The Virtual Shared Memory Performance of a Parallel Graph Reducer*

Hans-Wolfgang Loidl

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

Abstract

This paper assesses the costs of maintaining a virtual shared heap in our parallel graph reducer (GUM), which implements a parallel functional language. GUM performs automatic and dynamic resource management for both work and data. We introduce extensions to the original design of GUM, aiming at a more flexible memory management and communication mechanism to deal with high-latency systems. We then present measurements of running GUM on a Beowulf cluster, evaluating the overhead of dynamic distributed memory management and the effectiveness of the new memory management and communication mechanisms.

1. Introduction

Our parallel runtime system (GUM) implements a functional language and is based on parallel graph reduction [9]. In this model a program is represented as a graph structure and parallelism is exploited by reducing independent subgraphs in parallel. The most natural implementation of parallel graph reduction uses a shared heap for memory management. GUM implements a virtual shared heap on a distributed memory model, using PVM as generic communication library for transferring data. For efficient compilation we use a state-of-the-art, optimising compiler, namely the Glasgow Haskell Compiler [11].

In this paper we investigate the overhead incurred by this virtual shared heap model. We measure key parameters of the memory management subsystem to assess this overhead for programs with different communication characteristics. We present extensions to the memory management and communication subsystems to enhance the flexibility of the system and give detailed measurements for these extensions on a Beowulf cluster.

As programming language we use GPH, a parallel dialect of the functional language Haskell. Its only extension to Haskell is a primitive, `par`, which indicates a possible parallel execution for a program expression. All dynamic control of the parallelism is completely implicit. This programming model encourages the generation of massive amounts of fine-grained parallelism and puts even higher importance on the efficiency of its management in the runtime system.

* Supported by the Austrian Academy of Sciences (APART 624).

2. A Virtual Shared Heap in GUM

In this section we give an overview of the design of GUM with special emphasis on memory management. For a more detailed discussion of GUM see [13].

The key concepts in the design of GUM are an underlying distributed memory model, a virtual shared heap implemented on top of that model and automatic, dynamic resource control for both work and data. Based on this design we can identify several interrelated components:

- The *thread management subsystem* is responsible for generating new threads and scheduling them.
- The *memory management subsystem* is responsible for controlling access to local and remote data.
- The *communication subsystem* is responsible for transferring data and work between processing elements.

2.1. The Thread Management Subsystem

The *evaluate-and-die* thread management subsystem we use in GUM has been developed for the GRIP parallel graph reduction machine [10]. This model represents potential parallelism as *sparks*, i.e. pointers to structures in the distributed program graph. In the program code a `par` primitive is used to generate sparks, which are maintained by the runtime system in a flat, distributed 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, this thread will perform the computation itself. This behaviour is called *thread* *subsumption* because the potentially parallel work is inlined by another thread, typically its parent. 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 [8].

The synchronisation between threads is implicit via accessing shared *closures*, i.e. 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 computation, represented by a graph structure, that is currently active or that is executed on another processing element (PE), the thread is blocked on this graph structure. As soon as the result becomes available the thread is awoken and can continue. To realise this form of blocking, every thread has to lock a closure on entry, and a waiting list of all blocked threads has to be maintained. The locking of closures is one major source of sequential overhead.

The basic load balancing mechanism in GUM is one of *work stealing*, i.e. whenever a PE runs out of work it asks other PEs for work. If work is available it will be transferred as a spark from the busy to the idle PE. Based on measurements on a Beowulf cluster, we have recently refined the load balancing mechanism, so as to avoid the monopolisation of parallelism by PEs on a high-latency system [6].

2.2. The Memory Management Subsystem

As an implementation of a functional language GUM focuses on heap allocated data. It 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 address (GA), a globally unique identifier. Global indirection closures (FetchMes) always use the GA to identify the remote object. The mapping of these GAs to local heap addresses and vice versa is done via a hash table, the global indirection table (GIT). This relationship is further elaborated in the following section and in Figure 1.

