Determining Useful Parallelism in Higher Order Functions

Tore A. Bratvold
Heriot-Watt University[1]

Abstract

Programs written in explicitly parallel languages are often composed of pieces of task specific code structured and interwoven with a limited number of well known parallel patterns. While such programs may indeed be very efficient in terms of execution speed, they are usually very hard both to write and reason about, and portability is low. The approach of expressing parallel patterns as templates or higher order functions has received attention lately, both from a theoretical point of view and in practical implementations. This paper presents an outline of the compilation stages in a proposed system based on this approach in combination with functional programming. The presence of useful parallelism is discussed for three higher order functions implemented as processor farms, and an example is presented to illustrate the approach.

1 Introduction

An ideal language for parallel programming will have to satisfy simultaneously many conflicting requirements. The main problem is that of achieving a high performance without having to explicitly allocate low-level machine resources from within a program. Another is to obtain some degree of portability at the source code level. By restricting the ways in which parallelism may be exploited, we believe it is possible to automate the compilation process to such a degree that these requirements can be met.

It has been observed [9, 11] that parallel programs written in explicitly parallel languages consist of two different kinds of code, often tightly interwoven; pieces of *task specific code* implementing the individual steps of the algorithm, and code for structuring the program into *patterns of computation and communication* for parallel execution. It is typically only in the latter kind of code that there is a need for describing low-level resource allocation and for dealing with those aspects of parallel programming known to pose the greatest problems. The number of commonly used patterns is observed to be quite small [11, 13, 16], but the code for their efficient implementation varies greatly between target machines. This is an unfortunate consequence of the need to exploit and adapt to the precise characteristics of the underlying hardware to obtain satisfactory performance.

An appealing approach is to implement efficiently a fixed repertoire of useful patterns, which can be subsequently scaled and instantiated with task specific code during compilation. We believe very efficient code can be produced in this way. It may seem like a drawback that as far as parallelism is concerned, the only interface to the computer is through a limited number of patterns. Hence, if the parallelism present in the original algorithm cannot be expressed by the available patterns, it cannot be exploited. This is however the essence of the whole approach; it is *not* the aim to exploit all existing parallelism present in the algorithm. Instead emphasis is placed on exploiting precisely that parallelism which is *a priori* known to be efficiently exploitable. Also from the point of view of architecture independent computation, it has been argued that restricting the form of computations is a worthwhile sacrifice [18].

P^3L [8] of the Pisa Parallel Processing Project [2] is an example of a language in which this approach has been adopted. The programmer explicitly specifies available parallelism by the use of parallel *constructors*, each of which represents a pattern of parallel computation and communication, for example a pipeline, a tree or a farm. The task specific code is written in a sequential imperative language, and is hierarchically structured by the constructors. The compiling tools work out how the machine resources should be allocated to give the best parallel performance for the program on a given distributed memory (DM) MIMD target architecture.

Many parallel patterns can be expressed neatly as higher order functions (HOFs) in a functional programming language. Cole [7] has proposed the use of functional *skeletons*, which are HOFs designed to capture a class

[1] Author's full address: Department of Computing and Electrical Engineering, Heriot-Watt University, Riccarton, Edinburgh EH14 4AS, United Kingdom. Email: tore@cs.hw.ac.uk

of algorithmic parallel computations, for example divide-and-conquer. Skeletons and P^3L constructors can be seen as high-level templates for parallel computation.

Our approach is closely related, based on identifying HOFs in the functional program and exploiting efficiently their implicit parallelism. There are at least three additional advantages to this:

1. Parallelism is not expressed *explicitly*, but is derived from the *implicit* use of HOFs in the functional program. The programmer's task is thus shifted from expressing parallelism towards expressing functionality.
2. The task specific code is written in a declarative, functional language, considered to have advantages over traditional imperative languages in terms of program development [12].
3. The functional style allows program transformation techniques [4] to be applied in several stages of the compilation process.

We concentrate on DM MIMD architectures, using a Meiko Computing Surface as a testbed for our implementations. A pure subset of Standard ML with strict semantics is used for functional coding, and occam2 is chosen as the target parallel language. Initial work on vision algorithms [24, 25] have demonstrated promising results.

Related ideas of functional programming based on HOFs (or second order functions) can be found in Backus' FP [1] and in the Bird-Meertens formalism [3], which have both been proposed as bases for parallel implementations [14, 18, 19, 20, 22]. Work based on similar ideas is also reported to be in progress [10, 17].

The rest of this paper is organised as follows: section 2 gives a brief overview of the steps involved for compiling a functional program into parallel code, section 3 discusses how to determine whether useful parallelism is present in three selected higher order functions, section 4 presents an example program illustrating how these techniques can be put to use, and section 5 contains a few concluding remarks.

2 Skeleton Compilation

Compilation is based on the following sets, the contents of which are largely dependent on the characteristics of the target hardware :

- $\mathcal{P}$ - a set of patterns $\{p_1,\ p_2,\ ...,\ p_n\}$ for which efficient parallel implementations with corresponding performance models exist on the target architecture.
- $\mathcal{H}$ - a set of HOFs $\{h_1,\ h_2,\ ...,\ h_m\}$ corresponding to patterns in $\mathcal{P}$. We will refer to HOFs in $\mathcal{H}$ as $\mathcal{H}$-HOFs.

There is also a need for transformation rules between different HOFs in $\mathcal{H}$.

The compilation process aims to translate the functional source code into a parallel program for the target architecture, preferably with minimal guidance other than the source program itself.

