PARALLEL STANDARD ML WITH SKELETONS*

NORMAN SCAIFE†, GREG MICHAELSON‡, AND SUSUMU HORIGUCHI§

1. Introduction. The exploitation of parallelism in programs is greatly eased by tools based on implicit parallelism. The PMLS compiler realises higher order functions (HOFs) in Standard ML (SML) programs as parallel algorithmic skeletons. An SML program is treated as a prototype of the final parallel implementation. Static analysis and dynamic instrumentation, combined with performance models for skeletons, enable the identification of useful parallelism. Prototype transformation is employed to try and optimise parallelism. Where exploitable parallelism cannot be identified, program synthesis is used to introduce new instances of HOFs. Here, we describe the compiler and the methodology it is intended to support.

2. A Skeletons Methodology. The compiler was motivated by extensive exploration of algorithmic skeletons for the construction of parallel computer vision systems from functional prototypes [17, 11, 18]. Our source language, SML, is a mature functional language with a stable formal definition [12]. For prototyping, SML provides closeness to formalisms for proof and transformation. For parallel prototyping, SML's strictness is better suited than the laziness of Haskell [9] as it results in more predictable behaviour.

We focus on common low-level HOFs such as `map` and `fold` which are ubiquitous in functional programs. Explicit HOF names are used as the basis for identification:

```
fun map f [] = []
  | map f (h::t) = f h::map f t
fun fold (f:'a*'a->'a) b [] = b
  | fold f b (h::t) = f (h,fold f b t)
fun filter p [] = []
  | filter p (h::t) = if p h then filter p t else h::filter p t
fun compose (f:'b list -> 'c list) (g:'a list -> 'b list) x = f (g x)
fun tuple2 (fa,fb) = (fa (),fb ())
```

`map` and `fold` HOFs may be synthesised from arbitrary recursive functions [6], and implemented in parallel in a variety of ways [8]. `filter` can be implemented constructively from `map` with minimal overhead. Function composition can be realized in parallel provided the argument is a decomposable datatype. Tuple parallelism can be implemented by turning tuples into suspensions:

`(a,b)` $\Leftrightarrow$ `tuple2 (fn _ => a,fn _ => b)`

Transformations may be defined over these base functions to implement simple identification, distribution and equivalences. For example, we can mitigate communications costs by aggregation:

`(map f) o (map g)` $\Leftrightarrow$ `map (f o g)`

If a `fold` function argument is not associative, we can sometimes extract partial parallelism using `map`:

*This work was supported by the Japan JSPS Postdoctoral Fellowship P00778 and UK EPSRC grants GR/J07884 and GR/L42889

†School of Information Science, Japan Advanced Institute of Science and Technology, Tatsunokuchi, Ishikawa, Japan, 923-1292 (norman@jaist.ac.jp).

‡Department of Computing and Electrical Engineering, Heriot-Watt University, Riccarton, Edinburgh, EH14 4AS, (greg@cee.hw.ac.uk).

§School of Information Science, Japan Advanced Institute of Science and Technology, Tatsunokuchi, Ishikawa, Japan 923-1292 (hori@jaist.ac.jp)

```
fold (fn (h,t) => f (g h) t) b l   ⇔   fold f b (map g l)
```

We can switch between alternative implementations of the same skeleton:

```
map f   ⇔   fold (fn (h,t) => f h::t) []
```

Finally, we can move sites of argument instantiation to allow pre-computation before distribution:

```
hof (f x)   ⇔   let val x' = f x in hof x' end
```

Given the high computational complexity of program transformation systems we need a fast method of assessing the impact of transformations upon parallel performance. Using the SML definition in conjunction with an SML interpreter we can accurately summarise the semantic behaviour of an executing SML program [15]. This behaviour can then be related to the execution times of compiled programs.

Cost models for algorithmic skeletons have been well-studied [16, 13]. Combining the profiling information with data size measurements results in the ability to instantiate these cost models giving predictions of the effect of transformations.

Partitioning transformations into; identification, optimisation, and restructuring, we propose a methodology for parallel functional prototyping which can be summarised as follows:

1. A sequential prototype is transformed using restructuring and identification transformations to lift as many HOF instances from arbitrary code as possible.
2. The computational loads for HOF instance functions and communication costs for their arguments and results are determined.
3. This data is applied to models for the equivalent algorithmic skeletons to determine the viability of potential parallel implementations.
4. If no parallelism is predicted, the prototype is transformed using optimisation and retsructuring transformations. to optimise the computational versus communication costs in the models.
5. Where useful parallelism is predicted the HOFs should be realised as, possibly nested, algorithmic skeleton instantiations.

