Engineering a Parallel Compiler for Standard ML

Norman Scaife, Paul Bristow, Greg Michaelson and Peter King

Department of Computing and Electrical Engineering,
Heriot-Watt University, Riccarton, Edinburgh, EH14 4AS
norman/paul/greg/pjbk@cee.hw.ac.uk

Abstract. We present the design and partial implementation of an automated parallelising compiler for Standard ML using algorithmic skeletons. Source programs are parsed and elaborated using the ML Kit compiler and a small set of higher order functions are automatically detected and replaced with parallel equivalents. Without the presence of performance predictions, the compiler simply instantiates all instances of the known HOFs with parallel equivalents. The resulting SML program is then output as Objective Caml and compiled with the parallel skeleton harnesses giving an executable which runs on networks of workstations. The parallel harnesses are implemented in C with MPI providing the communications subsystem. A substantial parallel exemplar is presented, an implementation of the Canny edge tracking algorithm from computer vision which is parallelised by implementing the `map` skeleton over a set of subimages.

1 Introduction

We are investigating the development of a fully automatic parallelising compiler for the pure functional subset of Standard ML, identifying potential parallelism in sites of higher order function (HOF) use, for realisation through parallel algorithmic skeletons [1], ultimately generating C with MPI. The design of the compiler is discussed in detail in [2]. Here we consider the development of the compiler spine, which enables its use in generating instantiated parallel skeletons from explicitly nominated higher order functions.

A schematic outline of the design is shown in Fig. 1. The *front end* lexically analyses, parses and type checks the SML prototype. The *profiler*, based on the structural operational semantics for SML, runs the prototype on test *data sets* to determine abstract communication and processing costs in HOF use. The *analyser* uses *performance models* instantiated for the target architecture to predict concrete costs. Where communication outweighs processing, the *transformer* manipulates composed and nested HOFs to try and optimise parallelism. Finally, the *back end* selects appropriate *skeletons* for HOFs with predicted useful parallelism and instantiates them with sequential code.

Our design is novel in a number of respects:

- Many researchers have proposed the use of static cost models to predict parallel performance, in particular Skillicorn [3]. These are most effective where parallelism is limited to a small number of regular constructs, for example as in NESL [4] and FiSh [5]. However, in general cost modelling is undecidable: program analysis results in a set of recurrence relations which must then be solved [6]. In contrast, we use dynamic prototyping to try and establish typical behaviour. These implementation independent

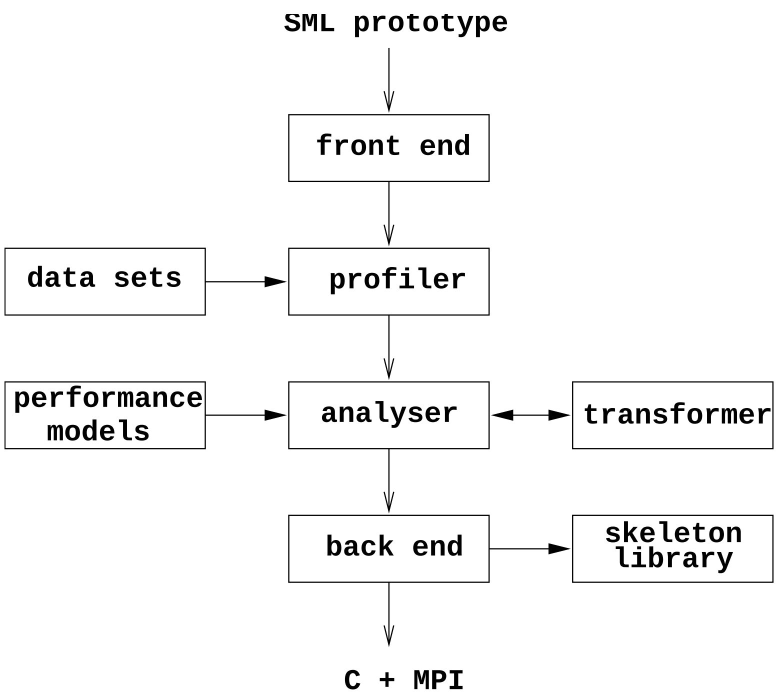

Fig. 1. Overall compilation scheme

measures are then used with target architecture metrics to predict actual costs. Using this approach on an SML subset, Bratvold [7] achieved around 85% accuracy.

