Hybrid Skeletons for Parallel Higher Order Function Evaluation

by

Paul Crispin Bristow

Submitted for the Degree of

Master of Philosophy

at Heriot-Watt University

on Completion of Research in the

Department of Computing and Electrical Engineering

November 2001

I hereby declare that the work presented in this thesis was carried out by myself at Heriot-Watt University, Edinburgh, except where due acknowledgement is made, and has not been submitted for any other degree.

Paul Crispin Bristow

Contents

List of Tables

List of Figures

Acknowledgements

There are a number of people and organisations I would like to thank for their assistance, and without whom this thesis could not have been completed.

From my time at Heriot-Watt during which I carried out the research detailed here, my supervisor Dr Peter King, Dr Greg Michaelson, and Dr Norman Scaife. Greg was the driving force behind the project of which this thesis is a part and a great help both during the research itself and the write up. Norman was the person whom I worked most closely with on a day to day basis and whose knowledge and experience I benefited from greatly. Thanks also to EPSRC, the funding body for this research as part of grant GR/L42889.

The majority of the thesis was written after joining Motorola UK Research Laboratory. From there I would like to thank the director of the laboratory, Dr Andy Aftalak, for his generous allowance of time to assist me in writing up. Also, my immediate supervisor, Dr Clive Jervis, for his generous supply of motivation and thinly veiled threats, without which I would probably not have vacated the local hostelries for sufficient time to complete this thesis. Finally, thanks to Dr David King and Dr Bill Mitchell for their comments on the draft of this thesis.

Abstract

Algorithmic skeletons are a promising method for parallelising higher order functions. Although much work has been published in the area of skeletons, very little has been done on nested skeletons. Here we present the design and implementation of nested skeletons for the higher order functions *map* and *fold* along with a coordination layer for connecting the skeletons together. We also give details of the compiler that uses them.

The topology of skeletons exploited in parallelising a program is based upon a model of a program's semantics that indicates which higher order functions are to be implemented as skeletons. The processors allocated to specific skeletons within a program are determined at program startup. Where insufficient resources are available to parallelise a given skeleton it will be executed sequentially. This does not affect the ability of other skeletons to be executed in parallel. As a result of this mixed mode operation, we have termed them hybrid skeletons.

Three example programs using the skeletons are presented, including a matrix multiplication example that shows the stages in the parallelisation process. Results are given for all the examples and are discussed in detail. The results achieved with unnested skeletons are encouraging and add weight to the case for skeleton based parallel programming. The results for nested skeletons are in all cases slightly worse than those with unnested skeletons. Considering the example programs, it is concluded that they contain little opportunity for effective utilisation of this approach to nested parallelism.

To Rosey

Chapter 1

Introduction

This thesis describes in detail the design, implementation and execution of a new style of hybrid skeletal parallel programming construct.

Skeletal programming constructs, or skeletons, are a method of parallel programming that abstract away from the complexities of parallel programming. An individual skeleton encapsulates a parallel implementation of a well understood programming paradigm. By combining skeletons a parallel program can be constructed without recourse to explicit parallel programming.

A program can combine the use of multiple skeletons in two ways, either one after another or at the same time. When skeletons are combined one after another we refer to them as being sequenced, or executing sequentially. In order for skeletons to be executed at the same time they must be combined in a hierarchical fashion with one or more skeletons being subordinate to one outermost skeleton. The hierarchical combination can be executed with the result that all the skeletons are executed at the same time, a process we refer to as nesting. These two methods can themselves be combined to create both sequences of nested skeletons and nestings of sequenced skeletons.

Nesting skeletons may lead to benefits not just in terms of types of problem that can be solved in parallel but may also help improve utilisation of specific platforms. For example, by transforming a problem written in terms of one skeleton into one using two, we might help distribute communications and coordination load and hence reduce bottlenecks.

Nesting of skeletons presents many problems. Differing skeletons may be implemented using processor topologies that are not easily combined. Additionally, it may be difficult to coordinate between skeletons with very different core algorithms.

In this introductory chapter we describe the underlying concepts that give rise to skeletons. We present in some detail the compiler that was the parent project and consumer of the skeletons. We also describe the contributions of this thesis and summarise its structure.

1.1 Higher Order Functions as Skeletons

Given a purely functional application of the *map* higher order function (Section 3.3) to a list of data objects, opportunities for data parallelism can be created by splitting the list into sublists. The *map* function can then be applied to each of the sublists on different processors with the resultant sublists being combined on one processor to form the final result.

With some higher order functions it may be necessary to enforce more rigorous restrictions to enable splitting and recombining without affecting the semantics of a computation. In the case of the *fold* higher order function (Section 3.4) we restrict the first argument to be associative and commutative to enable parallelisation.

Nesting of skeletons can be achieved at a syntactic level by providing one higher order function as the functional argument to another. For example,

$$map\ (fold\ op{+}\ \ 0) \tag{1.1}$$

could be applied to a list of lists of integers to return a list of integers, in which each element is the sum of one of the original lists of integers. The implementation of this syntactically simple combination is far more complex; more details are given in Chapter 3.

We have implemented two nestable skeletons for parallel evaluation of Standard ML's *map* and *fold* higher order functions, and a generic control structure. These are known as *pmap*, *pfold* and *pskel* respectively. Although *fold* is usually expressed as either *foldl* or *foldr*, we make no distinction as our preconditions of associativity and commutativity mean that they are equivalent.

The skeletons are able to nest arbitrarily deep, with one minor limitation; once the execution of a skeleton is completed (at any nesting level) we cannot execute another skeleton and then return to the first, this limitation is due to the processor allocation scheme. Once a skeleton terminates, its processors will be reallocated to any skeletons that subsequently begin execution, hence, the processors will only still be available to restart a skeleton if no further skeletons have begun execution. It is acceptable to restart the first skeleton again if we have executed no other skeleton. Such a situation, where it is acceptable to restart a skeleton, would occur if a function containing a single skeleton were called several times, one immediately after the other. However, if the function contained two or more skeletons such behaviour would not be possible.

1.2 An Automatically Parallelising Compiler for Standard ML

In the previous section the idea of simply replacing a higher order function with a parallel equivalent was presented. Given manual modification of Standard ML (SML) [28] source code this technique would suffice for simple experimentation with skeletons. Due to its functional nature and lack of side effects, in code written in a purely functional style, SML is eminently suitable for this task. Although the same could be said of other functional

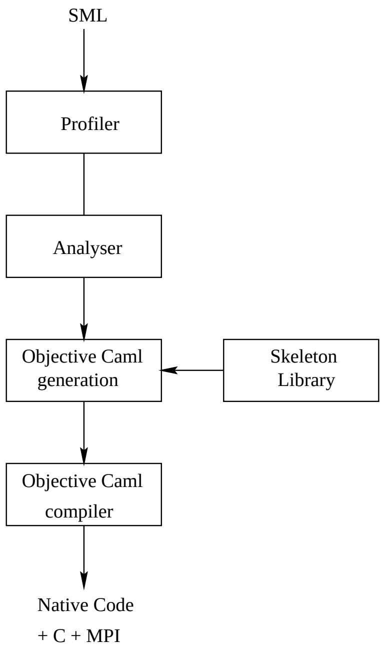

Figure 1.1: The compiler spine

languages, SML has the additional benefits of being standardised, complete with a large standard library, and having several implementations available.

The skeletons that we later describe were not developed for manual experimentation in the manner mentioned above. They form the parallelising component of an automatically parallelising compiler. The compiler is discussed in some detail and example applications are given in [24]. The compiler and the skeletons are still being used for research purposes, [26], [23], and [27] also describe applications in which the compiler has been used. Here we present a description of the compiler, biased towards the interfacing details between the skeletons and the compiler.

1.2.1 The compiler spine

Prior to describing the individual components of the compiler and how they interface with the skeletons, we first present an overview of how they combine to create a parallel program. Fig 1.1 shows a high level flow chart of the stages of compilation, the process can be summarised by the following four stages which correspond to the stages shown in the figure.

1. Identify which parts of the program take longest (Profiler).

2. Identify any higher order functions worth parallelising within the parts of the program identified by 1 and check for data dependencies within these (Analyser).

3. Perform program rewriting and code generation using information from 2 (Objective Caml generation).

4. Compilation of code generated by 3 (Objective Caml compiler).

It is at stage 2 that the skeletons are incorporated to create a parallel program, section 1.3 describes the interfacing of the skeletons to the compiler. It is only via the skeletons that parallelisation is provided, the purpose of the compiler being to identify where to parallelise a program, to make it amenable to skeleton usage, and to appropriately incorporate calls to the skeletons.

At the time of writing not all stages of the compiler have been completed. The stages that are incomplete are those for identifying which higher order functions to parallelise and for the identification of free values (Section 1.2.4). Nevertheless, with a certain amount of guidance in terms of manually replacing *maps* and *folds* with their parallel equivalents, and identifying free values when necessary, we were able to compile SML programs to parallel executables.

1.2.2 Profiler

The Profiler identifies which parts of the program require the greatest execution time. This is achieved in a platform independent way by counting each runtime occurrence of an SML Structural Operational Semantic rule (S.O.S). The S.O.S counts can then be combined with temporal costs for any target platform to generate actual run time estimates. In this way it is possible to generate costs for many different platforms by only profiling on one platform. The platform dependent costs are generated by a suite of test programs. This technique is discussed in more detail by Bratvold [2].

1.2.3 Analyser

The compiler performs a number of complex analyses on the source program. However, those of interest are the phases identifying free values, higher order functions (h.o.fs) and their dependencies. The actual h.o.fs that are to be parallelised are determined using the earlier profiling data. It is unlikely to be desirable to replace all h.o.f occurrences with parallel equivalents in any given program. The dependencies between the skeletons are described by a tree structure known as an *Abstract Network* (Section 4.1).

1.2.4 Free Values

Free value analysis is a complex issue in its own right and is discussed in detail in [24]. Free values arise when a value is calculated on one processor but is not available on other processors that require it. This occurs as a result of the way in which computations are split amongst processors. In general it can be difficult to detect. From our point of view there are two types of free value. The first is handled by the compiler rewriting source code to produce new functions with no free values, which we can ignore as it does not affect the skeletons. The second type results in extra values that are not arguments to a h.o.f but must still be transmitted to relevant processors at run time. Hereafter, when we refer to free values we mean this second type.

1.2.5 Objective Caml Generation

This in effect is the final stage of the compiler. The original program has been translated into Objective Caml [12]. In addition to the translation, the selected h.o.fs have been replaced with calls to the appropriate skeletons. The skeletons have been annotated with extra arguments encapsulating any free values. Objective Caml is an ML dialect

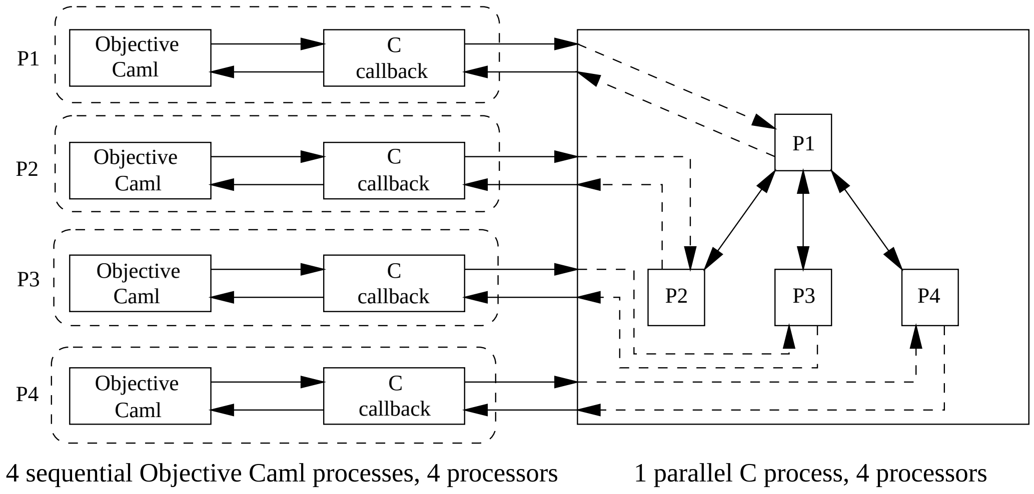

Figure 1.2: Interfacing sequential Objective Caml with parallel C

similar to SML but with a light weight and efficient implementation. The relatively small memory footprint of Objective Caml executables was important given the modest memory available on some parallel machines.

1.3 Interfacing and Implementation Languages

Although the skeletons form part of an automatically parallelising compiler for Standard ML, the skeletons do not interface to Standard ML. As noted above, the final translated program is in Objective Caml. From the perspective of the Objective Caml program, when a skeleton is invoked, it is simply calling a higher order function, indeed the function invocation looks no different to any other higher order function usage.

The Objective Caml runtime is aware that the higher order function is implemented by a call to an external object via its callback interface to C and calls the function, registered with its callback interface, in the skeleton library. The runtime system has no knowledge of the parallel nature of the call it is making and simply waits for the function to return. This process will have occurred on every processor in the parallel computer, allowing the skeleton library to take control of all processors and carry out a parallel computation. Once the parallel computation is complete all the processors return to their calling Objective Caml program and proceed sequentially, figure 1.2 illustrates this process.

The skeletons were coded using ANSI C [19] and MPI [15]. MPI is a well recognised system for parallel programming with several implementations available on many platforms. This combination enabled us to maintain a high degree of platform independence. In fact, the compiler and the compiled programs have been run on Solaris, Linux, Fujitsu AP1000 and AP3000 platforms either for development or performance measuring purposes.

1.4 Thesis Outline

This chapter has introduced the concepts of parallelising and nesting of higher order functions. The compiler that targets the skeletons was introduced and its various stages described. The rest of the thesis is as follows.

Chapter 2 is an in depth survey of particularly relevant works in the field of parallel skeletons. We describe in detail the various works and summarise and compare them.

Chapter 3 forms a description of the skeletons we have implemented. We give details of the algorithms for the skeletons and the related coordination layer.

Chapter 4 introduces the skeleton network that represents the occurrences of skeletons within a source program. It also describes, using this network, how processors are allocated to skeletons at program startup time. Additionally it notes that some skeletons may actually be executed sequentially if insufficient resources are available.

Chapter 5 presents a matrix multiplication example program. All the key stages of parallelisation are illustrated; two further examples are also presented. Performance data and graphs are included for all examples.

1.5 Contributions of Thesis

The contributions of the following are summarised here.

- A focused survey compares and contrasts the various skeleton methodologies currently employed. The survey shows the value of the skeleton concept in easing the writing of parallel programs.

- The thesis shows that it is possible to arbitrarily deeply nest skeletons, describes, and shows the results of a method for implementing these skeletons. This system is flexible enough to allow the final skeletal program to be executed on varying numbers of processors without any recompiling or coordination assistance from a user.

- The skeletons implemented include a lightweight startup phase rather than incur the overhead of an inter-skeleton, distributed runtime resource allocation and resolution algorithm. Additionally, unlike some earlier skeleton systems there is no limit on the number of sequentially occurring skeletons at a program's top level.

- The system is used on several different platforms, and in a heterogeneous manner although only for debugging and testing purposes (Section 1.3). This demonstrates the flexibility and robustness of the skeletons implemented, adding further weight to the arguments for functional, skeleton based approaches. All that was required to port the program was re-compilation.

- Experimental results are presented that prove the validity of the skeleton approach to parallel programming.

Chapter 2

Literature Review

2.1 Introduction

In Section 1.1 we introduced skeletons as a parallel replacement for equivalent higher order functions. The term algorithmic skeleton was first used by Cole [4]. However, there are earlier papers working in the same direction. For example, Cole cites ZAPP (Zero Assignment Parallel Processor) [22] that mapped a virtual, divide and conquer processor tree onto a real, fixed network of processor-memory pairs and provided a functional interface. Much work has since been carried out in the area of algorithmic skeletons under various names. Bratvold [2] identifies the terms "templates, paradigms, patterns, forms or parallel constructs" and notes that they are used to identify broadly the same concepts. The work surveyed here restricts itself to the terms skeletons ([4], [16], [3], [7]), parallel constructs [29], and recognised functions [32].

Here we first consider the work of Cole and then move onto the work of probably the three most prolific groups active within the skeletal field: Darlington *et al* at Imperial College, Pelagatti and Danelutto at Pisa, and Eden, a distributed project at Philipps Universität Marburg and Universidad Complutense de Madrid. We also describe HOPP and EKTRAN due to the similarities between these works and our own.

2.2 Cole

As noted in Section 2.1, it was Cole that introduced the term Skeleton. In [4] he states that a universal programming framework for parallel programming would be incompatible with a uniformly efficient parallel implementation. However, he notes that even though no such universal abstract model may exist we should not abandon high levels of abstraction in our search for efficient parallelisation. This highly abstract model should not even be parallel in nature due to the difficulty of parallel decomposition of problems.

Cole identifies two sources as the inspiration for his model. The first is the large amount of work on sequential algorithms that has identified common algorithmic structures. These can be targeted at specific instances just by adding problem specific details to the core algorithm. Examples of this generalised algorithmic approach cited are divide and conquer, dynamic programming, and simulated annealing.

The second inspiration is higher order functions. Higher order functions are identified as a useful way to realise the generalised algorithmic approach. This is because of the way in which they express the computational structure of a problem without specific details, the details being provided by the functional argument to the higher order function.

The concept of higher order functions encapsulating parallel algorithms is given the term algorithmic skeletons. Skeleton usage in a source program is identified as being only "those programs whose outermost structure matches one of the acceptable higher order functions." The skeletons themselves are restricted to being a subset of higher order functions that correspond to a "good algorithmic technique." It is noted that a similar approach could be used in imperative languages although it would be more suited to those languages that allow the passing of procedures as arguments.

2.2.1 Skeletons

Four skeletons are considered and are analysed with regards to an intended implementation on a 2-D mesh of processors. The purpose of considering this specific topology is to show that at least one effective implementation exists, although it is claimed that many others ought to exist. Example applications using the skeletons are considered but no actual implementation is carried out.