The compilation process can be divided into the following five steps (see figure 1) :

Lexical/Syntax Analysis and Type Checking
The inferred type information of all expressions in the program is needed later for determining the size of objects to be communicated between parallel processes.

HOF Identification/Extraction
The program graph is analysed to identify the direct or indirect presence of $\mathcal{H}$-HOFs. This will involve various transformation and structural matching strategies. As we only aim to exploit parallelism in the form of patterns, potential parallelism not uncovered at this stage will not be subject to exploitation. It is therefore of critical importance that the extraction mechanism is powerful.

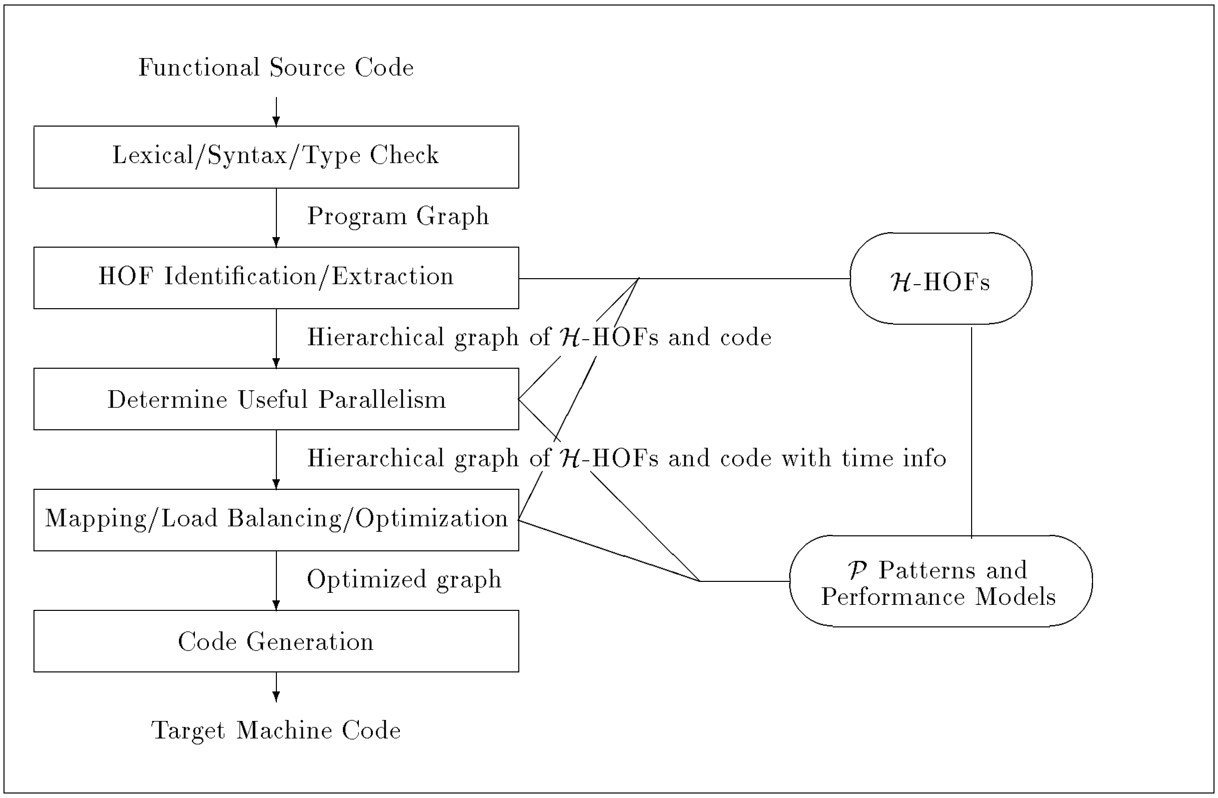


Figure 1: Compilation Stages

Determine Useful Parallelism
For each $\mathcal{H}$-HOF application it must be determined if and to what extent useful parallelism is present, i.e. whether exploiting the implicit parallelism will result in increased performance. This will normally involve a combination of static analysis and profiling, and is dealt with further in section 3.

Mapping, Load Balancing and Optimization
The information derived in the previous stage is used to determine how the machine resources should best be allocated for the different parts of the program. This may involve the use of transformation rules for local optimizations on the program graph.

Code Production
The $\mathcal{H}$-HOFs found worthy of parallel exploitation are replaced by code for the corresponding patterns in $\mathcal{P}$ scaled to be executed on an appropriate number of processors. The task specific code is translated to sequentially executable code [5] and slotted into the patterns as specified by the program graph. The resources of the target machine is distributed according to the mapping derived above.

3 Determining Useful Parallelism

We use the terms *ideal* parallelism or *potential* parallelism to mean all parallelism present in a program or an algorithm. By *useful* parallelism we mean that subset of ideal parallelism which will yield positive absolute speedup when exploited on a given parallel architecture. The $\mathcal{H}$-HOFs all contain ideal parallelism, known to be exploitable under certain conditions determined by the performance model of their parallel equivalents in $\mathcal{P}$ running on the target machine. Given the application of a $\mathcal{H}$-HOF to its argument function(s), we would like to know to which degree the potential parallelism present can be exploited. Erroneous assumptions in this respect may, if the amount of useful parallelism is underestimated, cause useful parallelism to be less than fully exploited. Correspondingly, if useful parallelism is overestimated that may in turn lead to waste of machine resources and even negative (i.e. less than 1) speedup. Such inaccurate estimates not only affects the local performance of the HOF itself, but because HOFs may be interconnected, it may also degrade the performance of the program as a whole. It is therefore desirable that the information about useful parallelism should be precise, or else sufficient slack must be built into the compiled code to compensate for possible inexact estimations. Further, to balance the load of all parallel patterns in the final program, we would like to be able to control the amount of parallelism exploited in each pattern within certain bounds, e.g. by adjusting the number of processors allocated.