– Our compiler is intended to enable code generation from HOFs nested to arbitrary depth. Rangaswami's HOPP general scheme [8] actually provided for nesting up to three deep of a fixed set of HOFs. While it may be argued that deeper nesting is rare, we think that arbitrary nesting of arbitrary HOFs presents interesting challenges, in particular for skeleton coordination to optimise data distribution.
– Our system requires the close integration of program proof and transformation components with the profile analyser to try to restructure HOF use to optimise parallelism. Darlington *et al.* [9] seem to be the only other group investigating this combination of technologies. We intend to draw on and elaborate proof planning techniques developed by the University of Edinburgh Mathematical Reasoning Group [10] both to prove hypothesised transformations on HOFs and to lift HOFs from prototypes.

We should clarify and qualify our bold assertion that we are developing a *fully automatic* parallelising compiler. Our aim is to construct a system within which users have no necessary knowledge of sources of parallelism in their programs. Instead the compiler tries to locate and optimise parallelism for them.

However, the location of parallelism in HOFs for exploitation through algorithmic skeletons appears both to restrict sources of parallelism in functional programs and to oblige the user to have some understanding that HOFs must be used for parallelism to be exploited. This is indeed the case for the spine of our system where only explicitly nominated HOFs are realised through skeleton instantiation. However, in our full system, the use of a HOF will be no guarantee of parallelism: that will depend on program analysis determining that such parallelism is useful in the sense of computation outweighing communication. Furthermore, the automatic lifting of HOFs from functional programs may identify latent opportunities for parallelism

In their survey of models and languages for parallel computation [11], Skillicorn and Talia identify six stages between *Nothing Explicit, Parallelism Implicit (NEPI)* models which abstract away from all aspects of parallelism and *Everything Explicit (EE)* models where all aspects must be treated explicitly. They note, as have others, that exploiting the implicit parallelism throughout functional programs through *dynamic* mechanisms, for example graph reduction, is very hard: it is more effective to limit sites of parallelism to a set of generic components with determinate behaviours. Thus, Skillicorn and Talia place algorithmic skeletons within this *static* subclass of NEPI. However, as will become clearer below, our system also encompasses the most concrete EE model as MPI is available to skeleton developers working at a variety of levels.

2 The Parallel Compiler

In building our compiler, we wish to avoid writing yet more SML analysis and code generation components. Instead, we have sought to plug together existing modules in building the compiler spine, which provides for the parallelisation of SML prototypes with explicitly nominated HOFs. The spine is shown in Fig. 2.

An SML program is parsed and elaborated into a bare language abstract syntax tree which is evaluated by the profiler. Instances of the HOFs are identified and replaced with calls to parallel skeleton constructs. Finally, the AST is output as Objective Caml and linked with the skeleton implementations to give a parallel SML executable.

2.1 Parsing Standard ML

We use the ML Kit [12] for the compiler front end, to parse and elaborate the source SML program. The main reason for choosing this system as our host compiler, as opposed to other more efficient compilers such as SML of New Jersey [13], is the fact that the ML Kit is closely based on the Standard ML definition [14] and provides an evaluator which mirrors the dynamic semantics. This is important since our profiling mechanism is based around relating actions in the dynamic semantics to operations in the implementation.

2.2 Exploitation of Parallelism

The principal components of our parallelisation system are the profiler, the performance predictor and the transformation system. The profiler, which is under development, is discussed further below. Data transmission costs for the parallel performance prediction are also being developed.

