The Efficiency of Parallel Graph Reduction on a Loosely-coupled Multiprocessor

Hans-Wolfgang Loidl

Department of Computing and Electrical Engineering,
Heriot-Watt University,
Edinburgh EH14 4AS, Scotland
Email: `hwloidl@cee.hw.ac.uk`

Abstract

Systems for parallel graph reduction, such as the GUM abstract machine, usually rely on dynamic resource management in order to automatically coordinate the parallel execution. However, in general the choice of the optimal resource management policy depends on the characteristics of the parallel machine, such as its latency.
In this paper we study the behaviour of GUM on a Beowulf multi-processor and focus on the dynamic distribution of work in the runtime-system. We observe how the high latency of this machine influences the resource management. Finally, we present modifications to the thread management and load balancing subsystems and discuss preliminary runtime measurements for some of these modifications.

1 INTRODUCTION

The implementation of GPH, our parallel extension of Haskell, is based on a model of parallel graph reduction with the programmer exposing parallelism in the source code and the runtime-system automatically managing the parallelism [THM+96]. This defers many decisions about the coordination of parallelism to the runtime-system. In general, these decisions depend on the underlying parallel machine.

In this paper we discuss possibilities of improving the dynamic resource management of GUM for high-latency machines. In particular we focus on the aspect of work distribution on a loosely-coupled multi-processor. The goal is to achieve a roughly even load distribution throughout the computation, without generating an excessive amount of communication. To achieve this goal we refine GUM's management of potential parallelism, enabling different levels of aggressiveness in its load distribution. We present preliminary measurements for some of these new runtime-system features, which demonstrate the potential performance improvements of these techniques.

The structure of the paper is as follows. Section 2 gives a general overview of the GUM runtime-system, focusing on thread management and load balancing. Section 3 presents measurements for a `raytracer` on a Beowulf multi-processor. Section 4 relates GUM to other systems for parallel functional programming. Finally, Section 5 concludes.

2 GUM: AN ABSTRACT MACHINE FOR PARALLEL GRAPH REDUCTION

In this section we give an overview of the design of GUM with special emphasis on thread management and load balancing. We motivate the design decisions made for GUM and discuss possible improvements to the basic runtime-system. For a more detailed discussion of GUM the interested reader is referred to [HMP$^+$95].

The key concepts in the design of GUM (Graph Reduction on a Unified Memory Model) are an underlying distributed-memory model, a virtual shared heap implemented on top of that model and automatic, dynamic resource management for both work and data. Based on this design we can identify several components with interrelated tasks:

- The *thread management model* is responsible for deciding when to generate a new thread and how to schedule the threads.
- The *load balancing model* is responsible for distributing the load in the parallel system so that idle time of PEs is minimised.
- The *memory management model* is responsible for controlling access to remote data.
- The *communication model* is responsible for transferring data and work between processing elements (PEs).

2.1 The Thread Management Model

The "evaluate-and-die" thread management model we use in GUM has been developed for the GRIP parallel graph reduction machine [PCSH87]. This model uses *sparks*, pointers to graph structures, in order to represent potential parallelism. Sparks are generated via an explicit `par` annotation in the program and maintained by the runtime-system in a flat *spark pool*. A "sparked" expression may be executed by an independent thread. However, if another thread needs the value of the expression, and no other thread is evaluating it, this thread will perform the computation itself. We call this behaviour *thread subsumption* because the potentially parallel work is inlined by another thread, typically the parent thread. This idea of dynamically increasing the granularity of the threads by deferring the decision whether to generate a thread is similar to the independently developed lazy task creation model [MKH91].

The synchronisation between threads is implicit via accessing shared "closures", nodes in the graph structure. We distinguish between normal-form closures, which represent data, and thunks, which represent work (unevaluated data). If a thread needs the result of a graph structure that is currently being evaluated or that resides on another PE it is blocked on this structure. As soon as the result becomes available the thread is awoken and can continue.

