The Multi-Architecture Performance of the Parallel Functional Language GPH*

Philip W. Trinder[1], Hans-Wolfgang Loidl[1], Ed. Barry Jr.[†], M. Kei Davis[2], Kevin Hammond[3], Ulrike Klusik[4], Simon L. Peyton Jones[5], Álvaro J. Rebón Portillo[3]

[1] Heriot-Watt University, Edinburgh, U.K; {trinder,hwloidl}@cee.hw.ac.uk
[2] Los Alamos National Laboratory, U.S.A; kei@lanl.gov
[3] University of St. Andrews, U.K; {kh,alvaro}@dcs.st-and.ac.uk
[4] Philipps–University Marburg, Germany; klusik@mathematik.uni-marburg.de
[5] Microsoft Research Ltd, Cambridge, U.K; simonpj@microsoft.com

Abstract. In principle, functional languages promise straightforward architecture-independent parallelism, because of their high level description of parallelism, dynamic management of parallelism and deterministic semantics. However, these language features come at the expense of a sophisticated compiler and/or runtime-system. The problem we address is whether such an elaborate system can deliver acceptable performance on a variety of parallel architectures. In particular we report performance measurements for the GUM runtime-system on eight parallel architectures, including massively parallel, distributed-memory, shared-memory and workstation networks.

1 Introduction

Parallel functional languages have several features that should, in theory, enable good performance on a range of platforms. They are typically only *semi-explicit* about parallelism, containing limited explicit control of parallel behaviour. Instead the compiler and runtime-system extract and exploit parallelism, with the programmer controlling a few key aspects of the parallelism explicitly. Purely functional languages also have *deterministic parallelism*: the value computed by a program is not dependent on its parallel behaviour, thereby avoiding the complications of race conditions and deadlocks. Many pure functional language implementations support *dynamic resource allocation*: the resources of the parallel machine are allocated during program execution. Dynamic resource allocation relieves the programmer from architecture dependent tasks such as specifying exactly what computations are to be executed where.

The cost of high-level, dynamically-managed parallelism is a complex compiler and/or runtime-system. Can such a sophisticated system deliver acceptable performance on very different parallel architectures? We tackle this question in

* Supported by research grants GR/M 32351, GR/M 55633, GR/M 93379 from UK's EPSRC and APART fellowship 624 from the Austrian Academy of Sciences.

the context of our GUM implementation of Glasgow Parallel Haskell (GpH), a non-strict functional language. GUM performs parallel graph reduction, and many aspects of the parallelism are determined dynamically e.g. threads are dynamically created and allocated to processors. GUM is designed to be portable, and uses a message-passing model (Sect. 2). Performance is measured for a simple test program with good parallel behaviour (Sect. 3) as well as for one larger application with irregular parallelism and complex data structures (Sect. 4). This complements our earlier research on parallelising substantial Haskell applications [1] and developing a suite of simulation and profiling tools.

2 The GUM Runtime System

GUM is the runtime-system for GpH [7], a parallel variant of the Haskell lazy functional language. Being a parallel graph reduction machine [3], GUM represents an architecture-independent abstract machine-model appropriate to both shared- and distributed-memory architectures. In this model both data and program are represented via graph structures. Executing a program means rewriting a graph with its result. Semi-explicit parallelism in GpH requires the programmer to annotate expressions that can be evaluated in parallel. The runtime-system then dynamically distributes data and work among the available processors. Potential parallelism may be subsumed [3] by existing threads in a similar way as in the lazy task creation mechanism [2], thereby dynamically increasing thread granularity. This dynamic granularity control, together with overlapping computation with communication (latency hiding), is crucial for achieving high performance on very different parallel architectures. Communication between threads is realised via a shared heap with implicit synchronisation on graph structures shared between several threads (implemented via message-passing on top of PVM or MPI). For efficient and portable compilation we use the Glasgow Haskell Compiler [4] (GHC), a state-of-the-art optimising compiler for Haskell. The design and implementation of GUM are discussed in detail in [7].

3 Measurement Setup

In our measurements we have used eight machine configurations: one MPP, a 97-processor Connection Machine CM-5 with the native CMMD communications library; one DMP, a 16-processor IBM SP/2 with MPI; one SMP, a 6-processor Sun SparcServer with PVM; a 56-node Beowulf cluster with 450MHz Intel Pentium II processors, 384MB RAM and 8GB local disk; and four networks of workstation (NOWs) all with PVM. The Beowulf uses a 100Mb/s fast Ethernet switch, the NOWs use standard Ethernet on the same subnet.

