A Skeleton-Based Parallelising Compiler for ML

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

Languages with explicitly parallel constructs typically give the programmer detailed control of low level machine resources. Such control allows programs to be finely tuned to deliver high performances, but developing massively parallel programs from scratch in this way is a tedious and delicate task. Skeletons provide powerful abstractions over common patterns of parallel computation, and can be expressed as higher order functions (HOFs) in functional languages. In this paper we present SkelML, a skeleton-based compiler for ML, that exploits parallelism in certain HOFs and in function composition. Profiling information and performance models for skeletons are used to decide whether and how parallelism should be exploited. Issues of design and implementation are discussed, and an example is presented illustrating the compilation process and the performance of a prototype of the system.

1 Introduction

Languages with explicitly parallel constructs typically give the programmer detailed control of low level machine resources. Such control allows programs to be finely tuned to deliver high performances, but developing massively parallel programs from scratch in this way is a tedious and delicate task. The programmer's unit of attention is often that of the *individual* processor with concern for low level issues, while paradoxically, parallelism is typically exploited according to a few well known parallel paradigms, each determining the operation and coworking of a group of processors. Over the last few years several compilers and packages for parallel programming have emerged which provide means for abstraction over common patterns of parallel computation, e.g. [1, 5, 7, 8, 14]. We will refer to such patterns as *skeletons*. A programmer has at his/her disposal a number of constructors, each corresponding to a particular skeleton. By instantiating the constructors with specifications for the particular subtasks to be solved, parallelism can be exploited with minimal concern for low level details. Ideally a programmer should specify no more than the semantic *meaning* of a program, while the compiler should be left to work out the exact *behaviour* on a given target architecture. However, the successful separation of these two fundamental issues requires compilers that do not perform significantly worse than a skilled programmer.

Unfortunately a high level of abstraction is often overshadowed by a decrease in performance. Dealing successfully with complex low level issues of parallelism often seems to require human insight and heuristics, and compilers have serious problems in competing with skilled programmers. However, the low level code for implementing a skeleton is mostly fixed, and can be implemented in optimized form once and for all. It can then be reused *ad infinitum*, each time supporting the particular task specific code with which it has been instantiated. Much complexity can be hidden inside skeletons, easing considerably the compiler's task. In fact it is argued that by restricting the exploitation of parallelism to that of skeletons only, a compiler may do at least as good a job as a programmer [12]. By using accurate performance models of each individual skeleton, the compiler can work out its optimal instantiation. As the program then consists of fewer entities, each of which has a well modeled behavior, automatic techniques can be applied for mapping and scheduling.

Due to evaluation order independence, functional programming languages are good candidates for parallel programming. Skeletons can be expressed naturally through their semantically equivalent higher order functions (HOFs), as pointed out by Cole [6]. An appealing property of functional languages is that they offer rich possibilities for semantics preserving transformations. Thus, by transforming one HOF or an application of HOFs into another, one can manipulate the corresponding structures in the parallel program. Such transformations can be used both for optimising performance, and also for compiling down to skeletons that have particularly efficient implementations on some specific architecture. Since functional programs generally contain rich amounts of parallelism at all levels, it may also be possible to successfully extract and exploit skeleton parallelism *without* the explicit occurrence of the corresponding HOFs. We will return to this possibility in section 2.3.

This paper presents a parallelising compiler for ML which is currently under development. Parallelism is identified in selected HOFs and in function composition. We target our implementations to a transputer based Meiko Computing Surface[1], but make few assumptions that conflict with implementation on other distributed memory MIMD architectures.

The following section contains an overview of the compiler, discussing issues of design and implementation. Next, an example is provided illustrating how a prototype of the compiler handles the parallelisation of a ray tracer. Performance figures are presented and commented upon. Finally, we give some concluding remarks and outline avenues of future work.

2 An Overview of the SkelML Compiler

The SkelML compiler is strictly speaking not a true compiler, but rather a source to source translator that produces parallel occam2 from ML. The task of the compiler is to identify useful parallelism in the source program and figure out how best to structure pieces of sequential code for parallel execution accordingly, in order to achieve optimal performance. For translation of ML to sequential occam2 we employ Busvine's translator [3], from now on referred to as *s2o*. s2o was originally developed for the *PUFF* compiler [4], and has proved to be both reliable and efficient.

