Parallel Programming Using Skeleton Functions

J. Darlington, A.J. Field, P.G. Harrison,
P.H.J. Kelly, D.W.N. Sharp, Q. Wu
Dept. of Computing, Imperial College, London SW7 2BZ
email: {jd,ajf,pgh,phjk,dwns,wq}@doc.ic.ac.uk

R.L. While
Dept. of Computer Science, University of Western Australia,
Nedlands, Western Australia 6009
email: lyndon@cs.uwa.edu.au

Abstract

Programming parallel machines is notoriously difficult. Factors contributing to this difficulty include the complexity of concurrency, the effect of resource allocation on performance and the current diversity of parallel machine models. The net result is that effective portability, which depends crucially on the predictability of performance, has been lost.

Functional programming languages have been put forward as solutions to these problems, because of the availability of implicit parallelism. However, performance will be generally poor unless the issue of resource allocation is addressed explicitly, diminishing the advantage of using a functional language in the first place.

We present a methodology which is a compromise between the extremes of explicit imperative programming and implicit functional programming. We use a repertoire of higher-order parallel forms, *skeletons*, as the basic building blocks for parallel implementations and provide program transformations which can convert between skeletons, giving portability between differing machines. Resource allocation issues are documented for each skeleton/machine pair and are addressed explicitly during implementation in an interactive, selective manner, rather than by explicit programming.

1 Introduction

The main obstacle to the commercial uptake of parallel computing is the complexity and cost of the associated software development process. Programming parallel machines is more difficult than programming sequential machines in at least two fundamental ways: **predictability of performance** and **portability.**

Predictability of performance

Sequential programming languages, incorporating the von-Neumann model of computation, enjoy a simple one-to-one mapping between language constructs and their underlying machine implementation. Issues such as memory allocation are resolved by the compiler with no performance implications, allowing the programmer to concentrate on high-level aspects of the algorithm. The programmer can fairly confidently predict the performance of a program on a particular machine, whilst avoiding the burden and complexity of run-time resource allocation.

In contrast, the mapping of a parallel program onto a multiprocessor machine is typically a complex process involving decisions about the distribution of processes over the processors of the machine, scheduling of processor time between competing processes, communication patterns, etc. Often the only way for the programmer to achieve the desired level of performance is to take explicit control of these decisions in the program, with the obvious increase in program complexity and a corresponding deterioration in program reliability. Some predictability is retained with shared-memory multiprocessors, which attempt to sustain the von-Neumann model at low degrees of parallelism, but such machines are not scalable to the levels of performance required by many application areas.

Portability

The universality of the von-Neumann model guarantees portability of sequential programs at the language level, with no danger of an unforeseen degradation in performance. A sequential program moved to a machine with a faster processor will, almost certainly, run faster.

In the world of parallel machines the explicit nature of resource allocation means there is rarely any portability at all. Even where a high-level language can be compiled for different machines, the wide disparity in the architectures available means that the performance of a program can vary wildly and in unpredictable ways unless it is radically altered as part of the porting process.

The diversity of parallel machine architectures and the lack of a common model of computation has led the application development community to fragment into incompatible, machine-oriented camps with proprietary languages/language extensions predominating at the expense of a proper understanding of the field.

There appear to be two routes out of the current state of affairs.

- One approach is the development of a 'parallel von-Neumann machine', an abstract machine to which any useful programming model can be compiled with predictable (small) loss of performance, and which can itself be implemented on a scalable physical architecture, again at a known cost. This is the route taken by research into the parallel random-access machine (PRAM[17]) and distributed shared memory[12], which attempts to

provide the illusion of a shared address space on a physically-distributed machine, in effect taking the shared-memory model to arbitrary degrees of parallelism.

- The second, perhaps more direct, approach is the development of a programming methodology for parallel machines which allows portability both of programs and their performance across the whole range of architectures. This is the approach taken in this paper.

Our approach involves abandoning the search for portability at the language level in favour of a structured decision-making process based on the use of high-level program forms, source-level program transformation and performance modelling.

2 An Overview of the Methodology

The central idea is to replace explicit parallel programming, using a parallel language, by the selection and instantiation of a variety of pre-packaged parallel algorithmic forms known as *skeletons*. The approach is similar to that taken by Cole[2] for imperative languages and follows Backus's principle[1] that the key to effective (functional) programming is the availability of a small fixed set of special operators (program-forming operations) which allow new functions to be created from old ones. The methodology can be broken down into three principal components: **skeletons**, **performance models** and **program transformation**.

