GPH: An Architecture-independent Functional Language

Philip W. Trinder, Ed. Barry Jr., M. Kei Davis,
Kevin Hammond, Sahalu B. Junaidu,
Ulrike Klusic, Hans-Wolfgang Loidl,
Simon L. Peyton Jones

July 18, 1998

Abstract

In principle, pure functional languages promise straightforward architecture-independent parallelism. We investigate the validity of this claim in the context of our highly-portable implementation of an implicitly-parallel functional language: the GUM implementation of Glasgow Parallel Haskell (GPH). We discuss architecture independence at two levels: low-level (i.e. the implementation) and high-level (i.e. the programmer).

Low-level architecture independence is achieved by chosing a message-passing model for GUM, and implementing it using portable C and a widely-supported message-passing library like PVM. In fact GUM is largely independent of the message-passing library, and has been adapted to use MPI and the CM-5 CMMD libraries as well as PVM. As a result, GUM is easily ported, and is currently available on seven platforms including shared-memory machines, distributed-memory machines, and networks of workstations. We provide indicative measurements of how efficient and effective our architecture-independent runtime system is across a range of architectures.

The GPH programming model provides *higher-level architecture independence*. The parallelism in GPH is mainly implicit, and hence relatively small parts of the program need to be changed for a new architecture. The coordination that is required is expressed with a new high-level construct, *evaluation strategies*. Evaluation strategies provide a clean separation between algorithm and coordination, easing the task of changing either facet to suit new parallel environments. Moreover, GPH programs can systematically be developed for multiple target architectures, using a suite of simulation, profiling and visualisation tools. Much of the development is architecture-independent but, once a particular target architecture has been selected, the tools are parameterised to support tuning for that architecture. We demonstrate the systematic development of two real programs to the point of achieving good speedups on four architectures.

Index terms Parallel Functional Programming, Architecture-independence

Note to referees. The measurements on the seven architectures reported here are very recent, and we hope to analyse them more thoroughly for the final submission. We also hope to include more measurements, in particular of the Naira compiler on other architectures and `parfact` on a new AP3000 port.

1 Introduction

The goal of architecture-independent parallel programming is to deliver reasonable performance across a range of architectures, with minimal effort required to move from one to another. There are good reasons in principle to hope that a purely-functional language might be a good "fit" with this goal:

- **Mainly-implicit parallelism.** The fewer aspects of parallelism explicit in a program, the less that needs to be changed when a program is transferred from one architecture to another. Compared with other parallel approaches, many pure functional languages contain little explicit control of parallel behaviour: they rely instead on the compiler and runtime system to extract and exploit parallelism. The programmer still has to write a parallel algorithm and, typically, control a few aspects of the parallelism explicitly. We discuss in Section 3.2, how functional languages allow the control of parallelism to be expressed in a modular, non-invasive, and largely architecture-independent way.

- **Deterministic parallelism.** Functional languages ensure the absence of race conditions and deadlocks. The program is guaranteed to give the same result, every time it is run, and on every architecture. This is a huge boon to developers: non-repeatable, architecture-dependent bugs are extraordinarily difficult to find. Furthermore, the algorithm can be developed once, on a sequential architecture, and the parallelism added later without compromising the algorithm — assuming it was indeed a parallel algorithm in the first place.

- **Dynamic resource allocation.** In many parallel functional-language implementations the run-time system is responsible for mapping both data and threads to processors, and for scheduling the threads. In some systems, including GUM, the runtime system can also choose how many threads to exploit. These systems relieve the programmer from the task of specifying exactly what computations are to be executed where, with benefits for architecture-independent programming, and costs for execution efficiency. Both benefits and costs are hard to quantify.

So much for the theory. How does it work out in practice? In this paper we describe our experience of using GPH, a parallel implementation of the functional language Haskell, along with a suite of simulation and profiling tools that go with it. GPH, short for Glasgow Parallel Haskell, extends Haskell with two parallelism primitives (Section 3). GUM, short for Graph Reduction on a Unified Model, is the parallel runtime system that supports GPH.

The focus of the paper is on (a) the architecture independence of GUM and (b) the systematic development of applications with irregular parallelism and complex data structures for multiple architectures. The Naira Compiler (Section 5.2 is a good example of our target applications. For such applications, a programmer may well accept less than 100% machine utilisation in exchange for substantially reduced programming effort. Functional programming aims at this corner of the marketplace.

We claim that GPH supports architecture-independent parallel programming in the following ways:

- GUM is itself highly portable, because it uses a message-passing model and is implemented using the tried-and-tested technology of portable C and a message-passing library such as PVM. GUM is currently available on seven platforms, ranging from shared-memory machines, distributed-memory machines, massively-parallel machines, and networks of workstations. GUM is constructed to be largely independent of the message-passing library, and ports have used PVM [PVM93], MPI [MPI94], and the CM-5 native (CMMD) library (Section 2.1).

  Portability is not enough, of course. GUM is a sophisticated piece of software because it automatically manages many of the parallel aspects of a GPH program: including the distribution of work, data and garbage over the processors. As a result it is very easy (and not uncommon) to build a runtime system for a parallel functional language with absolutely atrocious performance. So we also have to ask about the efficiency and effectiveness of the runtime system on a range of architectures. Can an architecture-independent runtime system be made efficient compared to the best sequential implementations? Is it possible to achieve good absolute speedups on each architecture for programs with good parallel behaviour?

  The measurements reported in Section 2.2 indicate that GUM is efficient and capable of delivering good speedups, at least for simple demonstration programs, across the range of architectures. The best results are an efficiency of 95%, and a speedup by a factor of 74 on 97 processors for a simple benchmark program with good parallelism.

- Deterministic parallelism means that it is possible to systematically develop real GpH programs for multiple architectures (Section 4.1). The development is supported by a suite of dynamic analysis tools (Section 4.2). Much of the development is independent of the eventual target architecture, but once this has been determined the tools are parameterised to support tuning for that architecture. Much of the development occurs in a familiar workstation environment, and the architecture-independent phase of the development can be reused when targeting a new architecture.
- We demonstrate the systematic development of two real programs: an accident blackspot analysis and the Naira compiler. Blackspots is relatively small (1.5K lines), has simple parallel structure and achieves good wall-clock speedups on an SMP and on three networks of workstations (Section 5.1). Naira is larger (6K lines) and involves elaborate symbolic computation. Despite having complex irregular parallelism it delivers wall-clock speedups on a network of workstations, and simulated speedups on an SMP and a DMP architecture (Section 5.2).
- Because parallelism in GpH is mainly implicit, relatively small amounts of code are required to describe the parallelism in a program. We illustrate this for both Blackspots and Naira (Section 5). More significantly, the coordination that is required is expressed in a new high-level construct, *evaluation strategies*. The idea behind strategies is explained in another paper [THLP98], and outlined in Section 3.2. Evaluation strategies permit a high-level specification of coordination, and provide a clean separation between algorithm and coordination, easing the task of changing either to suit a new architecture.

The remainder of this paper is structured as follows. Section 2 outlines the low-level architecture-independence of the GUM runtime system, and some measurements of its efficiency and speedups on a range of architectures. Section 3 describes the GpH programming model, and its implications for developing programs for multiple architectures. Section 4 outlines our multi-architecture program development model, and the programming environment that supports the development. Section 5 describes the development of two programs using the model. Section 6 surveys related work, focusing on functional approaches. Section 7 Concludes.

2 Low-level Architecture Independence of GUM

2.1 GUM Runtime System

GUM is an architecture-independent runtime system for Glasgow Parallel Haskell GpH [THMPP96], a parallel variant of the Haskell lazy functional language. At the time we designed GUM we needed to develop programs on both shared-memory MIMD (SMP), and distributed-memory MIMD (DMP) architectures. To support this objective the runtime system needs an abstract machine-model appropriate to both architectures, and hence it's name: Graph-reduction for a Unified Machine-model (GUM). The machine-model chosen is message-passing, which is the obvious choice for massively parallel machines (MPPs), and our earlier experience constructing runtime systems on SMPs lead us to believe that it is also suitable for them [Mat93].

To achieve an architecture-independent runtime system we elected to extend our sequential multi-threaded runtime system, already written in portable C, with a message-passing library. We initially selected the PVM library [PVM93]. The resulting system is a parallel graph-reduction engine. The price of architecture independence is relatively-high message-processing costs: e.g. approximately 2000 machine instructions are required to receive a message. Such a cost is acceptable for MPP and workstation-network architectures where communications latency is already high, but less so on low-latency DMPs and SMPs. The raised communication costs impact on the GpH language supported by GUM: requiring that thread granularities are larger to offset the communications cost.

GUM's parallel structure is straightforward. There is a system manager and a collection of workers, so-called Processing Elements (PEs). Each PE has a copy of both the program and

Class, and Architecture	Processors	Comms. Package	Compiler Version	Operating System
MPP				
CM-5	97	CMMD	0.29	
DMP				
IBM SP/2	16	MPI	2.10	AIX
SMP				
Sun-SMP	6	PVM	0.29	Solaris 2
Workstation-net				
Sun-4/15	16	PVM	0.29	SunOS 4.1.3
Sun-10	16	MPI	2.10	Solaris 2.5.2
Digital Alpha	12	PVM	0.29	OSF 3.2
Intel Pentium	4	PVM	3.02	Linux 2.1.86/2.1.89-SMP

Table 1: Configuration Summary

the multi-threaded runtime system; and one PE is distinguished as holding the 'main-thread'. At the start of the program the system manager spawns the PEs, synchronises them, and then program execution starts on the main-thread PE. During execution PEs automatically distribute work, data and garbage using just six messages: three to transfer work, two to transfer data and one to recover garbage. The main-thread signals the end of the program to the system manager which then supervises the shut-down of the PEs. We do not attempt to support fault tolerance or dynamic re-configuration on these networks, as is done by, e.g. the Cilk system [BJKL+95]. A full description of GUM can be found in [THMPP96].

GUM is available on a seven machines drawn from several classes of parallel architecture. It is available on an MPP: the CM-5; on a DMP: the IBM SP/2; on an SMP: the SunSPARCserver; and on several networks of workstations, as summarised in Table 1.

GUM is designed to be independent of the message-passing library, and there are two aspects to this independence. The first is that GUM only uses a small number of common communication patterns. Moreover, only point-to-point communication is used during execution of the GpH program; broadcast and barrier synchronisation are also used, but only during initialisation and finalisation, and hence an efficient implementation is not essential. The second aspect of message-passing library independence is that a layered architecture is used to isolate the use of message-passing routines to just 3 of the 150 modules in the runtime system. The independence from any specific message-passing library has been exploited to construct versions of GUM that use MPI [MPI94] and the CM-5 CMMD native libraries, as outlined in column three of Table 1.

GUM can be tuned for a new architecture in several ways. For example it is possible to specify the message-size, and typically the higher the communications latency of the machine, the larger the message-size chosen. In making such decisions we are guided by experiments that simulate architectures with a range of latencies [LoHa96]. It is also possible to specify the sizes of GUM internal structures. For example it is possible to ensure that heap allocation occurs within hardware caches. Such control is especially important when moving between 32-bit and and 64-bit architectures. We have not yet systematically exploited these tuning capabilities across the machines.

2.2 Measuring GUM on Multiple-architectures

It is easy to port GUM to a new architecture because it uses tried and tested technology: C and a message-passing library. However, GUM is a sophisticated program, and a far more significant question is: does GUM perform well on a range of architectures? More precisely, is GUM efficient compared to the best optimised sequential implementations, and does it deliver good absolute speedups on each architecture? *Absolute speedup* compares parallel runtime with the runtime of the optimised sequential version of the program, whereas *relative speedup* compares with the parallel program run on a single processor.

We investigate these issues by measuring a simple benchmark program with good parallel behaviour on each architecture. The measurements indicate GUM's potential on the architectures: if GUM does not deliver good efficiency and speedups for this benchmark on an architecture then there is no point in developing real programs for that architecture. Section 5.1 gives measurements of real programs on multiple architectures. We have not attempted to benchmark GUM thoroughly on every architecture.

The benchmark used in this section, `parfact`, sums a range of integers, and the GpH code is included in Appendix A. The `parfact` program uses a divide-and-conquer paradigm, with a granularity threshold to indicate when to evaluate sub-tasks sequentially rather than dividing them. It was originally used to study task granularity on two architectures [THMPP96]. For the purposes of the experiments described here we maintain the cutoff at a setting that gives both acceptable granularity and a reasonable number of tasks (2048), but make no attempt to determine optimal granularity for each architecture.

2.2.1 Experimental Setup

We have used several machine configurations: one MPP, a 97-processor Connection Machine CM-5; one DMP, a 16-processor IBM SP/2; one SMP, a 6-processor Sun SparcServer; and four workstation networks as summarised in Table 1.

As far as possible, the workstation networks are homogeneous, i.e. all of the processors are the same, and communications are symmetric, e.g. all of the processors are on the same Ethernet subnet. The Sun-4/PVM network comprised a group of 16 Sun 4/15 workstations. The Alpha/PVM network consisted of a group of 11 DEC 3000/300LX workstations running at 125MHz, plus a single Digital AlphaStation 200 running at 233MHz. The Pentium/PVM network comprised one 233MHz Intel Pentium MMX processor, one dual 200MHz Intel Pentium MMX board with two processors, and one 166MHz Intel Pentium MMX processor. Finally, the Sun-10/MPI network comprised 10 mixed Sun-5 and Sun-10 workstations on the same subnet plus 6 on a second subnet.