2.1 The programming environment

The chosen functional programming language is a large pure subset of Standard ML's core language [11]. In choosing a suitable set of features, the emphasis has been on providing full expressive power, but excluding a few comforts that could potentially add unnecessary weight to the task of compilation. The most important features not supported are records, ADTs and exceptions, with some of the expressive power of the former two being provided through tuples and user defined datatypes. We adopted a language with strict rather than with non-strict semantics as strict sequential implementations tend to have an edge in performance.

The front end of the SkelML system is an interactive garbage collecting interpreter for ML, built to have the same look and feel as the Edinburgh and New Jersey compilers. The interpreter has

[1] Meiko and Computing Surface are registered trademarks of Meiko Ltd

an additional set of commands and switches for controlling and invoking the parallelising compiler, but may be used independently for developing and running ML programs.

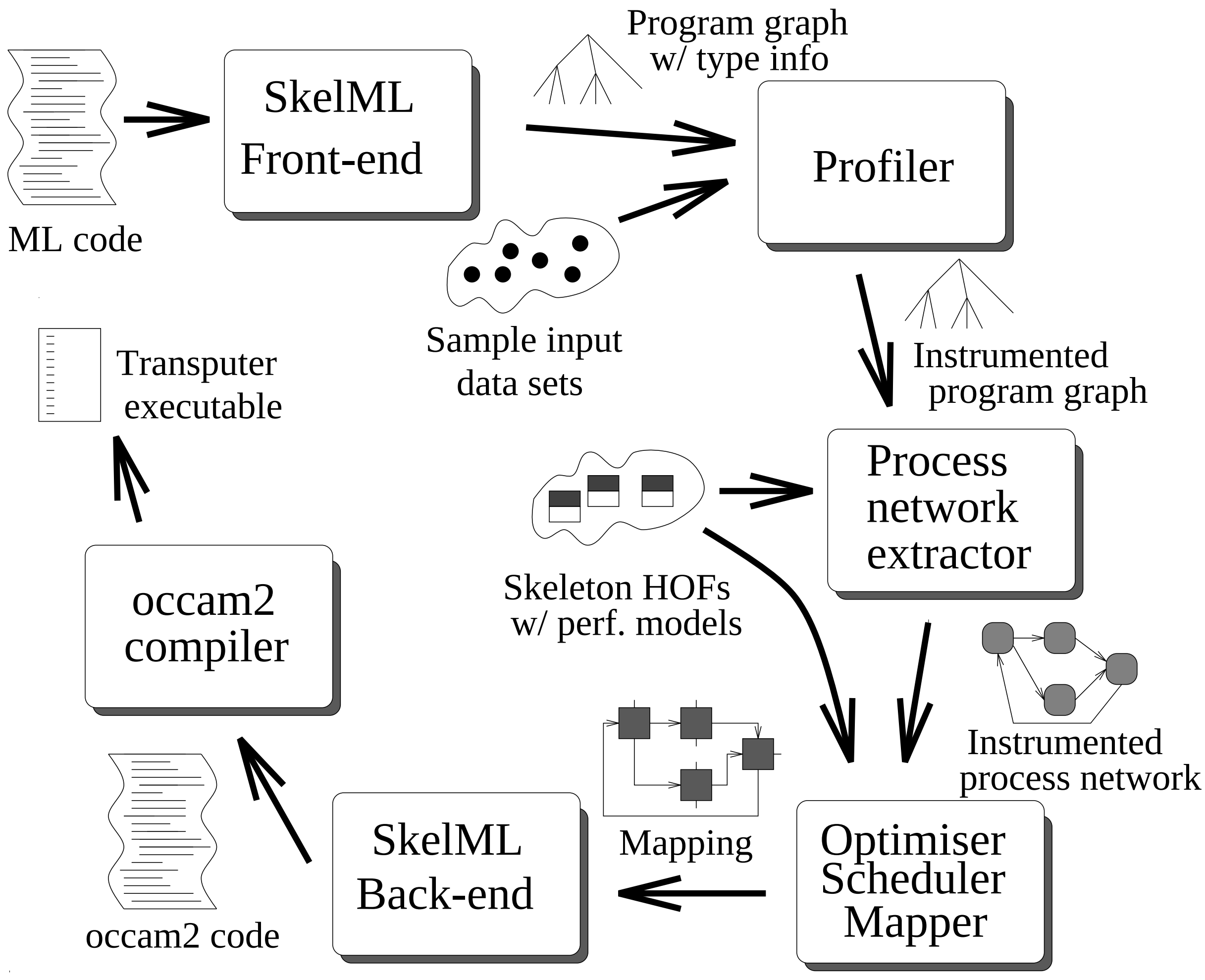

Figure 1. Overview of compilation stages

2.2 Handling complexity information

High quality information on the run time behaviour of the program is a prerequisite for the paralleliser to do a proper job. Such information includes branching probabilities, execution times for arbitrary expressions, depth of recursion and length/depth of recursive data structures. The type checker provides information for determining the size of data elements to be passed between parts of the program, and in some trivial cases it is also possible to work out at compile time the computational cost of evaluating certain expressions in the program. However, most of the information needed is dependent on the program's input, and can not be derived at compile time for realistic programs, even if advanced analysis techniques [13] were to be used. One possibility is to rely upon programmer annotations [9] or to have a dialogue between the programmer and the compiler. However, we do not find these alternatives particularly attractive. Our very reason for providing a high-level program development platform is precisely to free the programmer from having to deal with detailed issues of parallelism and the low level behaviour of his/her program. It does not seem logical to put this burden back onto the programmer's shoulders. Even if we were to do so, the sheer quantity, not to mention quality, of the information required would render this strategy impractical.

Our experience suggests that run time profiling with typical data sets is a better approach [15, 16]. By recording detailed statistics of the program's execution behaviour, the paralleliser has a rich base of accurate information for making its decisions. The profiling mechanism is activated from the SkelML interpreter by setting a flag, after which instrumentation data will be recorded during subsequent evaluation.

In determining the computational cost of evaluating an expression, it should be noted that the cost should in fact reflect the time for transputer execution of the equivalent sequential occam2 code as produced by the s2o translator. The profiler keeps track of the execution steps corresponding to the inference rules as defined in the formal structural operational semantics of Standard ML [11]. For each inference rule or combination of inference rules, a formula estimates the transputer execution time for the equivalent instructions in occam2. The formulas and a number of constants for the execution costs of atomic operations have been derived from series of test runs monitoring the performance of the occam2 code.

