JFP **15** (4): 615–650, 2005. © 2005 Cambridge University Press
doi:10.1017/S0956796804005489 Printed in the United Kingdom

A parallel SML compiler based on algorithmic skeletons

NORMAN SCAIFE, SUSUMI 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 with 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 backend 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.

Capsule Review

This paper provides a detailed description of a parallelizing SML compiler, which the authors have been developing over several years. The main objective of this work is to avoid runtime mechanisms by mapping out skeleton usage statically, in the form of a tree representing the control flow as a nest of sequentially-composed parallel skeleton calls. This paper represents a huge effort to test these ideas on a range of application examples.

1 Introduction

We present the construction of a parallel compiler for Standard ML based on *algorithmic skeletons*; abstractions over common patterns of parallel computation represented as higher-order functions in the source language. Our ultimate goal is to make the use of this compiler *fully automatic* and the work presented here is a significant step in this direction. Our specific technical contributions are:

- Fully automatic introduction of parallel constructs for sequential ones.
- *Static* generation of abstract network properties resulting in efficient use of runtime resources.
- Automatic handling of *non-functional* free variables of skeleton HOFs.
- *Optional* handling of functional free values using defunctionalization.
- Creation of a platform for building profiling and transformation tools.
- A degree of portability and architectural independence by the use of standardized communications and source languages.

Of these the most important contribution is in the abstract network analysis since this gives our compiler some advantages over systems which use dynamic parallelism at runtime.

2 Motivation

2.1 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 existing languages lack appropriate abstractions for parallelism and dedicated parallel languages 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 (McColl, 1993).

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, 1989). Essentially, a skeleton is a template for generic parallel activity.

For example, in task farming (Choi *et al.*, 1996), 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

https://doi.org/10.1017/S0956796804005489 Published online by Cambridge University Press

are running 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 (Scaife *et al.*, 2002a). In addition, skeletons are suitable for both regular and irregular parallel applications. Regular problems have been well-studied and compilers such as the various parallel Fortrans have expoited regular parallelism in a wide range of applications. Using algorithmic skeletons, however, allows a wider variety of parallel structures to be more easily investigated using suitable levels of abstraction.

2.2 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 accommodating skeletons in existing languages have proved more popular, especially where such languages are used 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).

Pure functional languages such as Haskell and pure functional subsets of languages such as Standard ML 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. The point is that although HOFs themselves can be fine-grained or coarse-grained with respect to the parallel machine they bring the *control* over granularity to the level of HOF selection in the source language.

HOFs are often 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` and `fold`. A `map` may be realised as a task farm, discussed above. Similarly, a `fold` for an associative argument function may be realised as a divide-and-conquer tree. 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

https://doi.org/10.1017/S0956796804005489 Published online by Cambridge University Press

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 language, implementing multi-processor behaviour using language constructs, for example channels in Clean (Kesseler, 1995) and Eden (Klusik *et al.*, 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). The main objective of this work is to build a compiler for which all parallelism is *implicit*. We use algorithmic skeletons 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.