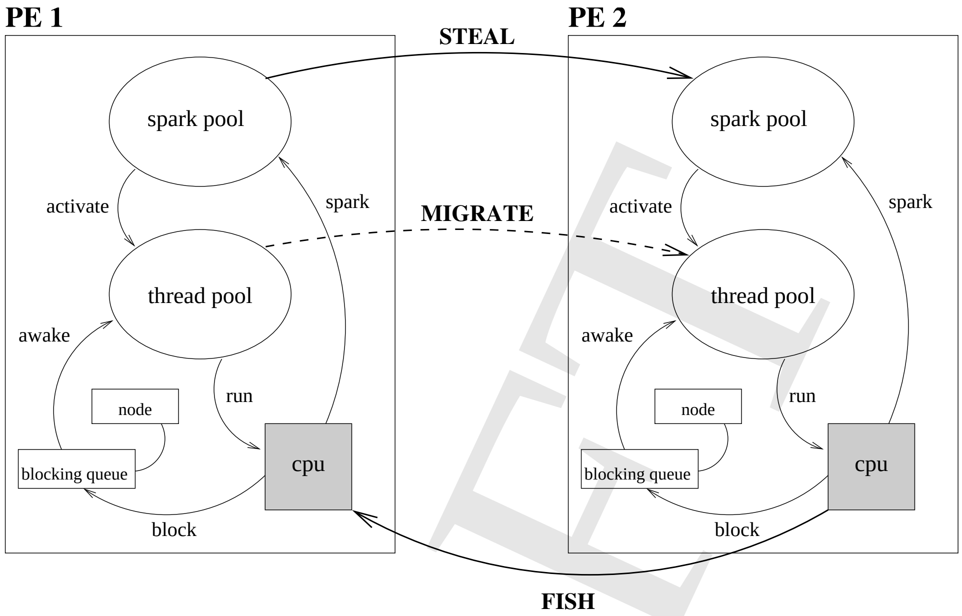

FIGURE 1. Interaction of the components of a GUM PE.

One important part of the thread management model is the *scheduler*. The scheduler decides which of the runnable threads, that are maintained in a flat *thread pool*, to execute. In GUM we use an unfair scheduler, which continues to execute the current thread until it is blocked, trying to access data that is locally under evaluation or remote. This policy helps to bound the total resource consumption of the system.

2.2 The Load Balancing Model

The load balancing mechanism in the original GUM design was deliberately simple. The main idea was to distribute the load by moving sparks to idle processors. The representation of sparks should be cheap, to minimise the overhead of creating parallelism and to reduce the costs for sending sparks between processors. Once a spark has been turned into a thread, or been "activated", the thread will remain on this PE. The main challenge for the load balancing model is to efficiently and effectively distribute the available sparks to ensure an even load balance, without imposing additional high load onto the system.

Figure 1 depicts the logical components that exist on each PE of the GUM abstract machine. Two pools are used for managing sparks and threads on the PE. Sparks are generated via executing the `par` primitive on the CPU. In the current version of GUM adding a spark to the spark pool amounts to just adding a pointer to an array of pointers to heap structures and thus meets the criterion of being a cheap operation. Since GPH programs often create a massive amount of parallelism, low-cost spark pool management is important to achieve high performance. This criterion is less important for the thread pool, where it might be advantageous to including additional information, e.g. the priority of a live thread, in order to achieve more flexible scheduling. In the current version of GUM both spark and thread pools are managed as FIFO queues. When the CPU is idle, and the thread pool is empty, a spark will be activated by generating a thread state object (TSO), which holds essential information of this thread such as registers and a pointer to its stack. The scheduler then determines how to choose one of the threads from the thread pool to run on the CPU. If a running thread blocks on unavailable data, it is added to the blocking queue of that node. It will be awoken again when the data becomes available, either because a local thread produces that data or the data arrives from another processor.

The arrows between the PEs in Figure 1 show the messages, related to the load balancing model, that are exchanged in GUM. Initially all processors, except the main PE, will be idle, with no local sparks available. It will therefore send a `FISH` message to a randomly chosen PE. On arrival of this message, PE1 will search for a spark and, if available, send it to PE2. This mechanism is usually called "work stealing" or passive load distribution, since an idle processor has to ask for work. If no spark is available on PE1 it will search for a runnable thread and, if available, migrate that thread to another processor. This is a very expensive operation since it includes the transfer of the entire TSO and of the stack attached to this thread. Thread migration is not fully functional in the current version of GUM.

