Chapter 1

Load Balancing in a Parallel Graph Reducer

Hans-Wolfgang Loidl[1]

Abstract: Parallel graph reducers such as GUM use dynamic techniques to manage resources during execution. One important aspect of the dynamic behaviour is the distribution of work. The load balancing mechanism, which controls this aspect, should be flexible, to adjust the distribution of work to hardware characteristics as well as dynamic program characteristics, and scalable, to achieve high utilisation of all processors even on massively parallel machines.

In this paper we study the behaviour of GUM's load balancing mechanism on a high-latency Beowulf multi-processor. We present modifications to the basic load balancing mechanism and discuss runtime measurements, which indicate that these modifications can significantly enhance the scalability of the system.

1.1 INTRODUCTION

The implementation of GPH, a parallel extension of Haskell [PH99], 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 and distributing resources [Pey89]. This approach defers many decisions about the coordination of parallelism to the runtime-system. Since these decisions depend on the underlying parallel machine, the runtime-system has to be flexible enough to adjust its behaviour to the characteristics of the parallel machine.

In this paper we discuss possibilities of improving the basic load balancing, or load management, mechanism in GUM on high-latency machines. The goal is to minimise idle time of processors during the computation, without generating an excessive amount of communication. To achieve this goal the load balancing

[1]Department of Computing and Electrical Engineering, Heriot-Watt University, Edinburgh EH14 4AS, Scotland. Email: `hwloidl@cee.hw.ac.uk`. This work was funded by the Austrian Academy of Sciences (APART 624).

mechanism should ensure that, given sufficient overall parallelism, each processor can create new work whenever it becomes idle. We refine GUM's distribution of potential parallelism, enabling different levels of aggressiveness. Similar mechanisms have been devised for older parallel graph reduction machines, such as GRIP [PCSH87] and the HDG machine [KLB91] (see Section 1.4). However, none of these mechanisms have been tested with large parallel programs on modern parallel machines, using state-of-the-art optimising compilation technology. We present measurements for some of these load balancing mechanisms in this context, and demonstrate the potential performance improvements and scalability of these techniques.

The structure of the paper is as follows. Section 1.2 gives a general overview of the GUM runtime-system, focusing on thread management and load balancing. Section 1.3 presents measurements for two non-trivial benchmark programs on a Beowulf multi-processor. Section 1.4 relates GUM to other systems for parallel functional programming. Finally, Section 1.5 concludes.

1.2 GUM: A PARALLEL GRAPH REDUCER

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 reader is referred to [THM+96].

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 processing elements (PEs) is minimised.
- The *memory management model* is responsible for controlling access to remote data and in GUM it implements a virtual shared heap.
- The *communication model* is responsible for transferring data and work between PEs (see [LH96] for a detailed discussion).

1.2.1 The Thread Management Model

GUM uses an *evaluate-and-die* thread management model originally developed for the GRIP distributed-memory graph reducer [PCSH87]. This model uses *sparks*, pointers to graph structures, to represent potential parallelism. Sparks are generated via an explicit `par` construct 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 a thread needs the value of the expression, and no other thread is evaluating it, the demanding thread will perform the computation itself. We call this behaviour *thread subsumption* because the potentially parallel work is inlined by another 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 program 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.

1.2.2 The Load Balancing Model

The basic philosophy in the design of GUM's load balancing mechanism is to allow, and indeed encourage, a high amount of potential parallelism and to distribute potential work in the form of sparks. Therefore, 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.