Skeletons

A skeleton captures an algorithmic form common to a range of programming applications. In our work, skeletons have been developed as polymorphic, higher-order, functions in a non-strict functional programming language.

Each skeleton has a declarative *meaning*, established by its functional language definition. This meaning is independent of any particular implementation of the skeleton: this allows skeletal programs to be prototyped rapidly on sequential platforms and to be fully portable between different parallel machines. A skeleton also has specific *behaviours* on particular parallel machines on which it is known to be implementable. Of course, in principle, any skeleton can be executed on any machine: however, each skeleton is associated with a set of architectures on which efficient realisations are known to exist.

All parallelism in a program derives from the behaviour of its skeletons on the machine in question. Functions to which skeletons are applied are executed sequentially. All aspects of a skeleton's parallel behaviour, such as process placement or interconnectivity, are either clear from its definition or documented as issues to be addressed explicitly during implementation.

Performance models

Each skeleton/machine pair has associated with it a performance model which can be used to predict the performance of a program written using the skeleton on that machine. These models are used by the programmer, the transformation system and the compiler to guide decision-making at all levels of the program development process. Resource allocation in particular relies heavily on the use of these performance models.

Program transformation

Program transformation is used in the development process at all levels. At the topmost level, for example, it can be used to transform high-level problem specifications into initial skeleton forms. At the lower levels it can be used to convert programs from one skeleton form to another e.g. for the purposes of portability. At the lowest level, transformation can be used to fine-tune an architecture-specific program to a particular machine in that class. This may involve, for example, partial evaluation[4] to vary the grain-size used in an application or to configure the program for a particular machine size.

Wherever possible, the methodology aims to replace *(re)invention*, both of programs and transformations, by *selection* from a limited range of possibilities determined by context. The skeletons and associated transformations form a *decision-tree* that can be navigated by the programmer to map high-level specifications onto concrete machine architectures.

Portability of programs is provided by the high-level nature of the the original program specification and the ability to record, replay and alter the derivation process from specification to implementation. Resource allocation is tackled explicitly by addressing the important performance questions directly rather than implicitly by writing a program with the desired properties.

The next three sections of the paper discuss the three main aspects of the methodology in more detail. Section 6 discusses the implementation of the methodology and Section 7 concludes the paper.

3 Parallel Algorithmic Skeletons

3.1 Initial Skeletons

An initial set of skeletons has been defined to capture the most common forms used in parallel algorithms. These are listed below, all definitions are expressed in Haskell [8].

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.

```
PIPE :: [α → α] → (α → α)
PIPE = foldr1 ( . )
```

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).

```
FARM :: (α → β → γ) → α → ([β] → [γ])
FARM f env = map . (f env)
```

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.

```
DC :: (α → Bool) → (α → β) → (α → [α]) → ([β] → β) → α → β
DC t s d c x | t x          = s x
             | not ( t x ) = ( c . map ( DC t s d c ) . d ) x
```

Another common class of algorithms describes systems where each object in the system can potentially interact with any other object. Each individual interaction is calculated and the results are combined to produce a result for each object. 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 form, for example by farming out the calculations for each object or by pipelining over the functions f and g.

```
RaMP :: (α → α → β) → (β → β → β) → [α] → [β]
RaMP f g xs = map h xs
              where h x = foldr1 g ( map ( f x ) xs )
```

More dynamic algorithms are typified by the DMPA skeleton ('Dynamic-Message-Passing-Architecture'). Here any process can interact directly with any other process via message-passing, the actual connections being determined using run-time data. Each process has an internal state which records values local to the process: messages from other processes may modify the process's state and generate new messages to other processes. Parallelism arises from evaluating the processes on different processors.

```
DMPA :: {{α} → {(Int, α)}} → {(Int, α)} → {α}
DMPA { P_i initState_i | 1 ≤ i ≤ n } initMess
        = filterms 0 mess
          where mess = P_1 initState_1 ( filterms 1 mess ) U ··· U
                       P_n initState_n ( filterms n mess ) U initMess
                filterms i ms = { conts | ( j, conts ) ∈ ms, i == j }
                P_i localState ( c U cs ) = replies U P_i updState cs
```

All these skeletons describe MIMD modes of operation. The work described in [10] brings SIMD machines, such as the Thinking Machines' CM-2, within the range of our techniques. There a small set of higher-order primitives is defined corresponding to the basic computation and communication capabilities of such machines. There is a very natural fit between these primitives and the aggregate view of computation, providing both a congenial abstraction of SIMD machines and a basis for the efficient support of array operations in functional languages. These primitives provide a platform on which skeletons describing SIMD computations can be defined.