Several versions of the GHC compiler are used on the different platforms. Most of the PVM-based systems and the Connection Machine use GHC version 0.29, which efficiently compiles Haskell version 1.2. The MPI systems use GHC version 2.10, and the Pentium system uses GHC version 3.02, which both compile Haskellversion 1.4. Both the latter ports are recently completed parallel ports, and some caveats apply.

All measurements reported here are given in terms of real (elapsed) time used by the program, and represent the average of several measurements (normally at least 5). As far as possible, timing runs were made on systems with minimal other load. While elapsed time is more variable than CPU time, especially for the workstation networks where neither processors nor networks are dedicated, some such measure is required to account for the costs of both communication and computation.

2.2.2 Efficiency

Table 2 gives the sequential runtimes and parallel efficiency (measured as the ratio of the sequential and single-processor parallel execution times) for the factorial program on all the host architectures. For the more mature PVM-based systems, the efficiency is high, in the range of 80% to 95%. The MPI systems show slightly lower efficiency in the range of 76% to 83%, perhaps reflecting the immaturity of these ports.

The sequential runtimes are obtained using the same level of optimisation as the parallel runtimes. For the systems using the older GHC0.29 compiler (Sun-SMP, Digital Alpha, and Sun-4), this is maximum optimisation. For the recent ports (SP/2, Sun/MPI and Pentium), however, we must use minimal optimisation to avoid a loss of parallelism through unwanted optimisation. The unoptimised figures are starred in the table. While intended primarily as indicators of system performance, the figures give a rough indication of the relative performance of the architectures

Class and Architecture	Sequential Runtime (s)	Parallel Efficiency
DMP		
IBM SP/2	91.19*	83%
SMP		
Sun-SMP PVM	35.9	92%
Workstation-net		
Alpha PVM	13.5	80%
Sun-4 PVM	43.2	92%
Sun-10 MPI	149.0*	76%
Pentium PVM	41.1*	95%

Table 2: Single-Processor Efficiency: Parfact; starred figures are without optimisation

within the starred or unstarred groups, Sequential times and efficiency figures are not available for the Connection Machine CM-5.

2.2.3 Speedup

Figure 1 shows the relative and absolute speedups that are obtained for `parfact` on each architecture with up to 16 processors. Figure 2 shows detail of relative speedup up to 4 processors plus the relative speedups obtained on the CM-5 up to 97 processors. Absolute speedup figures are not available for either the CM-5.

All the systems show good relative speedup up to 4 processors, as can be seen from Figure 2, and most show satisfactory relative speedup beyond that. The CM-5, Sun-SMP and Pentium systems all show acceptable speedup up to the number of processors that are available (in fact the performance of the Pentium network and the Sun-SMP is nearly identical on this example). The CM-5 also achieves the very good relative speedup of 74.1 on 97 processors, without appearing to approach a limit, indicating that GUM is capable of massive parallelism.

For most of the workstation networks and the SP/2 DMP a limit on performance is clearly being approached. This limit is about 6 (relative) in the case of the SP/2, or around 7 (relative) for the Sun-10 MPI port. In the latter case, this limit is not unexpected since this occurs at around the point where communication must be directed outside the local subnet (10 processors). It is somewhat surprising to find so stringent a limit on the SP/2, however. This may be a reflection of the small problem size.

The absolute speedup graph generally tracks the relative speedup graph, as expected. Interestingly, the two MPI ports deliver almost identical absolute speedup graphs. This suggests that the lower relative speedup for the SP/2 is due to greater efficiency of the MPI port on a single processor.

Finally, it worth noting the super-linear relative speedup that occurs for two Sun/MPI processors (most obvious in the detail plot of Figure 2. We have not yet determined the source of this effect, though we suspect it may simply be a consequence of the heterogeneous network.

3 Architecture-independent Programming in GpH

This section outlines the GpH programming model, and its implications for the development of programs for multiple architectures. It starts with the basic constructs for introducing parallelism, and outlines *evaluation strategies*, our new high-level mechanism for controlling parallelism within GpH.

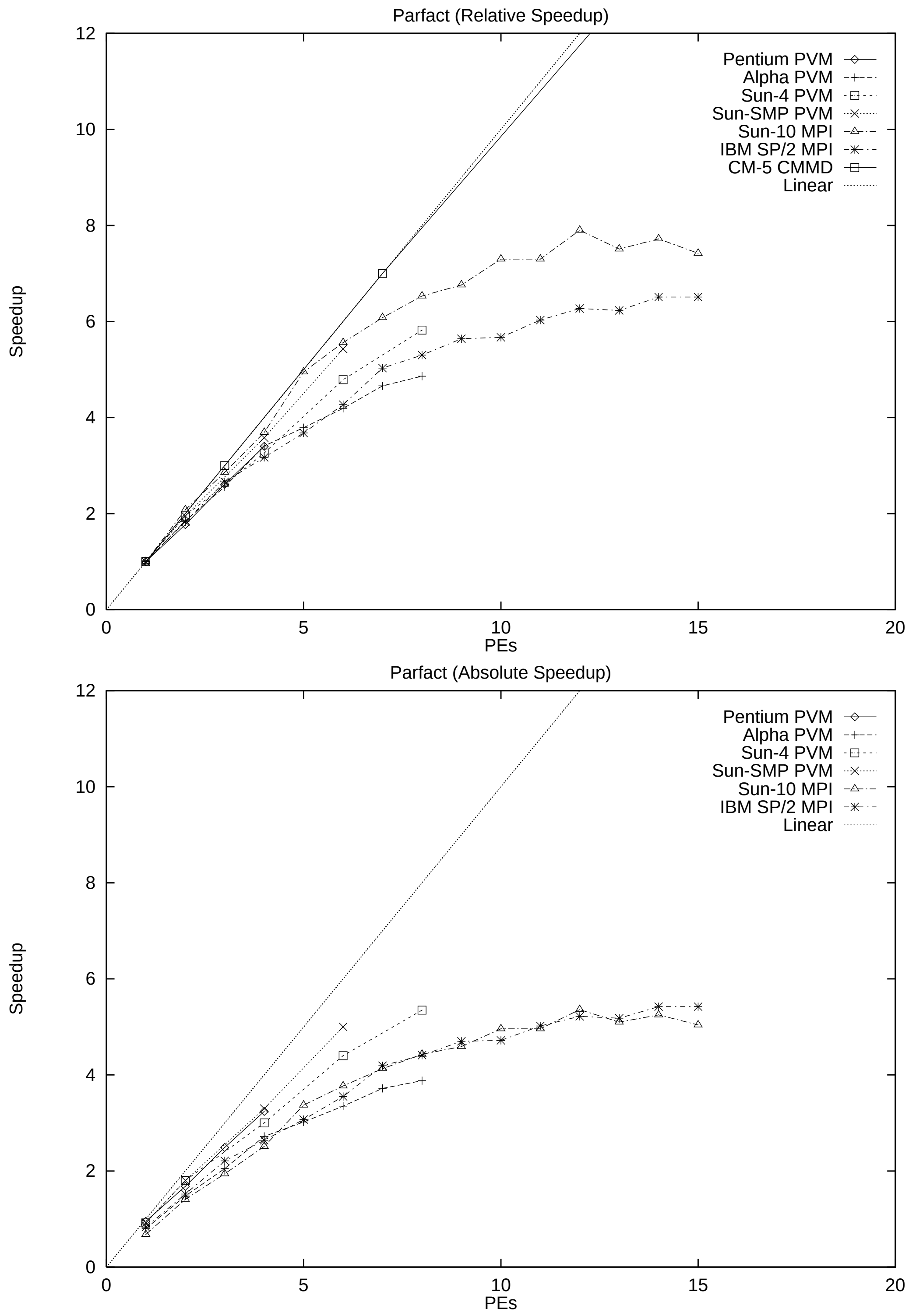

Figure 1: Relative and Absolute Speedups for `parfact`

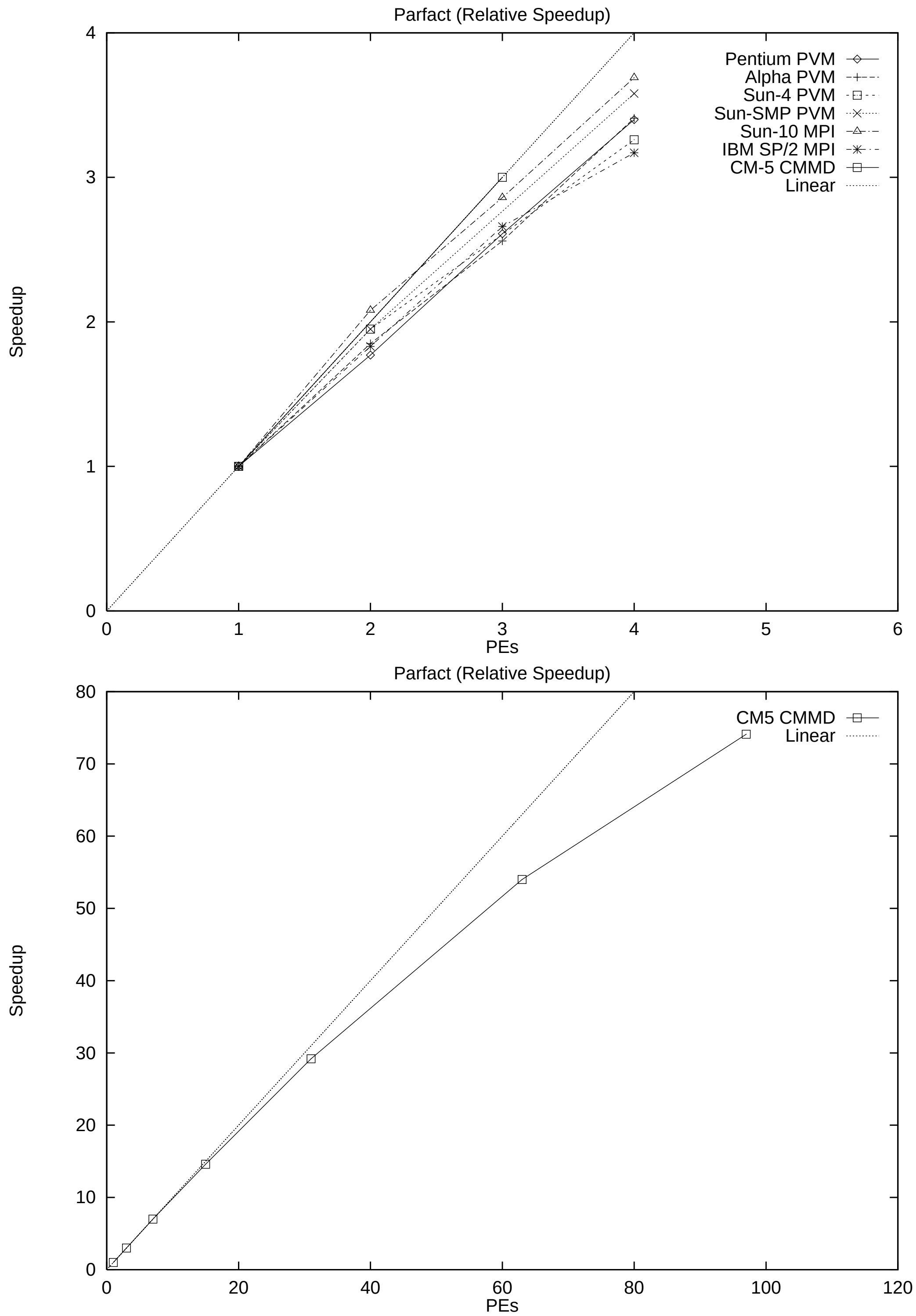

Figure 2: Relative Speedup for `parfact` – Detail up to 4 PEs (top) and CM-5 (bottom)

3.1 GpH: a Mainly-implicit Language

GpH is a small extension to the standard Haskell lazy functional language. The model of parallelism in GpH is mainly-implicit with dynamic resource allocation. The GUM runtime system manages most of the parallel execution, including the mapping of threads to processors, communication among threads and thread synchronisation. However the model does require explicit decomposition, i.e. the programmer must indicate those values that might usefully be evaluated by parallel threads and, since our basic execution model is a lazy one, also the extent to which those values should be evaluated. We term these programmer-specified aspects the program's *dynamic behaviour*.

Parallelism is introduced in GpH by the `par` combinator, which takes two arguments that are to be evaluated in parallel. The expression ``p `par` e`` (here we use Haskell's infix operator notation) has the same value as `e`, and is not strict in its first argument, i.e. $\perp$ ``` `par` e``` has the value of `e`. Its dynamic behaviour is to indicate that `p` could be evaluated by a new parallel thread, with the parent thread continuing evaluation of `e`. We say that `p` has been *sparked*, and a thread may subsequently be created to evaluate it if a processor becomes idle. Since the thread is not necessarily created, `p` is similar to a *lazy future* [MKH91].

Since control of sequencing can be important in a parallel functional language [Roe91], we introduce a sequential composition operator, `seq`. If `e1` is not $\perp$, the expression ``e1 `seq` e2`` also has the value of `e2`; otherwise it is $\perp$. The corresponding dynamic behaviour is to evaluate `e1` to weak head normal form (WHNF) before returning `e2`.