In the following, we will concentrate on determining useful parallelism in three $\mathcal{H}$-HOFs for processing of homogeneous multisets (bags) whose natural implementation is that of a processor farm. Composition of these HOFs is implemented as a process pipeline. ML lists are used for representing multisets, but the internal ordering of the elements is irrelevant.

3.1 Processor Farms and Performance

A processor farm is a very general pattern of parallel computation that can be used for implementing efficiently a number of HOFs. Briefly, it consists of a *farmer* processor that distributes packets of independent data to identical *worker* processors doing some computation on the data before returning the results back to the farmer (fig 2). The farm is typically used for processing a stream of data elements where a function is to be applied to each element.

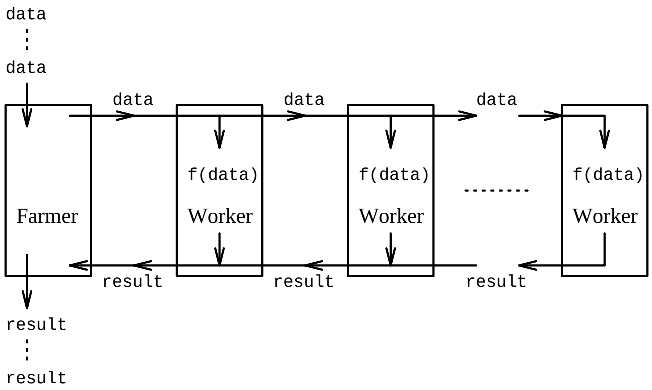


Fig 2: Linear processor farm

An attractive characteristic of the farm is that it is capable of balancing automatically the load evenly between the workers if the number of packets to be processed is sufficiently high. A large number of small packets is however undesirable as the cost of communication relative to computation will reduce the farm's performance. The packet size can be controlled by grouping the data to be processed together into *chunks* of appropriate size and treating each chunk as a packet. Details of performance modeling of farms are given in [11, 16, 21], showing that the farm's performance is determined mainly by two factors:

1. The time taken for interprocessor communication of data and results to and from the workers, dictated by the hardware characteristics of the communication network.
2. The time taken by a worker to process a data element, dictated by the hardware characteristics of the processors.

For determining communication time we need the functions

- $\mathsf{DistCost(n)}$ which gives the time for interprocessor distribution of n data bytes.
- $\mathsf{CollCost(n)}$ which gives the time for interprocessor collection of n result bytes.

These are both given by the farm's performance model in $\mathcal{P}$. The main problem is to determine the time needed for one worker to process a random data element. We will assume that the worker computes a function `f` on the data element, and that the computation takes time $\tau_{\mathtt{f}}$. Knowing the interprocessor communication times and $\tau_{\mathtt{f}}$, the performance model of the farm can be used to determine the optimal number of workers.

$\tau_{\mathtt{f}}$ is generally not constant for all data elements to be processed, but as the farm tends to balance its load automatically for carefully chosen packet sizes, the average $\tau_{\mathtt{f}}$ is often sufficient to yield precise performance estimates. The information needed about $\tau_{\mathtt{f}}$ is as suggested in [26]:

- *avr*, the average $\tau_{\mathtt{f}}$; indicates the number of workers most likely to be usefully employed.
- *min*, the lowest $\tau_{\mathtt{f}}$; guarantees that there is at least enough parallelism present to keep a minimum number of workers busy.
- *max*, the maximal $\tau_{\mathtt{f}}$; guarantees that there will never be more parallelism present than for keeping a certain maximal number of workers busy.
- *SD*, the standard deviation of $\tau_{\mathtt{f}}$ from *avr*. A low *SD* indicates that *avr* is a good indication of the level of useful parallelism. A high *SD* indicates that it may be beneficial to cluster data elements into chunks to get a good load balancing in the farm.

and in addition:

- *dist*, a notion of how *avr* varies over subparts of the input stream.
- *vol*, the amount of data to be processed. Indicates which size the chunks should have to ensure a proper load balancing.

For acquiring this information we will consider three strategies:

1. **Static program analysis**
   Methods for (automated) program complexity analysis, for example [6, 15, 26], of the function `f` succeed in deriving a constant $\tau_{\mathtt{f}}$ if `f` is relatively simple. The presence of conditional branching and recursion will however generally require assumptions to be made about the distribution of argument values in order to derive values for *avr* and *SD*. Note also that even if the time to execute some paths through `f` is found to be constant, `f`'s execution time in the *general* case must reflect the possible dependencies between `f`'s arguments and execution times for other paths. Hence, for realistic `f`'s, the best we can hope for from this kind of analysis is a time function `Tf` giving the predicted execution time of `f` for a given argument value. Values for *dist* and *vol* can only be found by analysing the input data, and can thus not be derived with static program analysis.
2. **Dynamic program analysis (profiling)**
   The success of this approach hinges on the hypothesis that there exists a typical set of input data to `f`, and that the main characteristics of $\tau_{\mathtt{f}}$ are similar for all sets of input data. If this holds, results from profiling runs can be used for producing values for *avr*, *min*, *max*, *SD*, *dist* and *vol* of $\tau_{\mathtt{f}}$ to be used for compilation.
3. **Programmer annotations**
   It is unlikely that the programmer can give even reasonably accurate estimates of *min*, *max* or *avr*, as this requires detailed knowledge about execution times of low level primitives on the target architecture. It may however, be possible to indicate if *SD* is low or high, or say something about *dist* or *vol*.