3.2 Example Applications

This section gives examples of the use of the skeletons in describing typical applications. Some functions which only perform low-level arithmetic or data manipulations are not fully specified.

As an example of the use of the PIPE skeleton the function compile below defines the general structure of a compilation route for a high-level programming language.

```
compile :: [Char] → [Char]
compile = PIPE [ writefile, genCode, typeCheck, parse, lex, readfile ]

writefile, genCode, typeCheck, parse, lex, readfile :: [Char] → [Char]
various stages in compiling a program
```

In the function exposedFaces, the FARM skeleton is used to determine which faces of a convex 3-dimensional body are visible from the origin of the co-ordinate system. Each face is checked individually by reference to a point which is inside the body. The co-ordinates of this point form the shared environment of the farm.

```
exposedFaces :: [Face] → [(Face, Bool)]
exposedFaces fs = zip fs ( FARM checkIfVisible ( pointInBody fs ) fs )

pointInBody :: [Face] → Point
calculate a point which is inside the body fs (assumed convex)
```

checkIfVisible :: $Point \rightarrow Face \rightarrow Bool$
given a point p *inside the body, check if face* f *is visible*

An example application of the DC skeleton is mergesort. Given a function merge which combines two sorted lists whilst retaining their ordering, mergesort works by recursively splitting its argument into smaller sublists until the sublists are trivially sorted, then using merge to build a sorted permutation of the original list.

```
mergesort :: (α → α → Bool) → [α] → [α]
mergesort = (DC isSingleton id split) . foldr1 . merge
            where isSingleton xs = length xs ≤ 1

split :: [α] → [[α]]
split xs into a list of its sublists

merge :: (α → α → Bool) → [α] → [α] → [α]
merge two sorted lists into a sorted list
```

An example of the RaMP skeleton is the classical problem of nBody simulation. At each step of the simulation, the force between each pair of bodies is calculated and these are summed to determine the total force acting on each body and hence its new position and velocity.

```
nBody :: [Planet] → [[Planet]]
nBody ps = ps : nBody ( map newPos
                              ( zip ps ( RaMP calcF sumFs ps ) ) )

newPos :: (Planet, Force) → Planet
calculate the new position and velocity of planet p

calcF :: Planet → Planet → Force
calculate the force exerted by planet p1 on planet p2

sumFs :: Force → Force → Force
combine the effects of forces f1 and f2
```

The DMPA skeleton describes the most dynamic algorithms, where the interactions between processes are determined using run-time data. Interaction is via message-passing. The function database describes a dynamically-changing database whose contents are distributed over a network of processors. Each node has to be capable of handling requests for the whole database: requests which cannot be handled locally are forwarded to the relevant processor.

```
data Message = Query DataItem | Add DataItem | Del DataItem
             | other message-types
```

```
database :: {(Int, Message)} → {Message}
database = DMPA { dbmanager_i initData_i | 1 ≤ i ≤ n }

dbmanager_i :: Localdata → {Message} → {(Int, Message)}
dbmanager_i dat ( Query info U ms )
            | DB == i = ( 0 , reply ) U dbmanager_i dat ms
            | DB /= i = ( DB , Query info ) U dbmanager_i dat ms
                        where DB = whereStored info
dbmanager_i dat ( Add info U ms )
            | DB == i = dbmanager_i ( insert info dat ) ms
            | DB /= i = ( DB , Add info ) U dbmanager_i dat ms
                        where DB = whereStored info
dbmanager_i dat ( Del info U ms )
            | DB == i = dbmanager_i ( delete info dat ) ms
            | DB /= i = ( DB , Del info ) U dbmanager_i dat ms
                        where DB = whereStored info

whereStored :: DataItem → Int
where is data of the type of item stored?

insert, delete :: DataItem → Localdata → Localdata
insert/delete an item into/from the local database
```

Many other examples of the DMPA skeleton in action are described in [16], including a novel approach using dynamically-generated patterns of communication to maximise the potential of the network facilities of MIMD machines. Examples include a new algorithm for parallel quicksort of $O(\log n)^2$ and new algorithms for fractal generation and tesselation.