2.3 The Memory Management Model

To achieve high performance on different classes of parallel machines the following design decisions have been made for GUM. GUM uses a *flat memory hierarchy*, i.e. the access to any closure in one PE's heap is uniform. Every globally visible closure in the heap is identified via a global addresses (GAs), i.e. globally unique identifiers. The mapping of these GAs to local heap addresses and vice versa is done via a hash table, the global indirection table (GIT). Global indirections (FetchMes) always use the GA to identify the remote object (see Figure 2) and weighted reference counting [Bev87] is used on GAs to reclaim unused remote references. This design enables separate local garbage collection on each of the PEs, provided that the GIT is rebuilt after every garbage collection to map those GAs, which are still alive, to their new addresses in the heap. Rebuilding the GIT is therefore one major source of overhead in GUM. This design is based on the assumption, verified on GRIP, that only a small fraction of the heap is globally visible and that rebuilding the GIT is therefore cheaper than allocating and maintaining one additional word, the GA, for every closure in the system.

The mapping of global to local addresses is only needed for determining whether a copy of a newly imported graph structure already exists on that PE. In this case, the less evaluated version of the graph will become an indirection to the further evaluated version. This avoids duplicating data that might have been imported via

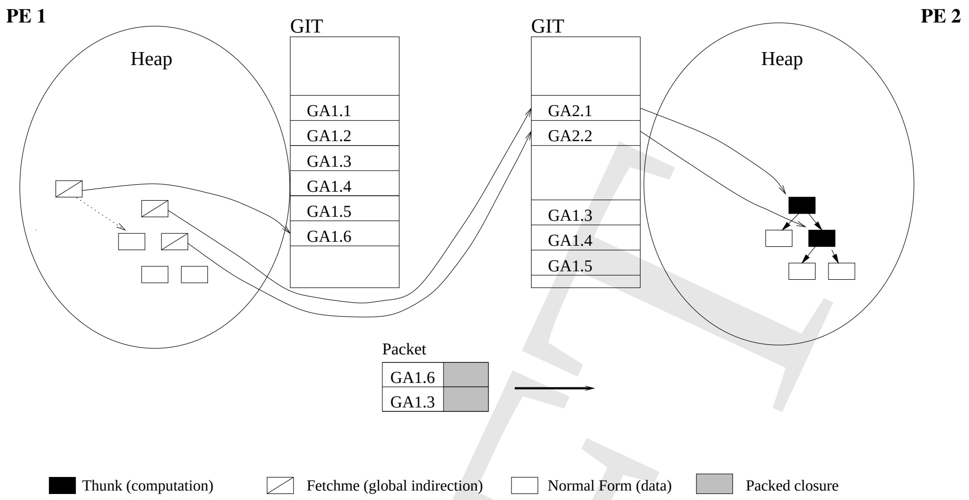

FIGURE 2. Transfer of graph structures in GUM.

different PEs. The decision of "globalising" both normal forms and thunks, i.e. generating GAs for both, reduces the total memory consumption in the system. However, it increases packing costs and the size of the packet which is sent.

2.4 The Communication Management Model

To reduce the total amount of communication in GUM and to permit latency hiding, i.e. overlapping communication with computation, GUM uses asynchronous, bulk communication, as shown in Figure 2. The *packing scheme* determines how much of the graph structure to put into a communication packet. By default GUM uses full subgraph packing, limited by the fixed packet size, i.e. when a graph is larger than the packet size only the initial portion of the graph is sent.

The packing algorithm traverses the graph in a breadth-first fashion. For each closure a new GA is allocated, if it does not already have one. When packing the closure itself, thunks and normal-forms have to be treated differently. In order to avoid the duplication of work, thunks are never copied but moved between processors. Therefore, the original thunk is replaced with a "revertible black hole". Should other threads demand the value before the transfer is complete they will be blocked. Normal-forms on the other hand can be freely copied.

On the receiver side, the graph is unpacked, checking for the presence of other copies of the imported closures in order to maintain sharing. For thunks new GAs are allocated, which determine the location of the closure. After unpacking the whole graph a mapping of old to new GAs is transmitted to the sender, which then replaces all revertible black holes with global indirections to the new GAs, and the old GAs become garbage.