2.3 Skeleton realization

Throughout this paper we illustrate the operation of our compiler on the following 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. We do not give a full definition of `fold` here, which is actually implemented sequentially using a split and merge method to ensure the same computational complexity as the parallel equivalent `pfold`, discussed below. `fold` is conceptually very like the `foldl` and `foldr` functions from the SML Basis library but less general. This can be seen in the type signatures:

```
val fold : ('a * 'a -> 'a) -> 'a -> 'a list -> 'a
val foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```

https://doi.org/10.1017/S0956796804005489 Published online by Cambridge University Press

In this example, `fold` is 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 `ff`, if the parallel implementation is to return the same result as the sequential implementation. The value `~x` is an identity for `ff`, where `~` is the unary minus operator in SML.

Given these conditions, which have to be met to allow the semantics to be preserved under parallel implementation there are still a number of problems which have to be overcome during compilation:

- To replace a HOF like `fold` with a parallel version like `pfold` the compiler needs to check that the context of application is suitable for parallel implementation.
- In a parallel implementations of HOFs as skeletons, free values, like `x` above, have to be transmitted to the remote environment prior to executing the function argument, like `ff` above.
- Furthermore, if there are free functional values, it may not be possible to directly transmit them.
- The runtime system should know what skeletons are going to be executed and in what order to allow efficient allocation of resources.

The following sections describe the realization of the PMLS (Parallelising ML with Skeletons) compiler that solves, at least in part, all of these problems. PMLS is an automatically parallelising compiler for a substantial subset of the strict language SML. We regard a strict language as more suitable for many aspects of our approach than a lazy language because of the more predictable evaluation order.

This paper starts by focusing on our compiler's mechanisms for handling the problems of topological analysis and data locality identified above, that arise from the presence of parallelisable HOFs. We then discuss the behaviours of five exemplars on four parallel architectures. Finally, we assess the achievements and limitations of our compiler, and consider future developments.

3 Compiler overview

The initial design of the compiler has been presented in Michaelson *et al.* (1997) and Scaife *et al.* (1998). We decided to use existing software as far as possible for the front end (lexical, syntax and type analysis) and back end (code generation) 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 128 nodes and 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),

https://doi.org/10.1017/S0956796804005489 Published online by Cambridge University Press

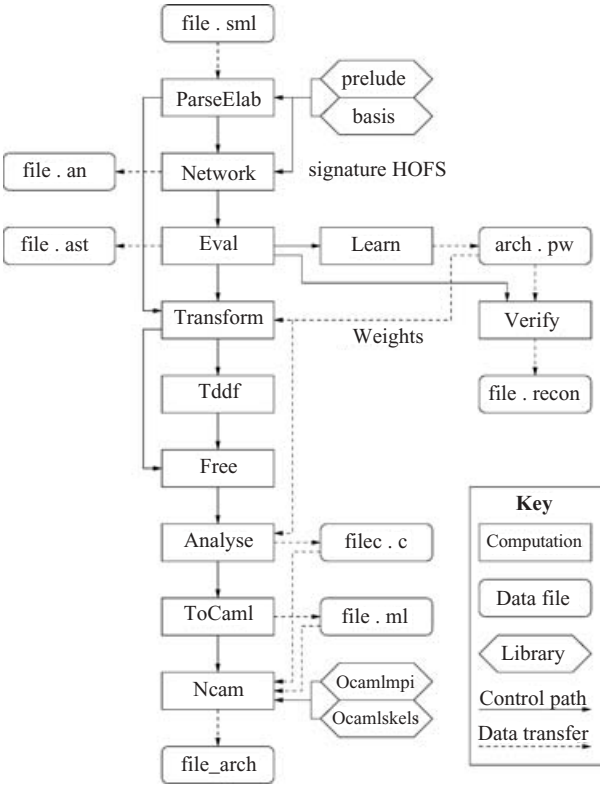

Fig. 1. The PMLS compiler modules.

which used Objective Caml as the host language, we decided to employ 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 interpreter capable of being modified to gather dynamic profiling information. The interpreter used by SkelML (Bratvold, 1994) was for a small SML subset where 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. The structure of the compiler is shown in Figure 1, and is organised as a series of modules:

ParseElab reads the input SML and lexically analyses, parses and performs type inference upon 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 be the

https://doi.org/10.1017/S0956796804005489 Published online by Cambridge University Press

relationship between HOFs in the source code and between skeletons in the executing program. This module uses the type information generated by the elaborator. It is also pivotal for the success of our compiler.

Tddf is an optional module which attempts to defunctionalise (Bell *et al.*, 1997) the entire program. Our runtime 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 runtime.

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 runtime. An attempt is made to limit the transmission of such values to those which will actually be required for remote computation. Note that if defunctionalisation is performed then free-variable analysis is not required since free values are bound up in the `datatype` representations.

Analyse is the backend 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. `Ncam` is not a module in the main compiler but is actually an external Perl script which calls the appropriate Objective Caml and C compilers, generating the correct compiler flags.

The modules **Eval**, **Learn**, **Verify** and **Transform** form the core of our performance modelling and transformational system. This feature is not discussed further in this paper and is left as a potential enhancement of our compiler:

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 is driven by the performance prediction.

The resulting language is based upon a pure functional subset of SML with some restrictions upon the syntax and type systems due to the translation into Objective Caml. References can be used outside of skeletal execution and within skeleton instance functions but access to references must not cross HOF/skeleton boundaries.

4 Runtime system overview

At runtime, we use an SPMD approach based on MPI Version 1 (Message Passing Interface Forum, 1994). Here, an executing program is a set of identical processes

https://doi.org/10.1017/S0956796804005489 Published online by Cambridge University Press

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.

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 processes. Here, all non-leaf processes 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 process tree. Parallel nesting is implemented solely from the leaf processes, i.e. if a skeleton HOF is encountered during execution of the `pfold` instance function it is only implemented in parallel on leaf nodes. If a nested instance is encountered from an intermediate node the skeleton is executed sequentially.

We identify three limitations to these skeletons. First of all, both 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. Furthermore, 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 applying the fold instance function to the partial results at each node. Finally, 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.

Our skeletons are implemented in C using the Objective Caml foreign function interface to access Objective Caml functions on remote processors. Thus the parallel implementation of the simple example in section 2.3 would be implemented as:

https://doi.org/10.1017/S0956796804005489 Published online by Cambridge University Press

```
val _ = register "ff" ff
val result = pfold "ff" (~x) [1,2,3]
```

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 (`named_value`) is provided by the runtime system. 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.

5 PMLS structure

Here we give more detailed descriptions of the compiler modules outlined in Figure 1.

5.1 `ParseElab`: Front-end and library support

The ML Kit 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 by Core expressions. The simple example in section 2.3 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 Figure 1.

5.2 `Network`: Network analysis

We need to map the program execution onto a static network of processors. To simplify this process we generate an *abstract network* description of the program. This 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. It is implemented by the `Network` phase in Figure 1.

5.2.1 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. This method is not as general

https://doi.org/10.1017/S0956796804005489 Published online by Cambridge University Press

or flexible as programmable skeletons such as those in Eden or GpH (Loidl *et al.*, 2003), but has some advantages in terms of enabling static resource allocation giving some performance advantages over such systems.

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

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

5.2.2 Abstract network description

The abstract network (Figure 2) is a tree consisting of nodes; `baseAn` indicating no skeleton instances, `nodeAn` indicating an instance of a (potentially nested) skeleton, `seqAn` indicating a set of skeleton instances where each skeleton must terminate before the next one can start and `altAn` is skeleton alternation. Note that a `seqAn` 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.

At present we cannot handle alternation in our current compiler and so each branch of a *match* construct must either have the same structure (e.g. single `map` or nested `map map`) as all the other branches or have no skeleton instances at all. The `altAn` constructor is required, however, since potentially useful skeletons often appear within conditional constructs. The term *match* refers to the *match* grammar object in the Standard ML definition. In the `Core` language, this is the only source of conditional execution. 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.

5.2.3 Abstract network extraction

Abstract network extraction is divided into two operations:

https://doi.org/10.1017/S0956796804005489 Published online by Cambridge University Press