This design enables separate local garbage collection (GC) on each PE, provided that the GIT is rebuilt after every GC to map all live GAs to their new addresses in the heap. Rebuilding the GIT is one major source of overhead in the memory management subsystem. 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 cheaper than allocating and maintaining one additional word, the GA, for every closure in the system.

Note, that the mapping of global to local addresses is 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 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. In Section 4.2 we will measure different globalisation schemes.

Independent local garbage collection on each PE is achieved in GUM by using *weighted reference counting* on GAs [1]. On creating a new reference to a globally visible closure it receives half the weight from the original reference. When during local garbage collection a reference to a remote closure is freed, the weight of that reference is added to the reference maintained in the GIT. Once a GA contains the maximal weight, there are no more remote references to this closure and it can be garbage collected.

2.3. The Communication Subsystem

GUM was designed for a generic distributed-memory machine. To reduce the total amount of communication and to permit *latency hiding*, i.e. overlapping communication with computation, GUM uses asynchronous, bulk communication. The *packing scheme*, or serialisation mechanism, 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 its initial portion is sent. Both work and data are represented as graph structures, and can be transmitted using this mechanism. However, we rely on the actual compiled code to be pre-loaded on each PE. In Section 4.2 we will measure different packing schemes.

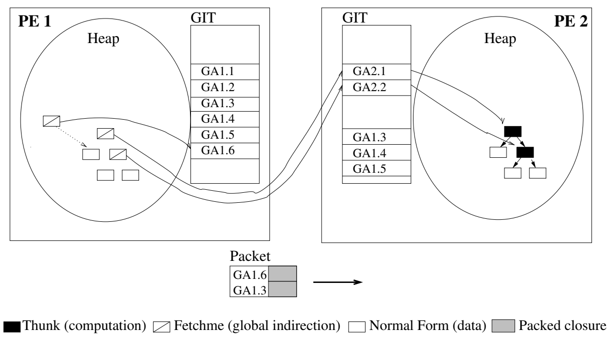

Figure 1. Transfer of graph structures

Figure 1 elaborates on the allocation of GAs and the transfer of graph structures on two PEs. This snapshot shows the heaps on two PEs after having completed the transfer of the five closure graph with root `GA2.1` on PE2 (originally `GA1.1` on PE1). 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, in this example introducing `GA1.1` to `GA1.5`. When packing the closure itself, thunks and normal-forms have to be treated differently. 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" closure. 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 to maintain sharing. For thunks new GAs are allocated, which determine the location of the closure. In this example `GA2.1` and `GA2.2` are newly allocated. Their old GAs, `GA1.1` and `GA1.2`, are not needed any more and can be garbage collected. 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 (FetchMes) to the new GAs, `GA2.1` and `GA2.2`, and the old GAs become garbage.

Figure 1 also shows an ongoing transfer of a two closure graph, that shares one closure with the first graph. Note, that in the packet `GA1.3` refers to the same (shared) closure as the one now available on PE2, so that when unpacking the second graph sharing of this closure is maintained on PE2.

3. Measuring the Virtual Shared Memory Overhead in GUM

3.1. Measurement Setup

All measurements have been performed on up to 16 nodes of a 32-node Beowulf cluster [12] 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.

All runtimes represent the median of three executions to factor out operating system interaction. Program inputs have been chosen to result in a sequential runtime between 3 and 13 minutes. All speedups reported are relative, i.e. the runtime on 1 PE divided by the runtime on n PEs. The sequential efficiency of GUM, i.e. sequential runtime divided by a 1 PE parallel runtime, has previously been measured as 85–90% for programs like `parfib` with a massive amount of parallelism [13], and we have measured a 96% efficiency for `linsolv`, which generates less parallelism.

In measuring the overhead for the three main subsystems in GUM we investigate the following questions:

- *How much parallelism is generated?* This characteristic of the *thread management subsystem* has an important impact on the overhead of the memory management subsystem. In particular, functional languages tend to encourage the generation of a massive amount of parallelism and thus the runtime system has to cope with many fine-grained threads.
- *How expensive is the management of GAs?* This characteristic, which measures the main overhead of the *memory management subsystem*, depends on the number of GAs produced during the computation. In particular the costs for rebuilding the GIT at every garbage collection can be substantial.
- *How expensive is the packing/unpacking of graph structures?* This characteristic, which measures the main overhead of the *communication subsystem*, depends on the packet sizes and the communication degree of the program, i.e. the number of packets sent per second of execution time. An excessive amount of communication is often an indicator of bad data locality, i.e. logically related pieces of data residing on different PEs.

