A toolkit for parallel functional programming

Pieter H. Hartel* Rutger F. H. Hofman† Koen G. Langendoen†
Henk L. Muller‡ Willem G. Vree L.O. Hertzberger

Dept. of Computer Systems, University of Amsterdam,
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands

Summary

Our toolkit for the design and implementation of parallel functional programs supports the stepwise development of parallel programs from a high level sequential specification to an optimised parallel implementation. The toolkit is used as follows:

1. The algorithm to be implemented is specified in a functional language. The program is debugged and tested using an interpreter.
2. The program is compiled for a sequential machine. Its performance is analysed an improved.
3. Annotation driven transformations are applied to the program to indicate parallel tasks. Simulations at task level, basic block level and bus transaction level make it possible to analyse the parallel performance of the program at three levels of detail.
4. When the performance is optimised using the simulators, the program is executed on a genuine parallel machine.

Several programs have been developed with the toolkit. A program that simulates tidal flow in an estuary of the North sea is presented as a case study to demonstrate the merits of the toolkit when developing complex parallel programs.

The toolkit not only supports the design of parallel applications; it also allows the study of important concepts in parallel computer architecture. These include the behaviour of cached memory systems, bus protocols, scheduling algorithms and memory management algorithms.

Key words

Parallel functional programming, simulation, real execution, validation, case study.

1 Introduction

*Corresponding author, email: pieter@fwi.uva.nl
†Now with the Computer Science Department, Free University, Amsterdam
‡Now with the Computer Science Department, University of Bristol, UK

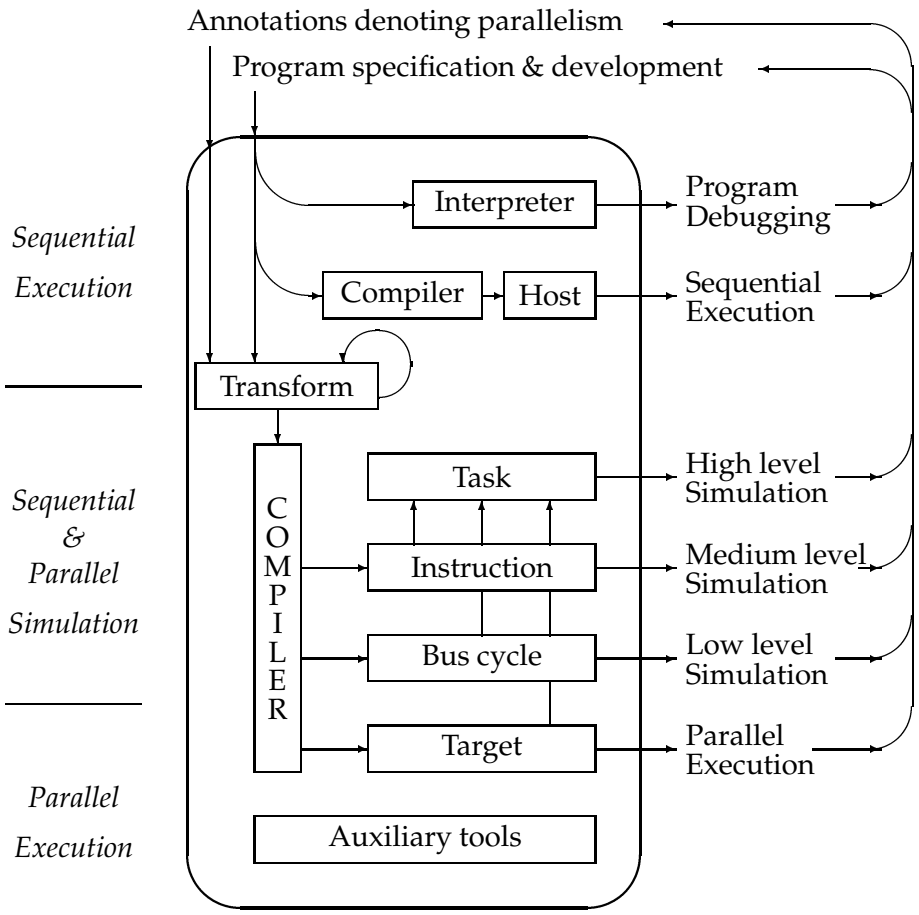


Figure 1: The structure of the toolkit for parallel functional programming.

To assist in the development of parallel functional programs, we have built a toolkit that automatically carries out some of the development steps. The toolkit makes it relatively easy to implement parallel programs. All stages of the software development cycle are supported: design, coding, testing, debugging, and performance analysis. The structure of the toolkit is depicted in Figure 1. Two major steps can be identified in the software development cycle. The first step is the design, coding and debugging of the program on a sequential platform with conventional techniques like profiling and interactive debugging. The second step adds parallelism by converting the sequential version to a correct parallel algorithm. In general, and for imperative programming in particular, this second step is difficult for two reasons. First, the programmer has to develop an intuition for concurrent programming, so that problems such as deadlocks and race conditions can be avoided. Second, parallelism often fails to deliver speedup, as communication and synchronisation tend to require considerable resources.

When using pure functional languages, the situation is not quite as bleak. In such languages, needed subexpressions of the program can be evaluated in *any* order, also in parallel. No matter what order of evaluation has been used, the same answer will always be found. (See Hughes [1] for an explanation of this point). The choice for a functional language removes an important burden for the parallel programmer. Although resource deadlocks such as "stack overflow" can still occur, data access deadlocks and race conditions cannot occur. So there is a good chance that writing parallel functional programs is easier than writing parallel imperative programs.

The other difficulty in writing a parallel program, that of achieving a speedup from parallel execution, still remains. Tools are indispensable to identify performance bottlenecks and for improving the code. Our toolkit provides the necessary utilities to monitor the program during either a simulation run at a chosen level of detail, or during execution on the parallel hardware. The monitor output is interpreted by other parts of the toolkit, to produce human readable output.

The next section presents a brief introduction to lazy functional programming and its implementation. The section "Program development cycle" describes how the toolkit supports the development of parallel functional programs. The various parts of the toolkit are highlighted in the sections: "Sequential execution tools", "Tools and annotations for parallelism", "Simulation tools", and "Parallel execution tools". The Section "Case study: tidal prediction" presents two implementations of an algorithm to predict the tidal flow in an estuary of the North sea. In the case study all the elements of the toolkit are applied to derive a good parallel program. The conclusions are presented in the last section.

2 A brief introduction to lazy functional programming

To make the paper reasonably self contained, the basic principles of lazy functional programming will be introduced with the help of a few examples. Refer to Bird and Wadler [2] for an introduction to functional programming. The function *fac* to compute the factorial of a number is defined as:

$$\begin{aligned} \mathit{fac}\ n &= 1, && \text{if } n = 0 \\ &= n \times \mathit{fac}\ (n-1), && \text{otherwise} \end{aligned}$$

This means that the factorial of n is defined as 1 if n equals 0, or n times the factorial of $n-1$ otherwise. The arguments of a function are separated by spaces, both when defining and applying functions; the brackets around $n-1$ are necessary because function application has a higher priority than subtraction. In the following example the same function is defined using *pattern matching* instead of the if-construct. Also a where-clause is used to introduce a local

definition:

$$\begin{array}{l} \mathit{fac} \quad :: \ \mathit{num} \to \mathit{num} \\ \mathit{fac}\ 0 = 1 \\ \mathit{fac}\ n = n \times \mathit{fac}\ m \text{ where } m \ = \ n - 1 \end{array}$$

The function m (which has no parameter in this example) is defined in the scope of *fac*: it is only visible inside this function definition. It also has access to the variables defined in the surrounding scope (i.e. n).

Function types are specified using the arrow notation $\to$. The type of a function is declared using a double colon. The type of the *fac* function specifies that it maps numbers to numbers. A function mapping a number to a character has the type $\mathit{num} \to \mathit{char}$. A function mapping two characters onto one number has the type $\mathit{char} \to \mathit{char} \to \mathit{num}$ (which should be read as a function mapping a single character to a function mapping a character onto a number).

The *list* is a standard data structure to store an ordered collection of items of the same type. A list is constructed with colons, and terminated with [] (the nil element):

$$2 : 3 : 5 : 7 : []$$

This is the list with the first 4 prime numbers; $[2, 3, 5, 7]$ is an alternative notation for the same list. An exclamation mark is used to select a list element:

$$[2, 3, 5, 7] \ ! \ 2$$

This yields 5 (counting starts at 0). The operator ! can be defined as a recursive function in the same way as the factorial function:

$$\begin{array}{lll} ! & :: \ [\alpha] \to \mathit{num} \to \alpha & \\ (x : \mathit{xs}) \ ! \ n = x, & \text{if } n \ = \ 0 & (!.1) \\ \quad = \mathit{xs} \ ! \ (n-1), & \text{otherwise} & (!.2) \end{array}$$

The type α indicates that the ! operator works for polymorphic lists, i.e. lists containing elements of an arbitrary type. The type $[\alpha]$ specifies a list with elements all of type α. (The asymptotic time complexity of the ! operator is $O(n)$. The list data structure and the associated indexing operator ! are not satisfactory for implementing arrays.)

The evaluation of an expression proceeds lazily, which means that no (part of the) expression is ever evaluated unless the result is required. Consider for example the function *from* which generates a list of numbers of indefinite length, starting with some number n:

$$\begin{array}{ll} \mathit{from} \quad :: \ \mathit{num} \to [\mathit{num}] & \\ \mathit{from}\ n = n \ : \ \mathit{from}\ (n+1) & (\mathit{from}.1) \end{array}$$

Laziness makes it possible to just write down the list of all natural numbers as $\mathit{from}\ 0$, and then to select one particular number say:

$$(\mathit{from}\ 0) \ ! \ 2$$

The lazy implementation makes sure that just enough of the structure of the list is computed to reach the requested element, and then it computes the element. This takes place as a number of separate steps (which are called *reduction* steps) as follows:

$$\begin{array}{lll} (\mathit{from}\ 0)!2 = (0 \ : \ \mathit{from}\ (0+1))!2 & & (\mathit{from}.1) \\ \quad = (\mathit{from}\ (0+1))!1 & & (!.2) \\ \quad = (0+1 \ : \ \mathit{from}\ (0+1+1))!1 & & (\mathit{from}.1) \\ \quad = (\mathit{from}\ (0+1+1))!0 & & (!.2) \\ \quad = (0+1+1 \ : \ \mathit{from}\ (0+1+1+1))!0 & & (\mathit{from}.1) \\ \quad = 0+1+1 & & (!.1) \\ \quad = 1+1 & & (+) \\ \quad = 2 & & (+) \end{array}$$

Here we have annotated each step with the defining equation of the function that was applied.

