A Parallel SML Compiler based on Algorithmic Skeletons

Norman Scaife, Susumu Horiguchi
School of Information Science,
Japan Advanced Institute of Science and Technology,
1-1 Asahidai, Tatsunokuchi, Nomigun, Ishikawa 923-1292, Japan
(*e-mail:* {norman,hori}@jaist.ac.jp)

Greg Michaelson, Paul Bristow
Department of Computing and Electrical Engineering,
Heriot-Watt University, Riccarton, Edinburgh, EH14 4AS, United Kingdom
(*e-mail:* {greg,paul}@macs.hw.ac.uk)

Abstract

Algorithmic skeletons are abstractions from common patterns of parallel activity which offer a high degree of reusability for developers of parallel algorithms. Their close association to higher order functions (HOFs) makes functional languages, with their strong transformational properties, excellent vehicles for skeleton-based parallel program development. However, using HOFs in this way raises substantial problems of identification of useful HOFs within a given application and of resource allocation on target architectures.

We present the design and implementation of a parallelising compiler for Standard ML which exploits parallelism in the familiar `map` and `fold` HOFs through skeletons for processor farms and processor trees, respectively. The compiler extracts parallelism automatically and is target architecture independant.

HOF execution within a functional language can be nested in the sense that one HOF may be passed and evaluated during the execution of another HOF. We are able to exploit this by nesting our parallel skeletons in a processor topology which matches the structure of the Standard ML source. However, where HOF 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.

We describe the analysis leading from input Standard ML through HOF instantiation and back-end compilation to an executable parallel program. We also present an overview of the runtime system and the execution model. Finally, we give parallel performance figures for several example programs, of varying computational loads, on the Linux-based Beowulf, IBM SP/2, Fujitsu AP3000 and Sun StarCat 15000 MIMD parallel machines. These demonstrate good cross-platform consistency of parallel code behaviour.

1 Introduction

1.1 Algorithmic Skeletons

Parallelism brings additional complexity to programming. As well as concerns of algorithms and data structures, exploitation of parallelism requires understanding of the subtleties of the interaction of independent processors, and how these are affected by underlying interconnection topologies and technologies.

Most extant languages lack appropriate abstractions for parallelism and parallelism-oriented languages such as Occam have failed to gain widespread acceptance. Typically, constructing parallel programs involves the use of libraries of inter-process communication constructs like PVM and MPI within a mainstream language. However, such libraries provide relatively low-level facilities, requiring deep programmer knowledge of implementation and application specific characteristics relating to communication and processing behaviours. Parallel facilities are relatively scarce and costly, so there is a strong temptation to craft application specific components to optimise performance on a particular platform, losing both genericity and portability.

Algorithmic skeletons offer useful coarse grain abstractions for parallel programming, and have been an active area of research since the term was coined by Cole in 1989 (Cole, 1989a). Essentially, a skeleton is a template for generic parallel activity.

For example, in task farming, a set of data is distributed amongst multiple processors which apply the same process to individual data items. The processed data items are then collected together to form a new data set. A task farm skeleton offers an inter-processor communication structure which enables a farmer processor to distribute data to and collect data from worker processors, all of which are runnning a common process. This structure is independent of the types of the data and the applied process, and parameterised on the number of available processors. In a program, the skeleton is instantiated with a specific number of processors, with type specific communication code and with a type specific process. At load time, one available processor is designated as the farmer, and given responsibility for overall coordination. The other processors are designated workers and are all initialised with the farmed process. At run time, the farmer and workers interact through the communication structure.

As well as genericity, skeletons offer a number of other advantages. It is relatively straightforward to construct accurate cost models for skeletons, parameterised on the number of processors, communication characteristics and processing cost for the farmed process. These may be used to identify whether or not a program offers exploitable parallelism at sites of potential skeleton use. When constructed using mature communications libraries, skeletons offer considerable cross platform portabilty with highly consistent performance.

1.2 Skeletons and Higher Order Functions

While novel languages supporting skeletons have been developed, for example Pelagatti's P3L (Pelegatti, 1998), they have, in common with novel parallel languages, yet to gain widespread acceptance. Approaches based on accomodating skeletons in extant languages have proved more popular, especially through their use as a coordination layer for skeletons. Here, considerable success has been obtained through the use of functional languages because of the close correspondence between skeletons and higher-order functions (HOFs).

Functional languages, like Standard ML and Haskell, lack side effects and have the Church-Rosser property of evaluation order independence. In principle, this makes them good vehicles for parallel programming. However, much latent parallelism in functional programs is too fine grain to be exploitable. HOFs appear to offer an appropriate level of granularity both for identifying and for exploiting parallelism.

HOFs are usually polymorphic functions that apply an argument function across all elements of a data structure. The best known are those for list processing, for example `map`:

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

which applies an argument function `f` to each element of a list:

```
map ff [e1, e2, ... eN]
  ==> [(ff e1),(ff e2), ... (ff eN)]
```

and `fold` (shown here in the right to left form `foldr`):

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

which applies a function `f` between the elements of a list:

```
foldr f b [e1, e2, ... eN]
  ==> f e1 (f e2 ... (f eN b) ...)
```

A `map` may be realised as a task farm, discussed above. Similarly, a `fold` for an associative and commutative argument function may be realised as a divide-and-conquer network. Typically, each processor splits the list and passes the sub-lists down to sub-processors. The leaf processors apply the function `f` to list elements and the base parameter `b`, and each intermediate processor then applies `f` to the results from its sub-processors.

The major benefit of associating skeletons with HOFs in a functional framework is that formal reasoning techniques, such as Backus' FP (Backus, 1978) and the Bird Meertens Formalism (Bird & de Moor, 1997), may be employed to transform programs whilst preserving meaning. Furthermore, transformations on HOFs have equivalent meaning-preserving transformations on skeletons. Given cost models for skeletons, transformations may be used to try and restructure programs to optimise parallelism.

There are two approaches to realising skeletons in functional languages. One is to write the skeletons in the functional language, implementing multi-processor behaviour using language constructs, for example channels in Clean (Kesseler, 1995), Eden (Klusik *et al.*, 2000) or Haskell (Herrmann & Lengauer, 2000), or using wrappers for library functions, for example MPI wrappers with Caml (Serot, 1999).

The other approach is to write the skeletons in another language, to identify HOF use in functional programs at compile time and replace them with the equivalent skeleton calls. That is, the functional language forms a coordination layer for skeletons. For example, Darlington's group (J. Darlington & To, 1996) use the Hope-like language SCL, Serot and Coudarcher's SKiPPER system uses Caml (Coudarcher *et al.*, 2001) and our PMLS (Parallel SML with Skeletons) compiler is based on Standard ML (Michaelson *et al.*, 2001).

A full treatment of parallel functional programming in general may be found in (Hammond & Michaelson, 1999).

1.3 Background

1.3.1 Parallel implementation from functional prototypes

Our research has its roots in joint work with the Computer Vision group at Heriot-Watt University, from 1987 to 1996, using Standard ML to prototype parallel vision algorithms for implementation in occam2 on a Meiko Computing Surface. We quickly became aware of Cole's work into algorithmic skeletons at nearby Edinburgh University (Cole, 1989b) and focused on using higher order functions (HOFs) as loci of potential parallelism, to be realised using a skeletal harnesses.

During this period, we explored the parallelisation of low-, intermediate- and high-level image processing algorithms. Low-level examples included the Canny edge detector (Canny, 1986; Koutsakis, 1993), where several implementations using a parallel `map` were investigated, and HK segmentation (Besl & Jain, 1986; Clarkson, 1994) where combinations of `map` and `fold` were used. Intermediate-level algorithms such as Hough transforms (Leavers, 1993; Michaelson & Scaife, 1995) and high-level algorithms such as the Perspective Inversion Algorithm (McAndrew & Wallace, 1989; Wallace *et al.*, 1992) used pipelines of `maps` which could be analysed for optimal load-balancing.

The success of these individual investigations led to the development of a complete intensity based object recognition system built from the Canny, Hough and PIA prototypes (Michaelson & Scaife, 1995) This in turn informed a more ambitious system whereby a complete parallel object recognition system from multiple data sources was implemented (Scaife *et al.*, 1997; Wallace *et al.*, 1998). This used four skeletons specialised for use in Computer Vision applications; Geometric Decomposition (GD), Data Farming (DF), Task Farming (TF) and GD with Interworker Communications (GDIW). These were sufficient to build the complete system with a high degree of code reuse but it is unclear as to how general this set of skeletons are for Computer Vision in a wider context. Coincidentally with this work, a method of performance prediction was developed to guide future prototype development (Scaife, 1996).

Although our methods worked well in practice we were hampered by having to hand-code both the prototypes and the final implementations where, in theory, the implementation could be made completely automatic.

1.3.2 Parallelising compilation

Our group's development of parallel compilers for functional languages also goes back to the early 1990s. Firstly, the PUFF compiler was developed by Busvine to implement linear recursive functions in parallel on a Meiko Computing Surface (Busvine, 1993). PUFF used sequential SML interpreter times to predict processing costs and to allocate processes to processors. It identifies linear recursion directly rather than through higher order functions and using Johnsson's lambda-lifting, and higher-order function removal, to expose it.

This work was extended and generalized by Bratvold into a prototype parallel compiler for SML using skeletons, again targetting Transputer systems (Bratvold, 1994). This realised un-nested `map`, `filter` and `fold` functions in parallel through skeletons with associated performance models. Prototypes are instrumented through a sequential interpreter based on the SML Structural Operational Semantics. While SkelML uses Busvine's Puff compiler to generate Occam, it is assummed that all higher order function calls are fully saturated and that arguments lack free values.

Finally, Hamdan's EKTRAN parallel compiler (Hamdan *et al.*, 2000) was developed to investigate nesting of higher-order parallel constructs. It is based on a simple FP-like functional notation to coordinate nested skeletons identified through `map` and `fold`. Ektran uses inlining to expose all higher order function use in a program. A simple scheme is then applied to identify single free variables in higher order function arguments for transmission to skeletons at run time. For nested skeletons, each layer passes free variables to the next. Hamdan discusses an unimplemented extension of this scheme to handle multiple free variables with functions.

Combining the work into skeletons for Computer Vision and the three prototype parallel functional compilers led to the design of the system described in this paper. We would handle a greater proportion of the SML language, where all the previous systems implemented restricted subsets, and we would target general-purpose parallel machines. In addition, the embryonic profiler-guided performance modelling would be built into the compiler and used to drive a transformation system and automated theorem-proving techniques would be used to generate valid sets of transformations. The system described in this paper represents the current state of this compiler.

For a more detailed description of these developments, see (Michaelson & Scaife, 2002).

1.4 PMLS

PMLS is a parallelising compiler for a pure functional subset of Standard ML in which the sole source of parallelism is indicated by specific HOFs. These are translated into parallel algorithmic skeletons implemented in native code linked with an MPI library (Message Passing Interface Forum, 1994).

In developing the PMLS compiler, we have focused upon `map` and `fold`. There is considerable evidence that these HOFs, along with function composition, suffice for a very wide range of typical processing requirements, especially for parallelism (Hammond & Michaelson, 1999). In particular many common processing patterns can be reformulated as homomorphisms based on `map` and `fold` (Cole, 1993; Hu *et al.*, 1997). Furthermore, there are approaches to developing programs through transformed `map/fold/compose` combinations (Bird & Wadler, 1988). Finally, there are well elaborated cost models for these functions (Hamdan, 2000).

For example, we can implement `filter` which selects list elements according to some criteria specified by a predicate:

```
fun filter p [] = []
  | filter p (h::t) = if p h
                      then h::filter p t
                      else filter p t
```

We instantiate `map` with the predicate to give a list of boolean values which can be used to select sequentially from the original list with `fold`:

```
fun pfilter p list =
  let val tfs = map p list
      fun ff (tf,e,l) =
        if tf then e::l else l
  in foldl ff [] (tfs,list) end
```

While there is a slight overhead compared to, say, a direct implementation of `filter` in C and MPI, this approach enables us to combine the benefits of coordination and direct implementation.

Using skeletons constructively in this manner requires the ability to handle skeletons as arguments to other skeletons. This is termed *nesting*. In theory, nested HOFs should lead to nested parallelism but this is known to be a difficult problem raising many program analysis and resource allocation issues (Rangaswami, 1995). Our compiler has the ability to *arbitrarily* nest, to any depth, the `map` and `fold` skeletons that we have implemented.

This paper concentrates on our compiler's mechanisms for handling these two skeletons and explores problems of topological analysis and data locality that arise from the presence of such constructs within a strict functional language. A method for extracting the network topology from arbitrary SML code is presented. We show how functional and non-functional free values can be propagated down the skeleton hierarchy to the point of instantiation using the two different but complementary methods of lambda-lifting and defunctionalization. We also give a brief summary of our runtime system. Finally, we describe the development of several example programmes using the `map` and `fold` skeletons and give the performance of our compiler on a range of MIMD parallel machines.

2 Compiler overview

The initial design of the compiler has been presented in (Michaelson *et al.*, 1997) and (Scaife *et al.*, 1998). A preliminary version of the compiler is described in (Michaelson *et al.*, 2001).

We decided to use extant software as far as possible for the front (elaboration) and back (code generation) ends of our compiler. This would enable us to focus on program analysis and transformation, rather than attempting to build yet another SML elaboration and code generation system from scratch.

We reviewed many languages as candidates for the backend compiler. The only parallel machine available to us at the time was the Fujitsu AP1000 at Imperial College - a Unix-based machine with 32Mb per node. Unfortunately, the leading candidate, SML of New Jersey, required more than 32Mb of memory for efficient execution. This problem was encountered by the developers of paraML (Bailey & Newey, 1993) which required extensive modifications to NJSML. Following the success of the Pict language (Pierce & Turner, 2000) which used Objective Caml as the host language we decided to use use Objective Caml (Chailloux *et al.*, 2000) as our backend. This is a relatively lightweight implementation of ML with low memory requirements and a bi-directional C language interface, allowing us to develop our skeletons in C, linked with MPI libraries. Other strengths include the compactness of the binaries and the modest memory footprint.

For the front end elaborator we needed an implementation of SML which provided an SML intepreter capable of being modified to gather dynamic profiling information. The interpreter used by SkelML (Bratvold, 1994) had some deficiencies and the ML Kit (Birkedal *et al.*, 1993) provided an interpreter up to Version 2, written in SML. Although this was relatively slow it would admit easy modification for our compiler.

The combination of the ML Kit with translation into Objective Caml for backend compilation has proved a very effective combination.

2.1 Compiler modules

The structure of the compiler is shown in Fig. 1 and is organised as a series of modules:

ParseElab reads the input SML and lexically analyses, parses and *elaborates* the SML prototype.