```
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. 2. `datatype absnet`: Abstract network type representation.

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. 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. Note that *dec*, *pat*, etc. do not have types.

5.2.4 *Abstract network type*

We start our inference from the existing type information generated by the ML Kit type inference and augment this with information about skeleton applications. Thus our absract network description, shown in Figure 2, includes a complete description of the native SML types. For example the type of constructors is encoded in the `constypeAn` constructor. Similarly, `rectypeAn` encodes records (and tuples), `arrowAn` encodes functional types and `tyvarAn` encodes type variables. This

https://doi.org/10.1017/S0956796804005489 Published online by Cambridge University Press

information is intended to be removed once the abstract network has been generated. The `nodeAn` constructor has several fields:

`hof` is the name of the identified HOF.
`hofAn` represents the type of the node and is used during function application and constructor dereference.
`nest` represents 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.

The additional `errorAn` constructor 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.

5.2.5 *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 *valbind*s (value bindings) 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:

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

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

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

Figure 3 outlines a schema, $\mathscr{T}$, 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

https://doi.org/10.1017/S0956796804005489 Published online by Cambridge University Press

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

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

Fig. 3. Schema $\mathscr{T}$: Traverse AST, annotate with HOF information.

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 $\mathscr{T}$ schema are AST grammar objects and the parameters of some of the objects have been omitted for brevity.

The $\mathscr{T}$ schema calls several other schemas. Schema $\mathscr{M}$ is a type unification algorithm for abstract network types. Schema $\mathscr{S}$ applies the abstract network type substitutions generated by $\mathscr{M}$ to grammar objects. Schema $\mathscr{A}$ is a combination of $\mathscr{M}$ and $\mathscr{S}$ which gives the result of applying one abstract network type to another. Schema $\mathscr{N}$ 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 (exception) into a *tyvar* (type variable) which is unified by the $\mathscr{T}$ algorithm,

https://doi.org/10.1017/S0956796804005489 Published online by Cambridge University Press

```
𝒮c : absnet → absnet → esn → esn
𝒮c (errorAn msg)        _                      esn = adde (errorAn msg) esn
𝒮c baseAn               _                      esn = esn
𝒮c (nodeAn node)        (arrowAn _)            esn = addn (nodeAn node) esn
𝒮c (nodeAn node)        (tyvarAn _)            esn = addn (nodeAn node) esn
𝒮c (nodeAn node)        _                      esn = adds (nodeAn node) esn
𝒮c (constypeAn (a,n))   (constypeAn (a',n'))   esn = 𝒮c a a' esn,
                                                     if n = n'
𝒮c (rectypeAn r)        (rectypeAn r')         esn = 𝒮c r r' esn
𝒮c (arrowAn (a,r))      (arrowAn (a',r'))      esn = 𝒮c r r' (𝒮c a a' esn)
𝒮c (tyvarAn tv)[a]      (tyvarAn tv')          esn = esn,
                                                     if tv = tv'
𝒮c nilAn                nilAn                  esn = esn
𝒮c (rowRecAn (l,a,r))   (rowRecAn (l',a',r'))  esn = 𝒮c r r' (𝒮c a a' esn),
                                                     if l = l'
```

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

Fig. 4. Schema $\mathscr{S}_c$: Compare instance of HOF with definition.

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 absnets 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 $\mathscr{T}$ schema. These are scanned separately by the patabsnets routine, giving absnets for bound identifiers. Syntax types are handled in exactly the same manner as semantics types.

5.2.6 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 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 $\mathscr{S}_c$ (Figure 4) compares the instance absnet with the original HOF absnet, generating a list of nested constructs and a list of sequenced constructs. The schema $\mathscr{S}_t$ (Figure 5) takes an absnet and strips out non-essential type information, calling $\mathscr{S}_c$

https://doi.org/10.1017/S0956796804005489 Published online by Cambridge University Press

