GUM: a portable parallel implementation of Haskell

Abstract

Department of Computing Science, Glasgow University
Email: {trinder,kh,simonpj}@dcs.glasgow.ac.uk

PW Trinder K Hammond JS Mattson Jr * AS Partridge † SL Peyton Jones ‡

Abstract

GUM is a portable, parallel implementation of the Haskell functional language. Despite sustained research interest in parallel functional programming, GUM is one of the first such systems to be made publicly available.

GUM is message-based, and portability is facilitated by using the PVM communications harness that is available on many multi-processors. As a result, GUM is available for both shared-memory (Sun SPARCserver multiprocessors) and distributed-memory (networks of workstations) architectures. The high message-latency of distributed machines is ameliorated by sending messages asynchronously, and by sending large packets of related data in each message.

Initial performance figures demonstrate absolute speedups relative to the best sequential compiler technology. To improve the performance of a parallel Haskell program GUM provides tools for monitoring and visualising the behaviour of threads and of processors during execution.

1 Introduction

GUM (Graph reduction for a Unified Machine model) is a portable, parallel implementation of the non-strict purely-functional programming language Haskell. Despite hundreds of papers, dozens of designs, and a handful of real single-site implementations, GUM is one of the first such systems to be made publicly available. We believe that this is partly because of the diversity of parallel machine architectures, but also because the task of implementing a parallel functional language is much more substantial than it first appears. The goal of this paper is to give a technical overview of GUM, highlighting our main design choices, and present preliminary performance measurements.

GUM has the following features:

- *GUM is portable.* It is message based, and uses PVM [29], a communication infrastructure available on almost every multiprocessor, including both shared-memory and distributed-memory machines, as well as networks of workstations. The basic assumed architecture is that of a collection of processor-memory units (which we will call PEs) connected by some kind of network that is accessed through PVM.

  PVM imposes its own overheads, but there are shortcuts that can be taken for communication between homogeneous machines on a single network. In any case, for any particular architecture a machine-specific communications substrate could readily be substituted.

- *GUM can deliver significant absolute speedups,* relative to the best sequential compiler technology[1]. It does so by using one of the best available sequential Haskell compilers as its basis, namely the Glasgow Haskell Compiler (GHC). Indeed GUM is "just" a new runtime system for GHC. The sequential parts of a program run as fast as if they were compiled by GHC for a sequential machine, apart from a small constant-factor overhead (Section 4.1)

- *GUM provides a suite of tools for monitoring and visualising the behaviour of programs.* The bottom line of any parallel system is raw performance, and a program's performance can only be improved if it can be understood. In addition to conventional sequential tools, GUM provides tools to monitor and visualise both PE and thread activity over time.

- *GUM supports independent local garbage collection, within a single global virtual heap.* Each PE has a local heap that implements part of the global virtual heap. A two-level addressing scheme distinguishes *local addresses*, within a PE's local heap, from *global addresses*, that point between local heaps. The management of global addresses is such that each PE can

*Author's present address: Hewlett Packard, California. Email: jmattson@cup.hp.com.

†Author's present address: Department of Computer Science, University of Tasmania. Email: A.S.Partridge@cs.utas.edu.au.

‡This work is supported by an SOED personal research fellowship from the Royal Society of Edinburgh, and the UK EPSRC AQUA and Parade grants.

[1] Needless to say, this claim relates to programs with lots of large-grain parallelism, and the name of the game is seeing how far it extends to more realistic programs. Nevertheless, such tests provide an important sanity check: if the system does badly here then all is lost.

garbage-collect its local heap independently of any other PE, a property we found to be crucial on the GRIP multiprocessor [30].

- *Thread distribution is performed lazily, but data distribution is performed somewhat eagerly.* Threads are never exported to other PE to try to "balance" the load. Instead, work is only moved when a processor is idle (Section 2.2). Moving work prematurely can have a very bad effect on locality.

  On the other hand, when replying to a request for a data value, a PE packs (a copy of) "nearby" data into the reply, on the grounds that the requesting PE is likely to need it soon (Section 2.4). Since the sending PE retains its copy, locality is not lost.

- *All messages are asynchronous.* The idea — which is standard in the multithreading community [1] — is that once a processor has sent a message it can forget all about it and schedule further threads or messages without waiting for a reply (Section 2.3.4). Notably, when a processor wishes to fetch data from another processor it sends a message whose reply can be arbitrarily delayed — for example, the data might be under evaluation at the far end. When the reply finally does arrive, it is treated as an independent work item.

Messages are sent asynchronously and contain large amounts of graph in order to ameliorate the effects of long-latency distributed machines. Of course there is no free lunch. Some parallel Haskell programs may work much less well on long-latency machines than short-latency ones, but nobody knows to what extent. One merit of having a single portable framework is that we may hope to identify this extent.

GUM is freely available by FTP, as part of the Glasgow Haskell Compiler release (0.26 onwards). It is currently ported to networks of Sun SPARCs and DEC Alphas, and to Sun's symmetric multiprocessor SPARCserver.

The remainder of this paper is structured as follows. Section 2 describes how the GUM run-time system works. Section 3 describes the tools available for monitoring the performance of programs running under GUM. Section 4 gives preliminary performance results. Section 5 discusses related work. Section 6 outlines some directions for its development. Section 7 concludes.

2 How GUM works

The first action of a GUM program is to create a PVM manager task, whose job is to control startup, termination and synchronise garbage collection. This manager task then spawns the required number of logical PEs as PVM tasks, which PVM maps to the available processors. Each PE task then initialises itself: processing runtime arguments, allocating heap etc. Once all PE tasks have initialised, and been informed of each others identity, one of the PE-tasks is nominated as the *main PE*. The main PE then begins executing the main thread of the Haskell program.

The program terminates when either the main thread completes, or an error is encountered. In either case a FINISH message is sent to the manager task, which in turn broadcasts a FINISH message to all of the PE tasks. The manager waits for each PE task to respond before terminating the program.

During execution each PE executes the following scheduling loop until it receives a FINISH message.