Network takes an elaborated syntax tree and extracts a description of the *topography* of recognised HOFs from the program. We define the topography to

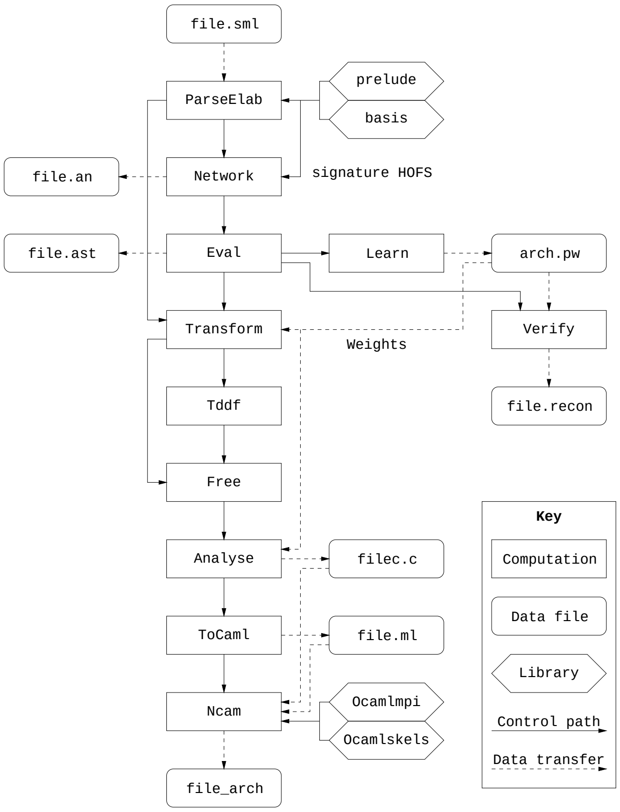

Fig. 1. The PMLS compiler modules

be the relationship between HOFs in the source code and between skeletons in the executing program. This module uses the type information generated by the elaborator.

Tddf is an optional module which attempts to defunctionalize (Bell *et al.*, 1997) the entire program. Our run-time system does not have the ability to transmit functional values so closures are represented by `datatype` values which allows us to transmit partially-applied functions at run-time.

Free performs free-variable analysis and modifies functions by adding their free-variables as arguments. This ensures that free-variables are present at the point of calling a skeleton and are available for transmission at run-time. An attempt is made to limit the transmission of such values to those which will actually be required for remote computation. Note that if defunctionalization is performed then free-variable analysis is not required since free values are bound up in the `datatype` representations.

Analyse is the back-end processing required to transform nominated HOFs into skeleton calls and add code to launch the skeletons at runtime. This involves registering functions with the runtime system, passing free values to the runtime system and also reconstructing the types of skeleton results.

ToCaml translates `Core` SML into Objective Caml. This is mostly a direct conversion between semantic entities but requires some compilation support to iron out minor differences, particularly between the two type systems.

Ncam compiles the output into an executable file. This is not a module in the main compiler but is actually a Perl script which calls the appropriate Objective Caml and C compilers, generating the correct compiler flags and library options.

The modules **Eval**, **Learn**, **Verify** and **Transform** form the core of our performance modelling and transformational system:

Eval is an SML interpreter which closely corresponds to the SML definition. This generates lists of semantics rules used in the evaluation.

Learn and **Verify** attempt to relate the sequential runtime of the compiled code with these rule counts.

Transform is a simple program transformation system which will eventually be driven by the performance prediction. Currently, we lack parallel performance models for our skeletons to allow parallel performance improvements. This feature is not discussed further in this paper and is left as a potential enhancement of our compiler.

The main objective of this work is to build a compiler for which all parallelism is *implicit*. Our use of algorithmic skeletons allows us to partially achieve this goal. On the one hand the selection and parallel implementation of the higher-order functions is automatic. On the other hand the programmer is constrained to use the particular set of HOFs provided by our compiler.

2.2 Runtime system overview

At run-time, we use an SPMD approach based on MPI Version 1 (Message Passing Interface Forum, 1994). Here, an executing program is a set of identical programs running on a fixed set of processors for the lifetime of the execution. Initially, before any skeleton functions are reached, all processes execute the same code. When the first skeleton is encountered, one processor becomes a “master” processor and the others become part of a pool of “worker” processors. Worker processors then receive work from the master and return partial results until the skeleton finishes execution. At this point, the master’s skeleton returns the actual result of the computation.

The processor hierarchy is managed by a generic skeleton management module called *pskel* which has access to the network information derived by our static analysis and co-ordinates skeleton initialisation and termination. This module is called by all skeleton instances in the executing program and, after initialisation, calls the correct master or worker code for the appropriate skeleton. The network information is used to decide the position of the skeleton in the HOF hierarchy, the position of the current processor in the processor network, and the correct data and instance functions registered in the Objective Caml code. If a nested skeleton is encountered during the execution of a sequential instance function the *pskel* module is re-entered. This requires the use of state variables to prevent duplicated transmission of data and to correctly organise initialisation and termination of nested skeletons and processors.

The `pmap` skeleton executes as a processor farm with one master processor and the remaining processors as workers. Since our skeletons are list-based the unit of granularity is a fixed-size block of list elements. Currently, the block size is set as a global constant with the value set at runtime. The `pmap` skeleton can be run in a geometric decomposition mode by the expedient of setting the number of blocks to the number of processors participating in the `pmap` instance.

The `pfold` skeleton operates as a binary tree of processors. Here, all non-leaf processors operate as nested masters where a distinction is made between the *root* master for the `pfold` skeleton and a *parent* processor for a node in the processor tree. Parallel nesting is implemented solely from the leaf processors. If a nested instance is encountered from an intermediate node the skeleton is executed sequentially.

Both of these skeletons use the Objective Caml `Marshal` module for linearising data. This works on polymorphic values and is platform independent but is not a very compact representation of the data. For instance, lists have an overhead per list element. Since there are already significant latencies associated with initialising the processor hierarchy plus relatively high latencies associated with our use of MPI as a communications layer, our skeletons are intended for coarse-grain parallelism only. Another limitation of these skeletons is that the master-worker model tends to cause a bottleneck in communications at the master processor. This is less pronounced for `pmap` since it is naturally load-balanced by the farming mechanism but is acute for the `pfold` skeleton since there is the additional overhead of folding the partial results at each node. Parallelism is only observed for the `pfold` skeleton if successive applications of the instance function results in significantly less work. This precludes its use in applications such as a mergesort where there is no reduction in processing time at successive nodes.

Our skeletons are implemented in C using the Objective Caml foreign function interface to access Objective Caml functions on remote processors. Thus, the parallel map skeleton `pmap` is implemented by shadowing the sequential `map` function replacing instance functions with registered names:

```
val _ = register "ff" ff
val result = pmap "ff" [1,2,3]
```

Note that the type information has to be reconstructed at the `pmap` call, so the `map` identifier is actually replaced by:

```
(pmap:string -> int list -> int list)
```

The function `register` is provided by Objective Caml as part of the foreign function interface. This registers Objective Caml values under string names and an access function is provided by the runtime system allowing these values to be accessed within C functions. For our purposes we have created an access function called `named_value` in Objective Caml allowing registered values to be returned. Note that this interface is not type-safe but since our executing code is derived from type-checked SML using semantics-preserving transformations this is not an issue for our compiler.

The next section discusses the extraction and use of an abstract network to guide parallelisation, and the free variable analysis and defunctionalisation approaches to free variable lifting. It also provides more detailed overviews of the compiler front and back ends.

3 PMLS structure

Here we give more detailed descriptions of the compiler modules outlined in Fig. 1. The compilation process is illustrated for a trivial SML program:

```
val x = 1
fun ff (y,z) = x + y + z
val result = fold ff (~x) [1,2,3]
```

This example uses a single non-nested instance of the `fold` skeleton instantiated with a function, `ff`, which has a free variable, `x`. Note that the default value for the `fold` function has to be an identity for the instantiation function if the parallel implementation is to return the same result as the sequential implementation. The value `~x` is an identity for `ff`.

3.1 `ParseElab`: Front-end and library support

The ML Kit (Birkedal *et al.*, 1993) was chosen for the front-end, principally because it provides a full *Evaluator* for the language which is a direct implementation of the dynamic semantics. The ML Kit itself is implemented in SML and the parser generates a syntax tree in the *Core* language of the SML definition (Milner *et al.*, 1990), a restricted form of SML where all *derived forms* have been substituted for Core expressions. The simple example above is parsed into the following:

```
val x = 1
val rec ff = fn (y,z) => ((op +) ((op +) (x,y),z))
val result = fold  ff (~x) ((op ::) (1,(op ::) (2,(op ::) (3,[]))))
```

This code is then elaborated and the syntax tree is annotated with the derived type information. Together with the parser, this forms the `ParseElab` component in Fig. 1.

Our compiler is a "whole program" compiler and as such needs access to the sources for the standard prelude during compilation. The prelude itself is prepended to the input code by the compiler executive. There is a complication in that both the compiled and interpreted versions have to be able to execute the same code. This is handled by defining prelude functions in terms of the primitive operator, the `prim` function, used by the ML Kit interpreter to introduce prelude functions. This has type: `int * 'a -> 'b` where the `int` is a function code value and `'a` encodes the arguments and exceptions used by the operator. The result then has to be cast into the return value's type. This is shared by the runtime system for which the `prim` function has been implemented in Objective Caml, although for efficiency, native Objective Caml equivalent functions are inserted between the prelude and the main body of the code for compiled output.

3.2 `Network`: 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 defines the relationship between the HOFs in the sequential prototype and between executing skeletons in the parallel version.

The analysis works by annotating the type information in the syntax tree generated by the `ParseElab` module with information concerning the potential activation of skeletons in the creation of objects with those types. This is implemented by the `Network` phase in Fig. 1.

3.2.1 Skeleton nomination

Skeletons are identified solely by the names of the sequential HOFs and the names of the functions implementing the parallel equivalent. These names are externally defined. Nomination of skeleton HOFs is determined by an SML `signature` called `HOFS` provided by the program being compiled, usually as part of the prelude.

In theory, this allows programmers to define their own skeletons by nominating HOFs and providing parallel equivalents. In practice, there are strong constraints upon which types of functions can be used as skeletons. This is a major area for future development and not considered further here.

The `HOFS` signature contains pairs of value definitions which are matched such that the first definition nominates the sequential SML HOF and the second gives the SML function corresponding to the parallel skeleton. The two must have the same type signature but with abstraction points in the HOF (e.g. the functional argument in the `fold` HOF) being replaced by `string` types in the parallel equivalent. This allows the skeleton instantiation function to be registered under a `string` name such that the runtime system can access the skeleton function through this name. Consider:

```
signature HOFS =
sig
 val map  : ('a->'b) -> 'a list -> 'b list
 and pmap : string   -> 'a list -> 'b list
 val filter : ('a->bool) -> 'a list -> 'a list
 and pfilter : string    -> 'a list -> 'a list
```

```
 val fold  : ('a*'a->'a) -> 'a -> 'a list -> 'a
 and pfold : string      -> 'a -> 'a list -> 'a
end
```

This `signature` specifies three skeletons, `map`, `fold` and `filter` which are to be replaced by `pmap`, `pfold` and `pfilter`, respectively. The types of the HOFs are used in the skeleton identification process by comparison with the types of their instances.

3.2.2 Abstract network description

The abstract network is described by the following datatype:

```
datatype absnet =
  base
| node of {hof:string, hofAn:absnet,
           nest:absnet list, inst:skelinfo ref}
| seq of absnet list
| alt of absnet list
```

Here, `base` indicates no skeleton instances, `node` indicates an instance of a (potentially nested) skeleton, `seq` indicates a set of skeleton instances where each skeleton must terminate before the next one can start and `alt` is skeleton alternation. Note that a `seq` means an unspecified (zero to potentially infinite) number of repetitions of the skeleton instances in the sequence since this is how recursion is represented in the network.

Note that at present we cannot handle alternation in our current compiler and so each branch of a *match*[1] construct must either have the same structure (eg. single `map` or nested `map map`) as all the other branches or have no skeleton instances at all. Skeletal parallelism in the presence of branches is a major research topic in its own right, involving deep questions to do with optimal process/processor allocation, data movement minimisation and dynamic processor reuse.

3.2.3 Abstract network extraction

Abstract network extraction is divided into two operations:

1. Scan the elaborated AST converting SML types into abstract network types.
2. Strip extraneous type information leaving an abstract network description.

An abstract network type is an SML type with additional HOF information, in the form of the `absnet` datatype attached to the type information. The first phase requires maintaining a context with correct scoping of identifiers bound in value definitions.

[1] Here the term *match* refers to the *match* grammar object in the Standard ML definition (Milner *et al.*, 1990). In the `Core` language, this is the only source of conditional execution.

```
datatype Type = CONSTYPE of Type list * TypeName (* ['a] list, [] int *)
              | RECTYPE of RecType               (* {a:int,b:int} *)
              | ARROW of Type * Type             (* int -> int *)
              | TYVAR of TyVar                   (* 'a *)

and RecType = NILrec                             (* {} (also unit) *)
            | VARrec of RowVar                   (* { ... } *)
            | ROWrec of Lab * Type * RecType     (* a:int, etc. *)
```

Fig. 2. Semantics types in the ML Kit

In addition, a simplified form of type unification is required over abstract network types to allow deduction of abstract network types for *function applications*, *function cases*, *constructor matching* and *explicitly typed expressions*.

The result is a context with a set of mappings from bound variables to abstract network type expressions: $\{v1 \mapsto n1, \ldots, vn \mapsto nn\}$ and a modified syntax tree with network information alongside the equivalent type information for expressions only[2].

3.2.4 Abstract network type

After removal of extraneous intermediate data, semantics types in the ML Kit can be represented by the datatype in Fig. 2. `TypeName` and `Lab` are string names. `TyVar` is a referenced type instance, updated during elaboration and `RowVar` is a referenced record instance. These should be fully instantiated if elaboration succeeds.

Since we need to shadow the type information during the derivation of abstract network types the abstract network datatype is an augmented version of the ML Kit datatype, represented as in Fig. 3. The `errorAn` entry is used to store error messages during abstract network derivation. These are intended to be extracted (and transformed into `baseAn`) during the second phase of the algorithm. The `nodeAn` entry has several fields:

`hof` is the name of the identified HOF.

`hofAn` stores the type of the node and is used during function application and constructor dereference.

`nest` contains the types of arguments of the original HOF definition which have been applied. These are needed during the second phase.

`inst` is a `skelinfo` datatype and is populated during later analysis to contain the names of the registered functions and information about processor allocation.

[2] *dec*, *pat* etc. do not have types.

```
datatype absnet =
  errorAn of string                    (* network algorithm failure *)

(* network info *)
| baseAn                               (* base values *)
| nodeAn of {hof:string,               (* name of HOF *)
             hofAn:absnet,             (* abstract network for bare HOF *)
             nest:absnet list,         (* list of nested constructs *)
             inst:skelinfo ref option} (* referenced skeleton instance *)
| seqAn of absnet list                 (* sequence of skeletons *)
| altAn of absnet list                 (* alternation of skeletons *)

(* shadow the type system *)
| constypeAn of absnet list * string
| rectypeAn of recAn
| arrowAn of absnet * absnet
| tyvarAn of string

(* record types *)
and recAn =
  nilAn
| varRecAn of string
| rowRecAn of string * absnet * recAn
```

Fig. 3. `datatype absnet`: Abstract network type representation

3.2.5 Scanning the AST

This is the first phase of the algorithm, based on a traversal of the SML syntax tree, converting SML types into abstract network types. The most important part is the identification of nominated HOFs, by the `signature HOFS` mechanism, and "wrapping" a `nodeAn` around the SML `Type` at the point the node is introduced. The rest of the algorithm is a pass over the AST, deriving abstract network types from the `Type` information deduced by the elaborator and applying (or unapplying) `absnet`s where appropriate to generate the right combination of abstract network types for all *valbinds* [3] in the program.

