Proceedings of PASCO'94

Parallelising a Functional Program Using a List-Homomorphism Skeleton

Tore A Bratvold
Department of Computing and Electrical Engineering
Heriot-Watt University
Riccarton
Edinburgh EH14 4AS
United Kingdom
E-mail: tore@cee.hw.ac.uk

Abstract: The problems involved in developing efficient parallel implementations have proved to be hard both for programmers and parallelising compilers. Using parameterised skeletons for common patterns of parallel computation is a promising approach because parallelism can be described and structured with a high level of abstraction. This frees the programmer from dealing with low-level details of parallelism, while at the same time enforcing a structure on parallel computation that eases considerably a compiler's task. This paper illustrates how the concept of skeletons can be used for implicit parallel programming using a functional language. The implementation of a skeleton-based compiler is discussed in the context of *SkelML*, a parallelising compiler for Standard ML. It is shown how a skeleton-based program can be developed from a problem specification, and how this program is subsequently parallelised and compiled by SkelML using a list-homomorphism skeleton. Performance figures are presented and discussed.

Keywords: Functional Programming, Parallelisation, Skeletons, List-Homomorphisms, Bird-Meertens Formalism

1 Introduction

Developing efficient parallel implementations involves finding good solutions to a number of hard problems. The tasks of partitioning, load balancing and mapping are all non-trivial, and place severe strains on human programmers and parallelising compilers alike. Dealing satisfactorily with these issues while at the same time encouraging portability and code reuse is a big challenge for tools and methodologies for parallel programming. A fully implicit approach would seem attractive from the programmer's point of view because programs could be written without any regard for parallelism. However, the efficient parallelisation of such programs is hard unless the compiler receives some form of explicit guidance.

The concept of *skeletons* introduced by Cole [9] is a promising idea for bridging this gap. A skeleton is a high level template description of some parallel operation in which part of the functionality is parameterised. Common examples are processor farms, process pipelines and divide-and-conquer trees. The parallel infrastructure of a skeleton is mostly fixed and can be carefully hand-crafted once and for all by the compiler designer. It can then be reused and instantiated with code for the particular problem specific functionality as described by the programmer. By using a mixture of skeletons, possibly nested, parallelism can be described in a structured and modular way. Most of the low-level details of parallelism are hidden inside the implementation of the skeleton, which significantly eases the compiler's task. Further, by using accurate performance models of each skeleton, a compiler can make high quality decisions about resource allocation. Skeletons seem attractive as a basis for parallel programming because they provide a level of abstraction which succeeds in separating precisely semantics from behaviour. The programmer can express algorithmic properties using high-level templates, while the compiler can concentrate on administrating parallelism already in a highly structured form. As long as the semantic properties are preserved, the compiler designer is free to choose whatever implementation gives the best performance for any skeleton on any platform. Moving from one machine to another is therefore a matter of using a compiler with support for different skeleton implementations rather than changing the program.

Many proposals have been put forward as how best to incorporate skeletons into a programming environment. The P^3L language presents the programmer with a fixed set of skeleton constructors for structuring sequential imperative code [10, 17]. The compiling tools then work out how to produce an optimised implementation for a specific target architecture. Other explicit approaches include PUL [5], which provides skeleton utilities in libraries, and Parsec [13] which provides an environment for developing and tuning skeleton-based applications. In Caliban/FAST [1, 15] process network topologies can be defined as higher order functions (HOFs) and reused as skeletons in a functional framework. At the other end of the spectrum is PUFF [7], which automatically extracts processor farm parallelism from linear recursion in functional programs.

Expressing skeletons as HOFs in a functional language is an attractive approach because HOFs are an integrated and natural part of functional languages, and not a feature added ad-hoc to incorporate skeletons. In fact, several HOFs found useful for describing skeletons are already widely used by functional programmers, not because of their content of implicit parallelism, but simply because they provide a concise and powerful way of expressing functionality. Examples are `map`, `filter`, `fold`[1] and function composition. Another argument in favour of functional languages is their good characteristics for software development in general, making them well suited for both symbolic and numeric computation.