Main Scheduler:

1. Perform local garbage collection, if necessary (Section 2.3).
2. Process any incoming messages from other PEs, possibly sending messages in response.
3. If there are runnable threads, run one of them (Section 2.1).
4. Otherwise look for work (Section 2.2).

The inter-PE message protocol is completely asynchronous. When a PE sends a message it does not await a reply; instead it simply continues, or returns to the main scheduler. Indeed, sometimes the reply may be delayed a long time, if (for example) it requests the value of a remote thunk that is being evaluated by some other thread. All of this is standard in the multithreading community [1].

2.1 Thread Management

A *thread* is a virtual processor. It is represented by a (heap-allocated) Thread State Object (TSO) containing slots for the thread's registers. The TSO in turn points to the thread's (heap-allocated) Stack Object (SO). As the thread's stack grows, further Stack Objects are allocated and chained on to the earlier ones.

Each PE has a pool of runnable threads — or, rather, TSOs — called its *runnable pool*, which is consulted in step (3) of the scheduling loop given earlier. Currently, once a thread has begun execution on a PE it cannot be moved to another PE. This makes programs with only limited parallelism vulnerable to scheduling accidents, in which one PE ends up with several runnable (but immovable) threads, while others are idle. In the future we plan to allow runnable threads to migrate.

When a thread is chosen for execution it is run until either space is exhausted, the thread blocks (either on another thread or accessing remote data), or the thread completes. Compared with fair scheduling, this has the advantage of tending to decrease both space usage and overall run-time [4], at the cost of making concurrent and speculative execution rather harder.

2.1.1 Sparks

Parallelism is initiated by the `par` combinator in the source program. (At present these combinators are added by the programmer, though we would of course like this task to be

automated.) When the expression

```
x `par` e
```

is evaluated, the heap closure referred to by the variable `x` is *sparked*, and then `e` is evaluated. Quite a common idiom (though by no means the only way of using `par`) is to write

```
let x = f a b in x `par` e
```

where `e` mentions `x`. Here, a *thunk* (or suspension) representing the call `f a b` is allocated by the `let` and then sparked by the `par`. It may thus be evaluated in parallel with `e`.

Normally sparking a thunk is a relatively cheap operation, consisting only of adding a pointer to the thunk to the PE's *spark pool*. However, if there isn't a thunk in the heap, we may need to create one in order to spark it. The costs of forced thunk creation are investigated in Section 4.1. A spark is an indication that a thunk *might* usefully be evaluated in parallel, not that it *must* be evaluated in parallel. Sparks may freely be discarded if they become too numerous.

2.1.2 Synchronisation

It is obviously desirable to prevent two threads from evaluating the same thunk simultaneously, lest the work of doing so be duplicated. This synchronisation is achieved as follows:

1. When a thread enters (starts to evaluate) a thunk, it overwrites the thunk with a *black hole* (so called for historical reasons).
2. When a thread enters a black hole, it saves its state in its TSO, attaches its TSO to the queue of threads blocked on the black hole (the black hole's *blocking queue*), and enters the scheduler.
3. When a thread completes the evaluation of a thunk, it overwrites the latter with its value (the *update* operation). When it does so, it moves any queued TSOs to the runnable pool.

Notice that synchronisation costs are only incurred if two threads actually collide. In particular, if a thread sparks a sub-expression, and then subsequently evaluates that sub-expression before the spark has been turned into a thread and scheduled, then no synchronisation cost is incurred. In effect the putative child thread is dynamically inlined back into the parent, and the spark becomes an orphan.

2.2 Load distribution

If (and only if) a PE has nothing else to do, it tries to schedule a spark from its spark pool, if there is one. The spark may by now be an orphan, because the thunk to which it refers may by now be evaluated, or under evaluation by another thread. If so, the PE simply discards the spark. If the PE finds a useful spark, it turns it into a thread by allocating a fresh TSO and SO[2], and starts executing it.

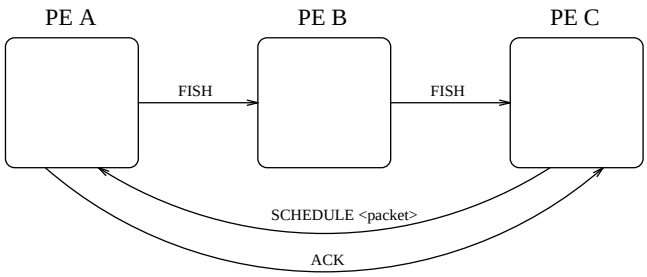

Figure 1: Fish/Schedule/Ack Sequence

If there are no local sparks, then the PE seeks work from other PEs, by launching a FISH message that "swims" from PE to PE looking for available work. Initially only the main PE is busy — has a runnable thread — and all other PEs start fishing for work as soon as they begin execution.

When a FISH message is created, it is sent at random to some other PE. If the recipient has no useful sparks, it increases the "age" of the FISH, and sends the FISH to another PE, again chosen at random. The "age" of a FISH limits the number of PEs that a FISH visits: having exceeded this limit, the last PE visited returns the unsuccessful FISH to the originating PE. On receipt of its own, starved, FISH the originating PE then delays briefly before launching another FISH. The purpose of the delay is to avoid swamping the machine with FISH messages when there are only a few busy PEs. A PE only ever has a single FISH outstanding.

If the PE that receives a FISH has a useful spark, it sends a SCHEDULE message to the PE that originated the FISH, containing the sparked thunk packaged with nearby graph, as described in Section 2.4. The originating PE unpacks the graph, and adds the newly acquired thunk to its local spark pool. An ACK message is then sent to record the new location of the thunk(s) sent in the SCHEDULE (Section 2.4). Note that the originating PE may no longer be idle because before the SCHEDULE arrives, another messages may have unblocked some thread. A sequence of messages initiated by a FISH is shown in Figure 1.

2.3 Memory Management

Parallel graph reduction proceeds on a shared program/data graph, so a primary function of the run-time system of a parallel functional language is to manage the virtual shared memory in which the graph resides.