Since we are maintaining an `Environment` NE from `Identifier` to `absnet` types: $NE \in VId \xrightarrow{fin} \texttt{absnet}$

we need a routine to lift identifier bindings from $pat \times$ `absnet` pairs:

$$\texttt{patabsnets} : pat \times \texttt{absnet} \to (\texttt{string} \times \texttt{absnet})\ \texttt{list}$$

This routine needs to make the distinction between value identifiers and constructors, although, in general, the overall algorithm does not. While the elaborator

[3] value bindings.

implements identifier resolution, constructors in patterns need to be inverted to get the correct abstract network type for the constructed (and bound) identifiers. It is possible for nullary constructors to appear in *pats*[4], in which case no identifiers are bound. This also applies for exception constructors.

Otherwise, the routine scans the *pat* and `absnet` simultaneously matching constructors, record labels and generating additional bindings for layered *pat*s. Suitable handling of local declarations allows us to maintain a `Context` containing the network environment, a single lookup routine is provided:

$$\texttt{lookupId} : \texttt{Context} \rightarrow \texttt{string} \rightarrow \texttt{absnet}$$

Fig. 4 outlines a schema, $\mathcal{T}$, which implements the main tree traversal. Note that we do not make any distinction between identifier forms that the SML definition treats as distinct i.e. long variables, constructors, exception constructors and identifiers are all treated as long identifiers. These are resolved, when necessary, by looking up the elaboration information.

Some notational liberties are taken here to simplify presentation. The arguments to the $\mathcal{T}$ schema are AST grammar objects and the parameters of some of the objects have been omitted for brevity. The latter can be found in the definition (Milner *et al.*, 1990).

The $\mathcal{T}$ schema calls several other schemas. Schema $\mathcal{M}$ is a type unification algorithm for abstract network types. Schema $\mathcal{S}$ applies the abstract network type subsitutions generated by $\mathcal{M}$ to grammar objects. Schema $\mathcal{A}$ is a combination of $\mathcal{M}$ and $\mathcal{S}$ which gives the result of applying one abstract network type to another. Schema $\mathcal{N}$ converts an SML type into an abstract network type without adding any `absnet` information.

In handling exceptions, nullary *excon*s can cause problems with later processing since nullary constructors are all ground to `baseAn`. This is circumvented by turning `exn`[5] into a *tyvar*[6] which is unified by the $\mathcal{T}$ algorithm, later during the traversal.

In handling sequenced objects it is necessary to preload the `Context` with bindings from all objects in the sequence prior to traversal. The `absnet`s for these bindings are merely derived from the semantics types of the objects. This applies to *valbind*s.

Patterns do not, as such, have abstract network types and are not included in the $\mathcal{T}$ schema. These are scanned separately by the `patabsnets` routine, giving `absnet`s for bound identifiers.

Syntax types are handled in exactly the same manner as semantics types.

[4] patterns.
[5] exception.
[6] type variable.

$\mathcal{T}$: ast → absnet			
$\mathcal{T}$ *scon*	=	baseAn	
$\mathcal{T}$ *longid*	=	nodeAn {*longid*,lookupId *longid*,[]},	*if longid* ∈ HOFS
	=	lookupId *longid*,	*otherwise*
$\mathcal{T}$ { *exprow* }	=	rectypeAn { $\mathcal{T}$ *exprow* }	
$\mathcal{T}$ *lab* = *exp* <, *exprow* >	=	rowRecAn (*lab*,$\mathcal{T}$ *exp*,<$\mathcal{T}$ *exprow*>)	
$\mathcal{T}$ let *dec* in *exp* end	=	$\mathcal{T}$ *exp*,	*if* $\mathcal{T}$ *dec* = seqAn []
		seqAn [$\mathcal{T}$ *dec*,$\mathcal{T}$ *exp*],	*otherwise*
$\mathcal{T}$ *exp atexp*	=	$\mathcal{A}$ ($\mathcal{T}$ *exp*) ($\mathcal{T}$ *atexp*)	
$\mathcal{T}$ *exp* : *ty*	=	$\mathcal{U}$ *ty exp*	
$\mathcal{T}$ *exp* handle *match*	=	$\mathcal{T}$ *exp*	
$\mathcal{T}$ raise *exp*	=	tyvarAn newTv[a],	*if* $\mathcal{N}$ *exp* = exn[b]
	=	$\mathcal{N}$ *exp*[c],	*otherwise*
$\mathcal{T}$ fn *match*	=	$\mathcal{T}$ *match*	
$\mathcal{T}$ *mrule* | *match*	=	altAn [$\mathcal{U}$ *match mrule*,$\mathcal{T}$ *match*]	
$\mathcal{T}$ *pat* => *exp*	=	arrowAn ($\mathcal{T}$ *pat*,$\mathcal{T}$ *exp*)	
$\mathcal{T}$ local dec_1 in dec_2 end	=	seqAn [$\mathcal{T}$ dec_1,$\mathcal{T}$ dec_2]	
$\mathcal{T}$ dec_1 ; dec_2	=	seqAn [$\mathcal{T}$ dec_1,$\mathcal{T}$ dec_2]	
$\mathcal{T}$ val *pat* = *exp*	=	$\mathcal{T}$ *exp*[d]	
$\mathcal{T}$ *tyvarseq tycon* = *conbind*	=	$\mathcal{T}$ (tyconAn)[e] *conbind*	
$\mathcal{T}$ (tyconAn) *con*[f] <of *ty* >	=	arrowAn ($\mathcal{T}$ *ty*,tyconAn),	*if ty* exists
		tyconAn,	*otherwise*
$\mathcal{T}$ *excon*[g] <of *ty* >	=	arrowAn ($\mathcal{T}$ *ty*,exnAn)[h],	*if ty* exists
		exnAn,	*otherwise*
$\mathcal{T}$ *excon* = *longexcon*	=	lookupId *longexcon*	
$\mathcal{U}$: ast → ast → absnet			
$\mathcal{U}$ *obj1 obj2*	=	$\mathcal{S}$ ($\mathcal{M}$ ($\mathcal{T}$ *obj1*) ($\mathcal{T}$ *obj2*)) ($\mathcal{T}$ *obj2*)	

[a] newTv generates a fresh, unique *tyvar*.
[b] By exn we mean constypeAn (_,exn).
[c] By $\mathcal{N}$ *exp* we mean the abstract network type derived from the semantics type of *exp*.
[d] Bindings from patabsnets ($\mathcal{T}$ *pat*,$\mathcal{T}$ *exp*) are added to Context.
[e] Traversal of *conbind* needs the absnet of the *tycon*, ie. (constypeAn ($\mathcal{N}$ *tyvarseq*,*tycon*)).
[f] The binding (*con*,$\mathcal{T}$ *con*) is added to Context.
[g] The binding (*excon*,$\mathcal{T}$ *excon*) is added to Context.
[h] By exnAn we mean constypeAn ([],exn).

Fig. 4. Schema $\mathcal{T}$: Traverse AST, annotate with HOF information

3.2.6 Stripping the abstract network type

The second phase of analysis is a *cleanup* operation in which irrelevant type information is stripped out leaving the "parallel" structure of the program suitable for further analysis.

The main part of the algorithm is in deciding which constructs are nested and which are sequenced. The scheme adopted here is that the instantiated HOF abstract network type is compared with the abstract network type of the original HOF definition. Where functional types or type variables in the original definition match nodeAns in the elaborated and analysed AST the node is considered to be nested.

All other occurrences of nodes are sequenced. This includes tuples, records and

$\mathcal{S}_c$: absnet → absnet → esn → esn

$\mathcal{S}_c$ (errorAn msg)	_	esn	=	adde (errorAn msg) esn
$\mathcal{S}_c$ baseAn	_	esn	=	esn
$\mathcal{S}_c$ (nodeAn node)	(arrowAn _)	esn	=	addn (nodeAn node) esn
$\mathcal{S}_c$ (nodeAn node)	(tyvarAn _)	esn	=	addn (nodeAn node) esn
$\mathcal{S}_c$ (nodeAn node)	An	esn	=	adds (nodeAn node) esn
$\mathcal{S}_c$ (constypeAn (a,n))	(constypeAn (a',n'))	esn	=	$\mathcal{S}_c$ a a' esn, *if* n = n'
$\mathcal{S}_c$ (rectypeAn r)	(rectypeAn r')	esn	=	$\mathcal{S}_c$ r r' esn
$\mathcal{S}_c$ (arrowAn (a,r))	(arrowAn (a',r'))	esn	=	$\mathcal{S}_c$ r r' ($\mathcal{S}_c$ a a' esn)
$\mathcal{S}_c$ (tyvarAn tv)[a]	(tyvarAn tv')	esn	=	esn, *if* tv = tv'
$\mathcal{S}_c$ nilAn	nilAn	esn	=	esn
$\mathcal{S}_c$ rowRecAn (l,a,r)	rowRecAn (l',a',r')	esn	=	$\mathcal{S}_c$ r r' ($\mathcal{S}_c$ a a' esn), *if* l = l'

[a] Since the absnets have been unified, *tyvar*s which match can be discarded, others are errors.

Fig. 5. Schema $\mathcal{S}_c$: Compare instance of HOF with definition

constructors where internal nodes are assumed to be evaluated in a predictable order. Note that although the alternation information is gathered during the first phase it is not processed during the cleanup operation other than to check that the alternation in the program is within the restrictions already outlined.

The algorithm is divided into two mutually recursive parts. The schema $\mathcal{S}_c$ (Fig. 5) compares the instance `absnet` with the original HOF `absnet`, generating a list of nested constructs and a list of sequenced constructs. The schema $\mathcal{S}_t$ (Fig. 6 takes an `absnet` and strips out non-essential type information, calling $\mathcal{S}_c$ when a node is encountered. In addition, both functions accumulate `errorAn` types into a separate list, replacing them with `baseAn` abstract network types.

Fig. 5 outlines the $\mathcal{S}_c$ schema. Prior to scanning two `absnet`s they are unified with appropriate abstract network type substitutions. This is to match up type variables which could have been renamed during analysis.

Note that:

1. The triple `esn` is a tuple of three `absnet` lists, the accumulated errors, sequenced constructs and nested constructs. The functions `adde`, `adds` and `addn` add a new member to the error, sequenced and nested construct lists, respectively. Initially, this contains empty lists.
2. `seqAn` does not appear in the rules for this schema. This would only be required if the HOF definition required tuples or records to be matched with abstraction points. None of the standard HOFs so far require this feature.

The final network is generated by the $\mathcal{S}_t$ schema, shown in Fig. 6. Mostly, this involves stripping out unnecessary type information. More is stripped out than is required by the subsequent analysis but it is intended to give human-readable results when the final abstract network type is printed out. Note that:

$\mathcal{S}_t$: absnet $\rightarrow$ absnet			
$\mathcal{S}_t$ errorAn msg[a]	=	baseAn	
$\mathcal{S}_t$ baseAn	=	baseAn	
$\mathcal{S}_t$ nodeAn {hof,hofAn,nest}	=	skelAn,	*if* s = []
	=	seqAn (skelAn::s),	*otherwise*
where		skelAn = nodeAn {hof,hofAn,n}	
and		instAn = $\mathcal{L}$ (nest @ ($\mathcal{E}$ hofAn))	
and		(e,s,n) = $\mathcal{S}_c$ instAn defAn[b] ([],[],[])	
$\mathcal{S}_t$ seqAn ans	=	baseAn,	*if* $\mathcal{S}_t$ ans = seqAn [][c]
	=	$\mathcal{S}_t$ ans,	*otherwise*
$\mathcal{S}_t$ constypeAn ([],_)	=	baseAn[d]	
$\mathcal{S}_t$ constypeAn (ans,name)	=	seqAn($\mathcal{S}_t$ ans)[e]	
$\mathcal{S}_t$ rectypeAn r	=	baseAn,	*if* $\mathcal{S}_t$ r = baseAn *or* $\mathcal{S}_t$ r = seqAn []
	=	$\mathcal{S}_t$ r,	*otherwise*
$\mathcal{S}_t$ arrowAn (a,r)	=	baseAn,	*if* $\mathcal{S}_t$ a = baseAn *and* $\mathcal{S}_t$ r = baseAn[f]
	=	arrowAn($\mathcal{S}_t$ a,$\mathcal{S}_t$ r),	*otherwise*
$\mathcal{S}_t$ tyvarAn tv	=	tyvarAn tv[g]	
$\mathcal{S}_t$ nilAn	=	baseAn	
$\mathcal{S}_t$ rowRecAn (lab,an,r)	=	seqAn(($\mathcal{S}_t$ an) @ ($\mathcal{S}_t$ r))	

[a] The error is accumulated in a separate list.
[b] `defAn` is the abstract network type derived from the HOF SML type.
[c] `baseAn` values are removed from the list.
[d] Nullary constructors cannot contain `nodeAn`.
[e] If $\mathcal{S}_t$ `ans` only contains `baseAn` then the result is `baseAn`.
[f] Functions involving only `baseAn` types are themselves `baseAn`.
[g] Type variables are retained to give meaning to partially applied nodes.

Fig. 6. Schema $\mathcal{S}_t$: Strip out unnecessary type information

1. The schema $\mathcal{L}$ turns a list of `absnet`s into a function type:
   $\mathcal{L}$ `[a,b,c]` = `arrowAn (a,arrowAn (b,c))`
2. The schema $\mathcal{E}$ removes all node wrappers, recursively, from an `absnet` and is applied to the result type of `nodeAn`s prior to comparison with the original definition.

3.2.7 Abstract network examples

The simple example of Section 3 results in the following abstract network types:

```
val x :: base
val ff :: base
val result :: node(fold,[base])
```

Fig. 7 gives further examples of the output from the first phase. The abstract network types are shown for each value. Note that the implementation uses a mixture of types derived from the semantics types deduced by the elaborator and from its own inferences resulting in some types being represented by their SML types and

```
fun ff x = x + 1
val ff :: (int->base)
fun gg lst = map ff lst
val gg :: (int list->node(map,base list,[(int->base), int list]))
fun hh llst = map gg llst
val hh ::
  (int list list->
   node (map,
         node(map,base list,[(int->base), int list]) list,
         [(int list->node(map,base list,[(int->base), int list])),
          int list list]))
val ggv = gg [1,2,3]
val ggv :: node(map,base list,[(int->base), int list])
val hhv = hh [[1],[2],[3]]
val hhv ::
  node (map,
        node(map,base list,[(int->base), int list]) list,
        [(int list->node(map,base list,[(int->base), int list])),
         int list list])
```

Fig. 7. Examples of abstract network generation

others by their abstract network types. Note that any SML types are automatically `baseAn`.

In the second phase the abstract network type for each skeleton implemented in parallel is stripped of extraneous type information yielding an abstract description of the parallel constructs contained within the program. For example:

```
val ff :: base
val gg :: (base->node(map,[base]))
val ggv :: node(map,[base])
val hh :: (base->node(map,[node(map,[base])]))
val hhv :: node(map,[node(map,[base])])
```

3.3 `Free`*: Free-value analysis*

For free value lifting, functions are augmented with additional formal parameters for their free variables and calls to those functions are extended with the corresponding free variables as the actual parameters. Our approach is closely related to Johnsson's lambda lifting (Johnsson, 1985) which was developed for the combinator-based implementation of Lazy ML. There, the objective was to turn a program into a set of global functions which could be used as rewrite rules for graph reduction.