3.2 Evaluation Strategies

Even with the simple parallel programming model provided by `par` and `seq` we find that more and more code is inserted in order to obtain better parallel performance. In realistic programs the algorithm can become entirely obscured by the dynamic-behaviour code.

Evaluation strategies use lazy higher-order functions to separate the two concerns of specifying the algorithm and specifying the program's dynamic behaviour. A function definition is split into two parts, the *algorithm* and the *strategy*, with values defined in the former being manipulated in the latter. The algorithmic code is consequently uncluttered by details relating only to the dynamic behaviour. In fact the driving philosophy behind evaluation strategies is that *it should be possible to understand the semantics of a function without considering its dynamic behaviour.*

A strategy is a function that specifies the dynamic behaviour required when computing a value of a given type. A strategy makes no contribution towards the value being computed by the algorithmic component of the function: it is evaluated purely for effect, and hence it returns just the empty tuple `()`.

```
type Strategy a = a -> ()
```

3.2.1 Strategies Controlling Evaluation Degree

The simplest strategies introduce no parallelism: they specify only the evaluation degree. The simplest strategy is termed `r0` and performs no reduction at all. Perhaps surprisingly, this strategy proves very useful, e.g. when evaluating a pair we may want to evaluate only the first element but not the second.

```
r0 :: Strategy a
r0 _ = ()
```

Because reduction to WHNF is the default evaluation degree in GpH, a strategy to reduce a value of any type to WHNF is easily defined:

```
rwhnf :: Strategy a
rwhnf x = x `seq` ()
```

Many expressions can also be reduced to *normal form* (NF), i.e. a form that contains *no* redexes, by the `rnf` strategy. The `rnf` strategy can be defined over both built-in and user-defined types, but not over function types or any type incorporating a function type - few reduction engines support the reduction of inner redexes within functions. Rather than defining a new `rnfX` strategy for each data type `X`, it is better to have a single overloaded `rnf` strategy that works on any data type. The obvious solution is to use a Haskell type class, `NFData`, to overload the `rnf` operation. Because NF and WHNF coincide for built-in types such as integers and booleans, the default method for `rnf` is `rwhnf`.

```
class NFData a where
  rnf :: Strategy a
  rnf = rwhnf
```

For each data type an instance of `NFData` must be declared that specifies how to reduce a value of that type to normal form. Such an instance relies on its element types, if any, being in class `NFData`. Consider lists and pairs for example.

```
instance NFData a => NFData [a] where
  rnf [] = ()
  rnf (x:xs) = rnf x `seq` rnf xs

instance (NFData a, NFData b) => NFData (a,b) where
  rnf (x,y) = rnf x `seq` rnf y
```

3.2.2 Data-oriented Parallelism

A strategy can specify parallelism and sequencing as well as evaluation degree. Strategies specifying data-oriented parallelism describe the dynamic behaviour in terms of some data structure. For example `parList` is similar to `seqList`, except that it applies the strategy to every element of a list in parallel.

```
parList :: Strategy a -> Strategy [a]
parList strat []     = ()
parList strat (x:xs) = strat x `par` (parList strat xs)
```

Data-oriented strategies are applied by the `using` function which applies the strategy to the data structure `x` before returning it.

```
using :: a -> Strategy a -> a
using x s = s x `seq` x
```

A parallel map is an example of data-oriented parallelism, and is used in several of the programs. The `parMap` function defined below applies its function argument to every element of a list in parallel. Note how the algorithmic code `map f xs` is cleanly separated from the strategy. The `strat` parameter determines the dynamic behaviour of each element of the result list, and hence `parMap` is parametric in some of its dynamic behaviour.

```
parMap :: Strategy b -> (a -> b) -> [a] -> [b]
parMap strat f xs = map f xs `using` parList strat
```

Because evaluation strategies are written using the same language as the algorithm, they have several other desirable properties. Strategies are powerful: simpler strategies can be composed, or passed as arguments to form more elaborate strategies. Strategies are extensible: it is easy to define new application-specific strategies. Strategies can be defined over all types in the language, and offer some level of type safety because the normal type system applies to strategic code. Strategies have a clear semantics, which is precisely that used by the algorithmic language. A complete description and discussion of strategies can be found in [THLP98].

3.3 Multiple-architecture Implications

The model outlined above has several implications for the programmer developing programs for multiple architectures. Because parallelism in GPH is so implicit, only small amounts of code are required to describe the parallelism in a program. For many programs, it is only this code that needs to be adapted for a new architecture. Moreover, strategies allow us to specify the coordination at a high-level, and to separate algorithm and coordination thus easing the task of changing either to suit a new architecture. We have demonstrated this for toy programs: for example a strategy can use thresholding granularity control to select a thread granularity appropriate to some architecture. We have not yet needed to adapt the strategies in a real program for new architectures.

Many aspects of the execution of a GPH program are determined dynamically, including the number of threads, and their structure, thread placement and the amount of data communicated. GUM determines these aspects based on the resources available at runtime, and hence automatically adapts to the underlying architecture. For example on an architecture with high latency GUM will create additional threads to hide the latency. The adaption is limited, for example performance will be poor if thread granularity is small relative to communication costs. However, some programs will perform well on multiple architectures without changing the strategies, for example the accident Blackspots program in Section 5.1

4 Multi-Architecture Program Development

Unusually for a parallel language GPH is deterministic. This means that we can start with a program without any explicit parallelism, which can be developed and tested in a sequential environment. A parallel version of the program (constructed by inserting strategies) will always compute the same result as the sequential version, and have the same termination properties as long as the strategies only introduce *conservative* parallelism. There are no race hazards, core dumps, and un-repeatable errors.

4.1 Development Model

4.1.1 Overview

Our programming model is characterised by the semi-explicit nature of parallelism in GPH. Thus, the parallelisation of a program amounts to marking certain expression, indicating that they should be executed in parallel. The semantics of the sequential program will not be altered by adding these annotations[1]. As a result, the programmer will naturally start to develop and debug a sequential algorithm, using such proven tools as an interpreter and a sequential time and heap profiler. Alas, there no industrial strength debugger for a lazy functional language has been developed, yet. Therefore, it is even more important to design the program as independent unit and to test individual components separately. The high level of modularity inherent in a language with a non-strict semantics further facilitates this process [Hug89].

The development is separated into two parts: an architecture-independent phase, that develops adequate parallelism on a simulated idealised machine; and an architecture-dependent phase, that tunes the parallelism for some target machine. The idealised machine used in the architecture-independent phase has, for example, zero communication costs, and an unbounded number of processors and is provided by our GRANSIM simulator, described below (Section 4.2.2). An advantage of using an idealised machine is that we know that poor parallelism is not due to artifacts of some real machine: if good parallelism cannot be achieved on the idealised machine it cannot

[1] However, two dangers remain in the parallelisation of a Haskell program: resource exhaustion and a too strict evaluation degree. The former may cause a program to run out of memory despite succeeding in a sequential environment. The latter may cause the program to fail or run forever because an unneeded erroneous or infinite data structure is evaluated in the parallel version.

be obtained on any machine. We find in practice that most of the development work is done in the first phase. This is fortunate because it can be reused when targeting a new machine.

The first stage of the architecture-dependent phase is to tune the program on GranSim parameterised to emulate the target architecture. The final stage is to measure and tune the program on the target architecture using the GUM runtime system [THMPP96] and profiling tools. Our experiences with the parallelisation of programs such as Blackspots [LoTr97], Naira [JDH98], and Lolita [LMT+97] show that this stage typically requires only few changes. Normally the simulated results under GranSim are sufficiently close to the parallel behaviour under GUM. Thus, almost all of the development of the parallel algorithm can be performed using the simulator in an architecture-independent way. Since the simulator can be run on a sequential workstation, the programmer can use a familiar and rather cheap environment before bringing the program to a real multi-processor.

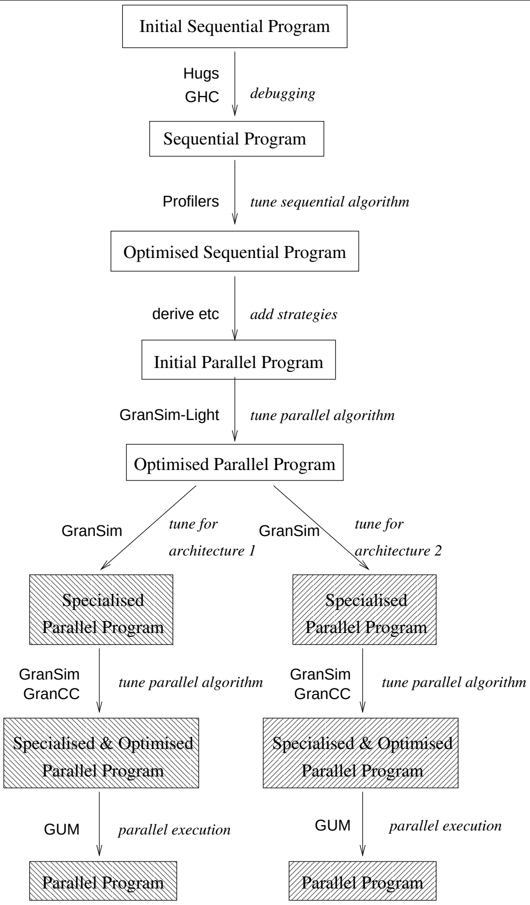

Figure 3: The Multi-Architecture Program Development Model

4.1.2 Development Stages

The program development model is summarised in Figure 3. The tools used at each stage are shown to the left of each arc, which indicates a modification to the previous version of the program. These tools are described in the next section. Note that this development process is iterative, i.e. at each stage it is possible to fall back to an earlier stage. This may happen if an efficient sequential algorithm turns out to have little inherent parallelism or if an algorithm with good idealised parallelism exhibits only poor parallelism in a realistic setup (see Section 5.1).

In the first stage a sequential program with some inherent parallelism is developed. In order to exploit parallelism in later stages, the algorithm should possess some inherent parallelism. In a functional language two expressions are inherently parallel if there is no data dependency between them, i.e. the result of one expression is not needed by the other. This stage uses the Hugs interpreter to test and debug individual components in the program. The next step after having developed a correct sequential algorithm is to tune its performance to make it more efficient. In this stage the time and space profilers of GHC are of crucial importance, in order to locate computationally expensive parts of the program. Based on this optimised version of the program evaluation strategies are added in order to expose parallelism in the program.

The initial parallel version is measured using the GranSim simulator parameterised to emulate an idealised machine with an infinite number of processors, zero communication costs, no thread overheads etc. In this stage the parallel algorithm is tuned until it exhibits a sufficient amount of parallelism.

Next, in the first architecture-dependent stage, GranSim is parameterised with realistic costs to model the characteristics of the target architecture. In this stage it might be necessary to e.g. modify the granularity of the generated threads in order to decrease the overhead for parallelism in the program. Finally, the parallel algorithm is executed on the real parallel machine using GUM. It might be necessary to slightly adapt the algorithm in order to deal with specifics of the parallel machine not covered by the simulator (e.g. system calls to special parallel runtime-system operations). However, based on our experience with large applications [LoTr97] this is rarely necessary.

It is sometimes necessary to iterate through the development process. For example, the most efficient sequential version of an algorithm is not always the most efficient parallel version. This has to be taken into account when choosing a particular sequential algorithm to be parallelised. Section 5.1 contains an example of iteration: an initial parallel version of the Blackspots program is constructed, but realistic simulation of the target architecture reveals that it cannot be made efficient, necessitating a return to the previous stage to develop an alternative parallelisation.

4.2 Development Environment

Building on several strands of development over recent years we have constructed an integrated environment for developing parallel functional programs. In particular, we make use of the infrastructure for developing and tuning sequential functional programs by using an interpreter, a compiler, and sequential profilers. Since GpH is only a minimal extension of Haskell these tools can be used in the first stages of developing a parallel program. In particular, profiling is useful for analysing the performance of independent pieces of computation, whose evaluation order is explicitly specified by a strategy. The GranSim simulator, which can accurately model the GUM runtime-system, enables the programmer to test the parallel program on a sequential workstation, without having to immediately rely on the environment provided by the parallel machine. Since GranSim, GUM, and GranCC, a parallel profiler, use the same log file format, the same visualisation tools can be used in all cases.

4.2.1 Execution Environment

The Hugs Interpreter. Hugs [Jon96] provides an interactive, interpreted environment, for fast development, experimentation and debugging of sequential code. Because of its user-friendly interface and its fast turn-around time in testing modified code, it is used at many universities for introducing functional languages. An new project is combining Hugs and GHC. The resulting environment would enable the programmer to mix interpreted modules with those compiled via GHC. This would achieve faster execution of the programs, because already-tested modules can be compiled, while retaining the flexibility of a simulator.

Hugs is also of special importance for debugging. In a functional language all constructs in a program are expressions with a deterministic value. All variables have a single-assignment property and no side-effects from calling other functions are possible. Clearly, such a language avoids many dangers of programming errors present in imperative languages. As a result, debugging usually boils down to examining the values of certain program expressions and testing individual subfunctions in isolation. All this can be done with an interpreter such as Hugs. Although a more sophisticated graphical environment that traces the exact evaluation order of expressions would be useful, we have found Hugs to be a useful tool for debugging large GPH programs.