2.3.1 Local Addresses

As mentioned earlier, GUM is based on the Glasgow Haskell Compiler, and most execution is carried out in precisely the

[2] Since we know exactly when we discard TSOs and SOs, and they are relatively large, we keep them on a free list so that we can avoid chomping through heap when executing short-lived tasks.

same way as on a uniprocessor. In particular:

- Each PE has its own local heap.
- New heap closures are allocated from a contiguous chunk of free space in this local heap.
- The heap-closure addresses manipulated by the compiled code are simply one-word pointers within the local heap, a *local address*.
- Each PE can perform *local garbage collection* independently of all the other PEs. This crucial property allows each PE cheaply to recycle the "litter" generated by normal execution.

Sometimes, though, the run-time system needs to move a heap closure from one PE's local heap to another's. For example, when a PE (call it A) with plenty of sparks receives a FISH message, it sends one of its sparked thunks to the idle PE (call it B). When a thunk is moved in this way, the original thunk is overwritten with a *FetchMe closure*, containing the *global address* of the new copy on B. Why does the thunk need to be overwritten? It would be a mistake simply to copy it, because then both A and B might evaluate it separately (remember, there might be other local pointers to it from A's heap).

2.3.2 Global Addresses

At first one might think that a global address (GA) should consist of the identifier of the PE concerned, together with the local address of the closure on that PE. Such a scheme would, however, prevent the PEs from performing compacting garbage collection, since that changes the local address of most closures. Since compacting garbage collection is a crucial component of our efficient compilation technology we reject this restriction.

Accordingly, we follow standard practice and allocate each globally-visible closure an immutable *local identifier* (typically a natural number). A global address consists of a (PE identifier, local identifier) pair. Each PE maintains a Global Indirection Table, or *GIT*, which maps local identifiers to the local address of the corresponding heap closure. The GIT is treated as a source of roots for local garbage collection, and is adjusted to reflect the new locations of local heap closures following local garbage collection[3]. We say that a PE *owns* a globally-visible closure (that is, one possessing a global address) if the closure's global address contains that PE's identifier.

A heap closure is *globalised* (that is, given a global address) by allocating an unused local identifier, and augmenting the GIT to map the local identifier to the closure's address. Of course, it is possible that the closure already has a global address. We account for this possibility by maintaining (separately in each PE) a mapping from local addresses to global addresses, the *LA* $\rightarrow$ *GA table*, and checking it before globalising a heap closure. Naturally, the LA $\rightarrow$ GA table has to be rebuilt during garbage collection, since closures' local addresses may change.

A PE may also hold copies of globally-visible heap closures owned by another PE. For example, PE A may have a copy of a list it obtained from PE B. Suppose the root of the list has GA (B,34). Then it makes sense for A to remember that the root of its copy of the list also has GA (B,34), in case it ever needs it again. If it does, then instead of fetching the list again, it can simply share the copy it already has.

We achieve this sharing by maintaining (in each PE) a mapping from global addresses to local addresses, the PE's *GA* $\rightarrow$ *LA table*. When A fetches the list for the first time, it enters the mapping from (B,34) to the fetched copy in its GA $\rightarrow$ LA table; then, when it needs (B,34) again it checks the GA $\rightarrow$ LA table first, and finds that it already has a local copy.

To summarise, each PE maintains three tables:

- Its GIT maps each allocated local identifier to its local address.
- Its GA $\rightarrow$ LA table maps some *foreign* global addresses (that is, ones whose PE identifier is non-local) to their local counterparts. Notice that each foreign GA maps to precisely one LA.
- Its LA $\rightarrow$ GA table maps local addresses to the corresponding global address (if any).

Whilst there are logically three tables, in practice we represent them by a single data structure.

2.3.3 Garbage collection

This scheme has the obvious problem that once a closure has an entry in the GIT it cannot ever be garbage collected (since the GIT is used as a source of roots for local garbage collection), nor can the local identifier be re-used. Again following standard practice [22], we use *weighted reference counting* to recover local identifiers, and hence the closures they identify [2, 35].

We augment both the GIT and the GA $\rightarrow$ LA table to hold a *weight* as well as the local address. The invariant we maintain is that *for a given global address, G, the sum of:*

- *G's weight in the GA* $\rightarrow$ *LA tables of all foreign PEs, and*
- *G's weight in its owner's GIT, and*
- *the weight attached to any Gs inside any in-flight messages*

is equal to MaxWeight, a fixed constant. With this invariant in mind, we can give the following rules for address management, which are followed independently by each PE:

1. Any entries in a PE's GIT that have weight *MaxWeight* can be discarded, and the local identifier made available for re-use. (Reason: because of the invariant, no

[3] The alert reader will have noticed that we will need some mechanism for recovering and re-using local identifiers, a matter we will return to shortly.

other PEs or messages refer to this global address.) All the other entries must be treated as roots for local garbage collection.

2. A PE can choose whether or not to treat the local addresses in itss GA $\rightarrow$ LA table are treated as roots for local garbage collection. If it has plenty of space available, it can treat them as roots, thereby preserving local copies of global closures in the hope that they will prove useful in the future.

   If the PE is short of space, it refrains from treating them as roots. After local garbage collection is complete, the GA $\rightarrow$ LA table is scanned. Any entries whose local closure has (for some other reason) been identified as live by the garbage collector are redirected to point to the closure's new location. Any entries whose closure is dead are discarded, and the weight is returned to the owning PE in a FREE message, which in turn adds the weight in the message to its GIT entry (thereby maintaining the invariant).

3. If a PE sends a GA to another PE, the weight held in the GIT or GA $\rightarrow$ LA table (depending on whether the GA is owned by this PE or not) is split evenly between the GA in the message and the GA remaining in the table. The receiving PE adds the weight to its GIT or GA $\rightarrow$ LA table, as appropriate.

4. If the weight in a GA to be sent is 1 it can no longer be split, so instead a new GA is allocated with the same local address. This is unfortunate because it introduces global aliases meaning that some sharing is not preserved, but we hope rare. To prevent every subsequent shipping of the GA from allocating a new GA, we identify the new GA, with weight to give away, as the *preferred* GA. LA $\rightarrow$ GA lookup always returns the preferred GA.

