

# Structured Parallel Programming

J. Darlington, M. Ghanem, H. W. To  
Department of Computing  
Imperial College  
London, SW7  
email: {jd, mmg, hwt}@doc.ic.ac.uk

## Abstract

*Parallel programming is a difficult task involving many complex issues such as resource allocation, and process coordination. We propose a solution to this problem based on the use of a repertoire of parallel algorithmic forms, known as skeletons. The use of skeletons enables the meaning of a parallel program to be separated from its behaviour. Central to this methodology is the use of transformations and performance models. Transformations provide portability and implementations choices, whilst performance models guide the choices by providing predictions of execution time. In this paper we describe the methodology and investigate the use and construction of performance models by studying an example.*

## 1 Introduction

Parallel machines have impressive computation to cost ratios, offering performance higher than that possible with sequential machines. Yet there are few commercial applications written for them. This is a reflection on the problems of writing parallel software, which is felt to be more difficult to produce than sequential software. Sequential machines have a single computational model, the von-Neumann model. In comparison there are many different parallel architectures each with their own computational model. This makes **portability** and **predictability of performance** much more of a problem than in the sequential case.

The von-Neumann model provides for sequential languages a simple relationship between language constructs and machine implementation. Run-time resource allocations, such as memory allocation, can be resolved by the compiler with no great performance implications. Thus the programmer can reason about the behaviour and predict the performance of their

program at the language level. This is not the case for parallel programming where mapping a parallel program to a multiprocessor machine is a complex task involving many decisions about task allocation, scheduling of competing processes' communication patterns, etc. Usually the only way to achieve the desired performance is through explicit control, which adds to the complexity of the program.

The universality of the von-Neumann model guarantees portability of sequential programs at the language level with no danger of unforeseen degradation in performance. With the explicit task allocation of parallel programs there is rarely any portability. Even when portability is possible through the use of high level programming languages, which can be compiled to other machines, it is difficult to predict or maintain performance.

Given the success of the von-Neumann model for sequential computing the conventional solution to the parallel software dilemma is to attempt to produce a 'parallel von-Neumann' model. The rôle of such an abstract machine model would be to provide an abstraction of the execution behaviour of all the target parallel machines. Programming languages would then be defined on this model, with programs written to organise the machine's activities to produce correct answers and behave efficiently. The need to express both correctness and efficiency in the same framework requires that there is a close correlation between the abstract model and the concrete architectures. So even if such a model is found, and there are several candidates [9, 20], programming will still remain a fairly low level and laborious activity. Critically, in the conventional programming language framework, the meaning and behaviour of an application is still required to be expressed via a single program. It is not possible to specify one without effecting the other.

Another approach is to exploit the **implicit parallelism** of functional languages. These languages provide a much higher degree of abstraction than their im-

perative counterparts. Using this implicit parallelism would remove the requirement to explicitly decompose the problem into concurrent tasks, and to control synchronisation and communication between tasks. Unfortunately, automatic exploitation of this has not yet been achieved. This is because the problems of automatic resource allocation have still to be overcome. Compilers can generate too much fine grain parallelism which overwhelms resources and is too costly to support. It is difficult to automatically produce load balanced code given the general nature of the problem. Even though the parallelism is implicit, it can be difficult to detect and is dependent on the programmer expressing it in a way which reveals the natural parallelism.

However, functional languages do have some important properties including a clean separation of behaviour from meaning (Church-Rosser property), the ability to apply correctness preserving transformations, and higher order functions, which provide a powerful control abstraction mechanism. In the next section we consider another solution which exploits the advantages of functional languages, but in a new, hopefully more practical, manner.

## 2 The skeletons approach

Both previous solutions to parallel programming result in the meaning and behaviour of a program being entwined. With explicit parallelism, through a uniform abstract machine, the two are expressed in the same language. With implicit parallelism, the behaviour is inferred from the programming language with no user control. Successful parallel programming requires that both meaning and behaviour are expressed. However, the two are orthogonal, programs should be written to be correct, then tuned to be efficient. This implies a separation of meaning and behaviour.

To achieve the separation we propose the use of a range of algorithmic forms, known as **skeletons**, which abstract useful parallel computational structures from applications. Applications are then constructed as instantiations of these algorithmic forms. The approach follows that taken by Cole [5], for imperative languages, and Backus's idea [1] of programming functional languages through a fixed set of operators.

Our skeletons are expressed in a functional language as higher order functions that can be defined in the language. The skeleton's declarative meaning is then established by this function definition. This meaning is independent of any implementation issues

and hence any behavioural constraints. These definitions also provide sequential prototypes which are portable across different machines. A skeleton's behaviour is defined by its implementation on a particular parallel machine. So the behaviour of a skeleton can vary from machine to machine, as one would expect. The only parallelism in a program arises from the use of skeletons. All other functions are executed sequentially. Thus, the parallel behaviour of a program is given by the behaviour of its constituent skeletons. All aspects of a skeleton's behaviour, such as process placement and interconnectivity are documented with the skeleton's implementation, along with optional behaviours which can be specified at compile or run time. There is no reason why each skeleton could not be implemented on every type of architecture. However, different skeletons are more naturally implemented on particular groups of architectures. It would seem wise to implement skeletons only on those architectures that were suitable. Thus the range of skeletons make up the programmer's target abstract machine model, but one that is expressed at a much higher level and is capable of accommodating a much greater variety of machines and behaviours more easily than a single abstract machine model.