Fixed Degree Divide & Conquer (FDDC)

The *FDDC* skeleton is based on the well known divide and conquer method of recursively splitting a problem into sub-problems until an indivisible problem remains; this is then solved recursively. A restriction is placed on the degree of splitting that can be performed. Each problem which is not indivisible must, upon splitting, generate exactly k sub-problems; the programmer must specify k.

Iterative Combination (IC)

The *IC* skeleton is intended for implementing the types of algorithm in which structure is imposed upon an initially unstructured set of homogeneous objects. This skeleton assumes rules for combining objects, placing a value on a combination of objects, and selecting a combination given a number of combinations and their values. Using these prerequisites, the skeleton combines all the objects into one structured object or terminates early with more than one object if no further acceptable combinations exist.

When attempting to combine objects it may be necessary for each processor to use information from all processors. This is avoided by using a ring based algorithm in which a processor merges its own combination data with its predecessor and passes it onto its successor. Once a complete circuit has been made all processors can make the correct selection. The ring used in the algorithm is a Hamiltonian circuit of the 2-D mesh, i.e. a non-repeating, connected sequence of nodes in the mesh that passes through all nodes.

2.2.2 *Task Queue (TQ)*

The *TQ* skeleton is the only skeleton to be explicitly parallel in its specification. Cole observes that the "isolation of individual skeletons within the overall framework means that parallelism can be introduced at this level in a restricted way."

The skeleton assumes a large, shared data structure containing a number of tasks. Each processor executes some procedure on an element of the task queue that maps from task to solution. The procedure in reaching its solution may generate further tasks which are placed on the task queue. The concurrent execution of the procedure continues until no further tasks remain.

2.2.3 *Cluster*

The motivation for the *Cluster* skeleton is the reverse of the other skeletons that identify a useful algorithmic technique and express it in a manner suitable for parallel implementation. This skeleton starts with the 2-D mesh and attempts to generate a useful pattern of decomposition and distribution to which it is well suited.

Based on the observation that a square grid is well suited to partitioning into independent sub-grids, the skeleton clusters together component parts of a whole problem into independent groups that contribute to a global solution. The flexibility of the grouping allows a variation of the *FDDC* skeleton that does not require the fixed sized division of problems. By combining this with the Hamiltonian circuit techniques in the *IC* skeleton the individual groups can be pipelined.

2.3 Darlington *et al*

Darlington's group at Imperial College have produced many works in the area of skeletons.

In [7] they argue that the meaning and behaviour of parallel programs are orthogonal issues and should be separated, and present skeletons as a method of achieving the separation. Their skeletons are expressed as higher order functions in a functional language with the skeletons providing the only parallelism in a program; all other functions are evaluated sequentially.

The idea that some skeletons may more naturally be implemented on some platforms than others is noted. As some skeletons may not have efficient implementations on some platforms, a library of inter skeleton transformations is proposed. Given that the set of skeletons is fixed, so too then are the transformations between them. By succeeding a stage of hardware driven skeleton choice with optimisations such as modifying granularity the group hope to maintain performance portability.

2.3.1 The Skeletons

Four skeletons are presented, covering both data and task parallelism, with the definitions being in Haskell [18]. The shape of the computations represented will probably be familiar to anyone with a knowledge of parallelism. The list functionals *map* and *foldr* are used in the following skeletons. We give their Haskell definitions here, taken from [13] with the exception of the type definition for *foldr* which is our own.

$$\begin{array}{lcl} map & :: & (a \to b) \to [a] \to [b] \\ map\ f\ [\,] & = & [\,] \\ map\ f\ (x :: xs) & = & f\ x\ ::\ map\ f\ xs \end{array}$$

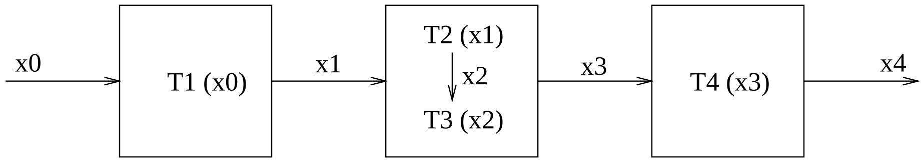

Figure 2.3: A pipeline

$$
\begin{aligned}
foldr \quad & :: \quad (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow' b \\
foldr\ op\ e\ [\,] \quad & = \quad e \\
foldr\ op\ e\ (x :: xs) \quad & = \quad x\ \text{`}op\text{`}\ \ foldr\ op\ e\ xs
\end{aligned}
$$

PIPE

PIPE represents the well known pipeline method for achieving task parallelism. Each processor is assigned one or more tasks, with the output of a task, and in many cases a processor, forming the input of its successor. Fig 2.3 shows an example of a pipeline that has four tasks and is distributed across three processors. *PIPE* is defined as follows:

$$
\begin{aligned}
PIPE \quad & :: \quad [[\alpha] \rightarrow [\alpha]] \rightarrow [\alpha] \rightarrow [\alpha] \\
PIPE \quad & = \quad foldr\ (\circ)
\end{aligned}
$$

FARM

The *FARM* skeleton describes the data-parallel processor farm paradigm. This is equivalent to the most obvious and natural implementation of a map skeleton (Section 3.3). *FARM* is defined as follows:

$$
\begin{aligned}
& FARM \quad :: \quad (\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow \alpha \rightarrow ([\beta] \rightarrow [\gamma]) \\
& FARM \quad f\ env = map\ (f\ env)
\end{aligned}
$$

Divide and Conquer - *DC*

Divide and Conquer (DC) will be as familiar to the sequential programmer as to the parallel. Tasks are recursively split into sub-tasks. The splitting continues until only trivial tasks remain. The trivial tasks are solved and combined to produce the final result. Darlington defines DC as follows:

$$
\begin{aligned}
&DC \;\; :: \;\; (\alpha \rightarrow Bool) \rightarrow (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow [\alpha]) \rightarrow ([\beta] \rightarrow \beta) \rightarrow \alpha \rightarrow \beta \\
&DC \;\; t\ s\ d\ c\ x \\
&\qquad\quad \mid t\ x = s\ x \\
&\qquad\quad \mid not\ (t\ x) = (c\ \circ\ map\ (DC\ t\ s\ d\ c)\ \circ\ d)\ x
\end{aligned}
$$

Reduce and Map over Pairs - *RaMP*

The final skeleton is a little more obscure than the preceding three. $RaMP$ is intended for situations involving two lists of objects where every element in the first list can interact with every element in the second. A typical example of this is the classic N-body problem, often used to model gravitational effects, that consists of a set of objects in which every object interacts with every other object. $RaMP$ is defined as follows:

$$
\begin{aligned}
&RaMP \;\; :: \;\; (\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow (\gamma \rightarrow \gamma \rightarrow \gamma) \rightarrow [\alpha] \rightarrow [\beta] \rightarrow [\gamma] \\
&RaMP \;\; f\ g\ as\ bs = \\
&\qquad\qquad map\ h\ as \\
&\qquad\qquad where\ h\ x = foldr1\ g\ (map\ (f\ x)\ bs)
\end{aligned}
$$

It should be noted that the paper suggests that this skeleton is intended for "initial specification" only. It should be replaced with a more machine oriented skeleton using the transformations mentioned earlier.

An Example

A ray tracing example is presented using the *RaMP* skeleton. It is transformed into two equivalent programs, one using *FARM* and the other *PIPE*. The programs were written in HOPE^{+} and compiled to C with the final executable being on a Meiko Computing Surface.

Performance models were developed for the skeletons and compared to the observed results. In the case of the *FARM* implementation the two were very close. For the *PIPE* there were some discrepancies, these were attributed to poor modelling of the garbage collector. As the system was only implemented on one platform we cannot make any judgements about the ability of the transformations in maintaining performance portability.

2.3.2 Application Specific Skeletons

In [10] the concept of application specific skeletons is introduced. These are skeletons with an intended use in specific application domains. An example based on solid modelling using Constructive Solid Geometry (CSG) trees is presented. Two skeletons are defined, *transformCSG* and *reduceCSG*. It is noted that these correspond to *map* and *fold* over CSG trees.

The paper points out that CSG trees, although a natural representation for solid objects, are not efficient to process. This motivates an equivalent pair of skeletons that utilise octrees, a representation based on spatial decomposition that is more amenable to efficient processing. By transforming from CSG trees to octrees the example allows solids to be represented naturally in CSG form but processed in the more efficient octree representation.

Although the paper discusses the implementation of the, non application specific, base skeletons (Section 2.3.1) it is not indicated that CSG or octree skeletons were actually implemented. Additionally, no examples of transformations from octree representations to their base skeletons are given, unlike the *RaMP* example. This is understandable given that efficient implementations using domain specific knowledge is a major motivating factor given by the paper for application specific skeletons. However, it would have been an elegant and cheap way to realise an implementation and demonstrate the power and practicability of skeletal transformations.

2.3.3 Structured Coordination Language - SCL

With [8] and [9] the group modified their approach to the skeleton concept. They proposed a two level system of parallel programs. The top level is a coordination model and the lower level a computation model consisting of theoretically any sequential language.

The coordination level, although functional and consisting of skeletons, is unlike anything seen previously from the group but it does have much in common with P^{3}L (Section 2.4). The skeletons describe all the relevant aspects of a parallel program, that is, "partitioning, data placement, data movement and control flow". Hence, these skeletons in many cases are at a much lower level than earlier examples. Additionally, a parallel array datatype, represented by << ... >>, has been introduced as the underlying datatype upon which skeletons operate. This parallel array datatype has an index and an element type; it is effectively a partitioning of sequential arrays (Section 2.3.4).

There are several motivating factors given for their new approach, some of the more important are as follows. By composing existing modules with the coordination language code reuse should be practical, particularly for sequential programs that are being parallelised. This is attractive but should be considered in the context of the precondition that all sequential modules should be self contained to allow transformations to be applied to the coordination language. These transformations are intended to optimise the parallel coordination in terms of the skeletons available on a platform. As with earlier papers, they also hope to achieve performance portability by having specialised implementations of the skeletons for specific platforms.

2.3.4 SCL Skeleton Types

Skeletons are split into three subtypes: configuration, elementary and computational. These are described below.

Configuration Skeletons

Configuration skeletons, it could be argued, are hardly skeletons at all. They lay bare, in a platform independent way, the partitioning and placement of data in a parallel system. These low level skeletons include operators such as:

- **partition**: creates a parallel array by partitioning a sequential one.

$$\begin{aligned} partition \;&::\; Partition\_pattern \rightarrow SeqArray\ index\ \alpha \rightarrow \\ &\quad ParArray\ index\ (SeqArray\ index\ \alpha) \end{aligned}$$

- **align**: groups corresponding subarrays in differing parallel arrays into a parallel array of triples.

$$\begin{aligned} align \;&::\; ParArray\ index\ \alpha \rightarrow ParArray\ index\ \beta \rightarrow \\ &\quad ParArray\ index\ (\alpha,\ \beta) \end{aligned}$$

- **gather**: recombines parallel arrays into sequential arrays.

$$gather \;::\; ParArray\ index\ (SeqArray\ index\ \alpha) \rightarrow SeqArray\ index\ \alpha$$

Hence, a parallel program might partition a pair of sequential arrays and align them such that the configuration of the parallel arrays matches the underlying hardware. After performing some computation in parallel using one of the elementary or computational skeletons described later the results could be gathered to form a final result.

Elementary Skeletons

Elementary skeletons abstract basic data parallel operations over the parallel array type. The more familiar skeletons are found in this category but with modified types to accommodate the parallel array type. There is also an *imap* skeleton which is similar to *map* but includes the index of the array element being processed as one of its arguments. The common skeletons are shown below.

$$map :: (\alpha \rightarrow \beta) \rightarrow ParArray\ index\ \alpha\ \rightarrow ParArray\ index\ \beta$$

$$map\ f\ \ << x_0, ..., x_n >> = << f\ x_0, ..., f\ x_n >>$$

$$imap :: (index \rightarrow \alpha \rightarrow \beta) \rightarrow\ ParArray\ index\ \alpha \rightarrow ParArray\ index\ \beta$$

$$imap\ f\ \ << x_0, ..., x_n >> = << f\ 0\ x_0, ..., f\ n\ x_n >>$$

$$fold :: (\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow ParArray\ index\ \alpha \rightarrow \alpha$$

$$fold\ (\oplus)\ \ << x_0, ..., x_n >> =\ x_0\ \oplus ... \oplus\ x_n$$

$$scan\ :: \ (\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow ParArray\ index\ \alpha \rightarrow ParArray\ index\ \alpha$$

$$scan\ (\oplus)\ \ << x_0,\ x_1, ..., x_n >> = << x_0,\ x_0\ \oplus\ x_1, ..., x_0\ \oplus ... \oplus x_n >>$$

This classification also includes a number of bulk data movement functions. These clearly fit into the description of data parallel operations, however, unlike *map et al* above, they do not transform their arguments. These include, amongst others, a rotate function for moving data around networks of processors; a broadcast function for sending a data item to all processors; send and fetch functions for irregular data movement where the destination processor is a function of the current index. Here we show the definition of the relatively simple rotate function.

$$rotate \; :: \; Int \rightarrow ParArray \; Int \; \alpha \rightarrow ParArray \; Int \; \alpha$$

$$rotate \; k \; A \; = \; << i \; := \; A \; ((i+k) \; mod \; SIZE(A)) \mid i \leftarrow [1..SIZE(A)] >>$$

Computational Skeletons

Computational skeletons abstract commonly used computational patterns. In this classification we find farm again (with its new parallel array typing) and *SPMD* (Single Program Multiple Data).

$$farm \; :: \; (\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow \alpha \rightarrow ParArray \; index \; \beta \rightarrow ParArray \; index \; \gamma$$

$$farm \; f \; env \; = \; map \; (f \; env)$$

$$SPMD \; [] \; = \; id$$

$$SPMD \; (gf, \; lf) \; : \; fs \; = \; (SPMD \; fs) \; \circ \; (gf \; \circ \; (imap \; lf))$$

We can see some overlap here between *farm* and *map*. Farm is defined in terms of map, and as noted previously (Section 2.3.1), *map* can easily be implemented in terms of *farm*. Two iteration skeletons are also defined, *iterUntil* and *iterFor*, which use a condition and a counter to control termination respectively.

It is claimed that by combining *SPMD* with a skeletal transformation it is possible to support nested parallelism. It is unclear as to how effective this method would be in general. Certainly, it should be noted that supporting nested parallelism by transforming it to unnested parallelism is very different, from an implementation point of view, to actually supporting nested parallelism at runtime.

SCL Examples

Two examples of a matrix multiplication are shown in [9], one using a row-column oriented algorithm and the other a block oriented algorithm. Both algorithms use a combination of the *gather*, *iterFor* and *SPMD* skeletons to achieve an algorithm at the coordination language level. The row-oriented example is developed further, the code being hand translated into an unnested form and then converted to FORTRAN by hand. The computation level code was also written in FORTRAN. The final executable was compiled for a Fujitsu AP1000. The relative speed-up figures for the example are good at around seventy on one hundred processors.

2.4 P^{3}L

P^{3}L is a parallel coordination language based on skeletons. It is a development of the Pisa Parallel Processing Project (P^4).

In [29] Danelutto *et al* present the methodology upon which the P^{3}L compiler is based. Like the group at Imperial (Section 2.3), they argue that parallel programs should be split into two distinct parts, however, they disagree as to what these two parts are. They identify the two parts as the code necessary for implementing a particular algorithm and the code for adapting an algorithm to a specific platform. These two code types are labelled "true algorithmic code" and "machine dependent code" respectively. The machine dependent code takes care of the specifics of a target platform such as communication, process topology, and load balancing.

It is important to note that true algorithmic code in P^{3}L is not analogous to Imperial's, sequential, function arguments. This highlights the most significant difference between P^3L and the Imperial work: separating the machine dependent code from the machine independent code instead of separating the parallel code from the sequential. Another significant difference is the lack of an explicitly parallel data type. However, they achieve a similar effect with free variables (Section 2.4.1). Additionally, P^3L uses imperative code throughout. In spite of the fundamental differences in paradigm there are major similarities between the two works in terms of compiler and skeletons.

2.4.1 Free Variables

Free variables are a mechanism for specifying array distributions over corresponding arrays of virtual processors. Although only one small part of the P^3L project, they are a particularly attractive and intuitive method of specifying data distribution. The following code snippet from [5] demonstrates their usage.

```
map foo in(int a[n][m], int b[n], int c[n][m]) out(int d[n][m])
    body in (a[*i][*j], b[], c[][*j]) out (d[*][*j])
end map
```

The meaning of free variables in terms of the example is that some virtual processor `(i,j)` will receive the array element `a[i][j]`. Also, every virtual processor will receive all of `b`; every virtual processor will receive the column of `c`, `(c[][j])` that corresponds to its position in the virtual processor grid; they will also receive the index `j` that corresponds to their position in the grid. Figure 2.4 represents this graphically for `a` and `c`.

Clearly, this example assumes a one to one correspondence between virtual processors and array size. More complex expressions can be constructed by including expressions that include a mixture of free variables and constants.

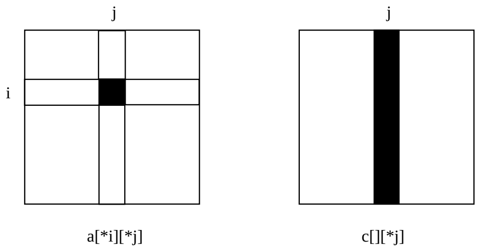

Figure 2.4: P^{3}L Free variables selection

2.4.2 P^{3}L Programs

In P^{3}L a user writes explicitly parallel programs. To do this they use a fixed set of high level parallel constructs. These are similar to computational skeletons (Section 2.3.4) in that they specify explicit parallel computation patterns. The programmer has no means of accessing low level primitive operations such as send/receive pairs. Although these high level operators are referred to as parallel constructs they are, as is apparent from the comparison with Imperial's work, analogous to skeletons. Indeed, in [5] they are actually referred to as skeletons. The parallel constructs reference sequential code blocks, in a similar way to the Imperial work, to create an abstract description of a parallel solution.

The P^{3}L compiler uses a set of implementation templates to map the parallel algorithm onto a real machine. It also carries out the process to processor mapping. In effect, the compiler is taking true algorithmic code and adding machine dependent code to it. It attempts to do this in an efficient way by using parametric data associated with the template implementations on a given platform.