There are two distinct problems associated with our execution model. Firstly, non-functional free-values in skeleton instantiation functions have to be detected and transmitted prior to running those functions remotely. Secondly, functional

```
fun ff1 (y,ff) = let val ff = named_value ff in ff y end
val _ = register "ff1" ff1
fun inner ff y l =
  (register "ff1_fvs"
   (y,let val _ = register "ff" ff in "ff" end);
   (pmap : string -> 'a list -> 'a list) "ff1" l)

fun ff y x = x + y : int
val y = 1

fun ff2 (y,ff) = let val ff = named_value ff in inner ff y end
val _ = register "ff2" ff2
val _ = register "ff" ff
val result =
  (register "ff2_fvs" (y,"ff");
   (pmap : string -> int list list -> int list list) "ff2" ll)
```

Fig. 8. Example showing transmission of free values

free-values have to made available to the remote instance. Since we cannot transmit functional values (and probably would not want to for efficiency reasons) this means lifting the free functions to the top level of the code, allowing them to be registered for remote use.

The problem of non-functional free-values is relatively simple to solve and some optimisation can be performed upon exactly which values need to be transmitted for a given skeleton. For instance, in the simple example in Section 3, the value `x` does not need to be transmitted since it is generated on all processors as part of the initial phase.

However, for the nested case, when the outer skeleton is reached, the worker processors are no longer generating values needed by the inner skeletons. For example, in the following, nested `map` program:

```
fun inner ff y l = map (ff y) l
fun ff y x = x + y : int
val y = 1
val result = map (inner ff y) ll
```

the non-functional value `y` and the functional value `ff` are free in the expression `(inner ff y)`. When the outer `pmap` is launched, the worker processors sit in a loop waiting for data and tasks to be sent from the master. They return to the Objective Caml code at the body of the function `ff` where the value `y` is bound as a function argument. Similarly for the intermediate-level processors, the values `ff` and `y` are arguments to the `inner` function.

Fig. 8 shows how this situation is handled by our approach. The anonymous expression is lifted to the top level with its free values as formal parameters (function ff2). The non-functional value y is handled without loss of generality by registering under the name "ff2_fvs". These values are passed to the instance function ff2 as additional parameters received from the root processor by the pmap function. These are then propagated from the intermediate processors to the lowest-level processors by re-registering as additional parameters to the ff1 function.

The problem of functional free values is much harder to solve. The functional free value ff is handled in the above case by exploiting its presence at the top level of the code and its full application in the pmap instance under the same bound name as at the top level. This is registered separately under the name "ff" and this handle is used as a substitute for the functional value.

This condition can be ensured for functions which can be lifted to the top level and do not participate in computation at the intermediate level, including partial evaluation. There is a loss of generality, however, since even renaming the function causes the registered handle to become invalid, for example, if the inner function had been defined as:

```
fun inner ff' y' l = map (ff' y') l
```

One possible approach would be to use inlining whereby formal functional parameters to functions are substituted for their bound values at the point of application. This method was used in the Ektran skeleton compiler (Hamdan, 2000) for a small functional language. It is not viable in a wider context, however. Firstly, it results in an explosion of code size for deeply nested functions. Secondly, it results in duplicated computation even for sequential code. Finally, it is extremely difficult to get bindings for values in the general case. Consider, for example, a higher-order function, taking a function as argument and returning a pair of functions:

```
fun ff x y = x + y
fun gg ff = (ff 1,ff 2)
val (hh,ii) = gg ff
```

The argument function would have to be substituted in the body of the HOF and the HOF scanned (through *match* constructs, if necessary) for the correct set of matches leading to the returned functions.

This analysis was partially implemented but subsequently replaced with defunctionalization where all values can be treated as for the non-functional free value y in the example above.

3.4 Tddf: Defunctionalization

3.4.1 Introduction

Defunctionalization is based on an idea by John Reynolds (Reynolds, 1972) for encoding functional arguments. A related technique is the "higher-order removal method" (Chin & Darlington, 1996). These methods were subsequently generalised to cover a full functional language for use in automatic derivation of a program generator from the semantics of a "domain-specific design language" (Bell *et al.*, 1997). Defunctionalization has also been used in the HDC compiler project (Herrmann & Lengauer, 2000), again motivated by the need to handle of free functionals in a skeletal functional language. In this technique, closures (in the sense of partially applied functions) 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.

There is, however, a problem associated with defunctionalization of SML. All of the previous algorithms make use of forward references in the code. Bell *et al.* define an ML-like language to develop the algorithm and HDC is a Haskell derivative with the advantage of a graph representation. To implement this algorithm for SML would require runtime registration of functions with attendant jump tables. The solution adopted here is to turn the entire program into a single mutually-recursive block.

Using runtime registration of functions would probably slow the execution down to an unacceptable extent since there is already an overhead of a datatype dereference for each function application. The solution using a mutually recursive block also has problems. Variables require renaming since, potentially, distinct variables from different contexts with the same name have to be wrapped into the mutually recursive block. This makes it difficult to retain the original semantics of the SML language, particularly for exceptions and types defined within `let` constructs. The complexity of our transformation phase could be reduced if the entire program were transformed by a unique renaming phase, including type and exception names. This is another future development which is not considered further here.

More seriously, the backend compiler is slowed down considerably by having to resolve the large mutually-recursive block. The computational complexity of the type analysis, for instance, is higher than for an equivalent code section without mutually-referential declarations.

The algorithm by Bell *et al.* attempts to perform specialization of polymorphic functions *in situ* whereas the HDC compiler uses a *monomorphisation* phase to eliminate all polymorphism. The algorithm presented here performs specialization although, with hindsight, the monomorphization approach would have been simpler. Bell *et al.* define seven rules for various aspects of the defunctionalization process including specialization of functions whereas Herrmann *et al.* use only four rules, assuming monomorphization. The method presented here implements the rules without explicit specialization of polymorphic functions but instead implements the other rules in the presence of a type substitution while analysing polymorphic code.

3.4.2 Defunctionalization method

Pre-analysis of the input program includes mutual-recursion removal by inlining and let-removal by the introduction of auxiliary functions (Johnsson, 1985). We use the

```
saturate (<exp> as (fn <match>)) =>
  for each <mrule> in <match>
    if      order > provided
    andalso <exp> is monomorphic
      then addVars (order - provided) <mrule>
      where order = order <exp>
        and provided = number of arguments up to <mrule>
```

Fig. 9. Rule saturate

type information generated by the elaborator but any generated code has to include synthesized elaboration information. This is used in subsequent analysis. The rules used by our transformation are essentially the rules in HDC with the complication of specialization and also the difference between expressions and atomic expressions (Milner *et al.*, 1990; Milner *et al.*, 1997).

Fig. 9 shows pseudocode for the addition of function arguments to functions returning functions as results and is the `etaExpand` rule of HDC:

```
fun add_y (y:int) (x:int) = x + y : int |-
saturate <fn y => add_y y> ==>
<fn y => fn tddf1 => add_y y tddf1>
```

The function `order` determines the number of arguments to the function for functional expressions and is zero for non-functional expressions. This is defined by (Bell *et al.*, 1997). The value `provided` is the number arguments provided by a function application. In this example, order = 2 (the type of `add_y` is `int -> int -> int`) and provided = 1 (`fn y =>`) so one extra argument is required.

Fig. 10 shows the rule for encoding a functional argument. A case is added to the datatype for the type of the expression and is further indexed by a unique index for the expression. The constructor encodes the free values of the expression. A case matching this constructor mapped to the expression is added to the apply function for the expression type. Thus:

```
encode <(add_y:int -> int -> int) (y:int)> ==>
datatype T_1 = C_1_1 of int
fun apply_1 F tddf1 =
  case F of
    (C_1_1 y) => add_y y tddf1 |-
<(C_1_1 y)>
```

Encoding is not done for function arguments since they will be encoded at the function application point. Polymorphic code is simply ignored: this will be encoded subsequently by the specialization process. For efficiency, initial values (in the SML prelude) are not encoded at all. These occur prior to the encoding block and are

```
addCase (<ty>,(<atexp>|<exp>),<fvs>) =>
   create datatype T_<T> and apply_<T>
   add constructor "C_<T>_<E> of <fvs>" to T_<T>
   add match "case (C_<T>_<E> <fvs>) => (<atexp>|<exp>)" to apply_<T>
   add (<id>,<dec>) for functional <id>s in <fvs> to free functions
   where <T> = index of type <ty>
     and <E> = index of expression (<atexp>|<exp>)

encode (<exp>|<atexp>) =>
  if      order (<exp>|<atexp>) > 0
  andalso (<exp>|<atexp>) is monomorphic
  andalso (<exp>|<atexp>) is not an initial value
  andalso <atexp> is not a function argument identifier
    then (<exp>|<atexp>) ==> C_<T>_<E> <fvs>
    where C_<T>_<E> = addCase (<ty>,(<exp>|<atexp>),<fvs>)
      and <ty> = the type of (<exp>|<atexp>)
      and <fvs> = free values in (<exp>|<atexp>)
```

Fig. 10. Rule encode

```
apply ((<exp> as <id>) <atexp>) =>
  if      <id> is a recursion variable
  andalso <exp> is being specialized
    then <exp> ==> <id>_<E>
    where <E> = index of expression <exp>
    else if <id> has not been specialized
    andalso <id> has not been applied
    andalso <id> is not a constructor
    andalso <id> is monomorphic
      then addCase (<ty>,<dummy exp>,[])
           <exp> ==> (apply_<T> <id>)
      where <T> = index of type <ty>
        and <ty> is the type of <exp>
```

Fig. 11. Rule apply

used in their original form. This creates the slight restriction that there can be no HOF arguments to functions, or as function results in the prelude.

Fig. 11 shows the handling of function application by the introduction of an "apply" function. Apply functions must not return functions so dummy arguments are synthesized up to the order of the function. Recursion variables within specialized functions need to be translated into the specialized function name. Note that we need to add a dummy type constructor to handle the situation where a function has an `apply` encoding but is not subsequently used later in the program. The dummy

```
translateTy (<ty> as (<ty_fn> -> <ty_arg>)) =>
  if <ty> is functional
    then <ty> ==> T_<T>
    where <T> is the index of <ty>
translateTy (<ty> as ({<tyrow>})) =>
  <ty> ==> {apply translateTy to each component of <tyrow>}
```

Fig. 12. Rule translateTy

```
hofWrapper (<atexp> as <id_HOF>) <ty_atexp> <ty_HOF> =>
  <wrapper> = saturate (<id_HOF>)
  foreach argument <atexp_arg> of <wrapper>
    if      functional <atexp_arg>
    andalso the type of <atexp_arg> in <ty_HOF> = string
      then <atexp_arg> ==> apply_<T'> <atexp_arg>
      where <T'> = type index of <atexp_arg>
  add "val <id_HOF>_<T> = <wrapper>" to free functions
  <atexp> ==> <id_HOF>_<T>
  where <T> = type index of <atexp>
```

Fig. 13. Rule hofWrapper

expression is a `raise Tddfexn<X>` exception where `<X>` is a unique index for the exception. For example:

```
apply <add_y y 1> ==>
<apply_1 (C_1_1 y) 1>
```

Fig. 12 shows the `translateTy` rule which transforms explicit types to match the encoding and specialization output. This is the equivalent of the `removeHOTypes` rule in HDC. Functional types are replaced by the type of the datatype encoding the function, type rows have to be scanned separately:

```
translateTy <int -> int> ==>
<T_1>
```

Fig. 13 shows the `hofWrapper` function which generates a wrapper for skeleton functions. This is needed because the instance functions to the HOF have to be "refunctionalized" prior to launching the skeleton. This process is not possible in general without creating an inverse of the defunctionalization process. Here, however, only HOF arguments to functions are being treated this way and the `apply` function is guaranteed to be generated by the defunctionalization process for the point of call. Note that the registration of wrapper functions for arguments to skeleton functions is deferred until after the encoding block. This is to prevent cycles in the encoding block. For example:

```
specialize <id> <ty> <exp> =>
  if <id> is not currently being specialized
    then return tddf <S> <exp>
    where <ty_exp> = type of <exp> in context
      and <S> = type substitution from <ty> to <ty_exp>

specializeId (<atexp> as <id>) =>
  if <id> is a skeleton HOF
    then <atexp> ==> hofWrapper <id> <ty> <ty_HOF>
    else <atexp> ==> specialize <id> <ty> <exp_id>
    where <ty> = type of <atexp>
      and <ty_HOF> = type of the unapplied HOF function
      and <exp_id> = the expression to which <id> is bound in context

specializeFn (<exp> as (fn <match>)) =>
  <exp> ==> specialize <id> <ty> <exp>
  where <id> = new unique identifier
    and <ty> = the type of <exp>
```

Fig. 14. Rule specialize

```
hofWrapper <"map">
           <(int -> int) -> int list -> int list>
           < string -> int list -> int list> ==>
val map_1 =
  (fn tddf1 => (fn tddf2 =>
    ((map (apply_2 tddf1)) tddf2))) |-
<map_1>
```

Fig. 14 shows how specialization is implemented. This has two main cases; one for functional identifiers and one for anonymous functional expressions. In the former case the binding for the identifier in the current context is used, in the latter a new identifier is generated to create the binding. In each case, the actual specialization process involves matching the type of the polymorphic function with its concrete type at the point of call and recursively calling the main defunctionalization routine under the auspices of the type substitution:

```
fun id x = x |-
specialize <(id:(int -> int) -> (int -> int))> ==>
fun id_1 x tddf1 = apply_2 x tddf1 |-
<(id_1:T_1)>
```

Fig. 15 shows the toplevel scan of the syntax tree. Note that the type substitution `<S>` is implicitly applied in all the other rules. This is omitted in the rule descriptions for brevity. All the other rules also check for monomorphic code after applying the substitution. The `tddf` rule has cases for functional identifiers, anonymous functional expressions and function applications.

```
tddf <S> (<atexp> as <id>) =>
  if      <id> is functional
  andalso <id> has HOF arguments
  andalso <id> is not a function argument
  andalso <id> is monomorphic
  andalso <id> is not an initial value
    then <atexp> ==> specializeId <atexp>
  if <id> is a fully applied function application identifier
  andalso <id> is not a function argument
    then <atexp> => encode <atexp>
tddf <S> (<exp> as (fn <match>)) =>
  if <exp> is not currently being specialized
  andalso <exp> has HOF arguments
    <exp> ==> specializeFn <exp>
tddf <S> (<exp> as (<exp_fn> <atexp_arg>)) =>
  if      apply has not been applied to <exp>
  andalso encode has not been applied to <exp>
  andalso specialize has not been applied to <exp>
    then <exp_fn> ==> tddf <exp_fn>
         <atexp_arg> ==> tddf <atexp_arg>
         <exp_fn> ==> apply <exp_fn>
         <atexp_arg> ==> encode <atexp_arg>
         if      order > 0
         andalso specialize has not been applied to <exp_fn>
         andalso apply has not been applied to <exp_fn>
         andalso encode has not been applied to <atexp_arg>
           then <exp> ==> encode (<exp_fn> <atexp_arg>)
```

Fig. 15. Rule tddf

The `tddf` rule is mostly associated with deciding where the four rules; `specialize`, `encode`, `apply` and `translateTy` are applicable. The result of the pass over the syntax tree of the program is a revised program with the four rules applied plus an encoding block. The encoding block contains:

- Hash table of integers to types
- Set of datatypes generated by encode and apply
- Set of `type` and `datatype` definitions lifted from the program
- Set of functions generated by specialize
- Set of functions generated by apply
- Set of function definitions referenced in datatype constructors

When the program is output this data block is turned into a set of datatype definitions plus a mutually recursive block of all the functions. This is inserted between the prelude code and the main body of the program.