Using transformation it is possible to derive equivalences between skeletons. Since we have a fixed set of skeletons, the possible equivalences can be derived mechanically once and stored for future use. These relationships between skeletons provide portability. An application expressed naturally in one skeleton may find that there are no implementations for that skeleton on a particular machine. Choosing the right equivalences will map the application onto the desired machine. Once that particular skeleton-machine pair has been selected then further transformations can be used to optimise the skeleton for the particular machine. An example of a common transformation would be to vary the grain size through partial evaluation [7]. Through transformation we have effectively preserved meaning while providing alternative implementations and behaviours.

For each skeleton-machine pair we can construct an associated performance model. These performance models are formulae which predict the execution time and whose variables are machine and problem parameters. By developing these models it is possible to make choices between equivalent skeletons for a given machine. Within a particular skeleton and machine pair the performance models guide the selection of optimal behaviours.

The goal of this collected work is to replace the pro-

cess of producing solutions by creating programs from scratch with the development of programs through selection and instantiation of a fixed range of alternatives. Thus there is an explicit identification of all the decisions that have to be addressed to produce an efficient mapping of an application onto a machine rather than having these decisions made implicitly by writing a program with the desired properties.

In this paper we introduce some basic skeletons and then investigate the performance model aspects of the work. Examples programs written using skeletons can be found in [6] along with a discussion of transformation technology. The consequences of using skeletons at the application level are shown in [8].

### 3 Examples of skeletons

To aid our discussion of performance models some sample skeletons are described. They comprise some of the basic parallel algorithmic forms but are by no means exhaustive. The definitions are expressed in Haskell [10] using some standard primitive functions which can be found in [2].

Simple linear process parallelism is captured by the PIPE skeleton. A list of functions are composed together so that elements can be streamed through them. Parallelism is achieved by allocating each function to a different processor. Note that this idea can easily be extended to higher dimensions.

$$\begin{aligned} \text{PIPE} &:: [[\alpha] \rightarrow [\alpha]] \rightarrow [\alpha] \rightarrow [\alpha] \\ \text{PIPE} &= \text{foldr1 } (\circ) \end{aligned}$$

The FARM skeleton captures the simplest form of data-parallelism. A function is applied to each of a list of ‘jobs’. The function also takes an environment, which represents data which is common to all of the jobs. Parallelism is achieved by utilising multiple processors to evaluate the jobs (i.e. ‘farming them out’ to multiple processors).

$$\begin{aligned} \text{FARM} &:: (\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow \alpha \rightarrow ([\beta] \rightarrow [\gamma]) \\ \text{FARM } f \text{ env} &= \text{map } (f \text{ env}) \end{aligned}$$

Many algorithms work by splitting a large task into several sub-tasks, solving the sub-tasks independently, and combining the results. This approach is known as *divide-and-conquer* and it is captured by the DC skeleton. Trivial tasks (t) are solved (s) directly on the home processor: larger tasks are divided (d) into sub-tasks and the sub-tasks passed to other processors to be solved recursively. The sub-results are then combined (c) to produce the main result.

$$\begin{aligned} \text{DC} &:: (\alpha \rightarrow \text{Bool}) \rightarrow (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow [\alpha]) \rightarrow \\ & \quad ([\beta] \rightarrow \beta) \rightarrow \alpha \rightarrow \beta \\ \text{DC } t \text{ s } d \text{ c } x & \\ & \quad | t \ x \quad \quad = s \ x \\ & \quad | \text{not } (t \ x) = (c \circ \text{map } (\text{DC } t \text{ s } d \text{ c}) \circ d) \ x \end{aligned}$$

Another common class of algorithms describes systems where we have two lists of objects and every element in the first list can interact with every element in the second list. Usually, we are interested in the combined results of these interactions for each element in the first list. This is described by the RaMP skeleton (‘Reduce-and-Map-over-Pairs’). This skeleton is typically used for initial specification and implemented by transformation to an alternative, more machine oriented, form, for example by farming out the calculations for each object or by pipelining over the functions  $f$  and  $g$ .

$$\begin{aligned} \text{RaMP} &:: (\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow (\gamma \rightarrow \gamma \rightarrow \gamma) \rightarrow \\ & \quad [\alpha] \rightarrow [\beta] \rightarrow [\gamma] \\ \text{RaMP } f \text{ g } \text{ as } \text{ bs} &= \\ & \quad \text{map } h \text{ as} \\ & \quad \text{where } h \ x = \text{foldr1 } g \ ( \text{map } (f \ x) \ \text{bs} ) \end{aligned}$$

### 4 Performance models

Programming parallel machines has produced a new phenomena known as *performance bugs*. Parallel machines are used to provide high performance, so it is vital for programs to use the resources efficiently. Yet often parallel programs give unexpected performance, due to the unpredictable result of a particular resource allocation strategy for a given architecture. This situation is exacerbated when porting to different architectures where an explicit strategy that performed well previously can give widely differing results. The usual cure is to perform a process known as performance debugging. A programmer repeatedly executes a program and observes the results. Using the results as a guide the resource allocation decisions are adjusted in an attempt to improve the performance.