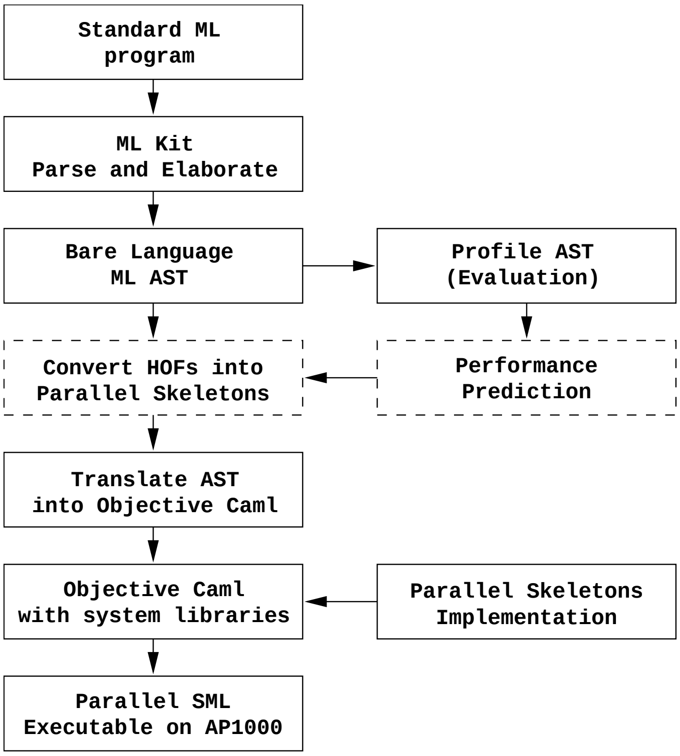

Fig. 2. Compiler spine

In the absence of performance analysis the transformation system applies the "identity" transformation. Thus, at present, all instances of HOFs are converted into parallel equivalents. Work is currently under way to develop a minimal set of useful transformations using automated proof assistants.

2.3 The Profiler

The profiler is based around the ML Kit evaluator. It will evaluate the program and count up rule firing and other costs. A variety of different methods of summarising the profiling information for the predictor will be considered. This technique has been investigated by Bratvold [15] and, provided a one-to-one mapping between semantic actions in the profiler and the evaluation scheme in the final implementation can be achieved, useful accuracy is possible. It is for this reason that we work with an elaborated bare language abstract syntax tree, where a simpler language than the full grammar is available and type information is also present allowing communication costs to be estimated.

Bratvold's work targeted networks of transputers and was based around an SML to Occam translator [16]. The profiling weights were mostly derived from this work and the resulting system was closely coupled with the translator. It is our intention to achieve a higher degree of orthogonality with our system which may mean more detailed information being gathered during the profiling process. We would also like to provide a means of automatically generating the profiling weights for the predictor allowing a higher degree of portability in our system. Since our backend compiler is likely to be performing optimisations on the generated code our sequential predictions are likely to be lower bounds on the sequential performance. The influence of this on the parallel performance prediction will have to be investigated.

2.4 Backend Compiler

Once program analysis has completed we are left with an annotated abstract syntax tree(AST) in which HOFs have been replaced by calls to parallel constructs written in C and MPI. In addition, the analyser will have generated processor/task allocations for the skeletons which have been implemented in parallel.

The AST has to be output in a form suitable for targeting the parallel machine. In principle, any system supporting MPI [17] is a suitable host. At present we target the Fujitsu AP1000 [18] at Imperial College and a local network of Linux based workstations.

One possibility for the backend compiler is the region based ML Kit version 2 compiler [19] which generates ANSI C code from the ML Kit AST that does not require garbage collection. Unfortunately, a small number of annotations are required at the top level of the code to prevent region growth and the C interface does not allow ML functions to be called from within C. The performance of the generated code is also something of an unknown quantity. Code from this compiler has been successfully run on the AP1000, however, and a future development may be to use this compiler as the backend.

Both `sml2c` [20] and SML of New Jersey [13] were used in the construction of ParaML [21], an implementation of SML for distributed systems, particularly the Fujitsu AP1000. The memory requirements of these compilers caused problems with the AP1000, however, and the AP1000 operating system had to be modified to allow a full SML/NJ implementation on each node.

Objective Caml [22] is a dialect of ML based around a different evaluation mechanism [23] from the SOS of Standard ML. This is a lightweight implementation with modest memory requirements and respectable performance. It was these features that led to the use of Objective Caml as the current backend for our system. Although Caml outputs assembly code directly there is a full C interface which allows us to incorporate both our parallel skeletons and the MPI interface libraries.

The MPI bindings are provided separately in a Caml module. This provides interfaces to *blocking* and *non-blocking* communications and a subset of the *communicator argument* functions. In general, functions are provided for `bool`, `int` (as MPI_INT), and polymorphic data transmission (as MPI_BYTE). Note, however, that the polymorphic routines are not type safe across the transmission. Another problem with the polymorphic routines is that the transmitted data representation is based around the Caml `Marshal` module which is not very space efficient. This routine should really be replaced by a more efficient, dedicated data compression algorithm. At present, ANU MPI [24] (for the AP1000), MPICH [25] and LAM 6.1 [26] (for networks of workstations) MPI implementations can be used.