The GHC Compiler and Sequential Runtime System. GHC [Pey96] is a state-of-the-art optimising compiler for the non-strict purely functional language Haskell. It incorporates various analyses phases that supply information about the program behaviour to the optimisation phase. Detailed measurements of the `pseudoknot` application show that GHC produces the fastest code of all existing Haskell compilers [HFA*96].

It should be emphasised, that for GPH an unmodified version of the GHC compiler can be used. The only new constructs, `par` and `seq`, are treated as built-in operators. Thus, the parallel program profits from all sequential optimisations, which are of particular importance in such a high-level language as Haskell. Furthermore, the maintenance of future versions of GHC and GUM is greatly simplified.

The GUM Parallel Runtime System. GUM is a portable parallel implementation of GPH. It supports one of the first publicly-available robust parallel functional languages. The robustness of GUM is demonstrated by its support of applications comprising tens of thousands of lines of source code [LMT+97]. It uses an unmodified version of GHC to generate optimised code. The two additional constructs `seq` and `par` are treated as built-in constructs, which specify evaluation order and generate parallelism, respectively. More details of the GUM implementation are given in Section 2.1, and in [THMPP96].

4.2.2 Dynamic Analysis Tools

The GPH parallel programming model is dynamic: e.g. the number and location of threads is determined at runtime. Moreover, the dynamic threads are managed automatically by the runtime-system. Therefore, it is rather difficult to perform a static analysis that approximates the runtime behaviour of the program. Some static analyses, such as a granularity analysis [LoHa96], are under development. However, most of the GPH program analysis tools are currently dynamic.

Sequential Time and Space Profilers. Because Haskell is a high level language it is often difficult to predict the efficiency of certain functions. The lazy evaluation mechanism may cause some data structures not to be evaluated at all. On the other hand, it may cause the program to retain big data structures, although they are not subsequently used. This is called a "space leak" and it is a common problem in non-strict languages.

In order to aid the programmer in the performance-tuning stage of the sequential program, time and space profilers are supplied together with GHC [SaPe95]. Both profilers use the novel approach of cost centers in order to attach computation costs and heap usage to pieces of source code. The exact semantics of cost centers has been defined formally in [SaPe97].

Many examples show the importance of such a sequential profiler for improving the performance of programs. Using a first prototype implementation of the profiler on GHC halved both the runtime and the heap consumption [SaPe95]. The development of a large (47Kline) natural-language processor Lolita also made use of the profiler. In fact, this particular use of the profiler sparked research on extending the profiler in such a way that cost centers can be aggregated in a post processing stage without having to re-run the program. This technology of cost center stacks [Jarv96] is now being integrated in the next version of GHC.

The GRANSIM Parameterisable Parallel Simulator. In the development of a parallel program within our software engineering environment, the highly-parameterised GRANSIM simulator [HLP95] plays a crucial role in achieving architecture-independence. Due to the large number of parameters available to GranSim two different simulations can be distinguished: an idealised and a realistic simulation. Using an idealised simulation, which assumes an unlimited number of processors and zero communication costs, GranSim allows the programmer to hide all details of the underlying parallel architecture. If this stage reveals a sufficient amount of parallelism in the algorithm the programmer can then use a realistic simulation, which models details of the processors, the interconnection network, and of the communication mechanism.

This approach of developing the parallel program on the GranSim simulator in various levels of abstraction has been used with success in parallelising several large applications such as Lolita the Naira compiler (see Section 5.2), and the Accident Blackspot program (see Section 5.1). Because of GranSim being highly parameterised, most of the parallelisation uses under the simulator. Only the final stage with some minor adjustments to the parallel architecture had to be performed under GUM.

The main features of GranSim are flexibility, accuracy, integration into GHC, and robustness. The following paragraphs discuss these features in turn. A complete presentation of all features of GranSim is given in [Loi98][Chapter 3].

- **Flexibility.** In addition to the basic distinction between an idealised and a realistic simulation, GranSim provides a large set of parameters that allow the user to specify characteristics of the architecture. In particular the user can specify details of the underlying processor (such as the costs for arithmetic operations etc), of the communication network (in particular its latency), certain runtime-system features (such as synchronous or asynchronous communication). Taken together the parameters allow the programmer to model the behaviour of a range of architectures accurately.

- **Accuracy.** In contrast to most existing simulators for the parallel execution of functional languages, GranSim measures time in clock cycles rather than abstraction reduction steps in the underlying model of parallel graph reduction. This yields a far more accurate result of the simulation, accounting for the exact costs of basic operations in the abstract machine.

- **Integration into GHC.** Like GUM, GranSim is fully integrated into GHC. The only modification in GHC is due to the instrumentation of the code generated by GHC. A large portion of the code in the runtime-system of GranSim is shared with GUM. This further ensures that the simulated execution time under GranSim is closely related to the runtime under GUM.

- **Robustness.** To date, GranSim has been used in the parallelisation of such large-scale packages, such as Lolita, which is one of the largest existing non-strict functional programs.

The GranCC Profiler. GranCC [HHLT97] is a profiler for GpH, constructed by combining cost centers with a modified GranSim runtime system. With GranCC the programmer can mark parts of the parallel program. In the resulting profiles the number of threads can be classified by the cost center that generated them. This additional dimension in the standard activity profiles adds valuable information for the programmer. Although GranCC is still in an early prototype stage, it has proven useful in the parallelisation of Naira.

Visualisation Tools. The log files from GranSim, as well as those from GranCC and GUM, can be used to generate graphical information about the execution of the program. In particular, two kinds of profiles can be generated: activity profiles and granularity profiles. The former show the number of threads during the computation. The latter counts the number of threads classified by e.g. total runtime in the form of a histogram. In experimenting with a range of visualisations we have found that these two classes of profiles convey the most useful amount of information.

Activity profiles can be generated in three levels of detail:

- An overall activity profile shows the activity of the whole machine. Over the runtime, as the x-axis, the total number of running, runnable, fetching, and blocking threads is shown as differently-coloured areas.

- A per-pe activity profile shows the activity on each processor. The realistic setup of GranSim is restricted to at most 64 processors, in which case this information can still be summarised in a single picture.
- A per-thread activity profile presents the most detailed level of information. It shows the state of a threads as the thickness of a horizontal line. In practice the log file has to be pre-processed in order to filter out the interesting threads.

This structuring of the profiles into a hierarchy is very useful for parallel computation. It has also been used for example in the VISTA [Hals96] toolset. Unlike VISTA, however, our environment only produces individual graphs and lacks the possibility of a dynamic browser. In an ongoing project are extending the log-file format to incorporate both sequential and parallel profiles, with the aim of re-using tools, such as a dynamic browser, in both a sequential and a parallel environment.

Our visualisation tools can also produce granularity profiles. These profiles are histograms, usually classified by the runtime of the individual threads, i.e. a granularity profile shows how many threads have a runtime within a certain interval. This information proves to be very important in order to avoid the production of an excessive amount of fine-grained parallelism in a GpH program. A detailed discussion and examples of all available profiles are given in [Loi98][Section 3.3.3].

5 Large Architecture-independent Programs

GpH's multi-architecture development model and development environment, have been constructed and refined by the engineering of several symbolic GpH programs. The programs are large (up to 50K lines of code), and cover a range of application areas [LoTr97]. All but one of the programs deliver modest wall-clock speedups; and the programs have been measured on half a dozen parallel architectures: mainly networks of workstations, and shared-memory multiprocessors.

This section illustrates our multi-architecture development model and development environment by describing the construction of two GpH programs: the Accident Blackspots program is a simple data-intensive application, and Naira is a larger and more complex compiler. The Blackspots program demonstrates architecture-independence with good efficiency and speedups on an SMP and on three networks of workstations. Naira achieves speedups on a network of workstations, and on simulated SMP and DMP architectures.

5.1 Accident Blackspots

5.1.1 Problem Description

The University of London Centre for Transport Studies wishes to analyse road traffic accident data. Given a set of police accident records (modified to preserve privacy) the task is to discover accident blackspots: locations where two or more accidents have occurred. A number of criteria can be used to determine whether two accident reports are for the same location. Two accidents may be at the same location if they occurred at the same junction number, at the same pair of roads, at the same grid reference, or within a small radius of each other. The radius is determined by the class of the roads, type of the junction etc. The problem is obviously data-intensive, and too complex for conventional database query languages like SQL.

Locating blackspots amounts to combining several partitions of a set into a single partition. For example if the partition on road pairs is `{{2,4,5},{3},{6,7}}` and on grid references is `{{2,5},{3},{4,6},{7}}`, the combined partition is `{{2,4,5,6,7},{3}}`. The problem of unioning disjoint sets, *union find*, has been much studied by algorithm designers as it has an interesting sequential complexity. For n union and m find operations, an algorithm with an amortised complexity of O(n + F(m,n)) can be given, where F is a very small function (the inverse of the Ackermann function) [Tar75]. These RAM algorithms are not directly applicable in our application because not all of a large data set may be randomly accessed in memory. We have adopted an index-, or tree-, based solution with complexity O(n log n) if n is the number of elements in the sets. The motivation for this choice is that for very large data sets not all of the tree need be memory resident at any time.

5.1.2 Parallelisation

Sequential Implementations The first stage of our development is to construct an efficient sequential implementation. The application was originally written at the Centre for Transport Studies [WuHa96] in PFL and has subsequently been rewritten in Haskell. PFL is an interpreted functional language [PoSm93], designed specifically to handle large deductive databases. Unusually for a functional language, PFL provides a uniform persistent framework for both data and program. The PFL program uses selectors, a special bulk-data manipulating construct, and hence an algorithm that is slightly different from that used in the Haskell program. It comprises approximately 500 lines.

The Haskell implementation constructs a binary *sameSite* relation containing an element for each pair of accidents that match under one of the four conditions. The combined partition is formed by repeatedly finding all of the accidents reachable in *sameSite* from a given accident. The program has four major phases: reading and parsing the file of accidents; building indices over the accident data; constructing sameSite, and indices over sameSite; forming the partition. The program is a 300-line module, together with 3 library modules totaling 1300 lines.

The original data set comprises 7310 accident reports, and the programs discover 1229 multiple-accident sites where a total of 5450 accident occur. The programs are run on similar, but not identical, workstations: PFL on a Sun ELC, and Haskell on a Sun Sparc Classic. The runtimes of the programs are as follows, PFL: 1105 seconds, Haskell: 123 seconds. The faster execution of the Haskell program is attributed to it being both compiled and highly optimised, where PFL is an interpreted research language. More measurements of the PFL and Haskell programs, together with a more detailed discussion can be found in [THLP98].

Initial Parallel Version The next stage of our development is also architecture-independent and entails constructing an initial parallel version of the program and measuring it with an idealised simulator. A good initial parallel version was obtained by successively introducing parallelism, and measuring the result. Table 3 summarises the results obtained when just 1000 accidents are partitioned on an idealised 4-processor machine. *Work* is the total number of cycles required to evaluate the program: it closely approximates the runtime of a sequential version of the program. *Runtime* is the time required for the program to run in parallel under GRANSIM in this case on 4 processors. Both work and runtime are measured in units of 10^6 GRANSIM machine cycles. *Average parallelism* is the parallelism averaged over the runtime: during the execution there will sometimes be more parallelism, and sometimes less. *Speedup* is relative because under GRANSIM there is no 'sequential' version of the program.

Version (I) Pipeline only. The first parallelisation is to convert the 4 phases of the program into a pipeline. The speedup of 1.2 is low because the pipeline is blocked by the trees passed between stages.

Version (II) Parallel Pipeline Stages. The next parallelisation introduces parallelism within each pipeline stage using a variety of parallel paradigms.

Version (III) Parallel Pipeline Stages and Preconstructed Indices. Parallelism is further improved by merging the first two pipeline stages. That is, the indices on the accident data are constructed before the program is run, and the program reads the indices from a file rather than constructing them.

Tune for Architecture Having obtained satisfactory parallelism on an idealised machine, for version III, we are now ready to tune the program for our target architecture. We start by measuring version III on GRANSIM parameterised to emulate the target 4-processor SMP. Unfortunately, as reported in table 4, it delivers poor results because of the fine grain of parallelism and the

Parallelisation	Work (MCycles)	Run Time (MCycles)	Average Parallelism	Speedup
Pipeline only (I)	327	273	1.2	1.2
Par. Pipeline Stages (II)	327	124	2.8	2.6
Par. Pipeline Stages & preconstructed Ixs (III)	304	87	3.5	3.5
Geographically Partitioned (Tiled) (IV)	389	105	3.7	3.7

Table 3: Idealised 4-Processor Simulation: Blackspots

Parallelisation	Work (MCycles)	Run Time (MCycles)	Average Parallelism	Speedup
Par. Pipeline Stages & preconstructed Ixs (III)	393	171	2.3	2.3
Geographically Partitioned (Tiled) (IV)	394	105	3.7	3.7

Table 4: Realistic 4-Processor SPARCserver Simulation: Blackspots

volume of data being communicated. This motivates us to return to the previous stage in our development method and develop a fourth parallelisation of the program.

