Nested Algorithmic Skeletons from Higher Order Functions

Greg Michaelson, Norman Scaife, Paul Bristow and Peter King

Department of Computing and Electrical Engineering,

Heriot-Watt University, Riccarton, Edinburgh, EH14 4AS

{greg,norman,paul,pjbk}@cee.hw.ac.uk

October 11, 1999

Abstract

Algorithmic skeletons provide a promising basis for the automatic utilisation of parallelism at sites of higher-order function use through static program analysis. However, decisions about whether or not to realise particular higher-order function instances as skeletons must be based on information about available processing resources, and such resources may change subsequent to program analysis.

In principle, nested higher-order functions may be realised as nested skeletons. However, where higher-order function arguments result from partially applied functions, free-variable bindings must be identified and communicated through the corresponding skeleton hierarchy to where those arguments are actually applied.

Here, a skeleton based parallelising compiler from Standard ML to native code is presented. Hybrid skeletons, which can change from parallel to serial evaluation at run-time, are considered and mechanisms for their nesting are discussed. Compilation stages are illustrated through a simple nested higher-order function based algorithm for multiplying matrices of arbitrary length integers and performance figures for compiled code running on a Fujitsu AP3000 are discussed.

Key Words: higher order functions; skeletons; parallelising compilation

Computing Reviews Categories: D Software; D3 Programming Languages; D3.4 Processors

1 Introduction

1.1 Higher-order functions and algorithmic skeletons

The congruence of Backus' seminal paper [1] advocating functional programming as a sound formal discipline, Bird and Meerten's calculi for manipulating functional programs based on higher-order operators [2], and Cole's characterisation of algorithmic skeletons [3] has led to considerable interest in the derivation of parallel implementations from programs based on higher-order constructs [4]. In particular, attention has focused on a small group of higher order functions; *map*, *fold* and *compose*. *map* applies a function `f` to each element of a list:

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

fold, shown here in left-to-right form, applies a function `f` "between" elements of a list:

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

compose, often written `(f o g) x`, applies a function `f` to the result of another applying another, `g`:

```
fun compose f g x = f (g x)
```

These functions capture paradigmatic programming patterns and prove to have wide applications in programming.

Each function may be realised through a parallel algorithmic skeleton, a template for a generic parallel computation pattern, which may be instantiated with particular argument function instances. Thus, *map* is often implemented as a *process farm*, shown in Figure 1(a), where a *farmer* process sends list elements to *worker* processes, each of which runs the mapped function `f`. The result from each worker is returned to the farmer for reassembly into the final result list.

Similarly, *fold* with an associative and commutative function may be realised as a divide-and-conquer network, shown in Figure 1(b), where, at each level, a parent *divides* it's argument list and sends sub-lists to it's children. Each child applies the folded function `f` to it's sub-list and returns the result to the parent for recombination (*conquer*ing) with `f`.

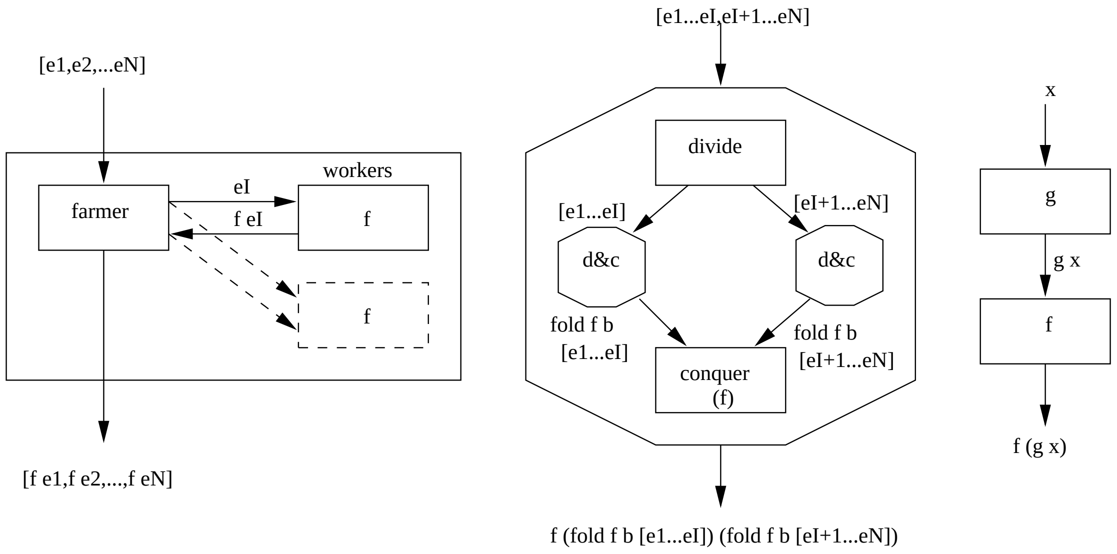

Figure 1: (a) Process farm (b) Divide and conquer (c) Pipeline

Finally, compose may be realised as a pipeline of processes, shown in Figure 1(c), where the process for the inner function `g` sends it's result to that for the outer function `f`.

1.2 Compiler overview

We are investigating the development of a parallelising compiler for a pure functional subset of Standard ML, identifying potential parallelism in sites of higher order function (HOF) use, for realisation through parallel algorithmic skeletons, ultimately generating native code linked with an MPI library [5]. The design of the compiler is discussed in detail in [6]. Here we consider the development of the compiler spine, which enables its use in generating instantiated parallel skeletons from explicitly nominated higher order functions.