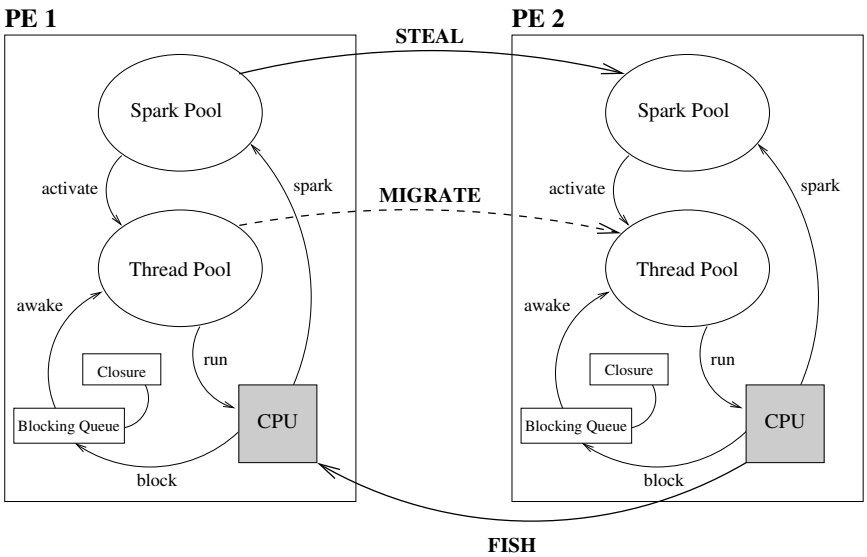

FIGURE 1.1. Interaction of the components of a GUM PE.

Figure 1.1 depicts the logical components that exist on each PE of the GUM abstract machine. Two pools are used for managing sparks and threads. Sparks are generated via executing the `par` primitive on a CPU. Adding a spark to the spark pool amounts to 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, 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 thick arrows between the PEs in Figure 1.1 show the messages, related to the load balancing model, that are exchanged in GUM. Initially all processors, except for the main PE, will be idle, with no local sparks available. PE2 sends 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 implemented in the current version of GUM and will not be discussed further in this paper.

