Low level Architecture-independence of Glasgow Parallel Haskell (GpH)

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

September 16, 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 focus here on the low-level architecture independence of GUM: the high-level architecture independence of the GpH programming model is discussed elsewhere.

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 the efficiency and speedups delivered by GUM on a range of architectures.

1 Introduction

The goal of architecture-independent parallel programming is to deliver acceptable 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 achieve 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.

- **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 determines how many threads are exploited. Such 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? We investigate in the context of the GUM implementation on Glasgow Parallel Haskell (GpH), a modest extension of Haskell. In particular we address the following issues.

- How do we design a parallel runtime system to be architecture independent (Section 2)?
- Can such a runtime system be efficient and deliver speedups across a range of parallel architectures (Section 3)? We investigate by measuring a simple program on seven architectures. The best results are an efficiency of 95%, and a speedup by a factor of 74 on 97 processors.
- Can real programs deliver speedups on multiple architectures (Section 4)? We measure efficiency and speedup of a 1500 line GpH program on four architectures.

This paper focusses on the high-level the low-level architecture independence of GUM, i.e. the implementation of GpH. GpH also supports higher-level architecture independence because the programming model supports the systematic development of programs for multiple parallel architectures. This is elaborated in a forthcoming article [TBDH+99].

2 Architecture Independence of GUM

GUM is the 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 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.

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

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.

3 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 4.1 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

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

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.

3.1 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.

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

3.3 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.

4 Large Architecture-independent Programs

To be truly architecture-independent a parallel language implementation must be efficient and deliver speedups for real, i.e. large, programs. This section reports efficiency and speedup measurements for Accident Blackspots, a 1500-line data-intensive GPH program. We are in the process of porting the 6000-line Naira compiler [JDH98] to a second parallel architecture.