The language's parallel constructs and the resulting implementation templates can be nested. Nesting is crafted by the application programmer by appropriate usage of the parallel constructs. Details of how the compiler deals with issues arising from nesting such as resolving conflicting parallel constructs is not given. However, much complexity is removed by forcing the programmer to explicitly specify the correspondence between skeletons and the data flow between them. The three most complex parallel constructs (Section 2.4.3) do not support nesting.

2.4.3 P^{3}L Constructs

As mentioned earlier, P^{3}L programs consist of a number of parallel constructs. These constructs form pure functional modules, hence, allowing transformation. The transformations are important as they facilitate load balancing by modifying the construct tree that is a direct mapping from the source program. The seven constructs that form the current version of P^{3}L are detailed below along with associated notation. Note that any of the following P^{3}L constructs can be denoted $\mathcal{S}_i$, that is, $\mathcal{S}_i$ can be considered a wild card that matches any construct.

- **Sequential**

  The sequential construct represents the leaves of the construct tree. Instances of this are directly analogous to the functional arguments of the Imperial skeletons. These constructs are not permitted to keep an internal state as this would conflict with any transformations that might be made to the construct tree. A sequential model i is denoted by S_i.

- **Pipe**

  Pipe expresses pipeline parallelism. A pipe with n stages is denoted $\Pi(\mathcal{S}_1, ..., \mathcal{S}_n)$.

- **Farm**

  Farm expresses the traditional process farm. A farm with a work module $\mathcal{S}$ is denoted $\mathcal{F}(\mathcal{S})$.

- **Loop**

  Loop iterates parallel computation of a loop body over its input stream. It is denoted $L(\mathcal{S}, C)$ where $\mathcal{S}$ is the loop body and C is a sequential module that computes termination.

- **Geometric**

  This expresses geometric parallelism over an n dimensional array. It is denoted

$\mathcal{G}(l, \overline{N}, \overline{S}, S)$ where l is the number of iterations, $\overline{N}$ is the input and output array dimensions, $\overline{S}$ is a stencil function for controlling data allocation to subcubes within the virtual processor grid, and S is a sequential computation body. Hence, this construct does not support nesting.

- **Map**

  Map , although familiar in name, is different to the conventional functional concept. It is in fact a special case of the geometric construct. The initial data distribution is the same as the geometric case but all computations are independent so no iteration is performed and no data stencil is necessary. The denotation is $\mathcal{M}(\overline{N}, S)$ with $\overline{N}$ and S having the same meaning as in the geometric case (i.e. S is sequential).

- **Reduce**

  Reduce is the familiar binary tree style reduction using an associative operator. The Denotation is $\mathcal{R}(S)$ with S being a sequential operator. Again, as with geometric and map, the requirement that S be sequential removes the possibility of nesting within the construct body.

2.4.4 Examples

[1] gives several examples of convincing applications expressed in terms of the above denotation. All the examples are parallelised and runtimes measured. One of the examples is a data-parallel ray tracer. The structure of the parallel program is given as:

$$\Pi(S_1, \mathcal{M}([W, H], S_2), S_3)$$

where S_1 is the input stage, S_2 is the complete sequential ray tracing program and S_3 receives and outputs the final image.

The compiler assigns a 6x6 block of processors to the ray tracing problem and predicts its runtime within 11% of the actual measured runtime. However, no relative speedup figures are given. The runtime for a sequential version of the ray tracer is given but as this is executed on different hardware we cannot derive speedup data.

2.5 Eden

Eden [3] is a functional language with explicit process definitions designed for use on distributed memory parallel systems. Its significant features are process abstractions that specify process behaviour in a purely functional way; process instantiations for creating new processes based on process abstractions; and implicit communication based on streams that are transmitted via communication channels between processes.

Eden is similar to Imperial's work in that it uses a two level structure based on separating computation from coordination. Unlike Imperial's work, the computation and coordination models are not specified in separate notations. The computation model is based on Haskell [18] with the coordination being provided by extensions to the base language.

2.5.1 Process Abstractions

The keyword *process* is added to Haskell to describe process abstractions. A process abstraction has the following type and form.

$$p \;::\; \tilde{\tau} \rightarrow \tilde{\tau_k} \rightarrow \quad Process \quad (\tau_1, ..., \tau_m\ )\ (\acute{\tau_1}, ..., \acute{\tau_n})$$

$$\begin{aligned} p\ par_1\ ...\ par_k \ = \quad & \texttt{process} \quad (in_1, ..., in_m) \rightarrow (out_1, ..., out_n) \\ & \texttt{where} \quad equation_1 ... equation_r \end{aligned}$$

The where part of the statement provides the process body. The parts prior to that provide information about the static interface of the process abstraction. Note that $par_1, ..., par_k$

are parameters of the process abstraction. The arguments $in_1, ..., in_n$ are input channels and $out_1, ..., out_n$ are output channels.

There are significant differences between process parameters and input channels. Process parameters are passed before process creation, with the actual expression bodies of the parameters being copied into the created process. The parameters themselves are then evaluated on demand by the created processes. Input channels' values are passed after process creation, their values being produced by some sender and received via the communication channel.

2.5.2 Process Instantiation

Process creation occurs when a process abstraction is applied to a complete tuple of input expressions. An infix operator # is provided for creating new processes. This is called process instantiation and can be expressed as follows.

$$(p\ e_1, ..., e_k)\ \#\ (input\_exp_1, ..., input\_exp_m)$$

The following example from [3] shows the Eden code for a parameterised pipeline.

```
pipe :: [a->a] -> Process  [a]  [a]
pipe    f_list = process values -> results
                 where results =
                   case f_list of
                     [] -> values
                     (f:fs) -> if fs then map f values
                                     else pipe fs # (map f values)
```

2.5.3 Communication

Communication is based on unidirectional, one to one channels. Lists are the usual types to be sent over the channels. They are treated as streams to allow the overlapping of communication and computation. Generally, the outermost list structure in the interface of a process is assumed to be a stream. However the operators '<' and '>' are provided for explicitly marking streams when necessary.

A mechanism is supplied for dynamically generating channels at runtime. A process creates a channel and sends the name, along an existing channel, to another process. The receiving process can either return some data to the initiator of the channel or forward the channel to another process. It is the programmers responsibility to ensure that only one to one communication ensues, otherwise a runtime error will result.

2.5.4 Skeletons

In [14] a number of traditional skeletons are implemented in Eden. These include a pipe and a variation with bidirectional data flow, a two dimensional grid, a farm, and a divide and conquer skeleton. Examples of their usage are also shown. A number of other, less well known, skeletons are developed for reactive systems. These are typified by having a closed process graph and include client-server, ring, and graph.

Of particular note is the divide and conquer skeleton and its definition in terms of the farm skeleton.

```
divCon :: (a -> Bool) -> (a -> b) -> (a -> [a]) -> (a -> [b] -> b)
          -> Process a b
divCon trivial solve split combine = process x -> y where
  y = if trivial x then solve x
      else combine x
```

```
(farm (divCon trivial solve split combine) # split x)
```

This demonstrates an implicit nesting of the farm skeleton within itself, that is, a call to `divCon` will cause a call to farm which will in turn call `divCon` and result in nesting of farm. As can be seen from the code, the crafting of the two skeletons together is not a trivial task, although with sufficient attention to process types it should not be too arduous.

[14] goes on to say that having a skeleton catalogue "very much simplifies concurrent programming in Eden". This is clearly the case when a programmer finds exactly the skeleton required within the catalogue. It is likely also to be true even when an explicit crafting of together of skeletons is required. It is claimed that a high degree of parallelism is extracted with the skeletons. However, no performance results are presented for the skeletons so we cannot judge if this is the case.

2.5.5 Examples

In [20] a simple raytracer is presented. Two different parallel programs are generated using two different versions of a parallel map. The first is a conventional process farm, the second transmits the entire input list to each process at startup. They refer to this technique as direct mapping. For the data set used, a 400x400 pixel scene with 17 objects, the direct mapping example achieved better performance although not significantly so. The result are broadly similar to our own for our raytracer example (Section 5.2).

In [21] Eden is compared with GpH for parallel performance. Two problems are used, a program for solving a linear system of equations and a raytracer. The same algorithms are implemented in the two languages and compiled for a Beowulf cluster, with results being presented for up to sixteen processors. Both algorithms are run on two different data sets each. In all cases the Eden system achieved considerably better performance than the GpH system.

2.6 HOPP

Rangaswami's HOPP [32] provides a functional model of parallel programming based on higher order function use that is heavily biased towards providing an accurate cost model. The HOPP model has three parts: a program model, a machine model, and a cost model. HOPP's program model has a number of recognised functions which are analogous to skeletons. A program consists of a number of sequential phases each of which may contain recognised functions and user defined functions.

The set of recognised functions include the usual higher order functions *map, fold, scan,* and *filter*. Two versions of *fold* and *scan* are included: a version where each intermediate result is of fixed size and one where the intermediate result size grows with each application. The implementation is the same for each, it is only the execution costs which differs. A number of other recognised functions are also provided, the overall set being rather larger than that of most skeletal systems.

Recognised functions can be nested up to three deep but each nested recognised function must be embedded directly within the proceeding recognised function's functional argument. Nested recognised functions occurring elsewhere, such as being embedded within a user defined function, will not be parallelised.

The machine model describes various types of machines on the basis of their connection networks. The machines considered are hypercube, 2-D torus, linear array, and tree. The machine model is necessary to describe the data distribution.

The cost model estimates the parallel execution costs of a program on a given topology. To do this it combines the following characteristics.

- **Program Tree**

  This is a tree representing the program, it is derived from the source program. Each node in the tree is either a recognised function or a user defined function.

- **Parallel Machine Characteristics**

  These encapsulate the machine topology and its associated machine model, the number of processors, and the cost of sending a message. Message costs are specified in terms of a startup cost for sending a message and a data transmission speed.

- **Input List Characteristics**

  These include the level of nesting in each input list, the list sizes at each level, the size of each list element, and the relationship between sizes at each level. It is assumed that input lists are regular in nature and that nested lists, e.g. lists of lists, consist of sublists of equal length. The relationship between list sizes is used as a constraint to guide the selection of a parallel implementation.

- **Specifications for Sequential Functions**

  These represent the costs of execution of sequential, i.e. non-recognised, functions and the output sizes from those functions. Execution costs must be specified as being either constant or proportional to input size. It is assumed that processing of each list element takes uniform time. The output sizes are inferred from the output type; the inference makes the assumption that output list is the same length as the input list.

By combining the above information, an estimate is formed of the parallel runtimes of several different implementations of the same parallel program. The different versions arise from deciding whether or not to implement recognised functions sequentially or in parallel and whether to parallelise nested recognised functions or not.

The HOPP system describes a methodology and provides models and an analyser to perform compile time estimation of runtime costs but the actual compilation is performed by hand. Results are presented for three examples including graphs of runtimes against numbers of processors used. They show similar shaped curves for the predicted and actual results although the actual results measured are not always as good as those predicted.

2.7 EKTRAN

In his thesis, Hamdan [16] introduces Ektran, a simple functional language based on Backus' FP. The language includes constant bindings, function definitions and expressions. Ektran is used as a vehicle for parallel programming exploiting skeletons that are provided as part of the language implementation. From a skeletal point of view, there are many similarities between Hamdan's system and our own. The most significant difference from a user's point of view is that his system requires that a user be aware of the parallelisation occurring and code using the appropriate constructs. Our system on the other hand attempts to automatically identify the areas.

Ektran stipulates, rather than enforces, a number of restrictions that the programmer must be aware of. Nesting of skeletons is not allowed in the body of recursive functions. Functions are not allowed to be mutually recursive. Finally, it is the programmers responsibility to ensure that *fold* arguments meet associativity requirements.

A parallel program is constructed by writing a functional program in the usual way but with the programmer using the higher order functions that are known to have parallel equivalents where they wish to exploit parallelism. The higher order functions are replaced with these parallel equivalents at compile time.

The major skeletal difference between Hamdan's system and our own is that his skeletons are much more dynamic than ours. That is, his system allocates processors to skeletons at runtime dependent on available resources whereas ours takes a hybrid approach (Section 4.2.2). Which approach is more effective is unclear at this time. Certainly there are arguments both for and against either allocation strategy.

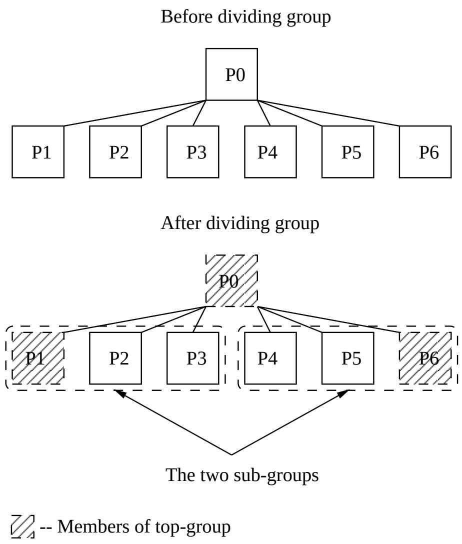

Figure 2.5: Group splitting

2.7.1 The Skeletons

The skeletons support nesting using routines for defining and controlling separate message passing groups in MPI. When a program begins execution, there is one root-group containing all processors. Whenever a nested skeleton is discovered a sub-group is created, a top-group connecting the sub-groups is also created. Figure 2.5 shows an existing root-group being split for a *map* skeleton with another, unspecified skeleton nested within it. Below we describe in some detail the algorithm for the *map* skeleton to illustrate how nesting is achieved. The other skeletons are described more succinctly.

map

There are four sub-algorithms in the *map* implementation, the process farm, the farmer algorithm, the flat worker, and the nested worker.

The process farm algorithm is basically a top level control structure for the *map* skeleton. It creates new communicator groups if necessary (due to nesting) and executes the farmer process on the skeleton's root processor. It also allocates the relevant processors to the appropriate sub-groups and executes the nested worker algorithm on them if nesting is present, otherwise the flat worker algorithm is executed.

The farmer algorithm sends a work packet to each of the available workers in its sub-group at start-up. When a worker returns its result the farmer inserts the result into the correct place in the result vector and sends another packet to the worker. This process continues until all work is completed. Once all work is completed the farmer sends a termination message to each worker.

The flat worker executes work packets received from the farmer until it receives a termination signal. The nested worker, however, is more complex. A work packet is received by the root process in a given sub-group. If free values are present, it broadcasts the packet to the other processors in the sub-group. The members of the sub-group, by executing their functional argument on the packet enter the nested skeleton, in the process evaluating any free values. If no free values are present no packet is broadcast and a dummy value is applied to the functional argument as a method of stepping into the nested skeleton. The new skeleton has all the processors in the sub-group as its resources.

fold et al

The *fold* skeleton is implemented using a broadcast binary divide and conquer paradigm. At start-up the root processor splits the work list into a number of sublists. It sends one sublist directly to each processor in the tree and retains one for itself to process locally. It is only after producing results that the binary tree comes into play in the implementation. Leaf nodes send their results to their parent processors which combine them with their own results and in turn send these to their parent processors. In this way the results filter back up to the root processor.

Two other forms of *fold* are also implemented, *Fold* and *SFold*. *Fold* implements a special case of *fold* where the function argument is of the following form.

$$fn\ h\ =>\ fn\ t\ =>\ f1\ (f2\ h)\ t$$

In this case associativity and commutativity is only necessary in f1. *SFold is* a special case of *Fold*. It is intended for use where the first of the two function arguments (for *Fold*) is not associative but it is possible to replace it with two functions where one of them is associative. These differing versions result in slightly different behaviour between the skeletons when actually executing the sequential h.o.f over the sublists and in how the results are combined. They do not affect the overall topology and algorithm of the skeletons.

The definition of the higher order functions that represent the three *fold* skeletons are as follows.

$$\begin{aligned} fun\ fold\ f\ b\ [] &= b \\ \mid fold\ f\ b\ (h :: t) &= f\ h\ (fold\ f\ b\ t) \\ val\ fold &= fn\ :\ ('a \rightarrow' b \rightarrow' b) \rightarrow' b \rightarrow' a\ list \rightarrow' b \end{aligned}$$

$$\begin{aligned} fun\ Fold\ f1\ f2\ b\ [] &= b \\ \mid Fold\ f1\ f2\ b\ (h :: t) &= f1\ (f2\ h)\ (Fold\ f1\ f2\ b\ t) \\ val\ Fold &= fn\ :\ ('a \rightarrow' b \rightarrow' b) \rightarrow ('c \rightarrow' a) \rightarrow' b \rightarrow' c\ list \rightarrow' b \end{aligned}$$

$$\begin{aligned} fun\ SFold\ f1\ f2\ f3\ b\ [] &= b \\ \mid SFold\ f1\ f2\ f3\ b\ (h :: t) &= f2\ (f3\ h)\ (SFold\ f1\ f2\ f3\ b\ t) \\ val\ SFold &= fn\ :\ 'a \rightarrow ('b \rightarrow' c \rightarrow' c) \rightarrow ('d \rightarrow' b) \rightarrow' c \rightarrow' d\ list \rightarrow' c \end{aligned}$$

compose

The *compose* skeleton is implemented as a pipeline. It is unrestricted in the sense that each function in the pipeline need have only the same domain type as the preceding function's range. Hamdan states that to deliver good performance it is necessary that the *compose* be nested inside another skeleton that is generating a stream of values.

2.7.2 Examples

A number of examples are presented on a variety of platforms with an impressive degree of conformity of performance across platforms. In particular, Hamdan presents the same matrix multiplication using arbitrary length numbers as ourselves. By using the special *Fold* and *SFold* skeletons the solution is implemented in four different nested ways (*map map, map SFold, SFold SFold,SFold map*). Although his results for this example are based on the older, slower, Fujitsu AP1000 and ours are on the newer AP3000, the speedup results are close to our own.