For an implementation to achieve such a sequence of steps requires three basic ingredients:

- A strategy that determines which expression to evaluate next. In a parallel implementation the strategy decides which expressions can be evaluated in parallel.
- Knowledge about the extent to which expressions are to be evaluated. If an expression representing a data structure is evaluated to *head normal form*, the structure of the data is revealed, but no more. For example, the head normal form of a non-empty list is $\cdots : \cdots$, where $\cdots$ represents an expression that need not necessarily be evaluated. The head normal form of an empty list is $[]$.
- A mechanism that allows unevaluated expressions to be stored so that they can be evaluated later. Such unevaluated expressions are often called *suspensions*.

This covers the aspects of lazy functional programming and its implementations in as far as we shall be needing them in the present paper. The interested reader is referred to Peyton Jones [3].

3 Program development cycle

At the start of the development cycle, the parallel algorithm to be implemented is specified in a functional language called *Intermediate*. Intermediate is a subset of the functional language Miranda[1] [4, 5]. An Intermediate program is debugged with the Miranda interpreter. Although the interpreter does not allow for efficient execution of the program, the development cycle is short when using small input data sets because there is no compilation phase. When the programmer is satisfied that the program is working, the program is compiled using the FAST/FCG [6, 7] compiler for Intermediate. This compiler translates the program into an equivalent C program, which is compiled to the target platform using a C compiler (resulting in code that is portable to any machine with a C compiler). The program can be executed on a sequential machine, and a standard profiler allows the programmer to analyse the performance of the sequential program, and to improve the sequential performance.

The program is then transformed into a parallel version by annotating certain parts as parallel *skeletons* [8]. The program code will probably be modified to improve parallelism. Code generation for the parallel machine is taken care of by the same FAST/FCG compiler. The only difference from the code for the sequential version is that any annotations for parallelism are translated into calls to the runtime support system to create new tasks, and to synchronise tasks on termination. The only machine dependent part of the parallel run time support consists of the code that implements the primitives used for synchronisation, the rest of the code is machine independent.

The translated program can be used to simulate execution of the program on both shared memory machines and distributed memory machines. Simulation can be used to analyse the scalability of the application itself, and simulation also gives important information about the grain size of the program and possible performance bottlenecks. The toolkit offers three levels of simulation: low level (individual bus cycles on a parallel hierarchical shared memory architecture), medium level (instruction level assuming a shared memory with uniform access time) and high level (using a task as the basic simulation entity).

Some auxiliary tools are included with the toolkit to process simulation and execution data. These auxiliary tools allow the programmer to compute the average and maximum parallelism, to graphically depict the performance, and to use several workstations to simulate the execution

[1] Miranda is a trademark of Research Software Ltd.

on different architectures in parallel. We will not describe these auxiliary tools here. Instead we will concentrate on the steps of the development cycle sketched above. These steps are described in the following sections, in the order in which they are used.

4 Sequential execution tools

The relation between the various tools is depicted in Figure 1. Three of the tools, the interpreter, compiler and transformation tool set, use the Intermediate specification of the program. The code generator which is used with the simulators and the target architecture works on a side effect free C program with fork-join parallelism.

4.1 The Miranda system

The Miranda system is used as an interactive program development environment. The system supports modular design and incremental program development through its interpreter. The system is robust, and produces good error messages. Miranda is well supported and well documented [4, 5].

4.2 The FAST/FCG compiler

The FAST compiler [6] together with the FCG code generator [7] implement Intermediate, which is a subset of Miranda. To allow Miranda programs to be compiled by the FAST/FCG compiler, three restrictions must be observed;

- Instead of using the Miranda type *num* which supports arbitrary types of numerals, the Intermediate programmer must explicitly indicate where fixed width integers, floating point numbers etc. are used.
- Miranda supports comparisons on arbitrary data structures, while Intermediate only supports comparisons on integer and floating point numbers. Comparison functions for arbitrary data structures must be programmed explicitly in Intermediate.
- Miranda offers a module system, which is not provided by Intermediate.

The FAST/FCG compiler offers efficient $O(1)$ access time arrays as well as parallel execution. These facilities are not provided by Miranda. Some auxiliary tools are available to assist in converting Miranda programs into Intermediate and work is in progress to further simplify conversions. As the main theme of this paper is on writing parallel programs, we will not say more about programming in Miranda.

5 Tools and annotations for parallelism

In a functional language two basic kinds of parallelism can be distinguished: strict argument parallelism and pipeline parallelism. The former is sometimes referred to as horizontal parallelism, and the latter is also known as vertical parallelism [9]. We first consider pipeline parallelism.

When a function is applied to an argument, the function can be considered as a consumer process, while the argument plays the role of the producer process. Both processes can synchronise and exchange data arbitrarily often. When and how frequently these exchanges take place is application dependent. Consider for example the expression (*square zs*). Here *zs* is a list of numbers, each of which is subsequently squared (the function *square* takes a list of

numbers and returns a list of their squares). In a lazy functional language the list *zs* is not a real list of numbers when the function *square* is applied. In fact it is represented by a "suspension": a piece of code that, when called, will produce a small amount of data plus a new suspension for the rest of the list. Therefore *zs* can be considered as a process that will produce a fraction of the real list whenever asked for by *square* (laziness). Different synchronisation schemes lead to different grain sizes:

fine grain Exchange data whenever *zs* reaches head normal form, i.e. when *zs* has produced the smallest amount of real data plus a new suspension. This real data is not yet the first number, but the list structure containing a suspension for the first number of the list and a suspension for the rest of the list.

medium grain Exchange data whenever *zs* reaches head normal form and the head element of the list is also fully evaluated, i.e. the suspension for the first number in the list *zs* is called and has computed a real number.

coarse grain Exchange data when the list *zs* is fully evaluated (normal form, strict).

Here fine grain and coarse grain are extremes of a spectrum of possibilities.

The first option represents the most natural type of synchronisation in a lazy functional language, because the evaluation mechanism of a lazy language is always aware of head normal forms. However, in general a head normal form occurs frequently during evaluation, and the corresponding data is not sufficiently large to guarantee a substantial amount of computation by the consumer, once the data exchange has taken place. This kind of parallelism is often too fine grained.

Quite the opposite is true in the third case, where synchronisation only happens once, namely when all numbers of the list *zs* have been computed. This effectively amounts to sequential computation on our machine, where we do not have the possibility to use parallel vector operations.

The second case seems to be a reasonable compromise, where communication happens each time a new number in the list is calculated. The computation of the square of this number overlaps with the computation of the next number. That is why this kind of parallelism is frequently called "pipeline" parallelism. The granularity of this kind of parallelism depends on a number of factors. Consider a simple pipeline with a producer and a consumer that run at exactly the same pace. Now a steady stream of data flows through the pipe, requiring communication of data, but no repeated synchronisation of the consumer and producer processes. This is coarse grain parallelism, because both producer and consumer run at full speed without waiting for each other. Process synchronisation is required when the producer and the consumer do not run at the same pace, thus making the grain size smaller. If this form of pipeline parallelism is too fine grained, data should be communicated in larger chunks.

The second source of parallelism, strict argument parallelism, originates from evaluating several arguments of a function in parallel. If the function really needs the value of two or more of its arguments, these values can be computed in parallel. At the same time, it is still possible to exploit pipeline parallelism, because in the course of computing the arguments the function itself may be able to do some work on partly evaluated data.

The toolkit supports restricted forms of strict argument parallelism and of pipeline parallelism. A fork-join annotation, called *sandwich* [10], is used to indicate strict argument parallelism. A synchronous process network skeleton is the annotation for pipeline parallelism. The name *sandwich* reflects the way in which one layer of eager, applicative order (parallel) evaluation is sandwiched between two layers of lazy, normal order evaluation.

The sandwich annotation causes the evaluation of two or more subexpressions in parallel (the *fork*), and the application of a *join* function to the resulting values. The choice we have made

in the implementation of the sandwich annotation is to support only coarse grain synchronisation between function and arguments. The synchronisation is thus based on fully evaluated expressions. This choice was made to simplify the implementation, avoiding repeated synchronisation between producer and consumer. The sandwich annotation is used naturally for parallelisation of divide and conquer algorithms. So with the sandwich it is possible to annotate coarse grain parallel programs that do not contain pipeline parallelism.

Synchronous process networks are supported by transformation into fork join programs. The necessary program transformations are carried out automatically, once the programmer has indicated which networks must be transformed. Work is also in progress to support some forms of asynchronous process networks.

The combination of annotating parallelism and using a purely functional language has an important benefit: once a satisfactory sequential version of a problem has been implemented, no debugging is required to make the parallel version also work. Independent of whether the implementation is sequential or parallel, the program will always result in the same output. This is in contrast with the parallelisation of imperative and object oriented languages, where parallelisation may introduce non-determinism, which means that when using such languages, timing aspects have to be taken into account.

5.1 Transformation tool set

The basic annotation that denotes the parallel execution of tasks is the *sandwich* annotation, which has the following form:

$$\mathit{sandwich}\ f\ (g\ x_1\ x_2 \ldots)\ (h\ y_1\ y_2 \ldots) \ldots$$

The *sandwich* annotation does not affect the meaning of the annotated expression, in the sense that the following equation holds:

$$\mathit{sandwich}\ f\ (g\ x_1\ x_2 \ldots)\ (h\ y_1\ y_2 \ldots) \ldots\ =\ f\ (g\ x_1\ x_2 \ldots)\ (h\ y_1\ y_2 \ldots) \ldots$$

The *sandwich* annotation causes $(g\ x_1\ x_2 \ldots)$, $(h\ y_1\ y_2 \ldots)$ etc. to be evaluated in parallel. The results of the parallel tasks are combined using the function f.

The *sandwich* is used naturally to denote the parallelism in divide and conquer programs. In small programming examples, such as quick sort, the annotation is always easy to apply because the structure of the programs is simple. Realistic application programs tend to hide the parallelism, and are therefore not trivially annotated. For these programs several transformations have been designed that restructure programs with other types of parallelism into programs suitable for *sandwich* annotations.

- Communication lifting [11] transforms synchronous process networks [12] (used for geometric parallelism and synchronous pipelines) into a divide and conquer structure. This opens a large class of applications.
- The grain size transformation [10] can be used to enlarge or reduce the grain size of parallel computations so that the grain size and the architecture can be matched.

The case study to be presented later describes in detail how communication lifting is used.

Parallel vector operations are not directly supported by the sandwich. However, it is possible to apply the divide and conquer paradigm to vector computations. As an example consider the definition of *parmap*, which applies a certain function f to all elements of a list *zs* in parallel:

$$\begin{aligned}
&\mathit{parmap} && :: (\alpha \to \beta) \to [\alpha] \to [\beta] \\
&\mathit{parmap}\ f\ [] && = [] \\
&\mathit{parmap}\ f\ (z : zs) && = \mathit{sandwich}\ (:)\ (f\ z)\ (\mathit{parmap}\ f\ zs)
\end{aligned}$$

The sandwich annotation causes $(f\ z)$ to be evaluated in parallel with $(parmap\ f\ zs)$, which recursively forks subsequent parallel applications.