4.1 Accident Blackspots

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

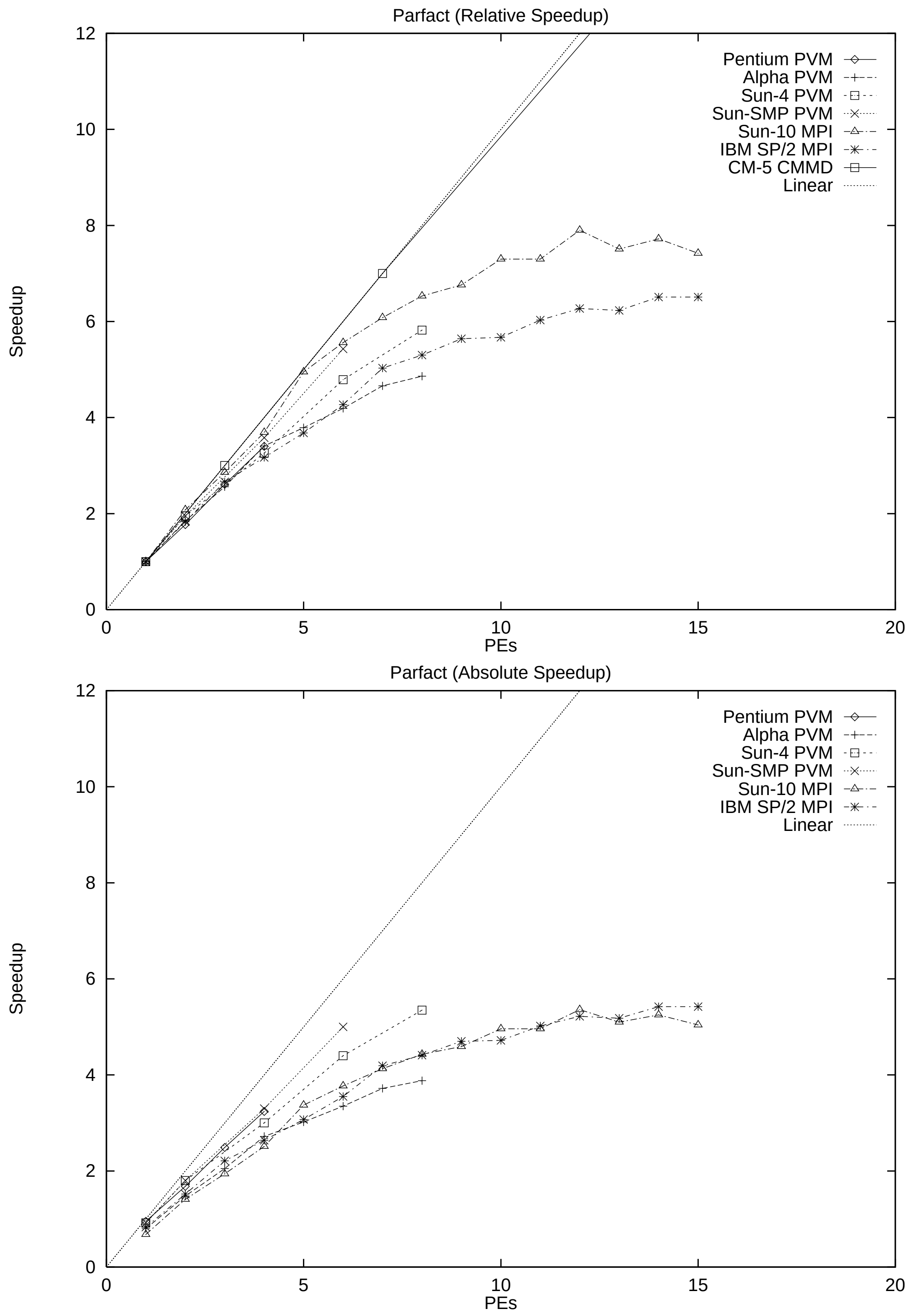
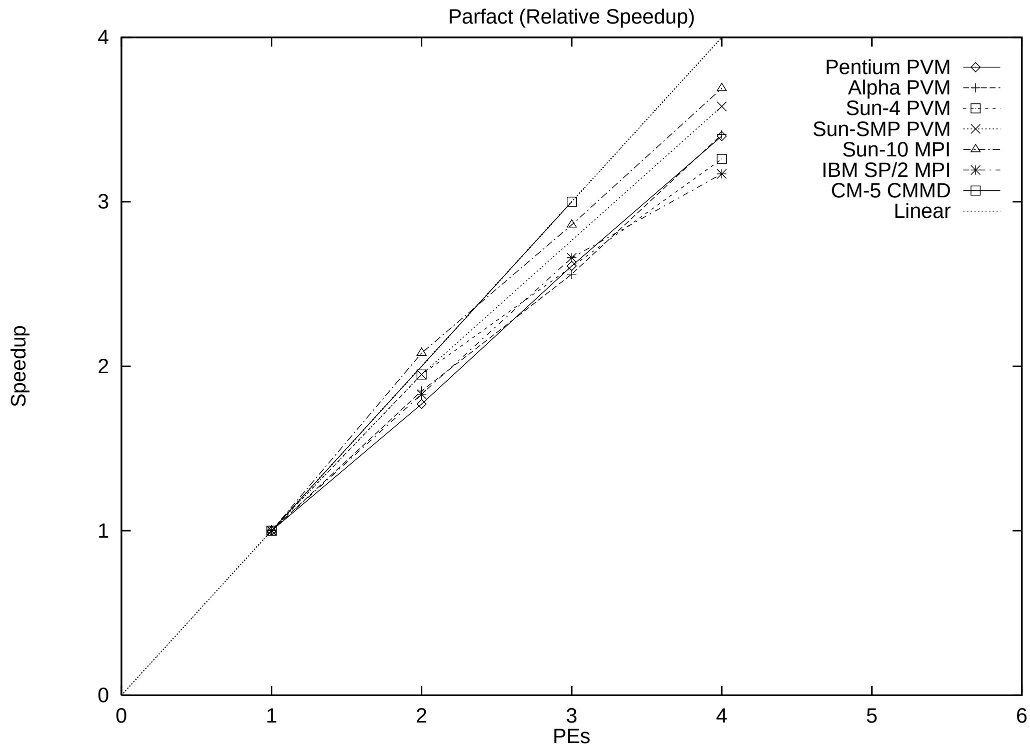