The approach described by Darlington et al [12] is based on functional programming and a selection of skeletons expressed as HOFs. Strong emphasis is placed on the use of program transformation techniques for transforming one skeleton or a group of skeletons into another in order to obtain a better mapping onto a particular target architecture. Functional higher order operators for manipulating data structures are found in many languages. Skeleton-style data parallelism can be exploited in such operators, see for example NESL [3], FP [11] and DPHaskell [14]. Similar ideas have led to the Bird-Meertens Formalism (BMF) [2] being warmly advocated as a basis for parallel programming [18]. BMF provides a set of higher order operators for a data type, as well as equational rules that can be used for program transformation.

Although there exists a rapidly growing body of skeleton related work, there is still very little experience with the approach applied in practice. Besides, most work deal only with some aspects of skeletons, and do not fully exploit the benefits offered for automatic parallelisation. In this paper we demonstrate how the concept of skeletons can be used for parallelisation of functional programs exemplified by *SkelML*, a skeleton-based parallelising compiler for Standard ML. We illustrate our approach with a case study: the development and compilation of a functional program for a best match problem.

The rest of the paper is structured as follows. The following section describes the concepts and implementation of SkelML. The case study is presented in section 3, where we look at how to develop a skeleton-based program from a problem specification with reference to list-homomorphisms. Section 4 shows step-by-step how this program is compiled by a prototype implementation of SkelML, while section 5 contains performance figures of the resulting parallel implementation. In section 6 we draw conclusions and outline avenues for future work. The appendix contains SML definitions of the higher order functions referred to throughout the paper.

2 The SkelML Parallelising Compiler

The SkelML compiler has been developed with the aim of exploring the use of skeletons for supporting implicit parallel programming [4]. SkelML parallelises functional programs written in a pure subset of Standard ML [16]. SML was chosen as a well known and "established" functional language with a formally defined semantics. Strict languages tend to have an advantage performance-wise for sequential execution, but we see no fundamental reason why a lazy language could not have been used.

[1] SML definitions of these and other HOFs are found in the appendix.

SkelML identifies potential skeleton parallelism by the use of certain HOFs in SML programs. Currently supported are `map`, `filter`, `fold` and various combinations of these. `map` and `filter` are implemented as linear processor farms, while `fold` is implemented as a chain of processors spanning a virtual process tree. For parallel implementation `fold` generally requires its argument function to be associative, while the implementation currently supported by SkelML is optimised for functions which are also commutative. Further, pipeline parallelism can be exploited in compositions of skeleton HOFs. Apart from pipelines, which may have skeleton components, SkelML does not support nested skeleton structures. If such compositions occur in the program, parallelism can only be expoilited at the outermost level.

SkelML produces parallel occam2 code ready to be compiled and run on the target platform; a transputer based DM MIMD Meiko Computing Surface[2]. A schematic overview of the compilation process is depicted in figure 1.

The programmer interacts with SkelML through a simple interface where programs can be typed in and run interactively using a built-in (sequential) interpreter for SML. The parallelising compiler is invoked and controlled through a number of switches from this environment. The interpreter is implemented directly from the inference rules of the structured operational semantics in the SML definition [16], and is used to collect profiling information about the source program as a first phase of compilation. The estimated execution time of the equivalent sequential occam2 code is calculated from a set of formulae and constants associated with each inference rule. Communication times are calculated from the sizes of data structures as recorded during profiling.

After the collection of profiling information SkelML converts the source program into an intermediate process network representation of nodes and arcs. *Skeleton nodes* encapsulate potential skeleton parallelism while *sequential nodes* represent non-parallel expressions. There is also a unique *terminal node* serving as the start and end points of the network. Nodes are connected by directed arcs, each of which has a type corresponding to the data it can carry.

A central feature of the compiler is its ability to make decisions regarding the extent to which potential parallelism should be exploited, if at all, and to handle the corresponding allocation