A schematic outline of the design is shown in Figure 2. The *front end* lexically analyses, parses and type checks the SML prototype. The *pre-analyser* is divided into two phases. The *network analysis* phase extracts a description of the topography of recognised HOFs from the program. The *defunctionalisation* phase optionally reduces the program to be both monomorphic and first order although not necessarily more efficient. The *profiler*, based on the structural operational semantics (SOS) for SML, runs the prototype on test *data sets* to determine abstract communica-

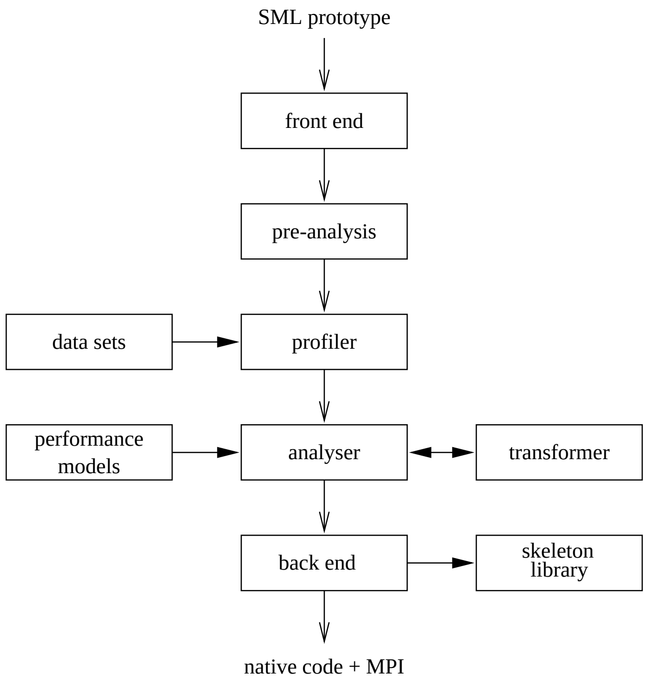

Figure 2: Overall compilation scheme

tion and processing costs in HOF use. The *analyser* uses *performance models* instantiated for the target architecture to predict concrete costs. This phase also performs free variable analysis to generate communications costs. Finally, the *back end* selects appropriate *skeletons* for HOFs with predicted useful parallelism and instantiates them with sequential code plus additional support code associated with the analysis phase.

Our design is novel in a number of respects:

- Many researchers have proposed the use of static cost models to predict parallel performance, in particular Skillicorn [7]. These are most effective where parallelism is limited to a small number of regular constructs, for example as in NESL [8] and FiSh [9]. However, in general cost modelling is undecidable: program analysis results in a set of recurrence relations which must then be solved [10]. In contrast, we use dynamic prototyping to try and establish typical behaviour. These implementation independent measures are then used with target architecture metrics to predict actual costs. Using this approach on an SML subset, Bratvold [11] achieved around 85% accuracy.

- Our compiler is intended to enable code generation from HOFs nested to arbitrary depth. Rangaswami's HOPP general scheme [12] actually provided for nesting up to three deep of a fixed set of HOFs. While it may be argued that deeper nesting is rare, we think that arbitrary nesting of arbitrary HOFs presents interesting challenges, in particular for skeleton coordination to optimise data distribution.

Our aim is to construct a system within which users have no necessary knowledge of sources of parallelism in their programs. Instead the compiler tries to locate and optimise parallelism for them. However, the location of parallelism in HOFs for exploitation through algorithmic skeletons appears both to restrict sources of parallelism in functional programs and to oblige the user to have some understanding that HOFs must be used for parallelism to be exploited. This is indeed the case for the spine of our system where only explicitly nominated HOFs are realised through skeleton instantiation. However, in our full system, the use of a HOF will be no guarantee of parallelism: that will depend on program analysis determining that such parallelism is useful in the sense of computation outweighing communication.

In their survey of models and languages for parallel computation [13], Skillicorn and Talia identify six stages between *Nothing Explicit, Parallelism Implicit (NEPI)* models which abstract away from all aspects of parallelism and *Everything Explicit (EE)* models where all aspects must be treated explicitly. They note, as have others, that exploiting the implicit parallelism throughout functional programs through *dynamic* mechanisms, for example graph reduction, is very hard: it is more effective to limit sites of parallelism to a set of generic components with determinate behaviours. Thus, Skillicorn and Talia place algorithmic skeletons within this *static* subclass of NEPI. However, as will become clearer below, our system also encompasses the most concrete EE model as MPI is available to skeleton developers working at a variety of levels.

Finally, we envisage that a full system would employ further strategies to try and optimise and identify parallelism. First of all, where the analysis indicates that HOFs cannot be usefully realised as skeletons, then the *transformer* will attempt to rearrange higher order function configurations to optimise parallelism, using standard HOF transformational identities. We have completed a prototype transformer but this is not discussed further here. Secondly, we are investigating the use of proof planning techniques [14] in synthesising HOF instances from programs that do not contain them, to try to locate latent opportunities for parallelism. Currently, we are completing a proof plan for HOF synthesis and have started to implement it within the λ-CLAM proof planner but, again, these are not discussed further here.

1.3 Related Skeletal Approaches

The term *algorithmic skeleton* was first used by Cole [3] but there are earlier papers working in the same direction. For example, Cole cites ZAPP (Zero Assignment Parallel Processor) [15] which 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, variously labelling skeletons as templates, paradigms, patterns, forms and parallel constructs [11].

