Chapter 1

Making GUM More Flexible

Hans-Wolfgang Loidl[1] and No-One Else[2]

Abstract: This paper motivates and discusses extensions of the GUM parallel runtime-system for GPH (Glasgow Parallel Haskell) in order to achieve high and architecture-independent performance. Experiences with large GPH applications measured on several different architectures suggest concrete improvements of the current dynamic management of parallelism in GUM. Specifically, we plan to improve the data locality in parallel programs, reduce the overhead required for performing bookkeeping of potential parallelism, and support speculative parallelism in order to achieve additional parallelism. The research discussed in this paper is still in its early stages and therefore no new results are presented.

1.1 INTRODUCTION

In order to achieve high architecture-independent performance of arbitrary GPH (Glasgow Parallel Haskell) [THLP98] programs, a flexible and sophisticated runtime-system is called for. The current GUM runtime-system [THM$^+$96] achieves a high degree of architecture independence in its implementation but not necessarily in the performance of the parallel program. Therefore, the programmer may have to adjust the parallel program, i.e. by changing thread granularities, in order to achieve optimal performance on a particular architecture. Although this architecture-dependent tuning phase is usually rather short, we want to reduce it as much as possible. In the starting project discussed in this paper, we therefore aim at enhancing the runtime-system, so that it can dynamically adjust to the characteristics of the underlying architectures.

The planned improvements follow three main strands:

- reduce the overall amount of communication by improving the data locality in the parallel program;

[1]Department of Computing and Electrical Engineering, Heriot-Watt University, Edinburgh E-mail: `hwloidl@cee.hw.ac.uk`

[1]No Where University, New Nirvana E-mail: `nowhere@at.all`

- improve the support for massively parallel algorithms on moderately parallel architectures by reducing the overhead required for managing potential parallelism in the program;
- improve the support for moderately parallel algorithms on massively parallel machines by enabling speculative parallelism in order to increase the overall degree of parallelism.

In previous work [LH96] we have already studied the *costs of communication* in our parallel system by using the GRANSIM simulator [HLP95], which shares to a large extent the code for communication with the GUM runtime-system. As a result of these measurements we were able to make concrete suggestions on how to improve the performance of communication depending on the latency of the parallel machine. Furthermore, experience with several large applications [LTH+99] indicated the importance of controlling the data locality in the program, so as to avoid excessive communication due to unnecessarily scattered data structures.

An important issue of massively parallel programs is the *overhead of parallelism* imposed even on a single processor version. Ideally, we want such a version to be as efficient as the sequential version of the code. However, bookkeeping costs and the need to have explicit closures available in order to express potential parallelism often seriously degrade the performance of highly parallel applications. Independent work on lazy threads [GSC96] in a dataflow context has achieved some progress to this end. We plan to incorporate these ideas into our parallel graph reduction machine, with the goal to make the management of potential parallelism as cheap as possible.

Finally, our work on several large applications has shown the usefulness of some form of *speculative parallelism* [LTH+99]. This technique can be used to increase the degree of parallelism in moderately parallel programs, although the parallel work in some cases may prove redundant. In order to improve the support for speculative evaluation we plan to extend the runtime-system, so that threads that turn out to be redundant can be automatically killed by the runtime-system.

1.2 DATA CLUSTERING

In order to reduce the communication in a (massively) parallel system it is desirable to specify some form of data locality. However, this often refers to the underlying topology, yielding an architecture-dependent program. As an alternative we propose as a means to improve the data locality in a program, data clustering, which dynamically specifies that certain program variables should be kept on the same processor. Currently the GUM system does not provide any facility for keeping several pieces of data together. For example whenever one unevaluated list element is needed on another processor the whole sub-list beginning with this element is moved to this processor. We have found examples of programs where this communication mechanism leads to "thunk stealing", i.e. the repeated and unnecessary transfer of work between two processors evaluating the same overall data structures. Motivated by these observations, we have already studied variants of the basic communication mechanism, so-called packing and rescheduling schemes, and implemented them in GRANSIM. These mechanisms permit more detailed control over the amount of data being communicated [LH96]. As a result of these measurements we have concluded that in low latency systems, where communication is rather cheap, an even load distribution is more important than good data locality. In high latency systems, however, data locality becomes more important than the load distribution. Thus, a self-adjusting runtime-system should pick the appropriate communication mechanism based on the latency it can either determine at start-up time or via a specific configuration file.