Figure 2 summarises the transfer of graph structures in GUM. It shows a snap-

```
ray :: Int -> Int -> Int -> [Sphere] -> [[((Int, Int), Vector)]]
ray chunk x y world
  = map do_line sizes_y
    `using` parListChunk chunk rnf
    where do_line :: Int -> [((Int,Int), Vector)]
          do_line i = map (\ j -> ((i,j), f i j)) sizes_x
          sizes_x = [0..x-1]
          sizes_y = [0..y-1]
          f i j = tracepixel world lights i j firstray scrnx scrny
          (firstray, scrnx, scrny) = camparams x y
```

FIGURE 3. GPH **code of the parallel `raytracer`.**

shot in the transfer of a five closure graph with root `GA2.1` (originally `GA1.1`) from PE1 to PE2. The transfer of the graph with root `GA1.6` is still in progress. Note that thunks have been moved from PE1 to PE2 and replaced by global indirections on PE1. Normal-form closures have been copied. Maintaining their GAs in the GIT will ensure that the sharing of closure `GA1.3` is maintained on PE2.

3 ASSESSING THE WORK DISTRIBUTION IN GUM

In this section we use a simple `raytracer` program to assess the effectiveness of GUM's mechanisms for controlling work distribution. The `raytracer` calculates a 2D image of a given scene of 3D objects by tracing all rays in a given grid, or window. The code for the top level function is shown in Figure 3. It performs two nested maps, the outer one over all lines of the window, and the inner one over all pixels on a line. Our parallelisation uses a `parListChunk` evaluation strategy over the outer map. This strategy causes list segments of the size `chunk` to be evaluated to normal form. By choosing an appropriate value for `chunk` the granularity of the parallelism can be tuned. A more detailed discussion of this algorithm and its parallelisation is given in [LKH$^+$00].

All measurements presented in this paper have been performed on a 32-node Beowulf cluster [RBMS97] at Heriot-Watt University, consisting of Linux RedHat 6.2 workstations with a 533MHz Celeron processor, 128kB cache, 128MB of DRAM and 5.7GB of IDE disk. The workstations are connected through a 100Mb/s fast Ethernet switch with a latency of 142μs, measured under PVM 3.4.2.

3.1 Default Load Distribution

By default GUM's load distribution is passive, relying on requests for work from idle PEs, as discussed in Section 2.2. On a Beowulf cluster the high latency might cause significantly different times for the startup and the initial work request of the individual PEs. In practice we have observed differences of up to half a second. In some cases this may result in a few processors monopolising the parallelism by fetching all available sparks. If additionally the computation represented by these sparks immediately requires data left on the original processor all of these

FIGURE 4. Per-PE activity profile for the `raytracer` with default load balancing.

sparks will almost immediately be turned into blocked threads, with the processor asking for new work each time the only active threads gets blocked. In the case of the `raytracer`, where all parallelism is generated at the beginning of the computation, this does indeed monopolise all available parallelism.

Figure 4 shows a per-PE activity profile for a `raytracer` execution with the default load distribution mechanism. A per-PE activity profile shows the behaviour for each of the PEs (y-axis) during the execution (x-axis) and thus serves well for studying issues related to load distribution. Each PE is visualised as a horizontal line, with darker shades of gray (green in a colour profile) indicating a higher load of the system. Usually load is measured as the number of runnable threads on the PE. Variants of this profile define the load to be the number of sparks, and thus allow to focus on the distribution of sparks, rather than threads, in the system. Gaps in the horizontal lines (red areas in the colour profile) indicate idleness.

From Figure 4 it is obvious that the default load distribution is very poor. Only 5 out of 16 processors participate in the parallel computation. While two PEs, PEs 16 and 15, are heavily loaded, the others remain idle, not succeeding in their search for work. In the current version of GUM no load information on remote PEs is maintained. Messages for stealing work are sent to random PEs. We have modified this mechanism to always send the first message to the main PE, which initially holds the root of the graph and is most likely to produce a lot of parallelism. This usually shortens the startup of the PEs but does not help in this case.