We propose an alternative approach that attempts to predict the performance of particular allocation strategies and in that way guide the choice of resource usage and implementation. Generating correct performance models for arbitrary parallel programs is difficult and time consuming. However the use of performance models is simplified when they are restricted to known program forms such as the skeletons. In these cases, the process of producing and verifying performance models for a skeleton and machine pair needs

only to be performed once. Studying the performance properties of restricted forms of parallel computation has been studied recently by several authors [11], [18].

The performance model associated with a skeleton is a function parameterised on the problem and machine characteristics and will return a prediction of the performance. The performance prediction is usually given as an estimated total time to execute a specific problem. Examples of the machine characteristics of interest are the communication time between processors, the time for a floating point operation and the number of processors dedicated to the skeleton. These figures can be calculated by benchmarking using the same compiler and runtime environment used by the skeletons. The problem parameters are measured by a mixture of static program analysis and profiling using typical data values. Further accuracy can be achieved by annotations provided by the programmer.

To introduce the ideas we present general performance models for three of the skeletons. A performance model is given for each, but the machine parameters have been left uninstantiated. Section 5 then describes a specific instantiation of the models.

#### 4.1 A divide-and-conquer example

A performance model for the DC skeletons needs to take into account the complexity of each of the arguments passed to DC and the communication time between stages of the algorithm. In the simplest case it is assumed that the instance of the DC generates a balanced binary tree and that the complexity of any argument is determined solely by its parameter size. The model is given in terms of the time to solve a problem of size  $x$ ,  $t_{sol_x}$ , the time taken to divide a problem of size  $x$ ,  $t_{div_x}$ , the time to combine the two results,  $t_{comb_x}$ , and the communication time to a child of the current stage,  $t_{comm_x}$ . There is also a set up time associated with each stage for spawning a child,  $t_{setup_x}$ , with a static process network this time will be negligible. The solution time can then be expressed as the following recursive equation.

$$t_{sol_x} = t_{div_x} + 2t_{setup_x} + t_{sol_{x/2}} + t_{comb_x} + 2t_{comm_x}$$

Sequential communication has been assumed, but this can be refined depending on the hardware. The base case equation occurs when  $x$  becomes a trivial task, and  $t_{sol_x}$  is the sequential time to solve a trivial task. This can be expressed in a similar fashion but there will be no overhead of parallelism and the children will not run concurrently.

$$t_{seq_x} = t_{div_x} + 2t_{seq_{x/2}} + t_{comb_x}$$

The above analysis assumed an unbounded number of processors with which to solve the problem. In general this will not be the case. Thus it would be necessary to predict the performance given  $p$  processors. Assuming that these processors are organised into a balanced binary tree we can predict the execution time as

$$t_{sol_x} = \sum_{i=0}^{\log(p+1)-2} (t_{div_{x/2^i}} + 2t_{setup_{x/2^i}} + t_{comb_{x/2^i}} + 2t_{comm_{x/2^i}}) + t_{seq_{x/2^{\log(p+1)-1}}}$$

This becomes clearer when one sees that the height of a binary balanced tree of  $x$  nodes is  $\log(x+1)-1$ . Another implementation strategy may be to only spawn one child for each recursive call of DC. The other is processed locally. This implies that all the processors will eventually be used as leaves, rather than some being only internal nodes of the tree. Noting that the height for a balanced binary tree of  $x$  leaves is  $\log x$ , for  $p$  processes the performance model is

$$t_{sol_x} = \sum_{i=0}^{\log(p)-1} (t_{div_{x/2^i}} + t_{setup_{x/2^i}} + t_{comb_{x/2^i}} + t_{comm_{x/2^i}}) + t_{seq_{x/2^{\log(p)}}}$$

This type of information allows us to compare the performance of different implementation strategies and to select the optimal case.

A common problem that occurs with divide-and-conquer algorithms is that the trivial case is usually too fine grained to be performed in parallel. This poses the problem of when to stop performing the subtasks in parallel. Issues such as these can use performance models to help make the decisions. For example when  $t_{seq_x} \geq t_{sol_x}$  there is no improvement in doing the call in parallel. The equality arises because even though there is no speed improvement there is a waste of parallel resources. Using this type of information in the preceding equations, it is possible to decide the number of processors to allocate to a DC or whether a given allocation is wasting resources.