4 Performance Models

The ultimate aim of a parallel programmer is to write a program that will execute efficiently on the chosen target machine.With today's software technology targeted at non-uniform machines it is a difficult task to even predict the performance of a given parallel program, let alone to ensure that it will be optimal. We would characterise today's approach by the term *performance debugging.* The programmer writes a program that he hopes is reasonably efficient, executes it and observes its behaviour. The information gained from these observations is then used to modify the resource allocation decisions embodied in the program, and the modified program is executed again to see if any improvement ensues. Often the programmer is proceeding in the dark, as he may not even know what factors are important in determining the performance of the program.

Here we seek to develop a more scientific methodology based on the use of

performance models which, given a program, can both predict its performance and suggest what may be done to improve that performance. Such a performance model is typically a set of analytical formulae parameterised by attributes of both the program and the machine. There has been an impressive body of work in producing such models for parallel hardware and software [7]. However, the state of the art is unable to provide practical methods to predict the performance of an arbitrary program executing on an arbitrary machine. By limiting our programs to instantiations of known skeletons, each targetted at a specific set of machines, the methodology becomes more practical.

A performance model is associated with each skeleton/machine pair and is used constructively in the programming process. A preliminary model is produced and verified and quantified experimentally. The model is adjusted until it is shown to be a reliable predictor of performance. This is equivalent to playing out the 'performance debugging' process once for each configuration and recording the result for future reference.

Consider as an example the Divide-and-Conquer skeleton, DC, targetted onto a distributed-memory machine. Such an architecture results in very non-uniform memory access times, with local store access being much cheaper than remote store access. The two most important factors governing program performance will thus be process granularity and data placement. The model, therefore, needs to take account of the complexity of each of the argument functions of DC and the speed of communication between processors. Taking all these factors into account, an application should be solved in parallel if the following condition holds (assuming a binary division function):

$$T_{sol_G} > T_{div_G} + T_{sol_{G/2}} + T_{comb_{G/2}} + T_{comms}$$

where T_{sol_x} is the time to solve a problem of size x on one processor, T_{div_x} is the time to split a problem of size x into two sub-problems, T_{comb_x} is the time to combine the results from two problems of size x and T_{comms} is the time to communicate problems and results between processors. The reasoning behind this formulae is that the right hand side represents the worst case involved in going parallel, i.e. there is no further gain to be made from further parallel execution and the two subproblems are solved sequentially. If this worst case is still less than the time to solve sequentially, T_{sol_x}, then it pays to keep dividing.

We can expand this to calculate the total time required to solve a problem of size G on M processors:

$$T_{sol_G} = \sum_{i=1}^{\log M} (T_{div_{G/2^{i-1}}} + T_{comb_{G/2^i}} + T_{comms}) + T_{sol_{G/M}}$$

Solving this equation for M will tell us the optimal number of processors to use in the evaluation. Note that further decisions will have to be made about whether shared data should be evaluated once and accessed remotely, evaluated

once and copied to each processor or re-evaluated at each processor. [5] gives a performance model combining all these factors.

Many decisions in resource allocation can be expressed as source-level transformations, for example balancing the stages of a pipeline or matching the number of pipe-stages to the number of physical processors available. Decisions such as these can be implemented as transformation routines to be applied by the programmer after consultation with the performance model. Other decisions sit more naturally in the compilation process from the skeleton to the native code of the target machine. In particular, some skeletons will have multiple implementations on some machines, and the choice of the optimal one will be guided by the performance model.

We believe that this constructive use of performance models complements our structured approach to parallel programming. We consider it important that factors affecting performance are identified and quantified so they can be addressed explicitly and the relevant decisions documented, rather than being left unstated and accomplished indirectly as a side effect of a program with the appropriate behaviour.

5 Program Transformation

Transformation provides a natural route to portability in that a program written in terms of a skeleton which cannot be implemented easily on a given architecture can be re-expressed in terms of another skeleton which does have an efficient implementation on that architecture. This particularly applies to the higher-level skeletons which may not map easily onto any architectures.

