of the root group. Therefore, all processes have to return the result back to the root process of the root group. The role of the other processes is to receive tasks from a process and perform the work on the data and return it to the sending process.

However, returning the final result is returned on the root process of the root group causes a problem. All non-root processes of the current group will have a type consistent but incorrect value as the final result was evaluated on the root process. Therefore, to prevent incorrect result being used later on during the execution of the program on non-root processes then all these processes need to have an updated value. However, the current run-time system does not broadcast the final result from the root process to non-root processes. The execution of the example programs shown in the next chapter gave correct results as the broadcast was not needed.

A quick solution is to broadcast the final result from the root process to all non-root processes. However, this solution is not implemented as we think it is important to analyse the program to find when and where the final result is needed in order to reduce the overhead associated with broadcasting the final result.

5.4.2 Group Synchronisation

Each parallel implementation is responsible for its own group synchronisation. The role of the synchronisation is to send termination or continuation messages to processes within a given group. Here, an abort message will be used to indicate to a certain process to terminate its execution and return the results. However, a continuation message can be used to tell a given process to resume execution as more tasks are coming. This is needed when the closure for a skeleton is entered once and it has to be re-entered another time. This can be seen in the algorithms presented in Figures 5.14, 5.23 and 5.31. Another technique used for synchronisation is to broadcast the number of elements in a sub-list from the root process of the same group. The received value will be used to decide how many times a skeleton needs to be re-entered.

5.4.3 Load Balancing

It is important to note that the number of processes is equal to the number of processors when a program starts its execution. Also, it is up to MPI to determine how to map the processes to the processors and the programmer has no control over this allocation. To explain how processes are subdivided amongst different skeletons, Figure 5.7 illustrates how the **map** HOF allocates processes when it nests itself on 257 processors.

The current parallel implementations of skeletons handle neither static nor dynamic load balancing. The use of static profiling information [20] or monitoring the dynamic behaviour of the parallel program [109] might be useful for looking at load balancing issues. However, the nature of the parallel implementations is to have simple local load balancing. For example, the process farm which implements the **map** HOF has dynamic behavior. This can be seen from the parallel implementation itself where the farmer distributes tasks to the free worker. Also, tasks are transmitted one at a time rather than a batch being broadcast to a given worker. If the execution time for the applied function depends on its argument, then it is possible to have major differences of work time on different workers

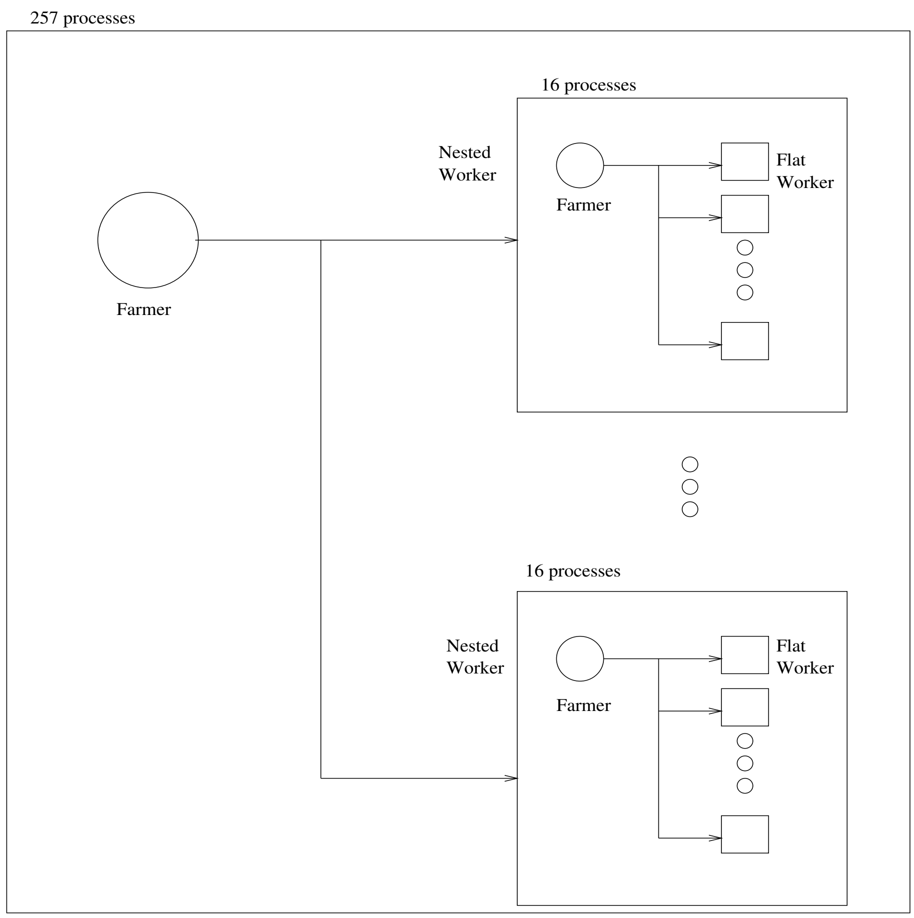

Figure 5.7: Dividing processes amongst nested skeletons.

with such an irregular problem and workers which end quickly will receive more tasks.

5.5 Parallel Implementation of Supported Skeletons

We present the parallel implementations in a general format to make them independent from MPI. We hope this generalisation will enable implementation of the skeletons without relying on MPI. In fact the *prose-code* form is used for the illustrations [172]. However, the availability of MPI's routines for managing message-passing groups will ease the implementation process. In the actual implementations, the concept of communicators is used instead of groups as MPI uses communicators to denote groups. MPI defines a **communicator** as an opaque object with a number of attributes, together with simple rules that govern its creation, use and destruction. Note that we shall refer to communicators as groups in the parallel implementations.

Before presenting the parallel implementations we will describe the major arguments that are passed to the C skeleton interface to make its decisions about the dynamic scheduling of nested skeletons. The arguments are the same for all skeleton implementations. All arguments were generated by the translator during compilation by interpreting the annotations on the HOFs in the AST. The arguments described below are used by the scheduler:

- **Parallel** is a boolean flag which is set to true by default for all skeletons. The scheduler uses it to run a skeleton in parallel. However, the scheduler can set its value to false at run-time in order to run a skeleton sequentially for subsequent executions. This will be explained below.
- **Nest** is a boolean flag. If it is set to true then the scheduler for a skeleton will know that it has a nested skeleton which has to be executed in parallel as well. In the case of a pipeline setting this flag to true, one or more stages of the pipeline have to be run in parallel as well.
- **Split** is an integer constant which is used by the scheduler to decide how many sub-groups have to be generated by dividing the original group. The current implementation generates this value at run time by rounding down the square root of group size (i.e. **Split** $= \lfloor sqrt(Size) \rfloor$).
- **Free** is a boolean flag. If it is set to true then the scheduler has to transmit the free value to all sub-groups to make sure that the closures for the inner skeletons are consistent across all processors.
- **Group** is a set of process IDs that are available to a skeleton as its resources. This group may be dynamically divided into sub-groups to support nested skeletons.

The following routines are used in the parallel implementations:

- **SizeofGroup**, returns the number of processes in a group when applied to a group
- **GetID** returns the process identifier for the calling processes when applied to a group.
- **Pack** is used to flatten a data structure into a stream of bytes suitable to be transmitted over the network.
- **Unpack** is used to unflatten a stream of bytes that has been received into a data structure.
- **GetSubList** returns a number of elements from a list data structure. It takes three parameters. The first is the position in the list to start from, the second is the number of elements to return and the third is the list data structure to be applied to.
- **ListSize** returns the number of elements in a list when applied to a list data structure.
- **MastertoWorkers** is used to broadcast the sub-lists from the root node to all nodes in the tree which includes intermediate and leaf nodes.
- **sfold** is a sequential implementation of the **fold** HOF.
- **sFold** is a sequential implementation of the **Fold** HOF.
- **smap** is a sequential implementation of the **map** HOF.
- **mpipe** is a sequential implementation of the compose HOF when nested within a **map** HOF.
- **pipe** is a sequential implementation of the compose HOF.
- **ModifyGroup** will change the value of the Group modifiable object to a new value at run-time. Note that we are referring to the root group here which was used as an argument for all skeletons C interface during compile time.
- **ModifyParallel** will change the value of the Parallel modifiable object to a new value at run-time. Similar to **ModifyGroup**.
- **VectorAllocate** is used to allocate a one dimensional array of a fixed length instead of a linked list to speedup access. It takes a single argument which is the size of the vector.
- **ListSize** will return the number of elements in a list data structure. It takes a single argument which is the list data structure.
- **MakeResultList** will return a linked list representation of a vector. It takes two arguments: a vector and the number of elements in the vector.

  The main variables used in the parallel implementations are explained in Table 5.2.

Size	number of processes in a group
ID	the current process identifier
Packet	a buffer for storing packed data structures
Free_value	the free value to be received
SubList	the data structure used to store the unpacked packet
Dummy	a temporary storage for the result of function application
Count	a counter for number of elements processed
NoElements	total number of elements in the list data structure
Workers	number of workers to use
DataToProcess	number of data still not processed
SubResult	the processed task
Result	the output of the binary divide and conquer
Length1	number of elements for each node in the binary tree
Length2	the reminder of elements for an uneven distribution
TmpLength	used for temporary storage
TmpResult	used for temporary storage
LocalArg	local argument
NF	number of functions (stages) in the pipeline

Table 5.2: Definition of variables used in the parallel implementations.

5.5.1 A Parallel Implementation of map

5.5.1.1 Introduction

As mentioned in Section 3.3.2 each HOF has a C interface for its parallel implementation. The type declaration for the C function implementing the map operator along with its wrapper are presented in Appendix A.1. The role of the C interface is to provide a declaration to the **Camlot** compiler to enable it to generate closures for the full and partial application of the HOFs. Note that all nested HOFs will be partially applied apart from the outer HOFs. Therefore, to enter their closures in order to execute them they have to be applied to their last argument which is in all cases the data argument. This can be clearly seen from the declaration of the C interface. Only when a partially applied function is applied to its last argument is it entered by calling the wrapper for the C interface and encoding the arguments to the actual parallel implementation.

However, the general parallel implementation discussed previously had to be modified in order to support nesting. Basically, the schemes for group division and creation have been inserted into the process farm skeleton to reallocate the processes to support the nested skeletons.

5.5.1.2 The Process Farm Algorithm

Figure 5.8 presents the main algorithm for controlling the process farm. The argument Parallel determines if a parallel or sequential implementation will be used

```
Input:   Function: the function to be applied
         Data: the list data structure
         (the other arguments were explained in Section 5.5)
Output:  a processed list data structure.

ProcessFarm (Function, Data, Free, Split, Group, Parallel, Nest)
BEGIN
  If (Parallel) Then      /* parallel implementation */
    Size ← SizeofGroup Group
    If (Size EQUAL 1) Then Nest ← 0 EndIf
    ID ← GetID Group
    If (Nest) Then      /* nesting */
      Split ← ⌊ sqrt(Size) ⌋      /* compute split */
      CommGroup ← CreateTopCommGroup Group Split
      If root process Then
        Size ← SizeofGroup CommGroup
        Return Farmer(Data, ID, Size, CommGroup)
      Else
        SubGroup ← SplitGroup Group Split
        NestedWorker (Function, Free, CommGroup, SubGroup)
        ModifyGroup Group
        Return [ ]
      EndIf

/* CONTD */
```

Figure 5.8: The process farm algorithm I.

for evaluating the process farm skeleton. In case it is a parallel implementation, the execution continues and checks if the skeleton is nesting another skeleton by examining the Nest argument. If there is nesting then a communication group is created (which is called CommGroup) to connect all sub-groups by calling **CreateTopCommGroup**. The **Farmer** process will be called on the root process of CommGroup which will eventually return the final result for the process farm skeleton. The rest of the processes in the the root group will be divided into subgroups by calling **SplitGroup** and the **NestedWorker** routine will be executed on all these processes with access to both CommGroup and SubGroup. As the final result will be returned by the root process, the other processes will simply return an empty list to maintain a consistent type across all processes. However, since the Group argument was assigned to the root group in all partial applications of the skeleton wrappers, it is important to re-assign the Group argument to the CommGroup or SubGroup that will actually be used at run-time. This switching from one group to another is done at run-time as groups are generated and divided only at run-time. To do so, the original argument is modified by changing its value to the new value using the **ModifyGroup** routine.

In case no nesting was found then a flat implementation of the process farm will be used for evaluating the process farm skeleton as shown in Figure 5.9. The

Farmer will be executed on the root process and **FlatWorker** will be executed on the remaining processes. A sequential case can be implemented using the **smap** routine which is a sequential implementation of the **map** HOF.

```
/* CONTD */
        Else      /* no nesting */
            If root process Then
                Return Farmer (Data, ID, Size, Group)
            Else
                FlatWorker (Function, Group)
                Return []
            EndIf
    Else      /* sequential implementation */
        Return smap (Function, Data)
    EndIf
END.
```

Figure 5.9: The process farm algorithm II.

5.5.1.3 The Farmer Algorithm

Figure 5.10 illustrates the first part of the **Farmer** algorithm. It consists of two steps. In step 1, all local variables are initialised and the number of workers to be used is defined. For example, if the number of elements n in the list data structure is less than the number of workers p available then only n workers will be used. The remaining workers $p - n$ will be idle. In Step 2, the farmer process will send a packet to every available worker in the Group. For every transmitted packet, it has to retrieve an element from the list data structure, create a tuple which consists of two elements: the order of the retrieved element and the element itself, flatten the tuple and then send it to a process in the Group. The receiving process could be either the root process of a sub-group for a nested skeleton or a process which corresponds to a flat worker. Also, a number of variables are updated during this step to keep record of the number of messages being transmitted and the number of packets that still need to be transmitted.

The second part of the **Farmer** algorithm is illustrated in Figure 5.11. Here, the remaining un-processed elements, if any are left, are transmitted once a flat or nested worker is available. Therefore, the farmer at this stage waits to receive a processed packet (sub-result) from a worker or the root process of a sub-group, flatten the received packet and insert it in its correct position in the result vector. The **Farmer** retrieves the next element from the list data structure and distributes it using a similar method to that in the first part. This process of "distribute-work-collect" continues until no elements to be processed. Finally the **Farmer** waits for the final patch of sub-results to arrive from workers. Once the farmer processes all the elements of the list data structure it broadcasts a termination message to all its workers or the root process of sub-groups in case

```
Input:   Data: the list data structure to be processed
Output:  the processed list data structure

Farmer (Data, ID, Size, Group)
BEGIN
  1. NoElements ← DataToProcess ← ListSize Data
     Result ← VectorAllocate NoElements      /* initialisation */
     Count ← 0
     If Size > NoElements Then
        Workers ← NoElements
     Else
        Workers ← Size - 1
     EndIf
  2. For i from 1 to Workers Do
        SubList ← GetSubList Count 1 Data
        packet ← unpack SubList      /* send packets */
        Send packet to Process_i      /* to every */
        DataToProcess ← DataToProcess - 1 /* worker in group */
        Count ← Count + 1
     EndFor
/* CONTD */
```

Figure 5.10: The farmer algorithm I.

of nesting to terminate execution. The final step is to convert the result vector into a list data structure using the **MakeResultList** routine. The idea of using a vector to hold the sub-results is faster in processing as direct access is used instead of list traversal to reach a certain position on the list.

5.5.1.4 The Flat Worker Algorithm

Figure 5.12 illustrates the algorithm for a flat worker where the applied function to the received packets is a sequential function. The basic idea of the worker is to iterate infinitely until an abort message is received from the **Farmer** process. During the loop, the worker will receive packets from the farmer, unflatten them into a suitable data structure and apply its local function to them. Finally, it will pack the sub-results and send them back to the farmer. Note that the worker process preserves the order of the received packets by packing both the sub-result and its original position in the list data structure.

5.5.1.5 The Nested Worker Algorithm

The algorithm for the **NestedWorker** which is shown in Figure 5.13 proceeds as follows: It has two cases, the first case is to have the algorithm running on the root process of each SubGroup. In this case execution will resume until an abort is received from the root process of CommGroup (i.e the farmer process). Note that this part of the algorithm will execute on each root process of all SubGroups.

```
/* CONTD */
    3. If (DataToProcess > 0) Then
            While (Count < NoElements) Do
                packet ← Receive from Worker_i        /* receive results */
                SubResult ← unpack packet
                Insert SubResult into Result
                SubList ← GetSubList Count 1 Data
                packet ← pack SubList
                Send packet to Worker_i
                Count ← Count + 1
            EndWhile
        EndIf
    4. For i from 1 to Workers Do /* receive the remaining */
            packet ← Receive from Worker_x /* sub-results */
            SubResult ← unpack packet
            Insert SubResult into Result
        EndFor
        Broadcast termination message to all processes in Group
        Return MakeResultList Result NoElements
END.
```

Figure 5.11: The farmer algorithm II.

The routine will receive a packet (often called a task) from the farmer, unpack it into a data structure suitable for processing, broadcast a synchronisation message to all processes in one SubGroup. Next, it will broadcast the received packet to all processes in one SubGroup in case the Free argument is set to true, and apply its function argument to the received packet which will be subsequently flattened and transmitted back to the farmer. Once the termination message has been received from the root process in CommGroup, an abort message is broadcast to all processes in one SubGroup. Note that the inner (nested) skeleton will be entered when its partial application is applied to the received packet. Only at this point the closure for the inner skeleton is entered and the inner skeleton starts its execution where all processes in one SubGroup will be available as its resources. The SubResult is flattened and transmitted back to Farmer.

The other case is to have the algorithm on non-root processes in SubGroup as shown in Figure 5.14 where they will receive the synchronisation message from the root process in SubGroup. The next step is to resume execution until the abort message is received from the root process of SubGroup. If the Free argument was set to true then the free value has to be received, unflattened and the function argument is applied to it. Otherwise the function argument will be applied to an empty list to enter its closure and hence enter the inner (nested) skeleton. Note that only this set of non-root processes needs to have the free value and that is why it has to be transmitted by the root process. Each time the closure is entered to evaluate a new task the partial application has to be applied to a single argument and that is why only one free variable has to be sent.

```
Input:    Group: the set of processes available to the worker
          Function: the function to be applied to every received task
Output:   returns the processed data item to the farmer

FlatWorker (Function, Group)
BEGIN
    While (Abort message not received) Do
        packet ← Receive from Farmer
        SubList ← unpack packet
        SubResult ← Function SubList      /* function application */
        packet ← pack SubResult
        Send packet to Farmer
    EndWhile
END.
```

Figure 5.12: The flat worker algorithm.

5.5.2 A Parallel Implementation of fold

5.5.2.1 Introduction

The type declaration for the C function implementing the fold operator along with its wrapper are presented in Appendix A.2.

Note that the binary divide and conquer algorithm was used with minor differences to implement in parallel the following HOFs: **fold**, **Fold** and **SFold**. Therefore, for brevity, one illustration of the binary divide and conquer algorithm will be used to explain the parallel implementations for the previous HOFs. However, each HOF has a different C interface which is clear from the prototypes. Therefore, some arguments in Figure 5.15 are redundant as they are not used depending on the implemented HOF. For instance, the **fold** HOF can be used in **Ektran** in the following way:

```
(((fold Function0) Base) Data)
```

Here, only Function0 will be passed to the first argument of the algorithm outlined in Figure 5.15. The second and the third argument are not used. This is similar to the other HOFs.

5.5.2.2 The Binary Divide and Conquer Algorithm

The algorithm for the binary divide and conquer is shown in Figure 5.15 and it works as follows: the first part is to initialise all local variables to their values. The second part, is determined by the value of the Parallel argument. If it is set to true then a parallel implementation will be used for evaluation, otherwise a sequential implementation is used. In case of a parallel implementation and the size of group is one process then nesting is disabled by setting the Nest variable to False. The next step is to check the value of the Nest argument and

```
Input:   CommGroup: the group of processes used to
         connect the sub-groups to enable inter-group communication
         SubGroup: the set to processes available to the
         worker as resources
Output:  No Output

NestedWorker (Function, Free, CommGroup, SubGroup)
BEGIN
    Size ← SizeofGroup SubGroup
    ID ← GetID SubGroup
    ModifyGroup SubGroup
    If (root process in SubGroup) Then /* the root process */
        While (Abort message not received) Do
            packet ← Receive from root process in CommGroup
            SubList ← unpack packet
            Broadcast synchronisation message to all processes in SubGroup
            If (Free) Then
                Broadcast packet to all processes in SubGroup
            EndIf
            SubResult ← Function SubList
            packet ← pack SubResult
            Send packet to root process in CommGroup
            Broadcast abort message to all processes in SubGroup
        EndWhile

/* CONTD */
```

Figure 5.13: The nested worker algorithm I.

if it is set to true then the scheduler has to handle nested skeletons. Therefore, it creates a communication group (which is called CommGroup) to connect all sub-groups by calling **CreateTopCommGroup**. The routine **NestedMaster** will be executed on the root process of CommGroup and SubGroup (which is the same process as both groups share the root process) after passing to it the new size of SubGroup. The rest of the non-root processes of CommGroup will execute either **RootNestedWorker** or **NestedWorker** routines after dividing the original group into a number of SubGroups. The former will run on the root process of SubGroup and the later will run on the non-root processes of SubGroup. Note that there will be a number of SubGroups according to the value of Split where each SubGroup will form an instance of a skeleton. The routine **ModifyGroup** will be called to update the value of root group to its original value as the called sub-routines will update its value to the value of their local SubGroups. Finally the process which called the **NestedMaster** routine will return the final result while the other processes just return the Base value. Note that the returned value on non-processes has no effect to the final solution. It is only returned to have consistent type.

In case no nesting was found, then the **Master** routine is called on the root

```
/* CONTD */
    Else /* other processes in SubGroup */
        Receive synchronisation message from root process in SubGroup
        While (abort message not received) Do
            If (Free) Then
                packet ← Receive from root process in SubGroup
                free_value ← unpack packet
                dummy ← Function free_value /* function application */
            Else
                dummy ← Function [] /* function application */
            EndIf
        EndWhile
    EndIf
    Return []
END
```

Figure 5.14: The nested worker algorithm II.

process of Group and the routine **Worker** is called on all other processes. These two routines are used for a flat implementation. The last case is to have a sequential implementation where **sfold** is called for the sequential evaluation.

5.5.2.3 The Master Algorithm

The first part of the **Master** process illustrated in Figure 5.16 works as follows. The number of elements in the list data structure is found first. This value will be used to find if all or a number of the available processes in the Group will be used for evaluation. The number of processes used is broadcast to all processes within the Group. A sequential implementation will be used for evaluation if the group include only a single process. Otherwise, the size of each sub-list is determined by dividing the number of elements by the number of workers. The remainder and quotient of the division will be used to find the exact number of elements to send to each worker. The Master will keep one sub-list for local processing and call the routine **MasterToWorkers** to send to each node in the tree (which includes intermediate and leaf nodes) a sub-list of the list data structure. The data is transmitted in a breadth first order. This order is needed in order to produce correct results in case of a non commutative operator. A sequential function is applied to the local sub-list.

Figure 5.17 presents the second part of the **Master** routine. It has two cases. The first case is to have only one child worker where it will receive the sub-result and unflatten it. Next it will apply its local function to it and its local sub-result to generate the final result. The second case is to have two children where it will receive their sub-results and unflatten them. Next it will apply its local function to its local result and the sub-result received from the left child. Finally the local function is applied again to the outcome from the previous operation and the sub-result from the right child before returning the final result to the calling

```
Input:    Function: the binary operator
          Base: the base value
          Data: the list data structure to be processed
Output:   the result of applying the operator
          between each element of the list

BinaryDC(Function0, Function1, Function2, Base, Data,
            Free, Split, Group, Parallel, Nest)
BEGIN
ID ← GetID Group
Size ← SizeofGroup Group
If (Size EQUAL 3) Then Split ← 2 Else Split ← ⌊ sqrt(Size) ⌋ EndIf
If (Parallel) Then /* parallel implementation */
    If (Size EQUAL 1) Then Nest ← 0 EndIf
    If (Nest) Then      /* nesting */
        CommGroup ← CreateTopCommGroup Group
        If (root process of CommGroup) Then
            SubGroup ← SplitGroup Group
            ID ← GetID CommGroup, Size ← SizeofGroup SubGroup
            Result ← NestedMaster(Function0, Function1, Function2,
            Base, Data, Free, Parallel, ID, Size, CommGroup, SubGroup)
            ModifyGroup Group, Return Result
        Else      /* non-root processes */
            SubGroup ← SplitGroup Group
            ID ← GetID SubGroup
            If (root process of SubGroup) Then
                RootNestedWorker(Function0, Function1, Function2,
                        Base, Free, Parallel, ID, CommGroup, SubGroup)
            Else      /* non-root processes of SubGroup */
                NestedWorkers(Function0, Function1, Function2, Base,
                        Free, ID, SubGroup)
            EndIf
        ModifyGroup Group, Return Base
        EndIf
    Else          /* no nesting */
        If (root process) Then
            Return Master(Function0, Function1, Function2, Base, ID,
                        Size, Group)
         Else  Worker(Function0,  Function1,  Function2,  Base,  ID,  Group)
EndIf
        EndIf
    EndIf
Else      /* sequential implementation */
    Return sfold(Function0, Base, Data)              /* for fold */
            sFold(Function0, Function1, Base, Data)      /* for Fold */
            sFold(Function1, Function2, Base, Data)      /* for SFold */
EndIf
Return Base
END.
```

Figure 5.15: The main scheduler for the binary divide and conquer algorithm.

```
Input:   ID: process identifier within Group
         Size: number of processes in Group
Output:  returning result to BinaryDC

Master(Function0, Function1, Function2, Base, Data, ID, Size, Group)
BEGIN
NoElements ← ListSize Data
If (Size > NoElements) Then Size ← NoElements EndIf
Broadcast Size to all processes in Group
If (Size EQUAL 1) Then
    Return sfold(Function0, Base, LocalArg)          /* for fold */
        sFold(Function0, Function1, Base, LocalArg) /* for Fold */
        sFold(Function1, Function2, Base, LocalArg) /* for SFold */
EndIf
Length1 ← ⌊ NoElements / Size ⌋
Length2 ← NoElements mod Size
TmpLength ← Length2
If (Length2 EQUAL 0) Then
    LocalArg ← GetSubList 0 Length1 Data
Else
    LocalArg ← GetSubList 0 (Length1 + 1) Data
EndIf
MasterToWorkers (0, Length1, Length2, Size, Group, Data)
TmpResult ← sfold(Function0, Base, LocalArg)          /* for fold */
TmpResult ← sFold(Function0, Function1, Base, LocalArg)/* for Fold */
TmpResult ← sFold(Function1, Function2, Base, LocalArg)/* for SFold */
/* CONTD */
```

Figure 5.16: The master algorithm I.

process.