There are some similarities between Cole's original work and our own. His Task Queue (TQ) and Fixed Degree Divide and Conquer (FDDC) skeletons share similar computation models with our parallel map and fold skeletons (Farm and divide and conquer respectively). Also, both sets of skeletons implicitly handle communications and data placement. The similarities end here. Cole's TQ is a general farm where as our map is implemented using a farm. His FDDC passes all work to the leaf nodes of the processor tree, with intermediate nodes only executing a join function. Our intermediate nodes execute both the main work function and the join function.

Darlington et al [16] develop Cole's ideas though with less emphasis on the target architecture. They introduce the concept of inter-skeleton transformation to aid efficient implementation on various platforms. Later papers [17] introduce performance models associated with each skeleton and instantiated with machine specific parameters, and a co-ordination language [18]. Their Structured Coordination Language (SCL) has three components: configuration and configuration skeletons, elementary skeletons and computation skeletons. SCL is combined with a base language to form a structured parallel programming scheme. While SCL is functional in form, the base language need not be; indeed, one intention is to enable the reuse of extant components in mainstream imperative languages.

Pelagatti [19] describes a methodology based on a small set of parallel constructs (skeletons) and combining rules. These skeletons are first class objects of the host language, supporting hierarchical nesting. She goes on to demonstrate this methodology with P^3L [20]. P^3L represents a conceptually quite different approach to that of Cole and Darlington with its parallel constructs being representative of the actual parallel operations rather than the semantics. Also, P^3L differs from other approaches in being an imperative language.

2 The Parallel Compiler

2.1 Compiler components and structure

The overall compiler structure is outlined in [21]. We use the ML Kit [22] for the compiler front end, to parse and elaborate the source SML program. The ML Kit is closely based on the Standard ML definition [23] and provides an evaluator which mirrors the dynamic semantics. All our analysis is conducted within this framework, including identification of skeletons, performance analysis and source code transformations both for exposing parallelism and for supporting the final compilation.

Skeletons are nominated by providing an SML signature called `HOFS` in the prelude. This is achieved by matching pairs of definitions in the signature such that the first definition corresponds to the HOF and the second to the parallel equivalent. The convention is adopted that the two must have the same type signature but with abstraction points in the HOF (e.g.. the functional argument in the `map` HOF) being replaced by `string` types in the parallel equivalent. The reason for this is that Objective Caml objects to be referenced within the C code, in which the skeletal harnesses are written, have to be registered under a `string` name in the Objective Caml code. It is convenient to pass the string name that the skeletal function has been registered under to the parallel implementation:

```
signature HOFS =
  val map : ('a -> 'b) -> 'a list -> 'b list
  and pmap : string -> 'a list -> 'b list
end
```

Here, the `map` HOF with one functional argument is to be replaced by calls to `pmap` which is a processor farm implementation. The identification of skeletons has greatly increased complexity as a result of this mechanism but the idea is to allow programmers to add their own skeletons. There is one incidental advantage, however, in that a manual system of controlling parallelism consists of simply renaming `map` in the HOFS signature to a name not found in the source code (e.g.. `manual_pmap`).

Once our analysis is complete, we have a parallel program in Core SML in which the HOFS to be implemented in parallel have been transformed into calls to predefined parallel constructs. These constructs are implemented in the target compiler for which we use the Objective Caml [24] language. This is a dialect of ML, based around a different evaluation mechanism [25] from the SOS of Standard ML, and has a lightweight implementation with modest memory requirements and respectable performance. Although Objective Caml outputs native code directly there is a full C interface which allows us to incorporate our parallel skeletons. The only additional requirement for this is generating Objective Caml from the annotated SML code. This is a simple process since there is a high degree of semantic equivalence between the constructs in the two languages. The main points of diversion are in the type systems and as a result our source language is a restricted form of SML limited to type and language constructs which are common to both SML and Objective Caml.

2.2 Network analysis

During our analysis, we need to map the SML syntax tree onto a static network of processors. To simplify this process we generate an *abstract network* description of the program. This has some conceptual similarities with process task graphs but at the level of skeletons rather than tasks.

This analysis works by annotating the program's types with information concerning the potential activation of skeletons in the creation of objects with those types. This is carried out in a post-elaboration phase which uses the type information generated by the elaborator. The abstract network is described by the following datatype:

```
datatype absnet = base
                | node of name * absnet list
                | seq of absnet list
                | alt of absnet list
```

Here, `base` means no skeleton instances, `node` indicates an instance of a (potentially nested) skeleton, `seq` means a set of skeleton instances where each skeleton must terminate before the next one can start and `alt` is skeleton alternation. Note that we cannot handle alternation in our final implementation since the implemented skeletons do not support it, this means we can only have one skeleton instance within a *match* [1] construct. The result of this analysis is illustrated by the following:

```
(* Simple map over base types *)
val skelres1 = map (fn x => x + 1) [1,2,3]
val skelres1 :: node(map,[base])

(* Nested map *)
val skelres2 = map (map (fn x => x)) [[1],[2]]
val skelres2 :: node(map,[node(map,[base])])

(* A recursive function with a HOF *)
fun ff [] = []
  | ff (h::t) = (map (fn x => x) h)::(ff t)
val ff :: (seq [seq ['a]]->seq [node(map,[base])])
val skelres3 = ff [[1]]
val skelres3 :: seq [node(map,[base])]
```

Notice that recursion over functions with skeleton instances results in a `seq` construct. This means that `seq` represents a sequence of zero to a potentially infinite number of instances.