3.2. Benchmark Programs

Six programs are measured as indicated in Table 1. Two are trivial divide-and-conquer programs, `parfib` and `stir`, to test GUM's resilience towards massive parallelism. These divide-and-conquer programs (with thresholding) compute Fibonacchi numbers and Stirling numbers of the first kind, respectively. The `sumEuler` program computes the sum over the application of the Euler totient function over an integer list. It is data parallel (with data clustering) and has a fairly cheap combination phase involving only a small amount of communication. The next two programs, `mandelbrot` and `raytracer`, are data parallel. The former computes a Mandelbrot set in a window of given size. The latter calculates a 2D image of a given scene of 3D objects by tracing all rays in a given grid, or window. Both programs use data parallelism over the window and employ data clustering, i.e. a tunable number of lines is processed by one thread. Compared to the above three programs, significantly more communication is required, but the structure of the parallelism is simple with largely independent computations. The `linsolv` program finds an exact solution of a linear system of equations. In contrast to classical iterative methods used to find an approximate solution, this symbolic algorithm uses a multiple homomorphic images approach. The input is mapped into several homomorphic images, the solutions are computed in each image independently, and finally the results are combined into a solution in the original domain. The computational structure of the parallel algorithm is divide-and-conquer with nested parallelism in the solution phase. Both `raytracer` and `linsolv` are discussed in detail in [7].

Table 1 summarises the dynamic properties of these programs executing on a 16-processor Beowulf cluster (the

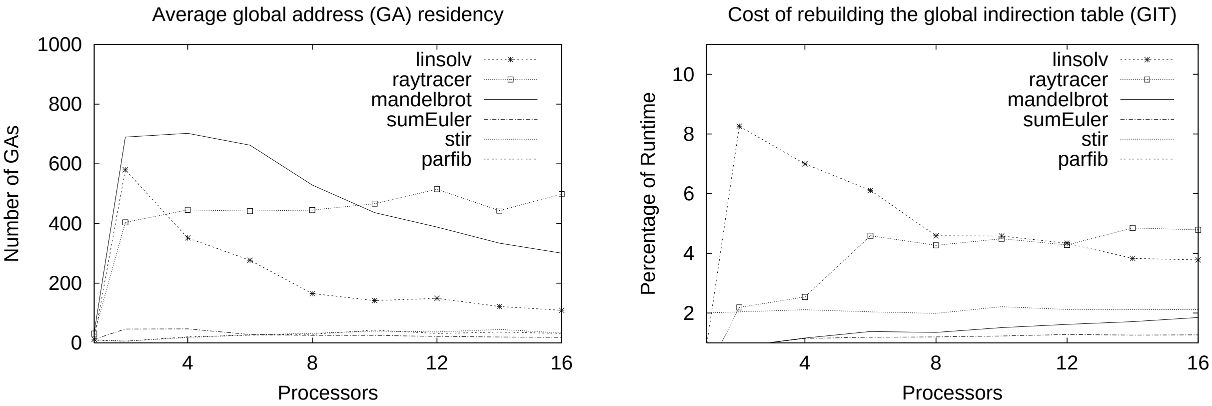

Figure 2. Overhead of maintaining global addresses (GAs) for all programs

Table 1. Dynamic properties of the programs

Program Name	No of Thrds	Max Heap Residency (kB)	Alloc Rate (MB/s)	Comm Degree (pkts/s)
`parfib`	811	5.0	2.5	109.2
`stir`	494	4.0	59.4	44.6
`sumEuler`	153	85.0	29.3	7.9
`mandelb.`	180	1124.0	42.5	539.9
`rayt.`	300	929.0	2.2	336.7
`linsolv`	195	159.0	71.4	265.9