As an example, a program written in terms of the RaMP skeleton can be implemented as a pipeline with length xs + 2 stages[11]:

$$
\begin{aligned}
\mathsf{RaMP\ f\ g\ xs} \equiv\ & \mathsf{(\ map\ snd\ .\ PIPE\ (\ map\ map\ (\ map\ g'\ xs\ )\ )} \\
& \quad \mathsf{.\ map\ (\ pair\ unit_g\ )\ )\ xs} \\
& \mathsf{where\ g'\ b\ (\ a,\ c\ ) = (\ a,\ g\ (\ f\ a\ b\ )\ c\ )} \\
& \quad \mathsf{pair\ a\ b\ \ \ \ = (\ b,\ a\ )}
\end{aligned}
$$

Alternatively it can be implemented on a distributed architecture as a FARM:

$$
\begin{aligned}
\mathsf{RaMP\ f\ g\ xs} \equiv\ & \mathsf{FARM\ h\ (\ f,\ g,\ xs\ )\ xs} \\
& \mathsf{where\ h\ (\ f,\ g,\ xs\ )\ x = foldr1\ g\ (\ map\ (\ f\ x\ )\ xs\ )}
\end{aligned}
$$

Note that transforming a RaMP to a FARM leaves many implementation issues still to be resolved, in particular whether the environment is to be accessed remotely or passed to each processor.

An inter-skeleton transformation which relies heavily on fine-tuning is DC to PIPE. By assuming that an application of DC is overrun-tolerant[19], we can obtain the equivalence[18][6]

$$\begin{aligned}
&\mathsf{map\ (\ DC\ t\ s\ d\ c\ )} \equiv \mathsf{PIPE\ (\ rept}\ q\ \mathsf{(\ map'\ n\ c\ )\ )\ .\ map\ s\ .}\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathsf{PIPE\ (\ rept}\ q\ \mathsf{(\ foldr1\ (\ ++\ )\ .\ map\ d\ )\ )}
\end{aligned}$$

$$\begin{aligned}
&\mathsf{rept} :: \mathit{Int} \to \alpha \to [\alpha]\\
&\mathsf{rept\ n} = \mathsf{take\ n\ .\ repeat}
\end{aligned}$$

$$\begin{aligned}
&\mathsf{map'} :: \mathit{Int} \to ([\alpha] \to \beta) \to [\alpha] \to [\beta]\\
&\mathsf{map'\ n\ f\ xs} \mid \mathsf{length\ xs} \geq \mathsf{n} = \mathsf{f\ (\ take\ n\ xs\ )\ :\ map'\ n\ f\ (\ drop\ n\ xs\ )}\\
&\qquad\qquad\quad \mid \mathsf{length\ xs} < \mathsf{n} = [\,]
\end{aligned}$$

where q is the number of levels in the evaluation tree and map' is a variation of map which consumes its argument list in chunks of n elements. In the above expression, n is the arity of each node in the evaluation tree, i.e. the length of the result list of d. This transformation gives us a version of the application which evaluates on a pipeline of length $2q + 1$ for arguments up to 'size' n^q.

For specific applications of DC we are often able to do much better, however. Take the definition of mergesort from Section 3.2:

$$\begin{aligned}
&\mathsf{mergesort} = \mathsf{(DC\ isSingleton\ id\ split)\ .\ foldr1\ .\ merge}\\
&\qquad\qquad\quad \mathsf{where\ isSingleton\ xs} = \mathsf{length\ xs} \leq 1
\end{aligned}$$

Unfolding the definition of mergesort once, and assuming the non-trivial case, we can derive

$$\mathsf{mergesort\ f} \equiv \mathsf{foldr1\ (\ merge\ f\ )\ .\ map\ (\ mergesort\ f\ )\ .\ split}$$

This equivalence holds for any implementation of split which satisfies the property

$$\mathsf{mergesort\ f\ .\ foldr1\ (\ ++\ )\ .\ split} \equiv \mathsf{mergesort\ f}$$

which is essentially the specification of split. We will choose a definition of split which reduces its argument list to singletons in one pass (it is trivally shown to satisfy the above property):

$$\begin{aligned}
&\mathsf{split} :: [\alpha] \to [[\alpha]]\\
&\mathsf{split} = \mathsf{map\ mkSingleton}\\
&\qquad\quad \mathsf{where\ mkSingleton\ x} = [\,\mathsf{x}\,]
\end{aligned}$$