Once we have the abstract network description, allocation of processing resources at compile time becomes relatively trivial. Given the profiler information, the sequential processing costs for the HOF functional arguments can be used to generate weights for the skeleton instances allowing the network description to contain information about the degree of parallelism present in each instance. Note that the abstract network is available at runtime and that processing resources can be allocated at runtime prior to executing the program. This is not intended as the basic mode of

[1] Here the term *match* refers to the *match* grammar object in the Standard ML definition. In the `Core` language, this is the only source of conditional execution.

operation for the system but gives more flexibility in terms of running compiled code on varying processing resources.

2.3 Backend analysis

Once the program has been analysed for parallelism the nominated HOFs have to be transformed into calls to parallel constructs and some additional code wrapped around them to allow the skeletons to be launched. There are several major complications in running a parallel construct from within the abstract syntax tree of an SML program:

- To call an Objective Caml function from a C skeleton implementation, the function has to be registered under a unique string name;
- Since our runtime system cannot transmit closures, all the functions that are called by remote processes have to be available at the top level of the code. This is so they can be registered with the skeletal code on remote processors;
- Free values in functions involved in skeleton instances have to be transmitted prior to running the remote processes. In addition, these have to be passed to the skeletal code on the process which initiates the skeleton.

Our system handles these problems on two levels, firstly by generating auxiliary functions at the top level with no free variables and secondly by implementing full defunctionalisation. Which of these is relevant depends on the structure of the program: automatic detection of this has not yet been attempted.

The first method involves identifying free values in expressions and annotating the generated program with code to transmit these values prior to parallel execution. This is the most efficient method of handling such values since there is minimal additional execution overhead and only necessary values are transmitted. The difficulty occurs with free functional values. It is possible under some circumstances to track the free functionals back to their original definitions and generate an auxiliary function with the free functional's definition built in.

```
val y = 2
fun ff sum z x = sum (x,sum (y,z))
val result =
  let
    val z = 3
    fun sum (z,x) = x + y + z
  in
    map (ff sum z) [1,2,3]
  end
```

Figure 3: A simple SML example with functional arguments and free values.

As an example of this method consider the SML fragment in Figure 3. In this code, the function `ff` is used as the argument to the `map` skeleton but it contains both free functionals and free data. This code is converted by our analysis phase into the code shown in Figure 4.

```
val y = 2
val rec ff = fn sum => (fn z => (fn x => sum (x,sum (y,z))))
val rec ff1 =
  function z =>
  let
    val rec ff = fn sum => (fn z => (fn x => sum (x,sum (y,z))))
    val rec sum = fn (z,x) => (op +) ((op +) (x,y),z)
  in
    ff sum z
  end
val _ =  Callback.register "ff1" ff1
val result =
  let
    val z = 3
    val rec sum = fn (z,x) =>  (op +) ((op +) (x,y),z)
  in
    (fn _ => (Hofs.pmap : (string -> (int list -> int list))) "ff1" [1,2,3])
      (Callback.register "ff1_fvs" z)
  end
```

Figure 4: The transformed code for the simple example.

The free functional `sum` in the expression `(ff sum z)` is not available at the top level so an auxiliary version of `ff` is created (`ff1`) at the top level with the value of `sum` at that instance built in. The function `Callback.register` registers the value `ff1` (here functional) with the skeletal code so that the unique string name `"ff1"` can be used as an argument to the skeleton implementation, the function `Hofs.pmap`. Note that the type of `pmap` at this instance has to be reconstructed since replacing the `map` argument function (`int -> int`) with a string breaks the type chain. Without the explicit typing, `pmap` would be of type `string -> int list -> 'b list`.

The free value `y` does not need to be transmitted during skeleton operation since it is a top level declaration and available on all processors. This is important since we do not want to reconstruct the entire prelude in each auxiliary function. The free value `z` is not a top level declaration and has to be registered with the skeleton using the convention that the free data is registered under the same string as the functional argument to the skeleton with `"_fvs"` appended. The auxiliary function `ff1` has an additional tuple argument consisting of the free values which are supplied by the skeletal code on the worker processors which have been transmitted from the master.

Note that there is a degree of *dead* code in the resulting program, for instance the `sum` function definition in the `result` value `let` construct. This is present to allow residual sequential code to compile correctly and could be removed with a dead code elimination phase, if necessary.

This method works well with simple code but tends to cause duplication of code and calculation in the auxiliary functions. The method is not implemented for functions held in data structures: tracking of definitions through *match*es has proved to be highly complex. The second method is to defunctionalise the entire program. This is based on an idea by John Reynolds [26] and subsequently generalised to cover a full functional language [27]. In this technique, closures are effectively lifted to the top level of the language and represented by datatypes. This allows free functionals to be handled in the same way as free data but creates a global overhead of a datatype dereference for every function application. This method is currently being implemented.

3 Higher Order Function Skeletons

The parallel harness for our compiler is provided via skeletons implemented in C with the MPI message passing library. An interface between the imperative back-end and functional front-end is implemented in Objective Caml. The skeletons may be accessed directly through Objective Caml using the language's C interface.

Wrappers have been provided in Objective Caml allowing direct usage of parallel constructs if required. Indeed, one could even write further skeletons in Objective Caml rather than C, an approach advocated by Serot[28]. This also enables indirect SML access to the MPI functions, providing the option of writing skeletons directly in SML.