Version (IV) Geographically Partitioned (Tiled). A very different, coarse-grained, parallelisation can be obtained by splitting the accident data into geographical areas Each area, or *tile*, can be partitioned in parallel before aggregating the results, using this standard technique [MiSc95]. Accidents occurring near the edges of a tile must be treated specially. This approach is only feasible because every accident has a grid reference and we assume that accidents occurring more than 200m apart cannot be at the same site. Accidents occurring within 100m of the nominal edge between two tiles are duplicated in both tiles, and this results in a 4% increase in data volume. Because some accidents are duplicated in the borders, some multiple-accident sites may be discovered in more than one tile, and must be combined in the aggregate result.

Version IV of the program delivers good speedups under both idealised and realistic simulation, as detailed in Tables 3 and 4. We can now develop this version of the program on the target machine.

Multi-Architecture Measurements.

Data. The original data set of 7310 accident reports occupies 0.3Mb and is too small to get good results on parallel machines. For the purposes of this section, the data is replicated 6 times. The larger data set could be kept in larger tiles, or in more tiles of the same size, and the latter approach is taken for the following reasons. We found that, as long as the tiles are large relative to the border area, many smaller tiles are more efficient than a few large tiles [LoTr97]. Moreover, peak resource usage is reduced because if there is one tile per PE then all of the file reading occurs at the start of the program, inducing intense network traffic. With multiple tiles per PE the file reading is spread through the program execution. Multiple tiles utilise the dynamic load management provided by GUM, demonstrating that the GpH program is independent both of the number of PEs and of the number and size of tiles. In contrast a small number of large tiles could be statically allocated to PEs. However it is tedious to maintain the allocation as the number of tiles or PEs change.

The replicated data occupies 1.8 Mb and is split into 40 tiles with two different sizes. There are 32 *small* tiles, each containing approximately 1000 accidents and occupying 37Kb, and 8 *large* tiles each containing approximately 2000 accidents and occupying 73Kb.

Program. Only one change is required to the GranSim version of the program to enable it

Class and Architecture	Sequential Runtime(s)	Efficiency
SMP		
Sun-SMP PVM	135.2	77%
Workstation-net		
Digital Alpha PVM	378.6	63%
Sun-4/15 PVM	815.4	84%
Sun-10 MPI	289.1	96%

Table 5: Single-Processor Efficiency: Blackspots

to run under GUM. GUM processes don't inherit file handles from the main thread, and hence to permit them to read files the program uses the 'unsafe' C-interface supported by Glasgow Haskell [LaPe95]. On both machines the program is warm started, i.e. it is run at least once before measurements are taken. Warm starts reduce runtime because the data is preloaded into RAM disk caches in the file system.

Efficiency Table 5 shows sequential times and parallel efficiency for the Blackspots program, on four of the architectures described in section 2.2.1. This table makes an interesting comparison with the corresponding figures for `parfact` in Table 2. The parallel efficiency is notably lower for all platforms, except for the Sun-10 MPI which has unoptimised `parfact`. For `Blackspots`, the fastest machine is the Sun SPARCserver, whereas for *parfact*, the fastest machine was the Digital Alpha. The Sun 4/15 is also much slower relatively for `Blackspots` than for `parfact`. The best explanations of these differences are probably that the `parfact` application is sufficiently small that the Alpha is able to keep its working set within the primary cache, where this is not possible for `Blackspots`, and that the I/O performance of the SPARCserver is much better than that of the Alpha and Sun 4/15 workstation networks.

Speedups Figure 4 shows the relative and absolute speedups obtained for the Blackspots application for the four architectures. The relative speedups are generally good, especially for the two PVM workstation networks, which achieved relative speedups of 11.94 (16 Suns) and 9.46 (12 Alphas). The absolute speedup for the Suns is much better than for the Alphas, however, reflecting the greater parallel efficiency of 84% as opposed to 63%. The Sun MPI port again showed a limit being reached of around 4 (relative) or 3 (absolute). As with the results for `parfact`, there is an unexplained super-linear spedup effect, this time for 3 processors. Further work is needed to investigate these results.

For this application, the Sun SMP was limited to 4 processors. Speedup for this configuration is less than for the workstation networks (2.82 relative, 2.16 absolute). This may reflect the significantly better overall performance – the 4-processor Sun **SMP** is 50% faster than the 16-processor Sun network, and marginally faster than the 12-processor Alpha network (62.51s v. 63.55s). Clearly there is still a future for SMPs, though the competition from workstation networks is increasing rapidly.

5.1.3 Discussion

The Blackspots program solves a real problem using real data and thus provides a realistic context in which to illustrate our multi-architecture development model and environment. An interesting stage in the development is that an apparently good initial parallelisation (III) proved to be unsuitable for our target architecture, and was superseded by a coarser-grain parallelisation (IV).

The Blackspots program demonstrates architecture-independence with good wall-clock speedups on both an SMP and three networks of workstations without requiring modification. The final parallelisation (IV) is extremely simple: essentially a single task farm, with only a small amount of irregularity in the task sizes. The program could easily be written using conventional architecture-independent techniques, e.g. C and MPI. Thus Blackspots is not representative of the class of symbolic programs with highly-irregular parallelism that we believe GPH is good at expressing. The Naira program presented next is far more representative of this class of program.

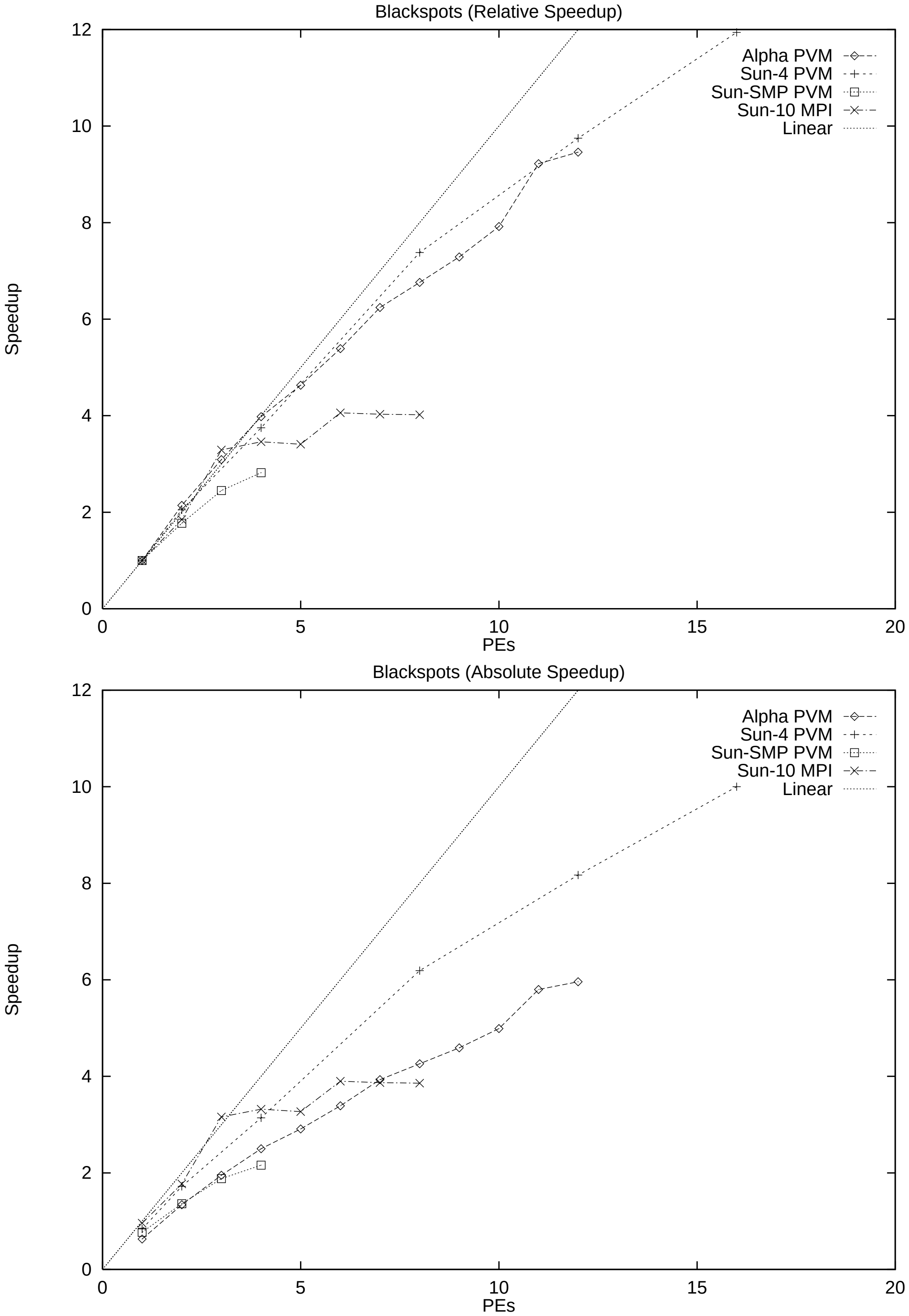

Figure 4: Relative and Absolute Speedups for `blackspots`

5.2 Naira Compiler

5.2.1 Program Description

Naira is a parallel, parallelising compiler for a rich, purely functional programming language. It processes, and its front-end is written in, a subset the standard Haskell 1.2 language with type classes as the main feature omitted. The front-end comprises about 5K lines of Haskell code organised in 18 modules. The back end is written, following popular tradition, in the C programming language.

The main motivation for writing Naira is to explore the prospects and problems of parallelising a modern functional language compiler [Jun98]. Another aspect is to make the compiler accept parallelised program inputs and to generate multithreaded parallel code so that we can assess the efficiency of the resulting parallel code. These two aspects of Naira—that it is itself parallel and that it generates parallel code—makes it, to our knowledge, the first functional language compiler of its kind. It is currently the second largest GPH program.

5.2.2 Parallelising Naira

Sequential Implementation The efficient sequential implementation that we parallelise was developed over a period of 4 years. The front-end compiles to a graph-reducing parallel abstract machine with a strong dataflow influence. The top-level structure of the compiler in terms of its main phases is shown in Figure 5. The first, analysis, pass consists of the lexical analyser and the parser. The next four passes implement the pattern matching compiler, the lambda lifter, the type checker and the intermediate language optimiser respectively. The detailed organisation and implementation of these passes is described elsewhere [Jun98]. The two-way split after the lambda lifting pass indicates that the result of the lambda lifter can be piped simultaneously to both the type checker and the optimiser and that these latter two phases can proceed in parallel combining their results, using `showModule`, to produce the intermediate code which is input to the code generator.

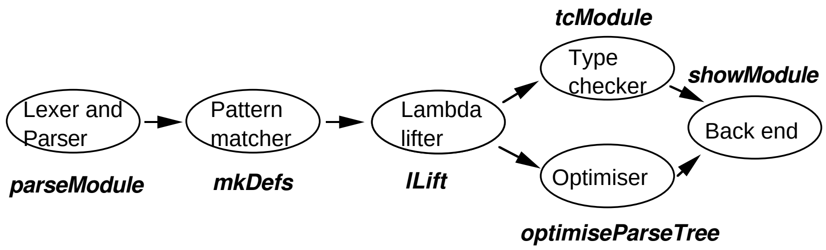

Figure 5: The Pipeline Structure of Naira's Main Phases

Sequential Time Profiling reveals that the phases of Naira have similar computational cost, except for type checking, which is significantly more expensive. Hence, in order to get good overall performance, we must pay special attention to type checking.

Initial Parallel Version The first stage of parallelisation is to introduce a top-level pipeline between the compilation phases. The laziness of the language is crucial here to ensure that the output of one phase is made available incrementally to the next phase(s) so that the analyses in the phases can proceed in parallel. Figure 6 shows the function, `analyseModule`, that implements the top-level pipeline. The underlined portions show the only code that need to be added to ensure the parallelisation of the top-level pipeline.

```
analyseModule fileName modName imports exports symbTabs defs =
    showModule modName impVals dats exports              $||
                                      parPair parForceList parForceList $
    fork (optimiseParseTree fileName exports stOpt  aInfo,
         tcModule fileName stTE exports tInfo  syns) $|| parForceList $
    lLift fileName stPM                                $|| parForceList $
    mkDefs fileName stPM                               $|| parForceList $ funs
  where (stPM,stTE,stOpt)   = symbTabs
        (dats,syns,funs)    = defs
        (aInfo,tInfo,impVals) = imports
fork (f, g) inp = (f inp, g inp)
parForceList = parList rnf
```

Figure 6: `analyseModule` rewritten using Pipeline Strategies

A second, nested, level of parallelism is introduced by parallelising several phases of the compiler, namely the pattern matcher, lambda lifter and the intermediate language optimiser. These are parallelised, generally, in a data-parallel manner by ensuring that the respective analyses in each phase are applied to the definitions in a module in parallel.

The type checker is carefully parallelised because profiling has revealed that it is the most expensive phase. We use a parallel name server to minimise data dependencies and thus avoid sequentialising the inference process. For instance, to typecheck two quantities d_1 and d_2, we analyse them simultaneously in the current type environment, each returning a type and a substitution record. If a variable v common to both d_1 and d_2 is assigned (possibly different) types t_1 and t_2 from these two independent operations, t_1 and t_2 will be unified in the presence of the resulting substitutions and the unified type associated with v. Parallelism has been exploited at four different stages in the type checker: in a data-parallel fashion when typechecking definitions in a module; in typechecking local definitions in parallel with the top-level ones; on calls to frequently used functions; and in typechecking aggregate expressions.