#### 4.2 A farm example

An ideal instance of the FARM has a master processor and  $P$  workers. The master has a pool of work packets,  $N$ , that it distributes to the workers. Whenever a worker finishes its work packet it sends the result back to the master and gets a new work packet. In the simple case, all work packets are of equal size and their number will generally be greater than that

of the available workers. The farm thus proceeds in a number of steps, each consisting of a “distribute-work-collect” phase. The number of steps,  $R$ , required to process the work pool is given as

$$R = \lceil \frac{N}{P} \rceil$$

There is also a skeleton startup overhead  $t_s$ , which is the time taken to prepare the farm. For statically allocated farms this time may be very small. However, for a dynamic system the time to create the farm processes and allocate them may be time consuming. The total time required for the farm to complete is given as

$$t_{farm} = t_s + R(t_e + 2t_c)$$

where  $t_e$  is the time required to solve a work packet on a worker and  $t_c$  is the communication time required to send the data or collect a result. The model can be extended to allow for multiple work packets to be sent to a worker in a single step. It can also be refined as more information about the underlying implementation and architecture is known. For example the communication time  $t_c$  can be expressed in terms of the communication startup time  $t_o$  and the time to communicate one byte of data  $t_b$ . Then  $t_c = t_o + t_b b$ .

### 4.3 A pipe example

Similarly, an ideal instance of a PIPE can be viewed as a chain of processes, where there is one process for each function of the pipeline. The list of problems enters, an element at a time, at one end, while a list of answers exit at the other end. In the simplest case, the pipeline is load balanced and there is no memory contention between the processors and the time spent in communication between two stages for a single element is taken as being equal and constant.

The analysis is similar to that of vector processor pipelines [12], with the differing in that the communication time must be accounted for. The execution time of a particular instance of a pipe is dependent on the startup time,  $t_s$ , the execution time of a stage for a single element,  $t_e$ , the communication time between stages,  $t_c$ , the number of stages,  $p$ , and the number of elements in the list,  $n$ . After the initialising period, it can be seen that the execution time is the time taken for the pipe to fill up, plus  $n - 1$  pipe cycles, the time for the rest of the list to pass through the pipe. A pipe cycle is the time taken to solve and communicate one element of the list. The total execution time of the skeleton can be given as

$$t_{pipe} = t_s + (t_e + t_c)(p + n - 1)$$

The ideal pipe assumed that one processor was allocated to each process. However given the limited resources of parallel machines or because of fine grained tasks, fewer processors may be allocated to the skeleton. This change in resource allocation can be viewed as a transformation which composes adjacent functions of the PIPE’s function list. In a working system it is more likely to be expressed as a change in the allocation strategy of the skeleton and not as a source level transformation. However, the transformation demonstrates the correctness of the strategy. Given that  $v$  is the number of virtual stages allocated to a real stage, the performance can be predicted to be

$$t_{pipe} = t_s + (t_e v + t_c)(p + n - 1)$$

In the case of unbalanced pipes one can see that the pipelining rate will be affected by the more computational intensive stages, which form bottlenecks. Other work, [19], has been done on optimally balancing pipelines for a set number of processors. The work is similar in that it generates all possible allocations, then predicts execution times for each allocation, selecting the best.

## 5 A prototype implementation

A prototype implementation of the skeletons was developed on a Meiko Transputer Surface running CS Tools [15]. Programs are written in the functional language Hope+ [17] then are compiled to C and linked with the CS Tools library and a general purpose start-stop garbage collector [3]. A separate copy of the garbage collector executes on each processor to manage its own heap, there is no global heap. Each skeleton employs a master processor and a number of slaves that are allocated dynamically at run time. Thus for the  $P$  processors allocated to a skeleton only  $P - 1$  perform the parallel processing. When only the master is used, the problem is solved sequentially.

The implementations were used to instantiate the machine parameters of the general performance models giving performance models for the skeletons on the Meiko Surface using our implementation. These parameters include the average execution time of instructions on each of the the processors, the number of processors used by a skeleton, and the communication costs of the machine. Other overheads are also relevant, such as the time taken to allocate processors to a skeleton, or its setup time. It is expressed as

$$T_{setup} = I_o + I_1 P$$

where constants  $I_o$  and  $I_1$  are determined by benchmarking once for each skeleton and depend on the initialisation protocol.