6 Simulation tools

The toolkit offers simulation at three levels of abstraction: task level, instruction level and bus cycle level. The highest level simulator can be used to obtain a rough impression of the performance figures within reasonable time. The lowest level simulator gives detailed results, at the expense of long simulation times. Because all three simulators are part of one integrated environment, it is possible to start with high level simulations of program and machine configurations, and zoom in on particular instances using the low level detailed simulation. No conversion efforts are required to switch between simulations.

6.1 Task level simulation

The input of the task level simulator consists of a task graph and an architecture description. A node of the task graph represents a task: the node is annotated with the execution time needed by the task, and by the size of the data structure that represents the task. All this information is produced by the instruction level simulation. The (directed) edges of the task graph show the fork join relations, a task that executes a sandwich with two expressions has two edges to the (newly generated) tasks, the join is represented with two edges to a new task that executes the join function. As an example, Figure 2 shows the task graph of a merge sort program that first needs about 4 milliseconds to generate the data set, then forks twice, sorts in parallel in four leaf tasks for about 25 milliseconds, and joins in two phases. In the merge sort algorithm, the second join phase needs twice the time as each of the join tasks in the first phase, corresponding to a merge operation of a list that is twice as long.

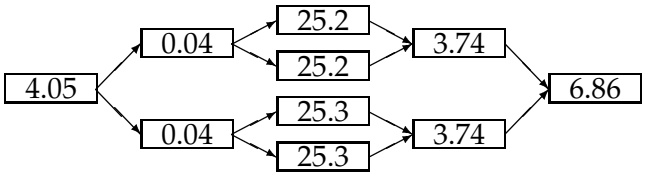


Figure 2: An example task graph of a merge-sort program. The nodes represent tasks and the edges represent the dependencies between the tasks. The figures denote the execution time of a task (in milliseconds).

The architecture specification for the task level simulation supports shared memory machines, distributed memory machines, and cluster machines, where the nodes of a distributed memory machine consist of shared memory multiprocessors [13]. The essential parameters to be specified are the number of processors, an interconnection topology for a distributed memory architecture, the number of nodes accessing a shared memory, cost parameters for network transport and overhead estimates for forking and joining. The task level simulator estimates the execution time of the task graph on the architecture in a time that is proportional to the number of tasks. The task level simulator is substantially faster than actual execution or simulation at a lower level.

The resources that are modelled by the task level simulator are processing and networking. Low level sources of contention are not modelled. These include memory contention and

contention in the run time support system. Garbage collection is part of the normal execution of a task. The timings predicted by the simulator are less accurate than the prediction by the medium level simulator, but it runs orders of magnitude faster, and it supports a wide variety of architectures.

Because contention on software resources is not modelled, the timing of the task level simulator can be optimistic in the case of heavy congestion. Usually the timing from the task level simulator is not more than 5–10% optimistic in comparison with the instruction level simulator. The error is even lower if the program is irregular and has sufficient parallelism.

6.2 Instruction level simulation on a shared memory machine

The instruction level simulator generates more detailed information than the task level simulator. The instruction level simulator models parallel execution of the program on a shared memory machine at the granularity of individual instructions. The instruction level simulator has two components: the first actually executes the parallel program and while executing, it generates an execution trace. This execution trace is used as input to the second component, which simulates a shared memory architecture with uniform memory access costs. The instruction level simulator assumes that accessing data or instructions delays the processor for one instruction cycle. The simulator outputs a break down of the execution time on the architecture in the following categories: the effective user time (seconds spent doing computations for the user's program code), the time spent in the run time support code (garbage collection, process management) and the time spent in software synchronisations.

The instruction level simulator is an order of magnitude slower than real execution, but it can be used to experiment with architecture parameters such as the number of processors and the average cost of memory accesses. In contrast with the simulation at task level, the program is actually executed. This simulator can generate a task graph to be used for the task level simulation. The time needed for software synchronisations and run time support is taken into account: a program that frequently accesses a shared resource (such as for example the memory allocator) will serialise on this resource. This effect is visible with the instruction level simulator, the task level simulator abstracts from this type of performance loss.

The instruction level simulator is particularly useful to experiment with software aspects in both the application and the run time support code in relation to the number of processors in the architecture. It does not properly model the memory architecture, and is consequently giving optimistic timing figures with respect to real hardware. The error is architecture dependent. For machines with few processors the simulator will predict 20–30% optimistic timings. For large physical shared memory machines the error is potentially unbounded, because the memory becomes a performance bottleneck. For scalable virtual shared memory machines the error will increase slowly with the number of nodes.

6.3 Bus cycle level simulation

The lowest level simulator models the hardware in full detail. Instead of assuming uniform memory access costs, as with the instruction level simulator, the execution of each transaction is simulated at bus cycle level. The bus cycle level simulator correctly models delay caused by the memory hierarchy, and contention on the memory hierarchy. As is the case with the instruction level simulator, an execution trace is generated, which is used to drive a shared memory simulator [14] that simulates (hierarchical) shared memory architectures with caches based on the Futurebus protocol. The architecture specification for this simulator consists of the number of levels of the architecture (one for a simple flat bus architecture), the cache parameters at the various levels of the hierarchy (line size, associativity, cache size), and the

speeds of caches, busses and memory.

The bus cycle level simulator runs orders of magnitude slower than the instruction level simulator. It is particularly useful to analyse the program on cached architectures, and to determine the scalability effects of hardware parameters. The simulator models all shared resources (in hardware and software), and is accurate within 15%. Because of the huge simulation costs, this level of detail is normally not used during the development cycle. The bus cycle level simulator is indispensable for detailed scalability analysis.

7 Parallel execution tools

An integrated component of the toolkit is a facility to execute a program on a parallel platform. At this moment only the execution on a four processor Motorola Hypermodule has been implemented. The parallel execution platform is small, but useful to verify the accuracy of the simulations on 1–4 processor systems. The Hypermodule is a board with four 88100 processors (25 Mhz), each with a 16 Kb instruction cache and a 16 Kb (coherent) data cache, connected via a shared bus with 64 MBytes of main memory. The board is mounted in a VME rack, and accompanied by a Philips 68020 board that takes care of booting, code loading, I/O implementation (over Ethernet) and performance monitoring. The Ethernet connection enables the use of the Hypermodule from any workstation on the network.

Remember that the code that runs on the Hypermodule machine is almost the same as the code run by the simulators. The only exception is the implementation of the semaphores: they are implemented using the indivisible exchange memory instruction (`xmem`). The implementation on the shared memory Hypermodule allows us to execute the program (the final goal of program development), and also allows us to measure the performance of the program. The resulting figures are, in contrast with figures from the simulator, irrefutable, since they are measured on a real system. However, there is no possibility to experiment with architecture parameters, except restricting the number of processors used.

8 Case study: Tidal prediction

When using a small or artificial problem, the hard aspects of parallel programming are easily overlooked. To show the power of the toolkit, the development of a realistic example problem of a reasonable size will be presented. This case study is based on previous work [15, 10], which explains in some detail the physical background of the application, as well as presenting a systematic derivation of the parallel implementation in a lazy functional language. To make the paper reasonably self contained, we discuss the essential aspects of the application domain, which almost immediately leads to a formulation of a parallel implementation. For a comprehensive treatment of the application domain, the reader is referred to Heemink's thesis [16].

8.1 The algorithm

The prediction of the tides in the North Sea and its estuaries are based on the *shallow water equations*. The linearised and slightly simplified versions of these equations [16] are shown below:

$$\frac{\partial u}{\partial t} + g\frac{\partial h}{\partial x} - fv + \lambda\frac{u}{D} - \gamma\frac{V^2\cos\psi}{D} = 0 \quad (1)$$

$$\frac{\partial v}{\partial t} + g\frac{\partial h}{\partial y} + fu + \lambda\frac{v}{D} - \gamma\frac{V^2\sin\psi}{D} = 0 \quad (2)$$

$$\frac{\partial h}{\partial t} + \frac{\partial (Du)}{\partial x} + \frac{\partial (Dv)}{\partial y} \quad = \quad 0 \tag{3}$$

h =small variations in the water height
D=depth of the water as a function of x and y
f =Coriolis parameter($1.25 \times 10^{-4} s^{-1}$)
γ =wind friction(3.2×10^{-6})
ψ =wind direction
u =water velocity in the X direction
v =water velocity in the Y direction
g =acceleration of gravity($9.8 ms^{-2}$)
λ =bottom friction($2.4 \times 10^{-3} ms^{-1}$)
V=wind velocity

Equations 1 and 2 represent the conservation of momentum and Equation 3 represents the conservation of mass. The effects of the earth rotation (f), the friction with the bottom (λ) and the friction with the wind (γ) are taken into account. The effect of the atmospheric pressure has been ignored because it is relatively small.

A numerical approximation for the partial differential Equations 1–3 is given in the finite difference scheme of Equations 4–6 below. For a formal derivation see van der Houwen [17]. The variables u, v and h in Equations 1–3 are approximated in Equations 4–6 by values $u^k_{i,j}$, $v^k_{i,j}$ and $h^k_{i,j}$ on a spatial grid at discrete points in time. The grid coordinates are i and j and the superscript k represents the discrete time steps. The depth D does not vary in time ($h^k_{i,j}$ is varying), hence no superscript k is required. When $\triangle x$ and $\triangle y$ are the distance steps in the grid, the relations between the true and approximated height and velocities are:

$$u^k_{i,j} \approx u\ ((2i-1)\triangle x, 2j\triangle y, \quad k\triangle t) \qquad h^k_{i,j} \approx u\ \ (2i\triangle x, 2j\triangle y,\ k\triangle t)$$
$$v^k_{i,j} \approx v\ (2i\triangle x, \quad (2j-1)\triangle y, k\triangle t) \qquad D_{i,j} = D\ (2i\triangle x, 2j\triangle y)$$

The spatial grids for u, v and h are slightly shifted with respect to each other, in about the same way as the red, green and blue dots that correspond to the pixels of a colour monitor. This alignment of the matrices is called a space staggered grid, and it has important advantages for the stability of the computations. Because of the choice of a space staggered grid the equations for $u^k_{i,j}$ and $v^k_{i,j}$ are asymmetrical:

$$\begin{aligned} u^{k+1}_{i,j} &= u^k_{i,j} - g\frac{\triangle t}{2\triangle x}(h^k_{i,j} - h^k_{i-1,j}) + f\frac{\triangle t}{4}(v^k_{i-1,j} + v^k_{i-1,j+1} + v^k_{i,j} + v^k_{i,j+1}) \\ &\quad -2\triangle t\frac{\lambda u^k_{i,j} - \gamma V^2 \cos\psi}{D_{i,j} + D_{i,j+1}} \end{aligned} \tag{4}$$

$$\begin{aligned} v^{k+1}_{i,j} &= v^k_{i,j} - g\frac{\triangle t}{2\triangle y}(h^k_{i,j} - h^k_{i,j-1}) - f\frac{\triangle t}{4}(u^{k+1}_{i,j-1} + u^{k+1}_{i+1,j-1} + u^{k+1}_{i,j} + u^{k+1}_{i+1,j}) \\ &\quad -2\triangle t\frac{\lambda v^k_{i,j} - \gamma V^2 \sin\psi}{D_{i,j} + D_{i+1,j}} \end{aligned} \tag{5}$$

$$\begin{aligned} h^{k+1}_{i,j} &= h^k_{i,j} - \frac{\triangle t}{4\triangle x}\left((D_{i+1,j} + D_{i+1,j+1})u^{k+1}_{i+1,j} - (D_{i,j} + D_{i,j+1})u^{k+1}_{i,j}\right) \\ &\quad -\frac{\triangle t}{4\triangle y}\left((D_{i,j+1} + D_{i+1,j+1})v^{k+1}_{i,j+1} - (D_{i,j} + D_{i+1,j})v^{k+1}_{i,j}\right) \end{aligned} \tag{6}$$

The overall computation of the approximations to the physical quantities of interest at a particular time step proceeds as follows: start with initial values at step $k = 0$, then compute the values of the next iteration at $k = 1$ etc. until the values of the k-th iteration have been computed.