Fig. 16 shows an example taken from (Bell *et al.*, 1997). This illustrates how the HOF `id` is treated differently in different contexts. Fig. 17 shows the simple example of Section 3 with a functional `fold` and the wrapper code generated for it.

```
fun id x = x
val result1 = id id (id 7)
val result2 = id (id id) (id 7)

(* Type hash map
  1=((int -> int) -> (int -> int))
  2=(int -> int)
*)
datatype T_2 = C_2_2 | C_2_1
fun apply_2 F tddf3 =
  case F of
    C_2_2 => ((id_1 C_2_1) tddf3)
  | C_2_1 => ((id) tddf3)
and id x = x
and id_1 x tddf1 = apply_2 x tddf1

val result1 = id_1 C_2_1 (id 7)
val result2 = id_1 C_2_2 (id 7)
```

Fig. 16. Higher order id example from (Bell *et al*, 1997)

```
(* Type hash map
  1=({1 : int, 2 : int} -> int)
  7=(({1 : int, 2 : int} -> int) -> (int -> (int list -> int)))
*)
datatype T_1 = C_1_1 of int

fun apply_1 F tddf4 =
  case F of
    C_1_1 x => ff x tddf4
and ff (x) (y,z) = x + y + z
and fold_7 tddf1 tddf2 tddf3 =
  (fn _ =>
   (pfold : string -> int -> int list -> int) "ff1" tddf2 tddf3)
  (register "ff1_fvs" tddf1)

fun ff1 tddf5 = apply_1 tddf5
val _ = register "ff1" ff1

val x = 1
val result = fold_7 (C_1_1 x) (~x) [1,2,3]
```

Fig. 17. Example illustrating a `fold` skeleton wrapper

The method of defunctionalization described here is able to extend the usefulness of the skeletons in the sense that they can be used in the presence of *partial evaluation* and *free functional values*. The global execution overhead introduced by datatype dereferencing is tolerable, a typical slowdown of about 10-20% is observed which can be counterbalanced by parallel speedup. Some classes of programs can be handled by our free-value analysis alone for which this overhead does not apply but the selection of these two, complementary methods has not been automated in our compiler. This could be achieved by detecting non-liftable free functionals in the program during free-value analysis but the detection of such values is complicated by the *renaming* problem mentioned in Section 3.3.

3.5 `Analyse`: Backend analysis

Once the abstract network has been deduced and free values have been handled by appropriate transformations the nominated HOFs have to be replaced by calls to their parallel equivalents. These parallel constructs are implemented in C and MPI and are linked into our program using the Objective Caml foreign function interface. This provides a function (`register`) to register Caml values on the Caml side with the runtime system under a `string` name. A function (`named_value`) is provided to access these values in C. We use this mechanism to allow communication of values between the executing Caml program and our parallel skeletons.

There are several complications in generating the correct code to launch a parallel skeleton:

- HOFs must be nominated by the `signature HOFS` mechanism.
- Type information has to be reconstructed at the point of call.
- Anonymous expressions have to be lifted and given names.

When a skeleton HOF is encountered the type of the unapplied HOF, the type of the parallel equivalent function and the instance expression are scanned simultaneously. The scan operates by successively stripping off arguments at function application points. Thus, when a `string` type in the HOF type matches a function type in the instance type the value of the expression at that point is an argument to the skeleton. This is then lifted under a new unique function name, registered with the runtime system under a unique `string` name and the expression is replaced by this `string`. Furthermore, the type of the parallel function can be deduced by unifying the type of the HOF with the type of the expression and substituting in the type of the parallel equivalent. Thus for the simple example in Section 3 the following situation arises:

```
fold : ('a*'a->'a)    -> 'a  -> 'a list  -> 'a
pfold: string         -> 'a  -> 'a list  -> 'a
<exp>: (int*int->int) -> int -> int list -> int
<exp>: ff                (~x)   [1,2,3]
```

At the first function argument during the scan, `('a*'a->'a)` matches `string *`

System	CPU	PEs	Speed	OS	MPI
Fujitsu AP3000	UltraSPARC	16	300 MHz	Solaris 2.6	MPIAP
IBM RS/6000 SP2	Power PC	16	332 MHz	AIX 3.4	MPCC
Beowulf	Celeron	32	533 MHz	Linux 2.2.16	LAM 6.3.2
Sun StarCat 15000	UltraSparc III	32	900 MHz	Solaris 2.8	MPICH 1.2.1[a]

[a] Using the MPICH `chshmem` driver.

Table 1. *MIMD target systems.*

ff. Thus the expression `ff` is lifted into a new function, say `ff1` and a declaration to register this is placed at the top level of the code, preceding the declaration in which the HOF was found. The expression and HOF types are unified giving the substitution `{int/'a}` which is applied to the `pfold` type giving:

```
string -> int -> int list -> int
```

This type is explicitly applied to the parallel equivalent function name. Finally, free values are passed to the runtime using the `register` mechanism under the string name for the argument function with `"_fvs"` appended. The free values' string names are picked up in the skeleton implementation, the values are accessed using `named_value` and transmitted to the remote instance function where they are added to the function's call. This means that no structural changes are required to the parallel equivalent function which would invalidate the `signature HOFS` mechanism. The resulting code for the simple example would be as follows, although note that in this case no free value passing is actually required:

```
val x = 1
fun ff1 (x) (y,z) = x + y + z
val _ = register "ff1" ff1
val result =
  (fn _ =>
    (pfold : string -> int -> int list -> int)
     "ff1" (~x) [1,2,3])
  (register "ff1_fvs" (x))
```

4 Example applications

The following sections present results for several example applications on four MIMD parallel machines, three distributed memory and one shared memory. Target machine characteristics are shown in Figure 1.

None of the exemplars requires defunctionalization so free-value analysis was used in all cases. The runtimes are the average of three executions of each program and the speedup is based on the single processor runtimes.

4.1 Prime filtering

Here we use the `pmap` skeleton constructively to build a `pfilter` skeleton. The application is filtering a list of randomly generated, arbitrary-length integers for primality. The prime test is a naive implementation:

```
fun isPrime' n l1 =
  n < l1*l1 orelse
  (if n mod l1 = 0 then false else isPrime' n (l1 + 2))

fun isPrime 2 = true
  | isPrime n = if n mod 2 = 0 then false else isPrime' n 3

val nPrimes = filter isPrime randomNumbers
```

There is no opportunity for parallelism in the `isPrime` function, as it stands, and we cannot directly use the `map` function over the list since the length of the list will change. We can, however, apply the predicate `isPrime` to the list of numbers and then use a sequential `fold` over the resultant list of boolean values. The resultant function is illustrated in Section 1.1 and has been incorporated into our compiler as the `pfilter` skeleton using the `signature HOFS` mechanism as shown in Section 3.2.1. The performance of the resulting program is shown in Fig. 18. This is virtually identical to a typical profile for a `pmap` implementation since the sequential overhead time is about 10^{-4} times the time to run the `pmap` in parallel. The integers need to be quite large to get speedup. The speedup for 31 processors on the Beowulf architecture, as the maximum size of the integers 10^N increases from $N = 4$ to $N = 9$, is show in Table 2.

Table 2. *Speedup versus integer size for the Beowulf architecture*

N	4	5	6	7	8	9
Speedup	0.13	0.14	0.49	1.13	2.48	6.61

4.2 Euler totient function

Here we illustrate the use of the `fold` skeleton. This application sums the Euler numbers of a list of integers (Hammond & Michaelson, 1999) and we are able to define several versions using different combinations of `map` and `fold` (Scaife *et al.*, 2002a). It turns out, however, that the method with the best performance is a single non-nested `fold`.

The sequential algorithm can be expressed as:

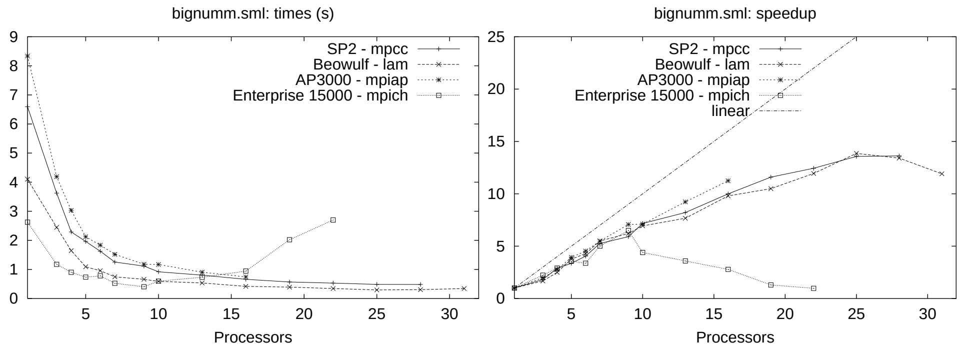

Fig. 18. Runtime and speedup for prime filtering

```
fun gcd x 0 = x
  | gcd x y = gcd y (x mod y)
fun relprime x y = gcd x y = 1
fun euler n = if n < 1 then 0 else zfilter (relprime n) n
val result = fold (op +) 0 (map euler nList)
```

where `zfilter p n` counts the number of elements for which the predicate `p` is true in the integer list `[1,...,n]`. This is structured as a sequence of a `map` followed by a `fold` but gives very poor parallel performance. However, the two operations can be combined into a single `fold` by using a datatype to mark Euler computation:

```
datatype EI = E of int | I of int
fun eiplus (I i1,I i2) = E ((euler i1) + (euler i2))
  | eiplus (I i,E e) = E ((euler i) + e)
  | eiplus (E e,I i) = E (e + (euler i))
  | eiplus (E e1,E e2) = E (e1 + e2)
val (E result) = fold eiplus 0 (seqmap I nList)
```

The function `seqmap` is an implementation of `map` which is not parallelized by our compiler. One additional overhead is the creation, transmission and deletion of the datatypes but this is offset by combining the two HOFs into one. The performance of the `fold` version is presented for each of the target architectures is presented in Fig. 19. The runtimes are very similar, given the different speeds of the processors and the relative speedups are identical apart from the Enterprise which, as for the prime filtering example above, tails off badly at higher numbers of processors.

4.3 Matrix Multiplication

Here we present the development of an $N \times N$ matrix multiplication algorithm. There are many approaches to matrix multiplication in parallel including block-structured and systolic algorithms (Golub & van Loan, 1989). A naive approach us-

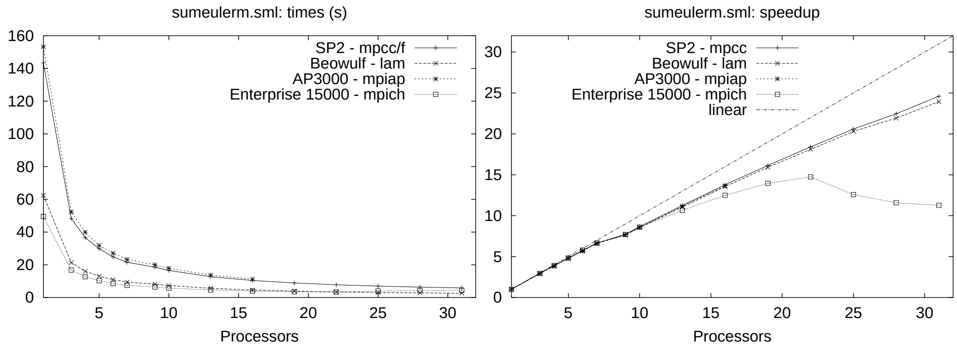

Fig. 19. Runtime and speedup for sum of Euler numbers

ing functional `map`s is presented which is improved slightly by re-organising the computation then a more sophisticated method is devised using a specialised toroidal topology skeleton (`ptorus`).

For $N \times N$ matrices $A = a_{ij}$ and $B = b_{ij}$, $1 <= i, j <= N$ the product is defined as:

$$c_{ij} \;=\; \sum_{k=1}^{N} a_{ik} b_{kj} \tag{1}$$

The data dependencies in this computation are surprisingly complex once the various decompositions are considered (Beaumont *et al.*, 2001). If, however, we assume that one matrix is present on all processors the computation can be decomposed row-wise into a `map` of dot products:

```
C = 0
for i = 1 to N
  for j = 1 to N
    for r = 1 to N
      C(i,j) = C(i,j)+A(i,k)*B(k,j)
```

The inner `r`-loop is a dot product of A's rows with B's columns. This allows both the outer loops to be expressed with our functional `map`, as follows:

```
fun inner Arow Bcol = dotprod Arow Bcol
fun outer BT Arow = map (inner Arow) BT
fun mmult A B = map (outer (transpose B)) A
```

We have implemented this program (Michaelson *et al.*, 2001) for integer matrices. It has also been used as an exemplar for Hamdan's EKTRAN skeleton-based system (Hamdan, 2000). The structure is a nested `map` where the two skeletons are very closely coupled. Note that the entire matrix `B` is passed as an argument to the `outer` function and is thus free in the outer `map`'s instance expression. No explicit handling of this data is required since our analysis detects this, resulting in transmission at runtime.

Fig. 20 (`matrixnumm.sml`) shows the performance of the nested `map` implementation for a 300×300 matrix of random integers. There is some speedup for small numbers of processors but, in general, this algorithm is *commmunications-bound*. We can attempt to alleviate this problem by reducing the overall communications costs. One problem with the naive approach is that data is retransmitted to processors (in our skeleton we cannot say which elements of the list will end up on which processor). A simple solution is to use the *outer product* of the columns of one matrix with the rows of the other:

```
fun BMmult (A,B) =
  let
    val outerAB = outer_product (A,transpose B)
  in
    split (length A) (map Mdotprod outerAB)
  end
```

This uses a manually-generated block-decomposition of the two matrices and a list of pairs of blocks forming the outer product of the two block-vectors is processed using the `map` skeleton. Fig. 20 (`matrixbopnumm.sml`) shows the resulting performance. Note that the runtimes are significantly greater due to the splitting and merging overheads but that there is much better relative speedup.