speedups are shown in Figure 6 and will be discussed in Section 4.2). The second column records the total number of threads generated. The remaining columns show averages over all processors for the maximal heap residency, i.e. the maximum amount of heap that is alive at garbage collection time, the allocation rate, i.e. the amount of local memory allocated per second of execution time, and the communication degree, i.e. the number of packets sent per second of execution time. The `parfib` program generates the largest number of threads and sends many small messages, without allocating much heap. The `stir` and `sumEuler` programs consume significantly more heap and have a lower communication rate, indicating a better computation to communication ratio. The `mandelbrot` and `raytracer` programs require a large amount of input data, reflected in a high heap residency, but only `mandelbrot` allocates a significant amount of intermediate data, reflected in a high allocation rate. Both of these programs generate a lot of communication, mainly at the beginning and the end of the computation for exchanging data. The low allocation rate with a relatively high heap residency for `raytracer` reflects its rather static nature of parallelism. Finally, `linsolv` has both a high heap residency and a high allocation rate, with the communication more evenly spread throughout the computation, and thus is the most severe test for the memory management mechanism.

3.3. Memory Management Overhead

In assessing the *thread management subsystem*, thread subsumption proved to be an effective mechanism of bounding the total amount of parallelism. For `parfib` and `stir` the amount of potential parallelism increases exponentially, generating two and three parallel threads in each level of recursion, respectively. The total number of threads is controlled by the implicit thread subsumption mechanism. For all other programs the total number of threads is roughly constant for increasing numbers of PEs. For `parfib`, with input 45 and 11 in total 2^{34} parallel threads could be generated. However, in practice only up to 811 threads are produced. For `stir` only 494 out of 3^{18} potential threads are generated. This ensures that thread creation costs do not radically decrease parallel performance.

A key parameter for assessing the overhead of the *memory management subsystem* is the average GA residency, i.e. the maximal number of live GAs measured over all garbage collections and averaged over all processors. As shown in Figure 2 this number (left hand graph) determines the time required for rebuilding the GIT at the end of every garbage collection (right hand graph), which represents the main overhead for maintaining GAs. For the trivial test programs such as `parfib`, `stir` and `sumEuler` only a very small number of GAs is alive throughout the execution, reflecting the small amount of data that is being transmitted. The corresponding overheads are very small: 0.1%, 2.2%, 1.3%, respectively. The significantly higher numbers of GAs for `mandelbrot`, `raytracer` and `linsolv` have different impacts on the memory management overhead. Both `mandelbrot` and `raytracer` exhibit a smaller allocation rate than `linsolv` (see Table 1), which results in fewer

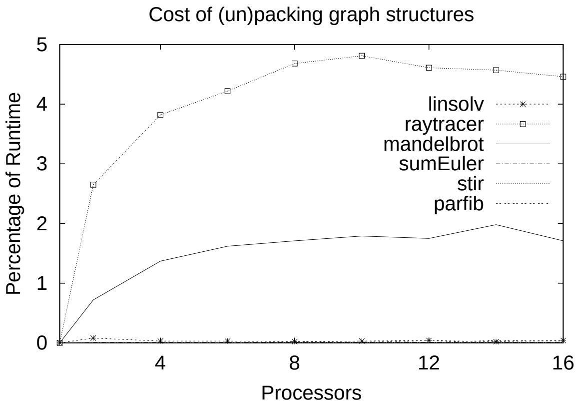

Figure 3. Communication overhead

GCs and thus a smaller impact of rebuilding the GIT on the overall memory management overhead. In the case of `linsolv` the high number of GAs is directly reflected by a high overhead for maintaining the GAs: up to 8.3%.

Interestingly the average number of GAs per processor does not radically increase with an increasing number of processors, as might be expected (see Figure 2). This is mainly due to the ability of the runtime system to garbage collect older GAs once they are not needed any more in the computation. In general, the number of live GAs is an indicator of heap fragmentation, which is program specific, and might also be of interest to the applications programmer.