We conclude that it is only in rare cases that a purely static program analysis alone can derive the desired properties about $\tau_{\mathtt{f}}$. Knowledge about the distribution of `f`'s arguments is generally required, and it seems most convenient to abstract this information from the data itself by profiling.

In the following we will use empirical performance models of farms implemented on a Meiko Computing Surface with the workers arranged in a linear chain. The performance models are obtained by measuring the execution times of thousands of test runs with variable computation times $\tau_{\mathtt{f}}$ and variable numbers of workers. Communication is restricted to data elements consisting of single integers and small tuples of integers, grouped in chunks of 100 elements.

Absolute speedup and efficiency relative to the number of workers in the farms are calculated by

$$\begin{aligned} speedup &= \frac{sequential\ run\ time}{parallel\ run\ time} \\ efficiency &= \frac{speedup}{number\ of\ workers} \end{aligned}$$

We will here assume that the object size of the elements can be determined at compile time by a function

$\mathsf{Sz}[\alpha]$,

which gives the size of the ML type α in bytes. The processing of elements whose sizes are not known at compile time, such as lists and strings, may require additional profiling information.

3.2 Process Pipelines

A process pipeline consists of a linear chain of processes with a behaviour similar to that of a factory assembly line. The first process in the chain computes some function on its input data and sends the result as input data to its succeeding process which again passes its output to its successor and so on. A process pipeline is the parallel equivalent of a function composition, where the result of one function is consumed as input of another, whose result may again be used as the argument of a third etc.

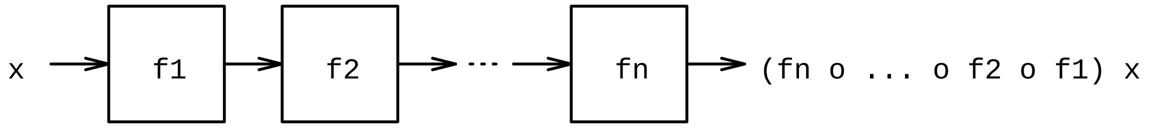


Fig 3: Pipeline as function composition

Parallelism can be exploited when the processes are computed on separate processors and several data elements flow through the pipeline, allowing all processors to be computing simultaneously on successive elements of data. The pace with which the pipeline can process data is limited by the processor taking the longest time to complete its computation. To exploit the resources efficiently, the processes of the pipeline should be mapped onto processors such that the processors have a roughly equal load of computation and communication [23].

3.3 The `map` HOF

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

The `map` HOF applies its argument function `f` to each element of its argument list and has a direct implementation on a processor farm by letting each worker processor compute `f` on disjunct elements of the list. This is illustrated in fig 4, where α's and β's represent objects of type α and β respectively, and `f` is a function of type `fn:` $\alpha \rightarrow \beta$.

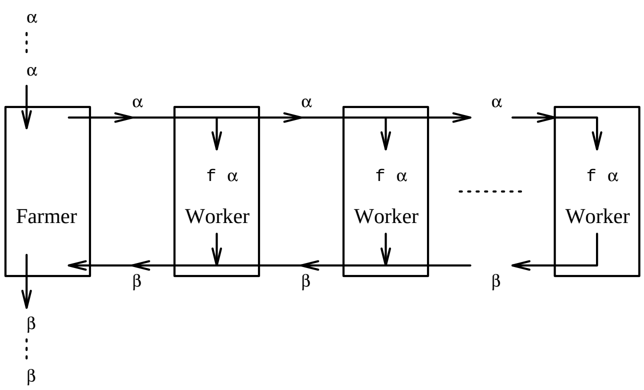


Fig 4: `map` *on a linear processor farm with* `f = fn:` $\alpha \rightarrow \beta$

Communication times are derived by inspecting the types of the arguments and result of `f`, i.e. if `f` is `fn:` $\alpha \rightarrow \beta$ the elementary cost of distributing `f`'s argument is

DistCost(Sz[α]),

while the cost for collecting the result is

CollCost(Sz[β]).

As the total number of processors available to the whole program is generally a globally restricted resource, the optimal number of workers in a farm is generally dependent on the resources allocated to the other parts of the program. We would therefore like to derive the relationship between local performance in a pattern and the number of processors employed locally and later use this for balancing the load harmoniously between all parts of the program. Fig 5 shows how the farm's performance is affected by $\tau_{\mathtt{f}}$ and the number of worker processors for `f = fn:int→int` [2]. For low values of $\tau_{\mathtt{f}}$ there is not enough useful parallelism present to keep more than a few workers fully occupied.