In order to improve the performance of GUM on low latency systems we plan to implement task migration and to possibly propagate load information as part of the passive work distribution process. Task migration enables the runtime-system to move an entire task to another processor. Since task migration does not refer to the underlying topology and is a dynamic mechanism for load balancing it fits our overall approach of architecture-independent and dynamic management of parallelism. Studies on the GUM system [LMT+97] and on its predecessor, the GRIP system [HP90], have shown that in cases of extreme load unbalance task migration is a very important, though rather expensive, operation.

Another simple possibility of improving the load distribution would be to use existing messages that search for new work in the system (so called "fish" messages in GUM), to also record information on the load of the processors. This should provide a partial picture of the load distribution in the system and can give a hint where to look for new work next time. Previous research on load balancing in parallel graph reduction systems (PAM [LKI89], Alfalfa [Gol88]) suggests that already simple mechanisms, which only provide information about the load of neighbouring processors, suffice to achieve a rather even load distribution.

In order to improve the performance of GUM on high latency systems we plan to augment the parallel programming language GPH with constructs for defining clusters of data, with the idea that all elements of the cluster should always be kept on the same processor. With this approach the programmer has the possibility to control data locality without specifying in detail on which processor a certain data item should reside. One possible language syntax would extend the syntax for strategies with an additional clause for specifying data clusters e.g.

```
f x y  `using`  rnf x `par` rnf y  `clustering`  x `with` y
```

In this example both `x` and `y` would have to be evaluated on the same processor. While this prohibits real parallelism it maintains the potential of hiding communication costs that might be embedded in either of the two computations by overlapping the communication of one thread with the computation of the other. Obviously, this approach requires explicit names for clustered data structures. However, the strategic function application operator [LTH+99] can be generalised to also specify clustering information without introducing new names for intermediate results.

In order to implement data clustering a new cluster pool, similar to the existing spark pool, has to be implemented. Each cluster in this pool consists of several pointers into the heap, representing the pieces of data that should be kept on the same processor. When a request for one data item in the pool arrives the whole cluster will be sent to the demanding processor. This also increases the granularity of the communication in the system and should make more efficient use of the available bandwidth. When one element of a cluster is being evaluated the cluster is committed to the current processor. After that, demands for other data items in the cluster will be answered by spawning a thread locally rather than sending the unevaluated data to the demanding processor. In any case, evaluated data is simply copied. In order to minimise the implementation overhead, the retrieval of cluster information can be encoded in the already existing GALA and LAGA tables of GUM [HMP+95], which translate global addresses to local addresses and vice versa.

An alternative approach to introducing explicit clusters would be to extend the type system to express clustering information. For example, a compound type or its data constructors could be annotated to be "compact", indicating that all elements of this type have to reside on the same processor. In this case, it would be possible to define locality information separately from the computation. This type information could then be used to optimise the code, in this case to minimise communication. Although this approach has rather limited expressiveness its clarity in providing locality information suggests a closer investigation.

1.3 SPECULATIVE COMPUTATION

For some applications it is useful to start a parallel thread although its result is not necessarily needed. On massively parallel architectures idle processors can be used to perform potentially redundant computations to speed-up computation. However, if the number of such speculative threads exceeds the number of processors it is important not to waste computation on threads that have turned out to be redundant. This can be done by giving lower priorities to the speculative tasks, and propagating information on the neededness of the result from one speculative task to all its children.

One example for the use of speculative parallelism, so far without special support in the runtime-system, is the Lolita natural language engineering system. In Lolita several possible parse trees are analysed in order to deduce the proper meaning of an ambiguous sentence. Thus only one of these analyses is actually needed in the computation. As soon as the result is available all other computations should be terminated. The main problem with speculative computation lies in the treatment of these computations that turn out to be redundant: the computations, and all their children, have to be killed. Furthermore, no additional overhead should be paid when no speculation is used.