2.8 Summary

We have seen many different skeletons from the groups surveyed in this chapter. It is apparent that in spite of the differences, there are many similarities between the groups' work. The same styles of parallel computation are seen repeatedly, for instance Cole, Eden, SCL and P^3L all have divide and conquer skeletons. EKTRAN has a *fold* skeleton, implemented using a binary tree, which expresses very similar semantics. HOPP also has a *fold* skeleton although implementation details are sparse as a result of the emphasis on cost models rather than skeletons. That the same computation styles are repeated adds weight to the argument that only a fixed set of constructs are needed for parallel programming, although this is by no means conclusive.

The splitting of a parallel program conceptually into two parts is certainly a common theme, variations occurring in SCL, P^{3}L, and Eden. The distinction in Eden is more subtle as the two parts are combined in one source file. Cole, EKTRAN and HOPP take a different approach. Their skeletons are implicitly parallel, with the exception of Cole's TQ, and do not require explicit handling of parallelisation issues by the programmer. Even Cole's explicitly parallel TQ skeleton only supports a restricted form of parallelism and does not impose significant burden on the user.

The nesting of skeletons is an issue which is handled using several different techniques. In SCL they hope to cope with the issue by transforming away any nested parallelism; in P^{3}L and Eden it is the programmer's responsibility to attempt to manually nest skeletons together; EKTRAN supports automatic nesting of skeletons. HOPP supports nesting up to three levels deep but the compilation is done by hand and details of how nesting is achieved are not given. It seems implicit in Cole's system that a program contains only one skeleton but his *Cluster* skeleton with its grouping together of subproblems has much in common with nested skeletons.

Nesting is clearly a complex issue and one which, with the exception of EKTRAN is not covered in any detail, or in some cases is either left to the programmer to express or is removed. Nesting presents significant problems to anyone trying to manage it automatically. In particular, creating skeletons that are able to arbitrarily nest themselves and other skeletons, and then mapping the resultant diverse and often conflicting topologies onto a target platform is a major challenge.

In many ways we have adopted a similar approach to EKTRAN. We too choose to create skeletons that are direct replacements for higher order functions. The overriding difference between our work and EKTRAN is the manner in which we achieve nesting. Whilst EKTRAN adopts the flexibility of a dynamic runtime system of skeletons we prefer a more, although not completely, static approach based on startup time configuration.

With our work we try to find a happy medium between the flexibility of the dynamic approach and the rigidity of the static approach without paying the significant extra costs of parallelisation that are inherent in dynamic systems.

Chapter 3

Skeletons and Control

3.1 Introduction

In this chapter algorithmic skeletons are described for implementing the list functions *map* and *fold* along with a control layer, *pskel*. The skeletons refer to three conceptually different types of processor, as follows:

Global Root Processors are found at the topological root of a series of one or more, possibly nested skeletons. Each entire program has only one global root processor; this processor is fixed for the lifetime of the program.

Master Processors are found at the topological root of a skeleton; they are fixed for the duration of the skeleton. We often refer to a nested master processor, this is a processor that becomes a master upon entering a nested skeleton. All master processors other than the global root root processor are nested master processors (see below).

Child Processors are processors that have parent processors; they themselves may also have children. Like master processors, their duration is for the lifetime of the skeleton.

Note that processors may belong to one or more of these groups. The following rules describe the possibilities:

1. A global root processor is also a master processor.
2. Child processors can become nested master processors by entering a nested skeleton.
3. Master processors either remain master processors if they are a global root processor or must eventually regress to child processors by exiting a nested skeleton.

When a series of possibly nested skeletons exist in a single program, the topology and type of the processors will change as each new top level skeleton starts, with only the global root processor remaining fixed. The top level skeleton is the skeleton rooted at the base of the processor tree, that is the skeleton rooted by the global root processor. For example, a *map* skeleton might be executed using the topology shown in Fig. 3.7, followed by a *fold* skeleton using the topology shown in Fig. 3.8. Only the processor at the root of both processor trees would necessarily be the same as before - the global root. If a nested computation were performed the topology of the entire nested topology would be fixed until all skeletons in the nesting were complete. This is discussed in more detail in Section 3.5.

3.2 Overview of a Parallel Computation

Prior to giving a detailed description of the components that are combined to form a parallel computation it is useful to consider the overall system in which they each have a role. We will not look at any individual component in detail at this stage, instead we will show how they interact at run time to form a parallel computation. Figure 3.6 illustrates the execution of a parallel computation and includes all the relevant components. When studying the figure it is important to remember that the diagram is representative of the computation as a whole and not the behaviour of any one processor within it.

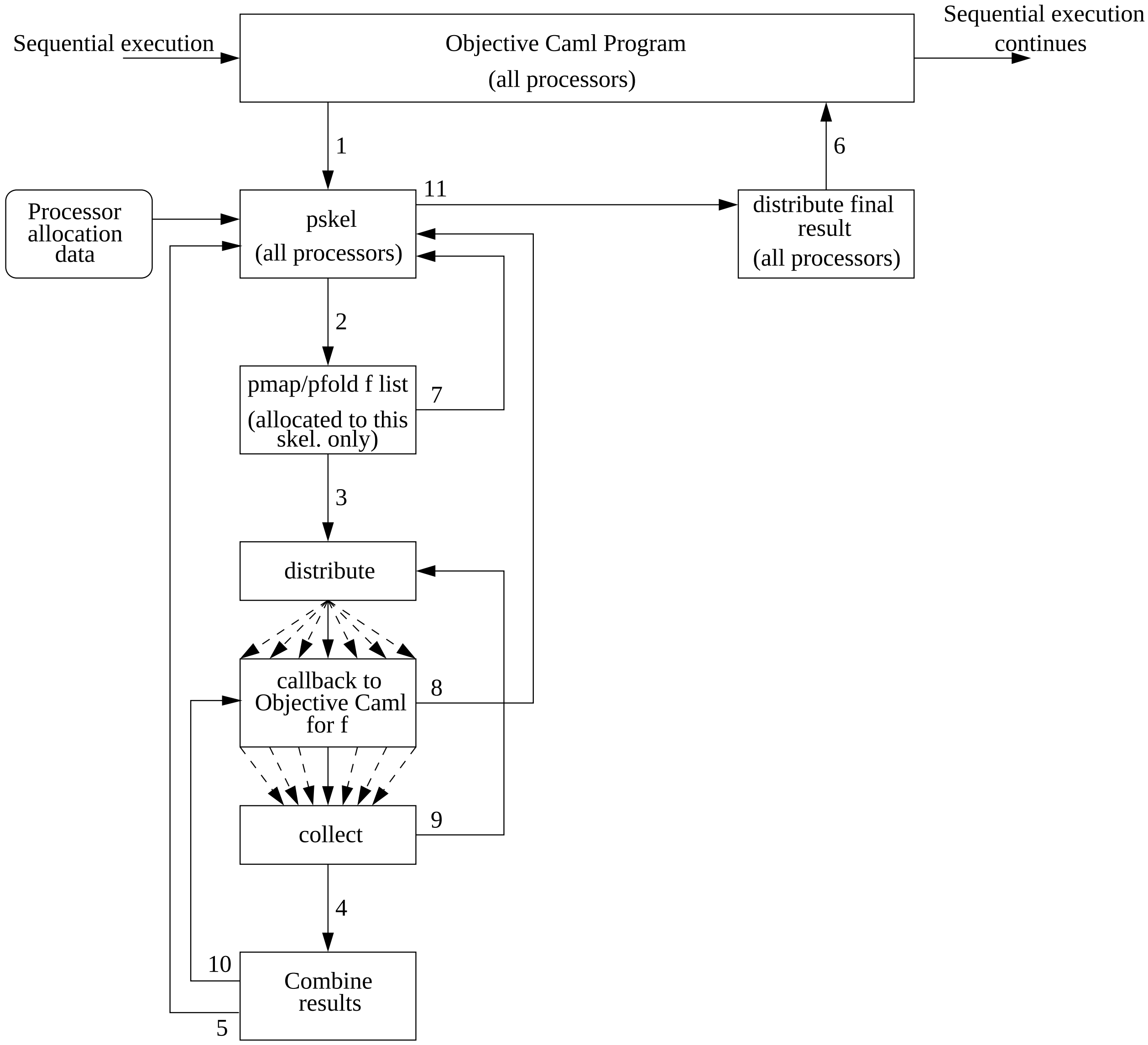

Figure 3.6: Flow of execution of parallel computation

The figure assumes the existence of processor allocation data without showing its creation. The data is created at program startup from the Skeleton Network, which in turn is created from a representation of the program at compile time. Chapter 4 describes the Skeleton Network and its use in processor allocation but for our purposes it is enough to assume its existence.

A parallel program begins by executing the same sequential program on all available processors. When a skeleton is called all processors enter the pskel layer (1), at which point the processor allocation data is used by each processor to identify its role in the skeleton or nested skeletons. Any processors which are not taking part in a skeleton immediately proceed to the routine for collecting the final result of the parallel computation (11) and take no other part in the computation. Processors allocated to any skeleton in the computation, even a nested one, call the code for this outermost skeleton (2). Those processors that are involved only in skeletons nested below this one do this so that they can call the functional argument of the skeleton and hence access the skeleton nested within it (7). This process will repeat until a processor finds itself in the skeleton in which it is to play a part.

Assuming a processor has reached the skeleton it is allocated to it will take part in the data distribution process (3). The exact mechanism is dependent on the skeleton but the result is that each worker processor gets a section of the list argument to the skeleton. The processors now work over their respective list sections, it may be that many of them by executing their functional argument to the skeleton enter a nested skeleton (8). These processors become nested masters in the nested skeletons; they use the list section that they each acquired earlier as the list argument to their nested skeleton and take their child processors from those that earlier stepped directly into the nested skeleton (7), exactly which processors they get being dictated by the processor allocation data.

The manner in which each skeleton organises itself and carries out is computation is dependent on the actual skeleton and is discussed later in this chapter. However, within each skeleton a collect process of some sort will result in its root processor having the complete results for the skeleton. Once this process is complete a nested master will return back to its original skeleton (10) and may collect more data and re-enter the nested skeleton, a process which can repeat many times. Eventually, all the skeletons will complete and their processors return to the original pskel layer (5), from here they will enter the final phase, distributing the complete result of the computation to all skeletons (11) and finally returning to the sequential Objective Caml program (6).

3.2.1 Summary and Comparison of a Parallel Computation

The technique that we have used to control the processors within our skeletons is unusual. As can be seen from Section 3.2, we do not have a single sequential program and a pool of subservient processors waiting to be called upon, as is the case with EKTRAN (Section 2.7). Instead we have adopted the approach of allowing Objective Caml to drive the system for us. This allows us to implicitly assume that all processors will eventually execute the entry code for a skeleton, or at least the code for the outermost skeleton for all processors in the nested case, and the inner skeletons for the appropriate processors in those cases. In a pool based system such behaviour would need to be explicitly handled via some mechanism, probably involving inter-processor communications and the extra costs that implies.

This is achieved by executing the sequential program on all processors until synchronisation is forced by the occurrence of a skeleton; at this point the skeleton takes over and controls the parallel computation. This technique is further utilised within the nested skeletons, with processors taking part in the nested skeletons using the Objective Caml runtime to bridge any intermediate code and connect between nested skeletons. Once a parallel computation is complete all the processors return to running their sequential Objective Caml programs until either another skeleton is encountered or the program terminates.

As noted above, this technique is very different to that used by EKTRAN. Whilst EKTRAN's pool based approach may be more adaptable within a heterogeneous system than ours, ours should have the advantage in a homogeneous system. In a homogeneous system, with processors reaching the parallel parts of a computation at approximately the same time, our approach provides a lot of control for the system that is essentially free, if at the cost of a slightly more complex implementation.

With a heterogeneous system our approach might incur some extra cost whilst waiting to achieve synchronisation at the beginning of a skeleton due to any difference in processor performance, and hence a difference in the time at which processors reach a skeleton. Even so, within heterogeneous systems and systems with differences in processor speeds our approach should not prove overly expensive given the time we save through decreased communications and the fact that the parallel parts of the computation ought to significantly outweigh the sequential ones in terms of runtime

3.3 *Map* skeleton

The list functional *map* applies a function to each element of a list, returning a list of the results [28]:

$$map\ f\ [x_1, ..., x_n] = [fx_1, ..., fx_n] \tag{3.2}$$

The function is defined in Standard ML as follows:

$$\begin{aligned} fun\ map \quad & f\,[\,] && = [\,] \\ \mid\ map \quad & f\,(x :: xs) && = (f\,x) :: map\ f\,xs \end{aligned} \tag{3.3}$$

and has type:

$$('a\ \rightarrow\ 'b)\ \rightarrow\ 'a\ list\ \rightarrow\ 'b\ list \tag{3.4}$$

Our *map* skeleton uses a process farm topology to implement *map's* functionality.

The underlying concept for a process farm can be described as follows. A farmer divides the available work into a series of equal sized packets; each worker in the farm being sent a packet. When a worker completes a packet of work it returns a result packet to the farmer and is sent another work packet. This process continues with the farmer serving and collecting packets but processing no packets itself until the supply of work is exhausted. The process farm is largely self load balancing although there is a possibility

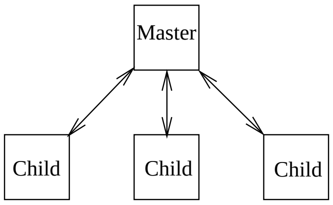

Figure 3.7: Unnested map topology with 4 processors

of swamping the farmer's communications links if the work packets are too small or the number of workers too large.

Figure 3.7 shows the processor topology of an unnested *map* skeleton. In terms of the processor taxonomy described earlier (Section 3.1), the farmer is a master processor and, in the unnested case as shown, also the global root. The workers are its child processors.

3.3.1 map - pseudo-code

- **Master Code**

```
if (haven't already mastered) then
    broadcast (free values)
if (no children) then
    seq_map fn list
else
    send (list packets)       //to children
    while (list packets remaining)
        receive (result packet) //from any child
        send (list packet)    //to child that sent result
return result                 //master has whole result
```

- **Child Code**

```
receive (free values)
```

```
while (true)

    receive (list packet or terminate message) //from master

    if packet (is NULL) then  //termination received

        terminate (children)  //if have any

  //only regressed nested masters may have children

        break (out of while loop)

     else

         seq_map fn (list packet)

         send (answer packet) //to master
```

Explanatory notes

The algorithm above shows the high level control that drives the *map* skeleton. The master processor repeatedly sends list packets to its children; the child processors execute Objective Caml's *map* over the contents (called seq_map in the psuedo-code) and return the results. The execution of the *map* may result in nesting, the handling of nesting is discussed in detail later (Section 3.5). When the master processor runs out of work packets it waits until it has all the result packets from its children and then exits returning the final result list.

3.3.2 Termination messages

Termination of the global root's child processors is carried out by the *pskel* control layer after the global root exits the *map* skeleton. Hence, in the unnested case all processors are terminated as a result of messages from the global root. In the nested case things are more complex as, logically, each processor should terminate its child processors prior to exiting the skeleton itself. However, due to the way nesting is achieved (Section 3.5) it is not possible for a nested master to know if its work within a skeleton is complete until it has exited the skeleton and regressed to a child in another skeleton. Hence the newly regressed child performs the termination; this can be seen in the pseudo-code above.

3.4 Fold skeleton

The list functional *foldl* applies a two argument function to each element of a list returning an element of a new type [28]:

$$foldl\ f\ e\ [x_1, ..., x_n] = f\ (x_n, (..., f\ (x_1,\ e)...)) \tag{3.5}$$

The function is defined in Standard ML as follows:

$$\begin{aligned} fun\ foldl \quad f\, e\, [\,] \quad &= \ e \\ \mid foldl \quad f\, e\ (x :: xs) \quad &= \ foldl\ f\ (f\ (x,\ e))\ xs \end{aligned} \tag{3.6}$$

and has type:

$$('a\ *\ 'b\ \rightarrow\ 'b)\ \rightarrow\ 'b\ \rightarrow\ 'a\ list\ \rightarrow\ 'b \tag{3.7}$$

foldl applies the function to each element of the list in a left-to-right fashion. There is a similar function, *foldr*, that applies the function right-to-left. Due to our restrictions (Section 3.4.1) of associativity and commutativity the two functions are equivalent. In fact, the type of our parallel fold skeleton is

$$('a\ *\ 'a\ \rightarrow\ 'a)\ \rightarrow\ 'a\ \rightarrow\ 'a\ list\ \rightarrow\ 'a \tag{3.8}$$

The parallel *fold* skeleton uses a divide and conquer topology (see Fig. 3.8). Unlike the *map* skeleton we have no notion of one master and several child processors per skeleton. Instead, the topology is better considered as one master processor with two children, those children in turn being parent processors to their own children. A parent processor

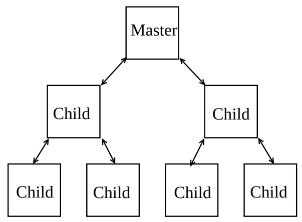

Figure 3.8: Unnested fold topology with 7 processors

is not analogous to a master processor. A master processor is the processor at the root of a skeleton whereas a parent is simply a processor within the skeleton that has child processors below itself. The distinction is only topological: in the algorithm all child processors assume that they may have child processors of their own and manage them appropriately.

To minimise communications each parent splits its list only once, a third for itself and each child. The decision to split the list equally amongst the processors was chosen to ease the implementation. The justification for this choice is that although processors closer to the root of the skeleton would receive significantly more data than those lower down, they would also have more time to process it as they are not involved in distributing the remaining data other than to pass it directly to their child processors. It may well be that this division is top heavy, and in general it may be necessary to use heuristic methods to aid effective data distribution, particularly so if the processing time for individual list elements is not uniform across the list.

In a nested skeleton, the process of data distribution may be repeated multiple times dependent on how many times the nested master re-enters. In the nested case further complications arise as processors not in the bottom-most skeleton must sequentially execute all skeletons appearing below theirs. In the pseudo-code this is referred to as the 'sequential phase'. This situation can also arise due to processor shortage (see Section 4.2). This situation does not arise with *map* as only the child processors are involved in the actual computation.