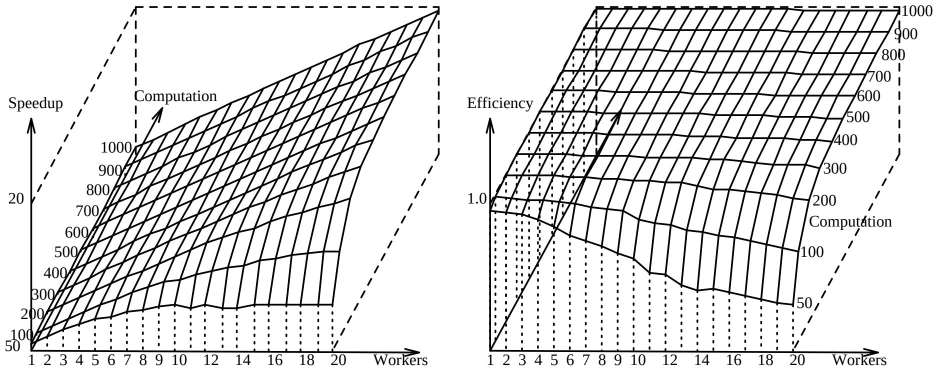


Fig 5: Performance figures of `map f` *for* `f = fn:int→int`

3.4 The `mapn` HOFs

```
fun map2 f [] []              = [] |
    map2 f (h1::t1) (h2::t2) = f h1 h2 :: map2 f t1 t2;

fun mapn f [] [] ... []                   = [] |
    mapn f (h1::t1) (h2::t2) ... (hn::tn) = f h1 h2 ... hn :: mapn f t1 t2 ... tn;
```

(The definition of `mapn` is given in a pseudosyntax of ML. Instantiate with $\mathtt{n} \geq 1$)

The `mapn` family of functions, i.e. map2, map3, etc, can be seen as a generalization of `map`, as `map1` and `map` are identical. `mapn` applies its `n`-ary argument function `f` to corresponding elements from each of the `n` argument lists with a single list as the result. From the point of view of parallel implementation, the only difference between `map` and `mapn` for $\mathtt{n} > 1$ is that for the latter the number of arguments to `f` is higher. This leads to increased communication time for distributing the arguments, i.e. for `f = fn:` $\alpha_1 \rightarrow \alpha_2 \rightarrow ... \rightarrow \alpha_n \rightarrow \beta$, the distribution cost is

DistCost(Sz[α_1]+Sz[α_2]+...+Sz[α_n]),

as all arguments can be passed out in one message. In fig 6, α's thru δ's represent objects of type α thru δ respectively, and `f` is a function of type `fn:` $\alpha \rightarrow \beta \rightarrow \gamma \rightarrow \delta$.

[2]The 'Computation' axis shows $\tau_{\mathtt{f}}$ on a linear scale where 1000 is equivalent to 762 microseconds (same for fig 7,8,10 and 11).

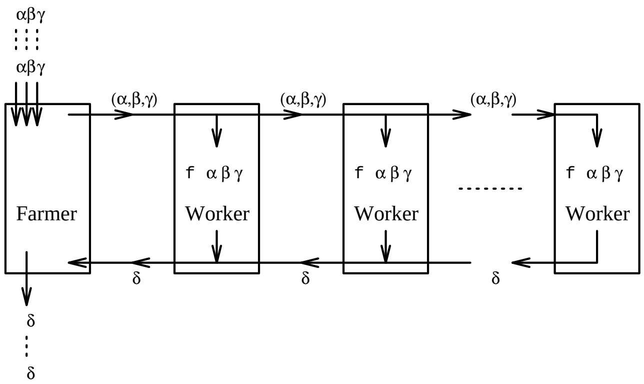
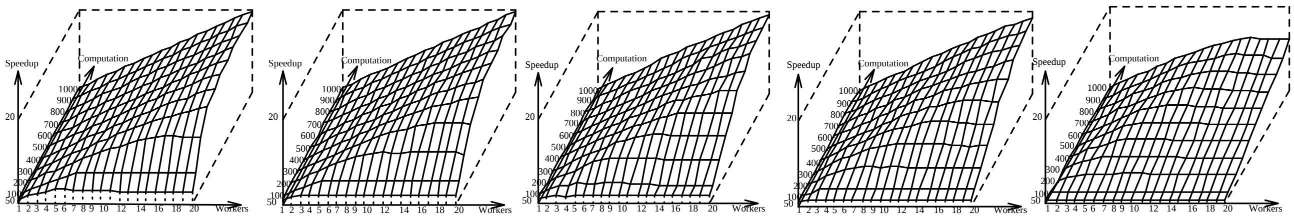


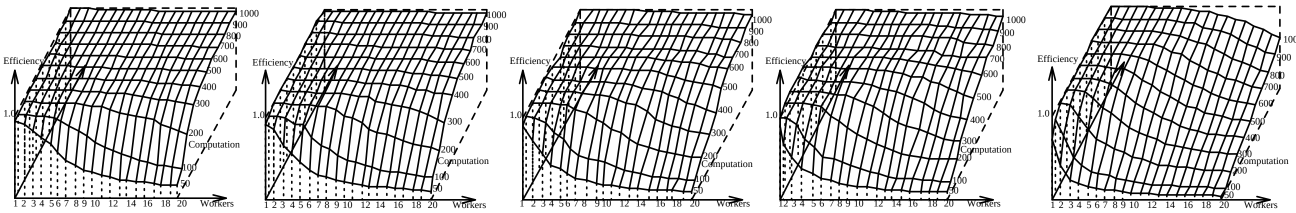
Fig 6: `map3` *on a linear processor farm with* `f = fn:` $\alpha \to \beta \to \gamma \to \delta$

Fig 7 and 8 illustrate how increased communication time (i.e. increased value of `n`) affects the performance of the farm for `f = fn:int→int → ... →int→int` (a total of `n`+1 `int`'s), varying level of computation in $\tau_{\texttt{f}}$ and varying number of worker processors. The higher the value of `n`, the higher the communication cost is, and the worse the farm performs.