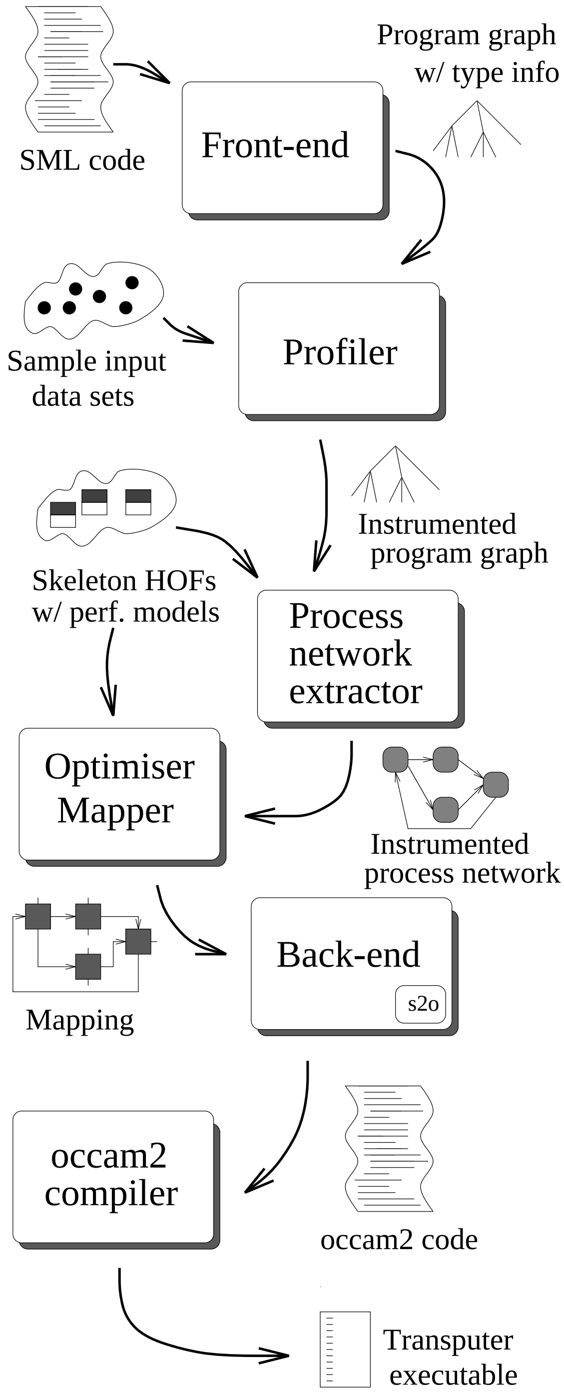

Figure 1: Overview of compilation stages

[2]Meiko and Computing Surface are registered trademarks of Meiko Ltd.

of physical resources, i.e. processors and communication links. Thus, the programmer's use of a particular HOF in a program is by no means to be regarded as a directive to the compiler to produce parallel code, it is merely an indication of the presence of potential parallelism. The performance characteristics of each skeleton are captured by parameterised performance models. Performance models are instantiated with profiling information on the execution characteristics of sequential pieces of code both with respect to time as well as sizes of data structures.

All performance estimates calculated and used by SkelML are tied to one particular target architecture. Re-targetting the compiler should however incur only minor, if any, changes in how performance information is handled. What will have to be adjusted are tables of weights and constants which reflect the target architecture's communication and computation times characteristics. This information would then be used by the existing performance models and profiling mechanisms.

Using profiling information and performance models SkelML can distinguish between useful and non-useful parallelism. For non-useful parallelism and for other pieces of sequential code, SkelML calls a sequential SML-to-occam2 translator, from now on referred to as *s2o* [6]. For skeletons found to contain useful parallelism, the corresponding performance model is used to calculate execution time characteristics, e.g. completion time, for any number of processors allocated to the skeleton. This information forms the basis for load balancing and mapping. For each supported skeleton there is an associated template implementation in occam2. Implementation templates are instantiated with sequential code, again produced by s2o.

The mapper maps the process network of nodes and arcs down to physical processors and communication links. As part of the mapping, SkelML supports a simple transformation system for optimising performance in pipelines. Transformations are represented as a set of equations, for example:

```
trans {f,g,l} map f (map g l) ==
              map (fn x => f (g x)) l;
```

where the variables `f`, `g` and `l` as listed within curly brackets can be replaced by expressions occuring in the program.

Finally, SkelML produces all occam2 code necessary to run the program. This includes code for communication, run-time system on each processor, simple I/O facilities and wiring tables.

3 A Case Study

We have chosen a simple best match problem to illustrate the process of developing and parallelising skeleton-based functional programs. Consider a fixed set of known objects, *candidates*, and a set of other objects, *problems*. The aim is to identify the candidate object which best matches the problem objects according to some matching criteria. This is a general problem description which has instances in many areas. The candidate objects may for example be some form of basic building blocks, and the problems may be composed objects. With a suitable matching function the program could be used to find the building block most commonly occuring as a component of the complex objects.