The communication parameters  $T_o$  and  $T_b$  correspond to the communication startup overhead and the time taken to send one byte on the network between two processors respectively. These values are available through standard benchmarks [13]. The implementation imposes further overheads on communication. The data structures used by Hope<sup>+</sup> must be converted to byte streams, *flattened*, at the transmitting end and converted back, *unflattened*, at the receiving end. This is a function of both the type of the data transmitted as well as its size. An approximate formula for the time taken to either flatten or unflatten the data that ignores the type of data is given as

$$T_{flatten} = T_{f_o} + T_f D$$

where  $T_{f_o}$  and  $T_f$  are measured by benchmarking and  $D$  is the size of the data. Though only an approximation it provides sufficient accuracy, as shown later. The garbage collection overhead must also be taken into consideration. Since each processor runs its own copy of the garbage collector, this overhead applies to both the sequential and parallel cases. It is invoked whenever the heap space used reaches a certain value [3]. The frequency at which this occurs,  $G$ , is both compile and run-time dependent and needs a mixture of both profiling and static analysis of the application to determine its value. The average time taken for a call to the collector,  $T_g$ , was determined by benchmarking. The garbage collection overhead is thus expressed as

$$T_{gc} = W G T_g$$

where  $W$  is the total amount of work done on the processor.

### 5.1 A farm model

The general model for the FARM described in section 4 was taken and instantiated with our implementation. For the prototype the total execution,  $T_{farm}$ , is given by

$$T_{farm} = T_{setup} + R (T_{comms} + T_{slave}) + T_{gc}$$

This is a summation of the set up time, the time required to distribute work packets of size  $d$  and collect the corresponding results of size  $r$ , the time to do the work on the slaves as well as the garbage collection overhead. There are  $N_w$  work packets each requiring  $T_w$  time units to execute. However,  $g$  work packets can

be clustered together in order to increase the grain size of the work done on the slaves. The number of rounds,  $R$ , required can thus be given as

$$R = \left\lceil \frac{N_w}{g(P-1)} \right\rceil$$

The time to process a work packet on any of the slaves,  $T_{slave}$  is calculated as  $g T_w$  and the communication time,  $T_{comms}$ , is given as

$$T_{comms} = 2T_o + g(d+r)T_b + g(2T_{f_o} + T_f(r+d))$$

Since the slaves work autonomously in parallel, the garbage collection overhead is determined by the number of garbage collections occurring on just one slave. This is similar to the uniprocessor case, but with less work done on each processor. The garbage collection overhead can then be given as

$$T_{gc} = \left\lceil W_s g G \left\lceil \frac{N_w}{g(P-1)} \right\rceil \right\rceil T_g$$

where  $W_s$  is the amount of work done for each packet.

### 5.2 A pipe model

As with the FARM model, we can refine the PIPE general model given in section 4 with our implementation parameters. For the implementation there are  $N_w$  work packets, each requiring  $T_w$  time units to process. The work packets can be clustered into groups of  $g$  packets. The number of bytes transferred between stages for each packet is  $d$ . The total number of virtual stages, which is independent of the number of processors is  $N_p$ . Thus, the time taken can be expressed as

$$T_{pipe} = T_{setup} + T_{fill} + (R-1)(T_{prc} + T_{comms}) + T_{gc}$$

where

$$\begin{aligned} R &= \left\lceil \frac{N_w}{g} \right\rceil \\ T_{prc} &= g T_w \left\lceil \frac{N_p}{P-1} \right\rceil \\ T_{fill} &= P(T_o + d T_b) + (P-1)(2(T_{f_o} + d T_f) + T_{prc}) \\ T_{comms} &= T_o + d T_b + 2(T_{f_o} + d T_f) \end{aligned}$$

Modeling the garbage collection time for the PIPE skeleton is not as simple as for the FARM skeleton, where it is very regular. For PIPE, in an ideal case all garbage collection events will occur simultaneously. In

the worst case this could happen sequentially, where a work packet might encounter the garbage collection event on each processor. This is highly dependent on the shape and values of the run-time data. A rough estimate for the overhead can be acquired by averaging both values.

## 6 Case study of ray-tracing

A case study of ray-tracing was made to test the methodology for more complex problems. Experimental results were gathered and compared with predicted values from the performance models.

### 6.1 Specification

The problem is similar to that described by Kelly in [14]. Ray-tracing is a technique used to generate images of scenes. The visual attributes of each pixel are determined by tracing a ray from the viewpoint, through the pixel and determining which object in the scene is struck first, if any. The intensity and colour of the pixel can then be calculated from the properties of the object. Taking the rays and objects as two pools of items we can naturally express this problem using RaMP. The rays and objects interact by intersecting and we are interested in the earliest intersection for each ray.

```
RayTrace :: Int → Int → Point → [Object] →
           [Impact]
```

```
RayTrace wd ht viewpoint scene =
  RaMP TestForImpact Earlier
  (GenerateRays wd ht viewpoint) scene
```

```
TestForImpact :: Ray → Object → Impact
-- Returns the impact of a ray with an object
```

```
Earlier :: Impact → Impact → Impact
-- Given two impacts it returns the one closest
```

```
GenerateRays :: Int → Int → Point → [ Ray ]
-- Returns a list of rays for a given
-- screen size and viewpoint
```