`map2` `map3` `map4` `map5` `map10`

Fig 7: Speedup figures for `mapn f` *for* `n` = *2,3,4,5,10*



`map2` `map3` `map4` `map5` `map10`

Fig 8: Efficiency figures for `mapn f` *for* `n` = *2,3,4,5,10*

3.5 The `filter` HOF

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

`filter` applies the argument function `f` to all elements of the argument list, retaining only those elements for which `f` gives the value `true`. A straightforward farm implementation of `filter` is to feed the elements of the argument list into the farm, and have the workers return *only those* elements for which `f` evaluates to `true`. Fig 9 shows the flow of data elements of type α through the farm with this implementation.

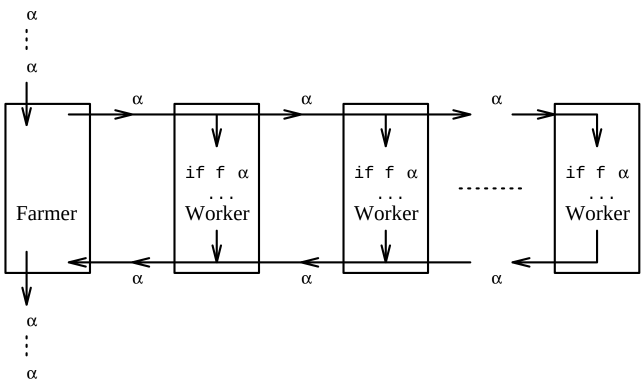


Fig 9: `filter` *on a linear processor farm with* `f = fn:` $\alpha \rightarrow$`bool`

This gives a reduced cost of communication for collecting the results of the farm, when compared to the farm implementation of `map`. Assume an argument function `f = fn:` $\alpha \rightarrow$`bool` and a probability ϱ of `f` evaluating to `true`. The reduced number of elements to be returned results in either fewer packets to be sent or, as implemented, packets of reduced size. In both cases the reduction is with a factor of ϱ. For $\varrho = 1.0$ the performance of `filter` is as for `map`, but for lower values of ϱ, `filter` has less communication, and hence a better performance. Fig 10 and 11 show a slightly improved performance (around 10% at best) from ϱ=50% to ϱ=1% for `f = fn:int`$\rightarrow$`bool`.

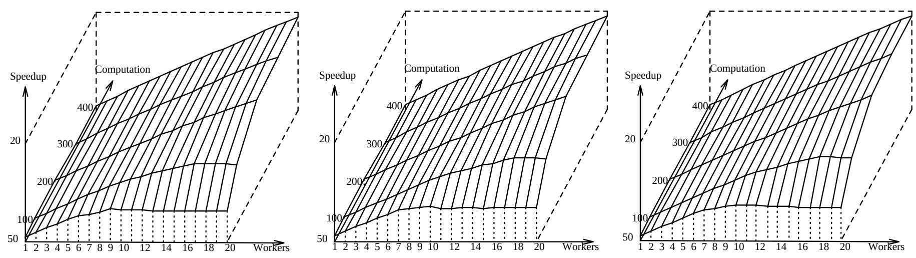


Fig 10: Speedup figures for `filter f` *for* ϱ *=50%, 10%, 1% and* `f = fn:int`$\rightarrow$`bool`

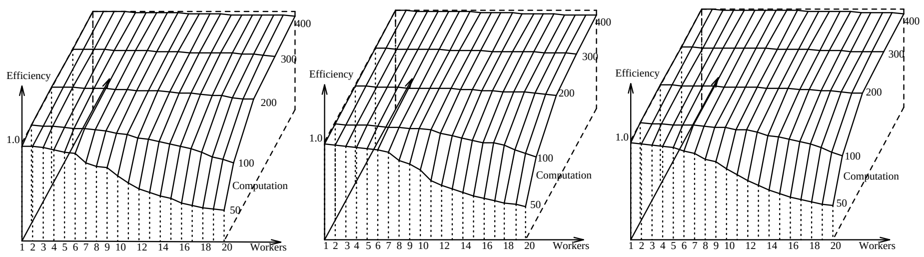


Fig 11: Efficiency figures for `filter f` *for* ϱ *=50%, 10%, 1% and* `f = fn:int`$\rightarrow$`bool`

4 Example

The following simple example aims to illustrate how a functional program (in SML) using the HOFs discussed previously might be compiled into parallel code (in occam2). For performance prediction, empirical performance models as presented in section 3 are used, and all time information is collected solely from profiling.

Assume that for a given function f we want to compute the integer value A given by:

$$A = \lfloor \int_a^b f(x)dx \rfloor$$

for a given function f, and large number of integers a and b. Further, we are only interested in those values of A that can be expressed as the sum of a multiple of a fixed number of integers, i.e.

$$A = x_1 c_1 + x_2 c_2 + x_3 c_3$$

for integer constants c_1, c_2 and c_3 and positive integers x_1, x_2 and x_3.

This problem can be solved with the following ML program (with types in comments) :

```
fun integral_of_f a b = ...  ;
      (* fn :  int → int → int *)

fun subset_sum A = ...  ;
      (* fn :  int → bool *)

val integral = map2 integral_of_f;
      (* fn :  int list → int list → int list *)

val subset = filter subset_sum;
      (* fn :  int list → int list *)

fun prog as bs = subset (integral as bs);
      (* fn :  int list → int list → int list *)
```

We will now demonstrate how this can be compiled into a parallel program. From the previous section we assume $\mathcal{H}$ = {map, map2, filter} and a set $\mathcal{P}$ with the corresponding farm patterns and performance models.