5.5.2.4 The Worker Algorithm

The algorithm for the first part of the **Worker** process is illustrated in Figure 5.18. It works as follows: the number of workers in the tree involved in the evaluation is received from the Master process. Here, all processes with IDs greater than the number of processes involved in the evaluation will receive a termination message from the Master process. The processes involved in the evaluation will be either leaf or intermediate nodes. In case of leaf nodes they will receive their sub-lists from the Master process, unflatten them and apply the local function to them. Next, it will flatten the sub-result and send it to its parent process. In case of intermediate nodes, they will receive their sub-lists from the Master process, unflatten them and apply their local function to them. All intermediate nodes will have either one or two child workers as shown in the second part of the algorithm in Figure 5.19. In the case of one child then it will receive its sub-result and apply its local function to its local result and the received sub-result. The

```
/* CONTD */
    If ((2*ID + 1) EQUAL (Size - 1)) Then      /* one child worker*/
        packet ← Receive from child worker within Group
        ChildResult1 ← unpack packet
        Result ← Function0 TmpResult ChildResult1      /* for fold, Fold */
                                        /* and SFold */
    Else      /* two children workers */
        packet ← Receive from left child worker within Group
        ChildResult1 ← unpack packet
        packet ← Receive from right child worker within Group
        ChildResult2 ← unpack packet
        TmpResult ← Function0 TmpResult ChildResult1      /* for fold, */
        Result ← Function0 TmpResult ChildResult2 /* Fold and SFold */
    EndIf
    Return Result
EndIf
```

Figure 5.17: The master algorithm II.

other case is to have two child workers where the intermediate node will receive both sub-results. The local function is applied to the local result and the result of the left child worker and then to the result of the right worker. Next, the intermediate node will flatten the final result and send it to its parent process.

5.5.2.5 The Nested Master Algorithm

Figure 5.20 illustrates the **NestedMaster** routine. It works as follows. First this routine will be placed on the root process of both CommGroup and SubGroup (which can be seen as the root process of the root group). It will coordinate the work between the sub-groups. A number of initialisation steps are carried out on the local variables. The number of sub-groups involved in the tree is transmitted to all sub-groups (which the processes of CommGroup). Also, all processes that belong to SubGroup will receive a synchronisation message from this process. Only one sub-group will be used in the evaluation if the size of CommGroup is one process or there is only one sublist in the list data structure. If this is not the case then a similar technique to the **Master** routine is used to divide the list into sub-lists. However, the sub-lists are broadcast to the root process of each sub-group using the CommGroup (i.e., inter-group communication). This is achieved by passing the group CommGroup to the routine **MasterToWorkers** instead of Group as in the **Master** process. The next step is to find the size of the local sub-list and broadcast this value to all processes in SubGroup. The value will be received by the non-root processes of SubGroup where the **NestedWorkers** process will be running on all of them. All non-root processes will also receive a copy of the free value in case the Free argument is set to True. Note that the free value in this case is the same local sublist which is available on the local

```
Input:    similar to Master
Output:   no output

Worker(Function0, Function1, Function2, Base, ID, Group)
Size ← Receive from Master within Group
If (Size EQUAL 1) Then Return Base EndIf
If (incoming message is not synchronisation message) Then
    packet ← Receive from Master within Group
    LocalArg ← unpack packet
    TmpResult ← sfold(Function0, Base, LocalArg) /* for fold */
    TmpResult ← sFold(Function0, Function1, Base,LocalArg) /* for Fold
*/
    TmpResult ← sFold(Function1, Function2, Base,LocalArg) /* for SFold
*/
    If (((2 * ID) + 1 ) >= Size) Then /* leaf node */
        packet ← pack TmpResult
        Send packet to parent process in Group
    Else          /* intermediate node */
/* CONTD */
```

Figure 5.18: The worker algorithm I.

process. Just before applying the local function to the local sublist, a call is done to the **ModifyGroup** with SubGroup as an argument to update the global variable Group. This is needed before the closures for the nested skeletons are entered. The partially applied HOF will be applied to the elements of the local sub-list. Hence the closure for the inner skeleton will be entered a number of times according to the value of CountFold.

After successful parallel evaluation, the **NestedMaster** process will receive the sub-results if any. It has two cases. The first is to have one child SubGroup where its sub-result is received. A similar process to the **Master** routine is done in receiving sub-results from workers. However, the CommGroup is used for communication. Note that the sub-results received were evaluated by a group of processes instead of a single process.

5.5.2.6 The Root Nested Worker

Figure 5.21 illustrates the algorithm for the first part of the **RootNestedWorker** process. It will be executed on the processes which happen to be non-root processes of CommGroup. Remember that **NestedMaster** is running on the root process of CommGroup (which is also the root process of one of the SubGroups). However, the routine will receive the number of SubGroups involved in the evaluation from the **NestedMaster** routine and broadcast the synchronisation message to all processes in the SubGroup. The routine terminates if it receives a termination message from the **NestedMaster** routine within CommGroup. If this is not the case then it will receive its local sub-list, unflatten it and find how many elements there are in the received sub-list. This value will be broadcast to all

```
/* CONTD */
    If (( 2 * ID + 1) EQUAL (Size - 1)) Then /* one child worker */
        packet ← Receive from child process in Group
        ChildResult1 ← unpack packet
        Result ← Function0 TmpResult ChildResult1
    Else        /* two children workers */
        packet ← Receive from left child worker in Group
        ChildResult1 ← unpack packet
        packet ← Receive from right child worker in Group
        ChildResult2 ← unpack packet
        TmpResult ← Function0 TmpResult ChildResult1 /* for fold, */
        Result ← Function0 TmpResult ChildResult2 /* Fold and SFold */
    EndIf
        packet ← pack Result
        Send Result to parent worker in Group
Else
    Receive abort message from Master within Group
EndIf
END.
```

Figure 5.19: The worker algorithm II.

processes in SubGroup. Also if the value of the argument Free is set to True then it will broadcast the received sub-list to all processes in SubGroup. The next step is to apply the local function to the local sub-list. At this point the routine could be the root process of a leaf or an of intermediate group. If it happens to be the first case then the local result is flattened and transmitted to the parent group using the group CommGroup. The second case as illustrated in Figure 5.22 can have another two cases. The first case is to have only one child sub-group where its sub-result is received and unflattened. The local function is applied to the local sub-result and the received sub-result. The second case is to have two children sub-groups where their sub-results are received and unflattened. The local function is applied to the sub-results in the following order: the local sub-result and the sub-result from the left sub-group then the sub-result from the right sub-group. Finally the final result on the intermediate group is flattened and transmitted to the parent SubGroup in CommGroup.

5.5.2.7 The Nested Workers Algorithm

The final routine involved in the parallel implementation of binary divide and conquer skeleton is illustrated in Figure 5.23. It will be executed by the scheduler on all non-root processes of SubGroup. Hence, it will not be executed on any processes that belong to CommGroup. It will receive the synchronisation message from the root process of SubGroup. If the synchronisation message is an abort the execution terminates at this point. If this is not the case, then the number of elements in the local sub-list in the root process of SubGroup is received which

```
 Input:    ID: process identifier within CommGroup
           Size: number of processes in SubGroup
 Output:   returning result to BinaryDC
NestedMaster(Function0, Function1, Function2, Base, Data, Free,
             Parallel, ID, Size, CommGroup, SubGroup)
BEGIN
NoElements ← ListSize Data
If (Size > NoElements) Then Size ← NoElements EndIf
    Broadcast Size to all processes in CommGroup
    Broadcast synchronisation message to all processes in SubGroup
If (Size EQUAL 1) Then
    Return sfold(Function0, Base, LocalArg)          /* for fold */
       sFold(Function0, Function1, Base, LocalArg) /* for Fold */
       sFold(Function1, Function2, Base, LocalArg) /* for SFold */
EndIf
    Length1 ← ⌊ NoElements / Size ⌋
    Length2 ← NoElements mod Size
    If (Length2 EQUAL 0) Then
       LocalArg ← GetSubList 0 Length1 Data
    Else
       LocalArg ← GetSubList 0 (Length1 + 1) Data
    EndIf
    MasterToWorkers(0, Length1, Length2, Size, CommGroup, Data)
    CountFold ← ListSize LocalArg
    Broadcast CountFold to all processes in SubGroup
    If (Free) Then
       packet ← pack LocalArg
       Broadcast packet to all processes in SubGroup
    EndIf
    ModifyGroup SubGroup
    TmpResult ← sfold(Function0, Base, LocalArg) /* for fold */
    TmpResult ← sFold(Function0, Function1,Base,LocalArg)/* for Fold */
    TmpResult ← sFold(Function1, Function2,Base,LocalArg)/* for SFold
*/
    If ((2*ID + 1) EQUAL (Size - 1)) Then      /* one child sub-group */
       packet ← Receive from child sub-group within CommGroup
       ChildResult1 ← unpack packet
       Result ← Function0 TmpResult ChildResult1 /* for fold, Fold and
                          SFold */
    Else      /* two children sub-groups */
       packet ← Receive from left child sub-group within CommGroup
       ChildResult1 ← unpack packet
       packet ← Receive from right child sub-group within CommGroup
       ChildResult2 ← unpack packet
       TmpResult ← Function0 TmpResult ChildResult1 /* for fold */
       Result ← Function0 TmpResult ChildResult2 /* Fold and SFold */
    EndIf
    Return Result
EndIf
```

Figure 5.20: The nested master algorithm I.

```
Input:   similar to NestedMaster
Output:  no output

RootNestedWorker(Function0, Function1, Function2, Base, ID,
             Free, Parallel, CommGroup, SubGroup)
Size ← Receive from NestedMaster within CommGroup
Broadcast Synchronisation message to all processes in SubGroup
If (incoming message is not synchronisation message) Then
    packet ← Receive from NestedMaster within CommGroup
    LocalArg ← unpack packet
    CountFold ← ListSize LocalArg
        Broadcast CountFold to processes in SubGroup
        If (Free EQUAL True) Then
            Broadcast packet processes in SubGroup
        EndIf
    TmpResult ← sfold(Function0, Base, LocalArg) /* for fold */
    TmpResult ← sFold(Function0, Function1, Base, LocalArg) /* for Fold
*/
    TmpResult ← sFold(Function1, Function2, Base, LocalArg) /* for SFold
*/
    If (((2 * ID) + 1) >= Size) Then /* root process of leaf group */
    packet ← pack TmpResult
    Broadcast packet to parent process in CommGroup
    Else            /* root process of intermediate group */
/* CONTD */
```

Figure 5.21: The root nested worker algorithm I.

is called CountFold. Also, if the argument Free has been set to true then the free value is assigned to the received packet from the root process otherwise is it assigned to an empty list. The next step is to apply the local function to the free value for a given number of times. This number is determined by the value of CountFold. Note that it is crucial to apply the partial application of the inner skeleton to a value (which might be the free value) to enter the closure for the skeleton. Therefore, the purpose of this routine is to work as an interpreter to enter the closures for the nested skeletons as many times as needed. Note the difference of the function applied for the different operators: fold, Fold and SFold.

5.5.3 A Parallel Implementation of Fold

Similar to the **fold** HOF, the **Fold** operator can be used in **Ektran** in the following way:

((((Fold Function0) Function1) Base) Data)

Here, Function0 and Function1 will be passed to the first and second arguments of the algorithm outlined in Figure 5.15, respectively. If Fold is nesting

```
/* CONTD */
    If (( 2 * ID + 1) EQUAL (Size - 1)) Then /* one child sub-group*/
        packet ← Receive from child process in CommGroup
        ChildResult1 ← unpack packet
        TmpResult ← Function0 TmpResult ChildResult1
    Else           /* two children */
        packet ← Receive from left child sub-group in CommGroup
        ChildResult1 ← unpack packet
        packet ← Receive from right child sub-group in CommGroup
        ChildResult2 ← unpack packet
        TmpResult ← Function0 TmpResult ChildResult1 /* for fold, */
        Result ← Function0 TmpResult ChildResult2 /* Fold and SFold */
    EndIf
        packet ← pack Result
        Send Result to parent sub-group in CommGroup
Else
Receive synchronisation message from NestedMaster within CommGroup
EndIf
END
```

Figure 5.22: The root nested worker algorithm II.

another HOF then it will be in Function1. The third argument passed to the algorithm is not used. Also, the type declaration for the C function implementing the Fold operator along with its wrapper are given in Appendix A.3.

5.5.4 A Parallel Implementation of SFold

Similar to the **fold** HOF, the **SFold** operator can be used in **Ektran** in the following way:

(((((SFold Function0) Function1) Function2) Base) Data)

Here, Function0, Function1 and Function2 will be passed to the first, second and third arguments of the algorithm outlined in Figure 5.15, respectively. Here, Function2 might contain another HOF where **SFold** is said to be nesting another HOF. Function0 will be used for recombining the sub-results and Functions will be used along with Function2 to fold the local sub-list. Also, the type declaration for the C function implementing the SFold operator along with its wrapper are given in Appendix A.4.

5.5.5 A Parallel Implementation of compose

5.5.5.1 Introduction

The type declaration for the C function implementing the compose operator along with its wrapper are given in Appendix A.5.

```
NestedWorkers(Function0, Function1, Function2, Base,
                         Free, ID, SubGroup)
BEGIN
    Receive synchronisation message from root process in SubGroup
    If synchronisation message is abort Then Return EndIf
        CountFold ← Receive from root process in SubGroup
    If (Free EQUAL True) Then
        packet ← Receive from root process in SubGroup
        free_value ← unpack packet
    Else
        free_value ← []
    EndIf
    While (CountFold > 0) Do
        dummy ← Function0 free_value      /* for fold */
        dummy ← Function1 free_value      /* for Fold */
        dummy ← Function2 free_value      /* for SFold */
        CountFold ← CountFold - 1
    EndWhile
END.
```

Figure 5.23: The nested workers algorithm.

5.5.5.2 The Main Scheduler for the Pipeline Algorithm

The main scheduler for the pipeline algorithm is presented in Figure 5.24. It is responsible for placing different routines on different processors. First it finds the size of its current group, the number of stages in the pipeline and the size of its list data structure. If only one process is in the group or the parameter Parallel is set to False then a sequential evaluation is used by calling the **mpipe** routine. Also, if the number of available processes is less than the number of stages in the pipeline then the control variable Nest is set to False. This initialisation has to be made as it is difficult to support nesting with the current available resources. For example, assume that each stage can be run in parallel on its own. To exploit the available parallelism in each stage then each stage needs to have access to more than one process which is difficult as the number of available processes is less than the number of stages in the pipeline. However, if more than one stage are placed on one process and forced to be executed sequentially by setting the control variable Parallel to False then it might be possible to run some of the other stages in parallel.

However, if the Nest argument is set to true then a group called CommGroup is created by calling the **CreateTopCommGroup** scheme. The size of this group is equivalent to the number of stages in the pipeline. Also the root group is divided evenly into a number of sub-groups by calling the **SplitGroup** scheme. On the root process of SubGroup the routine **RootPipe** will be executed and the ID of the calling process will be passed. Only the root process of CommGroup will have the final result and therefore only it will return the result to the main

```
Input:    a list of functions (FS) and data
Output:   the processed list data structure
MapPipe (FS, Data, Free, Split, Group, Parallel, Nest)
BEGIN
Size ← SizeofGroup Group
NF ← ListSize FS, NoElements ← ListSize Data
If (Size EQUAL 1) Then ModifyParallel 0 EndIf
If (Parallel) Then
    If (Size <= NF) Then Nest ← 0 EndIf
    If (Nest) Then
        CommGroup ← CreateTopCommGroup Group NF
        SubGroup ← SplitGroup Group NF
        If (root process of SubGroup) Then
            ID ← GetID CommGroup
            Result ← RootPipe(FS, Data, ID, CommGroup, SubGroup)
            ModifyGroup Group
            If (root process of CommGroup) Then
                Return Result
            Else
                Return []
            EndIf
        Else
            NonRootPipe(FS, SubGroup)
            ModifyGroup Group
            Return []
        EndIf
    Else
        ModifyParallel 0
        Result ← FlatPipe(FS, Data, Group)
        ModifyParallel 1
        Return Result
    EndIf
Else
    Result mpipe(FS, Data)
    ModifyParallel 1
    Return Result
EndIf
END.
```

Figure 5.24: The main scheduler for the pipeline algorithm.

program. All of the non-root processes will have the routine **NonRootPipe** running on them. However, if no nesting was found then a flat implementation will be used for evaluation. In this case, the variable Parallel is set to False to insure than no stage will be executed in parallel in case it has a call to a skeleton. This is important as each stage will have access to the process of the group. Therefore, it is possible for one of the stages to execute itself in parallel because it has access to more than one process. In fact this stage is placed only on one process and therefore it is important to have a sequential implementation of it. Notice that the pipeline itself is running in parallel where its stages are executed sequentially. After a successful execution the variable Parallel is set to true and the result is returned to the main program.

5.5.5.3 The Flat Pipe Algorithm

The first part of the algorithm for the **FlatPipe** is presented in Figure 5.25. It is a generalisation of the pipeline implementation discussed by **Wilkinson** and **Allen** in [169]. The algorithm first initialises its global variables. The most important part is to find how many functions (stages) there are in the pipeline and the number of processes in the Group. The comparison between these two variables generates three different cases. The first to have them equal where each process in the group will have a function assigned to it. The second is to have more processes than the number of functions in the pipeline where the extra processes will not be used. The third case is to have the number of processes less than the number of stages in the pipeline. In this case a number of functions have to be grouped together into a list and placed onto one process. The current partitioning used will evenly distribute the functions over the processes. Hence, each process will take a sublist of the functions list based on its ID and the number of functions assigned to it. The next step is to find where each process of the group is placed in the pipeline. This has three possibilities according to ID of each process: the first, the last or an intermediate process. The first process will send the number of elements it has in its list data structure to the last process in the Group. The next step is to retrieve one element from the list, apply the local function(s) to it, flatten the sub-result and send it to next process (which is ID + 1) in the pipeline. Once all elements have been processed and transmitted to the right process, an abort message is transmitted to the right process. The final step is to wait until the accumulated result on the final process is received, unflatten it and return it to the **MapPipe** routine.

The second possibility for the process is to be the last process in the pipeline as illustrated in Figure 5.26. In this case, the process will receive the number of elements in the list data structure and allocate a vector of that size. We use vector allocation as assigning fields in the vector which is much faster than traversing a list data structure to insert an element in it. The process will run infinitely until it receives an abort message from the immediate process to its left (which has ID equal to Size - 2). During the infinite loop, it will receive sub-results from its left process, unflatten them and apply it local function(s) to them. Also, the sub-results will be inserted into their ordered positions in the result vector. After

```
  Input:    similar to MapPipe
  Output:   returning result to MapPipe
FlatPipe (FS, Data, Group)
BEGIN
ID ← GetID Group
Size ← SizeofGroup Group
NF ← ListSize FS, NoElements ← ListSize Data
If (Size >= NF) Then
    Size ← NF
    If (ID < Size) Then FP ← GetSubList ID 1 FS EndIf
Else
    multi_functions ← True
    Length1 ← NF div Size
    Length2 ← NF mod Size
    If (ID >= Length2) Then
        FVEC ← GetSubList (ID * Length1 + Length2) Length1 FS
    Else
        FVEC ← GetSubList (ID * Length1 + ID) (Length1 + 1) FS
    EndIf
EndIf
If (ID EQUAL 0) Then      /* first process */
    Send NoElements to last process in Group
    While (Count NOTEQUAL NoElements) Do
        TmpArg ← GetSubList Count 1 Data
        ModifyParallel 0
        If (multi_functions) Then
            TmpRes ← pipe(FVEC, TmpArg)      /* apply list */
        Else
            TmpRes ← FP TmpArg      /* one function */
        EndIf
        packet ← pack TmpRes
        Send packet to process (ID + 1) within Group
    EndWhile
    Send Abort message to process (ID + 1) within Group
    packet ← Receive accumulated result from last process within Group
    Result ← unpack packet
    Return Result
Else
/* CONTD */
```

Figure 5.25: The flat pipeline algorithm I.

```
/* CONTD */
    If (ID EQUAL (Size -1)) Then        /* last process */
        NoElements ← Receive from first process within Group
        Result ← VectorAlloc NoElements
        While (Abort message not received) Do
                packet ← Receive from process (ID - 1) within Group
                TmpRes ← unpack packet
                If (multi_functions) Then
                    TmpRes ← pipe(FVEC, TmpRes) /* a list of functions */
                Else
                    TmpRes ← FP TmpRes /* a single function */
                EndIf
                Insert TmpRes into ResultVector
        EndWhile
                Receive Abort message within Group
                FinalResult ← MakeResultList ResultVector NoElements
                packet ← pack FinalResult
                Send packet to first process within Group
        Return []
/* CONTD */
```

Figure 5.26: The flat pipe algorithm II.

receiving the termination message, the final result is generated by creating a list data structure from the result vector. It will be flattened and transmitted to the first process in the pipeline. Finally the process will return an empty list to the **MapPipe** routine.

The last possibility is to have the intermediate process in the pipeline as illustrated in Figure 5.27. The process will go in an infinite loop until it receives an abort message from its immediate left process in the pipeline. During the loop, the process will receive partially processed packets from its immediate left process (which has ID - 1). It will unflatten them and apply its local function(s) to them. Also, it will flatten them and send them to its immediate right process in the pipeline. Once the abort message is received from the immediate left process, it will send an abort message to its immediate right process.

5.5.5.4 The Root Nested Pipe Algorithm

The first part of the **RootNestedPipe** algorithm is presented in Figure 5.28. It will be placed on the root process of the SubGroup which is the same as all processes of the CommGroup. After initialising the number of elements variable, the process ID of this process in CommGroup is broadcast to all processes in SubGroup. The ID will be used as an index to retrieve a function from the pipeline where all processes in a SubGroup have to run the same stage. Also, the local function is retrieved for this process from the list of functions. Based on the ID of the process in CommGroup, it could be the root process of the

```
/* CONTD */
If (ID < Size) Then      /* intermediate process */
        While (Abort message not received) Do
                packet ← Receive from process (ID - 1) within Group
                TmpRes ← unpack packet
                If (multi_functions) Then
                    TmpRes ← pipe(FVEC, TmpRes)
                    TmpRes ← FP TmpRes
                EndIf
                packet ← pack TmpRes
                Send Abort message to process (ID + 1)
        EndWhile
        Return []
        EndIf
EndIf
END.
```

Figure 5.27: The flat pipeline algorithm III.

first, last or intermediate group in the pipeline. In case it is the first group, then it will send the number of elements in the list data structure to the root process of the last group. The next step is to retrieve one element from the list, broadcast the synchronisation message to all processes in SubGroup and apply its local function to the local task. This function application will be the entry point to a stage which is possibly another skeleton. The returned sub-result will be flattened and transmitted to the root process of the immediate right group using the CommGroup for communication. Once all elements have been processed and transmitted to the neighboring stage, a termination message is broadcast to all processes within SubGroup. The final step is to wait until the accumulated result on the root process of the final group is received, unflattened and returned to the **MapPipe** routine.