We use HOFs as sites of parallelism. Our initial HOFs all take lists as their major data argument. The lists are partitioned into segments which are sent to each processor (real or virtual) in the parallel environment. In addition, where the function argument to a HOF results from partial application, free variable values must be transmitted from the root processor to all other processing nodes. The processors then all execute the entire program with their sublists, using an SPMD model. After local processing the results are returned to a root processor and combined in a fashion dependent on the particular HOF.

3.1 Managing skeletons

For nested HOFs realised as nested skeletons, the *pskel* module:

- coordinates inter-skeleton communication;
- provides a homogeneous layer between heterogeneous skeletons.

Conceptually, there are three different types of processor: global root, child and nested master. Control of the entire skeleton filters down from the global root along with the names of functions to be executed. It is necessary to transmit function names so that each processor can execute the appropriate skeleton. All skeletons enter *pskel* by calling the top level HOF in the Objective Caml. Children of the global root receive the name of the function argument to to the top level HOF and use the *skeleton network* derived for the program to determine the appropriate skeleton type to execute. Hence, they enter the same skeleton as the global root and become its children. These children will, in due course, enter a nested skeleton if there is one, becoming nested masters. The process of distributing function names is then repeated with any child processors they have.

Figure 5 shows a map nested within a fold. In this topology the processors in the dashed box are involved in the outer *pfold* and the processors in the dotted box are involved in the inner *pmap*. Of particular interest are those four processors common to both boxes. These take part in

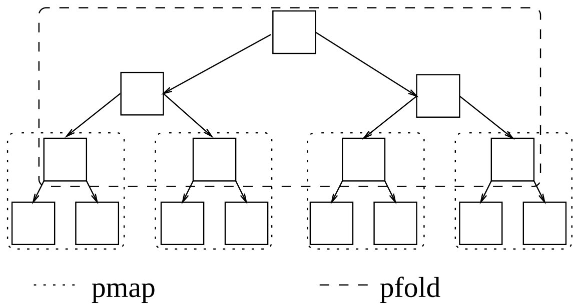

Figure 5: Topology for *pfold* nesting *pmap* on 15 processors

both skeletons, whereas all other processors are involved in exactly one skeleton. The processors involved in the nested *pmap*, but not in the *pfold*, will lack any data that may have been calculated between calling the *pfold* and the *pmap*. This data, along with top level skeletons' final results, must be supplied to the appropriate processors.

Currently the total final result is distributed to all processors in the system. This is as a result of our SPMD model where the outcome of the skeleton may determine the subsequent course of the program. It is intended that this final distribution will be replaced with free value analysis and distribution, removing unnecessary broadcasts.

3.1.1 Initialisation phase

The initialisation phase initialises data for each processor, describing its parent processor, child processors and other related coordination data. We refer to this as the *processor allocation data*. It also sets pointers to functions (known as registering) native to Objective Caml. These functions are the functional arguments to the HOFs that the skeletons replace. Additionally any utility functions the skeletons may need to use are also registered. The registration process takes strings representing the functions and returns pointers to function closures.

When a processor first starts it checks if it is *initialised* or has *not ever entered*. If the processor has not ever entered, the current skeleton is the first instance of a top level skeleton in the program execution that has been encountered. It must then build the skeleton network and create the processor allocation data before continuing. 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 it is unallocated then it waits for the skeleton to complete and collects a copy of the final result. A global termination message is used by a parent processor to signal to a child that there is will be no more work in the current skeleton and that it should exit.

3.1.2 Control structure

When the global root enters a skeleton it checks that it is the global root entry point. If it is not then the skeleton will be executed sequentially. Assuming the global root has found the global root entry point, then it will transmit the name of the function to be executed to only the child processors in its skeleton. If there is a nested skeleton below this, some of the child processors may, by executing the function supplied to them by the global root, become nested masters. The nested masters will then send a function name to their child processors. Because the HOFs 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 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 the total result is broadcast to all processors in the system to ensure that they are all in a consistent state when they are next entered.

3.2 *Pmap*, a parallel map replacement

As discussed above, our parallel map skeleton uses a process farm topology. Each skeleton of size *N* has 1 master processor and *N-1* workers. In an unnested skeleton the master will also be the global root; in a nested skeleton the master will be a nested master. The principle behind the processor farm is very simple. The master divides its list into a number of packets, one being sent out to each worker and the remainder being served on demand as workers complete their first packet.

The master processor repeatedly sends list packets to its children, which then execute Objective

Caml's *map* HOF over the contents and return the results. Note that this execution may result in nesting. The master processor is responsible for terminating the children.

Note that child processors terminate children within the skeleton, while the master leaves it to *pskel*. As stated above, a nested master may repeatedly re-enter the same skeleton instance. It cannot terminate its children as it may receive more work in the skeleton in which it is a child and hence become a nested master again. It is only when the nested master is a child again, and receives its own terminate message, that it can be certain that it no longer requires its child processors. It is then free to terminate children. This copes with terminating all children in nested *pmap* skeletons. In the unnested case, the global root is never a child processor and must therefore terminate its children at some other point, i.e. in *pskel*.

3.3 *Pfold*, a parallel fold replacement

The parallel fold skeleton uses a divide and conquer topology. Unlike the *pmap* skeleton we have no notion of one master and *N-1* worker processors. Instead the topology is better considered as one root processor with two children; those children in turn becoming parent processors for their own children. A parent processor is not analogous to a root processor. A root processor is the processor at the root of a skeleton: it may form the root of the entire processor tree (consisting possibly of a number of skeletons) or itself be nested below another skeleton. To minimise communications, each processor splits its list only once, a third for itself and each child. However, in a nested skeleton, this process may be repeated multiple times depending 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.