3.4.1 *Fold* restrictions

In order to ensure correctness of the final result computed by the *fold* skeleton, it is necessary that the functional argument of the *fold* skeleton be both associative and commutative. If the functional argument did not meet these requirements the final result would be incorrect and would depend on how the data was split and recombined. Specifically, if the argument were not associative we would not be able to repeatedly split the list into sections and recombine it. Our commutativity requirement allows the implementation the freedom to recombine the split list in in any order it wishes, removing the need to preserve any ordering inherent in the data. Furthermore, when the functional argument is commutative *foldl* and *foldr* are equivalent, removing the need for separate *foldl* and *foldr* skeletons and allowing the implementation to use either of these internally. A more rigorous implementation would be able to dispense with the commutativity requirement but the need for associativity would remain.

Associativity and commutativity are defined below.

- **Associativity**

$$x \oplus (y \oplus z) = (x \oplus y) \oplus z \tag{3.9}$$

- **Commutativity**

$$(x \oplus y) = (y \oplus x) \tag{3.10}$$

where $\oplus$ is some binary operator.

There is one final requirement for the *fold* skeleton. The value e, one of the skeleton's arguments, should be an identity for the function and the list type over which it is folded. That is, for some function f and any list element x the following should be true.

$$f\ x\ e = x \tag{3.11}$$

This is necessary as each parent processor takes its result and its child processors' results and folds them into one answer before returning it to its own parent processor. If e did not meet condition 3.11 then every time this folding together of sub-results occurred an extra quantity e would be folded into the subresult at that point, resulting in the final result being incorrect.

3.4.2 fold pseudo-code

- **Sequential Phase**

```
if (sequential phase) then
    seq_fold fn e list          //compute sequentially
    return result               //return sequential result
```

- **Master Code**

```
if (haven't already setup) then
    setup (packet size and free values) //master sends to all
if (no children) then
    seq_fold fn e list
else
    distribute list             //to children
    seq_fold fn e (list segment)
    collect answers             //from children
    seq_fold fn e (all results)
    if (global root processor) then
        terminate children
        return result           //only master has full result
```

- **Child Code**

```
receive (setup data)

receive (list segment)        //from parent

if segment (is NULL) then     //termination received

    terminate children

else

    if (nested master) then   //enter nested skel

        seq_fold fn e (list segment)

    else

        distribute (list segment) //to children

        seq_fold fn e ((list segment) segment)

        collect answers       //from children

        seq_fold fn e (all results)

        send answer           //to parent
```

Explanatory notes

If we are pursuing a parallel evaluation, as opposed to a sequential evaluation we firstly call the setup routine, this distributes setup data such as free values to all processors. The setup data is broadcast directly from the master to all processors in the skeleton. The processing then diverges with the master processor splitting its list argument into three approximately equal segments, one for itself and one for each child. Using the distribute list function the master processor sends a segment of the list and a copy of the value e, as defined in the *foldl* definition (Fig. 3.6), to each child; it then works over its own list segment.

After calling a routine to collect answers from its children, the master *folds* the results together with its own to produce a final result (these uses of Objective Caml's sequential fold, using *foldl* or *foldr* are indicated using seq_fold in the psuedo-code) . In this skeleton the global root processor terminates its children rather than leaving it to *pskel*, these

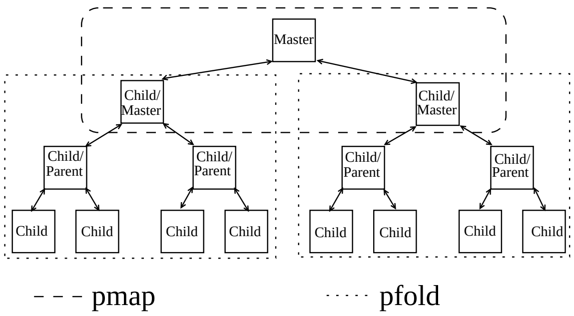

Figure 3.9: Topology for *map* with *fold* nested below it, 15 processors

children in turn then terminate their children. This is an artifact of the development process, we could equally well leave the termination to *pskel*.

The child processors behave in a similar manner although they must first receive a list segment from their parent processor. Note that prior to becoming involved in the parallel computation, a child must first check if it is to become a nested master . If the child is to become a nested master it must execute the outer skeleton sequentially, hence stepping into the skeleton where it will be a nested master. This information is available in the Skeleton Network (Section 4.1). As with *map*, a nested master terminates its children when it has regressed to being a child processor.

3.5 Skeleton Control and Nesting

If we wish only to parallelise one h.o.f in any given program then there is little need for skeleton management. In parallelising more than one h.o.f however, some of which may be nested, management becomes necessary. The major functions that the control layer, *pskel*, performs are those of providing an homogeneous layer between skeletons, initialising and starting the correct skeletons and managing the state of processors. Figure 3.9 shows a typical topology for a *map* skeleton with a *fold* nested below it, that might be managed by this scheme. The code snippet in Section 1.1 illustrates SML that might lead to this topology.

3.5.1 Skeleton Initialisation

The initialisation phase builds data structures for each processor, describing its parent processor, child processors and the structure of the skeletons within the program to be executed (Section 4.1). It also sets pointers to functions native to Objective Caml so that they can be used by the C code, a process known as registering. These functions are the functional arguments to the h.o.fs that the skeletons replace. Additionally any utility functions the skeletons may need to use, such as *map* and *fold*, are also registered. The registration process takes strings representing the functions and returns pointers to function closures.

The actual algorithm controlling the initialisation works as follows. When a processor first enters a skeleton, it checks if it is initialised or whether it has ever entered a skeleton before. If the processor has not previously ever entered a skeleton, the current skeleton must be the first instance of a top level skeleton in the program execution that has been encountered. The processor must then build the skeleton network (Section 4.1) and create the processor allocation (Section 4.2) data before continuing with the execution of the skeleton. If a processor is not initialised then it is entering a new top level skeleton and must collect its allocation data and check that it has been allocated to the current skeleton. If a processor is unallocated it waits for the skeleton to complete and collects a copy of the final result.

Initialisation pseudo-code

- **All Processors**

```
if (not initialised) then
    if (not ever_entered) then
        build (skeleton network)
        allocate processors
```

```
        get (registered functions) //Get O'Caml functions
    get (allocation data)          //get root, children etc
    if (unallocated) then          //proc not involved in skel
        wait (for final result)
else
    get (allocation data)
get (list of children)
```

3.5.2 Skeleton Control

Control of the entire system of skeletons filters down from the global root along with the names of functions to be executed. When the global root enters a skeleton it checks that it is the global root entry point (Section 4.2); if it is not then the skeleton will be executed sequentially. Assuming the global root has found the global root entry point, it will transmit the name of the function argument to the h.o.f, to the child processors in its skeleton. It is necessary to transmit function names so that each processor can execute the appropriate skeleton when a program contains more than one skeleton. It is also necessary that the function arguments have unique names so that a skeleton can be uniquely identified, any renaming necessary to ensure this is handled by the compiler.

When a child processor first enters a skeleton it waits to receive its function name. Once it receives the function name it is able to identify its position in any combination of skeletons in the skeleton network and, hence, acquire the processor allocation data for that skeleton. If it is involved in an unnested skeleton, or the outermost level of a nested skeleton, it will wait to receive data from its parent and will then continue with the execution of the skeleton. If the processor is involved in an inner skeleton of a nesting it must step through the outer layers until it reaches the appropriate level. It achieves this by sequentially executing the h.o.f corresponding to the outermost skeleton, causing it to step into the nested skeleton. This process is repeated at each nesting level until the processor reaches the skeleton it is a part of, at which point it will wait to receive data from its parent and will continue with the execution of the skeleton.

A processor can be allocated to two skeletons, as a child in the first skeleton and a master in the second. The second skeleton must be at a nesting level immediately below the first. The child receives data, as described above, in the first skeleton and executes the appropriate h.o.f entering into the second skeleton. Here it becomes a master processor and distributes data and collects results, with its exact behaviour being dependent upon the type of the skeleton in which it is involved.

Due to the fact that the h.o.fs these skeletons are based upon repeatedly apply a function to elements of a list, a processor may become a nested master in the same skeleton several times. Hence, we require state variables (not shown here) to ensure that free values and function names are sent only once. Note that a child processor only enters the skeleton in which it is a child once and hence only collects a function name and free values once. Finally we broadcast the total result to all processors in the system, this may be switched off with appropriate compiler flags (Appendix A). Also worth noting is the global termination message; this is used by a parent processor to signal to a child that there will be no more work in the current skeleton and that it should exit.

Control pseudo-code

- **Global Root Code**

```
if (global root entry point) then
    send (function name to all children) //Start the whole skel
    execute skeleton in parallel
else
    execute skeleton sequentially //Enter nested skel
```

```
        if (have mastered) then
            terminate children
```

- **Child Code**

```
start (non-blocking receive of global termination message)
wait (for termination message or function name)
if (received function name) then
    execute skeleton in parallel
else (received termination message)
    terminate children
```

- **Nested Master Code**

```
if (have children and am root of a skeleton) then
    send (function name to all children)
    execute skeleton in parallel
else
    execute skeleton sequentially
```

- **All Processors**

```
update skeleton network
if (done top level skeleton) then
    broadcast (total result)  //Copy result to all
    return (total result)     //All return result
```

Explanatory notes

The code segments are shown in order of execution, for example, the global root processor will execute its code section and then the all section. In the case of a child processor becoming a nested master processor, the processor will execute the all section each time it becomes a nested master and once as a child. The update of the skeleton network simply moves a pointer along to maintain the current position in the network, this helps to speed up the searching of the skeleton network the next time a skeleton is encountered. Processors will only become involved in the broadcast and return of the final results to a skeleton computation as either a global root processor, a child processor, or an unallocated processor, as nested masters can only exist in nested skeletons and not in a top level skeleton. This behaviour is correct as the intention of this broadcast is to place a copy of the final result on all processors as they exit a top level skeleton.

3.6 Alternative Strategies

If we wish to compare the strategy described in this chapter with those of other researchers there are two overriding issues that dominate. Firstly, is nested parallelism supported at runtime or is it flattened in some way? Secondly, is the processor allocation static or dynamic? Of lesser importance is the question of groups. We have chosen not to incur any overhead associated with utilising the notion of groups in MPI, the C library used to implement the parallel aspects of the skeletons. Other researchers, such as Hamdan, have used groups. Although the use of groups may incur some overhead, Hamdan creates groups only when skeletons are created and incurs no obvious extra costs.

Most works have limited, or allow only for manual, support of nested parallelism. As we observed in Chapter 2, Cole does not support nesting, at Imperial they hope to handle nesting by transforming it away, and HOPP does not give details of how nesting is achieved. In P^3L and Eden the user must explicitly craft the parallelism themselves. Details of processor allocation mechanisms are not given for Eden but it seems implicit that there is a one-to-one mapping between processes and processors, with the programmer being responsible for producing an appropriate algorithm. Processor allocation in P^3L is a complex issue and a significant stage of the compiler during which a given program is mapped onto a specific platform in a static manner. In Hamdan's EKTRAN nested parallelism is supported dynamically with processors being allocated at runtime. Our system constructs the processor topologies necessary to support nesting at startup time based on weights (Section 4.2).

The motivation for our hybrid system is two fold. Firstly, we wished to avoid the costs associated with dynamic parallelism. Secondly, we thought it desirable to be able to run the same program on differing numbers of processors without the need for recompilation. To enable this our skeletons had to be very flexible, even being able to run sequentially on a single processor (Section 4.2.6).

3.7 Summary

In this chapter we have described the strategies used to implement our skeletons. We have shown the complexity of the system required to combine topologically diverse skeletons within an arbitrarily deeply nested framework at runtime in a robust fashion, an achievement unique to this work. We have also shown that EKTRAN is the most closely related to our own and have made comparisons between the two.

In addition to their novel ability noted above our algorithms also certainly meet our goals of avoiding dynamic parallelism and maintaining the flexibility to run on differing numbers of processors. In the following chapter we describe how our allocation algorithm deals with these orthogonal issues.

Chapter 4

Skeleton Network and Processor Allocation

4.1 The Skeleton Network

The compiler, written by Norman Scaife[24], uses an extension to the type scheme, called an *Abstract Network*, to represent the topology of skeleton occurrences within a program and associated information for profiling purposes. For processor allocation and runtime coordination we need to know the skeleton topology and the weights assigned (See Appendix A) by the profiling system. We do not need to know all the information that is contained within the Abstract Network.

The skeletons use a trimmed down Abstract Network called a *Skeleton Network*. This has two constructs, a *NODE* and a *SEQ*. A SEQ contains NODEs that are to be executed sequentially. All NODEs contain skeleton occurrences and may also contain NODEs or SEQs. We impose the condition that all skeletons in the translated Objective Caml must have unique string identifiers. With this condition and our two constructs we can represent almost all combinations of skeletons being nested within and occurring after each other. Due to its static nature, the compiler cannot parallelise h.o.fs within alternative constructs

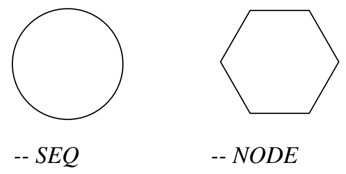

Figure 4.10: Key for Figure 4.11

that may result in differing h.o.fs being executed dependent on runtime data values.

To translate from the Abstract Network to the Skeleton Network we use a Standard ML program that examines the Abstract Network whilst the program to be parallelised is being compiled. This produces a C function that is compiled and linked at the same time. This takes the form of a series of nested function calls to library functions that recursively build the linked data structure we call the Skeleton Network. This approach not only provides a simplified network data structure, it also translates it into C saving run time costs associated with accessing a Standard ML data structure from C.

4.1.1 Skeleton Network examples

Consider again the example in Fig. **??**. This is a *map* with a *fold* nested below it. For convenience we repeat the relevant code snippet from Section 1.1 here.

$$map\ (fold\ op+\ \ 0)$$

This can be represented by the Skeleton Network in Fig. 4.11, where a and b are the nodes representing the *map* and *fold* respectively. In this example, were it not for the Skeleton Network, the child processors in the nested *fold* skeletons would not know that they were part of a *fold* skeleton, having entered pskel via the top level call to *map* (see Section 3.5). In general, a child processor in any nested combination of skeletons would not know which skeleton it was a part of.

By augmenting the Standard ML for the above example we can produce the following more complex example (line numbers are for reference purposes only).

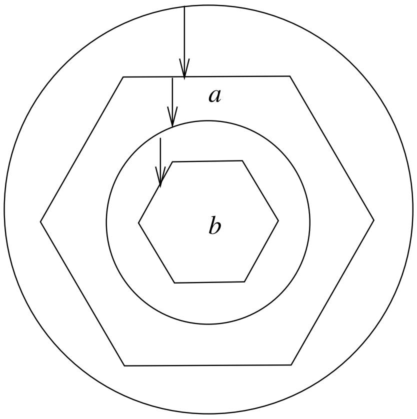

Figure 4.11: Skeleton Network example for Section **??**

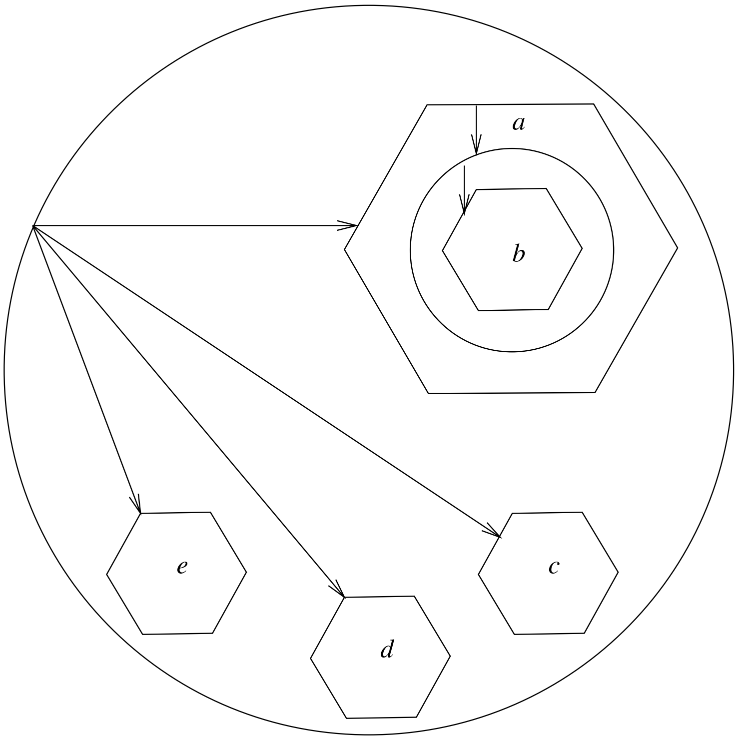

Figure 4.12: Skeleton Network example for Section 4.1.1

```
1 val x =
2   foldl (fn (a,b) => map (fn x => x)(a@b)) [] [[1,2,3],[4,5],[6]]
3 val y =
4   map (fn a => ~a) x
5 in
6   foldl op+ 0 (map (fn b => ~b) y)
7 end
```

The *Skeleton Network* is shown in Fig. 4.12. If we label as follows a = *foldl* on line 2, b = *map* on line 2, c = *map* on line 4, d = *map* on line 6, e = *foldl* on line 6, we have a NODE evaluation order (b, a, c, d, e) with b being nested inside a, and the resultant combination being sequenced with c, d, and e.

4.2 Processor Allocation

The Processor Allocation routines have to create the processor topologies when a parallel program begins execution. A strong motivation for startup time allocation, rather than compile time, was to avoid the tedious exercise or recompiling every time we might wish to run on a differing number of processors. Hence the routines are relatively simplistic, to avoid, for example, a long heuristic search that might take longer than the actual parallel computation!

4.2.1 Transforming between topologies

If we consider the Skeleton Network to represent the interconnection topology of the various skeleton instances within a program, we can think of the process of processor allocation as being one of transforming from that topology to another. That is, the allocation involves automatically merging the topologies of differing skeleton types and allocating processors to individual skeletons based on weights supplied by the compiler's profiler

(Currently a set of dummy weights are used but this can be disabled (see Appendix A)).