Our example is that of identifying 2D polygons from ditto model descriptions. The problem objects could for example be a stream of processed images from a camera, while the set of candidate objects is formed by the model descriptions. A candidate and a problem object match if one can be obtained from the other after suitable scaling and rotation. We want to identify the candidate object which matches most of the problem objects, and we want to know how many problem objects it matches. The top level function of the program, named `bestmatch`, should have the type

$$\begin{gathered} \textit{<list of candidate objects>} \rightarrow \\ \textit{<list of problem objects>} \rightarrow \\ (\textit{<candidate object>}\ * \\ \textit{<number of successful matches>}). \end{gathered}$$

In [8] Cole derives a highly parallel program from a problem specification in the context of BMF and list-homomorphisms. A similar approach is applicable here.

Assume $+\!\!+$ is the infix list concatenation operator, and $\cdot$ is the infix function composition operator. A function h defined on finite lists is a list-homomorphism if there exists an associative operator $\oplus$ with identity element e such that

$$h\ [\,] = e$$

and

$$h\ (x +\!\!+ y) = h\ x \oplus h\ y \tag{1}$$

for lists x and y. The relevance of this to parallel programming is seen from the Homomorphism Lemma [2] which states that a function h is a homomorphism with respect to $+\!\!+$ if and only if

$$h = (\oplus\ /)\ \cdot\ (f\ *) \tag{2}$$

for some associative operator $\oplus$ and function f, where $(\oplus\ /)$ means "fold with $\oplus$" and $(f\,*)$ means "map with f". Since the higher order functions `map` and `fold` both contain large amounts of implicit parallelism[3], any list-homomorphism has a potentially highly parallel implementation. It is clear however that our topmost function `bestmatch` is not a list-homomorphism because no binary function $\oplus$ can be devised such that (1) holds with h as `(bestmatch candidates)`. This follows from the fact that the result of `bestmatch` contains information only about the best matching object, which is insufficient because the best matching object of two concatenated lists may not necessarily be the best matching of either. Our task is thus to find a way of expressing the problem using a `map` and a `fold` with suitable arguments to produce a potentially highly parallel program.

It seems natural to split the solving of the problem into two phases: first to generate matching information about each candidate object and then to retrieve the candidate with the highest frequency of successful matches. Given an intermediate representation of successful matches for each candidate, a function `findmax` can be defined to retrieve the object with the highest frequency of matches. The generation of the matching information can be expressed as a composition of a `map` and a `fold` by first mapping a matching function with the candidate objects over the problem objects, and then fold the results to give the accumulated number of matches for each candidate object. We could however just as well have chosen to map a matching function with the problem objects over the list of candidate objects, which would have resulted in a different parallel implementation. A wider discussion of the problems involved in making such choices is outside the scope of this paper, but it is an important issue which deserves further attention.

For recording accumulated matching information we need an intermediate representation of successful matches for the candidate objects. A straightforward representation of frequencies is a list of integers, where each integer holds the number of successful matches for the corresponding candidate object. With candidate objects

```
[c1,c2,c3,c4,c5]
```

the frequency list

```
[12,6,0,43,8]
```

would mean 12 successful matches with `c1`, 6 with `c2`, none with `c3` etc.

Assume a boolean function `match` which takes two objects as arguments and returns `true` if they match. This can easily be used with a calling function `matchtop` to return a frequency list for matches between the candidates and one problem object. `matchtop` partially applied to the candidates list will be the argument function to `map`. The argument function to `fold` is an associative and comutative function `combfreq` which combines two frequency lists into one.

[3] This is true for `fold` only if its argument function is associative. Our current implementation also requires commutativity.

With respect to (2) we have now succeeded in finding suitable functions for $\oplus$ and f, `combfreq` and `matchtop` respectively, and the whole program can be specified as follows:

```
fun match objC objP = ... ;
   (* true if objP matches objC,
      rotated and/or scaled *)

fun matchtop [] objP = [] |
    matchtop (objC::T) objP =
       if (match objC objP) then [1]
       else 0::(matchtop T objP);
   (* Returns frequency list [0,,,1]
      (1 in pos i : obj#i matches) *)

fun combfreq [] L = L |
    combfreq L [] = L |
    combfreq (h1::t1) ((h2:int)::t2) =
       (h1+h2)::(combfreq t1 t2);
   (* zips two frequency lists with + *)

fun findmax candidates freqlist = ... ;
   (* picks best candidate from
      a list of frequencies *)

fun bestmatch candidates problems =
   findmax candidates (fold combfreq []
      (map (matchtop candidates)
           problems));
   (* toplevel function *)
```