The only garbage not collected by this scheme consists of cycles that are spread across PEs. We plan ultimately to recover these cycles too by halting all PEs and performing a global collective garbage collection, but we have not yet even begun its implementation. In practice, local garbage collection plus weighted reference counting seems to recover most garbage.

2.3.4 Distributing Data

Global references are handled by special *Fetch-Me* closures. When a thread enters a Fetch-Me the following steps are carried out:

1. The Fetch-Me closure is globalised, i.e. given a new local GA. (It will already have a foreign GA, namely the GA of the remote object, so this step creates a local alias for it.)

2. The Fetch-Me closure is overwritten with a *Fetching* closure.

3. The demanding thread is blocked, by queueing its TSO on a blocking queue attached to the Fetching closure.

4. A FETCH message is sent to the PE that owns the GA in the Fetch-Me.

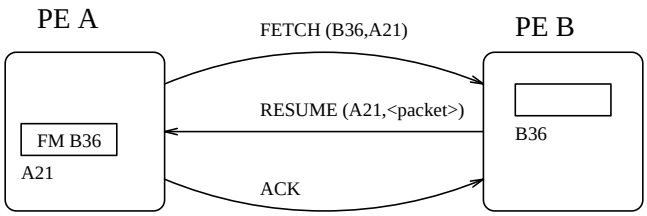

Figure 2: Fetch/Resume/Ack Sequence

5. The PE then returns to the main scheduler: i.e it may run other threads, garbage collect or process messages while awaiting the response to the FETCH. Any subsequent thread that demands the same foreign closure will also join the queue attached to the Fetching closure.

On receipt of a FETCH message, the target PE packages up the appropriate closure, together with some "nearby" graph, and sends this in a RESUME message to the originator. When the RESUME arrives, the originating PE unpacks the graph, restarts the thread(s) that were blocked on the Fetching closure, and redirects the Fetching closure to point to the root of this graph[4]. Having done this, an ACK message is returned to the PE that sent the RESUME (the following section explains why). Figure 2 depicts the whole process.

2.4 Packing/Unpacking Graph

When a closure is requested we also speculatively pack some "nearby" reachable graph into the same packet, with the object of reducing the number of explicit FETCH messages that need to be sent.

Packing arbitrary graph is a non-trivial problem, and the full paper discusses related work in this area [34, 27, 15], together with the algorithms and heuristics used in GUM to pack graph into packets. It also carefully describes the mechanism that ensures that, even when a thunk is transferred between PEs, there is always exactly one copy of it. For the purposes of this abstract we briefly outline the packing process.

Packing

Packing proceeds closure by closure, breadth-first into a single packet. As each closure is packed its address is recorded in a temporary table so that sharing and cyclic structures can be preserved. We stop packing when either all reachable graph has been packed, or the packet is full. Once the packet is full, any outstanding closures are globalised and packed as Fetch-Mes. Figure 3 shows an example of this.

The packet into which the graph is packed has a short

[4] Actually, it is possible that the RESUME might include in its "nearby" graph some closures for which there are other Fetch-Me or Fetching closures on the recipient PE. If so, they are each redirected to point to the appropriate closure in the newly-received graph, and any blocked threads are restarted.

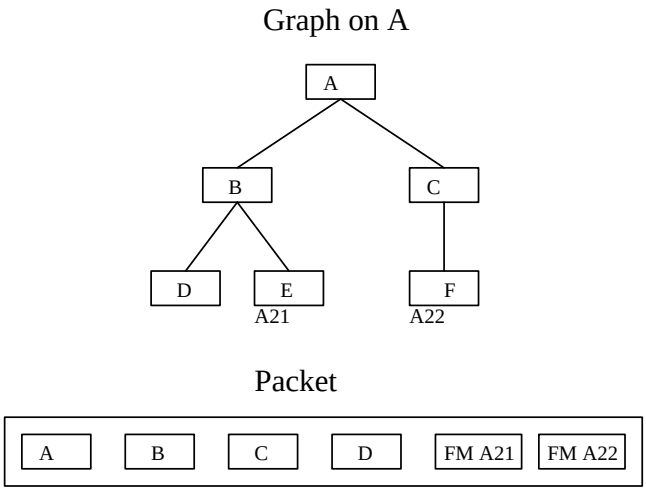

Figure 3: A Full Packet

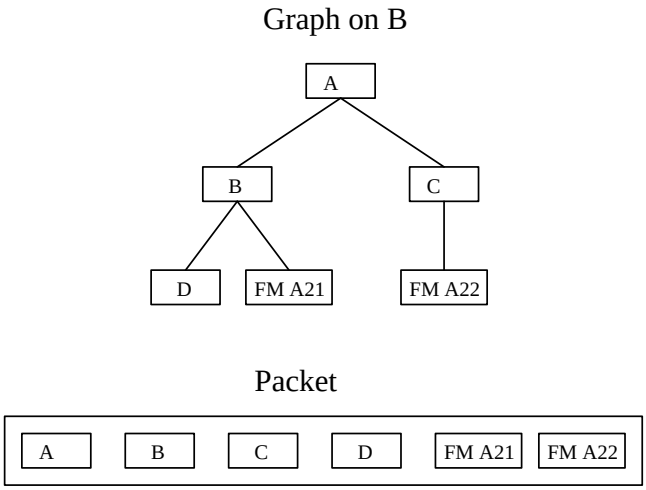

Figure 4: Unpacked Packet

header, which indicates how much heap will be needed to unpack the graph, followed by the actual packed closures. Each packed closure has exactly the same form as the corresponding heap closure, except that it has an additional GA/tag field that precedes the closure proper, and any pointer fields in the closure are omitted in the packet. The GA/tag field normally contains the closure's GA, but some special closure types use this as a tag field, as described in the full paper. We can omit the pointer field in packed closures, because the graph structure is implicit in the order in which closures appear in the packet.