Finally, we have developed a `torus` skeleton which implements Gentleman's Algorithm (Gentleman, 1978). This is a systolic method where the matrices are distributed across a square array of processors. Each processor receives the blocks associated with that processor's local block and maintains an accumulator of the result block for that co-ordinate. In addition, this skeleton makes use of the Objective Caml `Bigarray` module which allows C-compatible data to be accessed from within Objective Caml. This means that communications are much faster because there is no marshalling of data or copying of buffers but has a significant overhead for accessing the array elements in Objective Caml:

```
open Bigarray
val acc = Genarray.create kind c_layout [|n,2,r,c|]
val res = Array2.create kind c_layout r c

fun ff (cnt,isfirst,mydata,shiftbuf,shiftright,shiftdown) =
  let
    fun accumulate data = <add data to accumulator acc>
  in
    if isfirst then accumulate mydata else ();
    shiftright 0 1; shiftdown 1 1; accumulate shiftbuf;
    if cnt >= Int.min (m,n) - 1
      then (<sum accumulator acc data into result res>;
```

```
            (true,res))
      else (false,res)
  end

val result = torus ff AB
```

The $r \times c$ matrices have to be built into the four-dimensional array `AB` where each matrix is divided up into $m \times n$ submatrices. The `torus` skeleton provides the instance function `ff` with two functions, `shiftright` and `shiftdown` which move the blocks around within the grid, keeping track of the original indices of the data and providing the current data in `shiftbuf`. The skeleton then repeatedly calls the `ff` function until it returns `true` plus the local result. Finally, the `torus` skeleton uses MPI's collective communications to gather the local results into the final matrix product.

Fig. 20 (`matrixtorus.sml`) shows the performance of this skeleton. Here, we get slightly slower runtimes than with the naive version but for a 2×2 array of processors the new skeleton gives very good speedup. The problem size is not large enough for any further improvement on a 3×3 array.

4.4 Ray tracer

Ray tracing is a technique for mapping a representation of a three dimensional object, for example a CAD model, onto a two dimensional image, for example a monitor screen. Ray tracing is useful for investigating parallelism as it offers varying degrees of processing depending on the number of objects to be traced and the desired detail.

Here, we present a ray tracing algorithm for spheres. The original implementation in the dataflow language Id used a two-dimensional list comprehension to build the image up from cartesian co-ordinates plus a function to initiate tracing at each point. Our version has a nested `map` structure, implementing this comprehension:

```
val ind = indxs 0 (winsize-1) 0 (winsize-1)
fun ff (x,y) =
  ((x,y),tracepixel spheres lights
         x y firstray scrnx scrny)
val image = map (seqmap ff) ind
```

The `indxs` function generates a list of `(x,y)` indices for the image, the function `ff` calls the tracing code on the indices. In practice, the two maps are so closely coupled and the decomposition domain (pixels) is too fine-grained to give speedup from the inner `map`. The results are for a 350×350 image with the single outer `map` implemented in parallel. Fig. 21 shows the runtime and speedup from our implementation. For this application, all the architectures give very good relative speedups.

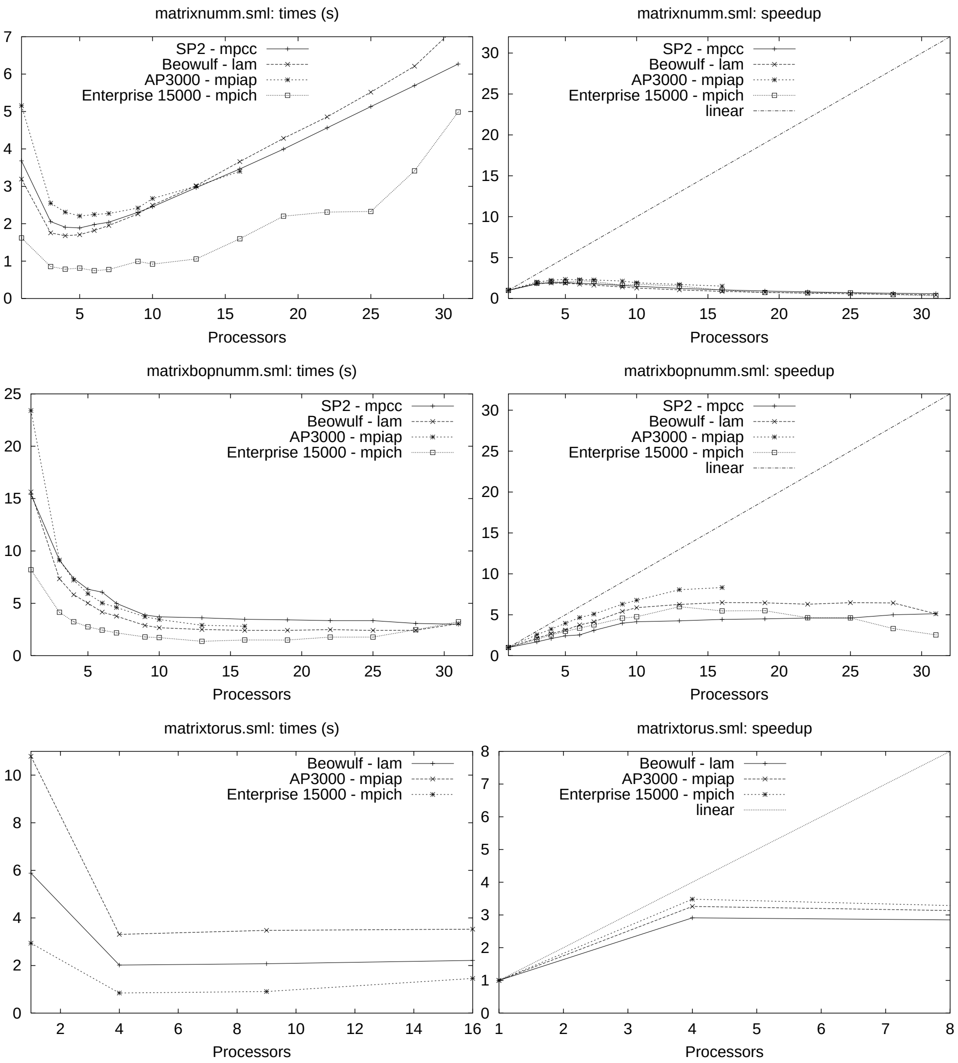

Fig. 20. Runtime and speedup for matrix multiplication

4.5 Genetic algorithm

Genetic algorithms are an evolutionary computing approach to traversing very large search spaces. Candidate solutions to problems are represented as genomes consisting of a sequence of gene codings. A genome may be interpreted to derive a fitness value indicating how good a solution it represents. Poor solutions may be discarded and promising solutions may be refined through the evolutionary operators of mutation, which randomly changes a genome, and cross-over, which swaps random

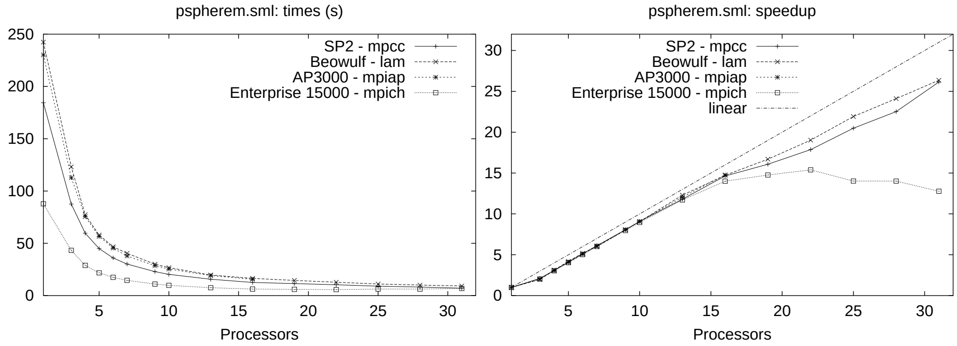

Fig. 21. Runtime and speedup for ray tracing

portions of genomes. Genetic algorithms are rich in potential parallelism for processing both individual genomes and genome pools.

We have implemented a subset of the libraries for RPL2 (Reproductive Plan Language) for developing parallel genetic algorithms. It is based around a simple flat imperative language with data-parallel iterators for manipulating genome pools. RPL2 was developed originally at the Edinburgh Parallel Computing Centre in a joint project with British Gas (Surry, 1993) investigating pipe-network optimisation problems. It is now maintained and distributed by Quadstone Ltd. In this subset we have implemented a linear equation solver applied to data generated by our profiler (Michaelson & Scaife, 2000). Here we have a triply-nested `map` structure: mapping over genome lists, mapping over genomes and calculating the fitness function itself. The innermost `map` is far too fine-grained for efficient parallel execution:

```
fun nextgeneration gsPop = map pfitness (breed gsPop)
fun run population =
  <until convergence>
  run (map nextgeneration population)
```

There is a lengthy sequential initialization cost which takes about 10 times as long as a single generation (iteration) but convergence takes about 50 to 100 generations so the iteration time dominates. We present the results for 10 generations. This is sufficient to reach a steady-state and also for the initialization costs to be counterbalanced. There is still a significant bottleneck between iterations, however, since the entire population is effectively redistributed between generations. This could be mitigated since the only reason for this data redistribution is to allow all the processors to terminate on the same iteration since the global fitness function needs to be computed at a central processor. A better solution would be to define an *iterative* `map` skeleton which retains the data on each processor between generations. Fig. 22 shows the performance for this application. Again, reasonable speedups are

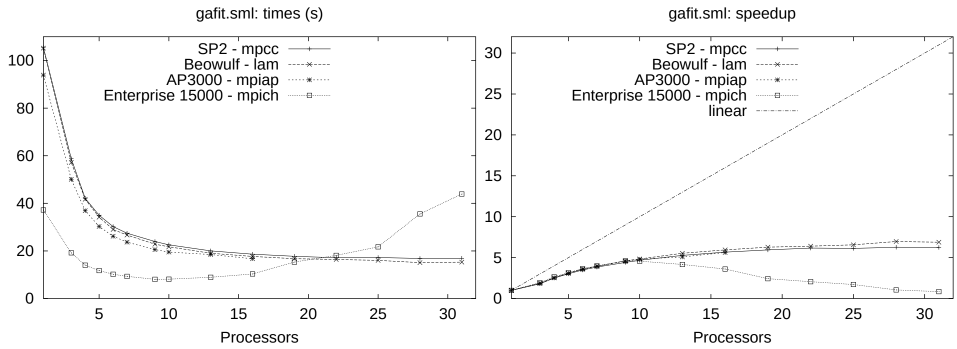

Fig. 22. Runtime and speedup for parallel GA

observed for small numbers of processors but the poor scalability of this method is apparent in the relative speedup profile.

4.6 Linear equation solver

This algorithm finds an exact solution of a linear system of equations of the form $Ax = b$, where all values are in the integer domain, and is discussed in detail by Loidl (Loidl *et al.*, 1999). The original algorithm was coded in Haskell and has been directly translated into SML. The method is to map the problem onto multiple *homomorphic images*, i.e. modulo a prime number, and solve in the homomorphic domain. The final solution is generated by applying the Chinese Remainder Algorithm to the homomorphic solutions. Parallelism is a single `map` over the images (the *forward mapping* phase) since nested parallelism within the image was found to be too fine-grained:

```
fun g_hS p = get_homSol aN bN p
val (xList:big_int list list) = map g_hS boundPrimes
```

The function `get_homSol` solves the equation (`aN x = bN`) modulo prime number `p`, the value `boundPrimes` is a list of prime numbers estimated to be the number of primes required for a solution. Note that the selection of prime numbers is problematical for this implementation since some prime numbers will result in singular matrices during image solution (so called *unlucky primes*). The technique adopted is to iteratively add more prime numbers until the estimated number of non-singular solutions is obtained. Successive iterations require fewer primes and the degree of parallelism becomes constrained. Luckily, the number of unlucky primes is small relative to the total number of estimated primes.

Fig. 23 shows runtimes and speedups for a dense 40×40 matrix of positive arbitrary-precision integers. Again we see the familiar `pmap` profile with good speedup for small numbers of processors tailing off for larger numbers of processors.

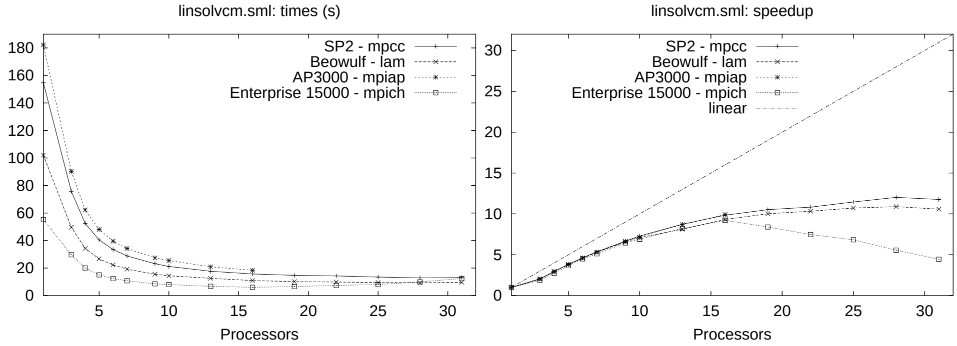

Fig. 23. Runtime and speedup for linsolv

4.7 Parallel molecular dynamics

Molecular dynamics (MD) is an important area of physics and chemistry in which parallel computing has been able to make an impact (Heermann, 1990). MD is a computationally intensive method of studying the time-evolution of a system of particles. For such systems, the particles' motion is governed by interatomic forces that depend on the location of the rest of the atoms in the system. Positions and velocities are updated subject to this field of interatomic forces, leading to a step-wise estimate of the particle trajectories over time. With this technique, the kinetic and thermodynamic behavior of particle assemblies may be studied.

Parallel molecular dynamics (PMD) covers a wide range of computational situations and there have been a large number of methods published in the literature (Beazley *et al.*, 1995). Factors which influence the choice of algorithm and data storage/locality include;

- the range over which particle interactions is significant,
- the number of atoms/particles in the simulation, and
- whether a domain decomposition or a particle decomposition is used.

In addition, the data structures and indexing mechanisms play an important role in the success of the chosen algorithm. The optimal data locality can also change as the simulation proceeeds, so load balancing is an important issue (Kalé *et al.*, 1998).

We have translated a naive $O(N^2)$ all-pairs algorithm implemented in FORTRAN90 (Hayashi & Horiguchi, 1997) into parallel SML suitable for analysis by our compiler (Scaife *et al.*, 2002b). Note that we are at a disadvantage in numerical-intensive applications, compared to FORTRAN90. The additional cost of imperative constructs plus factors such as boxed floating-point values means there is an inherent execution cost for equivalent imperative code between FORTRAN90 and SML. In addition, functional languages are usually optimized for symbolic applications rather than numeric ones, since this is where most successful functional programs have been deployed.

Here, we present two implementations. The first is a purely functional implementation using a new skeleton which computes all the inter-particle interactions (`pfpairs`). This is derived from our associative `fold` skeleton but cannot be expressed using `fold` itself because of the two-way interaction between list elements (symmetry is used to reduce the complexity of the algorithm by a factor of 2):

```
fun fc (acc,(qi,vi,fi),(qj,vj,fj)) =
  let
    val qq = limit3 (sub3 (qi,qj),~s2,s2)
    val rr = ssq3 qq
    val r6 = rr*rr*rr
    val r12 = r6*r6
    val frr = 1.0/(r12*rr)-0.5/(r6*rr)
    val ff = smul3 (frr * dt22) qq
  in
    (acc',ff,neg3 ff)
  end
fun ca (acc1,acc2) = sumacc (acc1,acc2)
fun addqvf (delta,(q,v,f)) = (q,v,add3 (f,delta))
val ((epot,vir),qvf) = fpairs ca add3 addqvf fc nullacc qvf
```

The `fc` function computes the force between particle pairs and also sums an accumulator `acc` between all pairs which is used in computing the physical properties of the system. The `ca` function sums the accumulators and the `addqvf` function updates the force component in the position (`q`), velocity (`v`) and force (`f`) triple which describes the particle ensemble. The `pfpairs` skeleton farms out groups of pairs of particle descriptions and uses the auxiliary functions to compute the final result. This skeleton sits within the outer simulation loop which iterates over a preset number of time increments.