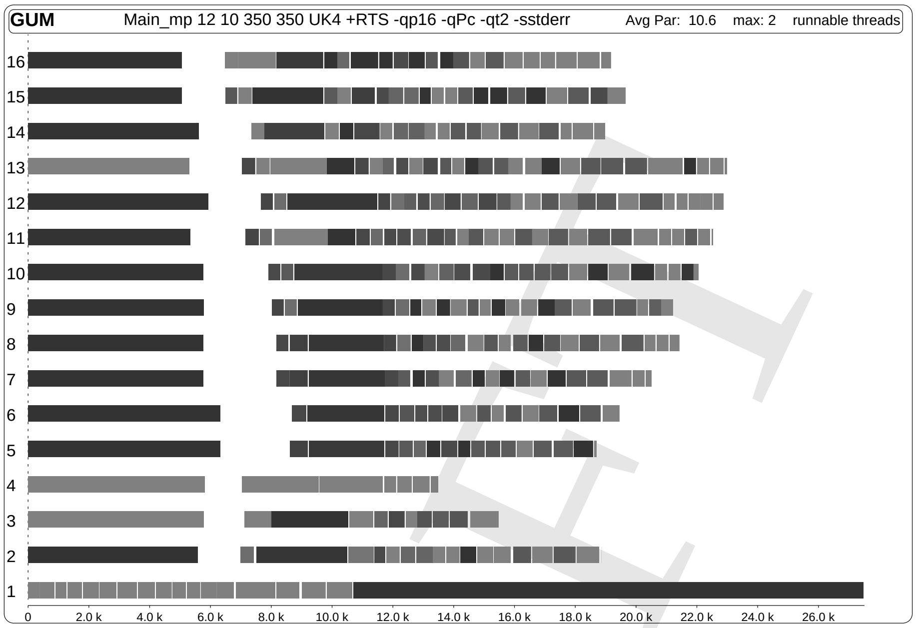

FIGURE 5. Per-PE activity profile for the `raytracer` with a live thread limit of 2.

3.2 Controlling the Number of Threads

One simple way of controlling the load distribution is to specify a hard limit on the total number of live threads, i.e. runnable or blocked threads. This avoids activating a huge number of sparks in the rather common case, where a newly activated thread needs remote data early on in its computation, therefore blocking quickly. However, it bears the danger of losing opportunities for latency hiding if the PE reaches this limit while still having sparks that represent useful work.

Figure 5 shows a per-PE activity profile when using a limit of 2 for the number of live threads. In contrast to the previous figure, all PEs participate in the computation, and the load is fairly well balanced. However, at about one third of the computation we observe a fairly large gap on all of the PEs except the main PE. At this point the active threads all require data initially held on the main PE. The hard limit on the number of live threads causes the PEs to become idle at this point. Therefore, this mechanism is a very crude way of improving load distribution. It is, however, well suited for this example, where all parallelism is generated at the beginning of the execution, and the individual threads are roughly the same size.

3.3 Low- and High-Watermarks for the Spark Pools

In order to achieve a more flexible mechanism for controlling and distributing work, we refine the management of the spark pool without using hard upper limits.

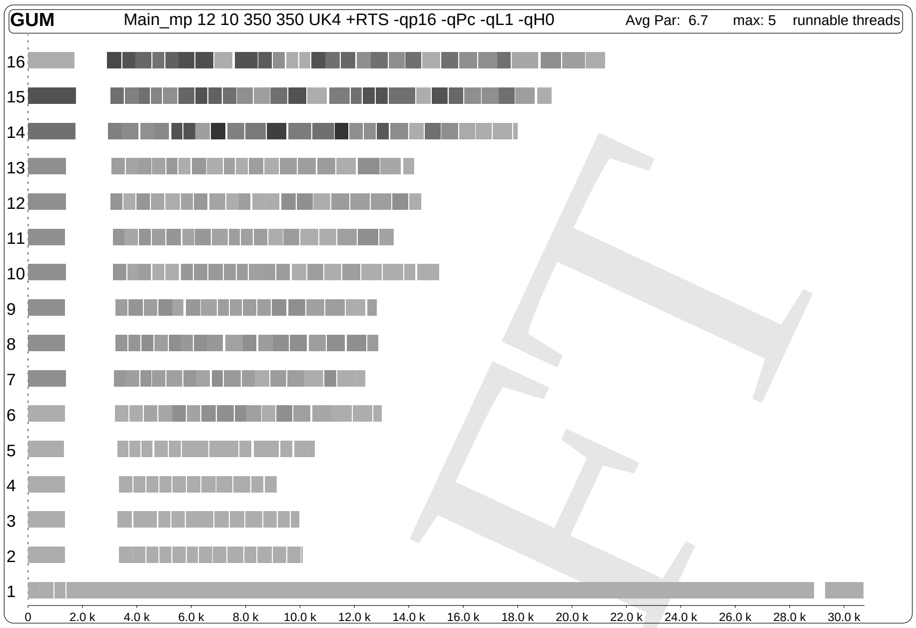

FIGURE 6. Per-PE activity profile for the `raytracer` with a low-watermark of 1.