Because the values of expressions remain constant, any normal form closure can be freely copied between processors. In contrast, thunks must be treated carefully since each one represents a potentially unbounded amount of work, that generally should not be performed more than once. The packing algorithm is careful to ensure that only one copy of a thunk ever exists. This is the reason for the ACK messages mentioned earlier; their purpose is to redirect the original copy of the thunk to refer instead to the new copy.

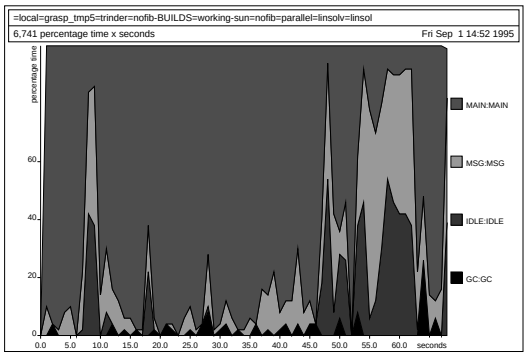

Figure 5: A PE Activity Profile

Unpacking

Unpacking traverses the packet, reconstructing the graph in a breadth-first fashion. The unpacking of the packet in Figure 3 results in the graph depicted in Figure 4. As each closure is unpacked the GA $\rightarrow$ LA table is interrogated to find existing local copies of the closure. If no copy exists, then the GA $\rightarrow$ LA and LA $\rightarrow$ GA tables are updated appropriately. However, if there is already a local copy of the closure, care is taken to choose the closure requiring the least amount of communication. For example, an incoming normal-form closure is preferred to an existing Fetch-Me. The weight of the incoming GA is added to the weight of the existing reference. The duplicate is overwritten by an indirection to the more defined closure.

3 Performance Monitoring

The bottom line of any parallel system is the raw performance that can be delivered. Performance monitoring tools can improve understanding of a program's behaviour, particularly if the results can be visualised. Because each PE task executes an elaborated version of the sequential Haskell runtime system, some of the profiling tools that are provided with sequential Haskell still yield information that can be used to tune parallel programs. Example sequential profiles are given, and their use is discussed in the full paper.

GUM also provides specific tools that give per-PE or per-thread activity profiles over time. Choosing the time base against which to record measurements in a distributed, multi-programmed environment is hard. The full paper considers this issue and the consequences of using both elapsed-time and virtual-time i.e. the time when the program is actually scheduled.

3.1 PE Activity Profiles

GUM uses cost-centre profiling [33] internally to record the activity of each PE during reduction. Figure 5 gives the PE activity profile for one of four PEs evaluating a linear

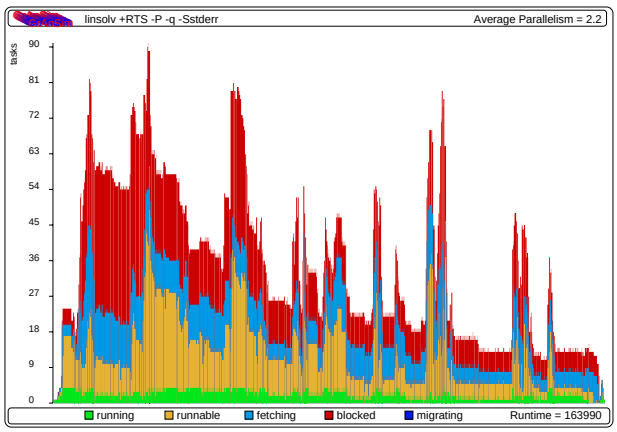

Figure 6: A Thread Activity Profile

equation program. An activity profile allocates time to four cost centres:

- **Main** — time spent performing reduction.
- **Msg** — time spent communicating.
- **GC** — time spent garbage collecting.
- **Idle** — time spent with no reduction, messages or garbage collection to perform.

PE activity profiling is performed in virtual time largely because it is convenient to record the statistics when a virtual clock 'ticks'. The virtual times differ for each PE, and currently there is no way of combining the activity graphs for all the PEs to construct a profile for the entire machine.

3.2 Thread Activity Profiles

An alternative style of profiling is provided by the visualisation tools originally produced for the GranSim simulator. GranSim was constructed to investigate aspects of parallel graph reduction [14], and it can be configured to simulate the running of programs compiled for GUM on most parallel machines.

When a GUM program is run with profiling enabled, each PE accumulates a record of significant execution events events (notably thread creation, blocking, and termination). These records are written to a separate file for each PE, merged after the run, and post-processed to produce various kinds of information, such as the average thread length, or profiles of thread activity against time. For example, the thread activity profile for the `linsolv` run is given in Figure 6. The number of running, runnable, fetching and blocked threads is plotted against time.

4 Preliminary Results

This section reports results of experiments performed to verify that the basic mechanisms in GUM are working properly, and also to perform preliminary performance evaluation and tuning. The results should be viewed as indicative that speedups are possible using GUM, rather than conclusive evidence that GUM speeds-up real programs. An investigation of the load distribution is reported in the full paper, and we plan to report the performance of GUM on useful parallel programs in a future paper.

4.1 Parallel Overheads

This experiment investigates the underlying costs of parallel evaluation, compared with sequential evaluation. Even if there is only one processor evaluating a program, there is still some overhead in GUM. Unless this overhead is small, we cannot hope to achieve good *absolute* speed-ups, i.e. speed-ups relative to a good sequential implementation. The overhead on a single processor can be categorised as follows.

- There is a more-or-less fixed percentage overhead on every program regardless of its use of parallelism. An example of these overheads is that GUM must test each new closure to see whether it is under evaluation already.
- There are overheads introduced by every spark site in the program, as described below.

We investigate these overheads using a single processor executing a divide-and-conquer factorial, which is a good stress-test for the second overhead, because it can be compiled for sequential execution so that the main loop does not generate any closures at all. However, in its parallel form, the compiler is obliged to insert code to build a closure for each spark site. If the program is written in the usual naive way, each thread does very little work before sparking another thread, and the overheads of parallelism are high.