3 Higher Order Function Skeletons

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

Wrappers have been provided in Objective Caml allowing direct usage of parallel constructs if required. Indeed, one could even write further skeletons in Objective Caml rather than C, an approach advocated by Serot[27]. This also enables enables indirect SML access to the MPI functions, providing the option of writing skeletons directly in SML.

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

3.1 Map Skeleton

The map skeleton uses a processor farm topology to control the processors working over the segments of the map's list argument. Initially the root processor allocates each worker processor a list segment, the size of which will be dependent on the architecture and determined at compile time, thereafter segments are allocated on demand until completion. The root processor does no work on the list itself but acts only as data server and collection point for the final result.

3.2 Fold skeleton

The fold skeleton uses a binary tree architecture. Unlike the processor farm, all processors work over a segment of the list. At each node, other than leaf nodes, the list is split into three, one segment retained locally, and one section forwarded to each child processor. The leaf processors simply work over the list segments supplied to them. As results filter back up the processor tree they are combined at each (non-leaf) node and folded together using the same function as the original segments.

Note that, to parallelise a fold, we must assume that the function argument is associative and commutative.

3.3 Nested Skeletons and Processor Allocation

Arbitrary level nesting of skeletons is being implemented by providing a skeleton as a functional argument to another skeleton. Additional co-ordination information is also required. Each processor needs to know which processor is the root of its skeleton and whether it is also the root for the entire processor tree. Complications arise when a processor is root of a skeleton but not of the entire processor tree: it is a child in some contexts and a parent in others.

Once profiling has decided where HOF's are to be replaced by their parallel equivalents, processors may be allocated. For unnested cases this is simply a matter of allocating the available processors according to the topology of the skeleton. For nested cases this becomes more complex due to processors appearing in more than one skeleton.

For example, in Fig. 3), the root processor sees 3 nested children below itself. Hence, it

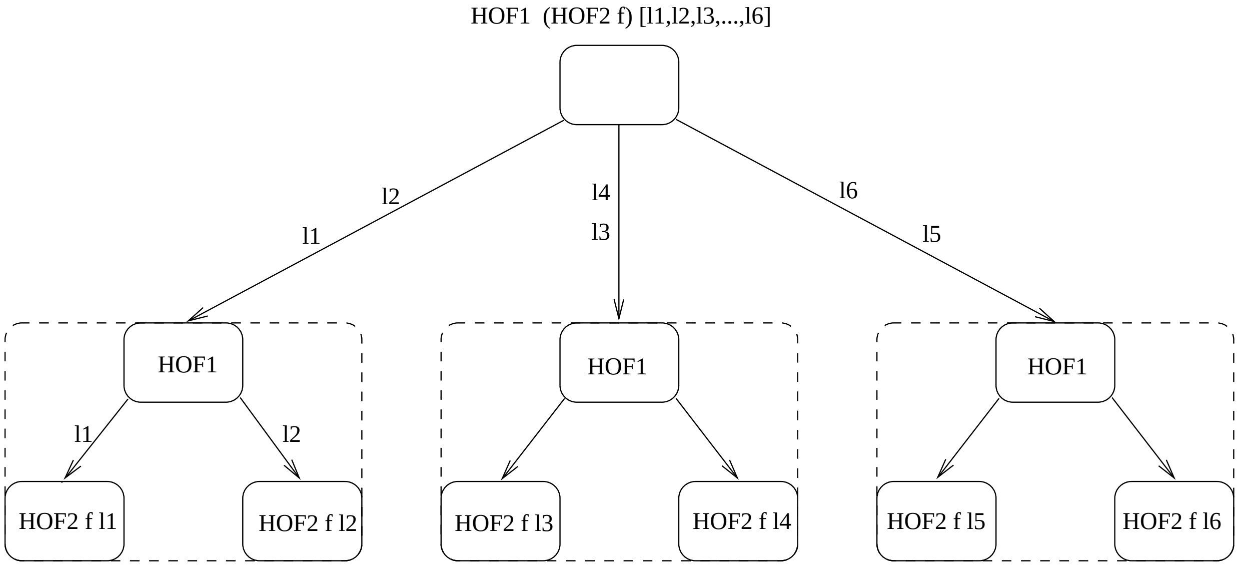

Fig. 3. Nested list decomposition

decomposes the list and sends a number of segments to each child. The receiving processors then re-enter the skeleton (or a different skeleton) once for each of the segments, with the corresponding segments as the data argument.