1.3 ASSESSING THE LOAD DISTRIBUTION IN GUM

In this section we use two symbolic algorithms to assess the effectiveness of GUM's mechanisms for controlling load distribution: a simple `raytracer`, and an exact linear system solver (`linSolv`). The `raytracer` calculates a 2D im-

```
ray :: Int -> Int -> Int -> [Sphere] -> [Light] -> [[((Int, Int), Vector)]]
ray chunk x y world lights
  = 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 1.2. Top level code of the parallel `raytracer`.

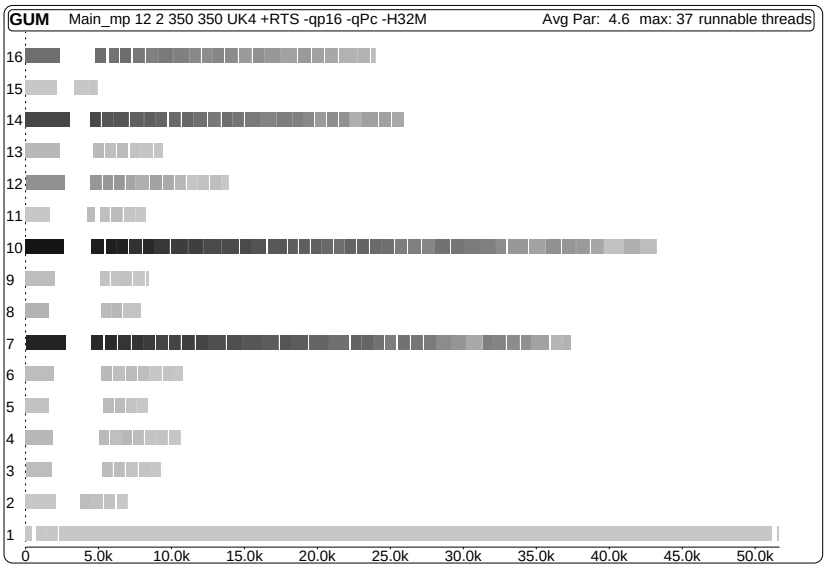

FIGURE 1.3. Per-PE activity profile for `raytracer` with default load balancing.

age of a given scene of 3D objects from a given light source by tracing all rays in a given grid, or window. The code for the top level function, shown in Figure 1.2, 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 [THLP98] over the outer map, achieving a parallel evaluation of subsequences of size `chunk`, where each of the subsequences is evaluated to normal form sequentially. The linear system solver finds an exact solution to a system of linear equations, represented as a matrix and a vector of arbitrary precision integer values. A more detailed discussion of both algorithms and their parallelisations is given in [LRS+01].

All measurements presented in this paper have been performed on a 32-node Beowulf cluster 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.

1.3.1 Default Load Distribution

Figure 1.3 shows a per-PE activity profile for a `raytracer` execution with the default load distribution mechanism in GUM. 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 larger number of runnable threads. Gaps in the horizontal lines (red areas in the colour profile) indicate idleness.

The poor load balance in Figure 1.3 is due to a combination of machine and program characteristics. On a Beowulf machine the high latency causes significantly different startup times and delays in the initial work request of the individual PEs. In practice we have observed differences of up to half a second. The `raytracer` program generates all parallelism at the beginning of the computation. Each of the parallel threads needs large input data, its portion of the window, early on in the computation. Therefore the thread will block almost immediately, fetching remote data, and the now idle PE will ask for additional work. In the execution shown in Figure 1.3 the 4 PEs with the fastest startup, PEs 7, 10, 14, and 16, obtain almost all available parallelism, yielding a very high load especially at the beginning, reaching up to 37 runnable threads. The other PEs remain idle most of the time, and are unable to find additional parallelism once the sparks have been activated on a PE. In essence, the fastest PEs monopolise the available parallelism by fetching all available sparks.

In this case thread migration would help to improve performance by transferring runnable threads from highly loaded to idle PEs. We consider thread migration to be an important, though expensive, last resort of balancing the load.

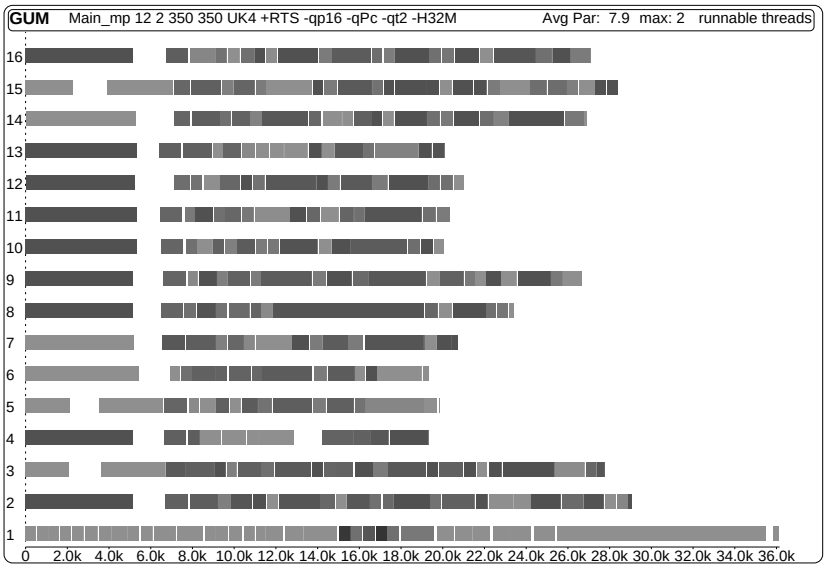

FIGURE 1.4. Per-PE activity profile for `raytracer` with a live thread limit of 2.

1.3.2 Controlling the Number of Live 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 potentially increases idleness if the PE reaches this limit while still having sparks that represent useful work. In essence, opportunities for latency hiding are lost in this case.

Figure 1.4 shows a per-PE activity profile when using a limit of 2 for the number of live threads. This very small limit is chosen as an extreme case to study the difference in the resulting load distribution. In contrast to the previous figure, each PE is utilised for at least about half of the computation, and the load is fairly well balanced. However, at about one fourth of the computation we observe a fairly large gap on almost all of the PEs except the main PE. At this point the active threads all require data initially held on the main PE without having sparks available to overlap the necessary communication with useful computation. In summary, 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.