The main overhead in the *communication subsystem* is time needed for packing and unpacking graph structures. As expected, the average packet size is very small for `parfib`, and `stir`. In total this yields an overhead of less than 0.1%. For `sumEuler`, `mandelbrot`, `raytracer` and `linsolv` the average packet sizes are significantly larger: up to 1692, 3384, 1520, and 548 bytes per packet. However, even then the percentage of packing and unpacking costs, as shown in Figure 3, is low. The `raytracer`, which receives a geometric model as input, exhibits an overhead of up to 4.8%, and `mandelbrot` up to 2%. All other programs have overheads smaller than 0.3%.

4. Improving the Virtual Shared Memory Management

4.1. New Memory Management Techniques

Based on the measurements in the previous section, we have identified the rebuilding of the GIT as the main source of overhead for the virtual shared heap implementation and we have investigated possibilities for reducing the memory management and communication overhead. In particular, we have refined the globalisation and the packing schemes of GUM.

In the rest of the paper we compare the efficiency of two globalisation schemes:

- *full globalisation*, the default setting, which allocates a GA for every closure that is packed and sent to another processor;
- *thunk-only globalisation*, which allocates a GA only for thunks but not for normal-form closures.

This seemingly rather minor change has important consequences for the data distribution in heap intensive applications. The former scheme maintains the sharing of both thunks (i.e. computations) and of normal-forms (i.e. data), whereas the latter may abandon sharing of normal-forms. The potential advantage of this scheme is a reduced overhead in managing GAs.

As a modification of the communication subsystem we also investigate two different packing, or serialisation, schemes:

- *full subgraph packing*, the default setting, which sends an entire subgraph, rooted at the closure originally requested, to the requesting processor;
- *normal-form-only packing*, which sends only data items in the subgraph up to the first thunk that is encountered.

The former scheme performs pre-fetching of data and work, whereas the latter only pre-fetches data, usually resulting in a higher number of small packets.

4.2. Measurements on a High-Latency Machine

In this section we measure the impact of the refined packing and globalisation schemes on the memory management overhead and ultimately on the speedup. We present runtime results for all programs in Table 2. We focus on the discussion of `linsolv` in Figure 5 because it is memory intensive and exhibits irregular parallelism (we use sparse matrices), and thus represents a class of programs for which our programming model is most useful.

Comparing Globalisation Schemes: Figure 4 shows results from executing `linsolv` using all four combinations of globalisation and packing schemes. The left hand graph shows the total number of GAs and the right hand graph shows the overhead for maintaining the GAs. Except for a 2 processor setup, the number of GAs generated by thunk-only globalisation is significantly smaller than for full globalisation, resulting in a drop of GA maintenance overhead from 8.6% to 3.5% on 16 PEs (from 5.5% to 3.8% with full-subgraph packing). With an increasing number of processors this overhead decreases because the larger amount of