Note that in the final system, nested functions of the type shown above will only be executed as nested parallel skeletons when profiling indicates that the amount of work justifies the overhead.

3.4 Related Skeletal Approaches

The term *algorithmic skeleton* was first used by Cole [1]. However there are earlier papers working in the same direction. For example, Cole cites ZAPP (Zero Assignment Parallel Processor) [28] which mapped a virtual, divide and conquer processor tree onto a real, fixed network of processor-memory pairs and provided a functional interface. Much work has since been carried out in the area of algorithmic skeletons, variously labelling skeletons as templates, paradigms, patterns, forms and parallel constructs [7].

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

Darlington et al [9] develop on Cole's ideas considering the skeletons with less emphasis on the target architecture. They introduce the concept of inter-skeleton transformation to aid efficient implementation on various platforms. We will be expanding on these transformation techniques, using them not only to convert to skeletons more efficient on a given architecture but also to identify parallelism that does not fall within the scope of our nominated set of HOFs. Later papers [29] introduce performance models associated with each skeleton and instantiated with machine specific parameters, and a co-ordination language [30]. SCL (Structured Coordination Language) has three components, configuration and configuration skeletons, elementary skeletons and computation skeletons. SCL is combined with a base language to form a structured parallel programming scheme. While SCL is functional in form, the base language need not be; indeed, one intention is to enable the reuse of extant components in mainstream imperative languages.

In her thesis, Pelagatti [31] describes a methodology based on a small set of parallel constructs (skeletons) and combining rules. These skeletons are first class objects of the host language, supporting hierarchical nesting. She goes on to demonstrate this methodology with P^3L [32]. P^3L represents a conceptually quite different approach to that of Cole and Darlington with its parallel constructs being representative of the actual parallel operations rather than the semantics. Also, P^3L differs from other approaches in being an imperative language.

4 A Distributed Canny Edge Detector

The Canny edge detector [33] is a method for detecting points of high intensity gradient in images.

Fig. 4 shows an example of the input (an intensity image) and the output (the strength component image) from the Canny edge detection algorithm, processing a 256x256 image. This algorithm has already been studied in some detail and an existing Standard ML implementation is available[34].

The basic structure of the algorithm is a sequence of convolutions, firstly with Gaussian filters for smoothing and then with derivatives of Gaussian filters for edge enhancement. This process is performed separately in the x and y directions and the two enhanced images are resolved into angular and strength components. Finally, some cleaning up is performed by non-maximal suppression filters.

The convolution operation is effectively a 2D fold simultaneously over the filter and a rectangular portion of the image, leading to a value at a point in the image. This has

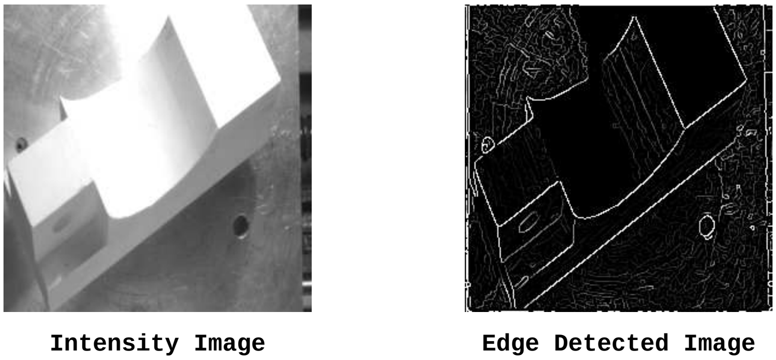

Fig. 4. Canny example

two consequences since data is required from outside the pixel (or region) of interest. Firstly, some form of default value is required for data outside the image. There are several methods of dealing with this, a null value can be used, leading to poor data at the image border or some form of extrapolation can be used which is complex and time-consuming. Another simpler solution which is adopted by the algorithm here, is to accept a degree of shrinkage in the processed image. The other problem is that when an image is decomposed into subimages data must be provided from neighbouring regions to prevent shrinkage of subimages. In this way, the filter size affects the degree of parallelism.