Figure 1: Relative and Absolute Speedups for `parfact`



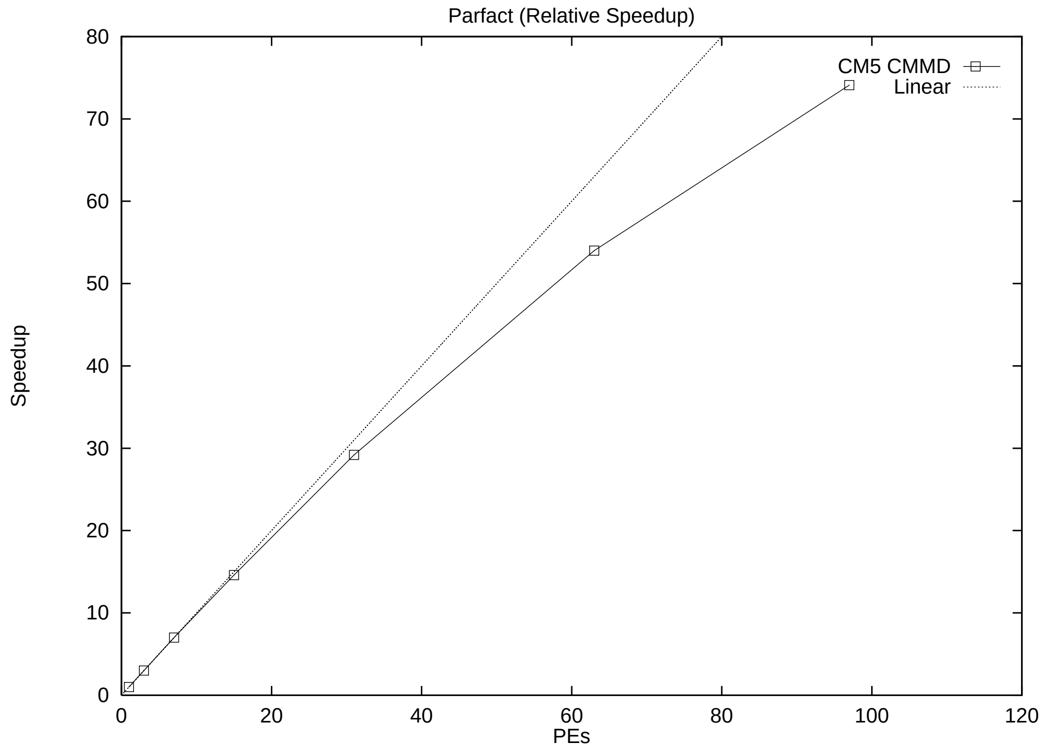


Figure 2: Relative Speedup for `parfact` – Detail up to 4 PEs (top) and CM-5 (bottom)

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 3: Single-Processor Efficiency: Blackspots

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.

4.2 Parallelisation

The development of Blackspots for multiple architectures is described elsewhere [TBDH$^+$99]. The essentials are that the GHC-compiled sequential Haskell version is 9 times faster than the original interpreted PFL [PoSm93] implementation. Several parallel versions are constructed; the version measured here divides the accidents into geographical areas (or *tiles*), that can be partitioned in parallel before aggregating the results.

4.3 Multi-Architecture Measurements.

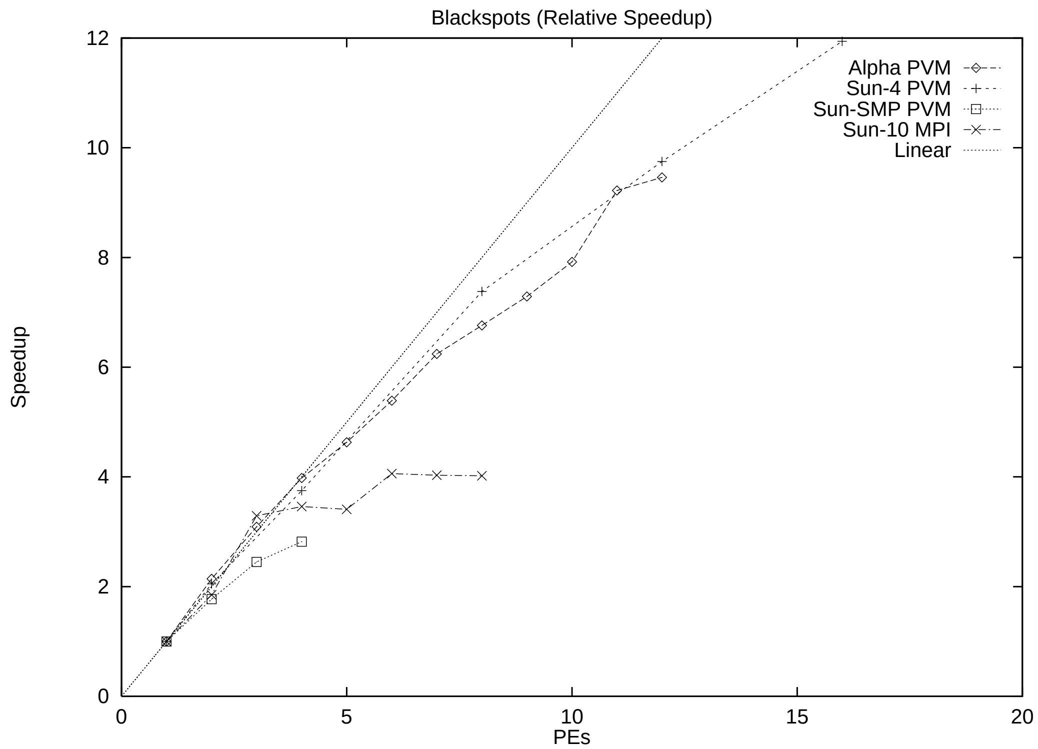
The original data set of 7310 accident reports occupies 0.3Mb and is too small to get good results on parallel machines. For the purposes of this section, the data is replicated 6 times. The replicated data occupies 1.8 Mb and is split into 40 tiles with two different sizes. There are 32 *small* tiles, each containing approximately 1000 accidents and occupying 37Kb, and 8 *large* tiles each containing approximately 2000 accidents and occupying 73Kb. 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.