It should be noted that profiling may in some cases fail to produce relevant data. If for the program in question there is no such thing as a "typical" set of input data, it is hard to make general assumptions at compile time about the program's behaviour. However, this is a general problem in making decisions at compile time, and one which can not be attributed to profiling in particular.

2.3 Creation of process network

SkelML exploits parallelism in two principal forms: in skeletons, and in process pipelines. A pipeline is sometimes viewed as being a particular kind of skeleton, but we prefer to think of it as a glue that holds the various parts of the program together.

The equivalent functional construct of a process pipeline is function composition. Consider the simple composition

```
f1 o f2
```

Parallelism can be exploited in two related cases:

- when the type of `f1`'s argument contains a constructed type, e.g. a list, that is consumed element by element, and likewise `f2` produces the same structure element by element. `f1` and `f2` can then be assigned to separate groups of processors, simultaneously working on *distinct parts of the same data structure*. A typical example of this is a pipeline of two processor farms.
- when the composition `f1 o f2` is repeatedly applied to a series of arguments, `f1` and `f2` can be assigned to separate groups of processors, `f2` passing its result on as an argument to `f1`. `f1` and `f2` are then always working on *distinct data elements*. An example is a processor farm in which each worker is a pipeline.

The successful exploitation of pipeline parallelism is subject to a number of factors, most notably the pace of computation at each stage and the amount of data to be passed through the pipeline.

In the current implementation skeletons are identified by the programmer's use of certain HOFs. This may seem a rather explicit approach, but one should note that

- HOFs are a natural part of functional languages, and not artificial constructs added to expose parallelism.
- The HOFs in question are in many cases widely used by functional programmers today, not because of their implicit parallelism, but simply because they are powerful and convenient ways of describing certain algorithmic properties.
- The use of a particular HOF in a program is *not* a directive from the programmer indicating that parallel evaluation should take place. This matter is left entirely to the compiler to decide.

There are however ways of discovering skeleton parallelism *without* relying on the presence of HOFs. One possibility demonstrated in [4] is to use processor farms for implementing arbitrary linear recursion. Other forms of recursion may for example be implemented in divide-and-conquer style using a tree of processors [2]. Such techniques of extracting skeleton parallelism are strong candidates for investigation and inclusion in a future version of our compiler. Compared to systems in which a pipeline is represented by an explicit HOF, extracting pipeline parallelism from function composition is one such technique that is already supported by SkelML.