1.3.3 Low- and High-Watermarks for the Spark Pools

To achieve a more flexible mechanism for controlling and distributing work, we refine the management of the spark pool without using hard upper limits. We build on the experience with similar threshold mechanisms in GRIP [PCSH87], as well as watermark mechanisms used in ZAPP [BS81], the HDG machine [KLB91], and the Manchester Dataflow Machine [Sar87].

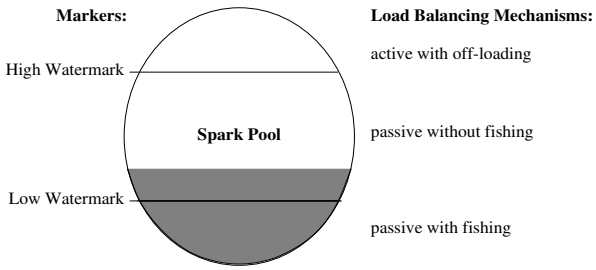

FIGURE 1.5. Low- and High-watermark mechanisms for load balancing in GUM.

To provide better control of the distribution of sparks we have implemented low- and high-watermarks for the spark pool. In this model the load balancing mechanism depends on the emptiness of the spark pool as sketched in Figure 1.5. The *low-watermark* indicates how many sparks should always be kept local on

a PE. If the number of sparks falls below this mark, no sparks will be exported and the PE will try to obtain new sparks from other PEs. In particular, 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. If the number of sparks 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. Excessive amounts of communication in the case of massive parallelism can be avoided by using a delay time between off-load messages, similarly to the already existing delay time between sending fish messages, which has proven to be very effective in bounding communication. However, at the moment we do not have any measurements available for the high-watermark mechanism to assess this potential danger. In the rest of the paper we focus on low-watermarks alone.

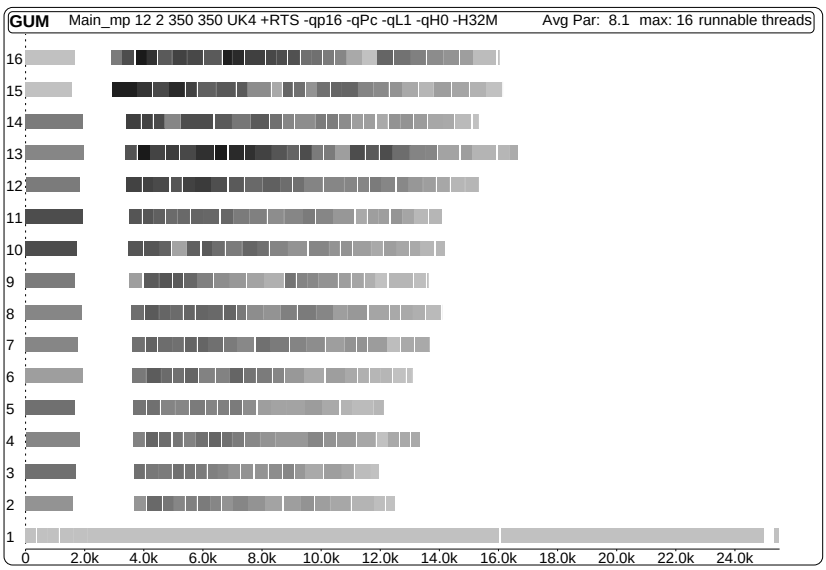

FIGURE 1.6. Per-PE activity profile for `raytracer` with a low-watermark of 1.

Figure 1.6 shows the per-PE activity profile of the `raytracer` when using a low-watermark of 1 on the spark pool. This value is chosen to guarantee 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 a sufficient number of sparks available when the work requests arrive from

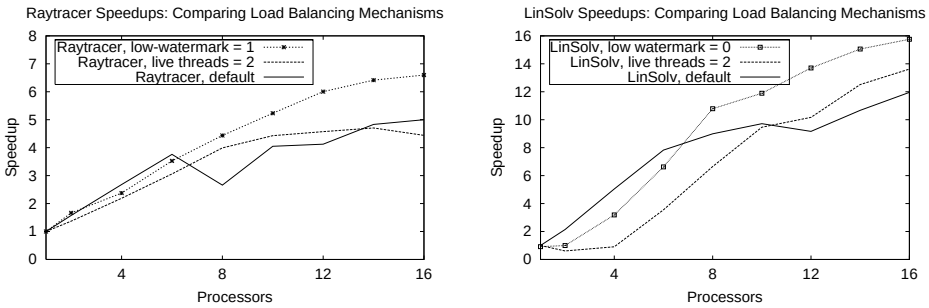

FIGURE 1.7. Speedups for `raytracer` and `linSolv`.

the other PEs. However, most of the parallelism is still allocated to the fastest PEs, and the maximum number of runnable threads is 16 in this case. Note that although this activity profile exhibits a poorer load balance compared to the previous profile, it has shorter gaps of idleness during the computation, resulting in a shorter total runtime of 25.5s compared to 36.1s. The average parallelism shows a much smaller difference. This suggests that the runtime-system overhead for the low-watermark mechanism is in fact smaller than that for the live thread limit.

1.3.4 Performance Comparison

The left hand side of Figure 1.7 compares the speedups achieved for the `raytracer` using the load balancing mechanisms discussed in the previous sections. For more than 6 PEs the best results are achieved for the low-watermark mechanism, with a speedup of 6.6 on 16 PEs. Most importantly, this mechanism shows better scalability than the other variants with increasing speedups up to 16 PEs. The smoothness of the speed-up curve, compared to the default load balancing mechanism, shows the superior flexibility of the low-watermark mechanism. Using a hard limit on the number of live threads initially achieves good speedups, however, due to the increase of idleness in this version it exhibits a poorer scalability, as latency hiding becomes more important for machines with more PEs. This behaviour is shown in a degrading performance after 14 PEs and a speedup of only 4.4 on 16 PEs. The default load balancing mechanism shows significant variations with varying numbers of processors. This is most likely due to the differences in latencies between the available machines in the cluster and suggests that this mechanism is not flexible enough to adjust the load balance to the characteristics of this parallel machine.