The two main operations in supporting speculative evaluation are the freezing of a partially completed evaluation and the killing of threads that turn out to be redundant. The former operation is closely related to a similar operation needed for handling interrupts in the concurrent version of Haskell [Rei98]: when an interrupt to kill a thread is received, the current state of the computation is frozen by traversing the stack and turning the data between update frames into partial application nodes thereby reconstructing the graph structure in the heap. (Reid discusses this process in detail in [Rei98].) If the data represented by the frozen evaluation is needed by another thread, the normal evaluation mechanism of the STG machine will copy the data into the stack again and thereby make use of the partial evaluations already made by the terminated thread. As an optimisation to this method the speculative thread may be turned into a "zombie" rather than being killed. This means that the thread is kept in the heap and also keeps its stack, but becomes inactive. Closures representing partially evaluated data keep a pointer to the zombie thread. In this case, when demanding partially evaluated data, the zombie thread would be reawakened, continuing its computation until updating the required data and then becoming a zombie again.

The key issues to be addressed by any mechanism for speculative evaluation are the following. Firstly, the overhead for permitting speculative parallelism should be kept to a minimum. To this end speculative threads would use the normal evaluation mechanism, keeping links to its children at the bottom of the stack. Secondly, the costs of freezing an evaluation and killing the threads must be low. In both cases a stack traversal will be necessary. With the zombie scheme it would be sufficient to keep links from partially evaluated pieces of data to the evaluating thread, in order to activate the thread again after it has been turned into a zombie. Copying data from the stack into the heap would be more expensive but could reuse a lot of existing code for concurrent evaluation. Finally, it must be possible to find all children of a thread that has become redundant in order to kill them, too. This can be achieved by reserving several words at the bottom of a speculative thread as pointers to future children of this thread. Obviously this would put a limit to the number of children per speculative thread, but this may also be desirable from a resource management point of view.

1.4 LAZY THREADS

In order to efficiently support massive parallelism, even on moderately parallel machines, which do not require a high degree of parallelism, it is important to minimise the bookkeeping overhead required to generate new threads. In other languages the number of generated threads is often made dependent on the number of available processor in order to avoid a flood of parallelism. In our system we want to avoid the explicit reference to such architecture dependent features in the program and control the number of threads dynamically as far as it is possible. Therefore, we plan to further reduce the overhead for keeping track of potential parallelism in the runtime-system.

In recent work, Goldstein [Gol97] has developed lazy threads, in order to reduce this bookkeeping overhead. The main idea is to avoid an explicit pool of potentially parallel computations, the spark pool in GUM. If additional parallelism is needed to keep the system busy the stack of an existing thread is traversed to find new work. This new piece of work is sent to the idle processor. This idea has been successfully implemented in several variants on a parallel dataflow machine [Gol97]. We plan to implement lazy threads on a graph reduction machine and assess the performance of this system. This work could also build on Rushall's PhD thesis [Rus95], which produced a prototype implementation of a similar mechanism within a simple graph reduction machine.

1.5 LANGUAGE SUPPORT

As a result of implementing the above new features the new parallel runtime-system will be highly parameterised. In order to choose a set of parameters fitting to a particular parallel machine we intend to use both information available at compile-time as well as run-time. The former information can characterise the structure of the parallelism in the program, for example using clustering annotations to capture information on data locality. Furthermore, critical information about the parallel machine, for example its latency, can be determined at the beginning of the program, at start-up time, and then be used to pick a certain communication mechanism.

Making use of such information at start-up time will result in a self-adjusting runtime-system, which adapts its behaviour to the characteristics of the underlying parallel machine. This is of particular importance when the parallel machine is a heterogenous network of sequential and parallel machines. Thus the system can avoid potential inefficiencies caused by the use of generic routines in the runtime-system. The latter is a known deficiency of GUM.

However, we do not want to rely totally on implicit methods for controlling parallelism. The programmer should be given high-level control of the new features without being burdened with specifying details of the parallelism. In particular, we want to embed the additional information into our framework of evaluation strategies [THLP98], which currently provide the possibility of specifying evaluation degree and evaluation order of GPH expressions. This has proven to be crucial in tuning the efficiency of several parallel algorithms [Loi97]. The example in Section 1.2 shows how data clustering can be specified in this framework.