The finite difference scheme presented is part of a large sequential Fortran program that is used at the Dutch Water Board to predict storm surges in estuaries of the North Sea. The explicit finite difference scheme and the particular choice for a space staggered grid have three important advantages for the structure of the implementation:

1. The calculations in each grid point only require neighbouring values, both in space and in time. For instance, the value of v in Equation 1 is approximated in Equation 4 by an average of four neighbours $(v^k_{i-1,j} + v^k_{i-1,j+1} + v^k_{i,j} + v^k_{i,j+1})/4$. The finite difference scheme thus leads to the locality that allows parallel computation to be used effectively.

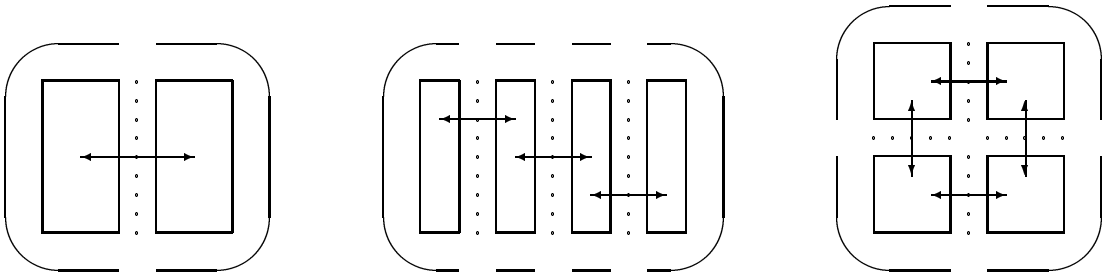
2. The finite difference schemes gives an order in which new approximations can be calculated from previous values: first compute all the $u^{k+1}_{i,j}$, then compute all the $v^{k+1}_{i,j}$ and finally all the $h^{k+1}_{i,j}$. This sequence must be repeated until the required time frame has been reached. This sequence has a good stability, which is important for numerical approximations.

3. The choice of a space staggered grid makes it possible to store the velocities and heights using only three arrays: one for each of the $u^k_{i,j}$, $v^k_{i,j}$ and $h^k_{i,j}$. These arrays can be updated in place, provided the computations are sequenced as discussed above. We shall not use this particular property here. In our earlier work [18] ways are discussed of using various techniques to embed destructive updates properly in a purely functional language. We also describe applications of some of the techniques to improve the performance of various programs.

We will exploit these advantages during the development of two implementations of the problem.

8.2 The data decomposition

A good way to build a parallel implementation of the tidal prediction problem is by dividing the spatial grid into partitions, such that each available process performs the computations required for one partition. The geometry and the size of such a partition determines the grain size of the parallel decomposition. Figure 3 shows three partitions of the space staggered grid. Partition (a) divides the grid into two equal parts in the X direction. The arrows represent the communication of border vectors between the two partitions. Partition (b) divides the grid in four parts along the X direction. Partition (c) divides the grid in both directions. As the number of partitions increase, also the number of communications increase. It should be noted that only neighbouring partitions exchange vectors of border data, as indicated by the dotted lines in Figure 3. In the performance evaluations, the behaviour of grids with up to 16 partitions will be studied. We will use the 2×1 partition (a), to discuss the principles of the grid partitioning.



(a) $X \times Y = 2 \times 1$ (b) $X \times Y = 4 \times 1$ (c) $X \times Y = 2 \times 2$

Figure 3: Three simple regular partitions of the space staggered grid. The boxes represent the partitions, the sequences of dots represent border vectors and the arrows represent the communication of the border vectors.

In the actual implementation we have parametrised the knowledge of how matrices are decomposed. There is only one program for all partitionings. In the case of a static process network this one parametrised program is in fact a program generator, which computes a process network for each partitioning.

Having looked at the data decomposition in general terms, it is useful to consider suitable representations of the grid matrix: as a row of columns, as a column of rows, as a quad tree [19] or otherwise. From the point of view of partitioning the matrix for parallelism, a quad tree or similar structure seems sensible. However, this would mean that the nearest neighbour operations that are performed within the partitions, would not be well supported. The nearest neighbour operations represent the majority of the work, so they must be supported efficiently. This calls for either a column of rows or a row of columns representation. Our compiler supports one dimensional arrays. This is sufficient for constant time access and monolithic array creation. Two dimensional arrays are implemented as a small number of library functions. Changing one representation of two dimensional arrays into another only requires rewriting these library functions. So we were able to measure which representation performs best, column major order or row major order (see Section 8.6).

Another matter to consider is how to implement the communication. In particular: given a partitioning and a representation of the spatial grid as a matrix, how can a process be provided with efficient access to matrix elements on borders allocated to a neighbouring process?

There are basically two options: the first is to partition and reassemble the grid matrix at each iteration. The second option is to partition the grid once, and to allocate the partitions on a permanent basis to the different parallel processes. After the required number of time steps has passed, the partitions are reassembled to form the final result of the program. In both cases the load of each process is fixed because the amount of work involved in computing a subsequent state of a grid matrix partition is independent of the actual data values. The partitioning as well as the scheduling for this problem can thus be done statically.

The first method is based on the *process farm*: each process has access to the entire grid. During an iteration, no communication needs to take place, because each process can freely access the grid points on the borders. Communication takes place only when a new matrix is assembled out of the individual partitions, ready for the next iteration. This method is not scalable to a system with a large number of processors because it requires all processes to have access to one global data structure.

The basis of the second partitioning method is a *process network*: throughout the computation, each process remains responsible for a partition of the grid, and exchanges messages with neighbouring processes to access values at the borders of the partitions. This method exhibits more locality, and should thus be more scalable than the farm partitioning method.

In both cases a similar form of synchronisation is required between the parallel processes. The farm approach and the network approach will be compared, where both are using the row of columns representation. The approaches have specific advantages and disadvantages, which will be explored, while keeping the remaining code the same wherever possible. In particular the implementation of the point update functions corresponding to Equations 4–6 should be the same for both methods. Keeping matters separate where possible makes the implementation more flexible, and thus facilitates experimenting with different ways of building a parallel implementation.

8.3 Data distribution based on the farm approach

The two-way parallel farm version of the tidal prediction program is shown in Figure 5. The program has an iterative structure, which is indicated by the use of the function *iterate* (see Figure 4 for the definition of this function and other useful stream processing functions).

$$
\begin{array}{ll}
\mathit{hd} & :: [\alpha] \to \alpha \\
\mathit{hd}\ (z : zs) & = z \\
 & \\
\mathit{tl} & :: [\alpha] \to [\alpha] \\
\mathit{tl}\ (z : zs) & = zs \\
 & \\
\mathit{map} & :: (\alpha_1 \to \ldots \alpha_n \to \beta) \to [\alpha_1] \to \ldots [\alpha_n] \to [\beta] \\
\mathit{map}\ f\ zs_1\ \ldots\ zs_n & = x : \mathit{map}\ f\ (\mathit{tl}\ zs_1) \ldots (\mathit{tl}\ zs_n) \\
 & \quad \text{where} \\
 & \quad x\ =\ f\ (\mathit{hd}\ zs_1) \ldots (\mathit{hd}\ zs_n) \\
\mathit{iterate} & :: (\alpha \to \beta_1 \to \ldots \beta_n \to \gamma) \to \alpha \to [\beta_1] \to \ldots [\beta_n] \to [\gamma] \\
\mathit{iterate}\ f\ x\ zs_1\ \ldots\ zs_n & = x : \mathit{iterate}\ f\ y\ (\mathit{tl}\ zs_1) \ldots (\mathit{tl}\ zs_n) \\
 & \quad \text{where} \\
 & \quad y\ =\ f\ x\ (\mathit{hd}\ zs_1) \ldots (\mathit{hd}\ zs_n) \\
\mathit{zip} & :: [\alpha_0] \to \ldots [\alpha_n] \to [(\alpha_0, \ldots \alpha_n)] \\
\mathit{zip}\ zs_0\ \ldots\ zs_n & = x : \mathit{zip}\ (\mathit{tl}\ zs_0) \ldots (\mathit{tl}\ zs_n) \\
 & \quad \text{where} \\
 & \quad x\ =\ (\mathit{hd}\ zs_0, \ldots, \mathit{hd}\ zs_n)
\end{array}
$$

Figure 4: Definitions of some stream processing functions. The value of the subscript n may be any natural number, including 0.

$$
\begin{array}{ll}
zs & :: [\beta] \\
zs & = \mathit{iterate}\ \mathit{nextstate}\ i \\
 & \\
\mathit{nextstate} & :: \alpha \to \beta \\
\mathit{nextstate}\ t & = h \\
 & \quad \text{where} \\
 & \quad h = \mathit{sandwich\ combine}\ (\mathit{gh}\ v\ \mathit{dleft})\ (\mathit{gh}\ v\ \mathit{dright}) \\
 & \quad v = \mathit{sandwich\ combine}\ (\mathit{gv}\ u\ \mathit{dleft})\ (\mathit{gv}\ u\ \mathit{dright}) \\
 & \quad u = \mathit{sandwich\ combine}\ (\mathit{gu}\ t\ \mathit{dleft})\ (\mathit{gu}\ t\ \mathit{dright})
\end{array}
$$

Figure 5: Two-way parallel farm version of the tidal prediction program. The functions *gu*, *gv* and *gh* perform the computations implied by the finite difference scheme. i is the initial state of the computations.

The computation of Figure 5 produces a stream *zs* of successive states of the grid. The initial state i of the grid appears as the first element of the output stream *zs*. The function *nextstate* is applied to this grid, yielding the state of the grid after one time step. This state is in turn fed to *nextstate*, to yield the state of the grid after two times steps etc.

The functions *gu* and *gv* compute the water velocities in the X and Y directions. These functions are the implementations of Equations 4 and 5 respectively. The function *gh* computes the water heights. This function is the implementation of Equation 6. The height and velocity values are stored in three separate matrices, so that each of the three functions works on its own matrix. The variables i, t, u, v and h all represent 3-tuples of matrices.