Applying the DC to PIPE transformation to the definition of mergesort gives us

$$\begin{aligned}
\mathsf{map\ (\ mergesort\ f\ )} \equiv\ &\mathsf{PIPE\ (\ rept}\ q\ \mathsf{(\ map'\ n\ (\ foldr1\ (\ merge\ f\ ))))\ .}\\
&\qquad\qquad\quad \mathsf{map\ id}\\
&\qquad\qquad\quad \mathsf{PIPE\ (\ rept}\ q\ \mathsf{(\ foldr1\ (\ ++\ )\ .\ map\ split\ )\ )}
\end{aligned}$$

It is trivial to show that the expression foldr1 (++) . map split is idempotent, so we have the equivalence

$$\mathsf{PIPE}\ (\ \mathsf{rept}\ q\ (\ \mathsf{foldr1}\ (\ +\!+\ )\ .\ \mathsf{map\ split}\ )\ )\ \equiv\ \mathsf{foldr1}\ (\ +\!+\ )\ .\ \mathsf{map\ split}$$

for $q > 0$, together with the obvious equivalences

$$\mathsf{map\ id} \equiv \mathsf{id}$$

$$\mathsf{f}\ .\ \mathsf{id} \equiv \mathsf{f}$$

The final pipeline for mergesort therefore has only $q + 2$ stages:

$$\begin{aligned}\mathsf{map}\ (\ \mathsf{mergesort\ f}\ ) \equiv\ & \mathsf{PIPE}\ (\ \mathsf{rept}\ q\ (\ \mathsf{map'\ n}\ (\ \mathsf{foldr1}\ (\ \mathsf{merge\ f}\ )\ )\ )\ )\ .\\ & \mathsf{foldr1}\ (\ +\!+\ )\ .\ \mathsf{map\ split}\end{aligned}$$

This is clearly a significant improvement over the naive application of the transformation.

In short, transformation allows us to take a high-level, portable specification and target it onto any architecture which is at hand, and to fine-tune an instantiation of the specification to take advantage of the particular characteristics of an architecture without compromising program legibility and reliability. Portability arises directly from the ability to replay the transformation using different rules for different architectures.

6 Implementation

We have constructed an initial implementation of the skeletons using the functional language Hope$^+$[15] as the source language and using C as the target language. This compiler makes extensive use of macros, giving us maximum flexibility to explore different implementation options, e.g. remote vs. local patterns of data access (e.g. FARM) and process placement options (e.g. DMPA).

The initial installation was carried out on a Meiko Transputer surface, using the CS Tools [13] library to provide flexibility in communication. Initial results, in terms of both speed-up and the usability of the methodology, have been promising although we have not, as yet, made direct comparisons with hand-coded versions of the same algorithms.. A subsequent, partial, implementation has been carried out on a Fujitsu AP1000 made available under Fujitsu Parallel Computing Centre Facilities programme. The AP1000 is of particular interest as its richer communication capabilities allow greater varieties of implementations to be considered. Further implementations of the skeletons on networks of workstations and a SIMD machine are planned.

7 Conclusions and Future Work

Implementation options

A preliminary study and implementation of compiler options has been carried out [9]. For each of the skeletons apart from DMPA two or three alternative implementation options were identified and the compiler extended to realise these options. Experiments showed that each of the options were more effective for some range of inputs than the general implementation.

Application-specific skeletons

Many potential application areas for parallel computing, for example databases and solid modelling, have their own characteristic high-level data and control structures. We plan to extend our skeleton-based methodology into these areas. We aim to construct domain-specific skeletons which would allow specialists to construct applications in these areas directly, without recourse to low-level programming. These initial system specifications could then be mapped onto the selected target machines by an extension of the program transformation and structured implementation techniques we have already developed. Preliminary studies in the area of solid modelling [14] and data bases have been encouraging.

'Languageless programming'

The ultimate goal of our work is to completely replace the requirement for invention or creation during application development by a process of selection from a range of possibilities determined by context. We aim to factor out all the decisions involved in creating an application and mapping it efficiently onto a machine and present them as a sequence of selections of appropriate skeletons, transformations and implementation options. Achieving this goal would have many benefits: simplifying application development; documenting the decisions made during the development of an application; and ensuring that the programmer addresses all the issues involved in the implementation process.