As illustrated in Figure 5.29, another possibility for the algorithm is to be on the root process of the last group. In this case, the process will receive the number of elements from the root process of the first group. This will be used to allocate a vector of the needed size. The process will next go into an infinite loop until it receives an abort message from the root process of its immediate left group. During this time, sub-results are received from the process with (ID - 1) within CommGroup and unflattened. The synchronisation message is broadcast to all processes within SubGroup just before applying the local function to the received sub-result. Similar to the root process of the first group, this is the entry point to the final stage which is possibly a skeleton. After successful evaluation, the result is returned and inserted into the result vector. Once the abort message is received, the process creates a list data structure from the results vector using the **MakeResultList** routine. Finally, the results list data structure is flattened and transmitted to the root process of the first group. Also, a termination message is

```
RootNestedPipe(FS, Data, ID, CommGroup, SubGroup)
BEGIN
NoElements ← ListSize Data
Broadcast local ID to all processes in SubGroup
Size ← SizeofGroup CommGroup
FP ← GetSubList ID 1 FS
If (ID EQUAL 0) Then       /* root process of first sub-group */
    Send NoElements to last process in CommGroup
    While (Count NOTEQUAL NoElements) Do
        TmpArg ← GetSubList Count 1 Data
        Broadcast synchronisation message to all processes in SubGroup
        TmpRes ← FP TmpArg       /* function application */
        packet ← pack TmpRes
        Send packet to process (ID + 1) within CommGroup
        Count ← Count + 1
    EndWhile
    Send Abort message to process (ID + 1) within CommGroup
    Broadcast Abort message to all processes within SubGroup
    packet ← Receive accumulated Result from last process
                within CommGroup
    Result ← unpack packet
    Return Result
Else
/* CONTD */
```

Figure 5.28: The root nested pipe algorithm I.

broadcast to all processes in SubGroup.

The last possibility is to have the root process of an intermediate group as shown in Figure 5.30. The process will go into an infinite loop until it receives an abort message from the root process of its immediate left group. During the loop, it will receive sub-results from the root process of its immediate left group and unflatten it. Also, it will broadcast a synchronisation message to all processes within SubGroup. Next, it will apply its local function to the received task, flatten it and transmit it to the root process of its immediate right group. Finally, once the abort message is received it will broadcast the an abort message to all processes within SubGroup.

5.5.5.5 The Non Root Pipe Algorithm

Figure 5.31 illustrates the **NonRootPipe** algorithm which will be executed on all non-root processes of SubGroup. It will receive from the root process of SubGroup the index value. This value will be used to retrieve the function when applied it will be the entry point to a stage in the pipeline. The routine will apply infinitely its local function to an empty list until it receives an abort message from the root process of the SubGroup. The function application is used to enter the closure for the stage.

```
/* CONTD */
If (ID EQUAL (Size -1)) Then       /* root process of last sub-group */
    NoElements ← Receive from first process of CommGroup
    Result ← VectorAlloc NoElements
    While (abort message not received) Do
        packet ← Receive from process (ID - 1) within CommGroup
        TmpRes ← unpack packet
        Broadcast synchronisation message to processes in SubGroup
        TmpRes ← FP TmpRes       /* function application */
        Insert TmpRes into Result
    EndWhile
    FinalResult ← MakeResultList ResultVector NoElements
    packet ← pack FinalResult
    Send packet packet to root process within CommGroup
    Broadcast abort message to all processes within SubGroup
EndIf
/* CONTD */
```

Figure 5.29: The root nested pipe algorithm II.

Note that the even division of the group into sub-groups or grouping a number of functions on one process may cause load imbalance. A better division would be to split the processes unevenly according to a list of weights which presupposes the number of functions in the pipeline. However, this requires an estimation of the execution costs for each stage in the pipeline using arbitrary data sets. Algorithms like [166, 120] can be used to balance an arbitrary collection of stages on an arbitrary number of sub-groups or processes.

5.6 Related Work

The idea of using groups in the implementations of parallel programming languages have been used by other researchers. We will look at different implementations and compare them to the work presented in this chapter in Section 5.6.1. Also a number of compile-time approaches have been developed to flatten nested parallelism. In Section 5.6.2 a few of them are outlined and compared to the **Ektran** model.

5.6.1 Explicit Hierarchical Group Parallelism

NestStep [98] is an explicit parallel programming language for the BSP (Bulk Synchronous Parallel) model [163] of parallel implementations. It extends the Java programming language by adding language constructs and run-time support for the explicit control of parallel program execution and sharing of program objects. It synchronises between its subsets in order to nest the supersteps and this allows nested parallelism. See [80] for motivations for subset synchronisation

```
Else      /* root process of intermediate sub-group */
    While (abort message not received) Do
        packet ← receive from process (ID - 1) within CommGroup
        TmpRes ← unpack packet
        Broadcast synchronisation message to all processes within SubGroup
        TmpRes ← FP TmpRes      /* function application */
        packet ← pack TmpRes
        Send packet to process (ID + 1) within CommGroup
    EndWhile
        Broadcast abort message to all processes within SubGroup
EndIf
EndIf
EndIf
END.
```

Figure 5.30: The root pipe algorithm III.

```
NonRootPipe(FS, SubGroup)
BEGIN
    index ← Receive from root process within SubGroup
    Receive synchronisation message from root process within SubGroup
    FP ← GetSubList index 1 FS
    While (abort message not received) Do
        dummy ← FP []      /* function application */
    EndWhile
    Return []
END.
```

Figure 5.31: The non root pipe algorithm.

and arguments proposed against it. A NestStep program executes on all available processors where they form the so-called root group. Here, groups can be dynamically subdivided during program execution into subgroups, following the static nesting structure of the supersteps. However, this model has an explicit control over groups division which is an implicit feature of the **Ektran** model. In NestStep the programmer specifies the number of subgroups to be created. Also, the programmer has direct access to the attributes of a group which we think will complicate parallel programming. Besides, the programmer is responsible for the control flow of the program in order to avoid deadlocks in case one or more processors do not reach the (nested) step statement. However, the model gives an edge to load balancing where the programmer can specify a weight vector for non-uniform splitting of the current group in order to balance the work load in different sub-groups. This facility is not available in the **Ektran** model as we think the programmer should not be responsible for load balancing. In future work, would be possible to include such a feature using the run-time system where we can support load balancing.

Fork95 [99, 100] is an imperative parallel programming language intended to express algorithms for synchronous shared memory machines. It is based on ANSI C and offers additional constructs to hierarchically divide processor groups into subgroups and manage shared and private address subspaces. The model offers two programming modes: the synchronous and the asynchronous. In the synchronous mode, processors form groups that can be recursively subdivided into subgroups, forming a tree-like hierarchy of groups. Programs are executed by groups of processors, rather than by individual processors. Initially, there is just one group containing all available processors. Groups may be recursively subdivided. Thus, at any point of program execution, all presently existing groups form a tree-like group hierarchy. Only the leaf nodes of the group hierarchy are active. As subgroups of a group can be distinguished by their group ID, the user is responsible for assigning reasonable values to them when splitting the current group. Also, the model supports a number of parallel programming paradigms which are abstract enough to be compared to skeletons. These are, farming, parallel divide and conquer, and pipelining. However, the programmer has to annotate the computation models to indicate whether they are to be executed in synchronous or asynchronous modes. In case of an asynchronous mode, the programmer also has to insert a barrier statement to preserve the inter-task data dependencies. It is important to note that programming in **Fork95** allows the programmer to take advantage of the benefits of each category of hardware features. However, this will decrease portability of parallel programs as programs will be machine dependent and greatly increase the programming complexity. Similar to **NestStep**, this model is too explicit regarding group management when compared to the **Ektran** model.

ForkLight [101] is an imperative, task-parallel programming language for massively parallel shared memory machines. It is based on ANSI C, follows the SPMD model of parallel program execution, provides a sequentially consistent shared memory, and supports dynamically nested parallelism. The model uses a similar group concept to **Fork95**. In this model, recursively nested parallelism is supported by nested levels of control-synchronicity. The discussion mentioned above for **NestStep** and **Fork95** applies here as well. Also, the model requires the programmer for some cases to specify that splitting a group is unnecessary. Otherwise, the control flow could diverge for different threads at conditional branches such as if statements, switch statements and loops.

The **TwoL** [137, 67] programming model exhibits a two level structure with both data and task parallelism. The model provides the programmer with a specification language with appropriate constructs in order to express explicitly in the hierarchical module specification which module executions must be performed in a fixed order because of data dependencies and which module executions may be performed in parallel because they are independent from each other. Also, scheduling decisions are expressed explicitly by language constructs. Besides, the model requires the programmer to give a specification of the application program that expresses the maximum degree of task parallelism and provides a specification of the data-parallel basic modules. Nonetheless, the model automatically transforms hierarchically structured task parallelism into a corresponding message-passing program using a hierarchy of group contexts. The programmer is responsible for annotating the program to specify the processor groups that execute the data-parallel parts while this is transparent in the **Ektran** model. Also, the model is designed to solve problems mainly for numerical algorithms and the processor groups are recursively partitioned according to the structure of the composed modules [138]. Here, modules correspond to numerical methods where the scheduling of the modules is done explicitly using a number of constructors.

In the work of **Chandy et al** [36] a specific archetype has been introduced which deals with the integration of task and data parallelism using group communication. This organises processes into groups and defines group communication between and within these groups. The system has two constructs for groups: process groups (for representing SPMD computations) and networks of process groups (for task-parallel compositions of SPMD computations). However, to create a process group the programmer has to specify explicitly a number of parameters like: the number of processes in the group; the processors to be used; how processes are mapped to processors and how this process group communicates with other process groups. The model allows inter- and intra-group communication through explicitly-defined typed communication paths or channels. All of the above parameters are hidden from the programmer in the **Ektran** model. Also, the model does not allow arbitrary compositions of process groups and networks. It only handles one- and two-level nesting of them.

The work of **Hammond** [86] extends the concept of evaluation strategies [161] to support the SPMD model of parallelisation. Here, the model allows SPMD sub-programs to be composed with other parallel paradigms, or to be arbitrarily nested with other parallel paradigms in order to support the Group SPMD model. In this model, the programmer is responsible for annotating his program in order to give additional hints to the dynamic run-time system. The annotations include a communications function in order to expose all implicit communication in a sub-program. However, these are implicit in the **Ektran** model. Also, the above definition has not yet been mapped to a real SPMD implementation in order to validate the proposed methodology.

5.6.2 Flattening Nested Parallelism

The run-time system for **Ektran** supports nested parallelism by dynamically scheduling nested skeletons which involves run-time decomposition of nested data structures. Therefore, by executing the inner skeletons in parallel the nested data structure, which is passed through the arguments of the outer skeleton, is flattened or decomposed at run-time automatically. A number of approaches have been introduced to flatten nested parallelism using compile-time techniques. Note that nested parallelism can be seen as a partial solution to the skeleton nesting problem and suffers from problems with respect to efficient implementations on distributed-memory machines [157]. Below we will look briefly at these approaches and the end compare them to the **Ektran** model.

Nesl [15] can be thought of as the first parallel programming language that implements nested parallelism. It supports nested data parallelism by mapping it to flat data-parallelism where it only supports vectors (i.e., homogeneous, ordered sequences). The intermediate language is called **VCODE** [14] and it has its own set of operators that work on segmented vectors. The key idea in the flattening process is that for every nested data-parallel program which uses a restricted set of operators there exists a corresponding flat data-parallel program which uses segmented vectors, and furthermore this flat data-parallel program can be realised with a restricted set of primitive operations that are specified by the so called scan vector model (SVM). However, **Nesl** does not support higher-order functions with nested parallelism as in **Ektran**.

There has been considerable interest in flattening nested parallelism by other researchers. For example, the work of **Palmer et al** [122] described an approach to realise work-efficient data-parallel indexing based on bounding the amount of replication and randomly dispersing concurrent reads. Also, nested data parallelism has been integrated into an object-oriented language in the work of **Sheffler** and **Chatterjee** [149]. The strength of the method lies in the compilation strategy, which translates nested data-parallel C++ into ordinary C++. The language **V** [32] integrates nested data parallelism into imperative languages using the example of C. Because of the imperative context, new problems like pointer lifting and handling control structures occur during the transformation. However, the last two approaches described above are imperative by nature and not strictly compatible to **Ektran**. More recent work like **Keller** and **Chakravarty** [102] looked at supporting user-defined recursive types, which allow parallel tree structures and how to flatten them. See **Riley et al** [140] for a clarification of a number of issues which may arise in the implementation of a "flattening" compiler for a nested-parallel programming language. The work of **Engelhardt** and **Wendelborn** [55, 54] proposes an execution model for nested-data parallelism based on multi-threading.

The compile-time flattening of nested parallelism reduces the overhead associated with run-time flattening compared to the run-time system for **Ektran**. However, these approaches are based on restrictions to the forms parallelism may take (for example parallel operations may only be applied to sequences as in **Nesl** [15]). Such limitations greatly ease both static analysis of programs and data communication between parallel control structures but necessarily constrain the language's parallel expressiveness.

5.7 Summary

We have presented the run-time framework used for executing combined skeletons in parallel. The mechanism operates at run-time but relies on the static-analysis done at compile time and uses message-passing groups to run combined skeletons in parallel. By running nested skeletons in parallel, the nested list data structure is decomposed dynamically. This has the advantage of making the system have a dynamic behaviour instead of a static one. In the next chapter, we will evaluate the methodology by presenting examples that contain combined HOFs.

Chapter 6

Experimental Environment

6.1 Introduction

This chapter presents the research apparatus used to evaluate the **Ektran** model for parallel programming. This consists of target architectures, benchmark programs and software tools that provide an experimental environment in which questions about performance issues of combined skeletons can be answered. The questions are about the behaviour of the benchmark programs across different platforms and performance results that arise when combined HOFs are implemented sequentially or in parallel. The apparatus must be able to address all these questions.

The target architectures (where all are MIMD systems) used for experiments are described in Section 6.2. In Section 6.3, the benchmark programs are presented along with their performance results. Also, the factors that affect the performance of nested skeletons are discussed and the notation used to label the experiments is outlined.

6.2 Target Architectures

6.2.1 Fujitsu AP1000

The Fujitsu AP1000 is a distributed memory parallel computer [70]. Processor elements, called cells, are connected with three networks called the broadcast network (B-Net) with data transfer rate of 50 MB/s, torus network (T-Net) with bandwidth of 25 MB/s, and synchronisation network (S-Net). The (T-Net) is a two dimensional worm-hole routed network connecting the processors in a torus topology. It provides the infrastructure for point-to-point communication operations between the cell processors. The (B-Net) and the (S-Net) provide support for efficient global communication and synchronisation operations between the processors. A SUN 4 workstation controls the AP1000 as a host computer.

The configuration of the machines used for the experiments comprises 128 scalar Sun Sparc-1 (25 Mhz) processors each with 16 MB of RAM and 128 KB of Cache. The ANU MPI [71, 150] implementation for AP1000 was used in the experiments.

6.2.2 Fujitsu AP3000

The Fujitsu AP3000 parallel server is a distributed memory parallel server based on powerful 64-bit UltraSPARC technology. The nodes of the AP3000 are connected by Fujitsu's AP-Net, a dedicated high-speed network specifically designed to provide high-speed internal data transfer between processing nodes. This is a wormhole routed network connecting the processing nodes in a 2-D Torus configuration. The data transfer of the network is 200 MB/s x 2 (bi-directional) for each port (4 ports/node). The AP3000 has a variety of processing nodes: 60 UltraSPARC 300MHz processors (U300), 16 UltraSPARC 200MHz processors (U200) and 4 UltraSPARC 200MHz dual processors. Each node has a local file space of 4GB 256 or 128 MB of main memory and it runs the Solaris 2 operating system. The LAM 6.1 [27] implementation of MPI was used for the experiments.

6.2.3 Cray T3D

The T3D is a distributed memory multicomputer based around a high performance three dimensional interconnect network [42]. The six channels out of each interconnect node are able to simultaneously support hardware transfer rates of 300 MB/s. The cost of routing data between processors through interconnect nodes is essentially negligible. The penalty is two clock cycles per node traversed and one extra clock cycle to turn a corner.

Each interconnect node hosts two independent processing elements, consisting of a DEC Alpha 21064 processor and 64 MB of RAM. The processor is clocked at 150 MHz, supports 64 bit integer and IEEE floating point operations, and delivers a peak performance of 150 64-bit Mflops/s. The processor has an 8 KB direct mapped data cache and an 8 KB instruction cache. The interconnect interface hardware of the T3D supports a single address space across the machine where every processor has direct access to the memory of every other processor. Each node runs the UNICOS operating system.

Edinburgh Parallel Computing Centre (EPCC) and Cray Research Incorporated have developed an implementation of MPI for the Cray T3D [96]. It was built on top of the basic communication routines like remote memory read and write. The T3D C compiler was used to generate target code.

6.2.4 Network of Workstations (NOWs)

It was possible to set up a distributed memory parallel architecture using a network of PCs running the Linux operating system using the LAM 6.1 MPI [27] implementation. The machine consisted of heterogeneous PCs where 8 of them were P90 with 16MB RAM and the rest were P166 with 32 MB RAM. The PCs were connected using an Ethernet network with 10 MB/s bandwidth.

6.2.5 Meiko Computing Surface

The Meiko computing surface is a distributed memory parallel machine based on transputers (T800). Each node has 4 MBs of local memory a 25MHz clock speed and four links to the surrounding nodes. The CHIMP implementation of MPI was used in the experiment. The machine was subsequently discarded because of its low resources.

6.3 Benchmark Programs

Benchmark programs define the constraints surrounding each question posed in the course of scientific investigation of computer systems. Given the self-imposed restriction of the Ektran model, these benchmarks are small and do not solve "real" problems. However, every effort has been made to select a set of codes that, individually, test the different HOFs and their combinations. A brief description of each benchmark program used in this study is given in the sections that follow.

However, the garbage collector which came with the run-time system for the Camlot compiler caused major problems on machines that did not have 100 % UNIX operating system like the Cray T3D and Fujitsu AP1000. Therefore, it was difficult to run experiments with large data sets and it was very hard to look at the garbage collector within the time constraints for the project. A solution to this problem would be to use a newer implementation of Caml like Objective Caml [107] which we hope causes less problems.

6.3.1 Testing the Software Tools

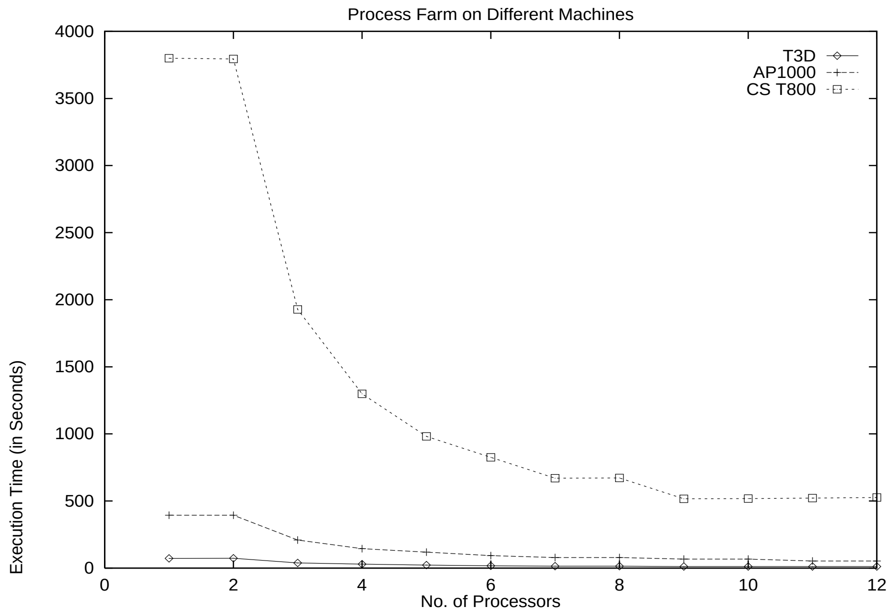

Figure 6.1: Processor farm behaviour on Cray T3D, Fujitsu AP1000 and Meiko CS with T800s.

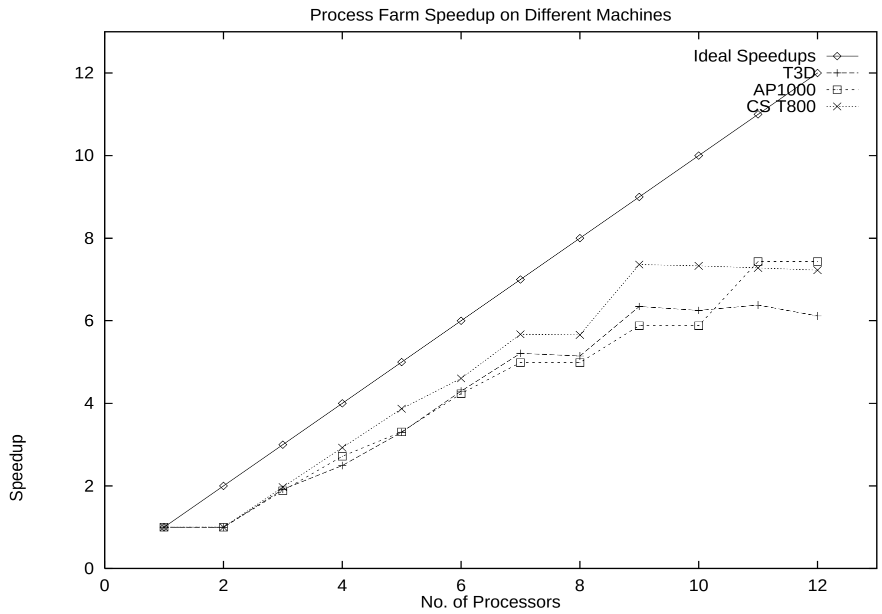

Figure 6.2: Processor farm speedup on Cray T3D, Fujitsu AP1000 and Meiko CS with T800s.

First of all we tested the software tools used for the experiments, that is the MPI library and the C compilers used to generate the target code for each parallel machine. The purpose of the experiment was to validate MPI's robustness and check that consistent behavior can be achieved on a number of different architectures. Figures 6.1 and 6.2 show the behavior and speedup for a process farm skeleton under a dummy processing load. The results show that the process farm had a similar behavior on the Cray T3D and Fujitsu AP1000. Variations in execution time were related to the behavior of the process farm itself rather than the communication library. Also, the results suggest that code portability as well as performance portability can be achieved [127]. However, the behaviour of the process farm on the Meiko CS T800 was poor because of the large overhead for MPI's communication routines. In fact MPI for this machine was built on top of CS tools and this extra layer caused major overheads. Therefore, this machine was excluded from further experiments.

6.3.2 Experimental Parameters

In the experiments, three factors were found to have a big effect on the performance of the example programs. These were:

- The division of a group into a number of sub-groups. As mentioned before it is possible to divide a group into a number of fixed size sub-groups using different heuristics. For the experiments the default was to use the square root heuristic which is denoted by **h1**. Another heuristic was also used to divide a group into only two sub-groups and is denoted by **h2**. The second heuristic is used to push the resources to the bottom HOFs of the nesting structure. Note that **h1** is the default heuristic used by the run-time system had to be modified manually for other experiments.

- Granularity of the nested list data structure. It is outlined in the experiments how granularity affects performance.

- The level to which the nested skeleton were implemented in parallel. The notation used is **1P**, **2P** and **3P** which denotes implementing in parallel the first one, two or three HOFs in the nesting structure. The current implementation of Ektran exploits parallelism in nested skeletons to the maximum level as no analyser is available at the moment. Therefore, the intermediate (Caml) code had to be manually modified by setting the Nest flag to false at the last level to be implemented in parallel. The runtime system will then run all nested skeletons beyond this level sequentially.

Also, it is possible to combine more than one factor in one experiment. For example the label "2P-h1" on a graph means implement the first two HOFs in parallel and use the square root of the group size for splitting the group into sub-groups.

The question how to divide a group into subgroups and how many levels of nested parallelism should be implemented in parallel should be done by the parallelising compiler. In this case the compiler needs to consult cost models in order to find the optimal option regarding these parameters. This might involve trying all the possible options by searching the search space and looking for the cost effective implementation as in the work of **Rangaswami** [136].