The task of the *process network extractor* is to transform the program graph into a network of nodes and directed arcs. An arc is means of transporting data between nodes. Each arc has a source node and a destination node, as well as information about the data it will carry, e.g. its type. A node may have any number of input and output arcs connecting it to other nodes.

Note that nodes and arcs are abstract entities, and should not be confused with physical processors and communication links. It is the subsequent task of the scheduler and mapper to decide how nodes are to be placed on the available processors, and how arcs are to be multiplexed on the communication links.

There are three principal types of nodes:

- A unique top level *terminal node* reads in the program's arguments, sends them out to other nodes for processing, collects the result and finally presents it to the user.
- A *sequential node* contains a piece of code for sequential execution.
- A *skeleton node* corresponds to a particular skeleton instantiation, and contains references to all the arguments of the corresponding HOF

Nodes can be connected in any number of ways depending on the corresponding structures in the functional program. The resulting network serves as an intermediate representation in which potential parallelism is exposed.

2.4 Optimisations, scheduling and mapping

The process of mapping the process network down to the physical architecture is a complex and non-trivial task. The number of available processors and their flexibility of interconnection impose severe restrictions that must be adhered to. We base ourselves on a global cost function produced partly on the basis of the performance models of the skeletons and on the characteristics of the physical architecture (e.g. bandwidth), and also on the instrumentation data derived during profiling. We take an iterative approach to mapping, aiming at each stage to apply transformations between skeletons to optimize the process network. The network is mapped down to the physical architecture, taking into account how tasks can be best be assigned and scheduled on the available processors. Then the cost function is applied to evaluate the result. This process can be applied repeatedly, trying various transformations until some saturation criteria is met and the best mapping is found.

2.5 Communication

There are three principal forms of sending and receiving data between parts of the program:

1. When data is passed between processes running concurrently on the same processor it suffices to communicate a pointer to the processor's local memory.
2. When passing data from the local memory of one processor to the local memory of another, data structures must be flattened, encoded, transmitted, and eventually rebuilt at the receiving end.
3. When data is routed through an intermediate processor it is transferred from one channel to another without any intermediate storage or rebuilding.

Efficient communication is crucial for the performance of the resulting parallel program, and finding good algorithms and protocols for data transfer constitutes an important part of developing the compiler. A matter of particular interest is the clustering of smaller data elements into larger packets in order to minimize the number of *rendez-vous*'s and associated synchronisation delays. Another is the efficient implementation of buffers for intermediate storage of data. The current implementation relies on simple and straightforward protocols, intended for further optimisations in the near future.

2.6 Code generation

Generation of occam2 code is shared between the s2o translator and SkelML itself. For sequential pieces of code SkelML calls s2o to produce sequential occam2. Standard code for each supported skeleton is emitted with minor adjustments to support the particular instantiation. In addition code is produced for the administration and scheduling of processes on each processor, including code for supporting communication protocols and for including sequential code from s2o. Code for processor placement and wiring tables are also emitted, as well as code for the interaction with the user; i.e. for the input of program arguments, and for the output of the result.

The resulting occam2 code is fairly readable, which is a particular advantage while the compiler is under development. It is thus relatively easy to debug the parallel code, and to modify it for example to produce diagnostics or to monitor its performance.

2.7 Supported skeletons

The compiler is fully extensible, such that skeletons and their respective occam2 implementations and performance models can be added easily. At the time of writing, four skeletons[2] plus the process pipeline are supported. The four are all based on processor farms:

- `map` applies a function to all elements of a list
- `filter` filters out those elements of a list satisfying a boolean condition
- `mapfilter` is equivalent to a composition of `map` and `filter`
- `filtermap` is equivalent to a composition of `filter` and `map`

These HOFs constitute a base for testing and experimenting with various aspects of the compiler. A fully fledged implementation should have a substantially larger base of skeletons, and our system is expected to be expanded gradually over time.