4 The Compilation Process

Feeding the source program to SkelML it is parsed and typechecked. Next we execute the program applied to sample test data with the SkelML interpreter in profiling mode. We used as input a list of 9 candidate objects and a list of 10 problem objects. This results in the program being instrumented with sizes of datastructures and costs of estimated sequential occam2 execution times. The compiler then analyses the program, identifies the topmost function `bestmatch` and generates a process network as depicted in figure 2. The HOFs `map` and `fold` are represented as skeleton nodes. The compiler also registers that these two HOFs can execute in parallel in a pipeline. A scaling facility allows profiling data to be scaled up to give a more representative picture of realistic input sets. We feed SkelML with a scaling factor of 50 for both skeletons, i.e. the

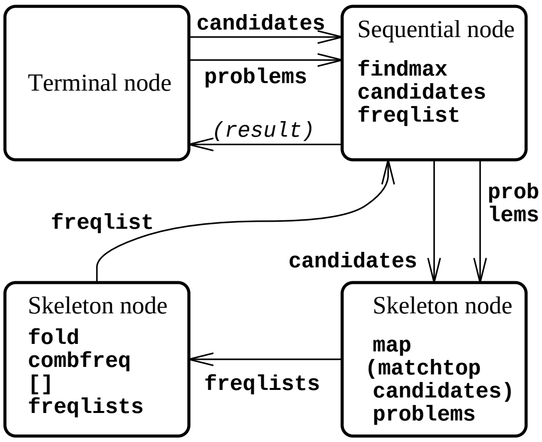

Figure 2: Initial process network

performance models assume that the input list of problem objects is 50 times longer. SkelML calculates the following performance estimates for the two skeleton nodes:

```
*** Performance model for map: ****
Type in : (real * real) list
Type out: int list
Input length: 500
Tf: 6678.5 Tk: 522.32 Tp: 361
Seq time: 3.33925e+06 microseconds
  No W    ParTime   Speedup  Efficiency
     1  3.56008e+06 0.937971 0.937971
     2  1.84942e+06 1.80557  0.902785
     3  1.26803e+06 2.63341  0.877804
     4   973919     3.42867  0.857168
   (...)
    15   334061     9.99594  0.666396
    16   325550    10.2572   0.641078
    17   321330    10.392    0.611292
 *> 18   321217    10.3956   0.577534
    19   325328    10.2642   0.540223
```

```
*** Performance model for fold: ****
Type in : int list
Type out: int list
Input length: 500
Tf: 268.366 Tek: 475.4 Tep: 315
Seq time: 134183 microseconds
  No W   ParTime  Speedup  Efficiency
 *>  1   371883   0.36082  0.36082
```

Note that `fold` on its own is found not to contain useful parallelism. As a pipeline component however a 1-processor skeletal implementation may still be preferable to a sequential implementation, the performance of the pipeline as a whole considered. At this point however, no attention is being paid to the usefulness of the parallelism exposed. The compiler notes that a transformation is applicable transforming the composition of the two HOFs to another HOF: `foldmap`. The relevant transformation rule is:

```
trans {f,g,a,l} fold f a (map g l) ==
                foldmap f a g l;
```

`foldmap` is a version of `fold` in which an additional argument function is mapped over all elements of the input list prior to folding. The corresponding parallel implementation is generally preferable to that of the combined `map` and `fold` because there is less inter processor communication involved. The `foldmap` skeleton can be seen as an optimised implementation of list-homomorphisms. A performance model of `foldmap` is instantiated with information extracted from the other two performance models:

```
*** Performance model for foldmap: ****
Type in : (real * real) list
Type out: int list
Input length: 500
Tf: 6946.87 Tek: 307.9 Tep: 205.5
Seq time: 3.47343e+06 microseconds
  No W  ParTime      Speedup  Efficiency
     1  3.62738e+06 0.957559 0.957559
     2  1.87896e+06 1.84859  0.924297
     3  1.29855e+06 2.67485  0.891615
     4  1.00836e+06 3.44462  0.861156
   (...)
    20   326343    10.6435   0.532176
    21   311833    11.1388   0.530417
    22   304578    11.4041   0.518367
   (...)
    31   261050    13.3056   0.429214
 *> 32   253795    13.686    0.427687
    33   261025    13.3069   0.403239
```