The right hand side of Figure 1.7 shows the speedups of an exact linear system solver, as described in [LRS+01], with the same load balancing mechanisms. This divide-and-conquer algorithm exhibits less regular parallelism, and generates its parallelism dynamically throughout the computation rather than only at the beginning. This program characteristic ensures a good load balance with the default

mechanism already. For large numbers of processors, where more parallelism is required to achieve a good work distribution, the new load balancing mechanisms are able improve the speedups of the program even further. For this program the best low watermark is the special case of 0, which means no sparks have to be retained on a processor. This differs from the default mechanism in that the latter retains the only spark on an idle PE. Therefore a low-watermark of 0 can be used to achieve a more aggressive distribution of sparks. Because of the dynamic creation of parallelism in this algorithm, there is little danger of a processor running out of work and therefore the total performance improvements are smaller.

1.4 RELATED WORK

In this section we relate GUM to other systems for parallel functional programming. A more thorough discussion, including detailed performance comparisons between GPH, Eden and PMLS versions of the `raytracer` and `linSolv` algorithms used here, is given in [LRS+01].

Most closely related to GPH is Eden [BLOP97], another parallel extension of Haskell, based on a similar parallel graph reducer. In contrast to GUM, it uses distributed heaps, avoiding the overhead of maintaining a virtual shared heap, and eager thread creation with active load distribution and no thread migration. No spark pool is necessary, but by committing potential parallelism to a process earlier than in GUM, the load balancing is less flexible. In practice, a good work distribution often has to be enforced in the program rather than by the system.

The π-RED$^+$ system [BHK+94] uses a finite pool of concessions to limit the amount of parallelism. This resembles our hard limit on the number of live threads. In contrast to GUM, however, π-RED$^+$ uses strict evaluation and eager task creation. Its concession mechanism has proven effective in a set of simple benchmark programs on a high-latency nCube multi-processor.

The load balancing mechanism in the HDG parallel graph reducer [KLB91], which was implemented on a high-latency Transputer network, is similar to our low-watermark mechanism in that it retains at least one inactive thread. This mechanism is based on a similar load distribution strategy used in the ZAPP system [BS81]. Furthermore, work requests are only sent to neighbouring PEs thereby trying to achieve enhanced data locality. Another technique for load bounding used in ZAPP is to switch between a FIFO and LIFO management of the task queue during work stealing, depending on the load of the processor. A similar mechanism, triggered by low- and high-watermarks on the length of the task queue, is used in the Manchester Dataflow Machine [Sar87].

The parallel ABC machine for Concurrent Clean, a Haskell-like lazy functional language, has been implemented on a Transputer network [Kes96]. To express parallelism, Concurrent Clean provides annotations similar to those in GPH. In contrast to GUM, it uses active load distribution and potentially absolute thread placement. In practice, skeletons [Col89], built on top of these annotations, with hard-wired, and thus less flexible, load balancing are used to achieve good performance. In this aspect it is similar to the skeleton-based, implicitly parallel

implementation of SML offered by the PMLS system [MSBK01].

The Alfalfa parallel graph reducer [Gol88], which implements the lazy functional language ALFL on a distributed memory machine, supports several load balancing mechanisms. In the performance measurements with these mechanisms a simple diffusion balancer, performing active load distribution with a low-watermark and sending work to neighbouring processors, performed best. Interestingly a load balancing mechanism which regularly exchanges load information between neighbouring processors failed to yield significant additional performance improvements.