The initial parallel version of Naira has been measured compiling each of it's 18 constituent modules, and these results are summarised in Table 6. The first two rows report the parallelism and speedups obtained for the best and worst inputs. The third row is mean parallelism and speedup obtained for all 18 modules. On an idealised machine Naira delivers satisfactory speedups for most inputs, although there are some where it does not. Because of the idealised machine, we know that the poor results for these modules are not an artifact of the underlying architecture.

Module(s)	**Avg. Par**	**Speedup**
Best (SyntaxUtils)	8.4	8.13
Worst (Syntax)	1.9	1.40
Mean (18 modules)	5.5	4.36

Table 6: Idealised Simulation: Naira

Tune for Architectures. We can now tune the parallelism in Naira for specific architectures. We proceed by measuring its performance on a simulation of the target architecture, and adapting it as required. In fact we have measured Naira on two architectures: one SMP, the other DMP. The simulated SMP has 8 processors, very low communications latency (10 cycles), and communicates lazily, i.e. individual data items are sent on demand. The simulated DMP also has 8 processors, but a higher communications latency (2K cycles), and is semi-eager in communication: it sends a block of related data on demand. As before, we measure the compilation of each of Naira's modules, and report the best, worst and mean figures in Table 5.2.2.

Module(s)	SMP		DMP	
	Avg Par	Speedup	Avg Par	Speedup
Best (TCheckUtils)	4.9	4.68	5.6	5.32
Worst (Syntax)	1.8	1.39	1.8	1.35
Mean (18 modules)	4.0	3.95	3.5	3.55

Table 7: Realistic 8-Processor Simulations: Naira

As expected on small numbers of processors, the DMP results are, on average, slightly worse than the SMP. However, the results are remarkably close for the two architectures. We interpret this to mean that the grain of parallelism is sufficiently coarse to allow good performance on both an SMP and a DMP.

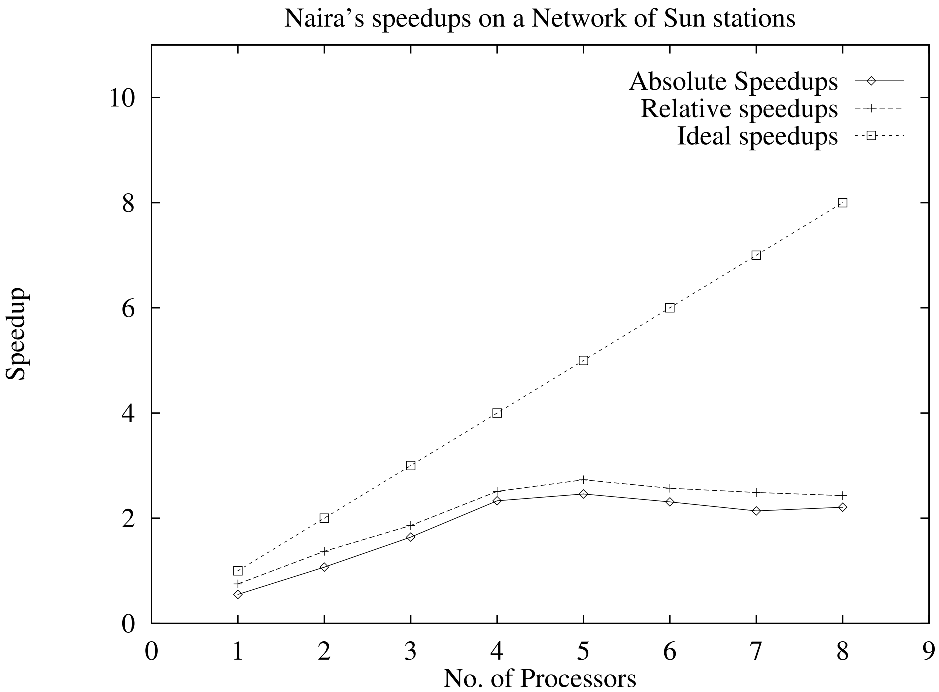

Figure 7: Absolute Speedups for Naira

Workstation Network Measurements We now have a version of Naira with satisfactory, if modest, simulated parallelism for our target DMP architecture. We measure it on a network of 8 Sun (4/20) workstations, running Solaris 2 and connected to a common Ethernet segment, and Figure 7 shows the results. The maximum relative and absolute speedups of 2.73 and 2.46 are achieved with five processors. The speedups are are in agreement with the relative speedup of 3.01 predicted by GranSim for this architecture.

6 Related Work

In this paper, we have considered the two extremes of architecture independence. At a low level, this is best characterised as portability: a program that was originally written for one architecture will compile and run without change on another. Performance, however, will usually be sub-optimal, and is likely to be poor; good performance being achievable only through careful tuning and rewriting of the program source. This level is characterised by portable library or system approaches such as PVM [PVM93] or MPI [MPI94, SOHW$^+$96], which we have exploited in our own runtime system, and which others have integrated as external libraries into Haskell(e.g. [WiOD97]). A similar approach is the HPC++ language and toolkit [GBJGL98] that aims to provide natural support for threads, synchronisation and RPC in a high-performance parallel version of C++. Since such approaches do not aim to support generic parallel programs, it is usually necessary for the programmer to specify considerable detail about the intended parallel behaviour on a target architecture.

High-level architecture independence frees a programmer from such low-level concerns. Indeed, in the ideal case it should be possible to attain good performance on any target architecture without source modification or tuning. This is the goal that we have worked towards with GpH, by exploiting the potential for highly implicit dynamic parallelism that exists within a purely functional language.

Clearly, such an exacting goal may not be attainable for all programs: there will inevitably be situations where good parallel performance for an architecture or class of architectures will require alternative algorithms (for example, where algorithms deliver good but non-scalable performance), and it will usually be necessary to accept a pragmatic tradeoff between portability and performance. However, it should not be necessary to produce several specially tuned variants of a single parallel algorithm in order to realise good performance on a variety of platforms.

This section surveys those architecture-independent systems and models that are most closely related to GpH and GUM. The breadth of work in this area means that it is not possible to survey all architecture-independent approaches to parallelism in a paper of this length. Skillicorn and Talia have, however, produced a good parallel programming models [SkTa98], to which the reader is referred for more general discussion of the issues and a much fuller coverage of the range of approaches that have been proposed.

6.1 Portable Parallel Functional Language Systems

In a functional context we are aware of very few portable parallel implementations, and no other approaches where architecture-independence has been satisfactorily demonstrated. This sub-section surveys portable implementations; the following sub-sections survey higher-level architecture-independent approaches.

The majority of parallel functional language implementations have been produced in order to investigate particular research issues in a specific parallel context. For example, the HDG-machine [KLB91] studied the use of independent closures and evaluation transformers in a distributed machine environment. Consequently, very few of these implementations have been been ported to multiple platforms.

A notable exception is the Concurrent Clean variant of the Clean language [NSvEP91], a functional language with a strong family resemblance to Haskell. Concurrent Clean extends Clean with a large number of parallel control annotations, which can be used to control parallel computation quite precisely. The language has been implemented on two architectures: a transputer network, and a distributed network of Macintosh workstations, where impressive performance results have been obtained for the simple benchmarks that have been tried to date [PlvE93, GMS93].

The Alfalfa/Buckwheat implementations of the Alfl language were produced for distributed-memory and shared-memory systems respectively [GoHu86, Gol88]. These systems aimed to support automatic program distribution through determining "optimal" granularity for parallel tasks. Problems with communication cost and granularity meant that the earlier distributed-memory system (Alfalfa) did not achieve this goal. The Buckwheat system did however meet the goal of automatic task distribution with good speedup for simple programs on the shared-memory architecture.

Sisal [FMS+95] is a first-order functional language based on dataflow techniques. Unlike Haskell, it does not aim to be a general-purpose symbolic processing language, but targets numeric processing. In this application domain, Sisal delivers exceptionally good performance, rivalling or exceeding that of Fortran for much less programmer effort [Can92]. Sisal has been implemented on a variety of platforms, including the Cray T3D and the CM-5, where it has proved possible to achieve good absolute speedup without source code modifications. Indeed, in several cases better speedup and performance has been achieved than could be obtained by using vectorised Fortran [LANL98]).

Finally, the implicitly parallel dataflow language Id has been ported to both the Monsoon dataflow machine as well as more conventional CM-5 and workstation networks [HCAA93, HLB95]. Threads are introduced automatically in loop constructs, with automatic throttling through k-bounded loops. Id has also formed the basis of a implicitly parallel functional language based on Haskell, pH [NAH93], though no parallel implementations have yet been produced.

6.2 Architecture-independent Systems and Approaches

The primary problems that must be addressed by any architecture-independent system are:

1. **decomposition**: partitioning a program into parallel tasks;
2. **mapping**: placing these tasks on multiple processors; and
3. **scheduling**: deciding which of these tasks should be run and when.

Depending on how they tackle these fundamental problems, systems that aim to support architecture-independence can be broadly classified into:

- paradigmatic approaches;
- static approaches; and
- dynamic approaches.

6.2.1 Paradigmatic Approaches

Paradigmatic approaches simplify the problem of parallel programming by restricting the programmer to one or more pre-determined parallel paradigms. It is then possible to provide suitable implementations of those paradigms for a variety of architectures. One major advantage of paradigmatic approaches is that the strong structuring they impose opens the prospect of constructing equally strong cost models. A second advantage is that partitioning and mapping are often straightforward. The primary disadvantage is a lack of flexibility: it will usually be necessary to rewrite any existing program to exploit a particular paradigm, if this is in fact possible.

Data Parallelism: Data-parallelism is the epitome of the single-paradigm approach, and there have been several data-parallel functional language implementations, including two based on Haskell: Data Parallel Haskell [Hil94] and CM-Haskell [KiOD98]. Most data-parallel approaches restrict data-parallelism to a single program level, the NESL language [Ble96] is interesting in providing support for nested data parallelism.

While implementations of data-parallel languages have usually focussed on massively-parallel machines with direct hardware support for data-parallelism, such as the Connection Machine, data-parallelism can be exploited in many other architectural settings too. For example, NESL has been implemented on the massively parallel CM-2, a distributed-memory Cray Y-MP, and a shared-memory Encore Multimax [BCH+93].

BSP/SPMD: BSP (Bulk Synchronous Processing) [SHM97] can be viewed as a kind of data-parallelism in which computation super-steps over a subset of the program are interspersed with synchronised communication steps involving all active processors. Like other data-parallel approaches, architecture-independence is limited by the need to ensure regular, balanced computation steps. SPMD (Single Program Multiple Data) [SMTD+95] extends this notion to allow irregular communication during computation steps.

There have been some attempts to combine BSP/SPMD with functional languages (e.g. [Ser98, FoCh95]), but we are not aware of any portable functional language implementations, or ones which have demonstrated clear architecture independence.

Algorithmic Skeletons: Algorithmic skeletons [Col88] embrace single-paradigm approaches such as the data-parallel approach and enhance them by providing the programmer with a fixed repertoire of parallel templates (*skeletons*), which can be parameterised to suit the parallel application. Since skeletons are naturally described as higher-order functions, there is an elegant and close fit with functional languages: several authors have described functional skeletons (e.g. [Col88, Rab93, Kes95, Bra94]).

While such approaches can work well for suitably structured applications, e.g. the computer vision algorithms that have been parallelised by Michaelson et al. [MiSc95], programs whose structure does not fit the pre-defined skeletons will exhibit little or no parallelism. There is thus a loss of generality compared with a more dynamic approach, such as evaluation strategies.

Work in the skeleton community is presently focussed on defining a common set of skeletons that is capable of capturing all common parallel paradigms, and on relaxing the constraint that only a single skeleton can be exploited. Success in achieving the latter goal would allow skeletons to be composed both vertically (nesting) and horizontally (pipelining), but would require more sophisticated cost models to be constructed. An example is the work by Darlington et al on SCL [DGTY95]. The work on *shapely types* is also relevant here [JaCo94], since it offers the prospect of automatically choosing suitable skeletons depending on the type of the data [JCSS97].

Both of the goals outlined in the previous paragraph are attained naturally by the evaluation strategy approach, of course, albeit at the cost of introducing dynamic overhead which can perhaps be avoided by judicious choice of skeletons and good implementation techniques.

6.2.2 Static Approaches

Static approaches rely on compilation techniques to ensure good decomposition, mapping, and communication patterns. Once determined, these aspects cannot be changed during program execution. The benefits are low dynamic overhead and straightforward cost modelling. In the absence of dynamic load balancing, however, a highly regular program may be required if starvation is to be avoided.

Caliban: Most parallel functional language systems have favoured either paradigmatic or dynamic approaches. One influential exception is Caliban/Advanced Caliban [Kel89, Tay97], which is intended to target loosely-coupled multiprocessors.

The Caliban language adds the concept of `moreover` clauses to a conventional functional language (most recently a subset of Haskell [Tay93]). These clauses specify behavioural information. While there is consequently a superficial resemblence to a `using` clause, the purpose of a `moreover` clause is rather to describe a static process network which can then be used to map a program to the target architecture at compile time. Different host configurations, including ones involving varying numbers of processors, may require recompilation from source.