In order to assess the overhead of the GUM runtime-system we have measured `parfact`, a simple binary divide-and-conquer program computing the sum of a given interval with little communication and no software bound for the achievable parallelism. The `parfact` program and additional details and measurements are available in [5]. The CM-5 achieves the best relative speedup of

Table 1. Single-processor efficiency of `blackspots`

Class and Architecture	Sequential Runtime (s)	Efficiency	Class and Architecture	Sequential Runtime (s)	Efficiency
SMP			**Workstation-net**		
Sun-SMP PVM	135.2	77%	Digital Alpha PVM	378.6	63%
			Sun-4/15 PVM	815.4	84%
			Sun-10 PVM	289.1	96%
			Pentium PVM	109.4	96%
			Beowulf PVM	19.8	90%

74.1 on 97 processors, without approaching a parallelism bound imposed by either the hardware or the GUM runtime-system. These speedups are significantly better than the PVM and MPI versions, 5.82 and 6.53 on 8 processors, respectively, indicating the high costs of the portable communication libraries. Furthermore the MPI version on the IBM SP/2 suffers from a slow (sequential) startup. The Beowulf achieves a relative speedup of 33.1 on 56 processors. However, its parallel efficiency (33.1/56 = 59%) is not as good as the CM-5's (54.0/63 = 86%).

4 Accident Blackspots: a Larger GpH Program

The *Accident Blackspots* program determines locations where two or more traffic accidents have occurred, based on a set of police accident records as input. A number of criteria can be used to determine whether two accident reports are for the same location, and each criteria partitions the set. The problem amounts to combining several partitions of a set into a single partition, or *union find.* The program comprises 1,500 lines of Haskell code and additional details are available in [1]. The parallel (GpH) version of the algorithm uses a geometric partitioning of the input data into 32 small and 8 large tiles. Evaluation strategies [6] are used to define the parallel evaluation over these tiles.

The best sequential *efficiency* (see Table 1) is obtained for machines based on SPARC and Pentium processors, since these architectures are best supported by the GHC compiler itself. The simpler `parfact` program has higher efficiency in most cases [5]. In GUM sequential efficiency is dominated by the costs for managing potential parallelism (essentially adding a pointer to an array of possible tasks) and for locking graph structures representing work. The former, although cheap in itself, prohibits optimisations because it requires the graph structures to exist in the heap. The latter requires several instructions and, without the help of sharing analysis, increases with program size.

Fig. 1 shows that the small amount of communication required by the geometric partitioning enables good *speedups* even on the NOWs: 11.94 relative, 10.00 absolute on 16 Suns and 9.46 relative, 5.96 absolute on 12 Alphas. The Beowulf cluster profits from its high efficiency and its scalability. As a result