1.6 SUMMARY

This paper has outlined the future development of the GPH language and its underlying runtime-system GUM. So far, this is just a wish-list of features we would like to have in our system, with the goal of achieving performance that does not depend on details of the underlying parallel machine. However, the usefulness of these features has become apparent in the implementation of several large systems, and is therefore solidly motivated. Furthermore, the current version of GUM is available on 8 different parallel architectures, which enables us to study the effect of the proposed enhancements on a variety of parallel architectures, in order to assess our progress towards truly architecture-independent performance.

Another development of GPH is the introduction of additional features to support distributed computation, culminating in the extended language GDH. We expect a fruitful cross-fertilization of these two developments, both being carried out by the same research group. Furthermore, we plan to support both the parallel runtime-system (GUM) and the parallel simulator (GRANSIM) in the future. Past experience has shown that both have important roles in the design and implementation of parallel GPH programs.

REFERENCES

[Gol88] B. Goldberg. Multiprocessor Execution of Functional Programs. *International Journal of Parallel Programming*, 17(5):425–473, October 1988.

[Gol97] S.C. Goldstein. *Lazy Threads: Compiler and Runtime Structures for Fine-Grained Parallel Programming*. PhD thesis, University of California, Berkeley, 1997.

[GSC96] S.C. Goldstein, K.E. Schauser, and D.E. Culler. Lazy Threads: Implementing a Fast Parallel Call. *Journal of Parallel and Distributed Computing*, 37(1):5–20, 1996.

[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.

[HMP+95] K. Hammond, J.S. Mattson Jr., A.S Partridge, S.L. Peyton Jones, and P. Trinder. GUM: a Portable Parallel Implementation of Haskell. In *IFL'95 — International Workshop on the Parallel Implementation of Functional Languages*, Bastad, Sweden, September 1995.

[HP90] K. Hammond and S.L. Peyton Jones. Some Early Experiments on the GRIP Parallel Reducer. In *IFL'90 — International Workshop on the Parallel Implementation of Functional Languages*, pages 51–72, Nijmegen, The Netherlands, June 1990.

[LH96] H-W. Loidl and K. Hammond. Making a Packet: Cost-Effective Communication for a Parallel Graph Reducer. In *IFL'96 — International Workshop on the Implementation of Functional Languages*, volume 1268 of *LNCS*, pages 184–199, Bad Godesberg, Germany, September 1996. Springer-Verlag.

[LKI89] R. Loogen, H. Kuchen, and K. Indermark. Distributed Implementation of Programmed Graph Reduction. In *PARLE 89 — Conference on Parallel Architectures and Languages Europe*, volume 365 of *LNCS*, pages 136–157. Springer-Verlag, 1989.

[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. In *IFL'97 — International Workshop on the Implementation of Functional Languages*, volume 1467 of *LNCS*, pages 199–214, University of St. Andrews, Scotland, September 10–12, 1997.

[Loi97] H-W. Loidl. LinSolv: a Case Study in Strategic Parallelism. In *Glasgow Workshop on Functional Programming*, Ullapool, Scotland, September 15–17, 1997.

[LTH+99] 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*, 11:701–752, 1999.

[Rei98] A. Reid. Putting the Spine back in the Spineless Tagless G-machine: an implementation of revertible blackholes. In *IFL'98 — International Workshop on the Implementation of Functional Languages*, September 9–11, University College London, UK, 1998. Draft proceedings.

[Rus95] D. Rushall. *Task Exposure in the Parallel Implementation of Functional Programming Languages*. PhD thesis, Department of Computer Science, University of Manchester, 1995.

[THLP98] P.W. Trinder, K. Hammond, H-W. Loidl, and S.L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, January 1998.

[THM+96] P. Trinder, K. Hammond, J.S. Mattson Jr., A.S Partridge, and S.L. Peyton Jones. GUM: a Portable Parallel Implementation of Haskell. In *PLDI'96 — Programming Languages Design and Implementation*, pages 79–88, Philadelphia, PA, USA, May 1996.