This approach matches GUM's original design philosophy of distributing load in the form of freely floating sparks as long as possible. It also builds on the experience of other parallel runtime-systems with dynamic resource management that often use some form of load bounding [OD92, Sar87] or a finite pool of concessions for parallelism [BHK+94] in order to avoid excess parallelism without discarding useful work.

To provide better control of the distribution of sparks we have implemented low- and high-watermarks for the spark pool. The *low-watermark* indicates how many sparks should always be kept local on a PE. This avoids exporting all potential work at a point where a processor is busy. Since we cannot predict for how long the current thread will run, it might be preferable to retain some potential parallelism to avoid communication once the PE becomes idle. The *high-watermark* indicates the maximum number of sparks that should be held in a spark pool. In contrast to the live thread limit discussed above, this is a soft limit. If the spark pool size exceeds this limit, the PE will start to off-load sparks to other processors without being asked for work. In other words, the processor will temporarily switch from passive to active load distribution, until the spark pool size drops below the high-watermark again. This mechanism increases the system's aggressiveness in terms of load distribution in the case where one PE generates a large amount of parallelism.

Figure 6 shows the per-PE activity profile of the `raytracer` when using a low-watermark of 1 on the spark pool. This setting ensures that at each point the main PE will retain at least one spark, and that the parallelism is not entirely monopolised by some PEs. Already such a low value suffices to have sufficient sparks available when the work requests arrive from the other PEs. However, most of the parallelism is still allocated to the fastest PEs. In some sense the low-watermark adds a certain amount of resistance to the work distribution in GUM. The complementary technique of switching to active load distribution via specifying a high-watermark has been implemented, but no results are available, yet.

3.4 Thread Migration

Another mechanism for potentially improving the load balance is thread migration. The idea of this mechanism is to transfer an entire thread from a highly loaded to an idle processor, if no sparks are available. Since this is a rather expensive operation it should only be performed when no other work is available in the system. However, for coarse grained parallel programs, this mechanism might be advantageous in some cases. For example our measurements on the parallel version of the Lolita natural language engineering system in [LMT+97] show poor load distribution in the later stages of the computation when all sparks have been activated already. Thread migration is partially implemented but not fully functional in the current version of GUM.

4 OTHER SYSTEMS FOR PARALLEL FUNCTIONAL PROGRAMMING

Most closely related to GUM are two systems for parallel functional programming: Eden and PMLS. In this section we relate GUM to these two systems. A more thorough comparison, including detailed performance studies, is given in [LRS+01], which is currently in preparation.

Eden [BLOP97] is another parallel extension of Haskell, designed for both distributed and parallel execution. The new constructs in Eden are process-abstraction and process instantiation. The former is analogous to lambda-abstractions but defines a process with a specific input-output behaviour. The latter generates a process defined by a process abstraction.

In terms of implementation, the abstract machine for Eden, DREAM [BKL+97], is closely related to GUM. It employs slightly less dynamic resource management by using mandatory threads exclusively, i.e. all parallelism denoted in the program is guaranteed to be created. Furthermore, rather than using lazy communication as in GpH, Eden implements strict communication. Due to these two design decisions Eden does not require a separate spark pool, or a virtual shared heap. The overhead for managing parallelism is consequently reduced.

PMLS [MSBK01] is a parallelising compiler for SML that realises parallelism in higher-order functions as algorithmic skeletons. In contrast to GpH and Eden it is not necessary to expose potential parallelism in the program. Instead the compiler automatically executes instances of certain common higher-order functions, such as map and fold, in parallel. PMLS therefore represents a system of implicit parallelism.

PMLS makes decisions about how to parallelise a program at compile time already. In particular, the distribution of work and data is fixed in the implementation of the parallel skeletons. This reduces the runtime overhead for parallel execution but also generates a system that is less flexible in general.

5 CONCLUSIONS

The dynamic resource management in GUM requires sophisticated runtime-system mechanisms to achieve a good work distribution on a high-latency multi-processor. In this paper we have reviewed the GUM design and presented several extensions to the original design aiming at improved work distribution. Some of these extensions, such as low- and high-watermarks on the spark pools, are already implemented and the preliminary measurements in this paper show encouraging performance improvements. For one example program that is particularly sensitive to load distribution, a `raytracer`, we can improve the average parallelism from 2.9 to 6.7 by using low-watermarks on the spark pool and to 10.6 by using a hard limit on the number of live threads.