6.3.3 Merge Sort using a Pipeline

This is the example previously outlined in Section 3.2.2.3 and its code was shown in Figure 3.1. The program works on Arbitrarily Large Numbers (ALN) [141]. Here, numbers are represented as lists of decimal digits. Each number consists of 40-digits. The numbers are generated at run-time and are stored in a deeply nested list data structure with the following dimensions. The outer list consists of 10 sublists. Each sublist has another 10 sublists. The inner sublist has also 10 sublists. This gives in total 1000 40-digit numbers. The execution time (in seconds) and speedup results are shown in Table 6.1 and Figure 6.4. The top level code for the example program where skeletons were used is shown below:

```
def merge1 l2:llli = (((fold merge) []) l2);
def sortmap l3:llli = ((map sort) l3);
def sortll ll = (((@ merge1) (map ((@ merge1) sortmap))) ll);
   (sortll (((((gen4 10) 10) 10) 9) 40));
```

In this example, we use a heterogeneous pipeline which consists of two stages: the first uses a **map** HOF and the second uses a **fold** HOF. Figure 6.3 shows the skeleton nesting structure for this program. The novelty here is in running the inner stages of the pipeline in parallel as well. Also, this example demonstrates

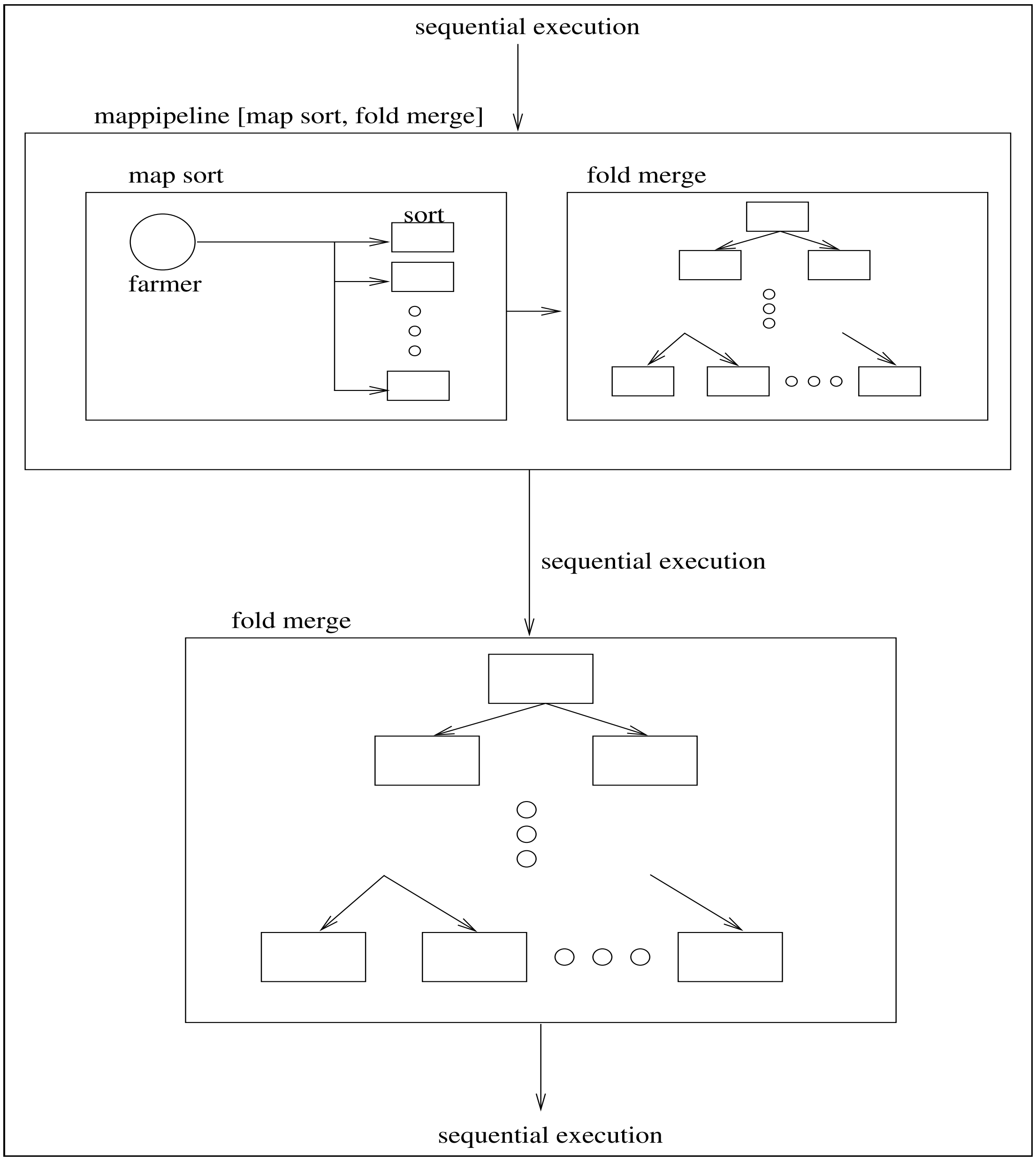

Figure 6.3: The skeleton nesting structure for the merge sort using a pipeline program.

No. of Proc.	1	2	3	4	5	6	7	8	9	10
Time (in sec.)	39.8	26.76	29.69	29.97	22.1	21.9	18.83	19.0	19.3	21.27
Speedup	1	1.49	1.34	1.48	1.8	1.82	2.11	2.1	2.06	1.87

Table 6.1: Merge sort using a pipeline, execution time and speedup on the AP1000.

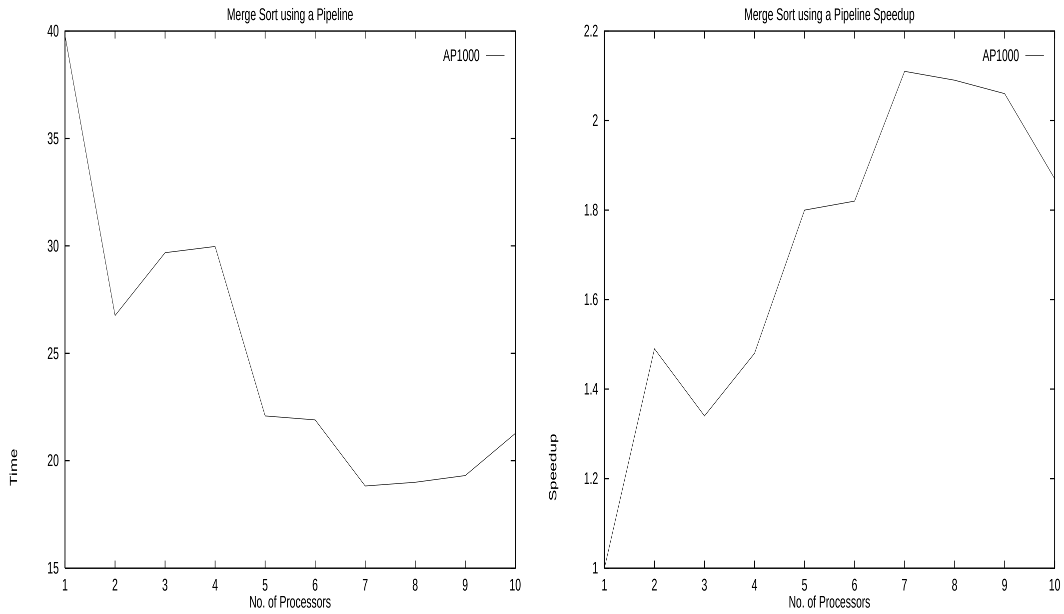

Figure 6.4: Merge sort using a pipeline, execution time and speedup on the AP1000.

the integration of task and data parallelism [33, 167] using the compose HOF. Here, the stages of the pipeline correspond to control or task parallelism and the parallel implementation of the stages themselves corresponds to data-parallelism. However, the results are disappointing as the best speedup was 2.11 on 7 processors. Nonetheless, this speedup suggests that running the inner stages in parallel improves the performance as speedup for 2 processors was 1.49. We think there are three reasons for the poor performance: 1) the parallelism exploited in the inner stages of the pipeline is poor. This might be caused by the cost to transmit a task to a worker is very high compared to the cost of processing the task. In other words the communication to computation ratio is high. We think this is caused by the parallel implementation of the fold HOF, 2) there are only two stages in the pipeline for this example and 3) it is possible that the pipeline is not load-balanced (i.e one stage needs more or less processors than other stages). Recall that the current parallel implementation of the compose HOF divides evenly the available processors into sub-groups.

It might be possible to improve the available parallelism by the merge sort program by 1) increasing the data size which was not possible due to memory problems with the garbage collector and 2) improving the parallel implementation of the **fold** HOF in order to reduce the communication overhead that is caused by transmitting the sublist to the workers.

6.3.4 Matrix Multiplication

We have chosen a well-known problem for evaluating our system for nesting skeletons. This is the problem of multiplying two matrices $A_{m\times n}$ and $B_{n\times k}$, resulting in the matrix $C_{m\times k}$. Also, we have added an extra level of complexity to the

Processors	1	2	3	4	9	10	16	17	25
map map	953.66	1058.92	601.12	535.76	268.85	231.67	116.26	115.88	78.15
map SFold	980.8	1060.97	607.24	454.28	198.0	192.4	91.26	91.3	91.8
SFold SFold	986.14	551.24	489.72	323.99	179.16	142.75	91.67	91.55	91.56
SFold map	976.57	546.8	545.38	545.38	234.19	176.56	119.35	119.34	79.55
Processors	26	36	37	49	50	56	64	65	
map map	77.5	77.22	77.1	40.76	40.57	40.18	21.30	21.3	
map SFold	91.37	89.1	88.6	43.32	22.94	22.95	22.96	22.96	
SFold SFold	90.96	81.83	49.41	23.82	23.82	23.82	23.82	23.82	
SFold map	79.61	78.98	78.83	41.45	41.45	21.95	21.95	21.95	

Table 6.2: Matrix multiplication execution time in seconds for the four different cases on the Fujitsu AP1000.

problem as we use arithmetic on ALNs. In a pure functional setting, a matrix of ALNs may be represented by a list (rows) of lists (column values) of integer lists (ALNs). The complete code for performing the matrix multiplication is shown in Appendix C.2. Figure 6.5 shows the skeleton nesting structure for the top level code of the program shown below:

```
def MatrixMul m1 m2 = ((map (map scalar)) ((combine m1) (transpose
m2)));

  ((MatrixMul ((((gen3 7) 7) 9) 150)) ((((gen3 7) 7) 9) 150))
```

First of all, the second matrix is transposed and then combined with the first matrix. The process of combination will generate a nested list data structure. The bottom list contains a pair of ALNs. The process of combination generates a list of lists of pairs. A pair is represented as tuple of two elements, where the first element is a row of the first matrix and a column of the second matrix. All the possibilities for combining a row with the columns are generated. Hence, the number of pairs in the inner list will depend on the number of columns in the second list and the number of sublists in the outer list will depend on the number of rows in the first matrix. Parallelism is exploited in the example by applying a scalar operation over the nested list data structure using two nested maps. Also the example is used to study the behavior of programs with two levels of nesting. Here the operation will take a pair of two numbers and find their scalar product. The scalar product is performed using recursive functions for adding and multiplying ALNs.

For this program, it was possible to replace one of the two maps with the SFold HOF without affecting the final result. Therefore, this swap gave four different cases for the same program: map nesting map (case 1), map nesting SFold (case 2), SFold nesting SFold (case 3) and SFold nesting map (case 4).

Tables 6.2 and 6.3 and Figure 6.6 show the execution time and speedup for the four different cases on the Fujitsu AP1000. The results are for multiplying two matrices where each has 7 rows and 7 columns with 150-digit ALNs. The results are analysed according to the four different cases. For the first case, the

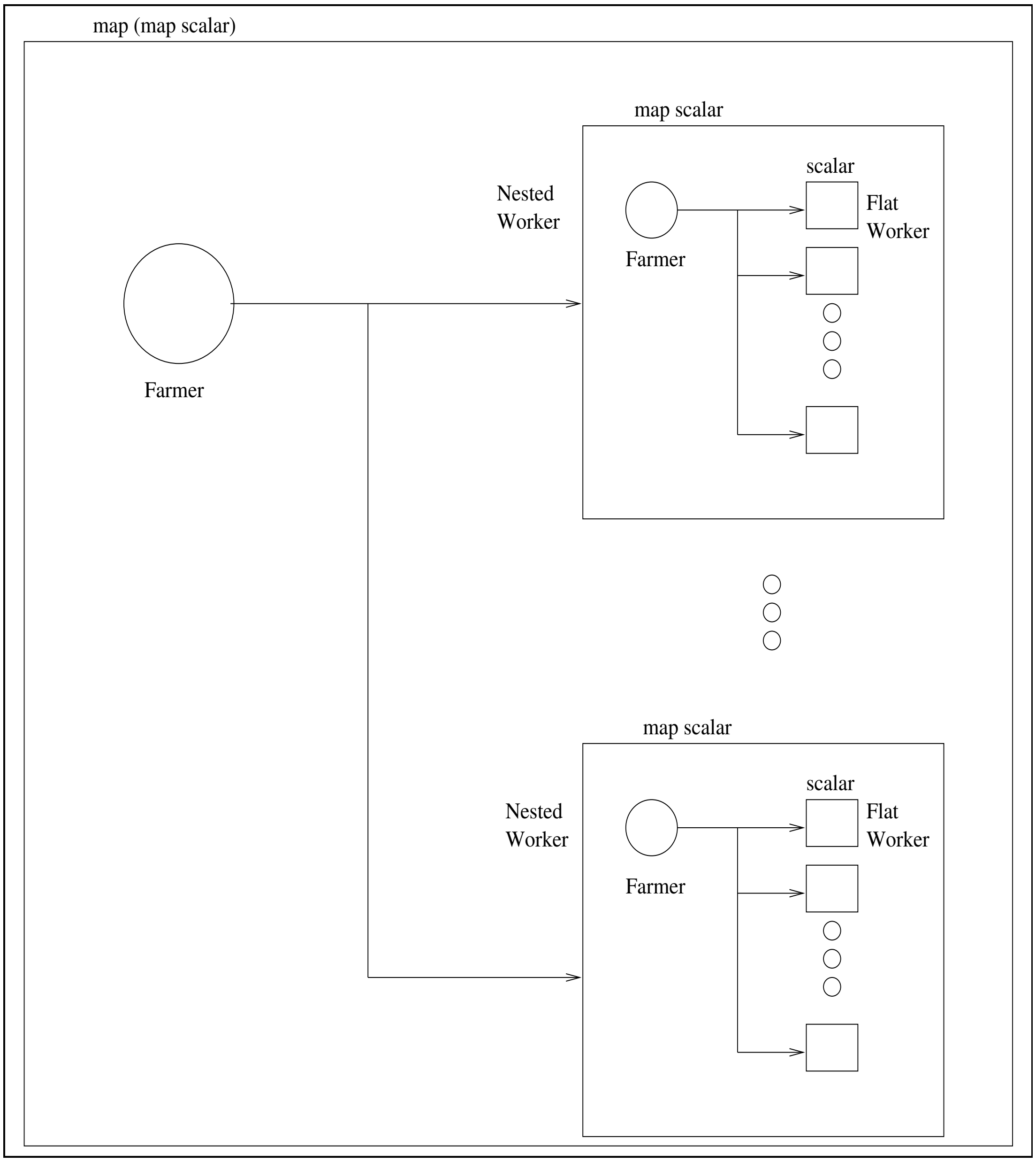

Figure 6.5: The skeleton nesting structure for the matrix multiplication program.

Processors	1	2	3	4	9	10	16	17	25
map map	1	0.9	1.59	1.78	3.55	4.12	8.2	8.23	12.2
map SFold	1	0.9	1.6	2.16	4.95	5.1	10.75	10.74	10.68
SFold SFold	1	1.79	2.01	3.04	5.5	6.9	10.75	10.77	10.77
SFold map	1	1.79	1.79	1.79	4.17	5.5	8.18	8.18	12.27
Processors	26	36	37	49	50	56	64	65	
map map	12.3	12.34	12.37	23.4	23.5	23.7	44.75	44.76	
map SFold	10.73	11.01	11.07	22.6	42.76	42.74	42.7	42.72	
SFold SFold	10.84	12.05	19.96	41.4	41.4	41.4	41.4	41.4	
SFold map	12.27	12.36	12.39	23.56	23.56	44.49	44.49	44.49	

Table 6.3: Matrix multiplication speedup for the four different cases on the Fujitsu AP1000.

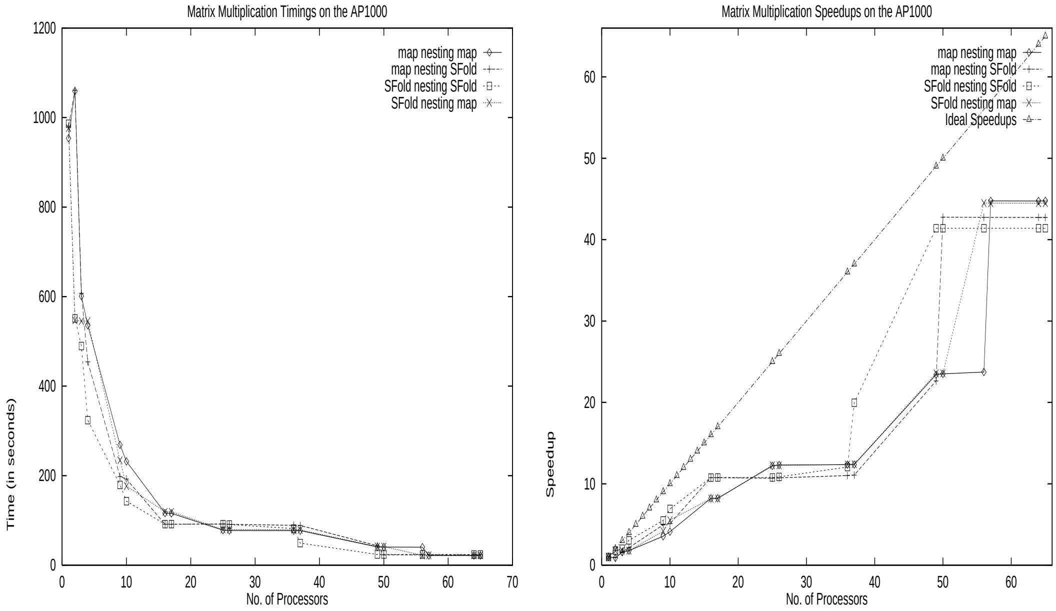

Figure 6.6: Matrix multiplication execution time and speedup for four different cases on the Fujitsu AP1000.

speedup results are poor for a small number of processors. It suddenly jumps to 23.4 on 49 processors after staying flat for between 25 and 37 processors. The reason for this jump is the availability of more processors to the program and allocating more resources to the groups. Remember that a process farm needs one process for its farmer and the rest of the processes are workers. Therefore, the remaining 48 processors will be divided into 6 sub-groups where each sub-group has 8 processors. However, since the outer list data structure has 7 sublists, one of the sub-groups will receive two sub-lists rather than only one. This will cause load-imbalance of the sub-groups where 5 sub-groups have processed only one sub-list while one sub-group has processed two sub-lists. Therefore, more time is needed to finish processing the tasks. Increasing the number of processors to 57 improved the speedup up to 44.7 which was the best we obtained for this case by having an efficiency of 78 %. This number of processors caused a good load balance for the program by splitting the 56 processors (after allocating one as the farmer of the process farm) into 7 sub-groups where each sub-group has exactly 8 processors. The inner process farm will allocate on process to the farmer and the rest as workers. Recall that we have 7 sublists where each sublist has 7 sub-lists as well. Hence the outer 7 sublists will be processed by the 7 sub-groups where each sub-group will process the inner 7 sub-lists. This perfect allocation of tasks to processes caused the good speedup and hence adding more processors after this point will not improve the results as the speedup stays flat beyond 57 processors. Note that 8 processors out of 57 will be farmers with the rest as workers which gives an 86 % efficiency in total.

For case 2, there were two big jumps in terms of speedup: from 11.1 on 37 processors to 22.64 on 49 processors and later to 42.76 on 50 processors. The last speedup gave an efficiency of 85.3 %. The jump resulted from the even division of the group. For example, on 50 processors, 1 process will be allocated to the outer process farm and the rest 49 processors will divided into 7 sub-groups where each has 7 processors in it. This will cause a good load balance and the list data structure will be mapped evenly to the workers of the binary divide and conquer skeleton. Note that speedup stays flat beyond 50 processors as adding more resources will not improve the performance. Also, the one processor which has been allocated as a farmer to the process farm will not do any work.

For case 3, the best speedup was 41.4 on 49 processors which gave an efficiency of 84.3 %. The reason for such a quick speedup compared to the other cases is that 49 processors are active working on the list data structure. Here, the 49 processors will be evenly divided into 7 sub-groups each of 7 processors. The speedup stays flat beyond 49 processors as the new resources will not be utilised.

Finally, for case 4, there is a jump in speedup from 23.56 on 49 processors to 44.49 on 56 processors which gave an efficiency of 79.5 %. Note that on 49 processors the inner skeleton will have 7 processors available to it. Since the inner skeleton is a process farm then only 6 processors will be available as its workers. Hence, one of the workers will receive two tasks while the other will get only one. Only when each sub-group has 8 processors then a good load balance between the workers is achieved. However, speedup stays flat beyond 56 processors as adding more will not be useful.

Processors	3P-h1	2P-h1	3P-h2	2P-h2	1P
1	801.9	803.4	799.37	799.15	804.3
2	1066.9	1080.7	1076.25	1075.03	1092.7
3	523.6	524.8	515.67	513.35	527.3
4	519.9	524.6	515.49	512.9	386.4
5	406.9	405.9	403.84	403.89	244.5
6	304.3	305.2	302.93	302.97	242.9
7	235.3	235.7	235.05	234.67	242.2
8	236.0	234.7	235.4	234.23	241.7
9	203.1	203.6	234.45	178.05	111.8
10	177.1	177.2	234.2	154.13	111.9
16	120.1	119.4	139.98	90.74	111.8
17	119.6	91.5	108.42	67.1	111.8
25	118.9	62.3	77.91	67.1	111.8
26	118.8	62.3	77.7	67.1	111.8
32	90.5	62.0	76.74	67.2	111.8
36	90.2	61.3	47.52	67.2	111.8
37	89.8	61.3	47.51	67.2	111.8
49	69.7	60.4	47.62	67.25	111.7
50	69.7	60.4	47.59	67.17	111.7
64	39.3	34.5	47.7	67.22	111.8
65	31.3	20.9	47.54	67.22	111.8

Table 6.4: Matrices matrix multiplication execution time in seconds on the Fujitsu AP1000.

6.3.4.1 Matrix Multiplication with a List of Matrices

We have modified the matrix multiplication program by applying it to a list of matrices using the map HOF. This will add another level of complexity by having three levels HOFs nesting for experimental purposes. We show below the top level code after modifying the original program and the skeleton nesting structure is shown in Figure 6.7 for 290 processes.

Figure 6.8 and Tables 6.4 and 6.5 show the results for matrix multiplication with list of matrices execution time and speedup on the Fujitsu AP1000.

```
def MatrixMul ll = ((map (map scalar)) ll);


def listarrange n = ((smap arrange)
   ((makelist ((((gen3 7) 7) 9) 50), ((((gen3 7) 7) 9) 50))
n));


   ((map MatrixMul) (listarrange 8))
```

The outer list contains 8 pairs where each pair has two matrices. Each matrix has 7 rows and 7 columns with 50-digit ALNs. This is an extra level of complexity to experiment with three levels of map nesting. For this experiment both heuristics (h1 and h2) were used and the purpose of the experiment was

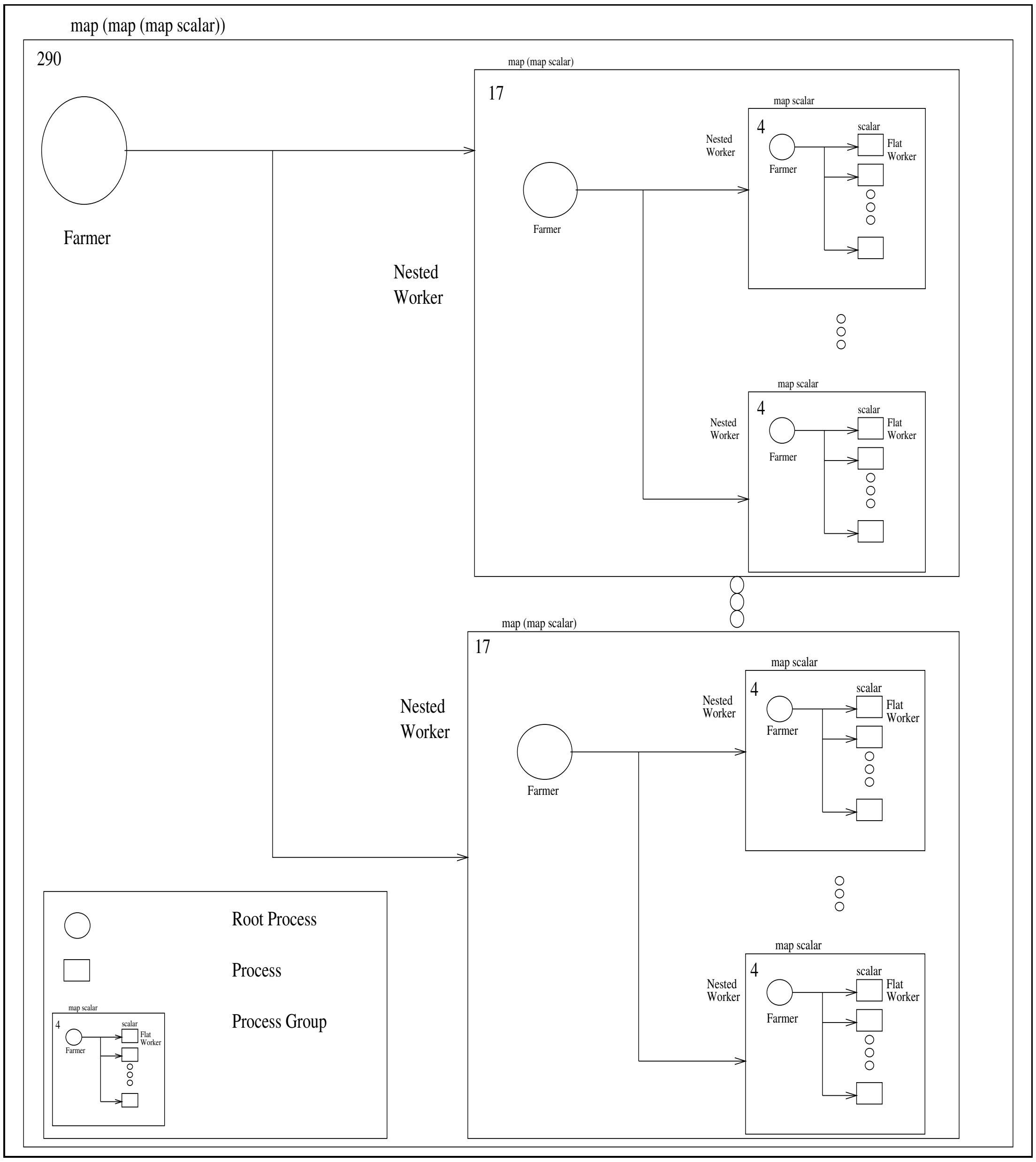

Figure 6.7: The skeleton nesting structure for the list of matrices matrix multiplication program.