1.5 CONCLUSIONS

The dynamic management of potential parallelism in the GUM parallel graph reducer requires sophisticated load balancing to achieve a good load distribution on high-latency multi-processors. In this paper we have presented several extensions to the original design of the load balancing mechanism in GUM together with measurements assessing the flexibility and the scalability of these mechanisms on a Beowulf cluster. For one example program that is particularly sensitive to load distribution, a `raytracer`, we can improve the relative speedup from 4.9 to 6.6 on 16 PEs by using low-watermarks on the spark pool. The low-watermark mechanism achieves the best results for both sets of measurements, although the optimal setting of its value depends on the characteristics of the parallel program.

Clearly it is not realistic to defer the decision about the exact value for the limits presented here to the application programmer. In the examples in Section 1.3 our choice has been motivated by the structure of the parallelism in the program. In particular, the rate and the locality of producing parallelism are important. Based on such information a pre-defined collection of settings, e.g. for programs generating all parallelism at the beginning, can be used to obtain better parallel performance. These tests with different values aim at determining good values for such bundles of runtime-system parameters.

As future work we plan to measure the effectiveness of the watermark mechanism on a larger set of test programs. We plan to complete the implementation of thread migration to make use of this additional feature in 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, and possibly making use of program analysis or profiling information about the structure of the parallelism in a program.

REFERENCES

[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 — Conf. on Parallel and Vector Processing*, LNCS 854. Springer-Verlag, 1994.

[BLOP97] S. Breitinger, R. Loogen, Y. Ortega Mallén, and R. Peña Marí. The Eden Coordination Model for Distributed Memory Systems. In *HIPS'97 — Workshop on High-level Parallel Programming Models*, pages 120–124, Geneva, Switzerland, April 1997. IEEE Computer Science Press.

[BS81] F.W. Burton and M.R. Sleep. Executing Functional Programs on a Virtual Tree of Processors. In *FPCA'81 — Conf. on Functional Programming Languages and Computer Architecture*, pages 187–194, Portsmouth, USA, 1981.

[Col89] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989.

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

[Kes96] M. Kesseler. *The Implementation of Functional Languages on Parallel Machines with Distributed Memory*. PhD thesis, Univ. of Nijmegen, 1996.

[KLB91] H. Kingdon, D.R. Lester, and G. Burn. The HDG-machine: a Highly Distributed Graph-Reducer for a Transputer Network. *Computer Journal*, 34(4):290–301, 1991.

[LH96] H-W. Loidl and K. Hammond. Making a Packet: Cost-Effective Communication for a Parallel Graph Reducer. In *IFL'96*, LNCS 1268, pages 184–199, Bad Godesberg, Germany, 1996. Springer-Verlag.

[LRS+01] H-W. Loidl, F. Rubio, N.R. Scaife, K. Hammond, S. Horiguchi, U. Klusik, R. Loogen, G.J. Michaelson, 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. Submitted.

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

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

[Pey89] S.L. Peyton Jones. Parallel Implementations of Functional Programming Languages. *Computer Journal*, 32(2):175–186, April 1989.

[PH99] S.L. Peyton Jones and J. Hughes (editors). Haskell 98: A Non-strict, Purely Functional Language, 1999. Available at `http://www.haskell.org/`.

[Sar87] J. Sargeant. Load Balancing, Locality and Parallelism Control in Fine-Grain Parallel Machines. Int. Rep. UMCS-86-11-5, Dept. of Computer Science, Univ. of Manchester, 1987.

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

[THM+96] P.W. Trinder, K. Hammond, J.S. Mattson, A.S Partridge, and S.L. Peyton Jones. GUM: a Portable Parallel Implementation of Haskell. In *PLDI'96*, pages 79–88, Philadelphia, USA, 1996. ACM Press.