Actual implementations of ray-tracing employ parallelism of the pipelined or farm type. However, there are equivalences between RaMP, FARM and PIPE.

```
RaMP f g as bs =
  ( map snd ∘ PIPE (map map (map g' bs)) ∘
    map (pair unitg)) as
  where g' b (a, c) = (a, g (f a b) c)
```

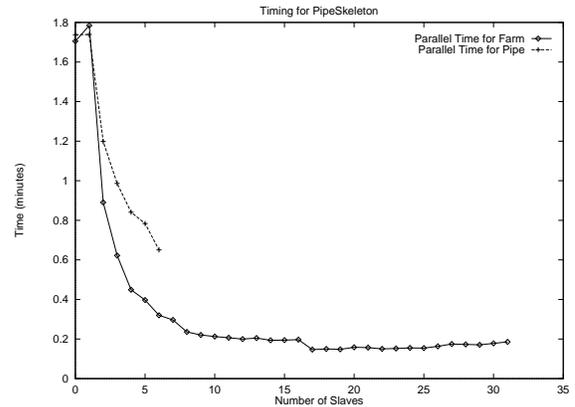


Figure 1: Comparison of PIPE and FARM real times

```
RaMP f g as bs =
  FARM h bs as
  where h ev x = foldr1 g (map (f x) ev)
```

These can be derived by formal transformations and hence are known to preserve meaning. The transformation to PIPE is discussed in [14] and for FARM is straightforward. Using these equivalences we immediately arrive at the same parallelism as found in actual implementations. Moreover, the move from natural specification to parallel implementation was one of selection and instantiation, rather than insight.

```
RayTrace wd ht viewpoint scene =
  (map snd ∘ PIPE (map map (map g' scene))
  ∘ map (pair NoImpact))
  (GenerateRays wd ht viewpoint)
  where g' b (a, c) =
    (a, Earlier ( TestForImpact a b) c)
```

```
RayTrace wd ht viewpoint scene =
  FARM h scene
  (GenerateRays wd ht viewpoint)
  where h ev x = foldr1 Earlier
    (map (TestForImpact x) ev)
```

One can see that in the PIPE case there is one object of the scene for each stage of the pipe. The rays are pipelined through this pipe with the nearest impact being updated in each stage. In comparison, for the FARM example each ray is processed independently. Hence many rays can be processed in parallel.

### 6.2 Results

Experiments were performed using both versions of the ray-tracer on our prototype implementation. The

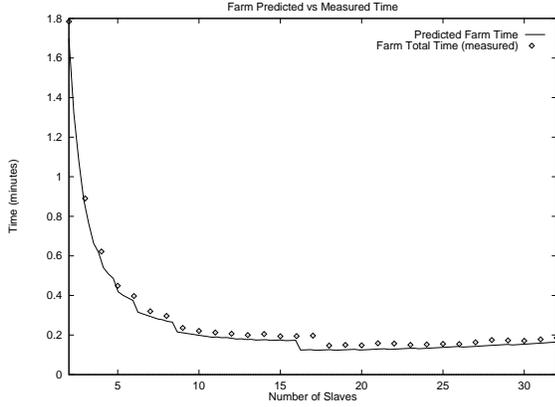


Figure 2: FARM predicted *vs* measured times

tests were carried out using a scene of six rectangles and a screen of resolution 15 by 15 pixels. In order to be able to choose between both implementations without executing the programs, the performance models from section 5 were instantiated with the problem specific parameters to give predictions. Our experiments compared the predicted times with the measured times to give some indication of the quality of the models.

Figure 1 shows a comparison between the total execution time on both the FARM and the PIPE skeletons. The *x-axis* represents the number of slaves used by a skeleton and when  $x = 0$  we have the sequential case. The *y-axis* gives the execution time. The problem is parameterised by the total number of objects in the scene,  $N_j$  and the total number of rays tested,  $N_r$ . The total number of tests for impact in the scene is thus  $N_j N_R$ . The average time for each test of impact is  $T_m$  and was measured through profiling. Ideally a mixture of both static analysis and profiling would be used to relate this value to the machine specific parameter,  $T_e$ . The time required for solving the whole problem on one processor can be expressed as

$$T_{seq} = N_r N_j T_m + T_{gc}$$

By profiling, the garbage collection was measured to occur on average once for every 91 impact tests. This gave  $G$  a value of  $1/91$  and  $W$  a value of  $N_j N_r$ .