3. The Parallel SML with Skeletons (PMLS) Compiler.

3.1. Design. Our experience with the PUFF [4] and SkelML [3] compilers, combined with that gained in developing parallel computer vision systems led to the design of a more general parallelizing compiler for SML [14, 10], with the following properties:

- The full SML Core language is supported.
- Dynamic profiling provides parallel performance prediction.
- Heuristics-guided transformations drive performance optimisation.
- Transformations may be generated using automated proof synthesis.
- Skeletons can be nested but are not first-class objects.
- The compiler targets a broad range of general-purpose parallel computers.

The spine of the PMLS compiler was constructed from 1997 to 2000, incorporating `map` and `fold` skeletons. The performance prediction has been developed to a state where usably accurate predictions are provided for these two skeletons. Further skeletons are under development but are not currently modelled. At present we can present results for the manual analysis of exemplars and are currently automating this process.

3.2. Implementation. The host compilers are the ML Kit [2] for the front end, profiling and transformation systems, and Objective Caml [5] for backend compilation and execution. This combination allows the transformational compiler to be implemented on a separate machine so that the backend can be kept small and retargetable.

3.3. Analysis. During our analysis, the SML syntax tree is mapped onto a static network of processors. We generate an *abstract network* description of the program which defines the relationship between the HOFs in the sequential prototype and between executing skeletons in the parallel version. A two-phase algorithm is implemented whereby the skeleton information is added to the existing types and then the type information is stripped out leaving the network description:

```
val ff3 = map (fn x => x + 1) [1,2,3]
val ff3 :: node(map,base list,[(int->base),base list])
```

The handling of free values complicates our analysis. We wish to avoid the transmision of functional data at runtime as large free value bindings can overload the communications. For non-functional free values, lambda-lifting [7] is sufficient. 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. For skeleton instances, the free values are registered with the runtime system and transmitted to the point of function application.

For free functionals we use defunctionalization [1]. In this technique, closures 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. The defunctionalization of SML is problematical, however. The published algorithms all require forward code-references which would necessitate runtime registration of functions with attendant jump tables for SML. We adopt a solution whereby the entire program is turned into a single mutually-recursive block.

Finally, we generate launch-code for the skeletons which replace HOFs. This involves detecting free values, replacing the HOF with the skeleton call and reconstructing the type information. Our analysis thus converts the following code:

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

into:

```
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))
```

3.4. Dynamic profiling. PMLS includes an integrated *dynamic* profiling mechanism. The ML Kit interpreter has been modified to annotate the syntax tree with counts of rules in the semantics which are fired during execution. Given a suitable set of training programs it is possible to assign a weight to each rule in the semantics. The test programs cause as many of the different rules to be fired as possible and the actual execution times for the test programs form the dependent variable for weight determination. This system can be expressed in matrix form as $Pw = x$, where P is an $r \times n$ matrix where r is the number of rules and n the number of program executions, w is a vector of weights and x is the vector of execution times. We solve this equation for the training-set, giving a set of weights which can be applied to a new, unknown profile of rule counts to give a predicted execution time. It is important to select a suitable set of test programs whereby each rule has a significant representation and no rules dominate the entire data set.

Currently, we can use either numerical analysis methods or genetic algorithm techniques [15] to solve the above equation for our training-set. The numerical methods give more accurate fits but suffer from numerical instability which limits the range of validity of the generated weights. The genetic solution is extremely slow and much less accurate but has less numerical instability. At best this technique only gives a *rule-of-thumb* estimate of the sequential performance of an arbitrary section of code, typically within about a 200% margin. This is sufficient, however, to drive a performance-improving transformation system.

3.5. Transformation. PMLS transformations are defined by associating equivalent SML constructs (either *expressions* or *declarations*). The transformations are elaborated and type information is preserved on transformation application. The resulting system allows simple identification:

```
dtrans T2008 (FF,F,H,T,TAIL) = fun FF [] = TAIL
                                 | FF (H::T) = F::FF T
                           ==> fun FF l = (map (fn H => F) l)@TAIL
end
```

and optimization:

```
trans T202 (F,G,L) = fn (F,G,L) => map F (map G L)
                 ==> fn (F,G,L) => map (fn x => F (G x)) L
```

We are currently constructing the transformation engine using the performance prediction results to rank improving transformations and prune non-useful ones.