The version of divide-and-conquer factorial that we use, `parfact`, has an explicit cut-off parameter: if the problem size is smaller than the cut-off then it is solved using purely sequential code; otherwise, the parallel code is used. In the next experiment the cut-off is varied to investigate how well GUM copes with various size threads. An interested reader can find the Haskell program in Appendix A.1.

We report all speedups in this paper relative to a fast sequential version of each program compiled using GHC with full optimisation. The following table compares the runtimes of parfact program with different parallel overheads on three Sparc-based platforms. The parallel runtimes are presented as a percentage increase over the sequential runtime. The table will be complete in the full paper.

Parallel Overheads

Platform	Seq. runtime	seq-par	par-worst	par-best
SparcClassic	42.7s		189%	11%
SunMP	35.9s	11%	177%	8%
Sparc 10	39.7s	11%	80%	

The **seq** column gives the runtime in seconds when the sequential version of the program is run under the conventional (sequential) runtime system.

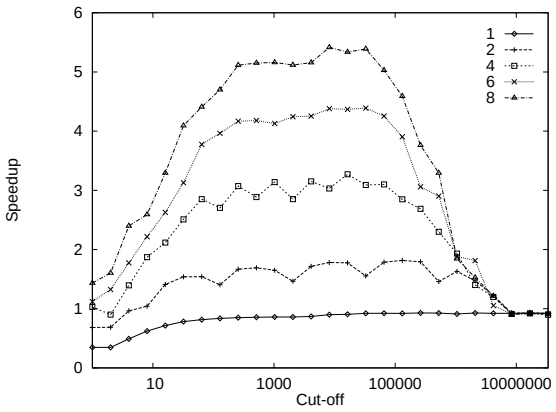

Figure 7: `parfact` speedups on Ethernetted SparcClassics

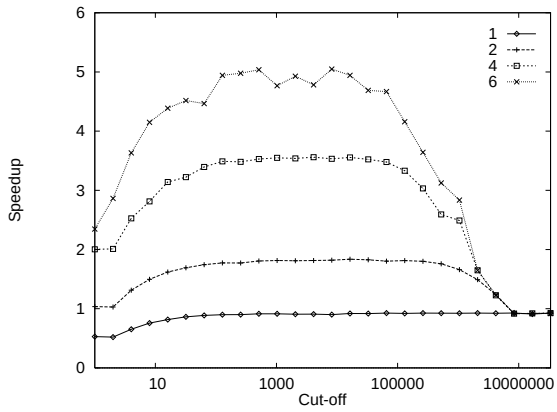

Figure 8: `parfact` speedups on SunMP

The **seq-par** column of the table gives the runtime when the sequential version of the program is run under GUM on a single processor. The increased runtime shows that the overhead imposed by GUM on all code, including sequential, is in the region of 10% for a typical parallel platform.

The **par-worst** column of the table gives the runtime when the parallel version of the program is run under GUM on a single processor, with the finest possible grain of parallelism. That is, the cut-off is set to 1, and the program is forced to create a closure to spark in every recursive call. The overheads are quite high, but it should be remembered that `parfact` is very much a worst case. In most programs that do real work, there will already be closures at most of the spark sites and the cost of the sparks will be quite low.

The **par-best** column of the table gives the runtime when the parallel version of the program is run under GUM on a single processor, with an 'good' grain of parallelism. In the next section we discover that chosing a cut-of value of 8192 produces a good thread-size for GUM on these architectures. The behaviour of `parfact` with cut-off 8192 is more typical of parallel programs in that it creates some closures in order to spark them, but also has large sections of sequential code.

The near-linear speedups compared to sequential execution reported in the next section assure us that the GUM's parallel overheads are not excessive. Indeed typical overhead that we have observed across a range of programs running on Sun Sparc machines is in the region of 35%. We intend to report the parallel overhead for a range of programs in the full paper.

4.2 Granularity Investigation

In the following experiment we investigate the minimum acceptable grain-size for two different architectures, again using `parfact`. Figure 7 shows the speedups obtained (relative to pure sequential compilation and execution) for `parfact` with different cut-off values and different numbers of processors. The 'parallel machine' in this case was a set of SparcClassic workstations (Sun 4/15), each with 24MByte RAM, and connected to a common Ethernet segment. Figure 8 shows results from the same experiments run on a Sun multiprocessor with 6 Sparc CPUs connected in a shared-memory configuration.

The speedups shown in these figures are median speedups obtained over 4 runs. The maximum speedup curves (not given in this paper) are smooth. However, these median curves are not smooth because (1) the network and the processors were lightly-loaded, but there was no way of preventing other people using them while the experiments were being run; and (2) the load distribution is sometimes poor because PEs fish for work at random, and without thread-migration one processor may end up with a number of runnable threads while another has none.

Things to note about the results shown in Figures 7 and 8:

- The graph for the SunMP, with its lower latency interconnect, is smoother than the one for the Ethernetted system. We have yet to satisfactorily explain this phenomenon, as there are a large number of variables involved, some of which, particularly usage of the systems by other jobs, are beyond our control.
- With *one* processor running the parallel system, the speedup goes from 0.5 to 0.92 (SunMP) or 0.3 to 0.92 (Ethernetted SparcClassics) as granularity is increased by varying the cut-off from 2 to infinity.
- The peak speedup achieved on the SunMP with 6 processors was 5.1, at a cut-off value of 128. For the Ethernetted SparcClassics, the peak speedup with 6 processors was 4.4, at a cut-off value of 8192. (With 8 processors, the Ethernetted SparcClassics achieved a peak speedup of 5.4 with the same cut-off value.)

  The thread size, or granularity, corresponding to a cut-off value of 8192 is about 45ms for the Ethernetted SparcClassic system. For the SunMP, the thread size corresponding to a cut-off value of 128 is about 0.6ms. Since at both these cut-off values there are still potentially thousands of parallel threads, this is a reasonable indication of the finest grain size that can be tolerated by each platform.
- For both machines, the best value of the cut-off parameter is independent of the number of processors.