As future work we plan to measure the effectiveness of the watermark mechanism on a larger set of test programs. Furthermore, we plan to test thread migration on some coarse-grained applications, which are notoriously susceptible to bad load balance. In the longer term we want to develop a self-adjusting runtime-system that is capable of tuning parameters such as low- and high-watermarks automatically based on characteristics of the underlying parallel machine.

REFERENCES

[Bev87] D.I. Bevan. Distributed Garbage Collection Using Reference Counting. In *PARLE'87 — Parallel Architectures and Languages Europe*, volume 259 of *LNCS*, pages 176–187, Eindhoven, The Netherlands, 1987. Springer-Verlag.

[BHK+94] T. Bülck, A. Held, W. Kluge, S. Pantke, C. Rathsack, S-B. Scholz, and R. Schröder. Experience with the Implementation of a Concurrent Graph Reduction System on an nCUBE/2 Platform. In *CONPAR'94 — Conference on Parallel and Vector Processing*, volume 854 of *LNCS*. Springer-Verlag, 1994.

[BKL+97] S. Breitinger, U. Klusik, R. Loogen, Y. Ortega Mallén, and R. Peña Marí. DREAM: the Distributed Eden Abstract Machine. In *IFL'97 — International Workshop on the Implementation of Functional Languages*, volume 1467 of *LNCS*, pages 250–269. Springer-Verlag, St Andrews, Scotland, 1997.

[BLOP97] S. Breitinger, R. Loogen, Y. Ortega Mallén, and R. Peña Marí. Eden — The Paradise of Functional Concurrent Programming. In *Glasgow Workshop on Functional Programming*, 1997.

[HMP+95] K. Hammond, J.S. Mattson Jr., A.S Partridge, S.L. Peyton Jones, and P.W. Trinder. GUM: a Portable Parallel Implementation of Haskell. In *IFL'95 —*

International Workshop on the Parallel Implementation of Functional Languages, Bastad, Sweden, 1995.

[LKH+00] H-W. Loidl, U. Klusik, K. Hammond, R. Loogen, and P.W. Trinder. GpH and Eden: Comparing Two Parallel Functional Languages on a Beowulf Cluster. In *Trends in Functional Programming*, volume 2, pages 39–52, St Andrews, Scotland, July 2000. Intellect. Selected papers from SFP'00 — Scottish Functional Programming Workshop.

[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, St Andrews, Scotland, 1997.

[LRS+01] H-W. Loidl, F. Rubio, N.R. Scaife, K. Hammond, U. Klusik, R. Loogen, G.J. Michaelson, S. Horiguchi, R. Peña, S.M. Priebe, Á.J. Rebón, and P.W. Trinder. Comparing Parallel Functional Languages: Programming and Performance. *Higher-order and Symbolic Computation*, 2001. In preparation.

[MKH91] 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, July 1991.

[MSBK01] G. Michaelson, N. Scaife, P. Bristow, and P. King. Nested Algorithmic Skeletons from Higher Order Functions. *Parallel Algorithms and Applications*, 16:181–206, 2001.

[OD92] G. Ostheimer and A.J.T Davie. Load Bounding for Implicit Parallelism. In *IFL'92 — International Workshop on the Parallel Implementation of Functional Languages*, RWTH Aachen, Germany, 1992.

[PCSH87] S.L. Peyton Jones, C. Clack, J. Salkild, and M. Hardie. GRIP — a High-Performance Architecture for Parallel Graph Reduction. In *FPCA'87 — Conference on Functional Programming Languages and Computer Architecture*, volume 274 of *LNCS*, pages 98–112. Springer-Verlag, 1987.

[RBMS97] D. Ridge, D. Becker, P. Merkey, and T. Sterling. Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs. In *IEEE Aerospace*, 1997.

[Sar87] J. Sargeant. Load Balancing, Locality and Parallelism Control in Fine-Grain Parallel Machines. Internal Report UMCS-86-11-5, Department of Computer Science, University of Manchester, 1987.

[THM+96] 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 *PLDI'96 — Programming Languages Design and Implementation*, pages 79–88, Philadelphia, PA, USA, May 1996.