Fig. 24 (`dn2mdf2.sml`) shows the performance of this program for 216 particles over 16 time-steps. Again, since we are using a master-slave model there is good speedup for small numbers of processors but with poor scalability. We do not compare favourably with FORTRAN90, however, which takes 142s compared to 2383s for SML, sequentially on the SP2. This is a slowdown of 16.8 which is not entirely unexpected for this type of application.

We have also developed a ring topology skeleton (`ptr`) and applied it to this problem. This skeleton also uses the `Bigarray` module interface to C-compatible data as described for the `ptorus` skeleton in Section 4.3. The functional interface to this essentially imperative skeleton is problematical but, briefly, requires two functions as parameters. One function is passed the position and force data for its own section of the particle ensemble along with a second set of position/force data (from another processor). On output this is required to compute the forces between the two sets of particles. A flag is provided to denote when both sets of

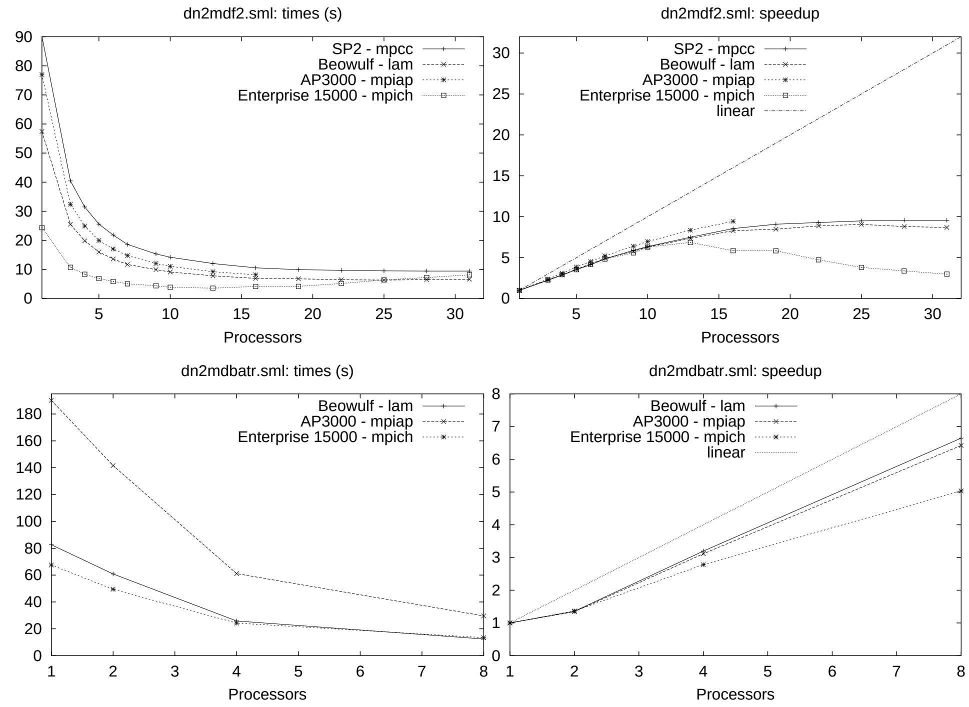

Fig. 24. Runtime and speedup for parallel molecular dynamics simulation

data represent the same set of particles (for computation of forces between local particles). The second function is required to update local forces with force data computed remotely. The `ptr` skeleton divides the particles among the processors and manages the communication pattern required to compute the force interaction between all pairs of particles.

Fig. 24 (`dn2mdbatr.sml`) shows the performance of this method for 512 particles over 200 time steps. This shows good speedup up to 8 processors (the number of processors must be a divisor of the cube root of the number of particles). The computation times also match up favourably with the FORTRAN90. The FORTRAN90 takes 913s sequentially and 216s on 4 processors on the SP2 whereas the SML gives 799s sequential and 256s on 4 processors on the Beowulf[7]. Speedups are 4.23s for FORTRAN90 and 3.13s for SML.

[7] We could not execute the two programs on the same machine because of a problem with `Bigarray` on the SP2 and the lack of a FORTRAN90 compiler on the other architectures

4.8 Summary

The seven applications shown here are from widely different applications and show good performance with small numbers of processors but with poor scalability to larger processor counts. The reasons for the modest performance are discussed in Section 2.2 and relate to the relatively poor communcation to computation ratio of our skeletons.

Table 3 shows the fastest parallel runtime and the speedup for the number of processors at which that time was achieved for each program. Overall, most applications give reasonable performance in terms of relative speedup.

Our experience suggests that while the development of even relatively simple parallel skeletons such as `pmap` and `pfold` is difficult and error-prone, it is eminently worthwhile. As these example applications show, significant automatic skeleton reuse can be achieved, with consistent behaviour across a variety of MIMD architectures, via the PMLS compiler.

Table 3. *Fastest runtime T (seconds) and peak speedup SP for all applications*

Program	Beowulf		SP2		AP3000		StarCat	
	T	SP	T	SP	T	SP	T	SP
`bignumm`	0.296	13.86	0.484	13.62	0.742	11.24	0.404	6.490
`sumeulerm`	2.606	23.94	5.831	24.60	11.22	13.66	3.360	14.74
`matrixnumm`	5.390	3.262	1.888	1.949	2.206	2.337	0.745	2.173
`matrixbopnumm`	2.410	6.492	2.999	5.131	2.810	8.328	1.371	5.986
`matrixtorus`	18.22	2.937	N/A	N/A	29.47	3.334	0.845	3.481
`pspherem`	9.195	26.36	7.056	26.14	15.59	14.77	5.702	15.39
`gafit`	15.09	6.970	16.84	6.268	16.70	5.623	8.076	4.605
`linsolvcm`	9.368	10.89	12.86	12.03	18.38	9.916	6.003	9.210
`dn2mdf2`	6.340	9.054	9.430	9.563	8.160	9.434	3.553	6.857
`dn2mdbatr`	12.43	6.647	N/A	N/A	29.61	6.423	13.41	5.034

5 Related approaches

Glasgow Parallel Haskell by Trinder *et al.* (GpH) (Loidl *et al.*, 1999) is a conservative extension of the Haskell language. The primitives `'par'` and `'seq'` are provided and results in a *thread-based* execution. Co-ordination is expressed independently from the main algorithm using *evaluation strategies* (Trinder *et al.*, 1998). GPH has the ability to transmit function closures by the expedient of providing a global heap and free-values are provided by the runtime system so this compiler does not have the complications encountered by our compiler. Within this framework skeletons such as parallel map can easily be expressed but require explicit control of granularity and data locality using user-defined strategies.

The HDC compiler project by Herrmann *et al.* (Herrmann *et al.*, 1999; Herrmann & Lengauer, 2000) is a skeletons-based compiler founded on a strict subset of Haskell. Unlike GPH, however, a global heap is not provided, giving skeleton implementations a cleaner semantics in the sense of *communication-closed blocks*. This means that, like our compiler, HDC cannot transmit function closures resulting in many similarities with our work. A transformational approach to compilation is used and compilation output is to C with MPI calls. Several skeletons are provided, including `map`, `red` (`fold`), `filter` and `scan`. Special emphasis is given to *divide-and-conquer* skeletons, for which two variants are provided; `dc0` is a general divide-and-conquer implementation, and `dc4io` is a divide-and-conquer over decomposable blocks of data. HDC uses the same lambda-lifting method as our compiler. Defunctionalization is also used but works using a *monomorphization* approach to handling polymorphism. HDC is slightly more expressive than our compiler, for instance list comprehensions are transformed into loops and filters which can be exploited by skeleton compilation. Nesting is not explicitly mentioned, however.

Skeletons have been implemented in Eden (Klusik *et al.*, 2000), a Haskell extension with primitives to allow the creation of processes and where co-ordination is expressed explicitly by the use of channels. Within this framework skeletons such as parallel map and fold can be defined but the application of such skeletons is controlled explicitly by the programmer.

Serot *et al.* have developed the SKiPPER language in which skeletons are built using Objective Caml and MPI (Coudarcher *et al.*, 2001). Here, co-ordination is carried out using a dataflow model and skeletons such as *task farm* and *data farm* are provided. Arbitrary nesting of skeletons is possible and recently, CASE tools have been used to map the dataflow structure of the program onto target architectures (Serot, 2001). This method is intended to allow irregular parallelism to be handled. One potential problem with the approach is that free-values have to be transmitted upon each instance.

We are currently engaged in empirical comparisons of PMLS with GpH and Eden. Preliminary results suggest that our HOF/skeleton approach achieves faster absolute times and similar speedups on the same tests on the same parallel architectures (Loidl *et al.*, 2003).

6 Conclusions

We think that PMLS represents considerable progress in the achievement of our overall objective of an automatic parallelising compiler for practical parallel functional programming. There are, however, many areas in which our compiler could be improved and further developed. Here, we assess the main achievements and limitations of PMLS, and consider future directions for its development.

6.1 Achievements

We have constructed a complete system for the selection of nominated HOFs for parallel realisation. As it stands, the compiler implements a *mostly-implicit* parallel programming system for a significant subset of Standard ML. Using the compiler to implement functional `maps` and `folds` in parallel is extremely simple.

The compiler analysis supports the programmer by automatically handling free-values and generating support code for launching parallel constructs. This gives our system significant advantages over explicit imperative and functional programming methods for parallel systems.

We are able to use our skeletons constructively to build new skeletons from existing ones. The `pfilter` skeleton is an example of this. Furthermore, the programmer is at liberty to define and implement new skeletons. Although adding new skeletons for our system is relatively complex, we can demonstrate potential for a high level of re-usability for such code, depending upon the degree of specialisation of the skeleton's application. For example, we have experimental implementations of:

- A general divide-and-conquer skeleton, similar to the `dcF` skeleton of HDC (Herrmann, 2000).
- An initial version of function composition implemented as a processor pipeline.
- Tuple parallelism, introduced by constructively building upon the `pmap` skeleton.

We have developed applications from a wide variety of domains and have demonstrated good speedup for small numbers of processors.

Several variants of dedicated parallel machines and general-purpose networks of workstations have been targetted by our compiler. Our skeleton deployment is architecturally-independent, within the range of Unix-based machines for which an implementation of MPI exists.

A common objection to HOF-based system development is that programmers find their use forced and unnatural. As part of our compiler project we have also been investigating the automatic synthesis of HOFs in programs that lack them, using proof planning. For example, in the matrix multiplication program discussed above we are able to automatically find 14 different combinations of `map` and `fold`, from a version written without HOFs (Cook *et al.*, 2001; Cook, 2002).

Although these achievements do not conclusively prove the general applicability of our methods to parallel processing, we think that they indicate good potential for evolving a robust and flexible tool for parallel programming applications.

6.2 Limitations

We do not cover the full Standard ML language. While we can compile the full *Core* language, the implementation of the type system is effectively the intersection of the ML Kit and Objective Caml type systems. We also only have limited support for the module system: unconstrained, non-nested structures are the only constructs handled.

Our compiler is *whole-program* with long compilation times which hamper program development. Separate compilation would be hard to implement because of the high volume of additional data attached to grammatical objects.

The network analysis phase uses type theoretic arguments to extract the topology of the parallel execution. Unfortunately, the type information alone is not sufficient to handle higher levels of polymorphism, such as two polymorphic functions applied to each other within a third polymorphic function, for example function composition. A dataflow analysis would be able to handle this.

The defunctionalization method is also partial, for example, we cannot handle higher-order constructors. Furthermore, we were unable to automate the decision between lambda-lifting and defunctionalisation, which at present has to be made manually.

Our skeletons use generic communications which result in high communications loads, so our system is only useful for coarse-grained parallelism. Specialised packing routines such as those from GpH and HDC could be deployed. Alternatively, datatypes which do not require packing are provided by Objective Caml (for instance the `Bigarray` module). The `ptr` and `ptorus` skeletons briefly described in this paper make use of this facility.

We currently use crude, hardwired heuristics for process placement. However, we have implemented a dynamic profiling technique which leads to sequential and parallel performance predictions. Preliminary results suggest that the predictions are sufficiently accurate to allow elimination of non-viable HOF instances. Furthermore, we are currently developing a transformation system for optimising the performance of HOF-based implementations. Integrating the sequential and parallel performance prediction into a feedback loop with this transformation system will provide a metric for program improvements.

In summary, while these limitations pose significant challenges, we do not think that they present fundamental obstacles to the general applicability of the skeletons method or our interpretation of it.

6.3 Reflection

Parallel functional programming (PFP) has had a chequered history; hardly surprising given that its origins lie in the conjunction of two minority tastes. Functional programming (FP) and parallel programming (PP) both have healthy niches, in formally motivated and high performance computing repectively, but neither has yet escaped into mainstream computing. Both have been bedeviled by the over enthusiasm of their proponents, much like Artificial Intelligence, and the inexorable evolution of imperative languages and uni-processors, often absorbing the most adaptive features of their rivals.

It is important to remember that both PP and FP are hard for people raised on high-level sequential imperative programming (SIP). Effective parallel imperative programming (PIP) involves realising an abstract multi-process model using relatively low-level programming construct for inter-process coordination and communication. This is not unlike SIP 50 years ago, in assembly languages and autocodes, before the development of architecture independent abstractions and compilers.

FP appears to offer appropriate abstractions for PP, in particular polymorphism and higher order functions, which might enable the dream of implicit parallelism. However, even sequential FP feels counter-intuitive to those steeped in von Neumannism, in particular the absence of assignment and the corresponding dependence on recursion and structure copying. Furthermore, those aspects of FP which suit it so well for PP also bear an aura of unfamiliar mathematicity.

Real-world computing votes with its feet. If it could be demonstrated convincingly that PFP offered real price/performance advantages over PIP then its adoption would be swift, just as object-orientation has swept away an older generation of structured languages and methodologies.

However, PFP has not enjoyed consistent, methodical development with a single paradigmatic approach emerging. Instead, a hundred schools of PFP contend, barely distinguishable to outsiders, but each with enough investment, both in time and reputation, to prevent convergence. Thus, Loogen (Loogen, 1999), in her major survey of PFP languages, discusses languages supporting control and data parallel variants of implicit, controlled and explicit parallelism. These include automatic parallelism, annotation based languages, para-functional languages, evaluation strategies, skeleton languages, process control languages, message passing languages, concurrent languages, data parallel languages and high level data parallelism. It is striking that Trinder et al's companion survey of PFP applications (Trinder *et al.*, 1999) lists around 30 different languages but the only ones which have found significant use outside of the groups which developed them are single assignment SISAL and Id. We think that PFP is unlikely to come into its own until a COBOL or C or Java emerges. Perhaps we should be pessimistic given that the FP world is still dominated by LISP with SML and Haskell trailing behind.

Nonetheless, we remain pursuaded that FP offers a good basis for PP, that implicit PFP is achievable, and that the skeletons approach is plausible and principled. We also think that seeking parallelism in the standard constructs of a major language avoids the pitfalls of building yet another language or retrofitting an extant language with non-standard extensions.