The *sandwich* annotation indicates that the functions *gh*, *gv* and *gu* are evaluated in parallel. The arguments *dleft* and *dright* are matrix descriptors, for the left and right halves of the grid respectively. The functions *gh*, *gv* and *gh* thus know for which part of the matrices new values must be computed. All three functions have access to the complete current grid, but produce only part of the new grid. The *combine* function assembles the new grid out of the parts.

There are no fundamental difficulties in partitioning the grid into more, smaller sections. The performance measurements pertaining to this version will be discussed as soon as we have described the second, network version of parallel evaluation.

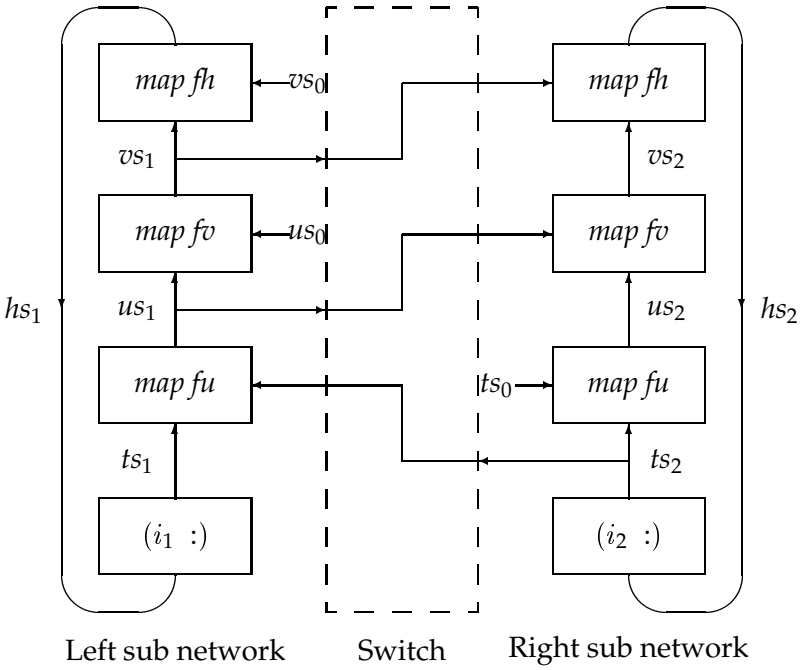


Figure 6: A process network with two cyclic sub networks and a connecting switch. The boxes represent processes and the arrows represent the flow of information between the processes. The external streams carrying 0 values are identified with a subscript 0

8.4 Data distribution using a process network

With the data distribution based on the farm approach, the data communication is completely hidden in the functions *gu*, *gv* and *gh*. In the network version of the tidal prediction program the communication is explicit in the form of connections between the processes, such as shown in Figure 6. The data flowing along the edges represents 3-tuples of matrices and/or vectors of height and velocity values. The matrices are each cut in half and the left halves are processed

$$
\begin{array}{ll}
hs_1 = map\ fh\ vs_1\ vs_0 & hs_2 = map\ fh\ vs_2\ vs_1 \\
vs_1 = map\ fv\ us_1\ us_0 & vs_2 = map\ fv\ us_2\ us_1 \\
us_1 = map\ fu\ ts_1\ ts_2 & us_2 = map\ fu\ ts_2\ ts_0 \\
ts_1 = i_1 : hs_1 & ts_2 = i_2 : hs_2
\end{array}
$$

Figure 7: The tidal prediction program as a synchronous process network. The external streams carrying 0 values are identified with a subscript 0

by the left sub network, whereas the right halves are processed by the right sub network. The difference with the farm approach is that the grid is partitioned once only, whereas the farm approach partitions the grid on every iteration.

The processes $(map\ fu)$, $(map\ fv)$ and $(map\ fh)$ represent the computations implied by Equations 4–6 respectively. Here *map* applies the function passed as its first argument (e.g. *fh*) to all elements of the remaining arguments, which must be streams. The processes $(i_1 :)$ and $(i_2 :)$ insert initial matrices (i_1 respectively i_2) on their output streams and propagate subsequent inputs unaltered.

The program of Figure 7 shows the implementation of the process network of Figure 6. The functions *fu*, *fv* and *fh* are essentially the same as the functions *gu*, *gv* and *gh* in figure 5. The only difference is that *fu*, *fv* and *fh* contain some code for the explicit communication of matrix-borders between the two partitions. The following relation exists between *gu* and *fu*:

$$
\begin{array}{ll}
gu\ t\ dleft & = fu\ (hd\ ts_1)\ (hd\ ts_2) \\
gu\ t\ dright & = fu\ (hd\ ts_2)\ (hd\ ts_0)
\end{array}
$$

Similar equivalences hold for *gv*, *fv* and *gh*, *fh*.

As indicated in Figure 6, some of the inputs of a sub network are provided by the sub network itself, some are provided by the switch, and the remainder receive zero values. These zero values are provided by external streams, which are identified with a subscript 0.

The sequencing of the computations in a process network is indicated by the arrows; the repetition is shown by the back edge. Figure 6 shows two sub networks of processes, each of which is responsible for a partition of the spatial grid. The processes are connected via a switch for the communication. Because of the locality in the finite difference scheme only the vectors that represent the borders rather than complete matrices need to be exchanged. The communication between processes is thus explicit in the structure of the network. This is in contrast with the implicit communication structure used in the farm approach.

The number of processors that may usefully be employed depends on several factors, including the size and geometry of the grid, the cost of the traffic through the switch and the cost of generating and distributing boundary vectors. In the next section we will show how a process network can be implemented efficiently on a divide and conquer architecture, and then move on to discuss the performance of the various implementations of the tidal prediction program.

8.5 From a process network to a divide-and-conquer program

A process network, such as that shown in Figure 6, can be represented by a set of recursive equations [12]. The processes in the network correspond to the functions used in the equations. The variables used in the equations represent the connections in the network. The equations representing the network of Figure 6 are shown in Figure 7. The switch has been configured to provide only those connections that are necessary for the two-way parallel process network. The configuration of the switch is reflected in the naming of the parameters for the *map* functions

in Figure 7. A program generator is used to create the switch for other, more finely partitioned configurations.

The set of equations in Figure 7 form a proper lazy functional program. Lazy evaluation, through the execution and building of suspensions, is completely automatic, but it is not without a cost: each element in each stream requires executing a suspension and constructing a new one. The cost can be kept low when the network is synchronous, which is the case in the tidal prediction program, and in other kinds of applications as well. In a synchronous network, executing one suspension will cause all suspensions on connected streams to be executed as well. All such closely related suspensions are said to belong to the same *generation*. It should thus be possible to revive and execute all suspensions in the same generation at the same time. This is indeed the case if all computations in one generation are tied together (in a way described below) and if all streams are advanced by one generation at the same time. The network of Figure 6 is such a synchronous network.

$$
\begin{array}{lll}
zs & :: & [(\beta_1, \beta_2)] \\
zs & = & \textit{iterate nextstate } (i_1, i_2) \\
 & & \\
\textit{nextstate} & :: & (\alpha_1, \alpha_2) \rightarrow (\beta_1, \beta_2) \\
\textit{nextstate } (t_1, t_2) & = & (h_1, h_2) \\
 & & \text{where} \\
 & & h_1 = \textit{fh } v_1\ v_0 \qquad\qquad h_2 = \textit{fh } v_2\ v_1 \\
 & & v_1 = \textit{fv } u_1\ u_0 \qquad\qquad v_2 = \textit{fv } u_2\ u_1 \\
 & & u_1 = \textit{fu } t_1\ t_2 \qquad\qquad u_2 = \textit{fu } t_2\ t_0
\end{array}
$$

Figure 8: The tidal prediction program after the communication lifting transformation. The operands v_0, u_0, and t_0 represent zero values.

The joint management of all suspensions in a synchronous network can be performed as follows: gather all computations in the network that can be performed at the same time in one tuple. The *zip* (see also Figure 4) of all streams in a network emanating from an expression of the form $\cdots : \cdots$ is a single stream, such that the original stream elements of one generation are gathered in one *state tuple*. The network as a whole will compute generation after generation, while managing only a single stream of tuples. Within a generation, the original stream elements are now accessible to the functions that used to operate on stream elements as the elements of one large tuple. The communication lifting transformation [11] has been developed to change a synchronous network into such a "joint management" form. Figure 8 presents the result of the transformation; a semi formal derivation of this result is given in the appendix. The transformation is described in full in our earlier work [11], where we also present a correctness proof and performance measurements using various other application programs. One of the ingredients of the toolkit is an automatic tool to transform programs with the form shown in Figure 7 to programs with the form as shown in Figure 8. This transformation tool is specifically geared towards certain kinds of process networks, but it is not specific to the Tidal prediction program. The restrictions are merely imposed to enable the tool to operate without user intervention, once the user has identified the network to be transformed.

The final step towards the second parallel version of the tidal prediction is to insert *sandwich* annotations. The expressions that are to be evaluated in parallel appear in six different equations under the "where" clause of *nextstate*. Pairing the equations two by two, so as to form tuples (h_1, h_2), (v_1, v_2) and (u_1, u_2), and inserting the *sandwich* annotation yields the *lifted network* version of the tidal prediction program shown in Figure 9. This final step requires human intervention because the transformation system does not know which expressions are worth evaluating in parallel. The automatic transformation system does the important preparations, by gathering all the expressions that offer potential parallelism.

$$
\begin{array}{ll}
\mathit{zs} & :: [(\beta_1, \beta_2)] \\
\mathit{zs} & = \mathit{iterate}\ \mathit{nextstate}\ (i_1, i_2) \\
 & \\
\mathit{nextstate} & :: (\alpha_1, \alpha_2) \to (\beta_1, \beta_2) \\
\mathit{nextstate}\ (t_1, t_2) & = (h_1, h_2) \\
 & \quad \text{where} \\
 & \quad (h_1, h_2) = \mathit{sandwich\ pair}\ (\mathit{fh}\ v_1\ v_0)\ \ (\mathit{fh}\ v_2\ v_1) \\
 & \quad (v_1, v_2) = \mathit{sandwich\ pair}\ (\mathit{fv}\ u_1\ u_0)\ (\mathit{fv}\ u_2\ u_1) \\
 & \quad (u_1, u_2) = \mathit{sandwich\ pair}\ (\mathit{fu}\ t_1\ t_2)\ \ (\mathit{fu}\ t_2\ t_0) \\
 & \\
\mathit{pair} & :: \alpha_1 \to \alpha_2 \to (\alpha_1, \alpha_2) \\
\mathit{pair}\ x_1\ x_2 & = (x_1, x_2)
\end{array}
$$

Figure 9: Transformed process network version of the tidal prediction program with annotations for parallelism.