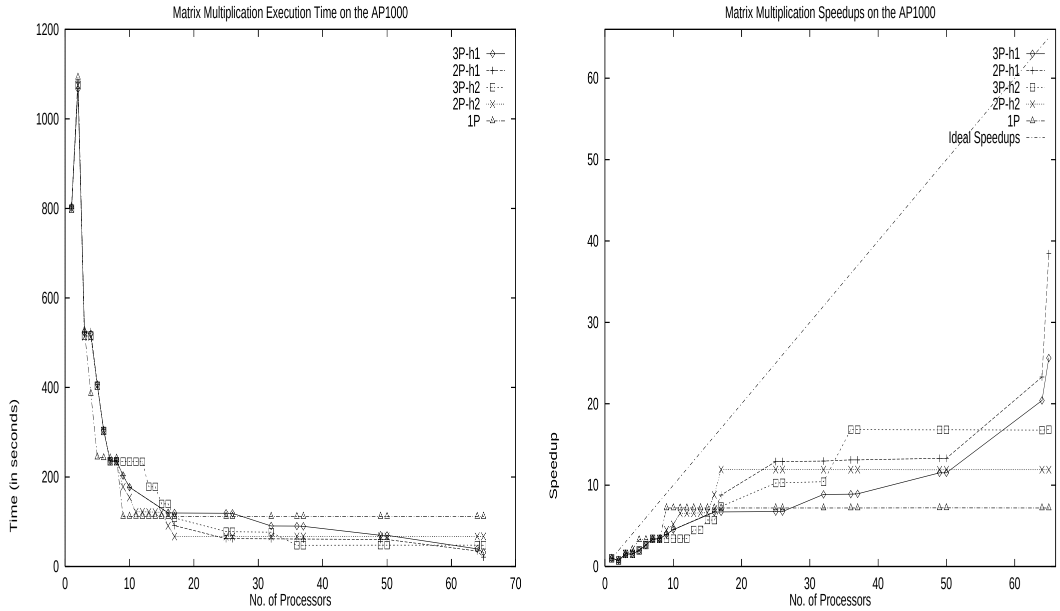

Figure 6.8: Matrices matrix multiplication execution time and speedup on the Fujitsu AP1000.

Processors	3P-h1	2P-h1	3P-h2	2P-h2	1P
1	1	1	1	1	1
2	0.75	0.74	0.74	0.74	0.74
3	1.53	1.53	1.55	1.56	1.53
4	1.54	1.53	1.55	1.56	2.08
5	1.97	1.98	1.98	1.98	3.29
6	2.64	2.63	2.64	2.64	3.3
7	3.4	3.4	3.4	3.4	3.32
8	3.4	3.42	3.4	3.4	3.32
9	3.95	3.95	3.4	4.49	7.19
10	4.53	4.53	3.4	5.18	7.19
16	6.68	6.73	5.7	8.81	7.19
17	6.7	8.78	7.4	11.91	7.19
25	6.74	12.9	10.26	11.91	7.19
26	6.75	12.9	10.29	11.91	7.19
32	8.86	12.96	10.42	11.9	7.19
36	8.89	13.1	16.82	11.89	7.19
37	8.93	13.1	16.82	11.9	7.19
49	11.5	13.3	16.79	11.88	7.2
50	11.5	13.3	16.8	11.89	7.2
64	20.4	23.29	16.76	11.89	7.19
65	25.62	38.44	16.81	11.89	7.19

Table 6.5: Matrices matrix multiplication speedup on the Fujitsu AP1000.

to see how both affect the behavior of the program. Also, the level of parallel implementation of nesting was varied, with all three maps (3P), the first two maps (2P) or the first map (1P) implementated in parallel. The quickest speedup was obtained for 1P with 7.2 on 9 processors (efficiency of 80 %). However, the speedup stayed flat beyond 9 processors. This was expected as there are only 8 sub-lists in the list data structure and the inner skeletons are not implemented in parallel. Implementing in parallel the first two maps and varying the heuristics gave different results. A speedup of 11.9 on 17 processors (with efficiency of 70 %) was achieved for "2P-h2" while it was only 8.78 for "2P-h1". The first case divided the processors into 2 sub-groups of 8 processors while the second case divided them into 4 sub-groups of 4 processors. The first case utilised the resources better and hence gave better performance. This is caused by having 3 processors working as farmers for the first case and 5 processors working as farmers for the second case. Therefore, more processes were available for the first case to be utilised as workers. However, speedup stayed flat beyond 17 processors as the inner data structure has only 7 sub-lists. However, the second case achieved a speedup of 38.44 on 65 processors with efficiency of 59 %. This was caused by having more sub-groups available to the outer process farm and better utilisation of the resources. The last case is to implement in parallel the three maps using both heuristics. For "3P-h2" a speedup of 16.8 on 36 processors (with efficiency of 47 %) was achieved compared to 8.89 for "3P-h1". However, for the first case speedup stayed constant and did not improve while the second case achieved a speedup of 25.62 on 65 processors (with efficiency of 39 %). The reason is the bad division of the processors beyond 17 for the first case where more workers will be available than the total number of tasks in the list data structure. Note that using h2 a better speedup was obtained for 3P compared to 2P. However, using h1 the opposite happened where 2P outperformed 3P.

6.3.4.2 Different HOF Combinations of List of Matrices Matrix Multiplication

Also, for the list of matrices matrix multiplication it was possible try all the different combinations of maps and SFolds. Table 6.6 shows the 8 different possibilities and their labels. The code for the different possibilities is presented in the subsections of Appendix C.2. For some of the combinations we also implemented the first three, two or one HOF(s) in parallel to give a thorough study of different HOFs and various levels of nesting. In the figures, the labels are used to denote different cases. For example the label MSM-2P can be interpreted as implement in parallel the first two HOFs. For all cases, the h1 (square root) heuristic is used for dividing the groups.

The results for the different implementations are shown in Tables 6.7 and 6.8 and Figure 6.9. The best speedup was 39.3 on 65 processors (with efficiency of 61.4 %) for SMM-2P and the next best was MMM-2P. Note that all cases that had SFold as the first HOF had very poor results on 2 processors. The reason for this could be the large overhead for transmitting messages of large size and for the large cost for flattening them as well. Remember that half of the original

Label	First HOF	Second HOF	Third HOF
MMM	map	map	map
SMM	SFold	map	map
SSM	SFold	SFold	map
SMS	SFold	map	SFold
SSS	SFold	SFold	SFold
MMS	map	map	SFold
MSM	map	SFold	map
MSS	map	SFold	SFold

Table 6.6: HOFs used for different implementations for the list of matrices matrix multiplication on the Fujitsu AP1000.

data structure is going to be flattened and transmitted for this case. In general, speedup improves without huge jumps if the first HOF is SFold instead of map as groups are more balanced. In the case of 3P the best speedup was 34 on 64 processors for SSS-3P and for 1P it was 7.2 on 8 processors for SMM-1P.

6.3.4.3 Portability of the Matrix Multiplication Program

Tables 6.9 and 6.10 and Figure 6.10 show the execution time and speedup results for an example of 3 levels of map nesting in solving matrix multiplication of 40-digit (ALNs) on the Cray T3D, Fujitsu AP1000 and a network of Linux workstations [84]. For this example we exploit parallelism for two cases 1) running all nested HOFs in parallel (denoted by 3P on the graph) and 2) executing in parallel only the first two nested HOFs (2P). Note that running only the first two nested HOFs in parallel gave better results compared to running three HOFs in parallel. The reason is that the bottom HOF lacks useful parallelism. Also, note how speedup stays the same for a range of processor numbers and then suddenly improves. This has to do with groups being balanced as more processors are added.

The results on the three machines suggest that good cross-platform portability and behavioral consistency can be achieved. However, the sequential execution on PCs was better than the other parallel machines but droped as more processors were added. This has to do with large communication overhead between the PCs because of the low bandwidth compared to the other parallel machines.

6.3.5 Merge Sort using Nested Folds

To demonstrate the case when a Fold HOF can nest itself we have implemented a parallel merge sort algorithm in Ektran. The skeleton nesting structure is shown in Figure 6.11. The algorithm is similar to the well known sequential merge sort. The basic algorithm works by dividing the original list into sub-lists of lengths as equal as possible and sorting them. The sorted sublists are merged together to

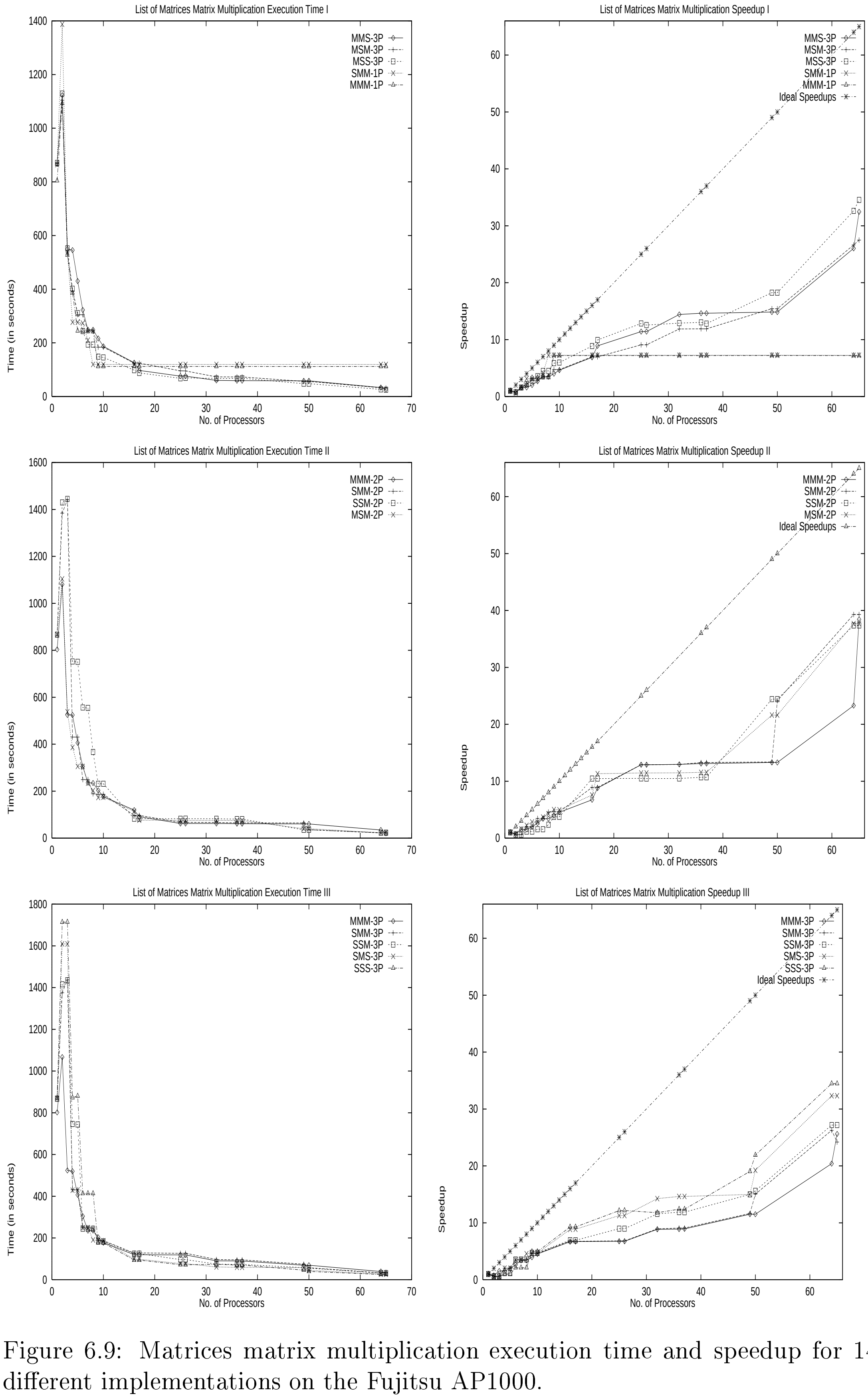

Figure 6.9: Matrices matrix multiplication execution time and speedup for 14 different implementations on the Fujitsu AP1000.

Processors	MMM-3P	SMM-3P	SSM-3P	SMS-3P	SSS-3P	MMS-3P	MSM-3P
1	801.9	866.0	867.1	869.5	871.7	866.6	868.4
2	1066.9	1375.1	1415.8	1609.5	1713.4	1123.1	1104.2
3	523.6	1436.8	1435.2	1609.1	1714	549.7	538.1
4	519.9	431.7	746.4	428.7	872.4	545.7	393.5
5	406.9	430.8	743.9	430.3	880.2	430.1	305.2
6	304.3	249.9	244.3	249.4	413.3	322.8	304.9
7	235.3	249.8	244.3	249.7	413.7	249.0	244.7
8	236.0	248.5	244.4	190.6	413.4	248.9	244.7
9	203.1	187.6	184.0	186.6	176.5	216.3	183.7
10	177.1	187.8	183.3	186.7	176.4	187.1	183.7
16	120.1	127.3	125	98.3	94.0	125.9	124.4
17	119.6	128.2	125.1	98.0	94.1	97.3	124.5
25	118.9	126.8	96.6	77.1	71.8	76.0	95.6
26	118.8	126.9	96.6	77.1	71.7	76.0	95.5
32	90.5	97.0	74.9	61.0	74.0	60.1	73.2
36	90.2	95.4	73.0	59.4	70.3	59.2	73.0
37	89.8	95.4	73.1	59.4	70.3	59.2	73.0
49	69.7	74.4	57.8	58.0	45.8	58.4	56.3
50	69.7	57.6	55.5	45.2	39.8	58.4	56.3
64	39.3	33.0	31.9	26.9	25.3	33.3	32.6
65	31.3	35.8	31.9	26.9	25.3	26.7	31.6
Processors	MSS-3P	MMM-2P	SMM-2P	SSM-2P	MSM-2P	SMM-1P	MMM-1P
1	870.7	803.4	864.3	866.9	866.9	864.5	804.3
2	1129.2	1080.7	1384.2	1430.0	1104.1	1387.4	1092.7
3	552.1	524.8	1441.6	1445.0	538.1	539.1	527.3
4	401.8	524.6	430.7	752.4	386.3	276.9	386.4
5	310.9	405.9	430.6	751.1	305.4	277.0	244.5
6	243.9	305.2	249.2	556.3	305.3	273.6	242.9
7	193.6	235.7	249.3	554.7	234.3	210.0	242.2
8	193.6	234.7	188.8	367.1	202.7	119.3	241.7
9	148.6	203.6	187.0	231.6	172.9	119.3	111.8
10	145.7	177.2	187.0	231.7	174.5	119.3	111.9
16	98.1	119.4	96.9	82.9	114.5	119.3	111.9
17	87.7	91.5	97.0	82.9	76.6	119.3	111.8
25	67.8	62.3	67.0	82.6	75.7	119.3	111.8
26	69.3	62.3	67.0	82.8	75.7	119.3	111.8
32	67.5	62.0	66.8	82.7	75.6	119.3	111.8
36	66.8	61.3	65.1	81.1	75.0	119.3	111.8
37	68.0	61.3	65.1	81.1	75.0	119.3	111.8
49	47.7	60.4	64.6	35.5	40.1	119.3	111.7
50	47.7	60.4	35.8	35.5	40.1	119.3	111.7
64	26.7	34.5	22.0	23.2	23.0	119.5	111.8
65	25.2	20.9	22.0	23.2	23.0	119.3	111.8

Table 6.7: Matrices matrix multiplication execution time for 14 different implementations on the Fujitsu AP1000.

Processors	MMM-3P	SMM-3P	SSM-3P	SMS-3P	SSS-3P	MMS-3P	MSM-3P
1	1	1	1	1	1	1	1
2	0.75	0.63	0.61	0.54	0.51	0.77	0.79
3	1.53	0.6	0.6	0.54	0.51	1.58	1.61
4	1.54	2.0	1.16	2.03	1.00	1.58	2.21
5	1.97	2.0	1.17	2.02	1.00	2.02	2.85
6	2.64	3.5	3.55	3.48	2.1	2.69	2.85
7	3.41	3.47	3.55	3.48	2.11	3.48	3.55
8	3.4	3.49	3.55	4.56	2.11	3.48	3.55
9	3.95	4.62	4.71	4.66	4.94	4.01	4.73
10	4.53	4.6	4.73	4.66	4.94	4.63	4.73
16	6.68	6.8	6.94	8.85	9.27	6.88	6.98
17	6.71	6.76	6.93	8.87	9.26	8.9	6.98
25	6.74	6.83	8.98	11.28	12.14	11.4	9.08
26	6.75	6.82	8.98	11.28	12.16	11.4	9.09
32	8.86	8.93	11.58	14.25	11.78	14.42	11.86
36	8.89	9.08	11.88	14.64	12.4	14.64	11.9
37	8.93	9.08	11.86	14.64	12.4	14.64	11.9
49	11.51	11.64	15.0	14.99	19.03	14.84	15.42
50	11.51	15.04	15.62	19.24	21.9	14.84	15.42
64	20.41	26.24	27.18	32.32	34.46	26.02	26.64
65	25.62	24.19	27.18	32.32	34.46	32.46	27.48
Processors	MSS-3P	MMM-2P	SMM-2P	SSM-2P	MSM-2P	SMM-1P	MMM-1P
1	1	1	1	1	1	1	1
2	0.77	0.74	0.62	0.61	0.79	0.62	0.74
3	1.58	1.53	0.6	0.6	1.61	1.6	1.53
4	2.17	1.53	2.01	1.15	2.24	3.12	2.08
5	2.80	1.98	2.01	1.15	2.84	3.12	3.29
6	3.57	2.63	3.47	1.56	2.84	3.16	3.31
7	4.5	3.41	3.47	1.56	3.7	4.12	3.32
8	4.5	3.42	4.58	2.36	4.28	7.25	3.33
9	5.86	3.95	4.62	3.74	5.01	7.25	7.19
10	5.98	4.53	4.62	3.74	4.97	7.25	7.19
16	8.88	6.73	8.92	10.46	7.57	7.25	7.19
17	9.93	8.78	8.91	10.46	11.32	7.25	7.19
25	12.84	12.9	12.9	10.5	11.45	7.25	7.19
26	12.56	12.9	12.9	10.47	11.45	7.25	7.19
32	12.9	12.9	12.94	10.48	11.47	7.25	7.19
36	13.03	13.11	13.28	10.69	11.56	7.25	7.19
37	12.80	13.11	13.28	10.69	11.56	7.25	7.19
49	18.25	13.3	13.38	24.42	21.62	7.25	7.2
50	18.25	13.3	24.14	24.42	21.62	7.25	7.2
64	32.61	23.29	39.29	37.37	37.7	7.23	7.19
65	34.55	38.44	39.29	37.37	37.7	7.25	7.19

Table 6.8: Matrices matrix multiplication speedup for 14 different implementations on the Fujitsu AP1000.

Processors	T3D-3P	T3D-2P	AP1000-3P	AP1000-2P	PCs-3P	PCs-2P
1	146.65	146.56	379.25	379.00	15.59	15.57
2	181.37	181.37	469.05	470.72	17.29	17.29
3	87.88	87.88	227.26	228.11	11.73	8.84
4	87.88	87.88	227.98	228.44	12.32	9.33
5	76.55	76.58	196.87	196.78	12.61	12.22
6	58.92	58.9	152.18	152.02	12.54	12.15
7	44.42	44.4	114.3	114.71	8.17	8.13
8	44.47	44.4	115.36	114.56	9.04	7.25
9	51.15	51.2	131.52	131.57	6.44	6.8
10	29.79	29.82	76.99	77.03	5.42	6.84
11	29.82	29.78	77.15	77.24	5.48	5.47
12	29.82	29.74	77.16	76.75	4.13	5.33
13	29.85	22.69	77.21	59.23	5.0	3.93
14	30.03	22.64	78.18	59.09	5.64	3.66
15	30.32	22.60	78.29	58.92	4.96	3.75
16	29.67	22.5	77.05	58.60	5.0	4.0
17	29.67	22.5	77.04	58.59	4.66	3.38
25	30.0	15.32	76.76	39.84	7.8	4.3
26	30.0	15.32	76.94	39.84	5.66	4.08
32	17.74	15.2	46.02	39.50		
36	15.72	8.39	41.06	22.50		
37	12.1	8.36	31.77	22.41		
49	9.6	8.34	25.59	22.33		
50	9.59	8.34	25.59	22.34		
64	7.57	4.9	20.48	13.74		
65	7.57	4.9	20.49	13.74		

Table 6.9: Matrices matrix multiplication execution time on the Cray T3D, the Fujitsu AP1000 and NOWs.

form the sorted list. Our algorithm works by sorting the sub-lists in parallel. In order to have two levels of nesting the original list is represented as a list of lists. The top level code of the program where the skeleton were used is shown below [1]:

```
def mergesort l = ((((Fold merge) sort) []) l);
def mergesort2 l = ((((Fold merge) mergesort) []) l);

    (mergesort2 (((((gen4 12) 12) 16) 7) 80))
```

Table 6.11 and Figure 6.12 show the execution time and speedup for the merge sort example on the AP1000 for a list of 12 sublists. Each sublist has 16 sublists and each inner sublist has 12 sublists, giving a total of 2304 40-digit ALNs. The results are poor but provide a basis for comparing the heuristics. They demonstrate better speedup for 2P-h1 than for 1P and 2P-h2. For 2P-h1 a speedup of 4.45 was achieved on 9 processors which stayed almost flat up to 16

[1] The complete code is shown in Appendix C.1.

Processors	T3D-3P	T3D-2P	AP1000-3P	AP1000-2P	PCs-3P	PCs-2P
1	1	1	1	1	1	1
2	0.8	0.8	0.8	0.81	0.90	0.90
3	1.67	1.67	1.67	1.66	1.33	1.76
4	1.66	1.66	1.66	1.66	1.27	1.67
5	1.92	1.93	1.91	1.93	1.24	1.27
6	2.49	2.49	2.49	2.49	1.24	1.28
7	3.3	3.32	3.3	3.3	1.91	1.92
8	3.3	3.29	3.3	3.3	1.73	2.15
9	2.87	2.88	2.86	2.88	2.42	2.29
10	4.92	4.93	4.92	4.92	2.88	2.28
11	4.92	4.92	4.92	4.91	2.85	2.85
12	4.92	4.92	4.93	4.94	3.78	2.92
13	4.91	4.91	6.46	6.4	3.12	3.96
14	4.88	4.85	6.47	6.41	2.76	4.25
15	4.84	4.84	6.49	6.43	3.14	4.152
16	4.94	4.92	6.51	6.47	3.12	3.89
17	4.94	4.92	6.51	6.47	3.35	4.6
25	4.89	4.94	9.57	9.51	1.99	3.62
26	4.89	4.93	9.57	9.51	2.75	3.82
32	8.27	8.24	9.64	9.6		
36	9.33	9.24	17.47	16.84		
37	12.12	11.94	17.53	16.91		
49	15.28	14.82	17.57	16.97		
50	15.29	14.82	17.57	16.97		
64	19.37	18.52	29.91	27.58		
65	19.37	18.51	29.91	27.58		

Table 6.10: Matrices matrix multiplication speedup on the Cray T3D, the Fujitsu AP1000 and NOWs.

processors. On 9 processors there were 3 sub-groups of 3 processors which caused good load balance and less communication between the nodes of the tree. On 16 processors 4 sub-groups were available where each has 4 processors. Because of the communication between the intermediate group and its child an additional overhead was caused. The poor speedup of 4.83 was related to this overhead. The other case 2P-h2 did not give a speedup of more than 3.9 on 14 processors and 1P did not give a speedup of better than 3.96 on 11 processors. Therefore, the nesting of Fold with itself and using h1 to divide the groups outperformed the other two cases.

Figure 6.13 shows the execution time and speedup for the merge sort example on the AP3000 for 2304 80-digit ALNs. The speedup results are similar to those for AP1000 experiments with very minor differences. The best speedup was 4.21 on 11 processors with efficiency of 38 % for 2P-h1.