To convert from the Skeleton Network into a processor network is a relatively complex step. Considering again our much used example (Fig. **??**), we see a *map* skeleton with a *fold* nested below it. There are in fact two separate instances of the same *fold* skeleton. If, instead of the three in the figure, we had allocated a larger number of processors to the *map* skeleton there would be more instances of the *fold* skeleton. Taking into account the fact that we may also vary the number of processors within the *fold* skeleton, the complexity of the problem starts to become apparent. To decrease the complexity of processor allocation, making run time allocation practical, we introduce a number of simplifying restrictions:

1. All *fold* skeletons are allocated in a symmetric fashion. That is, all levels in the binary tree of processors are fully populated.

2. Every instance of the same nested skeleton has the same number of processors.

3. Each skeleton type has a minimum number of processors associated with it. To be parallelised, there must be enough processors for every instance of a skeleton to have the minimum number of processors allocated to it. Currently, the minimum number of processors is set to be three for both *map* and *fold*.

4. Where the minimum number of processors is not available, all available processors will instead revert to any parent skeleton. Note as an exception to this condition, if there were not enough processors available to parallelise a top level skeleton, we would still allocate one processor to it and run the entire program sequentially.

5. Any spare processors that cannot be used by a skeleton due to rules 1 and 2 will be passed to child skeletons where applicable or else not used.

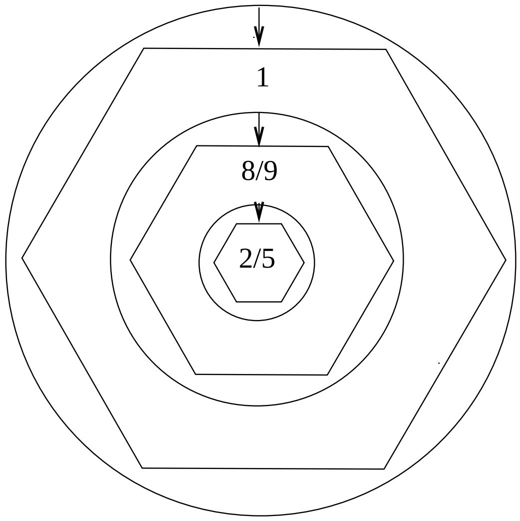

Figure 4.13: A Skeleton Network showing weights for nested NODEs

4.2.2 Weighting Skeletons

To guide the allocation process, weights are used to assign processors to skeletons. The weights represent the proportion of available resources that should be allocated to any given skeleton or series of skeletons within a SEQ. Each weight is a number between zero and one. Note that this definition implicitly means that the weight will include resources for any nested skeletons. Fig. 4.13 shows an example of this. Whilst this system of weights may at first seem a little confusing, as one might expect the weights to sum to one, it more accurately reflects the actual run-times of the skeletons.

Each skeleton, encapsulated within a NODE, has its own associated weight. Even so, when there is more than one NODE in a SEQ we use only the heaviest weight. The reason for this is straightforward: a fixed number of processors must be allocated for the duration of the entire SEQ construct. Also a SEQ may itself be contained within a NODE, representing a series of skeletons nested inside another. Any skeleton with other skeletons nested inside it will not be able to change its processor topology at the whim of its child skeletons. Hence, we allocate resources based on the most demanding child and also allocate the same resources to all children rather than waste them.

4.2.3 Allocation Algorithm

The actual algorithm used recursively traverses the Skeleton Network and creates allocation data for every processor each time it finds a new top level skeleton. Section 4.2.5 shows results for two different numbers of available processors for the topology illustrated in Fig. 4.13.

```
child_procs = available_procs * Max (child_weights)
my_procs = available_procs - child_procs
if my_procs < min_skel_procs then
    my_procs = available_procs
    child_procs = 0
else
 child_procs_per_skel = floor (child_procs / instance_count)
```

Explanatory notes

The available processors (`available_procs`) is either, at the top level, the total number of processors in the system or, in the case of a nested skeleton, the number of processors allocated to it by the parent skeleton in which the current skeleton is nested. The number of child processors (`child_procs`) allocated must be shared between all the instances of a nested skeleton (`child_procs_per_skel`). Any excess processors from the division of resources are abandoned. We cannot use the wasted processor by simply allocating it to the parent as we would now require an extra instance of the nested skeleton. Also, remember that our simplifications do not allow having differing numbers of processors in the nested skeletons.

4.2.4 Processor Data

Whilst allocating processors to skeletons configuration data for each processor is created. This effectively describes the processor topologies we see in the various examples (Figs. 3.7, 3.8, **??**). The data for each processor is a record containing the following, with respect to the skeleton the processor is actually involved in.

Root processor A root processor is any processor at the base of a skeleton's processor tree. Figure **??** has three root processors, one in the *map* and one in each of the two instances of the *fold* skeleton.

Parent processor The parent processor for a *map* skeleton is always the same as the root, however in a *fold* it may not be. For example, the leaf processors of the *fold* skeletons in figure **??** have parent processors that are not the root of the skeletons.

Number of child processors The number of direct descendants a processor has in the given topology.

first child processor A processor's first child processor is its leftmost child in the topology: this corresponds to the child with the lowest processor identification number (pid). The children within a level are numbered sequentially.

Is the processor at a nesting level? A boolean flag stating if a processor is nested (again as in the roots of the *folds* in the example). Used internally for coordination by *fold* to mark the difference between a nested skeleton and child processors within the same skeleton (we cannot assume a processor is a nested master because it has children).

Number of processors in the skeleton The total count of processors in the skeleton.

Explanatory notes

There is a processor allocation table for each occurrence of a top level skeleton. When a processor is involved in a nested skeleton, all data except the parent processor are in the context of the nested skeleton. A processor need know only its parent processor in the skeleton in which it is a child. This will enable it to collect work, enter the nested skeleton and return the results back to its parent.

All processors have a copy of the entire processor allocation table, they calculate it locally rather than distribute it at startup. Having a copy of all processors' allocation data allows the root processor to easily generate a list of all processors in the skeleton. This is necessary as, at the beginning of a skeleton, *pskel* sends a function name from the root processor in that skeleton to all other processors in the skeleton. For a *map* skeleton this is simple as we have the first child's pid and the number of children (pids being numbered sequentially within a level). However the root of a *fold* skeleton will have only two children but the skeleton may well contain many more processors. Calculating the pids for these is not straightforward in the nested case.

4.2.5 Allocation Variation

Depending on the number of processors available the degree to which the skeleton weights are adhered to varies. The following two examples show how the results would vary depending on available processors for the skeleton network in Fig. 4.13. For the purpose of these examples we assume that the NODEs represent a *map* nested inside a *fold*, nested inside a *map* skeleton. We show the resulting processor topologies and explain the allocation decisions made.

Fig. 4.14 shows the resultant topology for nine processors. As can be seen, all nine processors have been allocated to the top level *map* skeleton. This is a result of the middle *fold* requesting too many processors. The *fold* has weight $8/9$, and $floor(9 * 8/9)$ is eight. If the middle level *fold* took eight processors there would be only one left for the top level *map*. Hence, as per the simplifying restrictions, the top level *map* keeps all the

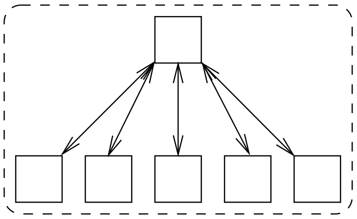

Figure 4.14: Processor topology for nine processors

processors.

Fig. 4.15 shows the resultant topology for twenty-two processors. The steps in the allocation are as follows:

1. The top level *map* starts with all twenty-two processors.

2. The middle level *fold* requests $floor(22 * 8/9)$, i.e nineteen processors.

3. Taking into account the shared processors between the top and middle skeletons, the middle *fold* now has twenty-one processors.

4. The bottom *map* requests $floor(21 * 2/5)$, i.e. eight processors.

5. Taking into account shared processors, the middle *fold* now has thirteen processors and the bottom *map* has fourteen. Note that the middle level *fold* needs to be left with a minimum of nine processors to grant any request for processors by the bottom map. That is, it needs to retain at least three processors for each of its instances, this is the minimum number of processors required by our system to parallelise a skeleton.

6. Due to its topology, the *fold* skeleton is unable to use more than nine processors so it passes the four spare processors onto the bottom *map*.

7. The bottom *map* now has eighteen processors, this is the minimum it requires to have three processors for each of the six instances of its skeleton.

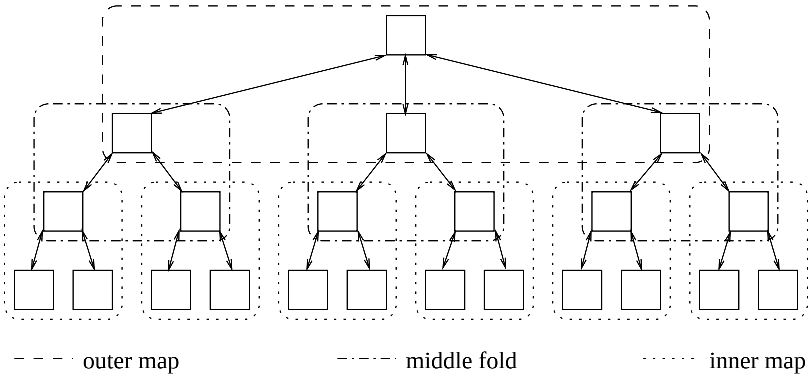

Figure 4.15: Processor topology for twenty-two processors

4.2.6 Sequential Execution of Skeletons

During processor allocation, if there are insufficient processors to parallelise a skeleton we make the decision to execute it sequentially. We cannot easily remove the sequential execution of skeletons at runtime as it would require rewriting the translated Objective Caml to replace the skeleton calls with Objective Caml's native h.o.fs and recompiling the program. Even at compile time, if we had chosen compile time processor allocation, it would be complex and undesirable to rewrite the program and re-run large parts of the compiler's analysis based on the target number of processors.

The sequential execution of skeletons adds a little extra complexity to the coordination process. Specifically, at execution time we need to be able to identify when we have a skeleton to be executed sequentially and to do so without preventing parallel execution of later skeletons.

4.3 Summary

The allocation algorithms described in this chapter do have some limitations. All but one of these is due to the necessity of having a simple, relatively quick algorithm. For example, the need to have a fully populated binary tree in a *fold* topology, and occasionally leaving a processor unused in nested topologies, are results of the simple algorithm employed. EKTRAN also uses a simple allocation system: whenever nesting occurs, and as a result

MPI groups are split at runtime, each new group is given a number of processors equal to the square root of the number of processors in the original group. This can lead to unbalanced groups. EKTRAN allows uneven groups, so no processors are wasted but load balancing may be affected.

It should be possible to generate an optimal, or at least close to optimal, processor topology for all situations. The algorithm required to do so would certainly be complex and might even be a worthy candidate for parallelisation itself! Certainly, such an algorithm would would need to be part of the compilation stage, as with the P^3L system, rather than part of a parallel program, and would hence defeat our aim of being able to run code on differing numbers of processors without recompilation.

The major limitation of the system not directly related to speed of processor allocation is that of handling alternative constructs. When a program contains a conditional statement with different skeleton instances in its branches it becomes very difficult to parallelise. If the various skeleton instances contain the same skeleton type then the problem is difficult but not insurmountable. If the instances contain more than one type of skeleton it is hard to see how it would be possible to generate an appropriate topology with anything but a dynamic approach.

In conclusion, the technique we have adopted gives us a trade off between the lower overheads of static behaviour and the flexibility of dynamic behaviour. Without changing the balance between the two, the only immediate improvement that could be implemented is to allow unbalanced topologies and hence prevent the occurrence of unused processors. The skeletons are certainly capable of supporting this. Currently, it is only the simple allocation algorithm that does not.

Chapter 5

Examples and Results

5.1 Matrix multiplication - a nested example

Multiplication of matrices containing arbitrarily long numbers was implemented in SML by Michaelson [24]. The numbers are represented by integer lists. An integer with n digits (dn)(dn-1)...(d1)(d0) is held as a list [d0,d1,...,dn-1,dn]. The sequential program was used as the basis for this parallel example.

5.1.1 SML to Objective Caml

Two *maps*, one nested inside the other, were parallelised. Here we show only the code for those two functions and the calls to *map*.

The SML

```
    //The nested fun
fun inner row cols = (dotprod row) cols
    //map over nested fun
fun rowmult'' row cols = map (inner row) cols
    //The top level fun
```

```
fun outer cols x = rowmult'' x cols

    //map over the top level fun

fun rowsmult'' rows cols = map (outer cols) rows
```

The compiler translates this to Objective Caml, in the process substituting the two *maps* with calls to the parallel *map* skeleton *pmap*. At the time of writing, the entirety of this phase was not yet complete as it is dependent on the free value analysis which is still being developed as part of the compiler spine. Currently, we must complete free value registration by hand. This involves registering any free values under a label that is the name of a skeleton's h.o.f argument with "fvs" appended. Also, we must prepend an extra argument to the skeleton's argument list so that it can accept the free value (e.g. `row` for the function `inner`. When there is more than one free value we use a tuple.

The Objective Caml

```
let rec inner = function row ->

    (function cols -> (function (row, cols) ->

        (dotprod  row)  cols) (row,  cols))


    //Register fun call instance under a unique name

let rec ff1 = function (row) ->  inner  row

let _ =  Callback.register "ff1" (ff1)


let rec rowmult'' = function row -> (function cols ->

    (function (row, cols) ->

        (function  _ ->

                //Inner map (with type info)

            (pmap :
```

```
            (string->(int list list list->int list list))
            ("ff1") cols)
             //Register free values (prior to calling pmap)
        (Callback.register "ff1_fvs" ((row))))
    (row,  cols))


let rec outer = function cols -> (function x ->
    (function (cols, x) ->  rowmult''  x  cols) (cols,  x))


        //Register fun call instance under a unique name
let rec ff2 = function (cols) ->  outer  cols
let _ =  Callback.register "ff2" (ff2)


let rec rowsmult'' = function rows -> (function cols -> (function
    (rows, cols) ->
        (function _ ->
                 //Outer map (with type info)
            (pmap :
            (string->(int list list list->int list list list))
            ("ff2") rows)
                //Register free values (prior to calling pmap)
        (Callback.register "ff2_fvs" ((cols))))
    (rows,  cols))
```

5.1.2 The Skeleton Network

The skeleton network represents the nesting of the two *pmaps*. Here are the C function calls that build the network.

```
create_seq ("%np",
  create_node ("ff2", 1.0, "%n%sp", "map" ,
    create_seq ("%np",
      create_node ("ff1", 1.0, "%n", "map" ))));
```

The outer `create_seq`'s first argument tells it to expect a pointer to a *NODE* as its second argument, by adding further occurences of `%np`s we can add more *NODE*s. The outer `create_node`'s `%n` states that its second argument is an instance of a *NODE*, similarly the `%sp` states that the subsequent argument is a pointer to a *SEQ*.

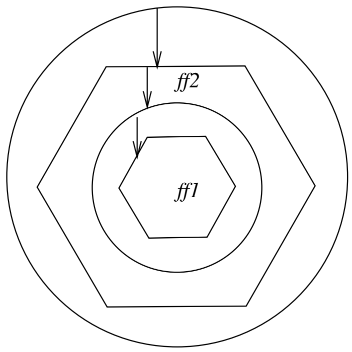

Figure 5.16: Skeleton Network for matrix multiplication

5.1.3 Processor Allocation data

For the purposes of showing a set of typical Processor Allocation data we will assume a run on seven processors, table 5.1 shows the data for each processor. We use -1 as a sentinel value to indicate that a processor has no parent (in the case of the global root processor) or no children. All data in **bold** is in the context of the outer skeleton, the rest being for the inner (with the exception of a processor's i.d., which has no context). This topology described by this data is shown in Fig. 5.17.

5.1.4 Results

The example was run on the Fujitsu AP3000 at Imperial College's Parallel Computing Centre. Tables 5.2 and 5.3 show program execution times for varying number length,

i.d.	*root*	*no. procs*	*parent*	*no. children*	*first child*	*nest*
0	**0**	**3**	**-1**	**2**	**1**	**true**
1	**0**	3	**0**	2	3	**true**
2	**0**	3	**0**	2	5	**true**
3	1	3	1	0	-1	false
4	1	3	1	0	-1	false
5	2	3	2	0	-1	false
6	2	3	2	0	-1	false

Table 5.1: Processor Allocation data for seven processors

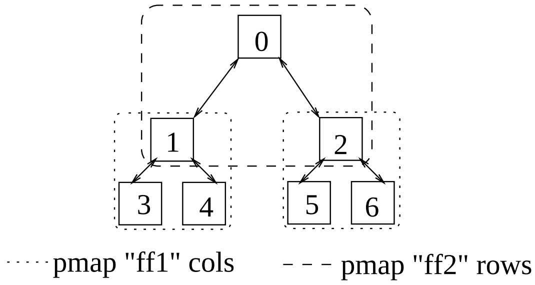

Figure 5.17: Nested topology for matrix multiplication, 7 processors

matrix size and number of processors. The numbers in **bold** are results from unnested runs and are provided for comparison purposes. For unnested runs only the outer map was parallelised, with all available processors being allocated to it. The topology is similar to that shown in figure 3.7 but with the actual number of child processors varying. Note, due to the symmetric nature of the processor allocation on nested runs with eight processors one processor is wasted, on twenty-four processors two are wasted.

There appears to be no direct benefit from nesting in this example. This is not surprising if we inspect the SML code snippet for the example. The outer *map* is simply concerned with supplying individual rows to the inner *map* so that it in turn can apply each row to each column. With the exception of a couple of anomalies with 40 digit numbers for 40x40 matrices and 30 digit numbers for 50x50 matrices that can probably be attributed as artifacts of specific test runs, the results are consistent and pleasing.