Step 1: HOF Identification

The $\mathcal{H}$-HOFs map2 and filter are identified by their explicit appearance in the program. Noting that prog consists of a composition of filter subset_sum and map2 integral_of_f, the program structure is:

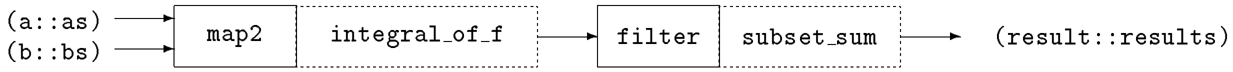


Figure 12: *Program Structure*

where hof f represents the HOF hof partially applied to the function f.

Step 2: Determine Useful Parallelism

The program's input are two integer lists as and bs. The length of each list is 100000. This gives (from section 3.1) *vol* = 100000 easing the task of load balancing. The value distribution within each list is not known a priori, but is assumed to be constant throughout the whole lists, i.e. *dist* is negligible. As we do not yet have the tools at hand to analyse the program statically, we will rely on profiling based on 10% samples of the lists, and assume that the results are representative for the results obtained when running with the full lists. The farms will process the lists in packets of 100 elements, and we will trust that this is a sufficiently large packet size to assume *avr* as the correct values to use for $\tau_{\texttt{integral\_of\_f}}$ and $\tau_{\texttt{subset\_sum}}$. The values for *min*, *max* and *SD* are also produced by profiling. Given the large packet size and the large amount of data to be processed, their values are not needed for this example, and are presented in tables 1 and 2 for the sake of completeness only . Normally we would like to profile the SML code and then derive the corresponding times for the equivalent occam2 code. Provided that the translation process from SML to occam2 is well understood, accurate figures should be obtainable. Work is still in progress on this, so for the current example we do all profiling directly on the occam2 code.

`map2` applied to `integral_of_f`

The cost of communication is derived from the type of `integral_of_f`; two integers as arguments, one integer as result. Profiling gives the following information about $\tau_{\texttt{integral\_of\_f}}$:

min	max	avr	SD
82	1271	503	326

Table 1: Profiling figures for `integral_of_f` *(microseconds per invocation)*

We note in passing that according to the performance model of `map2`, a farm implementation of `map2 integral_of_f` should give near linear speedup with predicted efficiency 0.942 for 20 workers.

`filter` applied to `subset_sum`

The cost of communication is derived from the type of `subset_sum`; one integers as argument, one integer as result. The required information about $\tau_{\texttt{subset\_sum}}$ is again found by profiling:

min	max	avr	SD
28	3504	664	793

Table 2: Profiling figures for `subset_sum` *(microseconds per invocation)*

In the profiling runs `subset_sum` evaluated to `true` in 9.52% of the cases. We approximate this by using the performance model of `filter` in which $\varrho = 10\%$. We note that according to this performance model, a farm implementation of `filter subset_sum` should give near linear speedup with predicted efficiency 0.989 for 20 workers.

Step 3: Mapping and Load Balancing

As the functional program is a function composition of two HOFs with farms as parallel equivalents, the equivalent parallel program is a pipeline of two farm processes. By carefully distributing the number of available worker processors between the two farms it is the goal to keep the pipeline balanced. For determining the optimal number of workers, we look at the performance models of the two farms given the communication and computation costs as derived above, and compare how the performances vary with the number of workers allocated. Table 3 shows that for this program, the rates of data processing in the farms should be equal when the ratio <workers in `map2` farm>:<workers in `filter` farm> is in the range 0.76 - 0.79.

Workers in `map2` farm	No of elements processed per sec	Required workers in `filter` farm	Ratio between workers
1	1974	1.31	0.76
2	3938	2.62	0.76
6	11685	7.79	0.77
8	15494	10.35	0.77
12	22987	15.40	0.78
16	30307	20.35	0.79

Table 3: Number of workers needed for equal processing power in the farms

An approximation of this (0.75) was used in two parallel implementations of the program. In the first, 14 processors were split into 6 workers (for `map2`) vs 8 workers (for `filter`) in the two farms. In the second, 28 processors were allocated as 12 vs 16 workers respectively.

Step 4: Code Production

The ML code of `integral_of_f` and `subset_sum` were translated into sequential occam2 by hand (and used for the profiling in step 2). The pieces of parallel occam2 code for `map2` and `filter` were instantiated with the sequential code and with communication protocols for supporting `integral_of_f` and `subset_sum` respectively.

4.1 Performance

As seen in table 4 below, both implemented configurations give a satisfactory speedup. The predicted parallel performance was derived from the performance models, and does not deviate much from what was actually measured.

Config	Measured Sequential	Measured Parallel	Speedup	Predicted Parallel	Deviation
6+8	116.76	8.64	13.51	8.56	1.41%
12+16	116.76	4.45	26.24	4.35	2.30%

Table 4: Performance figures for example program (seconds)

5 Conclusions and Future Work

As demonstrated, the exploitation of implicit parallelism in higher order functions may indeed result in parallel programs yielding acceptable performances. There are however two points of concern. First, whether the process of compilation can be automated without significant loss in performance, and secondly whether the method is powerful enough to support general implicit parallel functional programming. Not much experience with this form of parallelization exists, and it is hard at present to draw further conclusions.