SkelML sees that the best completion time for `foldmap` is better than that of the `map`-`fold` pipeline, and decides to apply the transformation. The resulting process network is depicted in figure 3. SkelML now prompts for the number of

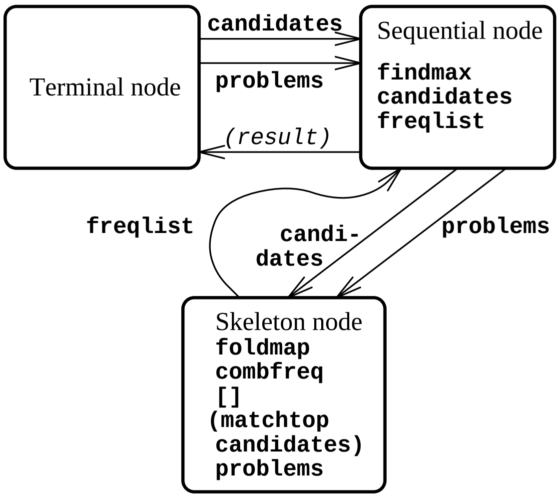

Figure 3: Transformed process network

processors available. In the following section we present results showing the execution times of the program when compiled to execute on a range of processors. The mapping is carried out in accordance with the number of processors given, and the resulting processor topology is depicted in figure 4. Finally SkelML produces occam2 code for

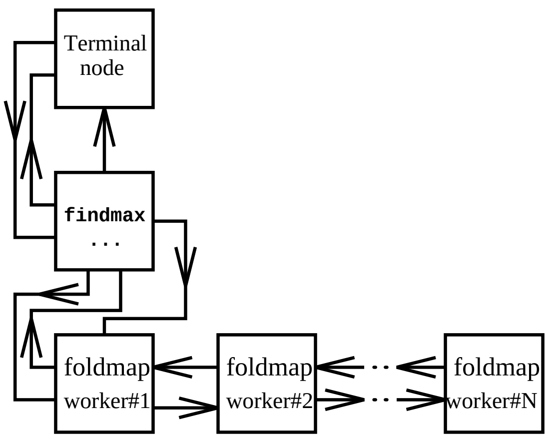

Figure 4: Processor mapping

all processors, calling s2o for sequential translation where applicable. Wiring and process placement tables are also generated. The resulting occam2 program is now ready to be compiled by an occam2 compiler for the target architecture.

5 Results

The program was configured using from 2 to 22 worker processors in the `foldmap` skeleton, and executed with three different data sets. All data sets used the same candidate objects, but had a different number of problem objects; 200, 500 and 1000. 500 objects should correspond to the scaled profiling data, while the other input sizes were tried to illustrate how the program behaves when input lengths deviate from profiling data. The results are shown in tables 1 - 3, where the number of processors in the leftmost columns refer to the varying number of processors in the `foldmap` skeleton, and speedup and efficiency figures are calculated in relation to this. All times given are in milliseconds.

s2o was used to produce a purely sequential version of the program used as a reference for calculating speedup and efficiency figures. We note that SkelML has converted the implicit parallelism present in the `foldmap` function into useful parallelism in the application. Further, the

Input length: 500 objects			
No procs	Time	Speedup	Efficiency
(seq)	3601	–	–
2	2026	1.78	0.89
4	1121	3.21	0.80
6	826	4.36	0.73
8	670	5.37	0.67
10	588	6.13	0.61
12	528	6.82	0.57
14	489	7.36	0.52
16	459	7.85	0.49
18	434	8.29	0.46
20	411	8.77	0.43
22	403	8.92	0.41

Table 1: Performance vs processors in skeleton

Input length: 200 objects			
No procs	Time	Speedup	Efficiency
(seq)	1438	–	–
2	816	1.76	0.88
4	460	3.12	0.78
6	341	4.21	0.70
8	288	4.98	0.62
10	250	5.74	0.57
12	234	6.14	0.51
14	218	6.59	0.47
16	209	6.87	0.43
18	200	7.18	0.40
20	190	7.54	0.38
22	198	7.25	0.33