The overall execution of the lifted network program proceeds as follows: start with an initial state (i_1, i_2) and *iterate* over the state transformation function (*nextstate*). The *iterate* function controls the succession of generations of state tuples and thereby captures all the recursion originally scattered throughout the network. Given the tuple produced by the previous generation, the *nextstate* function calculates the tuple of the current generation. The overall result of this program is thus again a stream *zs*, which produces successive states of the grid, starting with the initial state (i_1, i_2).

The parallel evaluation of expressions is automatically scheduled by the implementation (of a lazy functional language), as follows: first the expressions $(\mathit{fu}\ t_1\ t_2)$ and $(\mathit{fu}\ t_2\ t_0)$ are evaluated in parallel, followed by the next two parallel jobs, $(\mathit{fv}\ u_1\ u_0)$ and $(\mathit{fv}\ u_2\ u_1)$. Finally the jobs $(\mathit{fh}\ v_1\ v_0)$ and $(\mathit{fh}\ v_2\ v_1)$ are computed in parallel.

8.6 Task level simulation of the farm and lifted network versions

Starting from two different points of view, we have derived two implementations of the tidal prediction, which are similar in appearance, but different in the way the programs communicate. Both implementations use *iterate* to deliver a stream of grid matrices (Figures 5 and 9), and both use the *sandwich* annotation to generate divide-and-conquer parallelism. The difference lies in the way border vectors are accessed: implicit in the case of the farm version versus explicit in the case of the (lifted) network version.

The farm version of the tidal prediction program has two parameters (X and Y, see below), which define a partitioning of the grid. By selecting appropriate values for these parameters, speedups can be measured on various configurations. The lifted network version does not have such parameters, instead a specialised version is generated for each partitioning.

The preparations so far offer the possibility to experiment with different versions of the same program. Because of the speed of the instruction level simulator, it is indeed feasible to perform an extensive search of the architecture design space. The results of the search are shown in Table 1. The table is made up of three major columns and five major rows, each containing a small table. The left major column marked *Farm* shows the speedup figures for the farm version on architectures with 1, 2, 4, 8 and 16 processors. The second and third major columns show the speedup obtained with the *Lifted network* versions. The second major column applies to machines with 1, 2, 4, 8 and 16 processors, while the third major column shows the performance figures for the lifted network versions on some architectures with a number of processors that is not a power of 2: for 3, 6 and 12 processors.

Each "small" table shows that fifteen different X and Y parameter combinations have been

used. Both X and Y values were varied between 1 and 16, but only programs that split into 16 or fewer jobs have been taken into account. Otherwise programs would be too fine grained to benefit from parallel execution with the problem size of 64×64 grid points that we have used.

The speedup figures are given relative to the fastest implementation (according to the task level simulator) on a single processor. This is the single processor farm version of program (i.e. the 1×1 version that does not fork). This version needs 22.0 seconds to complete execution. As an example, the task level simulator predicts that the speedup of the lifted network program cut into 2 parts in the X direction and 8 parts in the Y direction running on 6 processors will be 3.94. The figure is bold to denote that this is the maximal speedup when running the lifted network program on 6 processors. Observe that a faster execution is predicted for the lifted network programs: around 10% for the fastest version on 8 processors, and up to a factor 2.5 on less optimal configurations.

Farm						Lifted network										
$x \backslash y$	1	2	4	8	16	$x \backslash y$	1	2	4	8	16					
1	**1.00**	0.90	0.90	0.90	0.89	1	**0.99**	0.92	0.91	0.90	0.81					
2	0.87	0.85	0.85	0.85		2	0.90	0.90	0.88	0.82						
4	0.85	0.83	0.83			4	0.85	0.83	0.78							
8	0.80	0.79				8	0.77	0.72								
16	0.72				1 processor	16	0.62				1 processor					
$x \backslash y$	1	2	4	8	16	$x \backslash y$	1	2	4	8	16	$x \backslash y$	1	2	4	8	16
1	1.00	**1.74**	1.73	1.73	1.71	1	0.99	**1.82**	1.79	1.75	1.56	1	0.99	1.82	1.80	**2.31**	2.08
2	1.37	1.64	1.64	1.62		2	1.77	1.76	1.72	1.59		2	1.77	1.78	2.27	2.12	
4	1.33	1.60	1.58			4	1.66	1.61	1.51			4	1.68	2.12	2.00		
8	1.26	1.51				8	1.49	1.39				8	1.94	1.83			
16	1.12				2 processors	16	1.16				2 processors	16	1.52				3 processors
$x \backslash y$	1	2	4	8	16	$x \backslash y$	1	2	4	8	16	$x \backslash y$	1	2	4	8	16
1	1.00	1.74	**3.23**	3.22	3.19	1	0.99	1.82	**3.47**	3.37	2.97	1	0.99	1.82	3.47	3.38	3.87
2	1.37	2.51	2.54	2.49		2	1.77	3.42	3.29	3.01		2	1.77	3.42	3.34	**3.94**	
4	1.87	2.44	2.56			4	3.18	3.06	2.83			4	3.18	3.11	3.67		
8	1.76	2.06				8	2.79	2.58				8	2.80	3.32			
16	1.55				4 processors	16	2.10				4 processors	16	2.65				6 processors
$x \backslash y$	1	2	4	8	16	$x \backslash y$	1	2	4	8	16	$x \backslash y$	1	2	4	8	16
1	1.00	1.74	3.23	**5.66**	5.58	1	0.99	1.82	3.47	**6.27**	5.37	1	0.99	1.82	3.47	**6.27**	5.46
2	1.37	2.51	4.50	4.43		2	1.77	3.42	6.11	5.48		2	1.77	3.42	6.11	5.59	
4	1.87	3.33	4.01			4	3.18	5.59	5.02			4	3.18	5.59	5.13		
8	2.19	3.11				8	4.95	4.54				8	4.95	4.59			
16	1.91				8 processors	16	3.52				8 processors	16	3.55				12 processors
$x \backslash y$	1	2	4	8	16	$x \backslash y$	1	2	4	8	16					
1	1.00	1.74	3.23	5.66	**8.94**	1	0.99	1.82	3.47	6.27	9.05					
2	1.37	2.51	4.50	7.15		2	1.77	3.42	6.11	**9.28**						
4	1.87	3.33	5.62			4	3.18	5.59	8.21							
8	2.19	3.77				8	4.95	7.28								
16	2.17				16 processors	16	5.32				16 processors					

Table 1: The speedup figures as predicted by the task level simulator. All figures correspond to a tidal simulation over 35 time steps using a 64×64 space staggered grid, and use the fastest single processor figure (22 seconds) as a reference.

Taking a closer look at the figures for the lifted network version, it can be observed that the matrix should not be divided into square parts (2×2, 4×4), but into long thin rectangular areas of 1×4, 1×8 or 2×8. This is due to the column-major matrix representation (a column of rows)

Farm						Lifted network											
											$x\backslash^y$	1	2	4	8	16
											1			1.82	**2.30**	2.11
											2		1.78	2.26	2.13	
											4	1.68	2.09	2.03		
											8	1.94	1.83			
											16	1.52			3 processors	
$x\backslash^y$	1	2	4	8	16	$x\backslash^y$	1	2	4	8	16	$x\backslash^y$	1	2	4	8	16
1			2.76	2.96	**3.04**	1			3.02	**3.18**	2.94	1			2.97	3.37	**3.78**
2		2.23	2.07	2.47		2		2.93	3.11	2.95		2		2.93	3.29	**3.78**	
4	1.69	2.29	2.15			4	2.74	2.85	2.79			4	2.74	3.01	3.56		
8	1.66	2.08				8	2.63	2.52				8	2.74	3.25			
16	1.51			4 processors		16	2.07			4 processors		16	2.58			6 processors	

Table 2: Speedup figures of some lifted network versions as predicted by the instruction level simulator.

and the asymmetry in the definition of the equations of $u^k_{i,j}$ and $v^k_{i,j}$ (Equations 4 and 5). The net result is more communication between columns than between rows, hence it is preferable to parallelise along the rows. The use of a row major representation results in general in lower performance (around 3%).

The figures above show the merits of applying the communication lifting transformation, and also show some interesting differences between the performance of various configurations, which are easily overlooked when developing a program. We should bear in mind however that the task level simulator is fast but optimistic, the actual speedups (and execution times) will always be lower. It is unlikely, but not impossible, that the task level simulator has a bias for one or the other algorithm. A comparison of one configuration with the instruction level simulator will reveal this (see below).

8.7 Instruction level simulation of the lifted network version

It is not only interesting to browse the design space, but also to find a solution that completes a given task within a certain amount of time. This question can be answered by using the toolkit and the underlying methodology. Suppose for example that we need an architecture to run the tidal prediction program (using a 64×64 grid, over 35 times steps) within 10 seconds. Table 1 gives an indication which configuration to use, a 1×4 solver on 4 processors for example is predicted to execute in 6.34 seconds (divide the sequential solution by the speedup, 22 / 3.47 = 6.34). However, all these figures were obtained from a fast but inaccurate task level simulator. In accordance with the program development method a more detailed simulator is now used to obtain more accurate performance figures for selected configurations. As a selection criterion, we take the configurations with a speedup greater than 2.2 (22 sec / 10 sec). "Greater than" because the task level simulator gives optimistic results. This means that 4 processors will be needed, with 4, 8 or 16 tasks at least, or perhaps the 8×1 configuration on three processors. We have seen that the lifted version will run faster, but we also show the results for the 4 processor farm to show that the task level simulator does not bias towards one of the two. Various configurations of the lifted version have also been simulated on three processors as a reference point.

The speedups predicted by the instruction level simulator are shown in Table 2. The speedups are directly comparable with the speedups of the previous table, because the task and instruction level simulators agree on the execution time of the fastest version (22 seconds).

Total Jobs	Conf $X \times Y$	Execution Time lift-3	farm-4	lift-4	Simulation Errors lift-3	farm-4	lift-4
8	1×8	11.67	9.72	8.70	18%	23%	25%
4	2×2	14.83	11.57	9.40	17%	15%	20%
4	1×4	14.53	9.72	9.16	17%	18%	20%

Table 3: Seconds execution time of some lifted network versions as measured on the four processor Hypermodule. The errors shown are respective to the instruction level simulator predictions.

There are some differences between the speedup figures of the two simulators. These difference are caused by contention on the garbage collector (the tidal prediction program is a memory intensive program invoking the garbage collector regularly). The reason why this difference is clearly visible for some configurations (e.g. 2×2 on 4 processors), and invisible for other configurations (e.g. 2×2 on 3 processors) lies in the load (im-)balance of certain configurations. Four (2×2) jobs on four processors balance perfectly according to the task level simulator, but the contention on the semaphore disturbs this. Running on three or six processors will always lead to a poorly balanced load, generating so many idle background processors that the contention on the semaphores is completely hidden. In general the preference for long thin slices is more pronounced.