Although the Canny implementation is rich in `map` and `fold` constructs our system is not ready to automate the extraction of parallelism since many of them are at too fine a degree of granularity to be of use. Instead, we re-implement the solution in [34] which works by splitting the image into subimages and farming over the subimages. Some additional "glue" code is required to split the image (with overlap between subimages) and re-constitute the processed subimages into the final image. This allows us to explicitly parallelise the algorithm by applying the `pmap` construct over a list of subimages and has provided good parallelism in the past.

A schematic of this process is shown in Fig. 5.

The image is split into a list of subimages and `pmap` is applied over these, resulting in a list of processed subimages which are then recombined into the processed image.

One optimisation is applied, however. It turns out to be more efficient to broadcast the entire image to the workers and instead of applying `pmap` over the subimages it is applied over a list of subimage definitions which comprise four integers. This technique is common in parallel computer vision systems where, very often, the image is available at frame rate on the worker processors by the provision of a dedicated image data bus. Simplified SML code is shown in Fig. 6.

The image is broadcast to all the workers along with parameters for the Canny phase. A list of subimage definitions as `((xmin,xmax),(ymin,ymax))` values is derived from the image and `pmap` is applied over this list. The result is a list of processed subimages which are recombined into a result image.

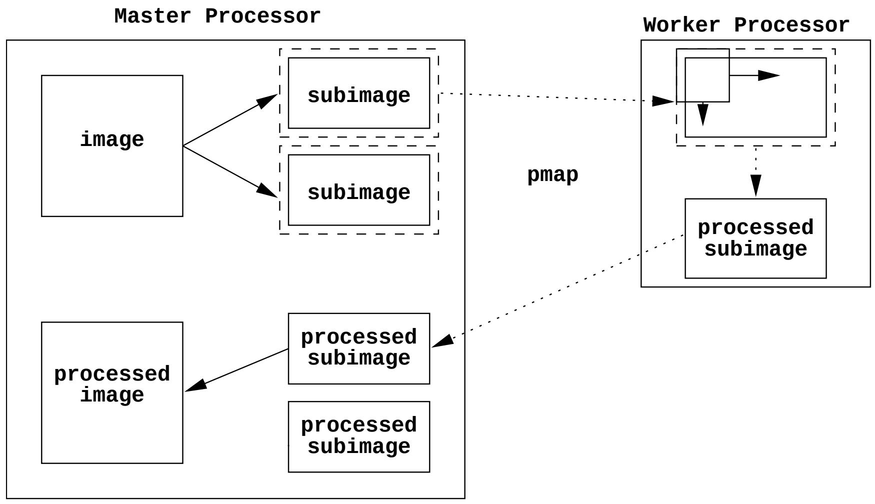

Fig. 5. Canny implementation

```
(* Load the image from file *)
val (hdr,img) = load_image hips_file

(* Broadcast image and parameters to workers *)
val _ = mpi_bcast img host mpi_comm_world
val _ = mpi_bcast (sigma,scaleFactor,lower) host mpi_comm_world

(* Generate list of subimage borders *)
val subimages = gd_portions img

(* Run pmap on the list of subimage definitions *)
val canny_subimage_list = pmap "canny_ff" subimages

(* Build the result image from the subimages *)
val (strength,orientation) = combine_subimages canny_subimage_list
```

Fig. 6. Canny SML code

The performance of the implementation on a network of PC workstations is shown in Figs. 7 and 8.