$\mathscr{S}_t$: absnet → absnet			
$\mathscr{S}_t$ (errorAn msg)[a]	=	baseAn	
$\mathscr{S}_t$ baseAn	=	baseAn	
$\mathscr{S}_t$ (nodeAn {hof,hofAn,nest})	=	skelAn,	*if* s = []
	=	seqAn (skelAn::s),	*otherwise*
where		skelAn = nodeAn {hof,hofAn,n}	
and		instAn = $\mathscr{L}$ (nest @ ($\mathscr{E}$ hofAn))	
and		(e,s,n) = $\mathscr{S}_c$ instAn defAn[b] ([],[],[])	
$\mathscr{S}_t$ (seqAn ans)	=	baseAn,	*if* $\mathscr{S}_t$ ans = seqAn [][c]
	=	$\mathscr{S}_t$ ans,	*otherwise*
$\mathscr{S}_t$ (constypeAn ([],_))	=	baseAn[d]	
$\mathscr{S}_t$ (constypeAn (ans,name))	=	seqAn($\mathscr{S}_t$ ans)[e]	
$\mathscr{S}_t$ (rectypeAn r)	=	baseAn,	*if* $\mathscr{S}_t$ r = baseAn *or* $\mathscr{S}_t$ r = seqAn []
	=	$\mathscr{S}_t$ r,	*otherwise*
$\mathscr{S}_t$ (arrowAn (a,r))	=	baseAn,	*if* $\mathscr{S}_t$ a = baseAn *and* $\mathscr{S}_t$ r = baseAn[f]
	=	arrowAn($\mathscr{S}_t$ a,$\mathscr{S}_t$ r),	*otherwise*
$\mathscr{S}_t$ (tyvarAn tv)	=	tyvarAn tv[g]	
$\mathscr{S}_t$ nilAn	=	baseAn	
$\mathscr{S}_t$ (rowRecAn (lab,an,r))	=	seqAn(($\mathscr{S}_t$ an) @ ($\mathscr{S}_t$ 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 $\mathscr{S}_t$ 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. 5. Schema $\mathscr{S}_t$: Strip out unnecessary type information.

when a node is encountered. In addition, both functions accumulate errorAn types into a separate list, replacing them with baseAn abstract network types.

Figure 4 outlines the $\mathscr{S}_c$ schema. Prior to scanning two absnets 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 $\mathscr{S}_t$ schema, shown in Figure 5. 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:

1. The schema $\mathscr{L}$ turns a list of absnets into a function type:
   $\mathscr{L}$ [a,b,c] = arrowAn (a,arrowAn (b,c))

https://doi.org/10.1017/S0956796804005489 Published online by Cambridge University Press

2. The schema $\mathscr{E}$ removes all node wrappers, recursively, from an `absnet` and is applied to the result type of `nodeAns` prior to comparison with the original definition.

5.2.7 *Abstract network examples*

The simple example of section 5 results in the following abstract network types:

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

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:

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

5.3 *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 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 free values have to made available to the remote instance. Since we cannot transmit functional values 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 5, the value `x` does not need to be transmitted since it is generated on all processors during program startup.

https://doi.org/10.1017/S0956796804005489 Published online by Cambridge University Press

```
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. 6. Example showing transmission of free values.

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.

The example in Figure 6 illustrates how this is handled in our approach. The Objective Caml foreign function interface 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. Thus 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.

https://doi.org/10.1017/S0956796804005489 Published online by Cambridge University Press

The problem of functional free values is much harder to solve. In the above case, the functional free value `ff` is handled 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. 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. Generation of bindings for functions by inlining was partially implemented but subsequently replaced with defunctionalisation where all values can be treated as for the non-functional free value `y` in the example above.

5.4 `Tddf`: Defunctionalisation

Defunctionalisation is based on an idea by John Reynolds (Reynolds, 1972) for encoding functional arguments. It has been further developed (Chin & Darlington, 1996), and subsequently generalised to cover a full functional language (Bell *et al.*, 1997). Defunctionalisation has also been used in the HDC compiler project (Herrmann & Lengauer, 2000), also motivated by the need to handle free functionals in a skeletal functional language.

In this technique, closures (in the sense of partially applied functions) are 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 defunctionalisation of SML. All of the previous algorithms make use of forward references in the code. 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. This has some problems, for instance it slows

https://doi.org/10.1017/S0956796804005489 Published online by Cambridge University Press

```
(* 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. 7. Defunctionalisation example illustrating a `fold` skeleton wrapper.

down the backend analysis and does not exactly preserve the semantics of the original program.

We do not present any details of our analysis which is directly implemented from previous work. Instead, we present a simple example of the output from our analysis. Figure 7 shows the example of section 5 with a functional `fold` and the wrapper code generated for it.

This method of defunctionalisation 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. However, some classes of programs can be handled by free-value analysis alone. The selection of these two, complementary methods has not been automated in our compiler.

5.5 *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. We use the `register`/`named_value` mechanism described in section 5.3 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. In particular, HOFs must be nominated by the `signature`

https://doi.org/10.1017/S0956796804005489 Published online by Cambridge University Press