8.8 Executing the lifted network version on the target system

Finally the farm and lifted network versions of the program are run on the real hardware. The results of these runs are shown in Table 3. The instruction level simulator figures are noticeably optimistic. This is (as already explained in Section 6) because the instruction level simulator assumes unit time access to the memory. On the Hypermodule hardware, a cache will only respond in a single clock cycle if the access was a hit, and if the cache was not just snooping a transaction on the memory bus. Otherwise, the access will be delayed. The error for 4 processors is slightly higher than for three. This is caused by the shared bus: a cache miss on a real system is more expensive on a larger system because of the higher traffic on the memory bus. These hypotheses about the miss rate and miss penalty have been verified by running the bus level simulator. It turns out that half of the error is caused by assumptions about the miss rate. The other half is caused by the caches of the Hypermodule hardware: they become slower when they are snooping. This is especially the case when the traffic on the memory bus increases.

8.9 A perspective on the tidal simulation program

The Tidal prediction program has been an object of our research for at least 8 years. It is therefore impossible to say how much effort we have put into the development of the variants that we have presented here. As a relatively poor substitute, we have counted the number of lines in the various programs (excluding blank and comment lines) to give an indication of the amount of work involved in developing these programs. The Farm version of the program consists of 340 lines. The Lifted network versions of the program consist of 560 lines of hand written code plus a number of lines of machine generated code. The size of the code increases as the grid is partitioned more finely. However, the differences between the grid configurations are confined entirely to the machine generated code. For comparison, a C version of the tidal

simulation program has been written on the basis of Equations 4—6. The functionality of the sequential C version is comparable to that of the Farm version. It is 25% larger than the Farm version.

The program generator and the communication lifting transformer are both relatively small (together about 800 lines of functional code). This is due to the well defined semantics of purely functional languages. Transformations can be simple because there are no side effects. Similar tools for the C version would necessarily be more complex.

On the down side of using a functional language is the fact that the execution time of the sequential C version on the 88000 takes only 4.9 seconds, which is about twice as fast as the four processor parallel lifted network version. The difference lies in the fact that the C compiler allocates all data statically. The functional versions of the program use an expensive garbage collected storage allocation mechanism to support lazy evaluation.

We have demonstrated the use of the toolkit on only one example program with a fairly small number of processors. However, we believe that the methodology scales well to more processors or more complex programs. The stepwise refinement and transformation step are independent of the size of the input, and the number of processors employed. When the complexity of the program increases these steps are indispensable for the development process. When the data set is large, it will be necessary to trim the data set for the single node execution. With large data sets, the low level simulator is prohibitively slow.

9 Related work

An important research topic in parallel programming is the selection of the appropriate method by which parallelism is expressed. One extreme is the approach using low level primitives, such as send/receive for distributed memory systems, or semaphores for shared memory systems. The other extreme is the use of complete sub-programs, such as a parallel fast Fourier transform kernel. Such high level sub-programs are complete parallel implementations of a specific algorithm for a specific machine. The programmer provides the "glue" between the sub-programs. An intermediate approach is the support of various skeletons for parallel programming paradigms. These are less flexible than low level primitives, but more flexible than complete sub-programs. The skeleton based approach usually leads to better parallel programs, because it puts parallelism first [20]. Three main types of parallelism have been identified [21]. Our skeletons support these forms in a restricted manner; a *process farm* is supported with the sandwich, *algorithmic parallelism* can be exploited using a synchronous process network skeleton, and *geometric parallelism* is supported with a thin layer of sugar around the sandwich, constituting a parallel map of a function over a domain.

Even with a skeleton, parallel programming remains difficult, as the programmer must find an efficient way of dividing the computations and/or data so that several processors may work on sub-problems. Furthermore, there must be a schedule so that the order in which subproblems are computed is optimal. This mapping problem [22], is difficult because parallel computations must necessarily interact. The use of a declarative programming style requires that these interactions be made explicit. This enables the programmer and compiler to identify precisely when interactions take place. Reasoning about declarative programs is thus easier than working with programs that allow side effects, but in either case the mapping problem must be solved.

A good toolkit to assist the programmer is essential. Most of the toolkits that have been developed to support parallel programming are analysis tools for existing codes and visualisation tools for performance monitoring [23]. We now discuss some specific examples of existing toolkits that do not only cover the performance monitoring side, but the program development side as well.

A programming environment to support the development of parallel declarative programs has been built at Imperial College [24, 8]. Programs are written in a combined functional/logic language (HOPE+), and program development for the parallel ALICE and FLAGSHIP parallel architectures is supported using skeletons with program transformations. Imperial College provides more skeletons, and also more general skeletons than we do. Their program transformation tools must therefore be more powerful in order to be effective [25, 26]. At the same time more user guidance is required during the program transformation process, which is a disadvantage. Unlike Imperial College, we do not rely on the programmer to build a performance model of the application. The creation of a performance model is easy enough when the application is a simple one, but building performance models for complex applications is harder. We agree that the construction of a performance model for the application leads to better understanding of the parallelism and hence to better parallel programs, but we consider the development of a complete performance model for larger (complex) applications not yet feasible. For this reason, our approach is to enable the programmer to experiment with the application by high level simulation, so that an entire architecture design space can be explored for any given application.

The implementation of Sisal [27], developed at Lawrence Livermore Laboratory, is primarily focussed on speed for floating point intensive programs. The declarative semantics of the language enable optimisations that result in an execution speed exceeding the speed of equivalent C or Fortran programs. To achieve this performance, Sisal does not offer the same level of abstraction as most other functional languages: Sisal is eager and does not support higher order functions. The parallelism is denoted using a `forall`-loop. This is efficient for vector and matrix operations, but cannot be used for other types of parallelism such as divide and conquer. The PAWS performance evaluation tool [28] is intended to be used with ADA programs. However, PAWS works with the same data flow code (IF1) that is used as an intermediate language for Sisal. PAWS should thus also be usable with Sisal code. Unlike our toolkit, PAWS does not support proper computation. With PAWS the number of times a loop or function call is executed does not depend on real data, but on an estimate based on a frequency count from an actual run.

Many vendors of parallel platforms supply a kit for the development of parallel programs in an imperative style, usually oriented towards one of the Fortran dialects. The KSR-1 [29] (a parallel virtual shared memory architecture) supplies such a toolkit. The Fortran source code is annotated to denote the parallelism. The annotations allow the user to denote for example how DO-loops should be parallelised, or that two function calls should be evaluated in parallel. The compiler can perform some checks on the DO-loops to verify that the introduction of parallelism does not violate the correctness of the program. The programmer is fully responsible that the parallel functions do not interfere. The tools are all oriented towards real execution, which means that studies on the behaviour of programs on more processors than physically present are not feasible.

The ParaScope Editor [30] is another toolkit for parallel programming in Fortran. It is based on a compiler that automatically generates code for parallel execution from explicitly annotated loop constructs. The fundamental problem faced by the compiler is the handling of data dependencies in the context of parallel execution. Often the compiler has to make conservative assumptions to ensure correctness, so for optimal performance user annotations describing data dependencies are necessary. The ParaScope Editor is an interactive tool that guides the user through a large set of applicable annotations and transformations. At each step the tool reports the (estimated) performance effects of the transformation under concern, so the user immediately knows whether the transformation is worthwhile or not. In addition, the ParaScope toolkit performs the suggested source-to-source transformations, which relieves the programmer from this tedious and error prone task. The success of the ParaScope Editor

can largely be attributed to the limited parallel-loop skeleton which makes the integration of compiler, editor, and transformation engine possible. A disadvantage of the ParaScope toolkit, in contrast to ours, is that the user controls the correctness of the transformations because the user has to identify the data dependencies by hand.

The HeNCE system [31] is a high-level development kit based on PVM [32]. A parallel application is written as two separate parts: the first part consists of sequential modules written in Fortran or C, the second part specifies the interaction between the modules in a dedicated language, called the Node language. Interfacing between the two parts is done via the procedure parameter lists of the sequential modules which are exported to the Node language. The Node language supports a number of fixed parallel constructs: *farm* (called "fan" in HeNCE), *loop* (but only coarse-grain loop iterations), and *pipe*. The separation results in a clear but also rigid separation of computations and parallelism. This is not always desirable: parts of the algorithm end up in the Node language, other parts in the Fortran/C code. In the toolkits described above and the toolkit presented in this paper, parallel skeletons are added to the program at the *program* level. A graphical representation of the Nodes enhances the attraction of the HeNCE system.

10 Conclusions

Our toolkit is a parallel program development environment based on the skeleton approach. The programmer is required to structure the application to use one of the predefined process skeletons (for example a computation farm) for parallel execution. Throughout the process of annotating the code with skeletons for parallelism, the correctness of the program is preserved because of the declarative semantics of the programming language supported by the toolkit.

The toolkit integrates a compiler, a simulation package for performance analysis, and runtime support libraries for sequential and parallel execution on different hosts. Portability is achieved by having the compiler generate C, which is translated by an ordinary C-compiler to the specific target architecture, and by also coding the accompanying simulation package and runtime libraries in C. The simulation package offers three different simulators to allow for high (task) level, medium (instruction) level and low (bus cycle) level simulation.

The support provided by the simulation package is twofold. First, by using the simulators for performance prediction, the user can experiment with various skeletons to parallelise the code and identify the proper alternative for the specific application. Second, the simulators can be used to browse through a range of different architectures (e.g. varying the number of processors) to find a suitable combination of program and hardware that will operate within certain timing constraints. With the high level simulator a large number of configurations can be examined, the most promising ones are then studied in more detail with a lower level simulator.

Once the user is satisfied with the simulated performance results, the program can be compiled and run on real hardware. Currently the toolkit supports the execution on a four processor shared memory Hypermodule accompanied by a 68020 processor for I/O operations and monitoring. The runtime support runs directly on the bare hardware, which means that accurate timing measurements are feasible since the execution is not disturbed by the operating system or other users. The additional I/O processor can perform monitoring of the system with little overhead.

The presented case study (a scientific application that models the tides in the North Sea) is a typical example of how to use the toolkit: writing code, adding skeleton annotations, assessing the best way to partition the computations on the parallel machine through simulation experiments, and running the application in parallel on the shared memory machine.

Besides parallel program development, the toolkit has also been used for runtime-support research (amongst others the evaluation of memory management strategies) and study into hardware-design decisions like selecting suitable cache hierarchies. In all cases the use of the toolkit has given us significant advantages: the user is encouraged to experiment with the application to identify good solutions. These experiments lead to better insight in the performance characteristics of the system and this insight eventually leads to better performance.

11 Acknowledgements

We thank Marcel Beemster, Hugh McEvoy, Paul Stallard and the referees for their comments on draft versions of the paper. This work was supported by the European Institute of Technology under grant No. I88.

References

[1] R. J. M. Hughes. Why functional programming matters. *The computer journal*, 32(2):98–107, Apr 1989.

[2] R. S. Bird and P. L. Wadler. *Introduction to functional programming*. Prentice Hall, New York, 1988.

[3] S. L. Peyton Jones. *The implementation of functional programming languages*. Prentice Hall, Englewood Cliffs, New Jersey, 1987.