Table 2: Performance vs processors in skeleton

Input length: 1000 objects			
No procs	Time	Speedup	Efficiency
(seq)	7211	–	–
2	4036	1.78	0.89
4	2220	3.24	0.81
6	1616	4.46	0.74
8	1320	5.46	0.68
10	1136	6.35	0.63
12	1023	7.04	0.58
14	934	7.72	0.55
16	874	8.25	0.52
18	828	8.71	0.48
20	789	9.13	0.46
22	760	9.48	0.43

Table 3: Performance vs processors in skeleton

results for the data set of 1000 problem objects gives a slightly better relative performance than that of 500 objects. As the number of objects increases, the workload may be split more evenly among the skeleton processors, and also the relative cost associated with the startup and shutdown of the skeleton decreases in significance. A similar explanation can be made for the relatively lower performance using 200 objects. This considered, the program seems to cope satisfactorily with the change in input data lengths.

Figure 5 shows the speedup figures in relation to that predicted by `foldmap`'s performance model. The graphs are not directly compara-

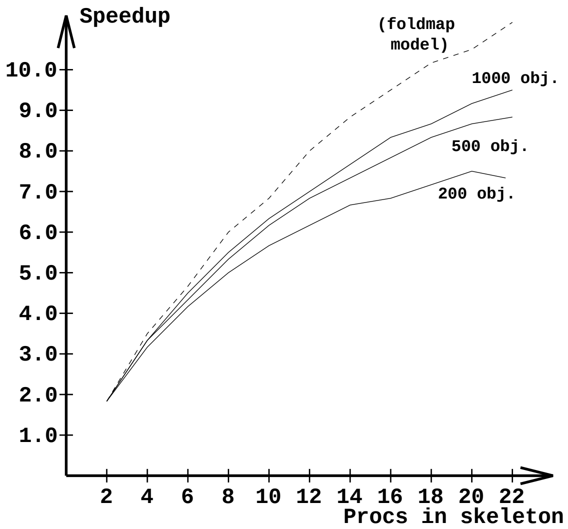

Figure 5: Speedup vs no processors

ble because `foldmap`'s model deals only with the performance of the skeleton, while the execution times are for the whole application including the time for executing the `findmax` function and the delay in transfer of data between the terminal node processor, the processor evaluating `findmax`, and the skeleton. One would thus expect the measured results to have a slightly lower speedup than that of the skeleton model, increasingly so with higher number of processors.

It is still not clear exactly how good the model's predictions are. One potential source of error is that the sequential execution times supplied to the model are inaccurate. The profiling run of the sequential code with 10 problem objects gave an estimated run time of the sequential occam2 code of 70370 microseconds. The actual time measured for the program on identical input data was 72320 microseconds, which we consider to be within satisfactory bounds. This suggests that any error may lie within the constants and formulae of the model itself. We aim to subject `foldmap`'s performance model to further investigation and tuning.

6 Conclusions

We have presented an overview of SkelML, a skeleton-based parallelising compiler for Standard ML, and shown how a program can be developed and compiled exploiting skeleton parallelism. Using the Homomorphism Lemma from the Bird-Meertens Formalism turned out to be a helpful guide when developing the program. On the basis of performance models and profiling data SkelML decided to apply an optimising transformation using a list-homomorphism skeleton, i.e. `foldmap`, for parallel implementation. The results were encouraging: the parallel program ran 8-10 times faster than the sequential, and changes in the program's input length gave no more than expected variations in speedup. While the accuracy of the predicted sequential occam2 execution time was satisfactory, the accuracy of `foldmap`'s performance model is subject to further investigation.

We have demonstrated that by using skeletons and profiling information as a basis for implicit parallel programming, encouraging results can be achieved. The development of SkelML will continue and it is intended to apply the compiler to more example programs in the future.

7 Acknowledgments

Acknowledgments are due to David Busvine who developed the s2o translator, and to Greg Michaelson for general assistance throughout the development of SkelML as well as for reading and commenting on a draft of this paper. Valuable comments have also been received from Norman Scaife on various linguistic points and from Murray Cole regarding the presentation of the case study. Finally, the author wishes to express his gratitude to Heriot-Watt University for financial support.