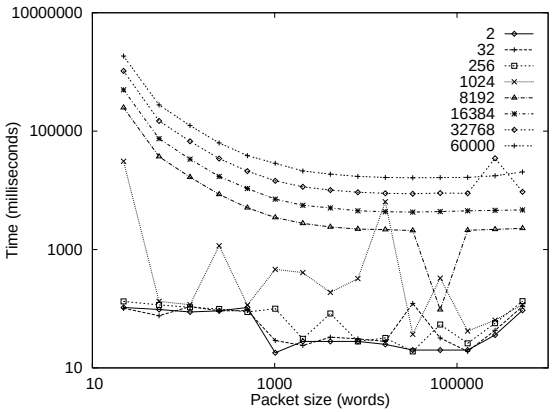

Figure 9: `bulktest` runtimes on Ethernetted SparcClassics

4.3 The effect of packet size

A program that generates a list of integers on one processor, and consumes the list (summing it) on another processor is used to investigate the optimal packet size for each architecture. The packet size is only optimal for programs that use all of the data in the packet. An interested reader can find the program, `bulktest`, in Appendix A.2.

Figure 9 shows the absolute runtimes for `bulktest` when run on a pair of SparcClassic workstations connected to the same segment of Ethernet. The x-axis shows varying packet sizes, while the multiple plots are for different list lengths, as set by the command-line argument. Both the x and y axes have log scales.

Interesting points from Figure 9:

- The time required to communicate very long lists (in excess of 8000 elements) is predictable, and reduces as the packet size increases.
- The time required to communicate short lists (less than 8000 elements) is chaotic, but nevertheless quite small; this is probably due to the random nature of the Ethernet.
- Most of the benefit of bulk fetching is achieved with packet sizes of about 4K words. Larger packet sizes improve performance slightly for this experiment, but for more realistic programs they may prove detrimental. This result is in close agreement with the PVM example program `timing`, which shows that for the machine configuration we used, most of the benefit of increasing message size is gained when messages are around 13KByte.

5 Related Work

There has been much work on parallel functional programming. Hammond [13] provides a historical overview and introduces the principal approaches that have been taken at both the language and implementation levels. This section describes the implementations that are most closely related to GUM: those based on compiled graph reduction for non-strict purely functional languages. There are many good implementations for strict languages, including Lisp derivatives such as Qlisp [11] or Mul-T [19], ML-based implementations such as Concurrent ML [32] or Caml Flight [7], and dataflow languages such as Sisal [24] and Id [16]. Indeed, in some cases, the Sisal implementation outperforms parallelising Fortran [5]. A more detailed comparison with these and other implementations will be provided in the full paper.

5.1 Shared-Memory Implementations

Shared-memory implementations have been quite successful, often showing good relative speedup for limited numbers of processors on simple programs.

One of the first successful implementations was Buckwheat [9], which ran on the Encore Multimax. This used a fairly conventional stack-based implementation of compiled graph-reduction, with a single shared heap, and a two-level task queue, which aimed to reduce memory contention. To avoid duplicating work, each thunk was locked when it was entered. This is a potential weakness of the design: locking can be costly, and consumes significant shared-memory bandwidth, generally unnecessarily. Relative speedups of 6 to 10 were achieved with 12 processors.

The $< \nu, G >$-Machine [3] was based on the sequential Chalmers G-Machine compiler, and ran on a 16-processor Sequent Symmetry. There was no stack, but instead thunks were built with enough space to hold all necessary arguments plus some local workspace for temporary variables. As with Buckwheat, a single shared heap was used, with thunks locked on entry. Garbage collection was implemented using a global stop-and-copy policy. The spark creation policy was similar to that for GUM, but only a single global spark pool was provided.

There are several obvious problems with this scheme: the global spark pool is a memory hot-spot; the lack of an explicit stack means cache locality is lost; and garbage collection requires inter-processor synchronisation, which can ultimately dominate execution costs. As a consequence, while the implementation did achieve real speedups, they were less than might have been expected: a factor of 5 to 11 on a 16-processor configuration.

Mattson observed similar problems with a similar shared-memory implementation of the STG-Machine on a 64-processor BBN Butterfly [26]. In this case the best overall performance was achieved using approximately 16 processors, and adding more processors had a negative effect on performance.

The root cause of these problems is that a shared-memory system is capable of supporting only a limited number of accesses before the bus or other inter-connect becomes saturated. Caching helps to some extent by eliminating some read accesses, but maintaining cache coherency on writes is expensive. It is thus vital to eliminate unnecessary accesses to shared memory.

The more recent GAML implementation is an attempt to address some of the shortcomings of the original $< \nu, G >$-Machine. GAML introduces the notion of *possibly shared* nodes, which are the only nodes that must be locked. It also uses a linked list of stack chunks similar to those we use in GUM. Garbage collection is done in parallel, with all processors synchronising first. Control of parallelism is by load-based inlining, where a simple measure of system load determines whether or not a task should be created. Clearly, such a strategy can lead to starvation, and should be used only on programs which are coarse-grained or which generate continual parallelism. On the Sequent Balance, GAML achieves relative speedups of between 3.3 and 5.8 for small programs [].

WYBERT [21] is based on the FAST/FCG sequential compiler, and runs on a 4-processor Motorola HYPERmodule. Rather than defining a general primitive for parallelism, as with the previous approaches, the implementation uses an explicit divide-and-conquer skeleton. While this limits the programs that can be run to those suiting a single paradigm, this paradigm does occur fairly frequently, and the regular structure of divide-and-conquer programs allows a number of optimisations to be performed.

Somewhat radically, WYBERT aims to avoid the cost of locking entirely by ensuring that shared redexes cannot arise! This is achieved by eagerly evaluating all shared data before a task is created. A secondary advantage is that a task can perform independent garbage collection since no remote processor can refer to any of its data (space leaks could presumably occur, though, since data generated by a parent cannot be collected). Each processor has a single stack of stacks (therefore only the currently executing task could migrate). Relative speedups are fairly good for the small configuration that was tested, between 2.4 and 4 on 4 processors. It is claimed that eliminating locking improves performance by up to a factor of two on 4 processors.