HOFS mechanism, type information has to be reconstructed at the point of call, and 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 compared. If a `string` type in the parallel HOF type signature matches a functional type in the sequential HOF type signature then that function argument is a skeleton instance function. To see this, consider the simple example in section 5 where 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, `('a*'a->'a)` matches `string` so `ff` is the instance expression. The instance expression is bound to a new unique function name, say `ff1`, and registered with the runtime system under a unique name (`"ff1"`). The instance expression (`ff`) 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 in the example the instance expression type (`int*int->int`) and HOF type (`'a*'a->'a`) 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 system 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))
```

5.6 Conclusion

We have described in some detail the phases of the PMLS compiler, focusing on network abstraction and the handling of free values through lifting or defunctionalisation.

https://doi.org/10.1017/S0956796804005489 Published online by Cambridge University Press

Table 1. *MIMD target systems*

System	CPU	PEs	Speed	CR[a]	OS	MPI
Fujitsu AP3000	UltraSparc	16	300 MHz	1.50	Solaris 2.6	MPIAP
IBM RS/6000 SP2	Power PC	16	332 MHz	2.21	AIX 3.4	MPCC
Beowulf	Celeron	32	533 MHz	5.33	Linux 2.2.16	LAM
Sun StarCat 15000	UltraSparc III	32	900 MHz	0.06	Solaris 2.8	MPICH

[a] Communications/computation ratio (MHz/(MB/s)).

The construction of PMLS proved a major Software Engineering activity. This required the integration of extant systems with new language processors, both through component modification and the production of specialised transducers. The success of PMLS is, in part, a tribute to the ML Kit and Objective Caml, which have proved robust and reliable foundations.

In the next section, we discuss the performance of SML programs for Euler totient, matrix multiplication, genetic algorithm, linear equation solving and molecular dynamics, parallelised through PMLS and run Fujitsu AP3000 and run on IBM SP2, Beowulf and Sun StarCat architectures. These exemplars display good performance consistency across platforms and illuminate both advantages and limitations of PMLS.

6 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 Table 1. The CR column gives a rough estimate of the ratio of processing speed to peak communications speed. The three distributed memory machines are broadly similar but the shared memory machine has very different characteristics and cannot be directly compared with the others.

None of the examples requires defunctionalisation 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.

6.1 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`, this having been determined by hand-modification and direct measurement. The sequential algorithm can be expressed as:

```
fun gcd x 0 = x
  | gcd x y = gcd y (x mod y)
```

https://doi.org/10.1017/S0956796804005489 Published online by Cambridge University Press

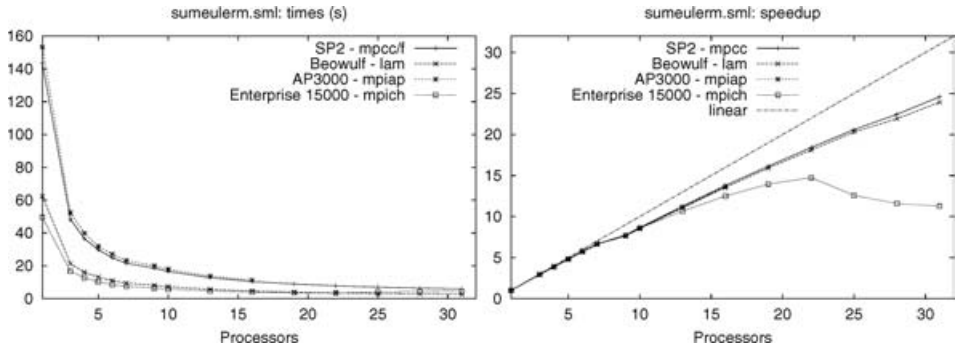

Fig. 8. Runtime and speedup for sum of Euler numbers. The input list is 7900 arbitrary precision integers from 100 to 8000.

```
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 `nList` is the input list of integers and `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 (E 0) (seqmap I nList)
```

The function `seqmap` is an implementation of `map` which is not parallelised 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 in Figure 8. The runtimes are very similar, given the different speeds of the processors and the relative speedups are identical apart from the Enterprise which tails off badly at higher numbers of processors.

6.2 Matrix multiplication

There are many approaches to matrix multiplication in parallel including block-structured and systolic algorithms (Golub & van Loan, 1989). Naïve approaches using functional `maps` over elements and submatrices have been implemented (N.Scaife *et al.*, 2002a) but give poor speedup.

https://doi.org/10.1017/S0956796804005489 Published online by Cambridge University Press