[4] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In J.-P. Jouannaud, editor, *2nd Functional programming languages and computer architecture, LNCS 201*, pages 1–16, Nancy, France, Sep 1985. Springer-Verlag, Berlin.

[5] D. A. Turner. *Miranda system manual*. Research Software Ltd, 23 St Augustines Road, Canterbury, Kent CT1 1XP, England, Apr 1990.

[6] P. H. Hartel, H. W. Glaser, and J. M. Wild. Compilation of functional languages using flow graph analysis. *Software—practice and experience*, 24(2):127–173, Feb 1994.

[7] K. G. Langendoen and P. H. Hartel. FCG: a code generator for lazy functional languages. In U. Kastens and P. Pfahler, editors, *Compiler construction (CC 92), LNCS 641*, pages 278–296, Paderborn, Germany, Oct 1992. Springer-Verlag, Berlin.

[8] J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. Sharp, and Q. Wu. Parallel programming using skeleton functions. In A. Bode, M. Reeve, and G. Wolf, editors, *5th Parallel architectures and languages Europe (PARLE), LNCS 694*, pages 146–160, Munich, Germany, Jun 1993. Springer-Verlag, Berlin.

[9] P. H. J. Kelly. *Functional programming for loosely-coupled multiprocessors*. Pitman publishing, London, England, 1989.

[10] W. G. Vree. *Design considerations for a parallel reduction machine*. PhD thesis, Dept. of Comp. Sys, Univ. of Amsterdam, Dec 1989.

[11] W. G. Vree and P. H. Hartel. Communication lifting: fixed point computation for parallelism. *J. functional programming*, 5(4):549–581, Oct 1995.

[12] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, Aug 1974. North Holland, Amsterdam.

[13] R. F. H. Hofman. *Scheduling and grain size control*. PhD thesis, Dept. of Comp. Sys, Univ. of Amsterdam, May 1994.

[14] H. L. Muller and L. O. Hertzberger. Evaluating all regular topologies of hierarchical cache architectures based on the Futurebus. In C. Eck, D. Del Corso, M. Hugelshorfer, K. Müller, C. Parkman, M. Pauker, W. Schoeps, R. Trechcinsky, H. Verweij, and D. Williams, editors, *Open Bus Systems '92*, pages 193–199, Zürich, Switzerland, Oct 1992. VFEA, Scottsdale, USA.

[15] W. G. Vree. The grain size of parallel computations in a functional program. In E. Chiricozzi and A. d'Amico, editors, *Parallel processing and Applications*, pages 363–370, L'Aquila, Italy, Sep 1987. Elsevier Science Publishers.

[16] A. W. Heemink. *Storm surge prediction using Kalman filtering*. PhD thesis, Twente technical Univ., Sep 1986.

[17] P. J. van der Houwen. Finite difference methods for solving partial differential equations. Mathematical centre tracts 20, Mathematical Centre, Amsterdam, 1968.

[18] P. H. Hartel and W. G. Vree. Experiments with destructive updates in a lazy functional language. *Computer languages*, 20(3):177–192, 1994.

[19] D. S. Wise. Matrix algebra and applicative programming. In G. Kahn, editor, *3rd Functional programming languages and computer architecture, LNCS 274*, pages 134–153, Portland, Oregon, Sep 1987. Springer-Verlag, Berlin.

[20] P. Brinch Hansen. Model programs for computational science: A programming methodology for multicomputers. *Concurrency: practice and experience*, 5(5):407–423, Aug 1993.

[21] A. J. G. Hey. Experiments in MIMD parallelism. In E. Odijk, M. Rem, and J.-C. Syre, editors, *2nd Parallel architectures and languages Europe (PARLE), LNCS 365/366*, pages 28–42, Eindhoven, The Netherlands, Jun 1989. Springer-Verlag, Berlin.

[22] M. G. Norman and P. Thanisch. Models of machines and computation mapping in multicomputers. *Computing surveys*, 25(3):263–302, Sep 1993.

[23] E. Kraemer and J. T. Stasko. The visualisation of parallel systems: an overview. *J. parallel and distributed computing*, 18(2):105–117, 1993.

[24] J. Darlington, P. G. Harrison, H. Khoshnevisan, L. M. J. McLoughlin, N. Perry, H. M. Pull, M. Reeve, K. M. Sephton, R. L. While, and S. Wright. A functional programming environment supporting execution, partial execution and transformation. In E. Odijk, M. Rem, and J.-C. Syre, editors, *2nd Parallel architectures and languages Europe (PARLE), LNCS 365/366*, pages 286–305, Eindhoven, The Netherlands, Jun 1989. Springer-Verlag, Berlin.

[25] P. G. Harrison and H. Khoshnevisan. Algebraic transformation techniques for functional languages. *The computer journal*, 31(3):229–242, Jun 1988.

[26] P. G. Harrison and H. Khoshnevisan. The mechanical transformation of data types. *The computer journal*, 35(2):138–147, Apr 1992.

[27] D. C. Cann. Retire FORTRAN? a debate rekindled. *CACM*, 35(8):81–89, Aug 1992.

[28] D. Pease, A. Ghafoor, I. Ahmad, D. L. Andrews, K. Foudil-Bey, T. E. Karpinski, M. A. Mikki, and M. Zerrouki. PAWS: A performance evaluation tool for parallel computing systems. *IEEE Computer*, 24(1):18–29, Jan 1991.

[29] Kendall Square Research. *KSR technical summary*. Kendall Square Research, Waltham, Massachusetts, 1992.

[30] K. Kennedy, K. S. McKinley, and Chau-Wen Tsjeng. Analysis and transformation in an interactive parallel programming tool. *Concurrency: practice and experience*, 5(7):575–602, Oct 1993.

[31] A. Beguelin, J. J. Dongarra, G. A. Geist, R. Manchek, and V. S. Sunderam. Graphical development tools for network-based concurrent supercomputing. In *Supercomputing*, pages 435–444, Albuquerque, New Mexico, Nov 1991. IEEE Computer Society Press, Los Alamitos, California.

[32] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: practice and experience*, 2(4):315–339, Dec 1990.

[33] J. Jeuring. *Theories for algorithm calculation*. PhD thesis, Dept. of Comp. Sci, Univ. of Utrecht, The Netherlands, 1992.

Appendix: Communication lifting transformation

The communication lifting transformation, which is described in full in our paper [11], can be described in a reasonably precise way without getting into the details required for a rigorous proof. Consider the equations corresponding to the process network of Figure 6:

$$
\begin{array}{ll}
hs_1 = map\ fh\ vs_1\ vs_0 & hs_2 = map\ fh\ vs_2\ vs_1 \\
vs_1 = map\ fv\ us_1\ us_0 & vs_2 = map\ fv\ us_2\ us_1 \\
us_1 = map\ fu\ ts_1\ ts_2 & us_2 = map\ fu\ ts_2\ ts_0 \\
ts_1 = i_1 : hs_1 & ts_2 = i_2 : hs_2
\end{array}
$$

First substitute hs_1, vs_1, us_1, hs_2, vs_2 and us_2 in the equations for ts_1 and ts_2. The streams ts_0, us_0, and vs_0 represent external streams carrying zero values. The substitution yields the rather unpleasant looking equations below. As some of the variables in the original equations appear in more than one place, some of the expressions have been duplicated. These duplicates will not be present in the final transformed version:

$$
\begin{array}{l}
ts_1 = i_1 : map\ fh\ (map\ fv\ (map\ fu\ ts_1\ ts_2)\ us_0)\ vs_0 \\
ts_2 = i_2 : map\ fh\ (map\ fv\ (map\ fu\ ts_2\ ts_0)\ (map\ fu\ ts_1\ ts_2))\ (map\ fv\ (map\ fu\ ts_1\ ts_2)\ us_0)
\end{array}
$$

To simplify the transformation steps that follow, define new functions f_1 and f_2, such that:

$$
\begin{array}{l}
ts_1 = i_1 : map\ f_1\ ts_1\ ts_2 \\
ts_2 = i_2 : map\ f_2\ ts_1\ ts_2
\end{array}
$$

Because all streams involved are infinite the *map* functions can all be combined. It is possible to prove that the definitions of f_1 and f_2 are (assuming that t_0, u_0, and v_0 represent externally defined zero values):

$$
\begin{array}{l}
f_1\ x_1\ x_2 = fh\ (fv\ (fu\ x_1\ x_2)\ u_0)\ v_0 \\
f_2\ x_1\ x_2 = fh\ (fv\ (fu\ x_2\ t_0)\ (fu\ x_1\ x_2))\ (fv\ (fu\ x_1\ x_2)\ u_0)
\end{array}
$$

The next and crucial step in communication lifting is to define a new stream zs as the *zip* of the two remaining streams ts_1 and ts_2 in the network:

$zs = zip\ (ts_1, ts_2)$
$\equiv$ (substitute the definitions of ts_1 and ts_2)
$zs = zip\ (i_1 : map\ f_1\ ts_1\ ts_2,\ i_2 : map\ f_2\ ts_1\ ts_2)$
$\equiv$ (unfold the definition of *zip*)
$zs = (i_1, i_2) : zip\ (map\ f_1\ ts_1\ ts_2,\ map\ f_2\ ts_1\ ts_2)$
$\equiv$ (define new functions $g_1\ (x_1, x_2)\ =\ f_1\ x_1\ x_2$ and $g_2\ (x_1, x_2)\ =\ f_2\ x_1\ x_2$)
$zs = (i_1, i_2) : zip\ (map\ g_1\ zs,\ map\ g_2\ zs)$
$\equiv$ (property of *zip*, see law 3.35 in Jeuring[33], and define $nextstate\ y\ =\ (g_1\ y,\ g_2\ y)$)
$zs = (i_1, i_2) : map\ nextstate\ zs$
$\equiv$ (property of *iterate*)
$zs = iterate\ nextstate\ (i_1, i_2)$

Gathering the definitions that have been introduced on the way and unfolding g_1 and g_2 yields:

$zs\ =\ iterate\ nextstate\ (i_1, i_2)$
$nextstate\ (t_1, t_2)\ = (h_1, h_2)$
where
$h_1 = f_1\ t_1\ t_2$
$h_2 = f_2\ t_1\ t_2$

Substitution of the definitions of f_1 and f_2 and common sub expression elimination to reverse the effect of the substitution above that introduced the duplicates:

$zs\ =\ iterate\ nextstate\ (i_1, i_2)$
$nextstate\ (t_1, t_2)\ = (h_1, h_2)$
where
$h_1 = fh\ v_1\ v_0$ $\qquad$ $h_2 = fh\ v_2\ v_1$
$v_1 = fv\ u_1\ u_0$ $\qquad$ $v_2 = fv\ u_2\ u_1$
$u_1 = fu\ t_1\ t_2$ $\qquad$ $u_2 = fu\ t_2\ t_0$

This completes the communication lifting transformation of the network of Figure 6.