When the processor is entered, it first checks if it is a skeleton instance that should be evaluated sequentially. If so a callback to the *fold* HOF is made and the answer immediately returned. This arises in the nested case when a processor at the top level enters a nested skeleton whilst computing its part of the final result. It can also arise due to processor shortage.

If parallel evaluation is required then the setup routine is called, to calculate a list segment size based on the number of processors in the skeleton and to distribute free values. The processing then diverges with the master processor distributing the list and working over its own list segment. After calling a routine to collect answers from its children, it *fold*'s the results together with its own to produce a final result. In this skeleton the global root processor terminates its children rather than leaving it to *pskel*. These 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 a child must first check if it is to become a nested master prior to becoming involved in the parallel computation. 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. As with *pmap* a nested master terminates its children when it has reverted to being a child processor.

3.4 The Skeleton Network

The skeletons use a trimmed down abstract network, called a *skeleton network*. This has two constructs, namely *NODE* and *SEQ*. A *SEQ* contains *NODE*s that are to be executed sequentially. All *NODE's* contain skeleton occurrences and may also contain further *NODE*s and *SEQ*s. We impose the condition that all skeleton in the translated Objective Caml must have unique identifiers. With this condition and our two constructs we can represent all combinations of skeletons being nested within and occurring after each other.

To translate from the abstract network to the skeleton network we use a Standard ML program that examines the abstract network during compilation. This produces a C function containing a a series of nested function calls to two library functions, to be linked into the program to build the skeleton network at run time.

Consider again Figure 5. This can be represented with the skeleton network in Figure 6. Here, the skeleton network enables the child processors in the nested *pmap* skeletons to determine that they are part of a *pmap* skeleton, having entered *pskel* via the top level call to *pfold*.

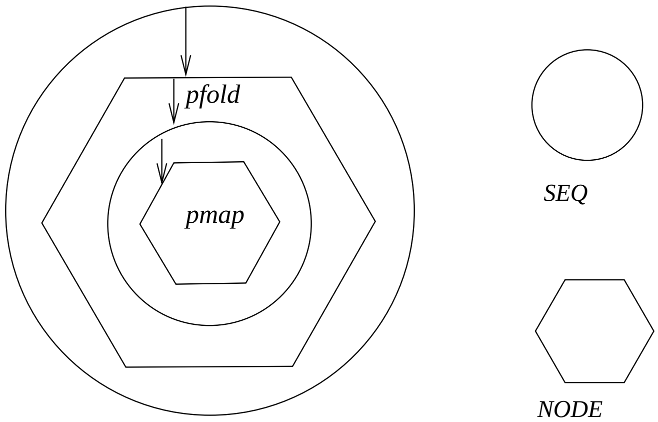

Figure 6: Skeleton network for *pfold* nesting *pmap*

4 Example: arbitrary length integer matrix multiplication

4.1 Overview

For example, consider multiplying matrices of arbitrary length integers(ALIs). Such integers are represented as lists of integer digits, from least significant to most significant, left to right:

`e.g. 189652784532 ==> [2,3,5,4,8,7,2,5,6,9,8,1]`

to ease carry propagation. Addition and multiplication of ALIs may be defined as:

```
(* add carry to ALI *)
fun addc 0 t = t |
    addc c [] = [c] |
    addc c (h::t) =
     (h+c) mod 10::addc ((h+c-((h+c) mod 10)) div 10) t;
fn : int -> int list -> int list

(* add ALI to ALI with carry *)
fun add c [] n2 = addc c n2 |
    add c n1 [] = addc c n1 |
    add c (h1::t1) (h2::t2) =
     (h1+h2+c) mod 10::
      add ((h1+h2+c-((h1+h2+c) mod 10)) div 10) t1 t2;
fn : int -> int list -> int list -> int list

(* multiply ALI by single digit *)
fun multd d [] = [] |
    multd d (h::t) =
     (d*h) mod 10::addc ((d*h-(d*h)mod 10) div 10) (multd d t);
fn : int -> int list -> int list

(* multiply ALI by ALI *)
fun mult [] n2 = [] |
    mult [d] n2 = multd d n2 |
    mult (h1::t1) n2 =
     add 0 (multd h1 n2) (0::(mult t1 n2));
fn : int list -> int list -> int list
```

In a pure functional setting, a matrix of ALIs may be represent by a list (rows) of lists (column values) of ALIs. This row-wise representation is profoundly inconvenient for matrix multiplication, where inner products are found for combinations of rows of the first and columns of the second. Thus, the second matrix must be transposed:

```
fun dist [] _ = [] |
    dist (h1::t1) (h2::t2) = (h1::h2)::dist t1 t2 |
    dist (h1::t1) [] = [h1]::dist t1 [];
fn : 'a list -> 'a list list -> 'a list list

fun transpose l = foldr dist [] l;
fn : 'a list list -> 'a list list
```

Next, given an inner product function:

```
fun innerprod (h1::t1) (h2::t2) =
     add 0 (mult h1 h2) (innerprod t1 t2) |
    innerprod _ _ = [0];
fn : int list list -> int list list -> int list
```

matrix multiplication may be expressed as:

```
(* multiply one row by all columns *)
fun rowmult row cols = map (innerprod row) cols;
fn : int list list ->
      int list list list -> int list list list

(* multiply all rows by all columns *)
fun outer cols x = rowmult x cols;
fun rowsmult rows cols = map (outer cols) rows;
fn : int list list list ->
      int list list list -> int list list list

(* transpose second and multiply with first *)
fun mmult m1 m2 = (rowsmult m1 o transpose) m2;
fn : int list list list ->
      int list list list -> int list list list
```

4.2 Parallelising the matrix multiplication

The overall multiplication function `mmult` consists of a composition of fold (`transpose`) and nested map (`rowsmult` and `rowmult`), shown in Figure 7. The fold for `transpose` cannot be realised as a parallel divide and conquer as it's argument `dist` is not associative or commutative. Furthermore, the compose is at the top level of the program. The equivalent pipeline will only receive a single data item rather than a stream so there is no exploitable parallelism unless `mmult` is used in turn within a repetitive construct.