Tables 6.13 and 6.14 and Figure 6.14 show the execution times and speedups for the merge sort example on the AP1000 for a list of 24 sublists. Each sublist has 12 sublists and the bottom sublist has 8 sublists giving a total of 2304 40-digit ALNs. Here we changed the granularity of the data structure by pushing

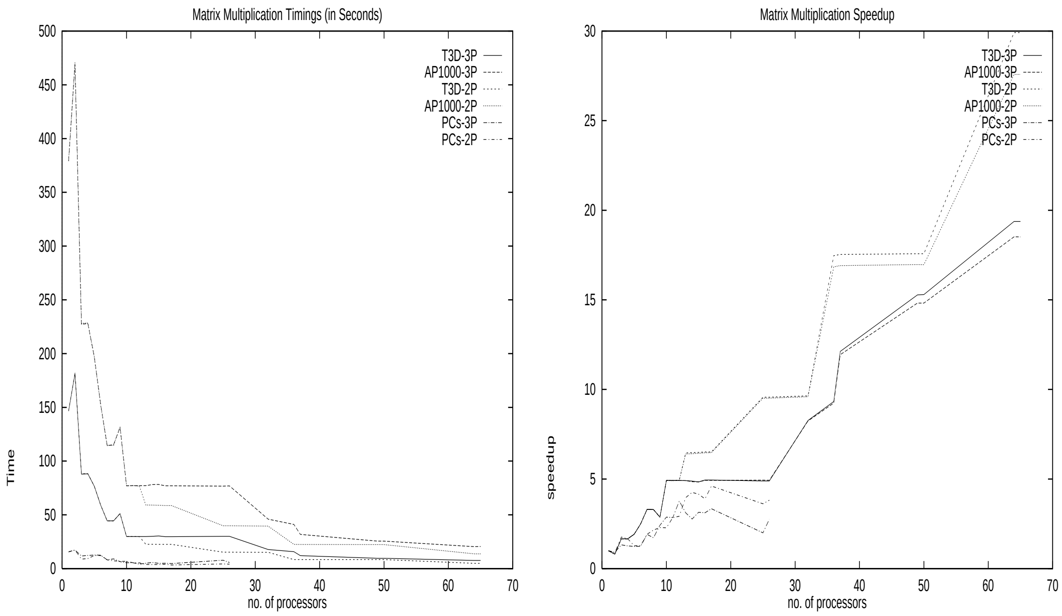

Figure 6.10: Matrices matrix multiplication execution time and speedup on the Cray T3D, the Fujitsu AP1000 and NOWs.

Processors	1	2	3	4	5	6	7	8
2P-h1	99.25	50.26	50.21	31.75	31.76	28.16	28.1	26.73
1P	99.25	50.26	37.03	32.28	33.03	27.6	27.6	29.83
2P-h2	99.25	50.2	50.21	31.75	31.75	28.15	28.16	26.72
Processors	9	10	11	12	13	14	15	16
2P-h1	22.3	22.3	21.73	21.36	21.38	20.96	20.96	20.57
1P	29.7	27.72	25.04	25.94	25.94	25.94	25.94	25.94
2P-h2	26.7	26.1	26.12	26.45	26.45	25.43	25.43	25.77

Table 6.11: Merge sort execution time on the Fujitsu AP1000.

the numbers to the outer lists rather than to the bottom ones as in the previous example. The purpose of this experiment was to make the bottom parallelism useless and see if a parallel implementation of only the outer Fold would give better results. The speedup graph shows superlinear speedup up to 3 processors. We think the cause of the superlinear speedup is due to the garbage collector which gets fired more frequently on one processor because of the size of the data structure. Also up to 8 processors the speedup for 1P is better than 2P compared to the previous results (with different granularity).

Figure 6.15 shows the execution time and speedup for the merge sort program on the Cray T3D for a list of 12 sublists. Each sublist has 12 sublists and the inner sublist has 12 sublists giving a total of 1728 50-digit ALNs. The experiment is an example of code portability but not performance portability [127]. It is believed that this is caused by the difference between the machines in terms of processor speed and bandwidth. Here code was portable but did not give good speedup results as the best speedup was 4 on 16 processors giving an efficiency of 25 %.

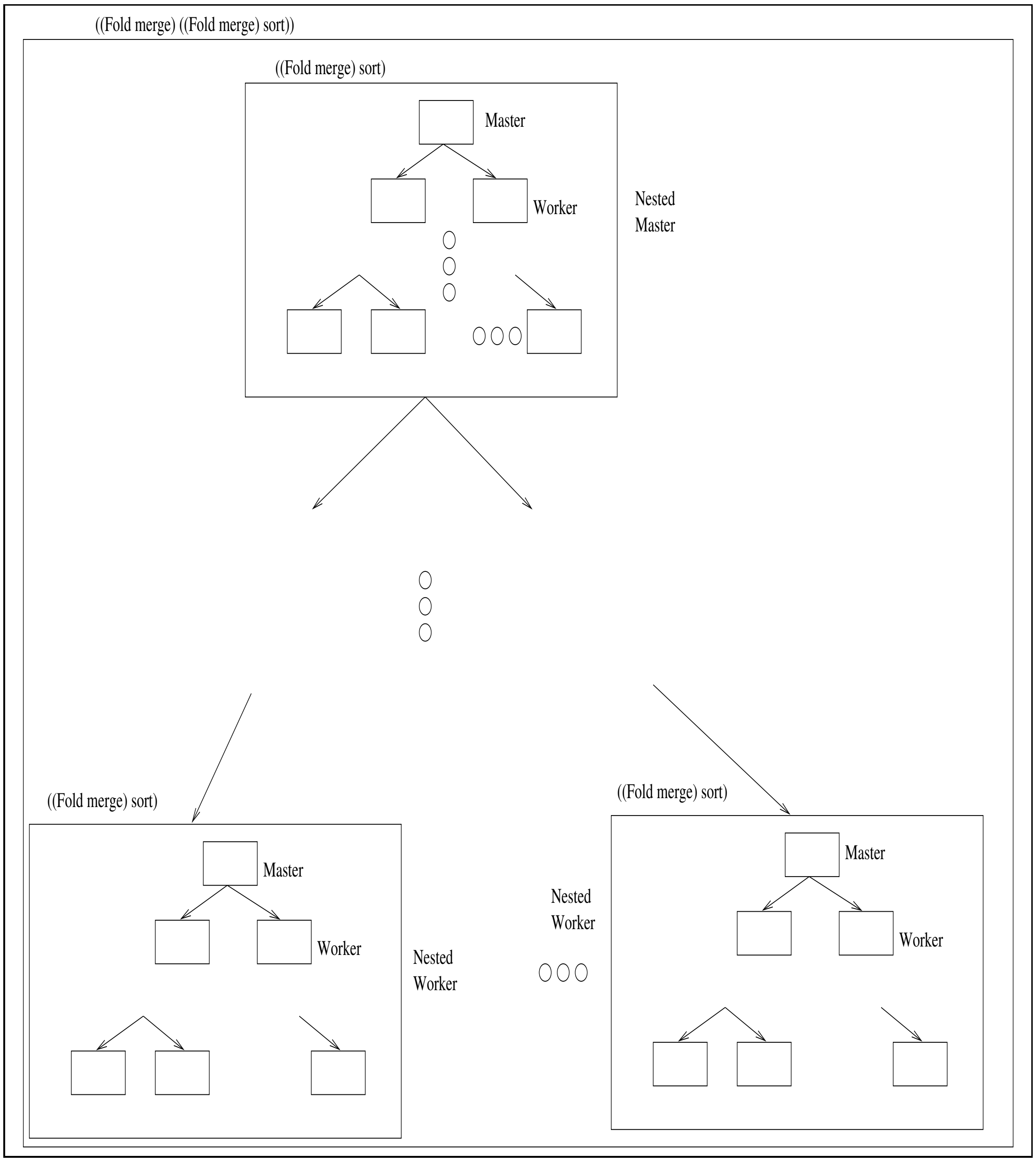

Figure 6.11: The skeleton nesting structure for the merge sort using nested Folds program.

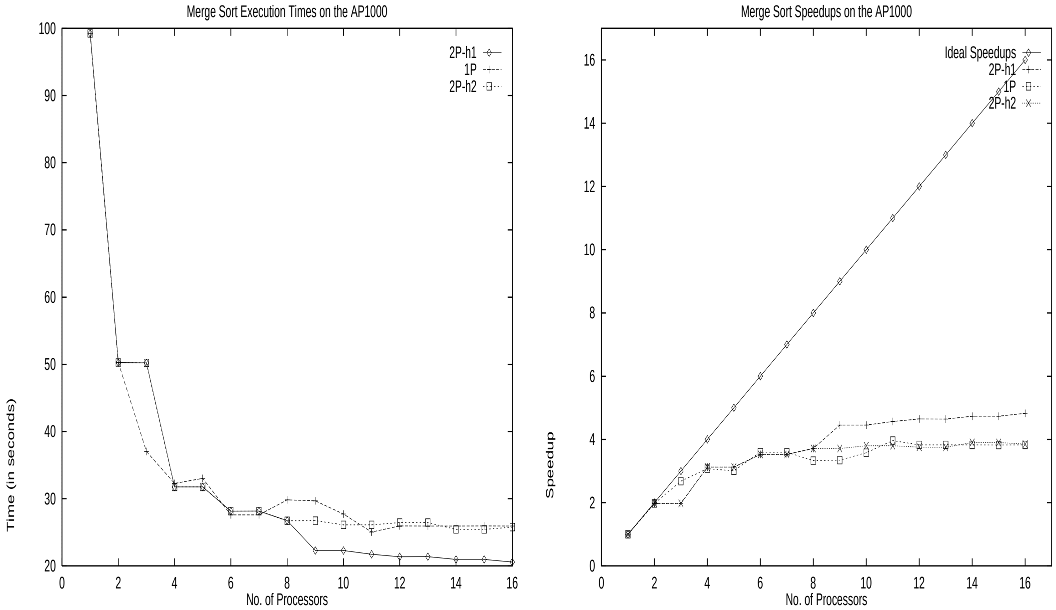

Figure 6.12: Merge sort execution time and speedup on the Fujitsu AP1000.

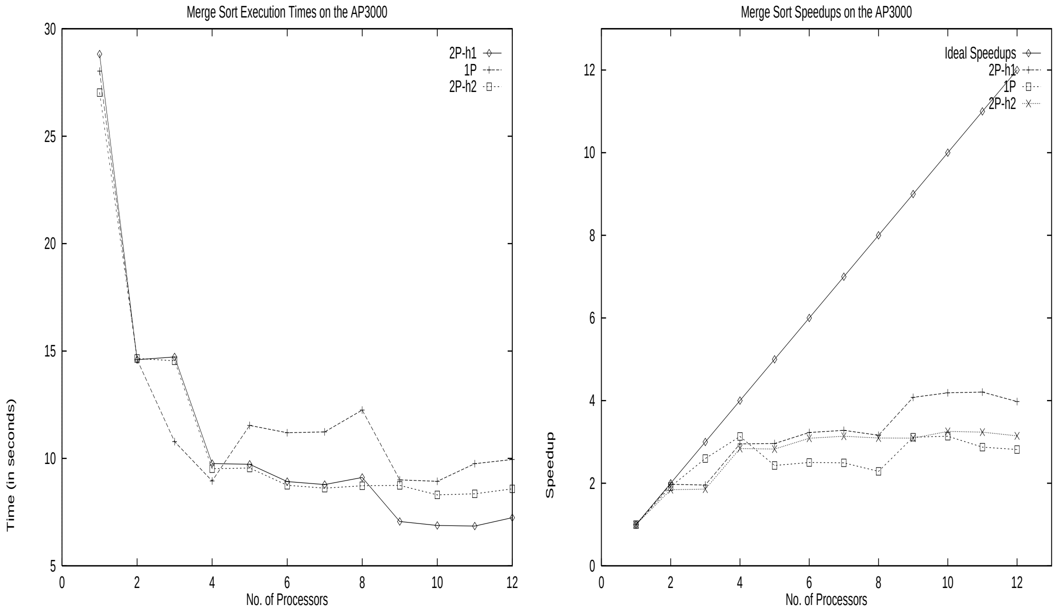

Figure 6.13: Merge Sort execution time and speedup on the Fujitsu AP3000.

Processors	1	2	3	4	5	6	7	8
2P-h1	1	1.98	1.98	3.13	3.13	3.53	3.53	3.7
1P	1	1.98	2.68	3.08	3.00	3.6	3.6	3.33
2P-h2	1	1.97	1.98	3.13	3.13	3.53	3.53	3.71
Processors	9	10	11	12	13	14	15	16
2P-h1	4.45	4.45	4.56	4.65	4.64	4.74	4.74	4.83
1P	3.34	3.58	3.96	3.83	3.83	3.83	3.83	3.83
2P-h2	3.71	3.8	3.8	3.75	3.75	3.9	3.9	3.85

Table 6.12: Merge sort speedup on the Fujitsu AP1000.

Processors	1	2	3	4	5	6	7	8	9	10
2P-h1	110.68	47.73	47.76	35.84	35.84	33.44	33.44	33.16	24.05	24.05
1P	110.69	47.73	33.67	29.05	28.57	24.96	23.96	23.83	26.83	25.97
Processors	11	12	13	14	15	16	17	25	32	36
2P-h1	23.83	23.83	23.83	24.09	24.09	21.84	21.88	22.69	22.42	20.24
1P	24.03	24.41	24.41	24.41	24.41	25.58	27.58	25.47	25.47	25.47

Table 6.13: Merge sort execution time in seconds on the Fujitsu AP1000 with different granularity.

However, the results for the nested Folds using h1 are better than for unnested Fold.

6.4 Summary

This chapter has presented the experimental framework used for evaluating the Ektran model for parallel programming. The target architectures were outlined along with their software tools (i.e. MPI library implementations) and the benchmarks used for the analysis. The results suggest that good cross-platform portability and behavioral consistency can be achieved. Also, the experiments looked at the different factors that have a big effect on the performance of combined (nested and/or composed) skeletons. These were 1) how a group is divided into sub-groups, 2) granularity of the nested data structure and 3) how deep the

Processors	1	2	3	4	5	6	7	8	9	10
2P-h1	1	2.32	2.32	3.09	3.09	3.31	3.31	3.34	4.60	4.60
1P	1	2.32	3.29	3.81	3.88	4.43	4.62	4.65	4.13	4.26
Processors	11	12	13	14	15	16	17	25	32	36
2P-h1	4.65	4.65	4.65	4.6	4.6	5.07	5.06	4.88	4.94	5.47
1P	4.61	4.53	4.53	4.53	4.53	4.33	4.01	4.34	4.35	4.35

Table 6.14: Merge sort speedup on the Fujitsu AP1000 with different granularity.

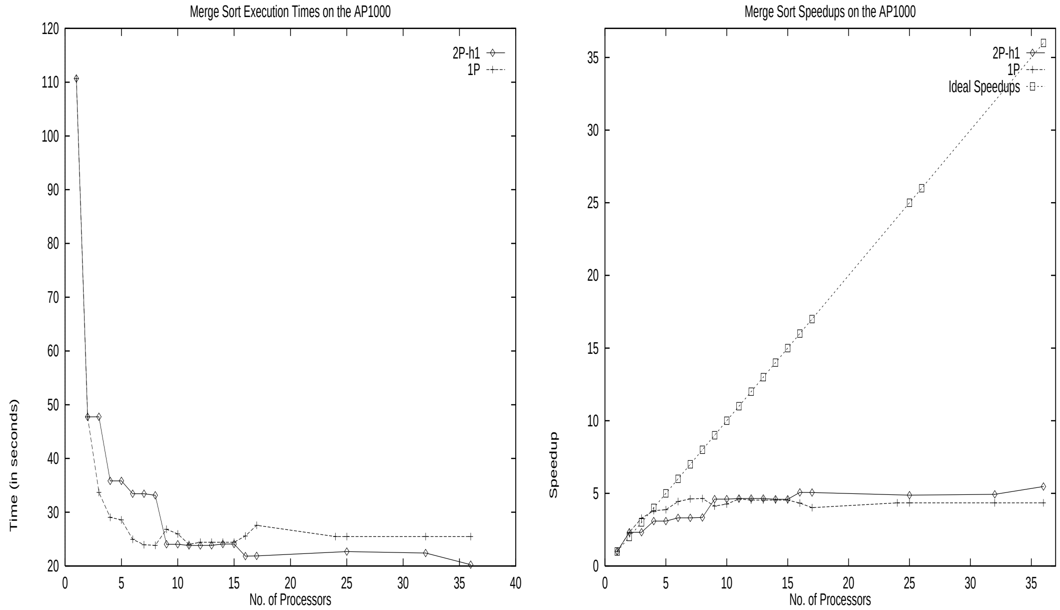

Figure 6.14: Merge Sort execution time and speedup on the AP1000 with different granularity.

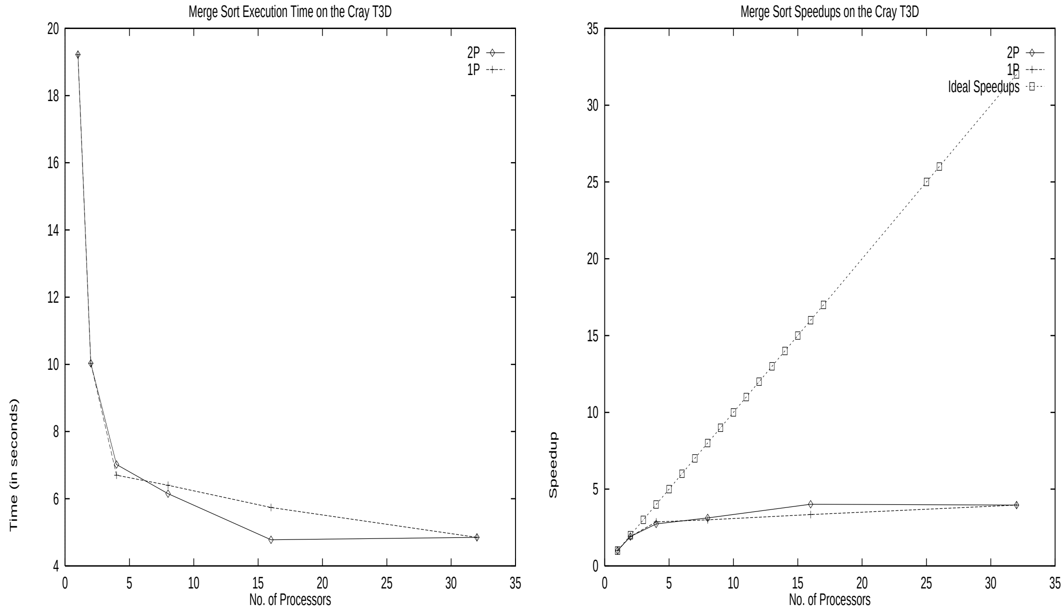

Figure 6.15: Merge sort execution time and speedup on the Cray T3D.

combined skeletons are implemented in parallel. Although the framework allows nesting up to an arbitrary level, the experiments suggest that 2 levels of nesting is better than 3 levels for the matrix multiplication. It is possible to say that arbitrary depth nesting is not needed but we think it is important to give a general solution to the problem and other scientific applications might require deep levels of nesting which are beyond the scope of this thesis to investigate.

One of the experiments showed the behavior of a program when one of its operators is replaced by another which gives a study of transformation. Hence, it might be possible to transform a HOF into one of its equivalents to improve performance over the original program.

However, the experiments looked at problems with a high degree of regularity of data sizes and processing costs. Problems can be called regular where performance does not depend on the nature of input data and have predictable communication structure. Future work should include a thorough analysis of combined skeletons for irregular problems and of the behavior of nested skeletons with non-trivial communication.

Chapter 7

Summary, Conclusions and Future Work

7.1 Introduction

This thesis has investigated a model for parallel functional programming based on combined algorithmic skeletons. Parallelism is identified through the use of HOFs with equivalent algorithmic skeletons. The thesis presented a novel framework for compiling and executing arbitrarily nested skeletons in order to exploit all the available parallelism in skeleton combinations and potentially achieve massive parallelism. The framework consisted of translation rules (schemes) used at compile-time for the static analysis of the source program and at run-time to execute the combined skeletons in parallel. The static-analysis of the source program generates a nesting structure which is used to combine the corresponding HOFs in a process termed nesting deduction. The run-time scheduling relies on the compile-time analysis and uses message-passing groups to run combined HOFs in parallel. The generated programs run in a Group SPMD mode where processes are reallocated within a group to support the nested skeletons. A number of DM MIMD machines were used for the experiments and the results suggest good cross-platform portability and behavioural consistency can be achieved.

A summary of the thesis chapters is given in Section 7.2, the contributions of the thesis are presented in Section 7.3, the limitations of the work presented in this thesis and a discussion are outlined in Section 7.4 and avenues for further work are proposed in Section 7.5.

7.2 Thesis Summary

The first chapter motivated the use of algorithmic skeletons for parallelising functional languages. A number of techniques for parallelising functional languages were outlined and the benefits of algorithmic skeletons as highly abstract with good performance models were identified.

Chapter 2 gave a comprehensive survey of skeleton-oriented parallel programming where a number of skeleton models and systems were described and compared. The chapter showed that previous systems do not fully utilise the benefits of skeletons as they do not implicitly exploit parallelism in combined skeletons. Also the importance of skeleton nesting and difficulties that arise in their parallel implementations were outlined.

Chapter 3 presented the language **Ektran** which acts as a coordination language for skeletons and is used to demonstrate the framework. The design and implementation of the parallelising compiler was also shown in this chapter.

Chapter 4 presented the compile-time analysis that is performed on Ektran programs in order to generate a nesting structure that combines all skeletons, in the form of abstract schema.

Chapter 5 presented the concept of message-passing groups which is used to run nested skeletons in parallel. Also, the parallel implementations for the operators supported in **Ektran** were shown in detail.

Chapter 6 presented the experimental environment used to evaluate the proposed methodology. It consisted of benchmark programs, target architectures and experimental parameters. Results from the benchmarks were presented and analysed.

7.3 Contributions of Thesis

The contributions of this thesis can be summarised as follows.

- A comprehensive survey was carried out which showed that one of the major benefits of skeleton-oriented parallel programming is not fully exploited, as most of the models disallow nesting of skeletons or allow it only in an explicit way.

- The major contribution of this thesis is the exploitation of parallelism available in nested skeleton hierarchies. We think this is an important contribution to fully utilise the benefits of skeleton-oriented parallel programming and to achieve massive parallelism. This was achieved by a framework which automatically compiles and executes arbitrarily nested skeletons.

- The language **Ektran** which acts as a coordination language for skeletons (by allowing nesting and composition of skeletons) was presented. It is a symbolic language and based on higher order operators that have parallel properties. Programmers naturally use these general operators to construct programs which can thus be automatically implemented in parallel.

- The parallelising compiler was engineered by integrating different components together. The integrated components were: the front-end (scanner, parser and type-checker), the middle-end (the Camlot compiler and skeleton library) and the back-end (an ANSI C compiler). This modular design greatly eased implementation.

- A compile-time framework has been presented which automatically extracts coordination information between composed and nested skeletons in the form of abstract schema. This simplifies programming using skeletons as the programmer does not need to learn about special constructs or combining forms to plug the skeletons together.

- The run-time framework based on the concept of message-passing groups has been discussed in detail. It was used to execute flat and nested skeletons in parallel. The outlined parallel algorithms encapsulate synchronisation and communication between processes. The run-time system is very general and can handle nested skeletons to an arbitrary depth.

- Special cases of the **fold** HOF: **Fold** and **SFold** have been introduced. These can be used to parallelise **fold** when its function argument is of a special form or if it is not associative. The extraction of pipeline parallelism from composed functions and a parallel implementation for a heterogeneous pipeline have been presented.

- Thorough experiments for the matrix multiplication example gave an in depth study of nested skeletons. These experiments show how different allocation of processors to skeletons affect performance and code and performance portability of the programs.

- A solution to the problem of nesting skeletons has been presented by taking advantage of useful properties of freely available products. We made use of the message-passing groups routines provided by MPI and interfacing C functions with Caml which was provided by the Camlot compiler. The Camlot compiler was able to generate closures for fully and partially applied skeletons which are essential for entering the skeletons at run-time.

7.4 Limitations and Discussion

The work in this thesis has several limitations which are now discussed.

- All flat and nested skeletons are executed in parallel whether they have useful or non useful parallelism. We rely on the programmer at this stage to use HOFs when he/she thinks that there is enough parallelism in the program. This limitation is due to the lack of integration of cost models for skeletons which can be used to automatically decide if it is worthwhile exploiting the available parallelism in flat and nested skeletons.

- Nesting of skeletons is not detected if HOFs are used within the body of a recursive function. The reason behind this limitation is the symbolic beta reduction stage which does not beta reduce recursive functions.

- The parallel implementation for the compose operator is not load balanced as it evenly divide its group of processes according to the number of stages in its pipeline. Also pipeline parallelism is only exploited if the compose operator is nested with the map HOF.

- **Ektran** is a small language and the programming constructs it has are limited. For example it lacks data types and let constructs. Also, it has a fixed repertoire of skeletons.

- The example programs shown are simple and are highly regular in terms of data sizes and processing costs.

7.5 Future Work

The research reported in this thesis opens several avenues for future research.