To the best of our knowledge there is as yet no complete parallel implementation of Caliban (Cox et al. describe a partial implementation on transputers [CHKLT92]), and it is consequently moot whether it is suitable for architecture-independent programming.

6.2.3 Dynamic Approaches

Dynamic approaches use load management techniques to ensure good parallel behaviour. The most dynamic approaches, as used by GPH, control not only the placement of parallel tasks but also the introduction of new tasks. The advantages of such an approach are that there is a high degree of resilience in the face of dynamically changing system characteristics, and that the system can be rapidly ported to new platforms. The disadvantages are that dynamic overhead is unavoidably incurred, the runtime system may be greatly complicated by the requirements of load management, and that it may be difficult or impossible to construct a suitable cost model.

Coordination Languages: Like evaluation strategies, coordination approaches allow the separation of behaviour and algorithm [GeCa92]. However, unlike evaluation strategies, behavioural control is achieved by the use of a meta-language (the coordination language) into which is embedded the normal algorithmic language (the computation language). It is therefore necessary to deal with two levels of language semantics in order to completely understand the behaviour of a parallel program. In compensation, it should be unnecessary to define a new semantics for each new binding of coordination language to computation language: the combination of the coordination language semantics with the computation language semantics should give a precise definition of parallel program behaviour.

The coordination language approach is typified by Linda [GeCa92] or PCN [FoTa94]. Linda uses a shared tuple space to express communication between sequential processes written in the computation language, whereas PCN uses three composition operators to link pairs of communication ports sequentially, in parallel, or through choice.

While the computation language is usually an imperative one such as C, there appears to be no intrinsic reason why it should be impossible to construct, for example, a Linda binding for Haskell: Haskell-Linda. The implementation requirements for Haskell-Linda would be similar to those for Concurrent Haskell [PGF96], which uses *monads* [Wad97] to ensure safe execution of concurrent Haskell threads using explicit communication.

In the functional community, a similar approach has been adopted by the Eden language, which Haskell with an explicit coordination language supporting process abstractions and communication channels [BLOMP97]. Each process has its own independent address space. Internally, a process may be multi-threaded in order to produce multiple output streams, each of which is connected to a different communication channel. Implementation is proceeding by modifying GHC and GUM to incorporate the necessary language and runtime system extensions.

A related development is Chakravarty et al.'s GOFFIN system [CGKL98]. This extends Haskell with a constraint-logic language whose purpose to control process creation and placement, and which like Eden possesses a formal operational semantics. Although there are no parallel implementations of GOFFIN, it may be possible to extend an existing parallel Haskell compiler with the necessary coordination mechanism, as for Eden.

Para-Functional Approaches: Para-functional approaches extend a functional language with control annotations to support dynamic placement and distribution [Hud86]. Communication is, however, implicit, as in the evaluation strategy approach. The para-functional approach thus inverts the coordination language approach by incorporating the coordination language into the computation language rather than vice-versa. The result is a more tightly coupled dynamic semantics than can be obtained with a coordination language approach.

The earliest para-functional languages incorporated a separate annotation language (e.g. [Hud86, Hud91, Sch95], but Mirani and Hudak's first-class-schedules [MiHu95] use monads to attach schedules to expressions. This allows schedules to be constructed as normal functional language expressions of type `Schedule`, and permits the use of standard higher-order functions etc.

Compared with evaluation strategies, the para-functional approach allows finer control: it is possible to express dynamic process networks for example, or to define schedules that include explicit task synchronisation. However, the lower level of control means that more detail must be specified, and the scope for architecture independent computation is thus reduced. It is also necessary to provide a full operational semantics for schedules, whereas evaluation strategies will require only an operational semantics for `par` and `seq`. The latter appears to be much a much simpler task, since it is not necessary to address issues of synchronisation, placement etc.

Dynamic Imperative Approaches: There have been several attempts to exploit multi-threading in an imperative context. Many of these have used C or C++ as a base language.

CId [Nik95] exploits ideas from the Id dataflow language and implementation. CId is a simple extension of C, that adds lightweight threads using explicit fork/join parallelism and globally synchronised objects. Like GUM, CId provides work stealing, with automatic granularity control. Unlike GUM, however, where lazy thread creation [MKH91] allows task creation decisions to be taken after those of task identification, CId uses dynamic load indications to determine whether a thread should be created immediately it is identified, as with the Star:Dust [Ost93] architecture. Such an approach allows cheap task creation, but runs the risk of starvation for less regular programs. CId has been implemented on a small network of workstations, where performance results are similar to those quoted in this paper for a range of small application programs.

ICC++ is a parallel version of C++ built on the portable Illinois Concert system [CDG+98a], which aims to support fine-grained parallelism in an object-oriented setting. The Concert system is available on a number of parallel platforms including the SGI Origin and the Cray T3D. Like GUM, Concert supports a globally distributed, but implicit namespace, and incorporates dynamic load management. It has been used to program a number of irregular parallel programs, achieving good relative speedups of up to 50 on a 64-processor Cray T3D.

6.3 Discussion

Paradigmatic and static approaches allow parallelism to be specified easily and costed accurately. The loss of generality and in some cases performance may, however, limit architecture independence. Dynamic approaches are extremely flexible, but inevitably carry some runtime overhead.

While it will almost certainly be necessary to invest significant effort constructing new cost models and tuning compilers for static approaches, our experience to date with our dynamic runtime system is that it is highly resilient, and needs relatively little tuning to cover even very different classes of parallel architecture, let alone difference platforms within the same class. We feel this is an important issue in architecture-independence, allowing rapid colonisation of new parallel architectures and fast ports to future systems.

Clearly it is both possible and desirable to combine static and dynamic techniques in order to obtain the benefits of both approaches. We are working on a cost model for one such hybrid model [Loi98]. The problem is rendered more complicated in our setting by the use of lazy evaluation, which makes it difficult to define tight bounds on cost formulae. Initial results suggest, however, that it should be possible to construct a system that is sufficiently accurate to provide useful granularity information to a dynamic runtime system.

7 Conclusion

7.1 Summary

We have investigated the architecture-independence of a pure functional language with dynamic and mainly-implicit parallelism. We have investigated the low-level architecture independence of the GUM runtime system by measuring its efficiency and effectiveness across a range of parallel platforms. We have seen how GPH's deterministic parallelism allows the systematic development of large programs for multiple architectures, and followed the development of two real programs for 4 architectures. We have seen that evaluation strategies isolate the algorithm from parallel coordination, making both easier to adapt for a new architecture.

7.2 Future Work

We intend to improve and extend the GUM runtime system, and to port it to new platforms. Many aspects of GUM could be improved, including the work-stealing algorithm and the message-processing as suggested by measurements in [LoHa96]. There are a number of obvious extensions to GUM, e.g. to introduce thread migration, i.e. the relocation of a running thread from one processor to another. A number of GUM ports are under way or planned, including to a Fujitsu AP1000, a Fujitsu AP3000, and a Beowulf-class system.

In the longer-term, we would like to develop a language in which the parallelism is even more implicit than in GPH. One means of doing so would be to automatically insert strategies into a program, guided by static analyses of the program text. Strictness analysis [AbHa87] indicates when it is *safe* to introduce parallelism, and granularity analysis [Loi98] indicates when it is worthwhile to do so. The automatic insertion of strategies could be parameterised by characteristics of the target architecture to produce a program suitable for that architecture.

References

[AbHa87] S. Abramsky, and C. Hankin (eds.), *Abstract Interpretation of Declarative Languages*, Ellis Horwood, Chichester, UK, 1987.

[BCH+93] G.E. Blelloch, S. Chatterjee, J.C. Hardwick, J. Spielstein, and M. Zagha, "Implementation of a Portable Nested Data-Parallel Language", *Proc. Fourth ACM Conf. on Principles & Practice of Parallel Programming (PPoPP)*, San Diego, CA, May 19–22, 1993, pp. 102–111.

[Ble96] G.E. Blelloch, "Programming Parallel Algorithms", *Comm. ACM*, **39**(3), 1996, pp. 85–97.

[BJKL+95] R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E.Leiserson, K.H. Randall, Y. Zhou, "Cilk: an Efficient Multithreaded Runtime System", *Proc. PPoPP '95 – 5th ACM Symp. on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995, pp. 207–216.

[Bra94] T.A. Bratvold, *Skeleton-Based Parallelisation of Functional Programs*, PhD thesis, Department of Computing and Electrical Engineering, Heriot-Watt University, Edinburgh, November 1994.

[BLOMP97] S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña, "The Eden Coordination Model for Distributed Memory Systems", *Proc. HIPS '97 — High-Level Parallel Programming Models and Supportive Environments*, IEEE Press, 1997.

[Can92] D. Cann, "Retire Fortran? A Debate Rekindled", *Comm. ACM*, **35**(8), August 1992, pp. 81–89.

[CDG+98a] A.A. Chien, J. Dolby, B. Ganguly, V. Karamecheti, and X. Zhang, "High Level Parallel Programming: the Illinois Concert System", *Submitted for publication*, 1998.

[CHKLT92] S. Cox, S.-Y. Hing, P.H.J. Kelly, J. Liu, and F. Taylor, "An Implementation of Static Functional Process Networks", *Proc. PARLE '92 — Parallel Architectures and Reduction Languages Europe*, Springer-Verlag LNCS 605, 1992, pp. 497–512.

[CGKL98] M.M.T. Chakravarty, Y. Guo, M. Köhler, and H.C.R. Lock, "Goffin: Higher-Order Functions Meet Concurrent Constraints", *Science of Computer Programming*, **30**(1-2), 1998, pp. 157–199.

[Col88] M.I. Cole, *Algorithmic Skeletons*, Pitman/MIT Press (1988).

[DdMO+91] M. Danelutto, R. di Meglio, S. Orlando, S. Pelagatti, and M. Vanneschi, "The P^3L Language: An Introduction", Technical Report HPL-PSC-91-29, Hewlett-Packard Laboratories, Pisa Science Centre, December, 1991.

[DGTY95] Darlington, J., Guo, Y., To, H.W., and Yang, J., "Parallel Skeletons for Structured Composition", *Proc. Fifth ACM Conf. on Principles & Practice of Parallel Programming (PPoPP)*, Santa Barbara, CA, July 19–21, 1995, pp. 19–28.

[FMS+95] J. Feo, P. Miller, S. Skedziewlewski, S. Denton, and C. Soloman, "Sisal 90", *Proc. HPFC '95 — High Performance Functional Computing*, Denver, CO, April 9-11, 1995, pp. 35–47.

[FoCh95] C. Foisy, and E. Chailloux, "Caml Flight: a Portable SPMD Extension of ML for Distributed Memory Multiprocessors", *Proc. HPFC '95 — High Performance Functional Computing*, Denver, CO, 1995, pp. 83–96.

[FoTa94] I. Foster, and S. Taylor, "A Compiler Approach to Scalable Concurrent-Program Design", *ACM TOPLAS*, **16**(3), 1994, pp. 577–604.

[GBJGL98] D. Gannon, P. Beckman, E. Johnson, T. Green, M. Levine "HPC++ and the HPC++Lib Toolkit", Technical Report, Indiana University, `http://www.extreme.indiana.edu/hpc++/index.html`.

[GeCa92] D. Gelernter, and N. Carriero, "Coordination Languages and Their Significance", *CACM*, **32**(2), February, 1992, pp. 97–107.

[GoHu86] B.F. Goldberg and P. Hudak, "Alfalfa: Distributed Graph Reduction on a Hypercube Multiprocessor" In *Workshop on Graph Reduction, Santa Fé, New Mexico*, Springer-Verlag LNCS 279, September 1986, pp. 94–113.

[Gol88] B. F. Goldberg, "Multiprocessor Execution of Functional Programs", *Intl. Journal of Parallel Programming*, **17**(5), 1988, pp. 425–473.

[GMS93] R. G. Goldsmith, D. L. McBurney, and M. R. Sleep, "Parallel Execution of Concurrent Clean on ZAPP", Chapter 21 of M. R. Sleep, M. J. Plasmeijer, and M. C. J. D. van Eekelen (eds.), *Term Graph Rewriting: Theory and Practice*, Wiley, 1993.

[Hal85] R. Halstead, "MultiLisp: A Language for Concurrent Symbolic Computation", *ACM TOPLAS*, **7**(4), 1985, pp. 501–538.

[Hals96] R. Halstead, "Understanding the performance of parallel symbolic programs" Parallel Symbolic Languages and Systems (Workshop Proceedings) LNCS 1068, Springer-Verlag, Apr 1996.

[HFA*96] P.H. Hartel *et al* "Pseudoknot: a float-intensive benchmark for functional compilers" *Journal of Functional Programming*, **6**(4), 1996.

[HHLT97] K. Hammond, C.V. Hall, H.-W. Loidl, and P.W. Trinder, "Parallel Cost Centre Profiling", *Proc. 1997 Glasgow Workshop on Functional Programming*, Ullapool, Scotland, September 1997.

[HLP95] K. Hammond, H-W. Loidl, and A. Partridge. Visualising Granularity in Parallel Programs: A Graphical Winnowing System for Haskell. In *HPFC'95 — High Performance Functional Computing*, pages 208–221, Denver, Colorado, April 10–12, 1995.

[HCAA93] J. Hicks, D. Chiou, B.S. Ang and Arvind, "Performance Studies of Id on the Monsoon Dataflow Processor", *Journal of Parallel and Distributed Computing*, July 1993.