4.3.1 Efficiency

Table 3 shows sequential times and parallel efficiency for the Blackspots program, on four of the architectures described in section 3.1. 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.

4.3.2 Speedups

Figure 3 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



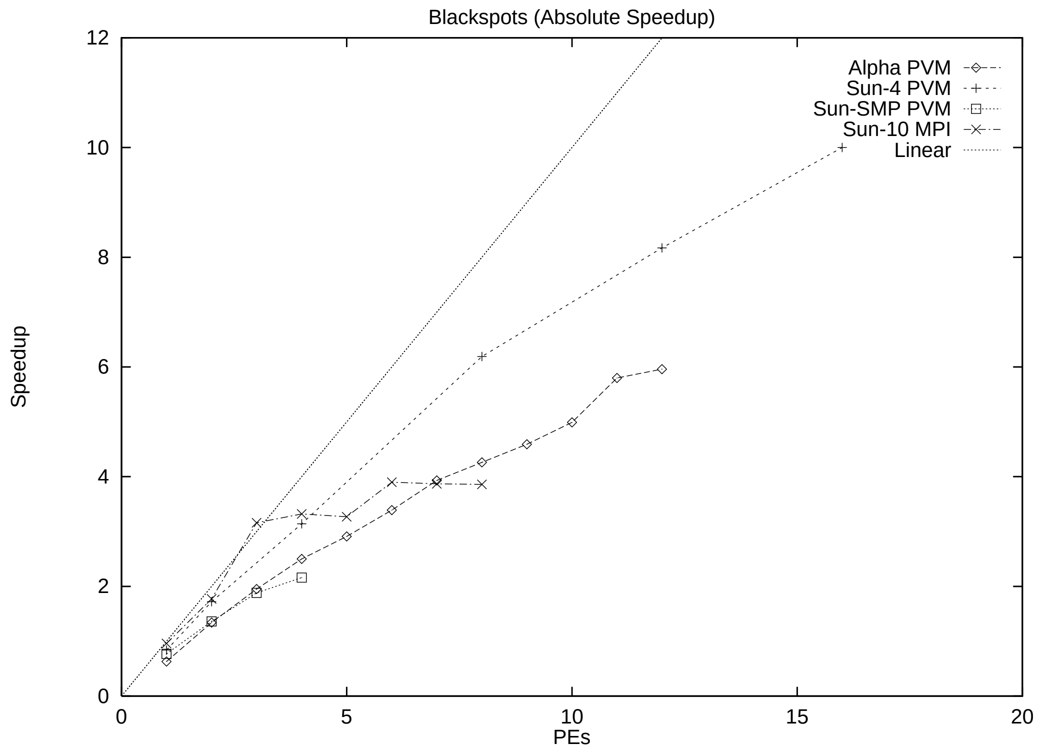


Figure 3: Relative and Absolute Speedups for `blackspots`

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 Related Work

In a functional context we are aware of very few portable parallel implementations, and no other approaches where architecture-independence has been satisfactorily demonstrated. We survey 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 Conclusion

We have investigated the architecture-independence of an implementation of a pure functional language with dynamic and mainly-implicit parallelism: the GUM implementation of GPH. We have outlined the architecture-independent design of GUM, and measured its efficiency and the speedups delivered for a simple program on a range of architectures. We have likewise measured a non-trivial program on 4 architectures.

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.

[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.

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

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[TBDH+99] P.W. Trinder, E. Barry, M.K. Davis, K. Hammond, S.B. Junaidu, U. Klusik, H-W. Loidl and S.L. Peyton Jones, "GPH: An Architecture-independent Functional Language" *submitted to* IEEE Trans. on Software Engineering.

[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 3.

```
-- 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
```