Thus, let us consider the nested maps in `rowsmult`. In the following discussion, we assume that we have $M*N$ matrices of D-digit numbers. We also assume that the cost of communicating

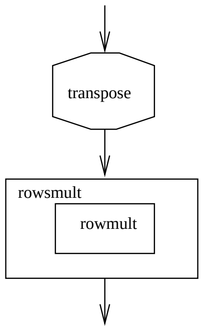

Figure 7: Nesting structure for `mmult`

a D-digit number is DC_{comm}, the cost of adding two D-digit numbers is DC_{add} and the cost of multiplying a D-digit number by one digit is DC_{mult}.

For `innerprod` with two N-number vectors, the processing cost will be:

$$ND^2(C_{mult} + 3/2C_{add}) - (N/2 + 1)DC_{add}$$

For `rowmult` the processing cost will be that for M calls to `innerprod`, divided by the number of processors performing `innerprod`. The communication cost will be: $(M + 2)NDC_{comm}$

For `rowsmult`, the processing cost will that of M calls to `rowmult`, divided by the number of processors performing `rowmult`. The communication cost will be: $3MND_{comm}$

Cursory examination of this naive analysis suggests that communication costs vary uniformly in the sizes of the matrices and of the ALI, while processing costs are dominated by the square of the size of the ALIs.

4.3 Parallel performance

A program to multiply two matrices of ALIs was compiled and run on the Imperial College Fujitsu AP3000, using up to 32 300 MHz UltraSPARC processors. Test cases were for 30*30, 40*40 and 50*50 matrices with 30, 40 and 60 digit ALIs, on 1, 3, 4, 8 and 16 processors. Up to 2 processors, both maps are realised sequentially. Between 3 and 7 processors, the outer map is realised as a farm skeleton. Beyond 7 processors, both maps are realised as farms, with symmetric processor allocation. Times in seconds are shown in Figure 8 and speedups (time on 1 processor/time on N

Rows	Cols	ALI size	Processors 1	3	4	8	16
30	30	30	70.05	35.27	28.29	18.59	7.68
		40	124.50	62.65	51.73	33.38	13.29
		50	194.04	98.08	77.96	50.76	20.53
40	40	30	166.58	87.93	62.89	47.32	20.52
		40	296.05	168.66	124.50	96.57	49.07
		50	460.85	242.71	173.58	130.28	56.25
50	50	30	326.01	195.06	140.80	116.08	55.52
		40	578.87	301.81	209.16	160.71	74.47
		50	902.23	470.16	326.19	249.53	116.07

Figure 8: Times in seconds for matrix multiplication on AP3000, by matrix size, ALI size and number of processors

Rows	Cols	ALI size	Processors 1	3	4	8	16
30	30	30	1.0	1.98	2.48	3.77	9.12
		40	1.0	1.99	2.41	3.73	9.37
		50	1.0	1.98	2.49	3.82	9.45
40	40	30	1.0	1.89	2.65	3.52	8.12
		40	1.0	1.76	2.38	3.07	6.03
		50	1.0	1.90	2.65	3.54	8.19
50	50	30	1.0	1.67	2.32	2.81	5.87
		40	1.0	1.92	2.77	3.60	7.77
		50	1.0	1.92	2.77	3.63	7.77

Figure 9: Speedup for matrix multiplication on AP3000, by matrix size, ALI size and number of processors

processors) are shown in Figure 9. Speedups for all matrix size cases are broadly consistent as the ALI size increases when both maps are sequential (1 and 3 processors) and when the outer map is parallel (4 processors). When both maps are parallel (8 and 16 processors), there is consistent speedup in all 30*30 cases. There are anomalies with 40 digit ALIs for 40*40 and with 30 digit ALIs for 50*50 matrices which require further investigation. Overall, speedups are roughly $N/2$ for N processors.

Figures 10 and 11 show times and speedups respectively for 50*50 matrices on up to 32 processors, where for 8, 16, 24 and 32 processors nested parallelism in the inner map may or may not be exploited. For up to 24 processors, the unnested implementation is consistently better for times and speedups. This is not surprising. Matrix multiplication is highly regular with little additional processing beyond the row/column multiplications. In the nested case, the processors for the outer skeleton act only to distribute data to, and receive data from, the inner skeletons,

Nesting	Processors 1	3	4	8	16	24	32
nested	1299.39	676.40	468.93	359.03	166.81	140.54	84.42
unnested	1299.53	677.49	469.68	235.84	158.02	80.35	81.17

Figure 10: Times for matrix multiplication on AP3000, of 50*50 matrices of 60 digit ALI by nesting and number of processors

Nesting	Processors 1	3	4	8	16	24	32
nested	1.0	1.92	2.77	3.62	7.79	9.25	15.29
unnested	1.0	1.99	2.77	5.51	8.22	16.17	16.00

Figure 11: Speedup for matrix multiplication on AP3000, of 50*50 matrices of 60 digit ALI by nesting and number of processors

thus adding to the communication overhead with no decline in the processing overhead. For 32 processors, the times and speedups are similar.