2.8 Current state of implementation

The SkelML compiler is currently under development, and a prototype implemented in C++ is compiled for execution on a Sun SPARC[3] platform. The prototype is by no means complete, but does facilitate successful compilation of simple programs. At the time of writing most of the compiler's core parts are implemented and running. This includes prototype versions of the front-end, the profiler, the process network extractor, the skeleton code generator, the integration with s2o and the generation of communication protocols and wiring tables. Not yet incorporated are performance models and the optimisation/mapping/scheduling stage. The latter is for the moment temporarily replaced by a direct mapping from nodes and arcs to processors and communication channels respectively.

3 Example - A Ray Tracer

We present an example application to illustrate some aspects of the compiler and its performance. It should be noted that the results given are a direct product of the current state of the prototype, and that a more complete and optimised implementation may perform differently.

Consider a ray tracer based on an example in [10] with the following top level structure:

```
datatype Impact = IMPACT of real * int | NOIMPACT;
  (* IMPACT(dist,objno) indicates an impact at distance dist with object objno,
     NOIMPACT indicates no impact *)

fun FirstImpact objects ray = ... ;
  (* Finds first impact, or NOIMPACT, between ray and objects *)

fun FindImpacts rays objects = map (FirstImpact objects) rays;
  (* Finds all impacts between objects and rays *)
```

[2] The definition of the corresponding HOFs can be found in the appendix
[3] Sun and SPARC are registered trademarks of Sun Microsystems, Inc.

```
fun GenerateRays (Detail,X,Y,Z) = ... ;
  (* Generates Detail * Detail rays from viewpoint (X,Y,Z) towards origin *)

fun top (Detail,XView,YView,ZView) Scene =
             FindImpacts (GenerateRays (Detail,XView,YView,ZView)) Scene;
  (* Toplevel function *)
```

Rays are traced from the viewpoint given by the `XView,YView,ZView` coordinates. The integer value `Detail` determines that `Detail * Detail` rays are to be generated. The `Scene` parameter is a list of objects against which the rays are tested for impact. The result of the program is a list of impact information for each ray. If a ray does not hit any objects then the constructor `NOIMPACT` is returned. If a ray strikes one or more objects then the impact closest to the viewpoint is returned, represented by the constructor `IMPACT`. This constructor carries information about the distance to the object, and the object's identifier which is an integer value. In a more realistic program the constructor could have contained other data, for example colour and intensity information.

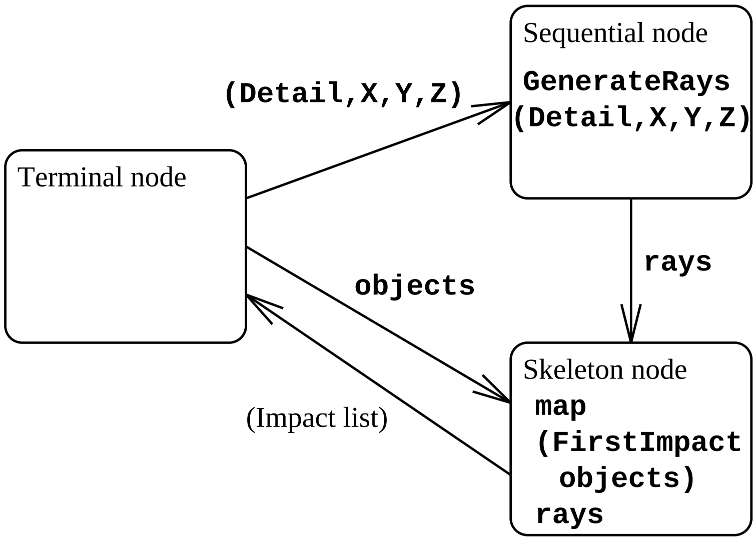

Figure 2. Process network

Feeding this program to SkelML, it identifies the `map` in `FindImpacts` as a potential source of farm parallelism, and produces a process network as shown in figure 2. As the current prototype has no performance models built in, it is unable to make any decisions about the usefulness of the parallelism exposed. We assume that the skeleton node contains useful parallelism, and allow SkelML to produce code for supporting a processor farm. The produced occam2 code and corresponding wiring information for the farm is flexible with respect to the number of farm workers, which can be controlled by adjusting a constant. For this example we decide to look at the performance for up to 20 workers.

We further instruct SkelML to produce sequential code for the sequential `GenerateRays` node, but decide to schedule that task on the dedicated *farmer* processor of the farm, rather than to use a further processor. The resulting mapping is shown in figure 3. In a complete implementation of the compiler this whole process would have been largely automated. Note that in the function to be computed in each farm worker, (`FirstImpact objects`), `objects` is free, and must thus be broadcasted to all the workers prior to farming out the rays.