References

[1] FAST: Functional Programming for Arrays of Transputers - The Collected Papers. Report CSTR 93-15, Declarative Systems Laboratory, Electronics & Computer Science,

University of Southampton, UK, 1993. Also available as Report Number DOC 93/47 from Department of Computing, Imperial College, London, UK.

[2] Richard S Bird. An introduction to the theory of lists. In M Broy, editor, *Logic of Programming and Calculi of Discrete Design*, volume F36 of *NATO ASI Series*, pages 5–42. Springer-Verlag, 1987.

[3] Guy E Blelloch. NESL: A Nested Data-Parallel Language (Version 2.6). Report CMU-CS-93-129, School of Computer Science, Carnegie Mellon University, PA, USA, April 1993.

[4] Tore A Bratvold. A Skeleton-Based Parallelising Compiler for ML. In *Proceedings of the Fifth International Workshop on Implementation of Functional Languages, Nijmegen, the Netherlands*, pages 23–34, September 1993.

[5] R A A Bruce, S R Chapple, N B MacDonald, and A S Trew. CHIMP and PUL: Support for Portable Parallel Computing. Technical Report EPCC-TR93-07, Edinburgh Parallel Computing Centre, March 1993. Presented at the Fourth Annual Conference of the Meiko User Society, Southampton, April 1993.

[6] David Busvine. Translation of SML to Sequential Occam2. Technical Report TR 91/7, Department of Computer Science, Heriot-Watt University, Edinburgh, 1991.

[7] David Busvine. Implementing Recursive Functions as Processor Farms. *Parallel Computing*, 19:1141–1153, 1993.

[8] Murray Cole. Parallel Programming, List Homomorphisms and the Maximum Segment Sum Problem. Technical Report CSR-25-93, Department of Computer Science, The University of Edinburgh, 1993.

[9] Murray I Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Pitman/MIT, 1989.

[10] Marco Danelutto and Roberto Di Meglio et al. A methodology for the development and the support of massively parallel programs. *Future Generation Computer Systems*, 8:205–220, August 1992. North Holland.

[11] Marco Danelutto and Susanna Pelagatti. Parallel Implementation of FP using a Template-based Approach. In *Proceedings of the Fifth International Workshop on Implementation of Functional Languages, Nijmegen, the Netherlands*, pages 7–21, September 1993.

[12] J Darlington et al. Parallel Programming Using Skeleton Functions. In A Bode, M Reeve, and G Wolf, editors, *PARLE '93 Parallel Architectures and Languages Europe, Munich, Germany*, pages 146–160. Springer-Verlag, LNCS 694, 1993.

[13] David Feldcamp and Alan Wagner. Parsec - A Software Development Environment for Performance Oriented Parallel Programming. In S Atkins and A S Wagner, editors, *Transputer Research and Applications 6*, pages 247–262. IOS Press, 1993.

[14] Jonathan M D Hill. Data Parallel Haskell: Mixing Old and New Glue. Technical Report 611, Queen Mary and Westfield College, University of London, UK, December 1992.

[15] Paul H J Kelly. *Functional Programming for Loosely-coupled Multiprocessors.* Pitman/MIT, 1989.

[16] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[17] Susanna Pelagatti. *A methodology for the development and the support of massively parallel programs.* PhD thesis, Università di Pisa-Genova-Udine, Italy, March 1993. TD 11/93.

[18] David B Skillicorn. Architecture-Independent Parallel Computation. *Computer*, 23(12):38–51, December 1990.

Appendix: SML Definitions of some Higher Order Functions

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

fun filter f [] = [] |
    filter f (h::t) =
      if f h then h :: filter f t
             else filter f t;

fun mapfilter f p [] = [] |
    mapfilter f p (h::t) =
      if p h then f h::mapfilter f p t
             else mapfilter f p t;
(* equivalent to (map f) o (filter p) *)

fun filtermap p f [] = [] |
    filtermap p f (h::t) =
       let val fh = f h in
          if p fh
             then fh::filtermap p f t
             else filtermap p f t
       end;
(* equivalent to (filter p) o (map f) *)

fun fold (f:'t->'t->'t) a [] = a |
    fold f a (h::t) = f h (fold f a t);
(* parallelism iff f is associative *)

fun foldmap (f:'t->'t->'t) a g [] = a |
    foldmap f a g (h::t) =
       f (g h) (foldmap f a g t);
(* equivalent to (fold f a) o (map g) *)
```