```
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
```

Fig. 9. Implementation of Gentleman's algorithm using `ptorus`.

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.

Figure 9 illustrates the top level code. The $r \times c$ matrices have to be built into the four-dimensional array `AB` where each matrix is divided up into $m \times n$ sub-matrices. 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. Figure 10 shows the performance of this skeleton. Here, we get slightly slower runtimes than with the naïve versions 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.

6.3 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

https://doi.org/10.1017/S0956796804005489 Published online by Cambridge University Press

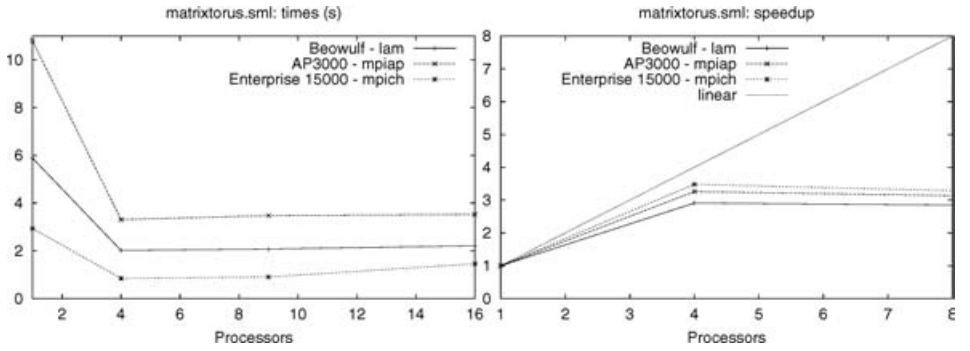

Fig. 10. Runtime and speedup for matrix multiplication. The input is a 300×300 matrix of random native integers from 0 to 65535.

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 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 counter-balanced. 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. Figure 11 shows the performance for this application. Again, reasonable speedups

https://doi.org/10.1017/S0956796804005489 Published online by Cambridge University Press

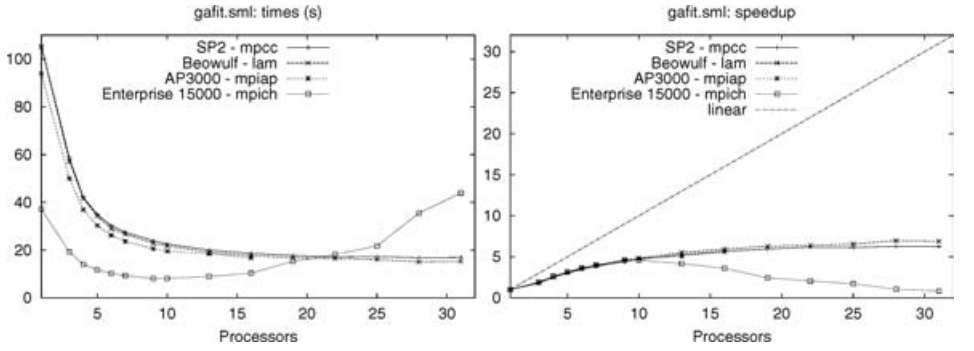

Fig. 11. Runtime and speedup for parallel GA. The input is profiling data consisting of 184 measurements of 38 parameters, the GA runs for 10 iterations.

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

6.4 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 has been discussed in detail (Loidl *et al*., 1999). The original algorithm was coded in Haskell and has been 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.

Figure 12 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.

https://doi.org/10.1017/S0956796804005489 Published online by Cambridge University Press

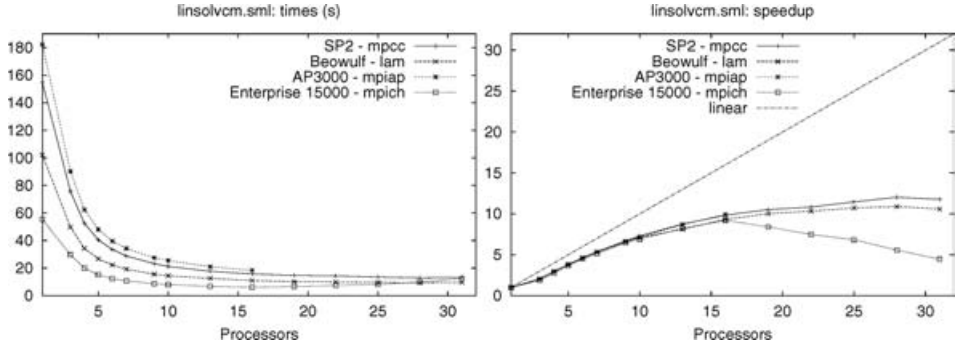

Fig. 12. Runtime and speedup for linsolv. This input is a 40×40 matrix of randomly generated, arbitrary precision integers from 0 to 10^{10}.

6.5 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 are 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 naïve $O(N^2)$ 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.

https://doi.org/10.1017/S0956796804005489 Published online by Cambridge University Press