**Farm case:** When using the FARM skeleton, the number of work packets corresponds to the total number of rays in the problem and thus  $N_w = N_r$  and  $T_w = T_m$ . Each processor holds a copy of all the objects in the scene and thus  $W_s = N_j$  and the time to process a work packet on a slave is  $g N_j T_m$ . The value of  $d$  is number of bytes in the representation of a ray and the value of  $r$  is the average of the number

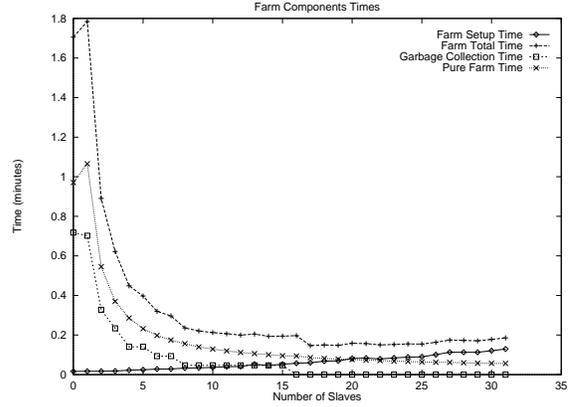


Figure 3: Components of the FARM execution

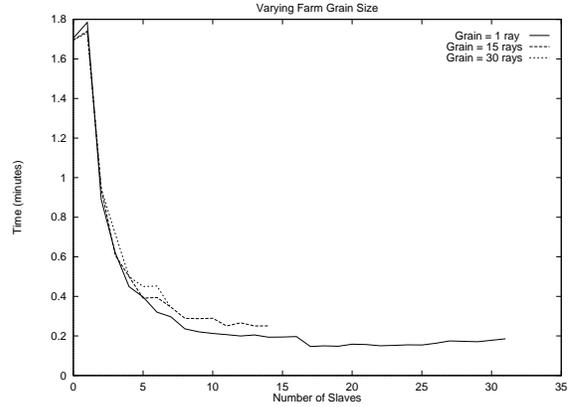


Figure 4: FARM execution time with grain size

of bytes in the impact and no impact representations. The garbage collector overhead is calculated as

$$T_{gc} = \left\lceil N_j G \left\lceil \frac{N_r}{P-1} \right\rceil \right\rceil T_g$$

Figure 2 shows the actual measured time *vs* the predicted time using the above model. A breakdown of the time taken by the components which make up FARM can be seen in figure 3. Interestingly, the garbage collection time decreases with the number of processors whilst the skeleton initialisation time increases as a function of  $P$ .

Experiments were also performed with different grain sizes. The results of which are shown in figure 4.

**Pipe case:** When using the PIPE skeleton, the number of work packets also corresponds to the number of rays in the problem and thus  $N_w = N_r$  and  $T_w = T_m$ . However, the number of virtual pipe stages corresponds to the number of objects in the scene and thus

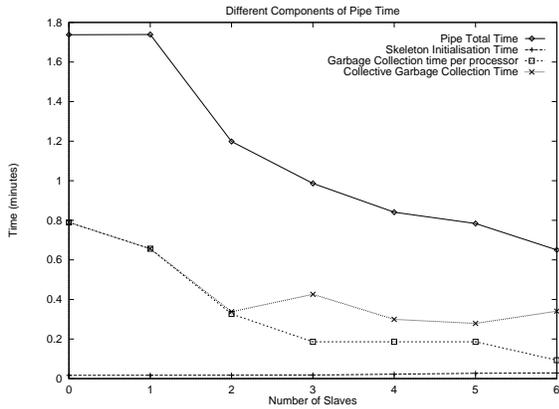


Figure 5: Components of the PIPE execution

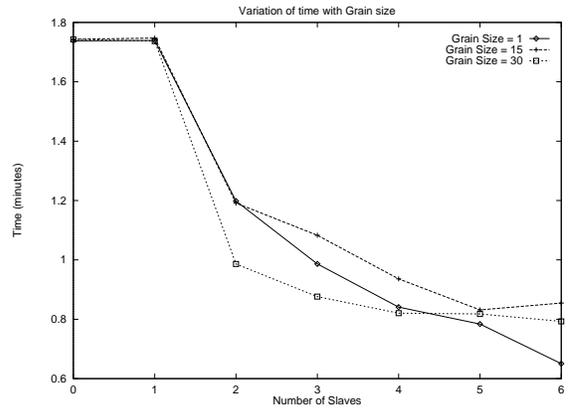


Figure 7: PIPE execution time with grain size

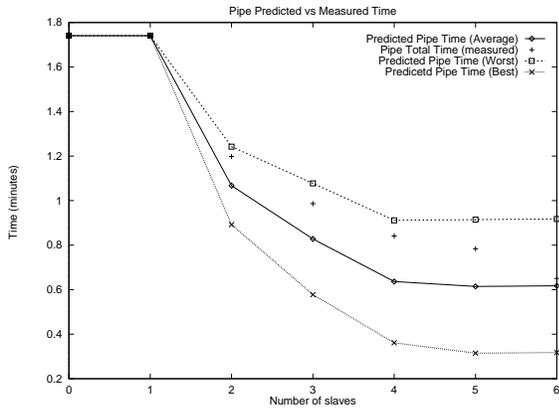


Figure 6: PIPE predicted vs measured times

$N_p = N_j$ . The value of  $d$  is the size of the data structure passed between the different stages. An average garbage collection value was calculated as described in section 5, by profiling.