A number of improvements can be made to the approach taken in the example. More HOFs should be supported with more precise and more general performance models. The process of determining useful parallelism should be based on automatic complexity analysis as well as profiling, in order to give better performance predictions and guarantees. The profiling should be of the SML code rather than the occam2 code, with a high degree of accuracy in prediction of occam2 execution times. Further, the whole system should be automated and be dependent on no more than at most *guidance* from the programmer. Work on a semi-automated system based on this is in progress.

Acknowledgements

Thanks are due to Greg Michaelson, David Busvine and Kevin Waugh for reading and commenting on early drafts of this paper. The author is also grateful to Heriot-Watt University for financial support.

References

[1] John Backus. Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. *Communications of the ACM*, 21(8):613–641, August 1978. 1977 ACM Turing Award Lecture.

[2] F Baiardi, M Danelutto, R Di Meglio, M Jazayeri, M Mackey, S Pelagatti, F Petrini, T Sullivan, and M Vanneschi. Pisa Parallel Processing Project on general-purpose highly-parallel computers. Technical Report HPL-PSC-91-23, Hewlett-Packard Laboratories, Pisa Science Center, Pisa, Italy, May 1991. Presented at the Fifteenth Annual International Computer Software And Applications Conference, September 11-13, 1991, Tokyo, Japan.

[3] 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.

[4] David Busvine. Transformation Techniques for Functional Programs. Technical Report TR 90/3, Department of Computer Science, Heriot-Watt University, Edinburgh, July 1990.

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

[6] Charles Cohen. Computer-Assisted Microanalysis of Programs. *Communications of the ACM*, 25(10):724–733, October 1982.

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

[8] Marco Danelutto, Roberto Di Meglio, Salvatore Orlando, Susanna Pelagatti, and Marco Vanneschi. The P^3L language: an introduction. Technical Report HPL-PSC-91-23, Hewlett-Packard Laboratories, Pisa Science Center, Pisa, Italy, December 1991. To appear in Future Generation Computer Systems, North Holland, as 'A methodology for the development and the support of massively parallel programs'.

[9] Marco Danelutto, Roberto Di Meglio, Susanna Pelagatti, and Marco Vanneschi. High level language constructs for massively parallel computing. In M Baray and B Özgüç, editors, *Computer and Information Sciences IV*, pages 777–788. Elsevier Science Publishers B.V., 1991.

[10] 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 Hugh Glaser and Pieter Hartel, editors, *Proceedings of the Workshop on the Parallel Implementation of Functional Languages*, pages 31–51. Department of Electronics and Computer Science, University of Southampton, 1991. Technical Report Series CSTR 91-07.

[11] Anthony J G Hey. Experiments in MIMD Parallelism. In E Odijk, M Rem, and J-C Syre, editors, *PARLE'89 Parallel Architectures and Languages Europe, Vol II*, pages 28–41. Springer-Verlag LNCS 366, 1989.

[12] R J M Hughes. Why Functional Programming Matters. Technical Report 16, Programming Methodology Group, University of Göteborg and Chalmers Institute of Technology, November 1984.

[13] H T Kung. Computational models for parallel computers. In R J Elliott and C A R Hoare, editors, *Scientific Applications of Multiprocessors*, pages 1–15. Prentice-Hall, 1989.

[14] Gyula Magó and David Middleton. The FFP Machine - A Progress Report. In *Proceedings of the International Workshop on High-Level Language Computer Architecture*, pages 5.13–5.25. University of Maryland, May 1984.

[15] Daniel Le Métayer. ACE: An Automatic Complexity Evaluator. *ACM Transactions on Programming Languages and Systems*, 10(2):248–266, April 1988.

[16] David J Pritchard. Mathematical Models of Distributed Computation. In Traian Muntean, editor, *Parallel Programming of Transputer Based Machines, Proceedings of the 7th occam User Group Technical Meeting, Grenoble, France 1987*, pages 25–36. IOS Press, 1987.

[17] Hideki John Reekie. Towards Effective Programming for Parallel Digital Signal Processing. Research Report 92.1, Key Centre for Advanced Computing Sciences, University of Technology, Sydney, May 1992.

[18] D B Skillicorn. Models for Practical Parallel Computation. *International Journal of Parallel Programming*, 20(2):133–158, April 1991.

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

[20] Douglas R Smith and Michael R Lowry. Algorithm Theories and Design Tactics. In *Mathematics of Program Construction*, pages 379–398. Springer-Verlag LNCS 375, June 1989.

[21] R W S Tregidgo and A C Downton. Processor Farm Analysis and Simulation for Embedded Parallel Processing Systems. In Stephen J Turner, editor, *Tools and Techniques for Transputer Applications, Proceedings of the 12th occam User Group Technical Meeting*, pages 179–189. IOS Press, 1990.

[22] Clifford Walinsky and Deb Banerjee. Compiling FP for Data-Parallel Systems (Extended Abstract). In *Proceedings of the Workshop of (symbolic) Languages for Parallel Computers, San Diego, CA*, pages 114–128. Argonne National Laboratory, November 1991. ANL-91/34.

[23] Kevin Waugh. An algorithm to balance a pipeline of processes. Technical Report 92011, Department of Computer Science, Heriot-Watt University, Edinburgh, July 1992.

[24] 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.

[25] Kevin Waugh and Greg Michaelson. Parallel Process Placement from Functional Prototype Instrumentation. In *IEEE TENCON '91 pre conference proceedings - volume II*, pages 279–282. IEEE, 1991.

[26] Ben Wegbreit. Mechanical Program Analysis. *Communications of the ACM*, 18(9):528–539, September 1975.