```
fun fc (acc,(qi,vi,fi),(qj,vj,fj)) =
  let
    val qq = limit3 (sub3 (qi,qj),~s2,s2)
    val rr = ssq3 qq
    val (r6,r12) = (rr*rr*rr,rr*rr*rr*rr*rr*rr)
    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
```

Fig. 13. Functional implementation of PMD simulation.

Here, we present two implementations. The first is purely functional 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 in which symmetry is used to reduce the complexity of the algorithm by a factor of 2.

Figure 13 shows the central code for this implementation. 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.

Figure 14 (`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 6.2. 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 data represent the same set of particles (for computation of forces between local

https://doi.org/10.1017/S0956796804005489 Published online by Cambridge University Press

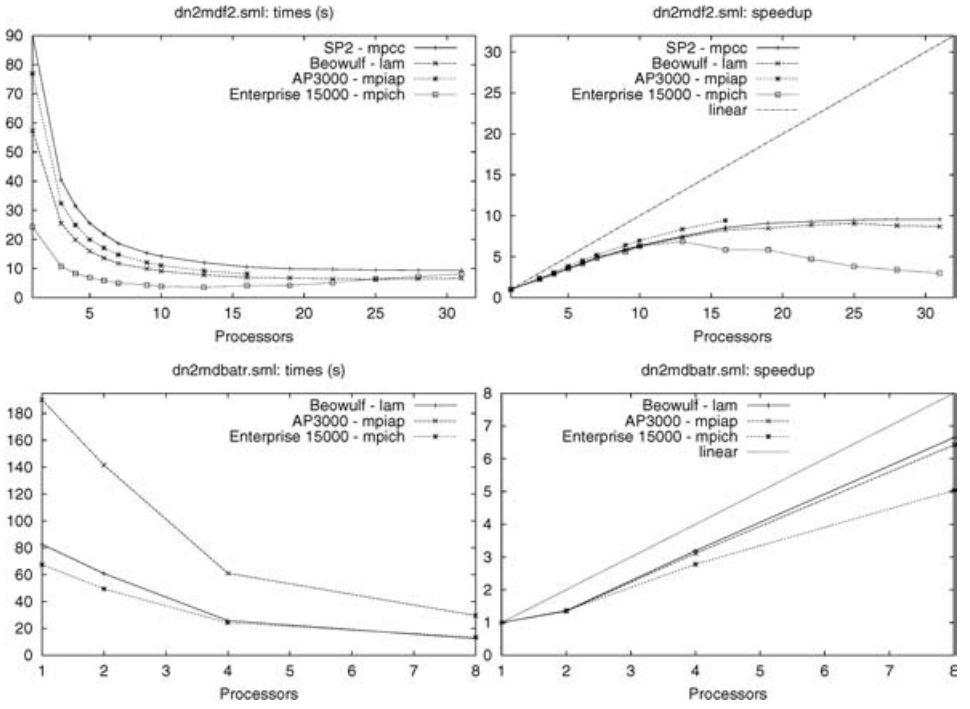

Fig. 14. Runtime and speedup for parallel molecular dynamics simulation. The upper plots (`dn2mdf2.sml`) are for the functional code with 216 particles and 16 time steps. The lower plots (`dn2mdbatr.sml`) are for the ring topology implementation for 512 particles over 200 time steps.

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.

Figure 14 (`dn2mdbatr.sml`) shows the performance of this method for 512 particles over 200 time steps. 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. 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. There is good speedup up to 8 processors (the number of processors must be a divisor of the cube root of the number of particles) speedups are 4.23s for FORTRAN90 and 3.13s for SML.

6.6 Summary

The examples discussed above 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 4 and relate to the relatively poor communcation to computation ratio of

https://doi.org/10.1017/S0956796804005489 Published online by Cambridge University Press

Table 2. *Fastest runtime T (seconds) and peak speedup SP for all applications*

Program	Beowulf		SP2		AP3000		StarCat	
	T	SP	T	SP	T	SP	T	SP
`sumeulerm`	2.606	23.94	5.831	24.60	11.22	13.66	3.360	14.74
`matrixtorus`	18.22	2.937	N/A	N/A	29.47	3.334	0.845	3.481
`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

our skeletons. Table 2 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 re-use can be achieved, with consistent behaviour across a variety of MIMD architectures.

7 Related approaches

Glasgow Parallel Haskell (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 can sometimes require explicit control of granularity and data locality using user-defined strategies.

The HDC compiler project (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 several variants are provided. HDC uses the same lambda-lifting method as our compiler. Defunctionalisation is also used but works using a *monomorphization* approach to handling polymorphism. HDC is slightly more expressive than our compiler, for

https://doi.org/10.1017/S0956796804005489 Published online by Cambridge University Press

instance list comprehensions are transformed into loops and filters which can be exploited by skeleton compilation. Nesting is not explicitly mentioned.

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.

The P^3L compiler (Bacci *et al*., 1995) allows the construction of parallel programs from a set of task and data parallel skeletons. Data parallel skeletons include `map` and `reduce` and task parallel skeletons include `pipe` and `farm` but there are restrictions upon which type of skeleton can be nested within each other, for example task parallel skeletons cannot be nested within data parallel ones. A cost model is also provided which mirrors the compositional nature of the compiler. It is intended that the cost model be used to *fine tune* the implementation which requires the cost model to be *congruent*, ie. accurate across optimizations. The cost model is quite simple, modelling communications as a linear function of data size and the costs for task parallel skeletons are divided into execution time and *service* time which is the average steady-state cost of an individual task. This compiler also has many features in common with our work but the programmer is expected to construct parallel programs using *only* the skeletons provided whereas our compiler is expected to deduce the skeletons from within arbitrary SML code.

The SKiPPER language (Coudarcher *et al*., 2001) provides skeletons which are built using Objective Caml and MPI. 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 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). Whether this is due to the performance of Objective Caml or to our static analysis methods is not clear from this comparison.

8 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 extended. We finally assess the main achievements and limitations of PMLS, and consider future directions for its development.

8.1 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*

https://doi.org/10.1017/S0956796804005489 Published online by Cambridge University Press

(Skillicorn & Talia, 1998) 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 major contributions of PMLS over other compilers for parallel functional programming are the automatic extraction of the abstract network which is integrated with the runtime system and the porting onto a range of architectures over which consistent behaviour is attained. Our methods are more static than others such as GpH and are therefore less flexible but the use of aggressively static methods allows optimizations which can give an edge in performance for some cases.

The relatively free hand which PMLS gives to programmers in placing skeleton constructs within arbitrary functional code is also an advantage over systems such as P^3L which constructively build skeleton programs from skeletal primitives. Our 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, an example being the `pfilter` skeleton, to filter elements of a list, which is built upon `pmap`. 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 general divide-and-conquer skeleton, a function composition skeleton implemented as a processor pipeline, and tuple parallelism built 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. Much research into skeletons has been directed towards automated derivations of specific classes of problems. 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). Thus, our compiler is also a suitable testbed for trying out ideas for the automated synthesis and optimization of skeletal programs.

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

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

Our skeletons use generic communications which result in high communications loads, so our system is most 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 above make use of this facility.

We currently use simple, 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.