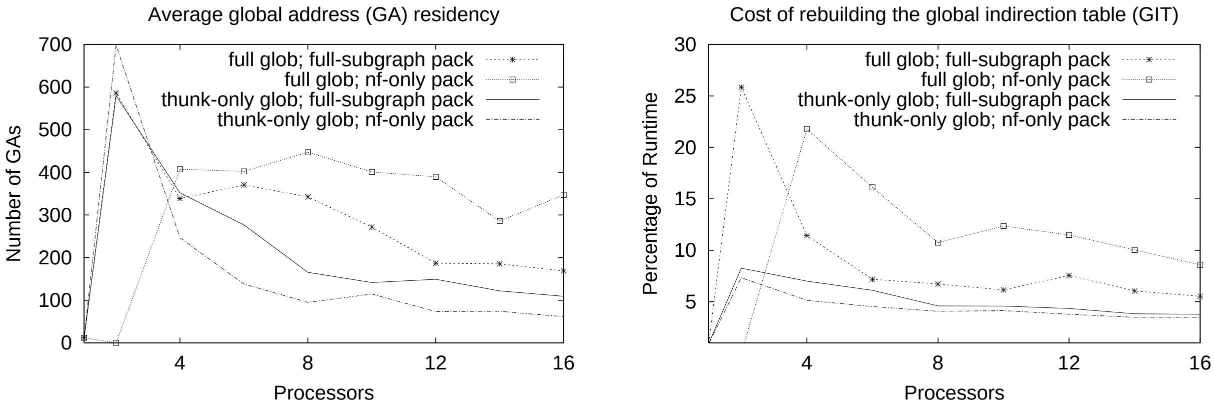

Figure 4. Overhead of maintaining global addresses (GAs) in linsolv

total memory leads to fewer GCs in total. An examination of the GA load, i.e. the number of live GAs on each processor at any point during the execution, reveals a much higher, more even load with full globalisation, whereas thunk-only globalisation produces only short peaks of such high GA load, particularly on the main PE. Longer periods of high GA load, which would have a significant impact on the runtime, are rare with thunk-only globalisation.

Comparing Packing Schemes: The influence of the packing scheme on the number of GAs is less obvious. In general, normal-form-only packing reduces the granularity of the communication. By avoiding unnecessary prefetching of work it can reduce the number of GAs that are allocated and improve data-locality. However, this advantage might be off-set by the increased number of packets in total. For `linsolv` the maximal number of packets increases from 4300 to 5600 on 16 PEs, which in turn leads to a higher number of GAs when combining normal-form only packing with full globalisation. However, with thunk-only globalisation, which generates far fewer GAs per packet, the total number of GAs drops.

Speedups: Figure 5 compares the speedups for all four combinations of globalisation and packing schemes. In general, thunk-only globalisation combined with normal-form only packing shows the best absolute performance with the best speedup of 13.8 on 16 PEs. Although thunk-only globalisation improves performance in most cases, we observe drops in performance when combined with normal-form-only packing for example at 10 PEs. This behaviour is due to an increased amount of communication that exceeds, at this point, the system's ability of latency hiding and thus introduces idle time in the execution.

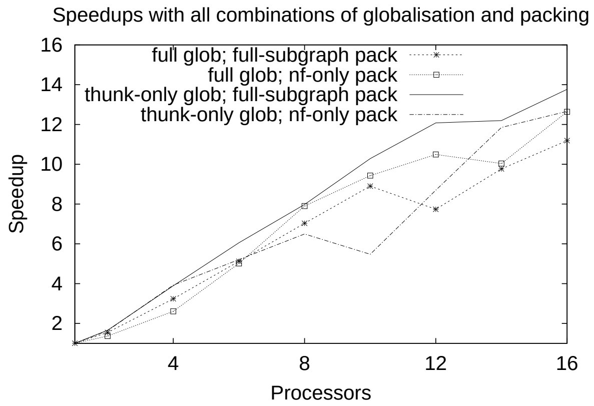

Figure 5. Speedups for linsolv

Other Programs: Table 2 summarises the reduction in runtime on 16 PEs relative to an execution in the default setting of full globalisation and full-subgraph packing. The first two lines specify the globalisation and the packing scheme, respectively. As expected, `parfib` and `stir` show little variation. As discussed, `linsolv` shows consistent improvements for all processors, with a reduction in

Table 2. Reduction in runtime (in %)

Program Name	thunk-only nf-only	thunk-only full-subgraph	full nf-only
`parfib`	+1.0	**-2.9**	+2.1
`stir`	-0.5	-2.3	**-4.8**
`sumEuler`	-9.2	**-12.2**	+9.2
`mandelb.`	-45.8	**-48.7**	-2.0
`rayt.`	-16.8	**-35.5**	+23.1
`linsolv`	-11.6	**-18.7**	-11.5

runtime of 18.7% in the best case. The data parallel programs, `mandelbrot` and `raytracer`, show improvements far exceeding the overhead for maintaining GAs. This is due to different sequences of interactions between the PEs caused by the different costs for the globalisation and packing schemes and by the independent local GCs. In particular, a change in the interactions between the PEs affects the dynamic distribution of work. The resulting difference in the load distribution can have a high impact on the total runtime. To some degree this dynamic behaviour is program specific: data parallel programs that distribute large data structures at the beginning will suffer from increased overhead due to full globalisation, and a too small degree of parallelism bears the danger of drastic load imbalances at the end of the computation. However, in factoring out the effects of dynamic load distribution, we observe a significant drop in the costs for maintaining GAs alone: from 6.5% to 1.8% for `mandelbrot` and from 4.8% to 0.3% for `raytracer`.