- An analyser might be build based on hierarchical performance (cost) models (similar to the one developed for HOPP [136]) for skeletons, sequential Ektran program profiling and performance characterisations of target architectures. Also, granularity analysis of nested data structure is important. Previous work on granularity analysis of functional languages (**GranSim**) [109] might be a good start to find out how a similar analysis can be integrated with our system.

- **Ektran** might be extended by adding: other HOFs like divide and conquer, scan, filter, map2, ...etc, data types, let construct and I/O facilities. However, the extended language might complicate the compile-time analysis and possibly introduce new problems regarding coordination between skeletons.

- The parallel implementation for the compose operator could be modified to divide its groups unevenly based on a list of weights for each of its stages. This would be useful for load balancing the pipeline and thus allocating processes to skeletons in an optimal way. Also, pipeline parallelism might be exploited when the compose operator is nested within HOFs other than the map HOF, for example the fold HOF.

- The compile-time analysis might be restructured to avoid the beta reduction stage. This stage retransforms the program and the generated program may be inefficient compared to the original program.

- Programs with deep levels of nesting and which solve more realistic problems could be investigated. It is important to see how such examples behave when ported across many different parallel target architectures especially if the skeletons deal with non-trivial communication.

- Techniques to reduce the cost of flattening and unflattenning Caml data structures might be investigated as they cause large run-time overhead. This might be achieved by generating code that will flatten/unflatten a data structure for a given type rather than using general routines for any type. These special routine would be linked to the skeleton library at run time. It would be the compiler's responsibility to generate the routine based on the type information it has.

- The nesting properties of the skeletons might be formalised. This formalism would require the employment of a mathematical notation such as operational semantics [34] or axiomatic semantics [35]. Using operational semantics, the properties of combination of skeletons can be defined in terms of their states and the possible transitions between them. In axiomatic semantics, the meaning of combinations could be defined by a set of proof rules based on predicate calculus. This would give a better understanding of nesting, especially if the language is to be extended.

- The work presented looked at skeletons within an eager implementation of function languages. It might be useful to study the proposed methodology within a lazy implementation of functional languages.

The work in this thesis has looked at an implicit approach to realising nested HOFs as nested skeletons. This was intended to exploit the available parallelism in nested HOFs to the maximum degree. Having achieved this, we add more weight to the argument that skeletons are well suited to parallelise functional program in specific and for programming parallel machines in general.

Appendix A

Type Declarations and Wrappers for Skeletons

A.1 The map HOF

We give below the type declaration for the C function implementing the **map** operator:

```
value map:
int → int ref → int ref → int → int → ('a → 'b) → 'a list → 'b list = 7 "map"
```

Note that the **map** operator is type declared, the number of arguments is specified and the name of the corresponding C function is given. The type declaration must match the C function implementing the parallel operator. Also, the prototype and the wrapper are given below:

```
extern value map(value, value, value, value, value, value, value);
extern value ProcessFarm(value, value, int, int, MPI_Comm, int, int);
extern value map(value Split, value Group, value Parallel, value Nest,
        value Free, value Function, value Data)
{
return ProcessFarm(Function, Data, Int_val(Free), Int_val(Split), (MPI_Comm)
Int_val((CPTR (Group))[0]), Int_val((CPTR (Parallel))[0]), Int_val(Nest));
}
```

where **value** is the data type recognised by the camlot compiler, **Int_val** is a conversion macro that converts from Caml Light value type to a C integer type, **MPI_Comm** is used to cast the integer type to a type recognised by MPI and **CPTR** is a macro that returns a C pointer to its argument data structure. Note that both arguments Parallel and Group are modifiable objects as they were declared to be of type **ref**. This is needed to enable the run-time system to be able to update their values.

A.2 The fold HOF

The declaration for the C function implementing the **fold** operator is given below:

```
value fold:
int → int ref → int ref → int → int → ('a → 'b → 'b)
→ 'b → 'a list → 'b = 8 "fold"
```

Also, the prototype and the wrapper are given below:

```
extern value fold(value, value, value, value, value, value, value, value);
extern value BinaryDC(value, value, value, value, MPI_Comm, int, int, int);

extern value fold(value Split, value Group, value Parallel,
        value Nest, value Free, value Function, value Base, value Data)
{
return BinaryDC(Function, Base, Data, Int_val(Free), (MPI_Comm) Int_val((CPTR
(Group))[0]), Int_val((CPTR (Parallel))[0]), Int_val(Nest), Int_val(Split));
}
```

A.3 The Fold HOF

The declaration for the C function implementing the **Fold** operator is given below:

```
value Fold:
        int → int ref → int ref → int → int → ('a → 'b → 'b) → ('c → 'a)
        → 'b → 'c list → 'b = 9 "Fold"
```

Also, the prototype and the wrapper are given below:

```
extern value Fold(value, value, value, value, value, value, value, value, value);
extern value BinaryDC(value, value, value, value, int, MPI_Comm, int, int, int);

extern value Fold(value Split, value MyComm, value Parallel, value Nest
        value Free, value Function0, value Function1, value Base, value Data)
{
return BinaryDC(Function0, Function1, Base, Data, Int_val(Free), (MPI_Comm)Int_val
        ((CPTR(MyComm))[0]),Int_val((CPTR(Parallel))[0]), Int_val(Nest), Int_val(Split));
}
```

A.4 The SFold HOF

The declaration for the C function implementing the **SFold** HOF is given below:

```
and SFold:
        int → int ref → int ref → int → int → ('b → 'b → 'b)
→ ('a → 'b → 'b) → ('c → 'a) → 'b → 'c list → 'b = 10 "SFold"
```

Also, the prototypes and the wrapper are given below:

```
extern value SFold(value, value, value, value, value, value,
        value, value, value, value);
extern value BinaryDC(value, value, value, value, value, int,
        MPI_Comm, int, int, int);

extern value SFold(value Split, value MyComm, value Parallel, value Nest, value
Free,
```

```
    value Function0, value Function1, value Function2, value Base, value Data)
{
Return BinaryDC(Function0, Function1, Function2, b, Data, Int_val(Free), (MPI_Comm)
Int_val((CPTR(MyComm))[0]), Int_val((CPTR(Parallel))[0]), Int_val(Nest), Int_val(Split));
}
```

A.5 The compose HOF

The declaration for the C function implementing the compose operator is given below:

```
value MapPipeline:
    int → int ref → int ref → int
        → int → gen list → 'a list → 'b list = 7 "MapPipeline"
```

Also, the prototypes and the wrapper are given below:

```
extern value MapPipeline(value ,value ,value, value, value, value, value);
extern value MapPipe(value, value, int, int, MPI_Comm, int, int);


extern value MapPipeline(value Split, value MyComm, value Parallel,
            value Nest, value Free, value FS, value Data)
{
return MapPipe(FS, Data, Int_val(Free), Int_val(Split), (MPI_Comm)Int_val
            ((CPTR (MyComm))[0]), Int_val((CPTR (Parallel))[0]), Int_val(Nest));
}
```

Appendix B

Caml Light Code

Below is the Caml Light code for the merge sort using a pipeline program that has been generated by the translator.

```
#open "bigsort2_1";;

#open "interface";;

let mworld = mpi_init(sys__command_line);;
let timestart = mpi_wtime();;
let world = ref mworld;;
let rank = mpi_comm_rank(world);;
let size = mpi_comm_size(world);;
let Systemhead = function x::_ -> x| _ -> raise (Failure"Systemhead");;
let Systemtail = function _::l ->l| _ -> raise (Failure"Systemtail");;
let Split = if size > 4 then int_of_float (sqrt (float_of_int (size))) else 2;;
let Parallel = ref 1;;
let Nest = 1;;
let NoNest = 0;;
let Free = 1;;
let NoFree = 0;;
let rec change y = world := y;;
let rec change2 y = Parallel := y;;
let rec mycomp99 f g x = f (g x) ;;

let rec mod1 x y  = (x  -  (y  *  (x /  y)))
;;

let rec gen1 n m  = if (n  =  m)
                    then []
                    else (mod1 n 10::gen1 (n  -  1) m)
;;

let rec gen2 i n s  = if (i  <  1)
                      then []
                      else (gen1 (n  +  s) n::gen2 (i  -  1) (n  +  s) s)
;;

let rec gen3 j i n s  = if (j  <  1)
                        then []
```

```
                        else (gen2 i n s::gen3 (j  -  1) i n s)
;;

let rec gen4 k j i n s  = if (k  <  1)
                          then []
                          else (gen3 j i n s::gen4 (k  -  1) j i (n  +  (i  *  s)) s)
;;

let rec length l  = if (l  =  [])
                    then 0
                    else (1  +  length (Systemtail l))
;;

let rec less1 n1 n2  = if ((n1  =  [])  &  (n2  =  []))
                       then false
                       else (((Systemhead n1)  <  (Systemhead n2))  or
                            (((Systemhead n1)  =  (Systemhead n2))  &
                            less1 (Systemtail n1) (Systemtail n2)))
;;

let rec less n1 n2  = if (length n1  <  length n2)
                      then true
                      else if (length n1  >  length n2)
                           then false
                           else less1 n1 n2
;;

let rec merge l1 l2  = if (l2  =  [])
                       then l1
                       else if (l1  =  [])
                            then l2
                            else if less (Systemhead l1) (Systemhead l2)
                                 then ((Systemhead l1)::merge (Systemtail l1) l2)
                                 else ((Systemhead l2)::merge l1 (Systemtail l2))
;;

let rec insert v l1  = if (l1  =  [])
                       then [v]
                       else if less v (Systemhead l1)
                            then (v::l1)
                            else ((Systemhead l1)::insert v (Systemtail l1))
;;

let rec rfold f b l  = if (l  =  [])
                       then b
                       else f (Systemhead l) (rfold f b (Systemtail l))
;;

let rec sort l1  = rfold insert [] l1
;;

let rec merge1 (l2:int list list list)  =
                   myfold Split world Parallel NoNest NoFree merge [] l2
;;
```

```
let rec sortmap (l3:int list list list)  =
                   mymap Split world Parallel NoNest NoFree sort l3
;;

let rec sortll ll  = mycomp99  merge1
                   (mymap Split world Parallel NoNest NoFree
                   (mycomp99  merge1 sortmap)) ll ;;

let res =  (mycomp99  ((function (l2:int list list list) ->
            fold Split world Parallel NoNest NoFree merge [] l2))
            (( mappipeline Split world Parallel Nest NoFree
            [(pipe11) (function (l3:int list list list) ->
            map Split world Parallel NoNest NoFree (function l1 ->
            rfold insert [] l1) l3); (pipe12) (function (l2:int list list list) ->
            fold Split world Parallel NoNest NoFree merge [] l2)] )))
            (gen4 10 10 10 9 40)
;;

if rank = 0
then let ok =  res in ()
else ();;
let timefinish = mpi_wtime();;
let totaltime = timefinish -. timestart;;
mpi_finalize();;
if rank = 0
then let ok = print_string "Total time = "; print_float totaltime in ()
else ();;
```

Appendix C

Ektran Code for the Example Programs

C.1 Merge Sort II

The code for the merge sort program is shown below. In this example the **Fold** HOF nests itself.

```
def mod1 x y = - x (* y (% x y));

def gen1 n m =
    if == n m
    then []
    else ++ ((mod1 n) 10) ((gen1  (- n 1)) m);

def gen2 i n s =
    if < i 1
    then [] else ++ ((gen1 (+ n s)) n) (((gen2 (- i 1))
                                       (+ n s)) s);

def gen3 j i n s =
    if < j 1
    then []
    else ++ (((gen2 i) n) s) ((((gen3 (- j 1)) i)  n ) s);

def gen4 k j i n s =
    if  < k 1
    then []
    else ++ ((((gen3 j) i) n ) s)
            (((((gen4 (- k 1)) j) i) (+ n (* i s))) s);

def length l =
    if (== l [])
    then 0
    else + 1 (length (tl l));

def less1 n1 n2 =
    if && (== n1 []) (== n2 [])
    then false
    else || (< (hd n1) (hd n2)) (&& (== (hd n1) (hd n2))
```

```
            ((less1 (tl n1)) (tl n2)));

def less n1 n2 =
    if < (length n1) (length n2)
    then true
    else
       if > (length n1) (length n2)
       then false
       else ((less1 n1) n2);

def merge l1 l2 =
    if == l2 []
    then l1
    else if == l1 []
         then l2
         else if ((less (hd l1)) (hd l2))
              then ++ (hd l1) ((merge (tl l1)) l2)
              else ++ (hd l2) ((merge l1) (tl l2));

def insert v l1 =
    if == l1 []
    then [v]
    else if ((less v) (hd l1))
         then ++ v l1
         else ++ (hd l1) ((insert v) (tl l1));

def sort l =
    if == l []
    then []
    else ((insert (hd l)) (sort (tl l)));

def mergesort l = ((((Fold merge) sort) []) l);

def mergesort2 l = ((((Fold merge) mergesort) []) l);

(mergesort2 (((((gen4 12) 12) 16) 7) 80))
```

C.2 Matrix Multiplication

```
def head l = hd l;

def tail l = tl l;

def cons l1 l2 = ++ l1 l2;

def append l1 l2 =
    if == l1 []
    then l2
    else ++ (hd l1) ((append (tl l1)) l2);

def mod1 x y = - x (* y (% x y));

def makelist f x =
    if == x 0
```

```
    then []
      else ++ f ((makelist f) (- x 1));

def gen1 n m =
    if == n m
    then []
    else ++ ((mod1 n) 10) ((gen1  (- n 1)) m);

def gen2 i n s =
    if < i 1
    then []
    else ++ ((gen1 (+ n s)) n) (((gen2 (- i 1))
            (+ n s)) s);

def gen3 j i n s =
    if < j 1
    then []
    else ++ (((gen2 i) n) s)
            ((((gen3 (- j 1)) i) (+ n (* i s))) s);

def addc c l =
    if == c 0
    then l
    else if == l []
         then [c]
         else ++ ((mod1 (+ c (hd l))) 10)
                   ((addc (% (- (+ (hd l) c)
                   ((mod1 (+ (hd l) c)) 10)) 10)) (tl l));

def addg c n1 n2 =
    if == n1 []
    then ((addc c) n2)
    else if == n2 []
         then ((addc c) n1)
         else ++ ((mod1 (+ c (+ (hd n1)
                 (hd n2)))) 10) (((addg (% (- (+ c (+ (hd n1) (hd n2)))
                 ((mod1 (+ c (+ (hd n1) (hd n2)))) 10)) 10)) (tl n1)) (tl n2));

def addfast n1 n2 = (((addg 0) n1) n2);

def multd d n =
    if == n []
    then []
    else ++ ((mod1 (* d (hd n))) 10)
            ((addc (% (- (* d (hd n))
            ((mod1 (* d (hd n))) 10)) 10)) ((multd d) (tl n)));

def mult n1 n2 =
    if == n1 []
    then []
    else if == (tl n1) []
         then ((multd (hd n1)) n2)
         else (((addg 0) ((multd (hd n1)) n2))
              (++ (0) ((mult (tl n1)) n2)));
```

```
def smap f l =
    if == l []
    then []
    else ++ (f (hd l)) ((smap f) (tl l));

def transpose l =
    if == l []
    then []
    else if == (hd l) []
         then []
         else ++ ((smap head) l) (transpose ((smap tail) l));

def scalar {x,y} =
    if (&& (== x []) (== y []))
    then []
    else ((addfast ((mult (hd x)) (hd y))) (scalar {tl x, tl y}));

def combine1 x l =
    if == l []
    then []
    else ++ {x, (hd l)} ((combine1 x) (tl l));

def combine l1 l2 =
    if == l1 []
    then []
    else ++ ((combine1 (hd l1)) l2) ((combine (tl l1)) l2);

def arrange {m1,m2} = ((combine m1) (transpose m2));

def MatrixMul m1 m2 = ((map (map scalar)) ((combine m1) (transpose m2)));

((MatrixMul ((((gen3 7) 7) 9) 150)) ((((gen3 7) 7) 9) 150))
```

The example program above show the first case when the **map** HOF nests itself. The other cases for matrix multiplication were acheived by replacing the **map** with the **SFold** HOF. The remaining three cases are shown below where we show only the new definition for the MatrixMul function.

The second case is for the **map** HOF to nest the **SFold** HOF.

```
def MatrixMul m1 m2 = ((map ((((SFold append) cons) scalar) []))
                      ((combine m1) (transpose m2)));
```

The third case is for the **SFold** HOF to nest itself.

```
def MatrixMul m1 m2 = (((((SFold append) cons)
                      ((((SFold append) cons) scalar) [])) [])
                      ((combine m1) (transpose m2)));
```

The fourth case is for the **SFold** HOF to nest the **map** HOF.

```
def MatrixMul m1 m2 = (((((SFold append) cons) (map  scalar)) [])
                       ((combine m1) (transpose m2)));
```

We managed to apply the matix multiplication algorithm on a list of matrices. The subsections below show the various cases by replacing the last two lines in the program outlined above with three lines in the code fragments.

C.2.1 Case MMM

```
def MatrixMul ll = ((map (map scalar)) ll);


def listarrange n = ((smap arrange)
                    ((makelist {((((gen3 7) 7) 9) 50),
                    ((((gen3 7) 7) 9) 50)}) n));

((map MatrixMul) (listarrange 8))
```

C.2.2 Case SMM

```
def MatrixMul ll = ((map (map scalar)) ll);

def listarrange n = ((smap arrange) ((makelist {((((gen3 7) 7) 9) 50),
                                    ((((gen3 7) 7) 9) 50)}) n));

(((((SFold append) cons) MatrixMul) []) (listarrange 8))
```

C.2.3 Case SSM

```
def MatrixMul ll = (((((SFold append) cons) (map scalar)) []) ll);

def listarrange n = ((smap arrange) ((makelist {((((gen3 7) 7) 9) 50),
                                    ((((gen3 7) 7) 9) 50)}) n));

(((((SFold append) cons) MatrixMul) []) (listarrange 8))
```

C.2.4 Case SMS

```
def MatrixMul ll = ((map ((((SFold append) cons) scalar) [])) ll);

def listarrange n = ((smap arrange) ((makelist {((((gen3 7) 7) 9) 50),
                                     ((((gen3 7) 7) 9) 50)}) n));

(((((SFold append) cons) MatrixMul) []) (listarrange 8))
```

C.2.5 Case SSS

```
def MatrixMul ll =
    (((((SFold append) cons) ((((SFold append) cons) scalar) [])) []) ll);
```

```
def listarrange n = ((smap arrange) ((makelist {((((gen3 7) 7) 9) 50),
                                   ((((gen3 7) 7) 9) 50)}) n));

(((((SFold append) cons) MatrixMul) []) (listarrange 8))
```

C.2.6 Case MMS

```
def MatrixMul ll = ((map ((((SFold append) cons) scalar) [])) ll);

def listarrange n = ((smap arrange) ((makelist {((((gen3 7) 7) 9) 50),
                                   ((((gen3 7) 7) 9) 50)}) n));

((map MatrixMul) (listarrange 8))
```

C.2.7 Case MSM

```
def MatrixMul ll = (((((SFold append) cons) (map scalar)) []) ll);

def listarrange n = ((smap arrange) ((makelist {((((gen3 7) 7) 9) 50),
                                   ((((gen3 7) 7) 9) 50)}) n));

((map MatrixMul) (listarrange 8))
```

C.2.8 Case MSS

```
def MatrixMul ll = (((((SFold append) cons)
                   ((((SFold append) cons) scalar) [])) []) ll);

def listarrange n = ((smap arrange) ((makelist {((((gen3 7) 7) 9) 50),
                                   ((((gen3 7) 7) 9) 50)}) n));

((map MatrixMul) (listarrange 8))
```

Bibliography

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullmann. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[2] P. Alpatov, G. Baker, H. Edwards, J. Gunnels, G. Morrow, and J. Overfelt. PLAPACK: Parallel linear algebra libraries design overview. In *Proceedings of the 1997 ACM/IEEE SC '97 Conference on High Performance Networking and Computing.* ACM Press and IEEE Computer Society Press, 1997.

[3] A. W. Appel and D. B. MacQueen. A standard ML compiler. In *Proceedings of Functional Programming Language and Computer Architecture*, volume 274 of *Lecture Notes on Computer Science*, pages 301–324. Springer-Verlag, 1987.

[4] F. Arbab, P. Ciancarini, and C. Hankin. Coordination languages for parallel programming. *Parallel Computing*, 24(27):989–1004, 1998.

[5] Arvind and R. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. In *Proceedings of the Conference on Parallel Architectures and Languages Europe*, volume LNCS 259, pages 1–29, Eindhoven, The Netherlands, 1987. Springer-Verlag.

[6] T. Axford and M. Joy. List processing primitives for parallel computation. *Computer Languages*, 19(1):1–17, 1993.

[7] B. Bacci, M. Danelutto, S. Orlando, S.Pelagatti, and M. Vanneschi. P(3)L - A structured high-level parallel language and its structured support. *Concurrency: Practice and Experience*, 7(3):613–641, 1995.

[8] B. Bacci, M. Danelutto, S. Pelagatti, and M. Vanneschi. SkIE: an heterogeneous HPC environment. Submitted for publication, 1999.

[9] J. Backus. Can programming be liberated from the von Neuman style? A functional style and its algebra of programs. *Communications of ACM*, 21(8):613–641, August 1978.

[10] P. Bailey and M. Newey. Implementing ML on distributed memory multiprocessors. *SIGPLAN*, 28(1):56–59, January 1993.

[11] Henri E. Bal. *Programming Distributed Systems.* Silicon Press, Summit, NJ, 1990.

[12] S. Benkner. Vienna Fortran 90 and its compilation. Technical Report CR-94-8, Institute for Software Technology and Parallel Systems, University of Vienna, 1994.

[13] R. Bird. A calculus of functions for program derivation. Technical report, Oxford University Computing Laboratory, Programming Research Group.

[14] G. Blelloch and S. Chatterjee. VCODE: A data-parallel intermediate language. In *Proceedings of the 3rd Symposium Frontiers of Massively Parallel Computation*, pages 471–480. IEEE, October 1990.

[15] G. Blelloch, J. Hardwick, J. Sipelstein, M. Zagha, and S. Chaterjee. Implementation of portable nested data-parallel language. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 102–111, San Deigo, CA, USA, 1993.

[16] G. Botorog. *High-Level Parallel Programming and the Efficient Implementation of Numerical Algorithms*. PhD thesis, Aachen University of Technology, Aachen, Germany, January 1998.

[17] G. Botorog and H. Kuchen. Skil: An imperative language with algorithmic skeletons for efficient distributed programming. In *Proccedings of the Fifth International Symposium on High Performance Distributed Computing (HPDC-5)*, pages 243–252, 1996.

[18] T. Bratvold. Determining useful parallelism in higher order functions. In *Proceedings of the 4th International Workshop on the Parallel Implementation of Functional Languages, Aachen, Germany*, pages 213–226, September 1992.

[19] T. Bratvold. A skeleton-based parallelising compiler for ML. In R. Plasmeijer and M. van Eekelen, editors, *Proceedings of the fifth International Workshop on Implementation of Functional Languages, Nijmegen, the Netherlands*, pages 23–33, September 1993.

[20] T. Bratvold. *Skeleton-based Parallelisation of Functional Programmes.* PhD thesis, Department of Computing and Electrical Engineering, Heriot-Watt University, 1994.

[21] S. Breitinger, R. Loogen, Y. Ortega-Mallen, and R. Pena. The Eden coordination model for distributed memory systems. In *High-Level Parallel Programming Models and Supportive Environments (HIPS)*, volume 1123 of *Lecture Notes in Computer Science*. IEEE Press, 1997.

[22] E. Brewer. *Portable High-Performance Supercomputing: High-Level Plateform-Dependent Optimization*. PhD thesis, Department of Electrical Engineering and Computer Science, MIT, 1994.

[23] A. Brull and H. Kuchen. TPascal – A language for task parallel programming. In *Proceedings of Euro-Par '96*, LNCS 1123, pages 646–654. Springer-Verlag, 1996.

[24] H. Burkhart, R. Frank, and G. Hachler. ALWAN: A skeleton programming language. In *Lecture Notes in Computer Science*, volume 1061, pages 407–415, 1996.

[25] A. Burn. *Programming in Occam 2.* Addison-Wesley, 1988.

[26] G. Burn. The evaluation tansformer model of reduction and its correctness. In *TAPSOFT '91*, pages 458–482, New York, NY, 1991. Springer-Verlag.

[27] G. Burns, R. Daoud, and J. Vaigel. LAM: An open cluster environment for MPI. In *Supercomputing Symposium*, Toronto, Canada, June 1994.

[28] D. Busvine. Implementing recursive functions as processor farms. *Parallel Computing*, 19:1141–1153, 1993.

[29] D. J. Busvine. *Detecting Parallel Structures in Functional Programs.* PhD thesis, Department of Computing and Electrical Engineering, Heriot-Watt University, April 1993.

[30] D. K. G. Campbell. *CLUMPS: A Candidate Model of Efficient, General Purpose Parallel Computation.* PhD thesis, University of Exeter, October 1994.

[31] N. Carriero and D. Gelertner. Linda in context. *Communications of ACM*, 32(4):444–457, April 1989.

[32] M. Chakravarty, F. Schroer, and M. Simons. V–nested parallelism in C. In *Proceedings of PMMP '95*, pages 167–174, Berlin, Germany, October 1995.