5 Conclusions

We have discussed the design of the spine of our HOF/skeleton based parallelising compiler, and given an overview of the mechanisms that enable the realisation of parallelism in nested HOFs through nested skeletons. We have also provided a practical demonstration of the compiler spine through the implementation of matrix multiplication with arbitrary length integers, from a Standard ML program, on a Fujitsu AP3000.

The times for matrix multiplication are not particularly impressive. However, this example shows that:

- the compiler spine can successfully nest algorithmic skeletons;
- generated parallel code achieves respectable, consistent speedups;
- the skeletons may be run in sequential and parallel modes.

Work on the compiler continues. Our next objectives are to:

- develop the profiler and analyser, to enable identification of useful parallelism in HOFs and optimal process/processor allocation;

- integrate proof planned HOF synthesis with the compiler;
- test the system on a wider range of more substantial examples.

Acknowledgements

This work is funded by EPSRC Grant GR/L42889. We wish to thank Harlequin Ltd and the Imperial College Fujitsu Parallel Computing Research Centre for their support. Major compiler components were developed by the SML Groups at the University of Edinburgh (ML Kit 1), the University of Copenhagen (ML Kit 2) and the Australian National University (Fujitsu AP1000 MPI), and by the Caml Group at INRIA (Objective Caml).

We wish to thank our colleagues Mohammad Hamdan and Robert Pointon for their helpful comments on this paper.

References

[1] J. Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[2] R. Bird and O. de Moor. *Algebra of Programming.* Prentice-Hall, 1997.

[3] M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* Pitman, 1989.

[4] M. Cole. Algorithmic Skeletons. In K. Hammond and G. Michaelson, editors, *Research Directions in Parallel Functional Programming*, pages 289–303. Springer-Verlag, to appear 1999.

[5] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. *International Journal of Supercomputer Applications and High Performance Computing*, 8(3/4), 1994.

[6] G. Michaelson, A. Ireland, and P.King. Towards a Skeleton Based Parallelising Compiler for SML. In C. Clack, T. Davie, and K. Hammond, editors, *Proceedings of 9th International Workshop on Implementation of Functional Languages*, pages 539–546, Sept 1997.

[7] D. Skillicorn. *Foundations of Parallel Programming.* Number 6 in Cambridge International Series on Parallel Computation. CUP, 1994.

[8] Guy E. Blelloch and John Greiner. A Provable Time and Space Efficient Implementation of NESL. In *ACM SIGPLAN International Conference on Functional Programming*, pages 213–225, May 1996.

[9] C.B. Jay, M.I. Cole, M. Sekanina, and P.A. Steckler. A monadic calculus for parallel costing of a functional language of arrays. In C. Lengauer, M. Griebl, and S. Gorlatch, editors, *Euro-Par'97 Parallel Processing*, volume 1300 of *Lecture Notes in Computer Science*, pages 650–661. Springer, August "1997".

[10] H. W. Loidl. *Granularity in Large Scale Parallel Functional Programming.* PhD thesis, Dept of Computing Science, University of Glasgow, April 1998.

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

[12] R. Rangaswami. A cost analysis for a higher order parallel programming model. In *PhD thesis.* Department of Computer Science, University of Edinburgh,, February 1996.

[13] D. B. Skillicorn and D. Talia. Models and Languages for Parallel Computation. *Computing Surveys*, June 1998.

[14] A. Bundy. The use of explicit plans to guide inductive proofs. In R. Lusk & R. Overbeek, editor, *Proceedings of 9th Conference on Automated Deduction*, pages 111–120. Springer-Verlag, 1988.

[15] 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.

[16] 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.

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

[18] 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.

[19] S. Pelegatti. *Structured Development of Parallel Programs*. Taylor and Francis, 1998.

[20] 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.

[21] N. Scaife, P. Bristow, G. Michaelson, and P. King. Engineering a parallel compiler for SML. In C. Clack, T. Davie, and K. Hammond, editors, *Proceedings of 10th International Workshop on Implementation of Functional Languages*, pages 213–226, University College London, Sep 1998.

[22] L. Birkedal, N. Rothwell, M. Tofte, and D. N. Turner. The ML Kit (Version 1). Technical Report 93/14, Department of Computer Science, University of Copenhagen, 1993.

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

[24] X. Leroy. *The Objective Caml System, available from http://pauillac.inria.fr/ocaml/*. INRIA, 1996.

[25] G. Cousineau, P. L. Curien, and M. Mauny. *The Categorical Abstract Machine*, volume 201 of *LNCS*, pages 50–64. Springer Verlag, 1985.

[26] J. C. Reynolds. Definitional Interpreters for Higher-order Programming Languages. *ACM National Conference*, pages 717–740, 1972.

[27] J. M. Bell, F. Bellegarde, and J. Hook. Type-driven defunctionalization. In *Proceedings of the ACM SIGPLAN ICFP '97*, pages 25–37. ACM, Jun 1997.

[28] J. Serot. Embodying Parallel Functional Skeletons: An Experimental Implementation on Top of MPI. In C. Lengauer, M. Griebl, and S. Gorlatch, editors, *Euro-Par'97 Parallel Processing*, volume 1300 of *Lecture Notes in Computer Science*, pages 629–633. Springer, August "1997".