The runtime and speedup curves for the unnested and nested examples exhibit exactly the behaviour we would expect. That is, the nested case is always slightly slower due to the extra cost it expends in maintaining a nested skeletal topology without gaining any

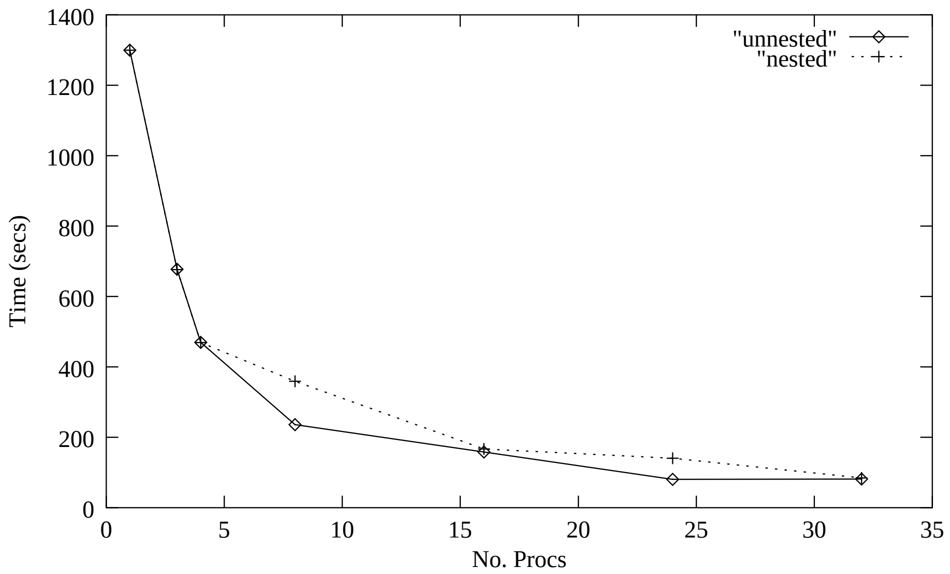

Figure 5.18: Execution times, number length 60, matrix size 50x50

benefit. The two exceptions to this case, where there are significant drops in performance, are with the results for eight and twenty-four processors. This can be largely attributed to a processor being discarded with these examples due to the simple symmetric processor allocation we have employed.

Consider figure 5.17 showing the processor allocation that we would employ for seven or eight processors (seven plus one discarded). Remember that the farmer in a *map* skeleton does no actual useful work, its role being to distribute tasks to the workers. Clearly, as a result of the two instances of the nested skeleton, only four of the seven processors are actually doing useful work in the example, this explains the sudden drop in performance when we move from four to eight processors. Note that with less than eight processors, although the skeletons themselves are nested the topology will not be. The ray tracing example (Section 5.2) describes an example of this in more detail. That the cost of nesting seems less significant on numbers of processors greater than eight (the twenty-four processor run excepted) is probably due to the ratio of farmers to worker processors decreasing as the number of workers in the nested skeletons increases.

no. procs	*number length*	*matrix size*	*time (secs)*
1	30	30x30	70.05
		40x40	166.58
		50x50	326.01
	40	30x30	124.50
		40x40	296.05
		50x50	578.87
	50	30x30	194.04
		40x40	460.85
		50x50	902.23
	60	50x50	1299.39
		50x50	**1299.53**
3	30	30x30	35.27
		40x40	87.93
		50x50	195.06
	40	30x30	62.65
		40x40	168.66
		50x50	301.81
	50	30x30	98.08
		40x40	242.71
		50x50	470.16
	60	50x50	676.40
		50x50	**677.49**
4	30	30x30	28.29
		40x40	62.89
		50x50	140.80
	40	30x30	51.73
		40x40	124.50
		50x50	209.16
	50	30x30	77.96
		40x40	173.58
		50x50	326.19
	60	50x50	468.93
		50x50	**469.68**

Table 5.2: Matrix multiplication example - 1 to 4 processors

no. procs	*number length*	*matrix size*	*time (secs)*
8	30	30x30	18.59
		40x40	47.32
		50x50	116.08
	40	30x30	33.38
		40x40	96.57
		50x50	160.71
	50	30x30	50.76
		40x40	130.28
		50x50	249.53
	60	50x50	359.03
		50x50	**235.84**
16	30	30x30	7.68
		40x40	20.52
		50x50	55.52
	40	30x30	13.29
		40x40	49.07
		50x50	74.47
	50	30x30	20.53
		40x40	56.25
		50x50	116.07
	60	50x50	166.81
		50x50	**158.02**
24	60	50x50	140.54
		50x50	**80.35**
32	60	50x50	84.42
		50x50	**81.17**

Table 5.3: Matrix multiplication example - 8 to 32 processors

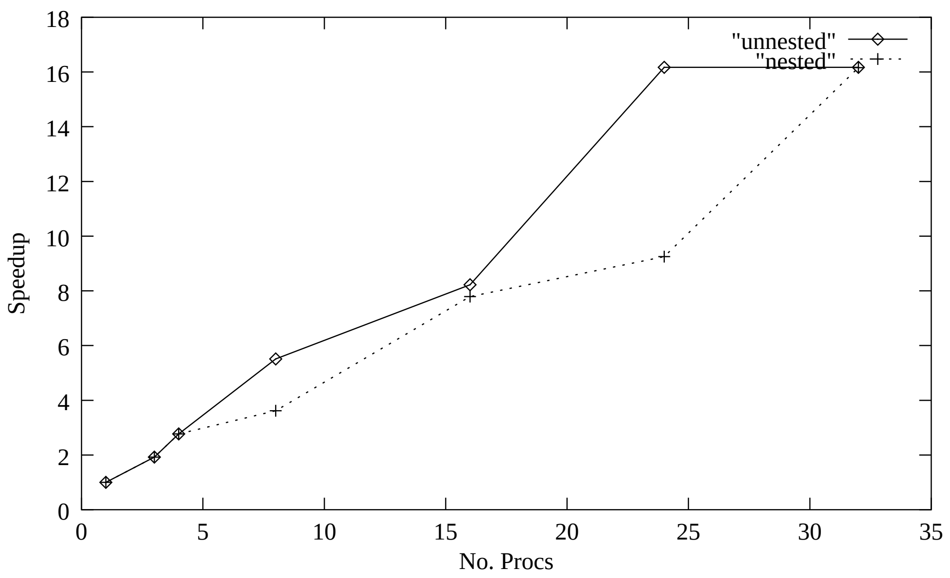

Figure 5.19: Speedup, number length 60, matrix size 50x50

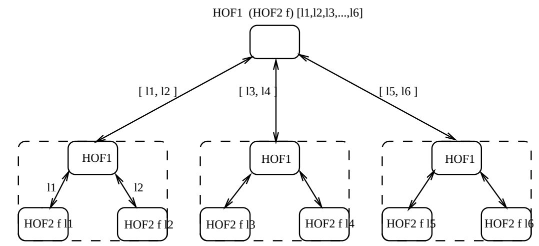

Figure 5.20: A nested topology for nested *map* skeletons

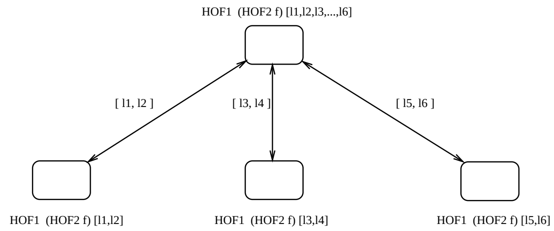

Figure 5.21: An unnested topology for nested *map* skeletons

5.2 Ray tracer

A simple sequential ray tracer program was parallelised by selecting likely h.o.fs by manually inspecting the code and replacing with parallel implementations. A pair of nested *maps* were identified and were replaced with nested calls to the *map* skeleton. Further consideration of the code showed that there was no inherent work between the outer and inner *maps*. This provided an opportunity to demonstrate the flexibility of our skeletons: the solution was implemented using nested skeletons on an unnested processor topology. The differences between a nested topology and the unnested topology we used can be seen more clearly in figures 5.20 and 5.21.

5.2.1 Results

The results for the example are much as might be expected for any implementation of a processor farm given a regular data parallel problem. The speedup figures, whilst not astounding, are certainly respectable, especially so given the generic solution of the problem. It is worth comparing the results from the AP1000 and AP3000 runs. Although these were executed on very different image sizes, this being necessary to the difference in processing power on the two platforms, the results are consistent.

The speedup figures are not quite as good for the AP3000 as for the AP1000 but this is not surprising as it is generally more difficult to achieve good speedup on data parallel problems on machines with faster processors. This is due to the considerably larger amount of work required at each processor to keep it busy. Clearly, more data implies larger communication costs and hence a rise in the overall cost of parallelisation. This statement assumes that as processing power of parallel machines has increased, the increase in throughput of communications hardware has not kept pace. Given that processing power, broadly speaking, still obeys Moore's Law and communications hardware never has, this is indeed the case.

no. procs	*Image size*	*time (secs)*
1	100	34.03
2		40.08
4		15.89
8		9.18
16		7.23
32		5.59
64		5.26
129		5.08
1	200	136.64
2		162.39
4		81.74
8		37.24
16		26.97
32		22.41
64		20.86
129		20.24

Table 5.4: Raytracer AP1000 results

no. procs	*Image size*	*time (secs)*
1	1000	149.17
2		149.16
4		70.37
8		41.66
16		38.73

Table 5.5: Ray tracerAP3000 results

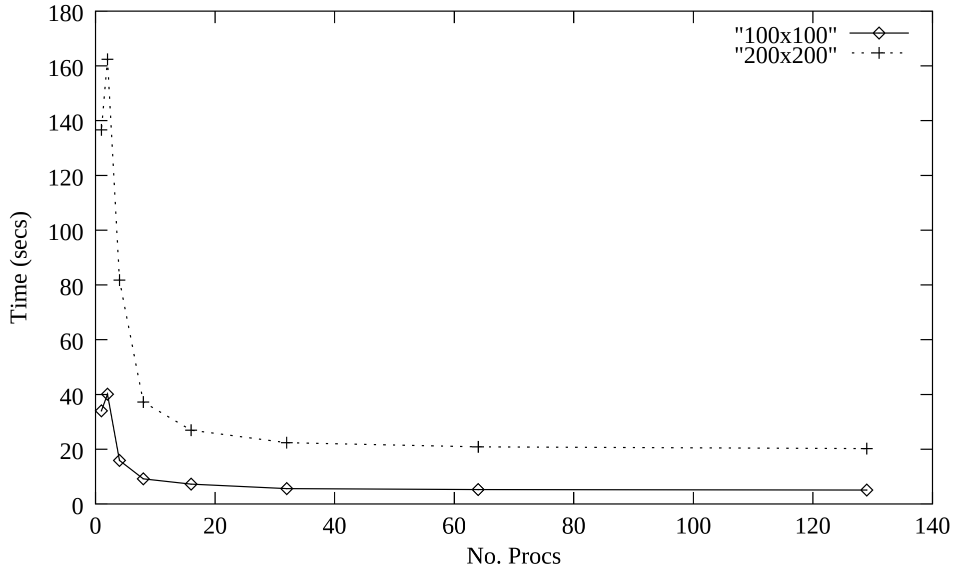

Figure 5.22: Ray tracer - AP1000 runtimes up to 129 processors

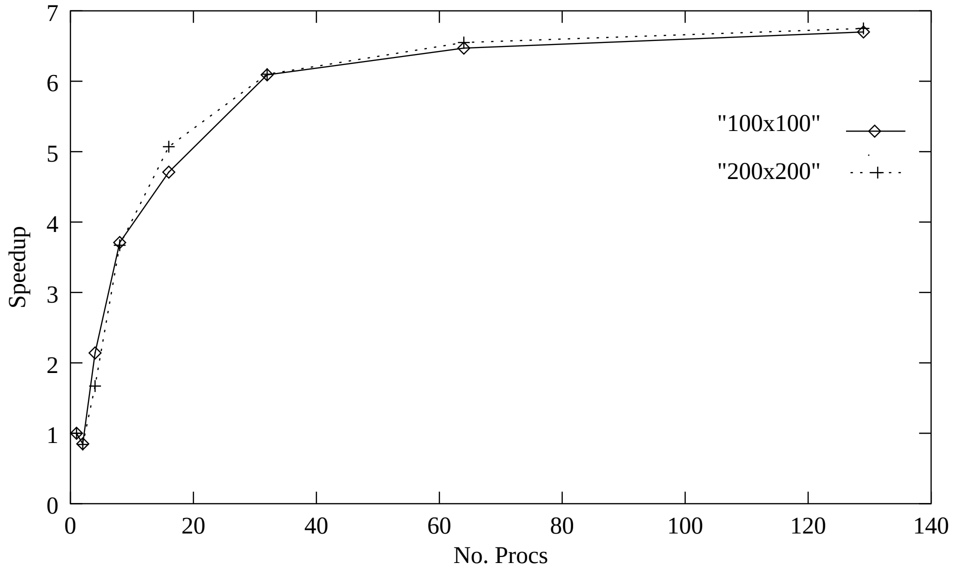

Figure 5.23: Ray tracer - AP1000 speedup up to 129 processors

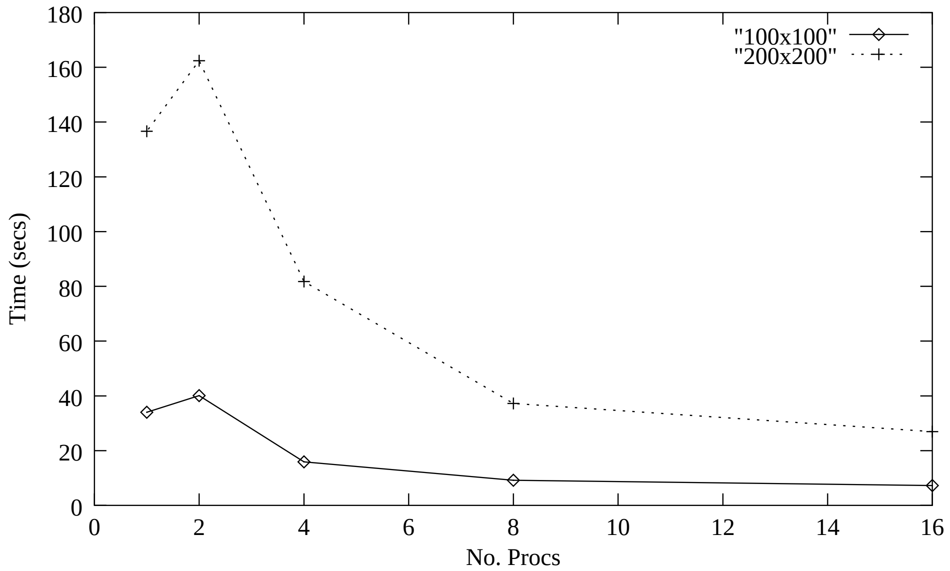

Figure 5.24: Ray tracer - AP1000 runtimes up to 16 processors

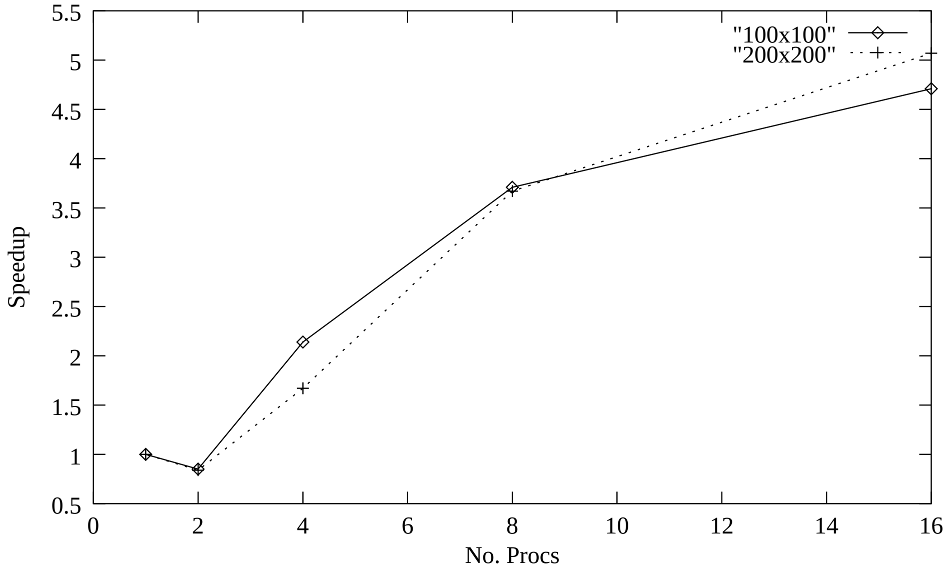

Figure 5.25: Ray tracer - AP1000 speedup up to 16 processors

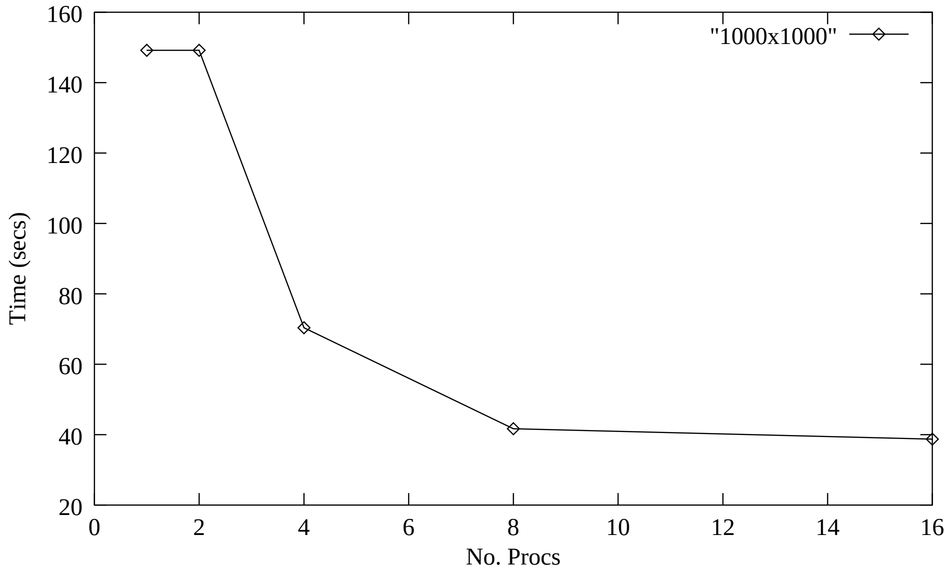

Figure 5.26: Ray tracer - AP3000 runtimes up to 16 processors

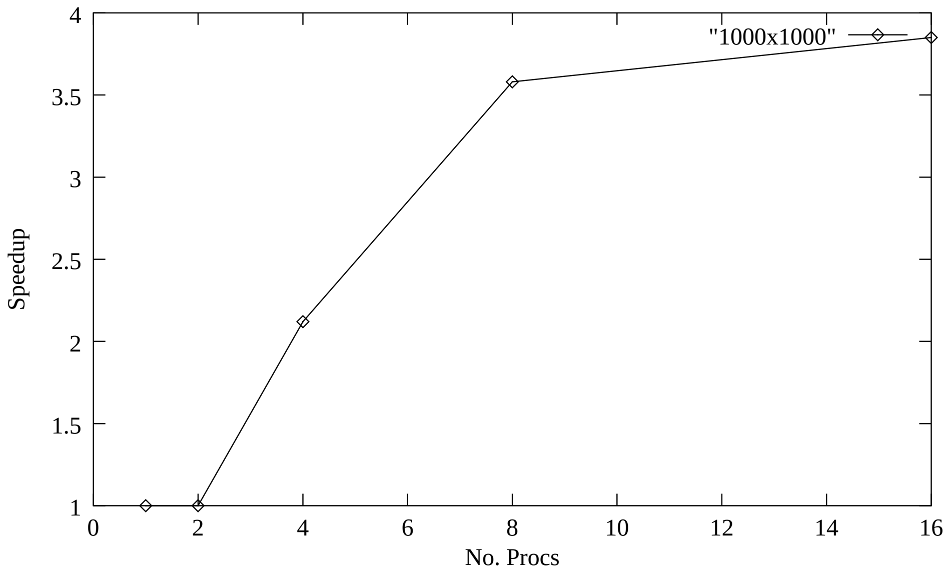

Figure 5.27: Ray tracer - AP3000 speedup up to 16 processors

5.3 Travelling Sales Person Genetic Algorithm

A set of standard functions from the parallel genetic algorithm language RPL2 were implemented in SML [23]. Using the SML versions of the library functions, a solution to the travelling sales person problem was implemented. One of the *maps*, determined by code inspection, was replaced with our parallel *map* skeleton. The example was implemented on the AP3000.