We give similar graphs for PIPE as we did for FARM. Most importantly, figure 5 shows the times for the different components of the PIPE skeleton and figure 6 shows the predicted time of the skeleton compared with the experimental time.

### 6.3 Conclusions of the results

It has been demonstrated that performance models of implementations can be used to predict the execution times of skeleton programs by instantiating such models with the problem's parameters. Accurate predictions were made for the FARM version even given our simple model. Less accurate predictions were obtained for the PIPE example, though they still gave a reasonable indication. We identified the problem to be the poor modelling of the garbage collector. This

indicates that more sophisticated methods need to be used or possibly a more predictable collector. Fortunately it is easy to refine the model because of the way they are described. Increasing the grain size produced interesting and unexpected results. Though the results were not intuitive the performance models were able to predict the execution times.

## 7 Conclusions

Parallel programs are difficult to write and prone to unpredictable performance and non portability. This paper proposes a methodology that aims to eliminate these problems. Skeletons are used to express parallelism, with transformations providing portability and control over some behaviour options. To aid in the choice of a particular skeleton, and the resource allocations within a given skeleton we suggest the use of performance models.

We demonstrated how models can be constructed generally for a skeleton, then refined for a particular implementation. The accuracy of these simple models was shown by comparison with real figures for an example problem. This example supports the validity of the method as an alternative way of programming parallel machines.

Further investigation is needed of more sophisticated implementations of skeletons to see if accurate performance models are still possible under these conditions. There are also still some unanswered questions concerning the composition and construction skeletons. Methods would be needed to model the performance of composed skeletons.

Related work includes [16] which uses performance models to guide the use of templates. [4] looks at the

implications of a compiler which identifies skeletons within a program.

## Acknowledgements

We would like to thank our colleagues in the Advanced Languages and Architectures Section at Imperial College for their assistance and ideas. The prototype system reported here was initially developed through a SERC/DTI funded project 'The Exploitation of Parallel Hardware using Functional Languages and Program Transformation'. This work is being continued under a British Council Scholarship to the second author and a SERC Research Scholarship to the third author.

## References

- [1] J. Backus, "Can programming be liberated from the von-Neumann style? A Functional Style and its Algebra of Programs." *CACM*, **21(8)**, 613-41, 1978.
- [2] R. Bird and P. Wadler, *Introduction to Functional Programming*. Prentice Hall, 1988.
- [3] H. Boehm and M. Weiser, "Garbage collection in an uncooperative environment." *Software—Practice and Experience*, **18(9)**, 807-820, 1988.
- [4] Tore. A. Bratvold, *A Skeleton-Based Parallelising Compiler for ML*. Department of Computing and Electrical Engineering, Heriot-Watt University, 1993.
- [5] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation*. Pitman/MIT Press, 1989.
- [6] J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp and Q. Wu, "Parallel programming using skeleton functions." *Parallel Languages And Architectures, Europe:Parle '93*. To appear, 1993.
- [7] J. Darlington and H. M. Pull, "A program development methodology based on a unified approach to execution and transformation." *Partial Evaluation and Mixed Computation*. North-Holland, 1988.
- [8] J. Darlington and H. W. To, "Building parallel applications without programming." *Proceedings of the Leeds' Workshop on Abstract Parallel Machine Models 93*. To appear, 1993
- [9] S. Fortune and J. Willey, "Parallelism in random access machines." 10<sup>th</sup> *ACM Symposium on Theory of Computing STOC*, 1978.
- [10] P. Hudak *et al*, "Report on the functional programming language Haskell." *SIGPLAN Notices*, **27(5)**, May 1992.
- [11] A.J.Hey, "Practical Parallel Processing with Transputers." *Third Conference on Hypercube Concurrent Computers and Applications*, 1988.
- [12] R. W. Hockney and C. R. Jesshope, *Parallel Computers 2*. Adam Hilger, 1988.
- [13] S. Y. Huang, "CSTools Performance on Transputers." Dept. of Computing, Imperial College, 1991.
- [14] P. H. J. Kelly, *Functional Programming for Loosely-coupled Microprocessors*. Pitman/MIT Press, 1989.
- [15] Meiko Ltd., *CS Tools for SunOS*. Edition: 83-009A00-02.02, 1990.
- [16] S. Pelagatti *A methodology for the development and the support of massively parallel programs*. PhD Thesis, Universita Delgi Studi Di PISA, 1993.
- [17] N. Perry, *Hope+*. IC/FPR/LANG/2.5.1/7 Internal Document, Dept. of Computing, Imperial College, 1989.
- [18] D. Pritchard, "Performance Analysis and Measurement on Transputer arrays", *Evaluating Supercomputers*, Chapman and Hall, 1990.
- [19] K. G. Waugh, *An Algorithm to Balance a Pipeline of Processes*. Technical Report 92011, Dept. of Computing, Heriot-Watt University, 1992.
- [20] L. G. Valiant, "General purpose parallel architectures." *Handbook of Theoretical Computer Science*. North-Holland, 1990.