Given this framework, the system could be used via a menu-driven interface, with the skeletons and options presented visually. Visual programming is very attractive, but we feel that many current systems miss the point and simply present an unchanged programming paradigm in a visual manner. We consider that it is important to first convert the programming process from one of invention to one of selection, which lends itself well to the visual style of presentation.

8 Acknowledgements

We would like to thank all our colleagues at Imperial College for their inputs and assistance. The influence of and Backus's ideas on our work is obvious. The work reported here was initially developed in the UK SERC/DTI funded project 'The Exploitation of Parallel Hardware using Functional Languages and Program Transformation' and used equipment funded under the SERC's Parallel Equipment Initiative.We are also grateful to Fujitsu, Japan, for making the AP1000 machine available under the Fujitsu Parallel Research Centre Facilities programme.

References

[1] J. Backus, *Can Programming Be Liberated from the von-Neumann Style? A Functional Style and its Algebra of Programs*, CACM vol. 21, no. 8, pp. 613-41, 1978.

[2] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation*, Pitman/MIT Press, 1989.

[3] J. Darlington, Y-k. Guo and H.M. Pull, *A New Perspective on Integrating Functional and Logic Languages*, Conf. on Fifth Generation Computing Systems, Tokyo, June 1992.

[4] J. Darlington and H.M. Pull, *A Program Development Methodology Based on a Unified Approach to Execution and Transformation*, in Partial Evaluation and Mixed Computation, North-Holland, 1988.

[5] J. Darlington, M.J. Reeve and S. Wright, *Programming Parallel Computer Systems using Functional Languages and Program Transformation*, in Parallel Processing '89, Leiden, 1989.

[6] P.G. Harrison, *Towards the Synthesis of Static Parallel Algorithms: a Categorical Approach*, IFIP TC2 Working Conference on Constructing Programs from Specifications, Pacific Grove, California, May 1991 (published as *Constructing Programs from Specifications*, North-Holland).

[7] P.G. Harrison and N. Patel, *Performance Modelling: Application to Communication Networks and Computer Architecture*, Addison-Wesley, 1992.

[8] P. Hudak, S.L. Peyton Jones, P.L. Wadler, B. Boutel, J. Fairburn, J. Fasel, M. Guzmán, K. Hammond, J. Hughes, T. Johnsson, R. Kieburtz, R.S. Nikhil, W. Partain and J. Peterson, *Report on the Functional Programming Language Haskell*, SIGPLAN Notices 27(5), May 1992.

[9] C. A. Isaac, *Structural Implementations of Functional Skeletons*, MSc Project Report, Dept. of Computing, Imperial College 1992.

[10] G.K. Jouret, *Compiling Functional Languages for SIMD Architectures*, 3^{rd} IEEE Symposium on Parallel and Distributed Processing, Dallas, December 1991.

[11] P.H.J. Kelly, *Functional Programming for Loosely-coupled Microprocessors*, Pitman/MIT Press, 1989.

[12] K. Li and P. Hudak, *Memory Coherence in Shared Virtual Memory Systems*, ACM Transactions on Computer Systems vol.7, no. 4, pp. 329-59, 1989.

[13] Meiko Ltd., *CS Tools for SunOS*, 1990 Edition: 83-009A00-02.02.

[14] G. Papachrysantou, *High Level Forms for Computation in Solid Modelling*, MSc Project Report, Dept. of Computing, Imperial College 1992.

[15] N. Perry, *Hope*$^+$, Internal document IC/FPR/LANG/2.5.1/7, Dept. of Computing, Imperial College, 1989.

[16] D.W.N. Sharp and M.D. Cripps, *Parallel Algorithms that Solve Problems by Communication*, 3^{rd} IEEE Symposium on Parallel and Distributed Processing, Dallas, December 1991.

[17] L.G. Valiant, *General Purpose Parallel Architectures*, in Handbook of Theoretical Computer Science, North-Holland, 1990.

[18] R.L. While, *Transforming Divide-and-Conquer to Pipeline*, Internal note, Dept. of Computing, Imperial College, 1991.

[19] J.H. Williams, *On the Development of the Algebra of Functional Programs*, ACM Transactions on Programming Languages and Systems vol. 4, pp. 733-57, 1982.