[33] K. Chandy, R. Manohar, B. Massingill, and D. Meiron. Integrating Task and Data Parallelism with the Group Communication Archtype. In *Proceedings of the 9th IEEE Symposium on Parallel Processing*, pages 724–733, Santa Barbara, CA, USA, April 1995.

[34] K. M. Chandy. Properties of concurrent programs. Technical report, California Institute of Technology, June 1993.

[35] K. M. Chandy and S. Taylor. *An Introduction to Parallel Programming.* Jones and Barlett, 1992.

[36] K. Mani Chandy, R. Manohar, B. Massingill, and D. Meiron. Integrating Task and Data Parallelism with the Group Communication Archtype. In *Proceedings of the 9th IEEE Symposium on Parallel Processing*, pages 724–733, Santa Barbara, CA, USA, April 1995.

[37] A. Chtchelkanova, J. Gunnels, G. Morrow, J. Overfelt, and R. A. van de Geijn. Parallel implementation of BLAS: General techniques for level 3 BLAS. Technical Report TR-95-40, Department of Computer Sciences, University of Texas, October 1996.

[38] A. Church. *The Calculi of Lambda Calculus.* 1941.

[39] P. D. Coddington. The application of transputer arrays to scientific problems. In John Hulskamp, editor, *Australian Transputer and OCCAM user Group Conference Proceedings*, pages 5–10, Jun 1988.

[40] M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* Pitman/MIT, London, 1989.

[41] G. Cousineau and M. Mauny. *The Functional Approach to Programming.* Cambridge University Press, Cambridge, UK, 1998.

[42] Cray T3D system architecture. Technical Report HR-04033, Cray Research Inc, September 1993.

[43] M. Danelutto and S. Pelagatti. Parallel Implementation of FP using a Template-based Approach. In *Proceedings of the 5th Internationl Workshop on Implementation of Functional Languages*, pages 7–21, September 1993.

[44] J. Darligton and M. Reeve. ALICE - A Multi-processor Reduction Machine for the Parallel evaluation of applicative languages. In *Proceedings of the ACM Conference on Functional Programming Language Computer Architecture*, pages 65–75, 1981.

[45] J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, and Q. Wu. Parallel Programming using Skeleton Functions. In M Reeve A Bode and G Wolf, editors, *PARLE '93 Parallel Architectures and Languages Europe, Munich, Germany*, pages 146–160, Leeds, 1993. Springer-Verlag, LNCS 694.

[46] J. Darlington, M. Ghanem, and H. W. To. Structured Parallel Programming. In *Proceedings of Massively Parallel Programming Models Conference*, pages 160–169, Berlin, September 1993. IEEE Computer Society Press.

[47] J. Darlington, Y. Guo, H. W. To, and J. Yang. Functional Skeletons for Parallel Coordination. In *Proceedings of the 1st EuroPar Conference*, pages 55–66, Stockholm, Sweden, August 1995.

[48] J. Darlington and H. W. To. Building Parallel Applications without Programming. In *Second Workshop on Abstract Machine Models for Highly Parallel Computers*, pages 140–154, Leeds, 1993.

[49] J. Darlington, H. W. To, and Y. Jing. Skeletons for Structured Parallel Composition. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '95)*, pages 19–28, Santa Barbara, CA, USA, July 1995.

[50] J. Davie, H. Deldari, and P. Dew. Constructive Sold Geometry Using Algorithmic Skeletons. In *Proceedings of the 5th Eurographics Workshop on Programming Paradigms for Graphics*, Also available as Technical Report No. 96.01, Division of Computer Science, University of Leeds, 1995.

[51] A. Davis and R. Keller. Data Flow Program Graphs. *IEEE Computer*, pages 26–41, February 1982.

[52] H. Deldarie. *A Performance Study for Parallel Algorithmic Skeletons.* PhD thesis, University of Leeds, Division of Computer Science, 1996.

[53] H. Deldarie, J. R. Davy, and P. M. Dew. The Performance of Parallel Algorithmic Skeletons. In *Proceedings of ZEUS '95*, pages 65–74, Also available as Technical Report 95.6, Division of Computer Science, University of Leeds, May 1995. IOS press.

[54] D. Engelhardt and A. Wendelborn. A Partitioning-Independent Paradigm for Nested Data Parallelism. *International Journal of Parallel Programming*, 24(4):291–317, August 1996.

[55] D. Engelhardt and A. Wendelborn. A Multi-Threaded Implementation of Nested Data-Parallelism. *Australian Computer Science Communications*, 19(1), February 1997.

[56] A. Agrwal *et al.* The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 2–13, May 1995.

[57] A. Geist *et al. PVM: parallel virtual machine: a users' guide and tutorial for networked parallel computing.* MIT, 1994.

[58] D. Lenoski *et al.* The directory-based cache coherence protocol for the dash multiprocessor. In *IEEE 17th Annual Symposium on Computer Architecture*, 1990.

[59] H. Aida *et al.* Simulation and performance estimation for the rewrite rule machine. In *The Fourth Symposium on the Frontiers of Massively Parallel Computation*, 1992.

[60] J. R. Gurd *et al.* A framework for experimental analysis of parallel computing. Technical report, Department of Computer Science, University of Manchester, M13 9PL, England, February 1993.

[61] J. S. Chase *et al.* The Amber System: Parallel Programming on a Network of Multiprocessors. In *The 12th ACM Symposium on Operating Systems Principles*, pages 147–158, December 1989.

[62] M. Daneluto *et al.* The P^3L Language: An Introduction. Technical Report HPL-PSC-91-29, Helwet-Packard Laboratories, Pisa Science Center, Pisa, Italy, December 1991.

[63] M. Snir *et al.* *MPI: The Complete Reference.* The MIT Press, 1996.

[64] D. Feldcamp, H. V. Sreekantaswamy, A. Wagner, and S. Chanson. Towards a Skeleton-based Parallel Programming Environment. In A. Veronis and Y. Paker, editors, *Transputer Research and Applications*, volume 5, pages 104–115. IOS Press, 1992.

[65] D. Feldcamp and A. Wagner. Parsec - A Software Development Environment for Performance Oriented Parallel Programming. In S. Atkins and A. S. Wagner, editors, *Transputer Research and Applications*, volume 6, pages 247–262. IOS Press, 1993.

[66] A. J. Field and P. G. Harrison. *Functional Programming.* Addison-Wesley, 1988.

[67] U. Fissgus, T. Rauber, and G. Runger. A framework for generating group-parallel programs. In *Proceedings of the Seventh Workshop on Compilers for Parallel Computers*, 1998.

[68] M. J. Flynn. Some computer organisations and their effectiveness. *IEEE Transactions on Computers*, 21(9), 1972.

[69] I. Foster and K. M. Chandy. Fortran M: A Language for Modular Parallel Programming. *Journal of Parallel and Distributed Computing*, 25(1), 1995.

[70] AP1000 Program Development Guide. Technical report, Fujitsu Laboratories Ltd, March 1994.

[71] ANU/Fujitsu CAP research program, MPI user's guide. Technical report, Department of Computer Science, Computer Sciences Laboratory, The Australian National University, 1994.

[72] D. Gelertner and N. Carriero. Coordination Languages and their Significance. *Communications of the ACM*, 32(2):97–107, 1992.

[73] M. Ghanem. *Structured Parallel Programming Using Performance Models and Skeletons.* PhD thesis, Department of Computing, Imperial College of Science, Technology and Medicine, February 1999.

[74] S. Gorlatch. From Transformations to Methodology in Parallel Program Development: A Case Study. *Microprocessing and Microprogramming*, 41:571–588, 1996.

[75] S. Gorlatch and C. Lengauer. Parallelization of Divide-and-Conquer in the Bird-Meertens Formalism. *Formal Aspects of Computing*, 7(6):663–682, 1995.

[76] S. Gorlatch and C. Lengauer. (De)Composition Rules for Parallel Scan and Reduction. In *Proceedings of the 3rd Massively Parallel Programming Models (MPPM '97)*, pages 23–32, 1997.

[77] A. Y. Grama, A. Gupta, and V. Kumar. Isoefficiency: Measuring the Scalability of Parallel Algorithms and Architectures. *IEEE Parallel and Distributed Technology*, 1(3):12–21, August 1993.

[78] J. Gurd, C. Kirkham, and I. Watson. The Manchester Prototype Dataflow Computer. *Communications of the ACM*, 28(1):34–52, January 1985.

[79] G. Hachler. *ALWAN: Design and Implementation of a Parallel Coordination Language*. PhD thesis, Department of Computer University, Basel University, 1997.

[80] G. Hains. Subset synchronization in BSP computing. In H.R.Arabnia, editor, *PDPTA'98 International Conference on Parallel and Distributed Processing Techniques and Applications*, volume I, pages 242–246, Las Vegas, July 1998. CSREA Press. Four volumes.

[81] R. Halstead. MultiLisp: A Language for Concurrent Symbolic Computation. *ACM TOPLAS*, 7(4):501–538, 1985.

[82] M. Hamdan. A Survey of Cost Models for Algorithmic Skeletons. Technical Report RM/99/11, Department of Computing and Electrical Engineering, Heriot-Watt University, UK, September 1999.

[83] M. Hamdan, G. Michaelson, and P. King. A Scheme for Nesting Algorithmic Skeletons. In T. Davie K. Hammond and C. Clack, editors, *Draft Proceedings of the 10th International Workshop on Implementation of Functional Languages, IFL '98*, pages 195–212, September 1998.

[84] M. Hamdan, G. Michaelson, and P. King. A Framework for Nesting Algorithmic Skeletons. In *Parallel Computing '99*, Delft, The Netherlands, August 1999. to be published IC press.

[85] K. Hammond. Parallel Functional Programming: An Introduction. In *Proceedings of First International Symposium on Parallel Symbolic Computation, PASCO 94*. World Scientific Publishing Company, September 1994.

[86] K. Hammond. Strategic SPMD. In *Proceedings of the 1998 Glasgow workshop on Functional Programming*, Pitlochry, Scotland, September 1998.

[87] K. Hammond and G. Michaelson, editors. *Research Directions in Parallel Functional Programming*. Springer-Verlag, November 1999.

[88] P. Hatcher and M. Quinn. *Data-Parallel Programming on MIMD Computers*. MIT Press, 1991.

[89] C. A. Herrmann, C. Lengauer, R. Gnz, J. Laitenberger, and C. Schaller. A compiler for HDC. Technical Report MIP-9907, Fakultt fr Mathematik und Informatik, Universitt Passau, May 1999.

[90] J. Hill. *Data-Parallel Lazy Functional Programming.* PhD thesis, Queen Mary and Westfield College, The University of London, September 1994.

[91] S. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

[92] S. Peyton Jones. Parallel Implmentations of Functional Programming Languages. *The Computer Journal*, 32(2):175–186, 1989.

[93] S. Peyton Jones, C. Clack, J. Salkild, and M. Hardie. GRIP - a High Performance Architecture for Parallel Graph Reduction. In *Functional Programming Languages and Computer Architecture*, pages 98–112, Berlin, Germany, September 1987. Springer-Verlag.

[94] H. Jonkers. Introduction to Probabilistic Performance Modelling of Parallel Applications. Technical Report 1-68340-44(1993)04, Delft University of Technology, Faculty of Engineering, Laboratory of Computer Architecture and Digital Techniques, April 1993.

[95] S. B. Junaido. *A Parallel Functional Language Compiler for Message-Passing Multicomputers.* PhD thesis, School of Mathematical and Computational Sciences, University of St Andrews, 1998.

[96] L. Clarke K. Cameron and A. Gordon. CRI/EPCC MPI for CRAY T3D. Technical report, Edinburgh Parallel Computing Centre.

[97] J. Keane. An Overview of the Flagship System. *Journal of Functional Programming*, 4(1):19–45, January 1994.

[98] C. Kebler. Neststep: Nested parallelism and virtual shared memory for the bsp model. In *Proceedings of the 1999 International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 613–619, Las Vegas, USA, July 1999. CSREA Press (Computer Science Research, Education, and Applications Tech.

[99] C. Kebler and H. Seidel. Integrating synchronous and asynchronous paradigms: the Fork95 parallel programming language. In *Proceedings of PMMP '95*, pages 135–141, Berlin, Germany, October 1995.

[100] C. Kebler and H. Seidel. The Fork95 Parallel Programming Language: Design, Implementation, Application. *International Journal of Parallel Programming*, 25(1):17–49, February 1997.

[101] C. Kebler and H. Seidl. Forklight: A control-synchronous parallel programming language. In *Proceedings of HPCN-Erope '99.* to be published by Springer LNCS. Also available as Technical Report No. 98-13, Fachbereich IV - Informatic, Univeritat Trier, D-54286 Trier, Germany, April 1999.

[102] G. Keller and M. Chakravarty. Flattening Trees. In *Proceedings of EuroPar '98*, pages 709–719, 1998.

[103] P. Kelly. *Functional Programming for Loosely-Coupled Multiprocessors*. Pitman/MIT Press, 1989.

[104] Leonidas I. Konothanassis and Michael L. Scott. Distributed shared memory for new generation networks. Technical report, "578, Department of Computer Science, University of Rochester, Rochester, NY 14627-0026", March 1995.

[105] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[106] C. E. Leirson, Z. S. Abuhamdeh, and D. C. Douglas *et al.* The Network Architecture of the Connection Machines CM-5. Technical report, Thinking Machines Corporation, July 1992.

[107] X. Leroy. The Objective Caml System, available from http://pauillac.inria.fr/ocaml/. Technical report, INRIA, 1996.

[108] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[109] H. Loidl. *Granularity in Large-Scale Parallel Functional Programming.* PhD thesis, Department of Computing Science, University of Glasgow, December 1997.

[110] R. Loogen. *Programming Language Constructs,* In [87], chapter 10, pages 61–84. 1999.

[111] Meiko Ltd. Computing Surface - CSTools Documentation Guide. Technical report.

[112] Jr. M. Wagner. Modeling Performance of Parallel Programs. Technical report, The University of Rochester, Computer Science Department, Rochester, New York 14627, June 1995.

[113] N.B. MacDonald and M.G. Norman. Issues in parallel programming environments. In *BCS Parallel Processing Specialist Group Workshop on "Abstract Machine Models for Highly Parallel Computers"*, March 1991.

[114] D. R. Mackay, J. Drake, T. Sheehan, and B. Shelton. Experiences in Programming Grand Challenge Applications on the Inter MP Paragon Supercomputer. In *Proceedings of the Inter Supercomputer Users Group*, Available on the web at http://www.cs.sandia.gov/ISUG, 1995.

[115] Wayne A. Sawdon Matthew J. Zekauskas and Brian N. Bershad. Software write detection for distributed shared memory. In *First Symposium on Operating Systems Design and Implementation (OSDI) 94*, 1994.

[116] M. Mauny. Functional Programming using Caml Light. Technical report, INRIA, 1995.

[117] G. Michaelson. *An Introduction to Functional Programming Through Lambda Calculus.* Addison-Wesley Publishing Company, 1989.

[118] G. Michaelson, A. Ireland, and P. King. Towards a Skeleton Based Parallelising Compiler for SML. In *Proceedings of the 9th International Workshop on Implementation of Functional Languages, IFL '97*, pages 539–546, University of St. Andrews, September 1997.

[119] G. Michaelson, N. Scaife, P. Bristow, and P. King. Nested Algorithmic Skeletons from Higher Order Functions. *Submitted to Parallel Algorithms and Applications, Special Issue on High Level Models and Languages for Parallel Processing*, September 2000.

[120] D. Nicol and D. O'Hallaron. Improved Algorithms for Mapping Pipelined and Parallel Computations. *IEEE Transactions on Computers*, 40(3):295–306, March 1991.

[121] P. S. Pacheco. *Parallel Programming with MPI.* Morgan Kaufmann, 1997.

[122] D. Palmer, J. Prins, and S. Westfold. Work-Efficient Nested Data-Parallelism. In *Proceedings of the 5th Symposium on the Frontiers of Massively Parallel Computation*, pages 186–193, McLean, VA, USA, February 1995.

[123] P. Panangaden and J. Reppy. The Essence of Concurrent ML. In *ML with Concurrency: Design, Analysis, Implementation and Application*, Monographs in Computer Science, pages 5–29. Springer-Verlag, 1997.

[124] P.J. Parsons and F.A. Rabhi. Generating Parallel Programs from Skeleton Based Specifications. *Journal of Systems Architecture*, 45:261–283, 1998.

[125] S. Pelagatti. *A methodology for the development and the support of massively parallel programs.* PhD thesis, Università di Pisa-Genova-Udine, March 1993.

[126] S. Pelagatti. Compiling and supporting skeletons on MPP. In *Proceedings of Massively Parallel Programming Models (MPPM) '97*, pages 140–150, London, UK, September 1997.

[127] S. Pelagatti. *Structured Development of Parallel Programs.* Taylor and Francis, London, UK, 1998.

[128] Martin Preston. Introduction to MPI. *From: A 1 Day Course, Manchester and North HPC Training & Education Centre, UK*, May 1994.

[129] A. Preto. Parallel Programming Using Composition of Templates. Master's thesis, Department of Computer Science, Univerity of Manchester, UK, 1994.

[130] A. Preto. *Composing Systems With Multiple Threads*. PhD thesis, Department of Computer Science, Univerity of Manchester, UK, 1997.

[131] F. A. Rabhi. Exploiting Prallelism in Functional Languages: a Paradigm -Oriented Approach. In *Abstract Machine Models for Highly Parallel Computers*, pages 118–139, April 1993.

[132] F. A. Rabhi and J. Schwarz. "Paradigm-Oriented" Design of Parallel Iterative Programs Using Functional Languages. In *Proceedings of the 3rd International Conference on Apllications of Supercomputers in Engineering*, pages 377–393, Bath, UK, September 1993.

[133] F. A. Rabhi and J. Schwarz. POPE: A Paradigm-Oriented parallel Programming Environment for SIT Algorithms. In *6th Workshop on the Implmentation of Functional Languages*, School of Information Systems, University of East Anglia, Norwich, UK, UK, September 1994.

[134] S. Raina. Virtual shared memory: A survey of techniques and systems. Technical report, CSTR-92-36, Department of Computer Science, University of Bristol, Bristol BS8 1TR, U.K., December 1992.

[135] S. Raina. *Emulation of a Virtual Shared Memory Architecture*. PhD thesis, Department of Computer Science, University of Bristol, Bristol BS8 1TR, U.K., September 1993.

[136] R. Rangaswami. *A Cost Analysis for a Higher-order Parallel Programming Model*. PhD thesis, University of Edinburgh, 1996.

[137] T. Rauber and G. Runger. The Compiler TwoL for the Design of Parallel Implmentations. In *Proceedings of the 4th International Conference on Parallel Architecture and Compilation Techniques*, pages 282–301, Boston, USA, 1996. IEEE Computer Society Press.

[138] T. Rauber, G. Runger, and R. Wilhelm. Deriving Optimal Data Distributiond for Group Parallel Numerical Algorithms. In *Proceedings of Conference of Massively Parallel Programming Models*, pages 33–41, Berlin, Germany, 1995.

[139] C. Reade. *Elements of Functional Programming*. Addison-Wesley, 1989.

[140] J. Riley, J. Prins, and S. Iyer. Provably Correct Vectorization of Nested-Parallel Programs. In *Proceedings of Programming Models for Massively Parallel Computers (PMMP '95)*, pages 213–222, Berlin, Germany, October 1995.

[141] P. Robinson. From ML to C via Modula-3: An approach to teaching programming. In M. Woodman, editor, *Programming Language Choice*, pages 149–169, UK, 1996. International Thomson Computer Press.

[142] P. Roe. *Parallel Programming using Functional Languages.* PhD thesis, Department of Computing Science, University of Glasgow, 1991.

[143] N. Scaife, P. Bristo, and G. Michaelson. Engineering a Parallel Compiler for Standard ML. In *Proceedings of the 10th International Workshop on Implementation of Functional Languages, IFL '98*, pages 213–225, University College London, September 1998.

[144] N. Scaife, G. Michaelson, and A. Wallace. Four Skeletons and a Function. In *Proceedings of the 9th International Workshop on Implementations of Functional Languages*, pages 529–538, St. Andrews, Scotland, September 1997.

[145] J. Schwarz and F.A. Rabhi. A Case Study in the Parallel Implementation of Functional Languages Using Skeletons. In *High Performance Computing and Networking*, volume 797 of *LNCS*, pages 357–362, Munich, April 1994. Springer-Verlag.

[146] J. Schwarz and F.A. Rabhi. A Skeleton-Based Implementation of Iterative Transformation Algorithms using Functional Languages. In *Abstract Machine Models for Parallel and Distributed Computing*, pages 119–133, April 1996.

[147] Michael L. Scott. The prospects for parallel programms on distributed systems. Technical report, Computer Science Department, University of Rochester, 1993.

[148] J. Serot. Embodying Parallel Functional Skeletons: An Experimental Implmenetation on Top of MPI. In *Proceedings of Euro-Par '97 Parallel Processing*, volume 1300, pages 629–633, Passau, Germany, August 1997. Springer.

[149] T. Sheffler and S. Chatterjee. An Object-Oriented Approach to Nested Data Parallelism. In *Proceedings of the 5th Symposium on the Frontiers of Massively Parallel Computation*, pages 203–210, McLean, VA, USA, February 1995.

[150] D. Sitsky. Implementation of MPI on the Fujitsu AP1000: Technical details release 1.1. Technical Report TR-CS-9408, Department of Computer Science, The Australian National University, September 1994.

[151] D. Skillicorn. *Foundations of Parallel Programming.* Cambridge University Press, 1994.

[152] D. B. Skillicorn and W. Cai. A cost calculus for parallel functional programming. *Journal of Parallel and Distributed Programming*, 28(1):65–84, 1994.

[153] D.B. Skillicorn and D. Talia. Models and Languages for Parallel Computation. *ACM Computing Surveys*, 30(2):125–169, June 1998.

[154] E. Smirini, A. Merlo, D. Tessera, G. Haring, and G. Kotsis. Modeling Speedup of SPMD Applications on the Intel Paragon: a Case Study. In *Proceedings of High Performance Computing and Networks (HPCN '95)*, Milan, Italy, 1995.

[155] H. Soosaipillai. An Explanation Based Polymorphic Type Checker for Standard ML. Master's thesis, Department of Computer Science, Heriot-Watt University, 1990.

[156] H. V. Sreekantaswamy. Performance Prediction Modelling of Multicomputers. Technical Report 91-27, University of British Columbia, Vancouver, Canada, November 1991.

[157] M. Sudholt. *The Transformational Derivation of Parallel Programs using Data-Distribution Algebras and Skeletons.* PhD thesis, Technische Universitat Berlin, August 1997.

[158] T. B. Tabe, J. Hardwick, and Q. F. Stout. Statistical Analysis of Communication Time on the IBM SP2. *Computing Science and Statistics*, 27, 1995.

[159] D. Tarditi, A. Acharya, and P. Lee. No Assembly Required: Compiling Standard ML to C. Technical Report 187, CMU-CS, November 1990.

[160] H. W. To. *Optimising the Parallel Behaviour of Combinations of Program Components.* PhD thesis, Department of Computing, Imperial College of Science, Technology and Medicine, London, 1995.

[161] P. W. Trinder, K. Hammond, H. Loidl, and S. Peyton Jones. Algorithm + strategy = parallelism. *Journal of Functional Programming*, 8(1):23–60, 1998.

[162] P. W. Trinder, K. Hammond, J. S. Jr. Matson, S. Peyton Jones, and J. Wu. GUM: a portable parallel implementation of Haskell. In *Proceedings PLDI '96*, pages 79–88, Philadephia, Pennsylvania, 1996.

[163] L. G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, 1990.

[164] A. Wagner, H. Sreekantaswamy, and S. Chanson. Performance Models for the Processor Farm Paradigm. *IEEE Transactions on Parallel and Distributed Systems*, 8(5):475–489, May 1997.

[165] A. M. Wallace, G. J. Michaelson, N. Scaife, and W. J. Austin. A Dual Source, Parallel Architecture for Computer Vision. *The Journal of Supercomputing*, 12:37–56, 1998.

[166] K. Waugh. An Algorithm to Balance a Pipeline of Processes. Technical Report TR92011, Department of Computing and Electrical Engineering, Heriot-Watt University, Riccarton, Edinburgh EH14 4AS, UK, 1992.

[167] E. West and A. Grimshaw. Braid: Integrating Task and Data Parallelism. In *Frontiers '95*, pages 211–219, 1995.

[168] A. Wikstrom. *Functional Programming Using Standard ML*. Prentice Hall, 1987.

[169] B. Wilkinson and M. Allen. *Parallel Programming: Techniques and Applications using Networked Workstations and Parallel Computers*. Prentice Hall, 1999.

[170] K. Yamazaki and S. Ando. A Case-Based Parallel Programming System. In *Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems*, pages 238–245. IEEE Computer Society, 1998.

[171] J. Yang. *Co-ordination Based Structured Parallel Programming*. PhD thesis, Department of Computing, Imperial College of Science, Technology and Medicine, University of London, September 1998.

[172] J. Zobel. *Writing for Computer Science: The Art of Effective Communication*. Springer-Verlag, 1997.