3.6. HOF Synthesis. HOF-oriented skeleton programming is often criticised for restricting the programmer to the supported HOFs. To support the programmer with automatic detection of these HOFs, Cook investigated automatic extraction of HOFs using proof-planning techniques [6]. Core SML is translated to λ-Prolog, and the λ-CLAM proof-planner is employed to locate instances of `map`, `fold` and `scan` using higher-order unification and middle-out reasoning. The search space is constrained using type information.

For example, the following recursive functions:

```
fun squares [] = []
  | squares ((h:int)::t) = h*h::squares t
fun squs2d [] = []
  | squs2d (h::t) = squares h::squs2d t
```

yield the following equivalent programs:

```
fn x => map (fn y => map (fn (z:int) => z*z) y) x
fn x => foldr (fn (y,z) => (map (fn (u:int) => u*u) y::z)) [] x
fn x => map (fn y => foldr (fn (z:int,u) => z*z::u) [] y) x
fn x => foldr (fn (y,z) => foldr (fn (u:int,v) => u*u::v) [] y::z) [] x
fn x => map (fn y => squares y) x
fn x => foldr (fn (y,z) => squares y::z) [] x
```

Currently, we use the HOF synthesizer as a pre-processor.

4. Performance. We have employed PMLS to parallelise a wide range of SML programs. Substantial exemplars include island model genetic algorithms, arbitrary length integer matrix multiplication, linear equation solving and ray tracing. In general, skeleton parallelism tends to be coarse grain. For such exemplars, PMLS offers useful, scalable speedup, typically on up to 16 processors.

PMLS can generate native code for a variety of CPUs. Consistent cross-platform performance as been shown on the Cray T3E, Fujitsu AP3000, Sun Enterprise, IBM SP2 and Beowulf-class workstation clusters.

5. Concluding Remarks. PMLS demonstrates the potential of exploiting implict parallelism through the combination of a variety of static and dynamic program analysis techniques targeted at skeleton-oriented parallel implementation. Our approach is novel in:

- matching parallel topology to algorithm rather than algorithm to topology;
- basing profiling on semantic entities rather than absolute or simulated times;
- closely coupling instrumentation, analysis and transformation.

PMLS is primarily a research vehicle. Current challenges include:

- combining speed with stability in performance prediction;
- broadening the range of exploitable HOFs;
- exploiting parallelism in the presence of conditionals.

REFERENCES

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

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

[3] T. Bratvold. *Skeleton-based Parallelisation of Functional Programmes.* PhD thesis, Dept. of Computing and Electrical Engineering, Heriot-Watt University, 1994.

[4] D. Busvine. *Detecting Parallel Structures in Functional Programs.* PhD thesis, Heriot-Watt University, Riccarton, Edinburgh, 1993.

[5] E. Chailloux, P. Manoury, and B. Pagano. *Développement d'applications avec Objective Caml.* O'Reilly, Paris, April 2000.

[6] A. Cook, A. Ireland, and G. Michaelson. Higher order function synthesis through proof planning. In *Proceedings of 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, pages 307–310, San Diego, USA, Nov 2001. IEEE Computer Society.

[7] T. Johnsson. *"Lambda Lifting: transforming programs to recursive equations"*, volume 201 of *LNCS*, pages 190–302. Springer-Verlag, 1985.

[8] H-W. Loidl, F. Rubio, N. Scaife, K. Hammond, S. Horiguchi, U. Klusik, R. Loogen, G. J. Michaelson, R. Pe na, Á. J. Rebón Portillo, S. Priebe, and P. W. Trinder. Comparing Parallel Functional Languages: Programming and Performance. *Higher-order and Symbolic Computation*, 2002. *Submitted December 2001.*

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

[10] G. Michaelson, N. Scaife, P. Bristow, and P. King. 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, 2001.

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

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

[13] R. Rangaswami. *A Cost Analysis for a Higher-Order Parallel Programming Model.* PhD thesis, University of Edinburgh, 1995.

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

[15] N. Scaife, G. Michaelson, and S. Horiguchi. Empirical performance prediction from semantics-based profiling. In *Sixth International Symposium on Functional and Logic Programming (FLOPS 2002)*, Aizu, Japan, Sep 2002. *Submitted February 2002.*

[16] D. B. Skillicorn and W. Cai. A Cost Calculus for Parallel Functional Programming. *Journal of Parallel and Distributed Programming*, 28(1):65–84, 1995.

[17] A. M. Wallace, G. J. Michaelson, P. McAndrew, K. Waugh, and W. Austin. Dynamic Control and Prototyping of Parallel Algorithms for Intermediate and High-level Vision. *IEEE Computer*, 25(2):43–53, Feb 1992.

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