When comparing different packing schemes the difference in runtime is less pronounced and mixed. For `linsolv` normal-form-only packing achieves an improvement of 11.5%. However, `raytracer` and `mandelbrot`, which both transfer large data structures, perform better with full-subgraph packing. Overall we cannot clearly identify a winner for the packing scheme alone.

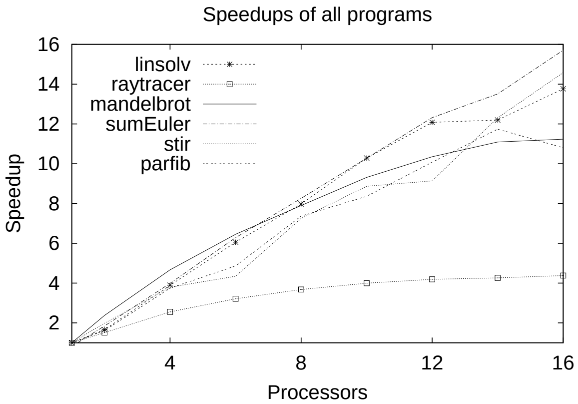

Figure 6. Speedups with thunk-only globalisation and full-subgraph packing

Finally, Figure 6 summarises the speedups obtained from all example programs when using a combination of thunk-only globalisation and full-subgraph packing, which achieves the best results as shown in Table 2. The comparatively low communication degrees of `parfib`, `stir` and `sumEuler` ensure good speedups for these simple programs. The data parallel programs, `mandelbrot` and `raytracer`, profit from fewer GCs initially but show comparatively poor speedups for 16 PEs. This is mainly due to a sequential bottleneck at the end when collecting the generated data from many PEs, contrasting a high utilisation in early stages of the execution.

5. Related work

Most closely related to our work are Kesseler's studies on graph copying costs in a distributed implementation of the functional language Concurrent Clean [5]. However, rather than adjusting the generic graph packing algorithm, which is similar to our normal-form-only packing, he focuses on using arrays as representation of data that has to be transported.

Another virtual shared heap implementation of a parallel functional language is pHluid [2]. In this implementation the heap is partitioned into pages, being owned by a specific processor. A cache coherence protocol decides when an entire page has to be sent to another processor, resulting in very coarse-grained communication without the flexibility of a special treatment based on the type of the closure, which we have in GUM.

Goldstein [3] presents a thorough analysis of the performance of different thread management and memory management subsystems in the context of the dataflow-inspired TAM machine. His focus, however, is on the efficiency of a work stealing scheduler with the transfer of stack segments (for different stack representations) rather than arbitrary heap objects. In contrast, our work makes use of the highly structured representation of both work and data to improve distributed memory management, but does not investigate the overhead of different thread representations.

6. Conclusions

In analysing the virtual shared memory overhead incurred in our parallel graph reduction machine GUM on a high-latency Beowulf cluster we have tackled the following questions in Section 3:

- *How much parallelism is generated?* GUM's thread subsumption mechanism can effectively limit the total amount of parallelism generated by divide-and-conquer programs: `parfib` produces only 811 out of 2^{34} potential threads; `stir` produces 494 out of 3^{18} threads. This confirms earlier results on the low-latency GRIP graph reducer [4].

- *How expensive is the management of global addresses?* For the memory intensive programs, `mandelbrot`, `raytracer` and `linsolv`, maintaining the global addresses can amount to as much as 8% of the total execution time (`linsolv`). Typical values lie between 4 and 6% for these memory intensive programs and below 2% for the remaining programs.

- *How expensive is the (un)packing of graphs structures?* Only for the two data parallel programs that require large data structures as input, `mandelbrot` and `raytracer`, does this overhead exceed 1%. Typical values lie between between 1% and 2% for these programs and below 0.3% for the remaining programs.

To reduce the overhead of maintaining global addresses, which represents the highest overhead of the virtual shared heap, we have implemented alternative runtime system techniques for key operations in memory management and communication: globalisation and packing schemes.