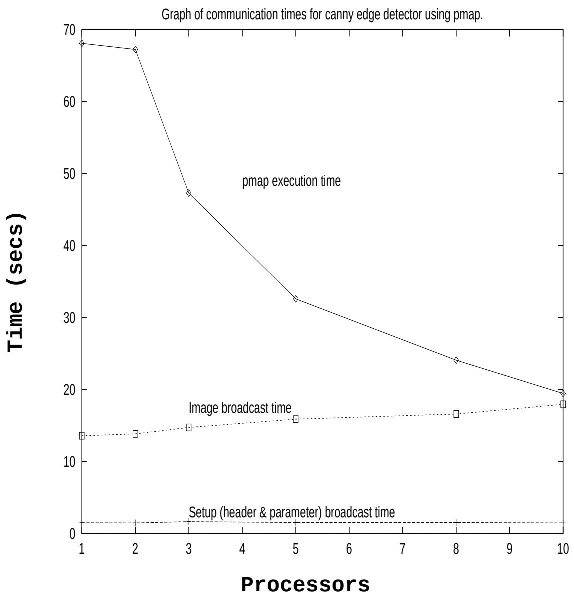

Fig. 7. Distributed Canny execution time

Times are shown for the broadcast of Canny operational parameters, broadcast of the raw image data and for execution of `pmap` over the subimage list. The parallel architecture consisted of ten PC workstations, (8 Pentium P90 with 16Mb ram and 2 P166 with 32Mb RAM), running the Linux operating system using the LAM 6.1 MPI implementation.

Results for Canny edge detector using `pmap`				
No. Processors	T_{setup}	T_{image}	T_{pmap}	Speedup
1	1.52	13.59	68.09	1
2	1.49	13.86	67.24	1.01
3	1.66	14.75	47.3	1.44
5	1.54	15.90	32.60	2.09
8	1.54	16.62	24.09	2.83
10	1.61	17.96	19.47	3.50

Fig. 8. Distributed Canny speedup

5 Conclusions

At first sight these results are not particularly impressive, showing a speedup of 3.5 on 10 processors. However, they compare very well with Koutsakis's three hand implementations of the Canny algorithm in occam from SML prototypes, based on processor farms. These achieved speedups of 5.0, 4.2 and 2.5 respectively on 10 processors, also with a 256x256 image[35].

Furthermore, our results are from a naive specification of the Canny algorithm implemented through direct compilation without optimisation. Thus, we are encouraged that our compiler spine coupled with a base skeleton can deliver respectable parallel performance from a realistic prototype.

Currently, work continues on completing the arbitrary nesting of skeletons, to provide a complete parallelising compiler from SML with explicitly nominated HOFs. The next phase of development will focus on finishing the SOS profiler, and instrumenting SOS test sets and skeletons to enable performance prediction and parallelism identification from SML prototypes. We will then turn to incorporating HOF oriented program transformation between the profiler and the back end, to optimise parallelism.

Acknowledgements

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

References

1. M. I. Cole. *Algorithmic skeletons: structured management of parallel computation.* Pitman, 1989.
2. G. Michaelson, A. Ireland, and P.King. Towards a skeleton based parallelising compiler for sml. In C. Clack, T. Davie, and K.Hammond, editors, *Proceedings of 9th International Workshop on Implementation of Functional Languages*, pages 539–546, Sept 1997.
3. D. Skillicorn. *Foundations of Parallel Programming.* Number 6 in Cambridge International Series on Parallel Computation. CUP, 1994.
4. Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of nesl. In *ACM SIGPLAN International Conference on Functional Programming*, pages 213–225, May 1996.
5. C.B. Jay, M.I. Cole, M. Sekanina, and P.A. Steckler. A monadic calculus for parallel costing of a functional language of arrays. In C. Lengauer, M. Griebl, and S. Gorlatch, editors, *Euro-Par'97 Parallel Processing*, volume 1300 of *Lecture Notes in Computer Science*, pages 650–661. Springer, August 1997.
6. H. W. Loidl. *Granularity in large scale parallel functional programming.* PhD thesis, Dept of Computing Science, University of Glasgow, April 1998.
7. T. Bratvold. Skeleton-based parallelisation of functional programs. In *PhD thesis.* Dept of Computing & Electrical Engineering, July 1995.
8. R. Rangaswami. A cost analysis for a higher order parallel programming model. In *PhD thesis.* Department of Computer Science, University of Edinburgh,, February 1996.
9. J. Darlington, A. J. Field, P. G. Harrison, D. Harper, G. K. Jouret, P. J. Kelly, K. M. Sephton, and D. W. Sharp. Structured parallel functional programming. In H. Glaser and P. Hartel, editors, *Proceedings of the Workshop on the Parallel Implementation of Functional Languages*, pages 31–51. CSTR 91-07, Department of Electronics and Computer Science, University of Southampton, 1991.
10. A. Bundy. The use of explicit plans to guide inductive proofs. In R. Lusk & R. Overbeek, editor, *Proceedings of 9th Conference on Automated Deduction*, pages 111–120. Springer-Verlag, 1988.
11. D. B. Skillicorn and D. Talia. Models and langauges for parallel computation. *Computing Surveys*, June 1998.