We have followed a purist skeletons approach, albeit in a highly pragmatic manner. Our compiler design recapitulates all the stages in hand-based skeleton programming: work with a sequential prototype; identify potentially useful parallelism in HOF use through analysis and instrumentation; try to optimise HOF parallelism through transformation; realise HOF parallelism as skeletons. While PMLS is hardly the last word in compiler technology, we think that our design decisions have been broadly appropriate and that PMLS shows that PFP through skeletons has an engaging if challenging future.

Once again, our project highlights a number of well-known broad problems for PP:

- Arbitrary programs seem to offer poor parallel scalability beyond relatively small numbers of processors. What are good techniques for deciding whether it is actually worth trying to parallelise an arbitrary program?

- Naive identification of parallelism offers poor performance gains but optimising parallelism is very hard. What are good measures of potential parallelism? What are good techniques for optimising such parallelism?
- Inter-processor communication bedevils all parallel programming. What are good techniques for minimising data movements between processing phases in parallelising arbitrary programs? Can such data movements be minimised through static analyses at compile time?

Our community makes strong claims about the benefits of functional languages for parallel programming. When we can offer an average programmer the same seamless benefits from parallelism that they take for granted from sequential compilation, then we will have justified those claims.

Acknowledgements

This work is supported by grant number GR/L42889 from the UK's Engineering and Physical Sciences Research Council (EPSRC) and Postdoctoral fellowship P00778 of the Japan Society for the Promotion of Science (JSPS). We also wish to thank the Imperial College Fujitsu Parallel Computing Research Centre for the use of the AP3000 and Quadstone Ltd for access to RPL2. Major compiler components were developed by the University of Copenhagen (ML Kit) and by the Caml Group at INRIA (Objective Caml). We would also like to thank Ryoko Hayashi for the FORTRAN implementation of the Parallel Molecular Dynamics Simulation and Hans-Wolfgang Loidl for the `linsolv` application program and for helpful comments on an earlier draft of this paper.

References

Backus, J. (1978). Can Programming be Liberated from the Von Neumann style? A Functional Style and its Algebra of Programs . *CACM*, **21**(8), 613–641.

Bailey, P., & Newey, M. (1993). Implementing ML for Distributed Memory Multiprocessors. *ACM SIGPLAN Notices*, **28**(1), 59–63.

Beaumont, O., Rastello, F., & Robert, Y. (2001). Matrix Multiplication on Heterogeneous Platforms. *IEEE Transactions on Parallel and Distributed Systems*, **12**(10).

Beazley, D. M., Lomdahl, P. S., Gronbech-Jensen, N., Giles, R., & Tamayo, P. (1995). Parallel Algorithms for Short-range Molecular Dynamics. *Annual Reviews in Computational Physics*, **3**, 119–175.

Bell, J. M., Bellegarde, F., & Hook, J. 1997 (Jun). Type-driven defunctionalization. *Pages 25–37 of: Proceedings of the ACM SIGPLAN ICFP '97*. ACM.

Besl, P.J., & Jain, R.C. (1986). Invariant surface characteristics for 3D object recognition. *Computer Vision, Graphics and Image Processing*, **33**, 33–80.

Bird, R., & de Moor, O. (1997). *Algebra of Programming*. Prentice-Hall.

Bird, R., & Wadler, P. (1988). *Introduction to Functional Programming*. Prentice-Hall.

Birkedal, L., Rothwell, N., Tofte, M., & Turner, D. N. (1993). *The ML Kit (Version 1).* Tech. rept. 93/14. Department of Computer Science, University of Copenhagen.

Bratvold, T. (1994). *Skeleton-based Parallelisation of Functional Programmes.* Ph.D. thesis, Dept. of Computing and Electrical Engineering, Heriot-Watt University.

Busvine, D. (1993). Implementing Recursive Functions as Processor Farms. *Parallel computing*, **19**, 1141–1153.

Canny, J. (1986). A Computational Approach to Edge Detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **PAMI-8**, 679–698.

Chailloux, E., Manoury, P., & Pagano, B. (2000). *Développement d'applications avec Objective Caml.* Paris: O'Reilly.

Chin, W.-N., & Darlington, J. (1996). A Higher-Order Removal Method. *Lisp and Symbolic Computation*, **9**(4), 287–322.

Clarkson, M. I. (1994). *Parallel Implementation of a Range Image Segmentation Algorithm.* M.Phil. thesis, Department of Computing and Electrical Engineering, Heriot-Watt University, Riccarton, Edinburgh, EH14 4AS.

Cole, M. (1993). *Parallel Programming, List Homomorphisms and the Maximum Segment Sum Problem.* Tech. rept. CSR-25-93. University of Edinburgh.

Cole, M. I. (1989a). *Algorithmic skeletons: Structured management of parallel computation.* Pitman.

Cole, M. I. (1989b). *Algorithmic Skeletons: Structured Management of Parallel Computation.* London: Pitman/MIT.

Cook, A. 2002 (Jul). *Using proof in transformation synthesis for automatic parallelisation.* Ph.D. thesis, Dept of Computing and Electrical Engineering, Heriot-Watt University.

Cook, A., Ireland, A., & Michaelson, G. (2001). Higher-order Function Synthesis through Proof Planning. *Pages 307–310 of: Proceedings of 16th Annual International Conference on Automated Software Engineering (ASE 2001).* San Diego, USA: IEEE Computer Society.

Coudarcher, R., Serot, J., & Derutin, J.-P. (2001). Implementation of a Skeleton-based Parallel Programming Environment Supporting Arbitrary Nesting. Meuller, F. (ed), *High-Level Parallel Programming Models and Supportive Environments.* LNCS, vol. 2026. Springer.

Gentleman, W. M. (1978). Some Complexity Results for Matrix Computations on Parallel Processors. *JACM*, **25**, 112–115.

Golub, G. H., & van Loan, C. F. (1989). *Matrix Computations.* Johns Hopkins University Press.

Hamdan, M. (2000). *A Combinational Framework for Parallel Programming Using Algorithmic Skeletons.* Ph.D. thesis, Department of Computing and Electrical Engineering, Heriot-Watt University.

Hamdan, M., Michaelson, G., & King, P. (2000). A Framework for Nesting Algorithmic Skeletons. *Pages 542–549 of: Proceedings of 1999 International Conference on Parallel Computing ParCo'99.* Delft, Holland: Imperial College Press.

Hammond, K., & Michaelson, G. (1999). *Research Directions in Parallel Functional Programming.* Springer.

Hayashi, R., & Horiguchi, S. (1997). Domain Decomposition Scheme for Parallel Molecular Dynamics Simulation. *Pages 595–600 of: Proc. of High Performance Computing Asia '97.* IEEE CS Press.

Heermann, D. D. (1990). *Computer Simulation Methods in Theoretical Physics.* 2nd edition edn. Springer, Tokyo.

Herrmann, C. A. (2000). *The Skeleton-Based Parallelization of Divide-and-Conquer Recursions.* Ph.D. thesis, Universität Passau.

Herrmann, C. A., & Lengauer, C. (2000). The HDC Compiler Project. *Pages 239–254 of:* Darte, A., Silber, G.-A., & Robert, Y. (eds), *Proc. Eighth Int. Workshop on Compilers for Parallel Computers (CPC 2000).* LIP, ENS Lyon.

Herrmann, C. A., Laitenberger, J., Lengauer, C., & Schaller, C. (1999). Static Parallelization of Functional Programs: Elimination of Higher-Order Functions & Optimized Inlining. *Pages 930–934 of:* Amestoy, Patrick, Berger, Philippe, Daydé, Michel, Duff, Iain, Frayssé, Valérie, Giraud, Luc, & Ruiz, Daniel (eds), *Euro-par'99.* LNCS 1685. Springer.

Hu, Z., Iwasaki, H., & Takeichi, M. (1997). Formal Derivation of Efficient Parallel Programs by Construction of List Homomorphisms. *ACM Transactions on Programming Langauges and Systems*, **19**(3), 444–461.

J. Darlington, Y-K. Guo, & To, H. W. (1996). Structured Parallel Programming: Theory meets Practice. *Pages 49–65 of:* Wand, I., & Milner, R. (eds), *Computing Tomorrow: Future Research Directions in Computer Science.* CUP.

Johnsson, T. (1985). Lambda Lifting: Transforming Programs to Recursive Equations. *Pages 190–302 of:* Jouannaud, J.-P. (ed), *Functional Programming Languages and Computer Architecture.* LNCS, vol. 201. Springer.

Kalé, L. V., Bhandarkar, M., & Brunner, R. (1998). Load Balancing in Parallel Molecular Dynamics. *Fifth International Symposium on Solving Irregularly Structured Problems in Parallel.* LNCS, vol. 1457.

Kesseler, M. (1995). Constructing Skeletons in Clean The Bare Bones. *Pages 182–192 of:* Bom, A. P. Wim, & Feo, J. T. (eds), *High Performance Functional Computing.* .

Klusik, U., Loogen, R., Priebe, S., & Rubio, F. (2000). Implementation Skeletons in Eden: Low-Effort Parallel Programming. *12th Int. Workshop on Implementation of Functional Languages (IFL 2000).* LNCS. Springer.

Koutsakis, G. 1993 (Oct). *Parallel Low Level Vision from Functional Prototypes.* M.Phil. thesis, Dept. of Computing and Electrical Engineering, Heriot-Watt University, Riccarton, Edinburgh.

Leavers, V. F. (1993). Survey: Which Hough Transform? *CVGIP: Image Understanding*, **58**(2), 250–264.

Loidl, H-W., Trinder, P.W., Hammond, K., Junaidu, S.B., Morgan, R.G., & Peyton Jones, S.L. (1999). Engineering Parallel Symbolic Programs in GPH. *Concurrency - Practice and Experience*, **11**, 701–752.

Loidl, H-W., Rubio, F., Scaife, N., Hammond, K., Horiguchi, S., Klusik, U., Loogen, R., Michaelson, G. J., Peña, R., Portillo, Á. J. Rebón, Priebe, S., & Trinder, P. W. (2003). Comparing Parallel Functional Languages: Programming and Performance. *Higher-order and Symbolic Computation. To Appear.*

Loogen, R. (1999). Programming Language Constructs. *Chap. 3, pages 63–92 of:* Hammond, K., & Michaelson, G. (eds), *Research Directions in Parallel Functional Programming.* Springer.

McAndrew, P., & Wallace, A. M. 1989 (Jul). Rapid Invocation and Matching of 2D Images to 3D Models using Curvilinear Data. *Pages 83–87 of: Proc. of 3rd Int. Conf. on Image Processing and its Applications.*

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

Michaelson, G., & Scaife, N. 2000 (Sep). Parallel Functional Island Model Genetic Algorithms through Nested Skeletons. *Pages 307–313 of:* Mohnen, M., & Koopman, P. (eds), *Proceedings of 12th International Workshop on the Implementation of Functional Languages.*

Michaelson, G., & Scaife, N. (2002). *Skeletons and Patterns for Parallel and Distributed Computing.* Wiley. Chap. Skeleton Realisations from Functional Prototypes, pages 129–153.

Michaelson, G., Ireland, A., & P.King. 1997 (Sept). Towards a Skeleton Based Parallelising Compiler for SML. *Pages 539–546 of:* Clack, C., Davie, T., & Hammond, K. (eds), *Proceedings of 9th International Workshop on Implementation of Functional Languages.*

Michaelson, G., Scaife, N., Bristow, P., & King, P. (2001). Nested Algorithmic Skeletons from Higher-Order Functions. *Parallel Algorithms and Applications special issue on High Level Models and Languages for Parallel Processing*, **16**(2-3), 181–206.

Michaelson, G. J., & Scaife, N. R. (1995). Prototyping a parallel vision system in Standard ML. *Journal of Functional Programming*, **5**(3), 345–382.

Milner, R., Tofte, M., & Harper, R. (1990). *The Definition of Standard ML.* MIT Press.

Milner, R., Tofte, M., Harper, R., & MacQueen, D. (1997). *The Definition of Standard ML (Revised).* MIT Press.

Pelegatti, S. (1998). *Structured Development of Parallel Programs.* Taylor and Francis.

Pierce, B. C., & Turner, D. N. (2000). Pict: A Programming Language Based on the Pi-Calculus. Plotkin, G., Stirling, C., & Tofte, M. (eds), *Proof, Language and Interaction: Essays in Honour of Robin Milner.* MIT Press.

Rangaswami, R. (1995). *A Cost Analysis for a Higher-Order Parallel Programming Model.* Ph.D. thesis, University of Edinburgh.

Reynolds, J. C. (1972). Definitional Interpreters for Higher-order Programming Languages. *ACM National Conference*, 717–740.

Scaife, N., Bristow, P., Michaelson, G., & King, P. 1998 (Sep). Engineering a Parallel Compiler for SML. *Pages 213–226 of:* Clack, C., Davie, T., & Hammond, K. (eds), *Proceedings of 10th International Workshop on Implementation of Functional Languages.*

Scaife, N., Michaelson, G., & Horiguchi, S. (2002a). *13th International Workshop, Implementation of Functional Languages.* LNCS, vol. 2312. Springer. Chap. Comparative Cross-Platform Performance Results from a Parallelizing SML Compiler, pages 138–154.

Scaife, N., Hayashi, R., & Horiguchi, S. 2002b (Sep). Parallel Molecular Dynamics in a Parallelizing SML Compiler. *Pages 447–454 of: The Third International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'02).*

Scaife, N. R. (1996). *A Dual Source, Parallel Architecture for Computer Vision.* Ph.D. thesis, Department of Computing and Electrical Engineering, Heriot-Watt University.

Scaife, N. R., Michaelson, G. J., & Wallace, A. M. 1997 (Sep). Four skeletons for computer vision. *Implementation of functional languages '97.*

Serot, J. (1999). Explicit Parallelism. *Pages 379–396 of:* Hammond, K., & Michaelson, G. (eds), *Research Directions in Parallel Functional Programming.* Springer.

Serot, J. (2001). Tagged-token Data-flow for Skeletons. *HLPP 2001: International Workshop on High-Level Parallel Programming and Applications.* Parallel Processing Letters. World Scientific Publishing.

Surry, P. (1993). *RPL2 Functional Specification.* Tech. rept. EPCC-PAP-RPL2-FS 1.0. University of Edinburgh/British Gas.

Trinder, P., Loidl, H-W., & Hammond, K. (1999). Large Scale Functional Applications.

Chap. 19, pages 399–426 of: Hammond, K., & Michaelson, G. (eds), *Research Directions in Parallel Functional Programming*. Springer.

Trinder, P. W., Hammond, K., Loidl, H-W., & Peyton-Jones, S. L. (1998). Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, **8**(1), 23–60.

Wallace, A., Michaelson, G., McAndrew, P., Waugh, K., & Austin, W. (1992). Dynamic control and prototyping of parallel vision algorithms. *Ieee computer special issue on parallel processing for computer vision and image understanding*, **25**(2), 43–53.

Wallace, A. M., Michaelson, G. J., Scaife, N., & Austin, W. J. (1998). A Dual Source, Parallel Architecture for Computer Vision. *The Journal of Supercomputing*, **12**(1/2), 37–56.