Finally, PMLS is currently restricted to a small set of simple skeletons. Many generic and domain specific skeletons have been presented in the literature: our compiler could be generalized by the implementation of a more comprehensive set of skeletons.

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.

8.3 Current position and future direction

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. To this extent we have advanced the concept of algorithmic skeletons into a practical and usable tool.

Furthermore, seeking parallelism latent in the standard constructs of a major language avoids the pitfalls of building yet another language or retrofitting an existing language with non-standard extensions. Our realisation of this concept within a working parallelising compiler persuades us that functional programming is a good basis for parallel programming, that implicit parallel functional programming is achievable, and that the skeletons approach is plausible and principled.

While we think that PMLS shows that parallel functional programming through skeletons has an engaging if challenging future, our project highlights a number of well-known general problems associated with fully automated parallel programming.

First of all, an arbitrary program may not offer sufficient potential for achieving scalable parallel performance. Methods are needed for analysing arbitrary programs with a view to classifying code according to its potential for parallelism. We think that there may be benefit in marrying static analysis with dynamic sequential profiling information at compile time. We are investigating the development of profiling tools within PMLS which should aid such integrated analysis.

Furthermore, inter-processor communication bedevils all parallel programming. We think that static analysis augmented with dynamic profiling is also applicable to minimising data movements between processing elements when parallelising arbitrary programs.

Finally, naïve identification of parallelism can offer poor performance gains but optimising parallelism is very hard. We are currently engaged in developing a transformation methodology for arbitrary programs driven by profiling. At present this identifies our base skeletons using pattern matching: we would like to implement more sophisticated transformations for exposing and optimizing parallelism.

The parallel functional programming community makes strong claims about the benefits of functional languages for parallel programming. We think that our compiler represents a valuable step in justifying those claims.

Acknowledgements

This work was 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 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

Bacci, B., Danelutto, M., Orlando, S., Pelagatti, S. and Vanneschi, M. (1995) P(3)L – A Structured High-level Parallel Language and its Structured Support. *Concurrency: Practice & Exper.* **7**(3), 225–255.

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

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

Beazley, D. M., Lomdahl, P. S., Gronbech-Jensen, N., Giles, R. and Tamayo, P. (1995) Parallel Algorithms for Short-range Molecular Dynamics. *Ann. Rev. Computational Phys.* **3**, 119–175.

Bell, J. M., Bellegarde, F. and Hook, J. (1997) Type-driven defunctionalization. *Proceedings ACM SIGPLAN ICFP '97*, pp. 25–37. ACM.

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

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

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

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

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

Choi, J., Demmel, J., Dhillon, I., Dongarra, J., Ostrouchov, S., Petitet, A., Stanley, K., Walker, D. and Whaley, R. (1996) *Scala-pack: A portable linear algebra library for distributed memory computers – design issues and performance: LNCS 1041*, pp. 95–106. Springer.

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

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

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

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

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

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

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

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

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

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

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

Johnsson, T. (1985) Lambda lifting: Transforming programs to recursive equations. In: Jouannaud, J.-P. (ed), *Functional Programming Languages and Computer Architecture: LNCS 201*, pp. 190–302. Springer.

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

Kesseler, M. (1995) Constructing Skeletons in Clean The Bare Bones. In: Wim Böhm, A. P. and Feo, J. T. (eds.), *High Performance Functional Computing*, pp. 182–192.

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

Loidl, H.-W., Trinder, P. W., Hammond, K., Junaidu, S. B., Morgan, R. G. and Peyton Jones, S. L. (1999) Engineering Parallel Symbolic Programs in GPH. *Concurrency – Practice & Exper*. **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. and Trinder, P. W. (2003) Comparing Parallel Functional Languages: Programming and Performance. *Higher-order and Symbolic Computation*, **16**(3). *To Appear*.

McColl, W. F. (1993) General Purpose Parallel Computing. In: Gibbons, A. M. and Spirakis, P. (eds.), *Lectures on parallel computation. proc. 1991 ALCOM spring school on parallel computation*, pp. 337–391. Cambridge University Press.

Message Passing Interface Forum (1994) MPI: A Message-Passing Interface Standard. *Int. J. Supercomputer Applic. & High Perf. Comput*. **8**(3/4).

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

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

Michaelson, G., Scaife, N., Bristow, P. and 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.

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

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

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

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

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

Scaife, N., Michaelson, G. and Horiguchi, S. (2002a) *13th International Workshop, Implementation of Functional Languages: LNCS 2312*, pp. 138–154. Springer.

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

Serot, J. (1999) Explicit Parallelism. In: Hammond, K. and Michaelson, G. (eds.), *Research Directions in Parallel Functional Programming*, pp. 379–396. 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.

Skillicorn, D. B. and Talia, D. (1998) Models and Languages for Parallel Computation. *Computing Surveys*, June.

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

Trinder, P. W., Hammond, K., Loidl, H.-W. and Peyton-Jones, S. L. (1998) Algorithm + Strategy = Parallelism. *J. Funct. Program.* **8**(1), 23–60.