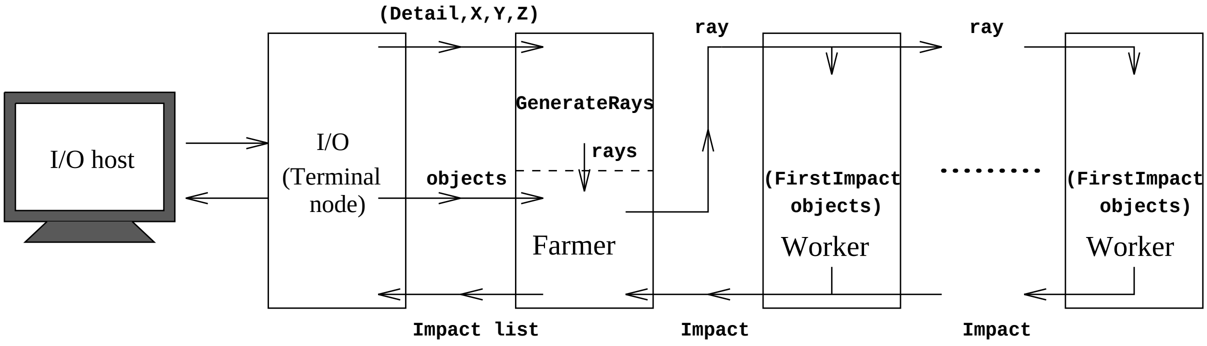

Figure 3. Processor mapping

The resulting parallel program was tested with two scenes. Scene 1 contains six quadrilaterals, while scene 2 contains eight hexagons, three quadrilaterals and one octagon. Two values for `Detail` were tried: 12, giving 144 rays, and 25, giving 625 rays.

Farm workers	Scene 1, Detail 12		Scene 1, Detail 25		Scene 2, Detail 12		Scene 2, Detail 25	
	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
(seq)	636	-	2769	-	1857	-	8116	-
1	683	0.93	2926	0.95	2000	0.93	8696	0.93
2	382	1.66	1610	1.72	1042	1.78	4500	1.80
3	281	2.26	1172	2.36	728	2.55	3099	2.62
4	231	2.75	954	2.90	567	3.28	2403	3.38
5	202	3.15	822	3.37	471	3.94	1982	4.09
6	181	3.51	735	3.77	408	4.55	1705	4.76
8	156	4.08	625	4.43	329	5.64	1350	6.01
10	142	4.48	560	4.94	286	6.49	1144	7.09
12	131	4.85	516	5.37	261	7.11	1006	8.07
16	120	5.30	462	5.99	240	7.74	831	9.77
20	112	5.68	430	6.44	237	7.84	728	11.15