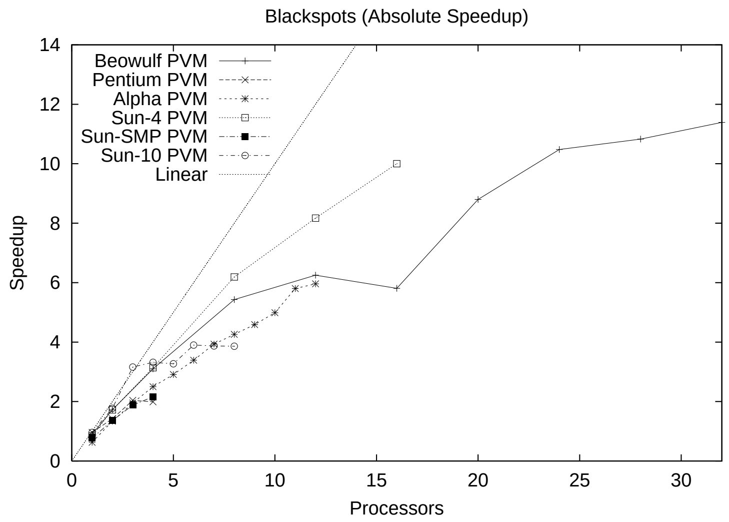

Fig. 1. Absolute speedups for `blackspots`

it delivers the highest absolute speedup for this application: 11.39 (with larger input data 18.69) on 32 processors. Occasional drops in performance reveal that the dynamic scheduling is not always effective for this rather coarse-grained application. The overall poorer performance for the Pentium, Alpha and Sun-10 PVM NOWs is due to the higher ratio of communication costs to processor speed, exacerbated by the unusually low efficiency of the Digital Alpha. In this configuration the available parallelism is not sufficient to effectively hide communications latency. In the Beowulf cluster, with lower communications costs, this effect is less pronounced. In order to obtain good performance on machines with such characteristics we could perform architecture-dependent tuning, e.g. splitting the data into more, smaller tiles. The Sun-10 NOW exhibits a super-linear speedup for 3 processors. We believe this is due to reduced garbage collection costs in a parallel setting (with n processors we have n times the sequential heap available). The rather low speedup on the Sun SMP (2.82 relative, 2.16 absolute) is partly due to the higher overall performance of the processors and partly due to the competition with other user processes when performing the measurements. Our implementation would also profit from direct support of shared-memory — currently we use PVM or MPI even on SMPs.

5 Conclusion

In this paper we have assessed the architecture-independent performance of the GUM runtime-system for GPH by taking measurements on eight platforms, ranging from MPPs to networks of workstations. From the `parfact` results we conclude that, for a program with little communication and no software bound on the parallelism, GUM is efficient on all platforms (at least 83%), capable of delivering acceptable speedups on all architectures, and also capable of massive parallelism (a relative speedup of 74 on a 97-processor CM-5). From the `blackspots` results we conclude that GUM can achieve absolute speedups (up to 18.69 on 32 processors) for symbolic programs with irregular parallelism. We consider GPH to be best suited for such symbolic applications, obtaining moderate speedups with only minimal code changes.

Our measurements suggest several improvements of GUM, which are currently being examined in the new implementation of the runtime-system. The implicit work and data distribution could be improved, e.g. by refining the work-stealing algorithm, by constructing clusters of data to avoid excessive communication or by supporting the migration of running threads. Furthermore, better interaction between granularity control and the generation of parallelism could be achieved via a low-watermark scheme maintaining a minimal amount of parallelism despite thread subsumption.

References

1. H-W. Loidl, P.W. Trinder, K. Hammond, S.B. Junaidu, R.G. Morgan, and S.L. Peyton Jones. Engineering Parallel Symbolic Programs in GPH. *Concurrency — Practice and Experience*, 11(12):701–752, Oct. 1999. Available from [8].
2. E. Mohr, D.A. Kranz, and R.H. Halstead Jr. Lazy Task Creation: a Technique for Increasing the Granularity of Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, Jul. 1991. <URL:`ftp://crl.dec.com/pub/DEC/CRL/tech-reports/90.7.ps.Z`>
3. S.L. Peyton Jones, C. Clack, and J. Salkild. High Performance Parallel Graph Reduction. In *Parallel Architectures and Languages Europe (PARLE'89)*, LNCS 365, pp. 193–206, Eindhoven, The Netherlands, Jun. 1989. Springer-Verlag.
4. S.L. Peyton Jones, C.V. Hall, K. Hammond, W.D. Partain, P.L. Wadler. The Glasgow Haskell Compiler: a Technical Overview. In *Joint Framework for Information Technology Technical Conference*, pp. 249–257, Keele, U.K, Mar. 1993. See also <URL:`http://www.haskell.org/ghc`>
5. P.W. Trinder, Ed. Barry Jr., M.K. Davis, K. Hammond, S.B. Junaidu, U. Klusik, H-W. Loidl, S.L. Peyton Jones. Low Level Architecture-Independence of Glasgow Parallel Haskell (GPH). In *Glasgow Functional Programming Workshop*, draft proceedings, Pitlochry, Scotland, Sep. 1998. Available from [8].
6. P.W. Trinder, K. Hammond, H-W. Loidl, and S.L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, Jan. 1998. Available from [8].
7. P.W. Trinder, K. Hammond, J.S. Mattson Jr., A.S. Partridge, and S.L. Peyton Jones. GUM: a Portable Parallel Implementation of Haskell. In *Programming Language Design and Implementation (PLDI'96)*, pp. 79–88, Philadelphia, PA, May 1996. Available from [8].
8. GPH Web Pages. <URL:`http://www.cee.hw.ac.uk/~dsg/gph`>