The travelling sales person problem is a well known problem based on finding the shortest route between N cities. The number of possible routes through the cities is N factorial. Genetic algorithms are generally highly non-deterministic and dependent on random effects. However, the genetic algorithm used for this example exploits regular data parallelism and terminates after a fixed number of cycles.

5.3.1 Results

Table 5.6 shows the runtimes for the travelling sales person on up to eight processors and with up to 150 cities. Table 5.7 shows the speedup data. The results are disappointing and show poor speedup. There are also several anomalies in the performance. These cannot easily be explained in terms of the models used for the *map* skeleton nor do they compare well with the results from previous examples. Significant further investigation of this particular implementation would be necessary to ensure the repeatability of these results and to investigate the anomalies.

no.procs	*Cities*								
	5	10	15	20	25	50	100	125	150
1	23	52	158	221	689	5223	34028	65164	106128
2	35	85	161	382	674	3100	39394	51005	170472
4	26	48	109	246	51	1855	13248	15333	32276
8	27	51	77	159	222	985	6650	13052	26211

Table 5.6: TSP AP3000 runtimes

no.procs	*Cities*								
	5	10	15	20	25	50	100	125	150
1	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
2	0.7	0.6	1.0	0.6	1.0	1.7	0.9	1.3	0.6
4	0.8	1.1	1.4	0.9	13.5	2.9	2.6	4.2	3.3
8	0.9	1.0	2.0	1.4	3.1	5.3	5.1	4.8	4.0

Table 5.7: TSP AP3000 speedup

Chapter 6

Conclusions

6.1 Introduction

This thesis has investigated the use of parallel skeletons for executing higher order functions in functional languages and has implemented skeletons for use in SML. It has demonstrated that it is possible to arbitrarily deeply nest skeletons and describes the novel design and implementation used to achieve this. The hybrid combination of static and dynamic techniques for allocating skeletons to processors, allowing a static combination of skeletons to be executed on varying numbers of processors is particularly notable. Additionally, the implementations of the skeletons are robust and are still being used for research purposes; details of research using the skeletons can be found in [26], [24], [23], and [27].

6.2 Thesis Summary and Contributions

The thesis introduced the concepts of parallelising and nesting of higher order functions and described an automatically parallelising compiler which used the skeletons in the thesis to achieve parallelisation. A literature survey considered various skeleton and skeleton-like methodologies, and in summary compared them to the system described in this thesis.

The majority of this thesis described two skeletons, *map* and *fold*, and the coordination layer used to combine and control them. These skeletons are novel in their ability to be nested to an arbitrarily deep level. They are also novel in their ability to be executed either in parallel or sequentially. This makes them particularly flexible and obviates the need for recompilation should the number of processors in a target system change. This was a feature that was found to be particularly useful when benchmarking on varying numbers of processors. The flexibility of the skeletons was further demonstrated by the number of platforms on which they have been used during development and testing (Section 1.3), including heterogeneous use.

The thesis also described a skeleton network that is used to represent the occurrences of skeletons within a source program. It was explained, with an example, how this network is used as a basis for the light-weight processor allocation phase that occurs at program start-up time.

A number of examples were presented with performance data, including one which shows the various phases in transforming from a sequential SML program to a parallel skeleton based program. The phases include translating from SML to Objective Caml, the conversion from higher order function calls to skeleton calls, the associated network, and the actual processor topology for a specific number of processors.

6.3 Limitations and Discussion

Limitations of the work presented are now noted and discussed.

- The actual allocation of processors to skeletons relies on weights. Although these weights should really be provided by some form of program analysis, hopefully to be implemented in the parent compiler, currently a set of dummy weights are used. These have been selected to give reasonable, average processor allocations but are unlikely to result in optimal processor allocation in general.

- Due to time limitations, only two skeletons, map and fold, were implemented. Although these provide good evidence for skeleton techniques and are used to achieve some effective parallelisation, a complete system would require more skeletons. In particular, skeletons to allow task parallelism, such as compose, would be a worthy addition.

- The examples we have shown here are all quite regular in nature. This has undoubtedly aided parallelisation. In would have been useful, given more time, to have investigated techniques for efficiently parallelising less regular problems.

- Although allowing arbitrary depth nesting can hardly be termed a limitation, its true value is not obvious. The motivation for nesting is that where nesting occurs we can improve parallelisation by splitting the outer and inner parts of a nested problem. Many problems of this nature, such as our examples, do not appear to contain significant work inherent in the outer level; the outer higher order function is only for manipulating the data structure. However, if problems containing significant work both in the outer and inner levels do exist it is likely that significant performance benefits may be reaped using a nested strategy.

- We have noted in our discussion of processor allocation techniques that logical branching statements cannot be represented by a conventional static processor network. For this reason we do not support parallelising higher order functions in conditional statements.

6.4 Future Work

- If more effort is to be employed in the research and development of nested skeletons it would be desirable to investigate the nature of nested problems themselves. A number of questions remain to be answered. How often do nested problems occur? Is there useful parallelism within these nested problems? How deeply does nesting occur? It may be that deeply nested problems are atypical and are not worthy of significant attention. It would be helpful to construct a library of benchmark nested applications to aid this research.

- Within the context of our own system, it would be beneficial to implement further skeletons. Given a larger set of base skeletons, investigation could be carried out into which skeletons find naturally efficient implementations on which platform types.

- As noted earlier, that we have not carried out research into techniques for irregular data structures could be a significant limitation. Investigation into skeletons that provide predictable and good speedup with irregular data structures could produce much useful work.

- Although a conventional static network cannot represent higher order functions in conditional statements, they will need to circumvent this problem if they are to gain widespread acceptance. One possible solution might include generating topologies for all possible paths through conditional statements. If the topologies differed only at the points where the relevant conditional statements came into effect, it might be possible to delay selection of topology until after a specific branch of a conditional statement was chosen.

Appendix A

Modifiable Skeleton Properties

A few properties of the skeletons can be modified using preprocessor definitions in the header files for the system. Judicious modification of these values could significantly alter the behaviour of the skeletons and, hence, can be used as a crude way of tuning the skeletons to a particular problem. These various properties are noted below. They are shown in the form of their C definitions, including their standard values.

A.1 System Sizes for the Skeleton Network

The following are constants relating to the static size of the skeleton network.

```
#define ABS_POOL_SIZE 250

#define ABS_LIST_SIZE 100
```

To prevent conflicts with Objective Caml's memory management we use no dynamically allocated memory. `ABS_POOL_SIZE` is the size of the memory pool from which all *NODE*s and *SEQ*s are drawn. In the implementation of *SEQ*s each *SEQ* has an array of pointers to *NODE*s or *SEQ*s; the length of this list is `ABS_LIST_SIZE`. If these need expanding a runtime error should be generated that states this.

A.2 Processor Allocation and Logging

```
#define MIN_PAR_PROCS 3

#define LOGGING TRUE
```

`MIN_PAR_PROCS` is used in generating the processor allocation data, each parallelised skeleton must have at least this number of processors.

During processor allocation we may wish to output data describing the allocation that was chosen, we switch this on by setting `LOGGING` to true. The output file is called proc_alloc_log$n$ where n is the process i.d. of the process running on the global root processor (not to be confused with the processor i.d). A typical file may look like the following:

```
Total No. procs in system:7


fn:ff2, Noskels:1, perskel:3, tot:3, bcount:2, bfirst:1

fn:ff1, Noskels:2, perskel:3, tot:6, bcount:4, bfirst:3

Unallocated processors:0
```

For each skeleton we have:

- **fn** - the function name (from the *Skeleton Network*),
- **Noskels** - the number of instances of that skeleton,
- **perskel** - the number of processors in each instance of the skeleton,
- **tot** - the total number of processors in all instances of the skeleton,
- **bcount** - the number of processors in the bottom row of all instances,
- **bfirst** - the processor i.d of the first leaf processor in this skeleton.

Note that ff1 has 6 processors and ff2 has 3 processors but there are only 7 in total, this is because two of ff2's processors are shared with ff1. We need to know how many processors there are at the bottom of all instances of a skeleton as this is how many instances there will be of any skeleton nested immediately below it. Also, we need to know the processor i.d. of the first leaf processor as this is also the first root processor of any nested skeleton.

A.3 System Constants for the Skeleton Implementations

The following are a number of system constants that are used by the skeletons.

```
#define OPT_MESG_SIZE 3

#define UNPROFILED_PROC_ALLOC TRUE

#define SEND_FINAL_RES FALSE

#define MAX_SKEL_BUF_SIZE 500000

#define MAX_LISTS 5000
```

The `OPT_MESG_SIZE` is how many list elements we put in each of *pmap*'s work packets. `UNPROFILED_PROC_ALLOC` tells the Processor Allocation to use dummy weights. This is necessary as at the time of writing the compiler was not able to generate weights for use in processor allocation. This should be set to `FALSE` when the profiling is implemented. `SEND_FINAL_RES` switches the final distribution of results on and off. The final distribution of results to all processors places a copy of the total result for a parallel computation onto all processors. It is necessary to transmit a final result to all processors when actions in the code depend on the values calculated as part of a skeleton instance. If this were not done, processors would have only their own partial results which might well result in undefined behaviour of some processors in the system. `MAX_SKEL_BUF_SIZE` is the size of the pack buffer for sending messages. It is common to both *map* and *fold* skeletons. Finally, `MAX_LISTS` is the maximum number of sublists of answers a *pmap* master processor can receive: it collects all the sublists together prior to combining them into one list. As before, if any of these statically declared sizes are exceeded a runtime error is generated stating which was exceeded. This can be fixed by enlarging the appropriate number and recompiling.

Bibliography

[1] B. Bacci, B. Cantalupo, M. Danelutto, S. Orlando, D. Pasetto, S. Pelagatti, and M. Vanneschi. An environment for structured parallel programming. In L. Grandinetti, M. Kowalick, and M. Vaitersic, editors, *Advances in High Performance Computing*, pages 219–234. Kluwier, Dordrecht, The Netherlands, 1997.

[2] T. Bratvold. Skeleton-based parallelisation of functional programs . In *PhD thesis.* Dept of Computing and Electrical Engineering, July 1995.

[3] Silvia Breitinger, Rita Loogen, Yolanda Ortega-Mallén, and Ricardo Pe na. Eden - Language Definition and Operational Semantics. Technical Report 96-10, Philipps Universität Marburg, Reihe Informatik, Fachbereich Mathematik und Informatik, 1998.

[4] M. I. Cole. *Algorithmic skeletons: structured management of parallel computation* . Pitman, 1989.

[5] M. Danelutto, F. Pasqualetti, and S. Pelagatti. Skeletons for Data Parallelism in P^3L. In C. Lengauer, M. Griebl, and S. Gorlatch, editors, *Proc. of EURO-PAR '97, Passau, Germany*, volume 1300 of *LNCS*, pages 619–628. Springer, August 1997.

[6] J. Darlington, A. J. Field, P. G. Harrison, D. Harper, G. K. Jouret, P. J. Kelly, K. M. Sephton, and D. W. Sharp. Structured Parallel Functional Programming . In H. Glaser and P. Hartel, editors, *Proceedings of the Workshop on the Parallel Implementation of Functional Languages*, pages 31–51. CSTR 91-07, Department of Electronics and Computer Science, University of Southampton, 1991.

[7] J. Darlington, M. Ghanem, and H. W. To. Structured parallel programming . In *Proceedings of MPPM, Berlin*, 1993.

[8] J. Darlington, Y. Guo, H.W. To, and J. Yang. Parallel Skeletons for Structured Composition. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 19–28. ACM Press, 1995.

[9] J. Darlington, Y-K. Guo, H. W. To, and J. Yang. Functional Skeletons for Parallel Coordination. In S.HAidi, K. Ali, and P. Magnusson, editors, *Proceedings of EuroPar'95*, pages 55–69. Springer-Verlag, August 1995.

[10] J. Darlington and H. W. To. Building Parallel Applications Without Programming. In *Abstract Machine Models*, Leeds, 1993.

[11] John Darlington, Yike Guo, Hing Wing To, Qian Wu, Jin Yang, and Martin Kohler. Fortran-S: A Uniform Functional Interface to Parallel Imperative Languages. In *Proceedings of the Third Parallel Computing Workshop*, Fujitsu Laboratories Ltd, Kawasaki Japan, 1994.

[12] Xavier Leroy et al. *The Objective Caml system release 3.00.* INRIA, April 2000. Available at http://caml.inria.fr/ocaml/htmlman/.

[13] Jeroen Fokker. *Functional Programming.* 1995. Available online at http://www.haskell.org/bookshelf/functional-programming.dvi.

[14] L.A. Galán, C. Pareja, and R. Peña. Functional Skeletons Generate Process Topologies in Eden. In *International Symposium on Programming Languages: Implementations, Logics, Programs (PLILP)*, volume LNCS 1140. Springer, 1996.

[15] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI.* The MIT Press, 1994.

[16] M. Hamdan. *A Combinational Framework for Parallel Programming Using Algorithmic Skeletons.* PhD thesis, Department of Computing and Electrical Engineering, Heriot-Watt University, 2000.

[17] Kevin Hammond and Greg Michaelson, editors. *Research Directions in Parallel Functional Programming.* Springer, November 1999.

[18] Paul Hudak, John Peterson, and Joseph H. Fasel. A gentle introduction to haskell 98. Technical report, Yale University and University of California, October 1999. Available online at http://www.haskell.org/tutorial/.

[19] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice Hall, 1978.

[20] Ulrike Klusik, Rita Loogen, and Steffen Priebe. Controlling Parallelism and Data Distribution in Eden. In Stephen Gilmore, editor, *Proceedings of the Second Scottish Functional Programming Workshop*, pages 117–130, 2000.

[21] Hans-Wolfgang loidl, Ulrike Klusik, Kevin Hammond, Rita Loogen, and Philip W. Trinder. GpH and Eden: Comparing Two Parallel Functional Languages on a Beowulf Cluster. In Stephen Gilmore, editor, *Proceedings of the Second Scottish Functional Programming Workshop*, pages 149–164, 2000.

[22] D.L. McBurney and M.R. Sleep. Experiments with the ZAPP: Matrix Multiply on 32 Transputers, Heuristic Search on 12 Transputers. Internal Report SYS-C87-10, School of Information Systems, University of East Anglia, 1987.

[23] Greg Michaelson and Paul Bristow. Parallel functional genetic algorithms in Standard ML from RPL2. Technical Report RM/99/9, Heriot-Watt University, Dept of Computing and Electrical Engineering, August 1999. Also in Draft Proceedings of 1st Scottish Functional Programming Workshop, P.Trinder and G.Michaelson (Eds).

[24] Greg Michaelson, Norman Scaife, Paul Bristow, and Peter King. Nested Algorithmic Skeletons from Higher Order Functions. *Parallel Algorithms and Applications*, 16:181–206, Aug 2001. Special issue on High Level Models and Languages for Parallel Processing.

[25] R. Milner, M. Tofte, and R. Harper. *The definition of Standard ML.* MIT Press, 1990.

[26] G.Michaelson N.Scaife and S. Horiguchi. Comparative cross-platform results from a paralleliszing SML compile. 2001. submitted to IFL 2001.

[27] G.Michaelson N.Scaife, P.Bristow and P.King. Engineering a parallel compiler for SML. In T.Davie Clack and K.Hammond, editors, *Proceedings of 10th International Workshop on Implementation of Functional Languages*, pages 195–212. University College London, September 1998.

[28] L.C. Paulson. *ML for the Working Programmer.* Cambridge University Press, second edition, 1996.

[29] M. Danelutto R. Di Meglio S. Orlando S. Pelagatti and M. Vanneschi. A Methodology for the Development and the Support of Massively Parallel Programs. *Future Generation Computer Systems*, 8:205–220, August 1992.

[30] S. Pelagatti. Compiling and supporting skeletons on MPP. IEEE Computer Society Press, 1997. Presented at MPPM'97, London, November 1997.

[31] Susanna Pelagatti. *A Methodology for the Development and Support of Massively Parallel Programs.* PhD thesis, Dipartimento di Informatica, Universita degli studi di Pisa, November 1993.

[32] Roopa Rangaswami. *A Cost Analysis for a Higher-order Parallel Programming Model.* PhD thesis, University of Edinburgh, 1996.