Table 1. Farm size vs execution times in milliseconds and speedup

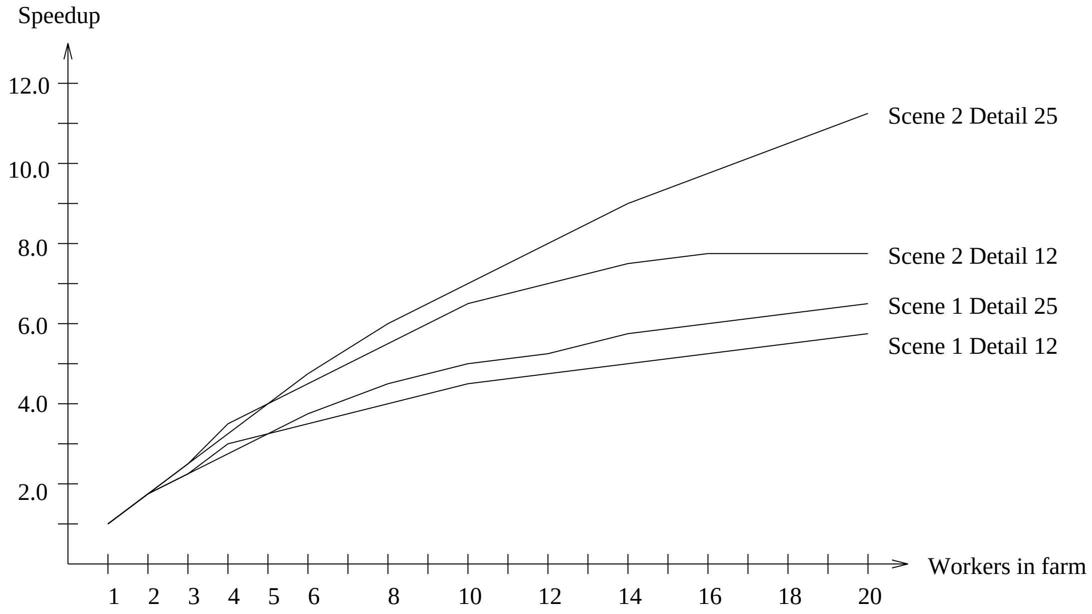

Figure 4. Farm size vs speedup

SkelML has a command for calling s2o to produce a sequential occam2 version of the program, without any regard for parallelism. We use the execution times for this version as a basis for calculating absolute speedup figures for the parallel program. The sequential run times as well as the parallel ones and the respective speedup figures are presented in table 1, while figure 4 compares the speedup figures of the four input sets.

Looking at the performance results, we see that when the number of farm workers is large, efficiency is far from optimal. However, we rate the speedup figures obtained for up to five workers as acceptable. At this stage in the development of the compiler we consider the results to be promising, especially as they are comparable with those obtained by PUFF for the same application [4].

4 Conclusions and Future Work

We have argued that using skeletons for parallel programming eases the job of both the programmer and the compiler. The high level of abstraction enables the programmer to specify his/her program in terms of its semantics, while the compiler can adopt a structured and modular approach to parallelisation and mapping. Functional programming is a promising frame in which to work with skeletons, as they can be expressed naturally as higher order functions. We have presented issues of design and implementation of a parallelising compiler for ML based on this approach. The results of applying a prototype of the compiler to a ray tracer example indicated that acceptable speedup figures are obtainable, but that there should be room for improvements. There is still much work to be done, both in terms of refining current features, and in adding more facilities. In particular we will dedicate the near future to incorporating performance models for skeletons, and to implementing algorithms for mapping and optimisations.

5 Acknowledgments

Acknowledgments are due to David Busvine, who developed the s2o translator, and who also wrote the SML code for the ray tracer used in the example. Thanks are also due to Greg Michaelson for helpful suggestions during the design and implementation of SkelML, and for reading and commenting on an early version of this paper. Finally, the author is grateful to Heriot-Watt University for financial support.

References

[1] 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.

[2] F W Burton and M R Sleep. Executing Functional Programs on a Virtual Tree of Processors. In *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*, pages 187–194. ACM, 1981.

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

[4] David Busvine. Implementing Recursive Functions as Processor Farms. *To appear in Journal of Parallel Computing*, 1993.

[5] Lawrence W Chan. Algorithmic Skeletons in Concurrent ML. Technical Report EPCC-SS92-07, Edinburgh Parallel Computing Centre, September 1992.

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

[7] Marco Danelutto and Roberto Di Meglio et al. A methodology for the development and the support of massievly parallel programs. *FGCS*, 8:205–220, 1992.

[8] J Darlington et al. Parallel Programming Using Skeleton Functions. In *Proceedings of PARLE93, Munich, Germany*. Springer-Verlag, LNCS, 1993.

[9] Paul Hudak. Para-Functional Programming. *Computer*, 19(8):60–70, August 1986.

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

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

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

[13] Mads Rosendahl. Automatic Complexity Analysis. In *Functional programming languages and computer architecture, conference proceedings*. ACM press, 1989.

[14] Chris Thornborrow and Chris Feagle. Parallel Modules in Modular Visual Environments. In *Parallel Processing for Graphics and Scientific Visualisation, Conference Proceedings*. University of Edinburgh, May 1993.

[15] A M Wallace, G J Michaelson, P McAndrews, K G Waugh, and W J Austin. Dynamic Control and Prototyping of Parallel Algorithms for Intermediate- and High-Level Vision. *IEEE Computer*, 25(2), February 1992.

[16] Kevin Waugh, Patrick McAndrew, and Greg Michaelson. Parallel Implementations from Functional Prototypes. Technical Report TR 90/4, Department of Computer Science, Heriot-Watt University, Edinburgh, 1990.

A Appendix: HOF definitions in ML

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

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

fun mapfilter f p = (map f) o (filter p);

fun filtermap p f = (filter p) o (map f);
```