[Hil94] J. M. D. Hill, *Data-parallel lazy functional programming*, PhD Thesis, Department of Computer Science, Queen Mary and Westfield College, University of London, 1994.

[HLB95] J. Hammes, O. Lubeck and W. Böhm, “Comparing Id and Haskell in a Monte Carlo Photon Transport Code”, *Journal of Functional Programming*, **5**(3), July 1995, pp. 283–316.

[Hud86] P. Hudak, “Para-Functional Programming”, *IEEE Computer*, **19**(8), 1986, pp. 60–71.

[Hud91] P. Hudak, “Para-Functional Programming in Haskell”, In *Parallel Functional Languages and Computing*, ACM Press (New York) and Addison-Wesley (Reading, MA), 1991, pp. 159–196.

[Hug89] R.J.M. Hughes. Why Functional Programming Matters. *The Computer Journal*, 32(2):98–107, April 1989.

[JaCo94] C.B. Jay, and J.R.B. Cockett, “Shapely Types and Shape Polymorphism”, *Proc. ESOP '94 — 5th European Symposium on Programming*, Edinburgh, U.K., Springer-Verlag LNCS, April 1994, pp. 302–316.

[JCSS97] C.B. Jay, M.I. Cole, M. Sekanina, and P.A. Steckler, “A Monadic Calculus for Parallel Costing of a Functional Language of Arrays”, *Proc. Euro-Par '97*, Springer-Verlag LNCS, Passau (Germany), 1997.

[Jarv96] S.A. Jarvis, “Profiling large-scale lazy functional programs” PhD thesis, Department of Computer Science, University of Durham, 1996.

[JDH98] S. Junaidu, A.J.T. Davie, and K. Hammond “Naira: A Parallel^2 Haskell Compiler”, *Proc. IFL '97 — Intl. Workshop on the Implementation of Functional Languages*, September 10-12, St. Andrews, Scotland, Springer-Verlag LNCS 1467, 1997, pp. 215–231.

[Joh87] T. Johnsson, *Compiling Lazy Functional Languages*, Ph.D. Thesis, Department of Computer Science, Chalmers University of Technology, Gothenborg, Sweden, 1987.

[Jon96] M.P. Jones, Hugs 1.3 The Haskell User's Gopher System User Manual, Technical Report NOTT-CS-TR-96-2, Nottingham University, August 1996.

[Jun98] S. Junaidu, *A Parallel Functional Language Compiler for Message Passing Multicomputers*, PhD thesis, School of Mathematical and Computational Sciences, St Andrews University, Scotland, March 1998.

[Kel89] P.H.J. Kelly, *Functional Programming for Loosely-Coupled Multiprocessors*, Pitman/MIT Press, 1989.

[Kes95] M.H.G. Kesseler, “Constructing Skeletons in Clean: the Bare Bones”, *Proc. HPFC '95 — High Performance Functional Computing*, Denver, CO, 1995, pp. 182–187.

[Kes96] M.H.G. Kesseler, *The Implementation of Functional Languages on Parallel Machines with Distributed Memory*, PhD Thesis, Wiskunde en Informatica, Katholieke Universiteit van Nijmegen, The Netherlands, 1996.

[KiOD98] D.J. King, and J.T. O'Donnell, “Connection Machine Haskell”, Internal Report, Glasgow University, 1998.

[KLB91] H. Kingdon, D. R. Lester, and G. L. Burn. “The HDG-Machine: a Highly Distributed Graph-Reducer for a Transputer Network”, *The Computer Journal*, **34**(4), 1991.

[LANL98] Lawrence Livermore National Laboratories, “Sisal Performance Data”, `http://www.llnl.gov/sisal/PerformanceData.html`, 9th June 1998.

[LaPe95] J. Launchbury and S.L. Peyton Jones, "State in Haskell", *Lisp and Symbolic Computation*, **8**(4), Dec. 1995, pp. 293–342.

[LMT+97] H.-W. Loidl, R. Morgan, P.W. Trinder, S. Poria, C. Cooper, S.L. Peyton Jones, and R. Garigliano. "Parallelising a Large Functional Program; Or: Keeping LOLITA Busy", *Proc. IFL'97 — International Workshop on the Implementation of Functional Languages*, University of St. Andrews, Scotland, UK, September 10–12. Springer-Verlag LNCS 1467, pp. 199-214.

[LoHa96] H.-W. Loidl, and K. Hammond, Making a Packet: Cost-Effective Communication for a Parallel Graph Reducer", *Proc. IFL'96 — International Workshop on the Implementation of Functional Languages*, LNCS 1268, Springer-Verlag, Bad Godesberg, Germany, pp. 184–199.

[Loi98] H-W. Loidl, *Granularity in Large-Scale Parallel Functional Programming*, PhD Thesis, Department of Computer Science, University of Glasgow, 1998.

[LoTr97] H.-W. Loidl, and P.W. Trinder, "Engineering Large Parallel Functional Programs", *Proc. IFL'97 — International Workshop on the Implementation of Functional Languages*, University of St. Andrews, Scotland, September 10–12, 1997, Springer-Verlag LNCS 1467, pp. 179–198.

[Mat93] J.S. Mattson Jr., *An Effective Speculative Evaluation Technique for Parallel Supercombinator Graph Reduction*, PhD thesis, Dept. of Computer Science and Engineering, University of California, San Diego, 1993.

[MiSc95] G. Michaelson and N. Scaife, "Prototyping a parallel vision system in Standard ML", *Journal of Functional Programming*, **5**(3), July 1995, pp. 345–382.

[MiHu95] R. Mirani, and P. Hudak, "First-Class Schedules and Virtual Maps", *Proc. FPCA '95 — Functional Programming and Computer Architecture*, La Jolla, CA, June, 1995, pp. 78–85.

[MKH91] E. Mohr, D.A. Kranz, and R.H. Halstead, "Lazy Task Creation – a Technique for Increasing the Granularity of Parallel Programs", *IEEE Transactions on Parallel and Distributed Systems*, **2**(3), July 1991, pp. 264–280.

[MPI94] MPI Forum, "MPI: A Message-Passing Interface Standard", *International Journal of Supercomputer Applications*, **8**(3/4), 1994.

[NAH93] Nikhil, R.S., Arvind and Hicks, J., "pH language proposal", DEC Cambridge Research Lab Tech. Rep. 1993.

[Nik95] R.S. Nikhil, "Parallel Symbolic Computing in Cid", *Proc. Wkshp. on Parallel Symbolic Computing*, Beaune, France, Springer-Verlag LNCS 1068, , October 1995, pp. 217–242.

[NSvEP91] E.G.J.M.H. Nöcker, J.E.W. Smetsers, M.C.J.D. van Eekelen, and M.J. Plasmeijer, "Concurrent Clean", *Proc. PARLE '91 — Parallel Architectures and Reduction Languages Europe*, Springer Verlag LNCS 505/506, 1991, pp. 202–220.

[Ost93] G. Ostheimer, *Parallel Functional Programming for Message Passing Multicomputers*, PhD Thesis, University of St Andrews, 1993.

[PHAB+97] J.C. Peterson, K. Hammond (eds.), L. Augustsson, B. Boutel, F.W. Burton, J.H. Fasel, A.D. Gordon, R.J.M. Hughes, P. Hudak, T. Johnsson, M.P. Jones, S.L. Peyton Jones, A. Reid, and P.L. Wadler, *Report on the Non-Strict Functional Language, Haskell, Version 1.4*, 1997.

[Pey87] S.L. Peyton Jones, *The Implementation of Functional Programming Languages*, Prentice Hall International, 1987.

[Pey96] S.L. Peyton Jones, "Compiling Haskell by Program Transformation: a Report from the Trenches", *Proc. ESOP '96 — European Symposium on Programming*, Linköping, Sweden, April 22–24, 1996, Springer-Verlag LNCS 1058, pp. 18–44.

[PGF96] S.L. Peyton Jones, A.D. Gordon, and S.O. Finne, "Concurrent Haskell", *Proc. POPL '96 — 23rd ACM Symposium on Principles of Programming Languages*, St Petersburg Beach, Fl, January 1996, pp. 295–308.

[PlvE93] M.J. Plasmeijer, and M. van Eekelen, *Functional Programming and Parallel Graph Rewriting*, Addison-Wesley, 1993.

[PoSm93] A.P. Poulovassilis, and C. Small, "A Domain-Theoretic Approach to Logic and Functional Databases", *Proc. VLDB '93 — 19th Intl. Conf. on Very Large Databases*, 1993, pp. 415–426.

[PVM93] Oak Ridge National Laboratory, University of Tennessee, "Parallel Virtual Machine Reference Manual, Version 3.2", August 1993.

[Rab93] F.A. Rabhi, "Exploiting Parallelism in Functional Languages: a 'Paradigm-Oriented' Approach", In *Abstract Machine Models for Highly Parallel Computers*, P. Dew, and T. Lake (eds.), Oxford University Press, 1993.

[Roe91] P. Roe *Parallel Programming using Functional Languages*, PhD thesis, Department of Computing Science, University of Glasgow, February 1991.

[SaPe95] P.M. Sansom, and S.L. Peyton Jones, "Time and Space Profiling for Non-strict Higher Order Functional Languages", *Proc. POPL '95 — 22nd ACM Symposium on Principles of Programming Languages*, San Francisco, CA, 1995.

[SaPe97] P.M. Sansom, and S.L. Peyton Jones, "Formally Based Profiling for Higher-Order Functional Languages", *ACM Transactions on Programming Languages and Systems*, **19**(2), 1997, pp. 334–385.

[Sch95] W. Schreiner, "Application of a Para-Functional Language to Problems in Computer Algebra", *Proc. HPFC '95 — High Performance Functional Computing*, Denver, CO, 1995, pp. 10–24.

[Ser98] J. Serot, "Parallel FP: the Pragmatical Way", In*Research Directions in Parallel Functional Programming*, K. Hammond and G. Michaelson (eds.), To appear, Springer-Verlag, 1998.

[SkTa98] D.B. Skillicorn, and D. Talia, "Models and Languages for Parallel Computation", To appear in *ACM Computing Surveys*, 1998.

[SHM97] D.B. Skillicorn, J.M.D. Hill, and W.F. McColl, "Questions and answers about BSP" , *Scientific Programming*, **6**(3), 1997, pp. 249-274.

[SMTD+95] E. Smirni, A. Merlo, D. Tessera, G. Haring, and G. Kotsis, "Modeling Speedup of SPMD Applications on the Intel Paragon: a Case Study", *Proc. HPCN '95 — High Performance Computing and Networks*, Milan, Italy, 1995, pp. 94–101.

[SOHW+96] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI: The Complete Reference*, MIT Press, 1996.

[Tay93] F.S. Taylor, "The Design and Implementation of the Caliban Compiler Phases", In *FAST: The Collected Papers*, Internal Report CSTR 93-15, University of Southampton, 1993.

[Tay97] F.S. Taylor, *Parallel Functional Programming by Partitioning*, PhD Thesis, Department of Computing, Imperial College, London, 1997.

[Tar75] Tarjan R.E. “Efficiency of a good, but not linear set union algorithm”, *Journal of the ACM*, **22**, 1975, pp. 215–225.

[THMPP96] P.W. Trinder, K. Hammond, J.S. Mattson Jr., A.S. Partridge, and S.L. Peyton Jones, “GUM: A Portable Parallel Implementation of Haskell”. *Proc. PLDI '96 — Programming Language Design and Implementation*, Philadelphia, PA, May 1996, pp. 78–88.

[THLP98] P.W. Trinder, K. Hammond, H.-W. Loidl, and S.L. Peyton Jones, “Algorithm + Strategy = Parallelism”, *Journal of Functional Programming*, **8**(1), January, 1998.

[Wad97] P.L. Wadler, “How to Declare an Imperative”, *ACM Computing Surveys*, **29**(3), September 1997, pp. 240–263.

[WiOD97] N. Winstanley, and J.T. O'Donnell, “Parallel Distributed Programming with Haskell+PVM”, *Proc. Euro-Par '97*, Springer-Verlag LNCS 1300, August 1997, pp. 670-677.

[WuHa96] J. Wu, and L. Harbird, “A Functional Database System for Road Accident Analysis”, *Advances in Engineering Software*, **26**(1), 1996, pp. 29–43.

Appendix A: Source of Parfact

The following is the GPH source code for the simple test program measured in section 2.2.

```
-- parfact:
-- parallel version of a factorial-like function (i.e. divide-and-conquer)
--
-------------------------------------------------------------------------------

module Main(main) where

import Parallel

pfc :: Int -> Int -> Int -> Int
pfc x y c
   | y - x > c = f1 `par`
                 (f2 `seq` (f1+f2))
   | x == y    = x
   | otherwise = pf x m + pf (m+1) y
   where
      m  = (x+y) `div` 2
      f1 = pfc x m c
      f2 = pfc (m+1) y c

pf :: Int -> Int -> Int
pf x y
   | x < y     = pf x m + pf (m+1) y
   | otherwise = x
   where
      m  = (x+y) `div` 2

parfact x c = pfc 1 x c

main
 =  let x = 8399608
        c = 8192
    in
        appendChan stdout
                   (show (parfact x c))
                   exit done
```