A *globalisation scheme* offers a choice between *optimising for speed or for space*. By avoiding to allocate global addresses for data the memory management overhead is reduced, but sharing of data might be lost. All programs, except `stir`, run fastest when avoiding to globalise data. However, the effectiveness of the refined globalisation scheme depends on the dynamic properties of the program. The data parallel programs show the largest improvements, but they are in part due to altered processor interactions and load distribution. The `linsolv` program with its high memory consumption and irregular parallelism gains about 18.7% in performance with thunk-only globalisation.

A *packing scheme* allows to *tune the granularity of the communication* via pre-fetching either both data and work or only data. In combination with the best globalisation scheme, full subgraph packing delivered the best performance. For `linsolv` this is the best combination across the entire range of PEs. For the other programs the choice of the best packing scheme depends on the communication characteristics of the program. Deriving such information either via profiling or program analysis would be required to choose the best packing scheme in general.

As future work we plan to investigate the possibility of generating lifetime profiles for global addresses to measure heap fragmentation. To further reduce memory management overhead the currently flat global indirection table could be partitioned into generations, matching the structure of a generational garbage collector. A longer term goal is to develop a more sophisticated environment, that can deduce system parameters and choose the appropriate runtime system scheme, e.g. for packing or globalisation, automatically based on these parameters. Such a self-adjusting runtime system would be far more flexible in providing high architecture-independent performance.

Acknowledgements

The author would like to thank André Rauber Du Bois, Greg Michaelson, and Phil Trinder for commenting on drafts of the paper and the Austrian Academy of Sciences for funding this work under APART fellowship 624.

References

[1] D. Bevan. Distributed Garbage Collection Using Reference Counting. In *PARLE'87 — Parallel Architectures and Languages Europe*, LNCS 259, pages 176–187, Eindhoven, The Netherlands, June 12–16, 1987. Springer-Verlag.

[2] C. Flanagan and R. Nikhil. pHluid: The Design of a Parallel Functional Language Implementation on Workstations. In *ICFP'96 — Intl Conf on Functional Programming*, pages 169–179, Philadelphia, PA, May 24–26, 1996. ACM Press.

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

[4] K. Hammond and S. Peyton Jones. Profiling Scheduling Strategies on the GRIP Multiprocessor. In *IFL'92 — Intl Workshop on the Implementation of Functional Languages*, pages 73–98, RWTH Aachen, Germany, Sept. 1992.

[5] M. Kesseler. Reducing Graph Copying Costs. In *PASCO'94 — Intl Symp on Parallel Symbolic Computation*, Linz, Austria, Sept. 26–28, 1994. World Scientific Publishing.

[6] H.-W. Loidl. Load Balancing in a Parallel Graph Reducer. In *Trends in Functional Programming*, volume 3, pages 63–74. Intellect, 2002.

[7] H.-W. Loidl, P. Trinder, K. Hammond, S. Junaidu, R. Morgan, and S. Peyton Jones. Engineering Parallel Symbolic Programs in GPH. *Concurrency – Practice and Experience*, 11:701–752, 1999. Available from: <URL:`http://www.cee.hw.ac.uk/~dsg/gph/`>.

[8] E. Mohr, D. Kranz, and R. 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.

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

[10] S. Peyton Jones, C. Clack, J. Salkild, and M. Hardie. GRIP — a High-Performance Architecture for Parallel Graph Reduction. In *FPCA'87 — Conf on Functional Programming Languages and Computer Architecture*, LNCS 274, pages 98–112, Portland, OR, Sept. 14–16, 1987. Springer-Verlag.

[11] S. Peyton Jones, C. Hall, K. Hammond, W. Partain, and P. Wadler. The Glasgow Haskell Compiler: a Technical Overview. In *Joint Framework for Information Technology Technical Conference*, pages 249–257, Keele, UK, Mar. 1993. Available from: <URL:`http://www.haskell.org/ghc/`>.

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

[13] P. Trinder, K. Hammond, J. Mattson Jr., A. Partridge, and S. 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.