12. 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.
13. A. W. Appel and D. B. MacQueen. *Standard ML of New-Jersey*, volume 528 of *LNCS*, pages 1–13. Springer-Verlag, 1991.
14. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.
15. T. Bratvold. *Skeleton-based Parallelisation of Functional Programmes.* PhD thesis, Dept. of Computing and Electrical Engineering, Heriot-Watt University, 1994.
16. D. Busvine. *Detecting Parallel Structures in Functional Programs.* PhD thesis, Heriot-Watt University, Riccarton, Edinburgh, 1993.
17. Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. *International Journal of Supercomputer Applications and High Performance Computing*, 8(3/4), 1994.
18. H. Ishihata, T. Horie, S. Inano, T. Shimizu, and S. Kato. CAP-II Architecture. In *Proceedings of the First Fujitsu-ANU CAP Workshop*, Kawasaki, Japan, Nov 1990. Fujitsu Labs.
19. Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Hjfeld Olesen, Peter Sestoft, and Peter Bertelsen. Programming with Regions in the ML Kit. Technical Report 97/12, Department of Computer Science, University of Copenhagen, 1997.
20. D. Tarditi, A. Acharya, and P. Lee. No assembly require: Compiling Standard ML to C. Technical Report CMU-CS-90-187, Carnegie Mellon University, Pittsburg, Pennsylvania, School of Computer Science, March 1991.
21. P. Bailey and M. Newy. Implementing ML on Distributed Memory Multiprocessors. *SIGPLAN*, 28(1):56–59, Jan 1993.
22. X. Leroy. *The Objective Caml System, available from http://pauillac.inria.fr/ocaml/.* INRIA, 1996.
23. G. Cousineau, P. L. Curien, and M. Mauny. *The Categorical Abstract Machine*, volume 201 of *LNCS*, pages 50–64. Springer Verlag, 1985.
24. D. Sitsky. Implementation of MPI on the Fujitsu AP1000: Technical details release 1.1. Technical Report TR-CS-94-08, Department of Computer Science, The Australian National University, Sep 1994.
25. W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI Message-Passing Interface Standard. *Parallel Computing*, 22(6):789–828, 1996.
26. G. D. Burns, R. B. Daoud, and J. R. Vaigl. LAM: An Open Cluster Environment for MPI. In *Supercomputing Symposium '94*, Toronto, Canada, Jun 1994.
27. J. Serot. Embodying parallel functional skeletons: An experimental implementation on top of MPI. In C. Lengauer, M. Griebl, and S. Gorlatch, editors, *Euro-Par'97 Parallel Processing*, volume 1300 of *Lecture Notes in Computer Science*, pages 629–633. Springer, August 1997.
28. D.L. McBurney and M.R. Sleep. Experiments with the zapp: Matrix multiply on 32 transputers, heuristic search on 12 transputers. Internal Report SYS-C87-10, School of Information Systems, University of East Anglia, 1987.
29. J. Darlington, M. Ghanem, and H. W. To. Structured parallel programming. In *Proceedings of MPPM, Berlin.* 1993.
30. J. Darlington, Y-K. Guo, H. W. To, and J. Yang. Functional skeletons for parallel coordination. In S.HAidi, K. Ali, and P. Magnusson, editors, *Proceedings of EuroPar'95*, pages 55–69. Springer-Verlag, August 1995.
31. Susanna Pelagatti. *A methodology for the development and support of massively parallel programs.* PhD thesis, Dipartimento di Informatica, Universita degli studi di Pisa, November 1993.
32. M. Danelutto R. Di Meglio S. Orlando S. Pelagatti and M. Vanneschi. A methodology for the development and the support of massively parallel programs. *Future Generation Computer Systems*, 8:205–220, August 1992.
33. J. Canny. A Computational Approach to Edge Detection. *IEEE Trans. Pattern Analysis and Machine Intelligence*, PAMI-8:679–698, 1986.
34. N. R. Scaife, G. J. Michaelson, and A. M. Wallace. Four Skeletons for Computer Vision. In *Implementation of Functional Languages '97*, Sep 1997.
35. G. Koutsakis. Parallel Low Level Vision from Functional Prototypes. Master's thesis, Dept. of Computing and Electrical Engineering, Heriot-Watt University, Riccarton, Edinburgh, Oct 1993.