5.2 Distributed-Memory Implementations

There have been several Transputer-based distributed-memory implementations, and a few on other architectures.

Alfalfa was a distributed-memory implementation for the Intel iPSC, similar to, but predating Buckwheat [8]. Unfortunately, the communication overhead on this system was extremely high. Because of this, and perhaps because the implementation techniques used were less suited to distributed-memory than shared-memory, performance results were extremely disappointing: relative speedups of around 4 to 8 being achieved for 32 processors.

Like WYBERT, ZAPP [23] aims to implement only divide-and-conquer parallelism, using an explicit fork-and-join skeleton. Once generated, tasks can either be executed on the processor that generated them or stolen by a neighbouring processor. There is no task migration, so the program retains a high degree of locality. A simple bulk-fetching strategy is implemented, with all necessary graph being exported when the task that needs it is exported. Performance results on the transputer were impressive for the few programs that were tried, with relative speedups generally improving as the problem size increased, up to 39.9 on 40 transputers for naive Fibonacci. The improvement with problem size suggests either that the dispersal algorithm may not work effectively for small problems, or that there is significant overhead to exporting tasks.

The HDG-Machine [18] uses a packet-based approach to memory allocation that is similar to that of the $< \nu, G >$-Machine, but with a distributed weighted reference-counting garbage collection scheme [22]. Task distribution is similar to ZAPP. Only incremental fetching strategies were tested with this scheme, though presumably a bulk fetching strategy would also be possible. Relative speedups for naive Fibonacci were of 3.6 on 4 transputers.

Concurrent Clean runs on transputers and networks of Macintoshes [28]. Like GUM, it is stack-based, and uses tables of "in-pointers" to allow independent local garbage collection. A bulk graph-fetching algorithm is implemented, but in contrast to GUM, there is no limit on the size of graph that will be sent, and graph is reduced to normal form before it is transferred. In contrast to the GUM fishing strategy, tasks are statically allocated to processors by means of annotations. Relative speedups of 8.2 to 14.8 are reported for simple benchmarks on a 16-processor Transputer system [17].

5.3 GRIP

GUM's design is a development and simplification of our earlier work on the GRIP multiprocessor [30].

GRIP's memory was divided into fast unshared memory that was local to a PE, with separate banks of globally addressed memory that could be accessed through a fast packet-switched network. Closures were fetched from global memory singly on demand rather than using GUM-style bulk fetching. While we no longer have two kinds of memory, we have retained the two-level separation of local and global heap that permits independent garbage collection. In addition to the advantages for garbage collection, this has the secondary benefit of identifying heap that is purely local, and can thus be held in faster unshared memory on a shared-memory machine.

On GRIP, unlike GUM, the spark pool was also divided in two. Each PE maintained a pool of local sparks, with special hardware maintaining a distributed global spark pool, which acted as a buffer between producers and consumers of sparks. Although, this scheme had the advantage that PEs never processed FISH messages unnecessarily, because local memory was entirely unshared a negative consequence was that sparks could only be exported after *all* graph reachable from that graph was also exported. This could lead to significant performance losses. The work-stealing scheme used in GUM avoids this problem.

Relative speedups on GRIP were generally good if the algorithm was reasonably parallel. For example, parallel ray tracing achieved speedups of around 14 on 16 processors, and the Fibonacci function achieved a speedup of around 17 on 20 processors.

6 Discussion and Further Work

We have described a portable, parallel implementation of Haskell, built on the PVM communications harness. GUM is currently on public α-release with version 0.26 of the Glasgow Haskell compiler.

It is quite ambitious to target such a variety of architectures, and it is not obvious that a single architectural model will suffice for all machines, even if we start from such a high-level basis as parallel Haskell. We do however believe that it is easier and more efficient to map a message-based protocol onto a shared-memory machine than to map a shared-memory protocol onto a distributed-memory machine. While we have initially targeted PVM because of its wide availability this is not a fixed decision and our implementation could be easily re-targeted to other message-passing libraries such as MPI. We also expect that considerable speedups could be obtained by using machine-specific communication primitives, particularly on shared-memory machines.

The performance figures given here are indicative rather than conclusive. They show that we have not fallen into the trap of building a parallel system whose performance is fundamentally slower by a large factor than the best uniprocessor compilation technology. They do not, however, say much about whether real programs can readily be run with useful speedups. Indeed, we believe that considerable work is required to tune the existing system, to further develop the system, and to support possible extensions to parallel Haskell.

The two most pressing things to tune are the maximum packet size for common architectures, and the load management strategy. In the medium term, the addition of multiple-packet messages and distributed garbage collection for cyclic graphs would increase the number of programs that could be run, and thread migration would improve the ability of the system to cope with arbitrarily partitioned programs. In the longer term, we plan to investigate adding speculative evaluation and support for explicit concurrent processes [31]. We hope that the public availability of the system will encourage others to join us in these developments.

References

[1] Arvind and Iannucci RA, "Two Fundamental Issues in Multiprocessing", *Proc DFVLR Conference on Parallel Processing in Science and Engineering*, Bonn-Bad Godesberg (June 1987).

[2] Bevan DI, "Distributed Garbage Collection using Reference Counting", *Proc PARLE*, deBakker JW, Nijman L and Treleaven PC (eds), Eindhoven, Netherlands (June 1987).

[3] Augustsson L, and Johnsson T, "Parallel Graph Reduction with the $< \nu, G >$-Machine", *Proc. FPCA '89*, London, UK, (1989), pp. 202–213.

[4] Burton FW, and Rayward-Smith VJ, "Worst Case Scheduling for Parallel Functional Programming", *Journal of Functional Programming*, **4(1)**, (January 1994), pp. 65–75.

[5] Cann DC, "Retire Fortran? A Debate Rekindled", *CACM*, **35(8)**, (August 1992), pp. 81–89.

[6] Chakravarty MMT, "A Self-Scheduling, Non-Blocking Parallel Abstract Machine for Non- Strict Functional Languages", *Proc 6th. Intl. Workshop on Implementation of Functional Languages*, Glauert JRW (ed.), University of East Anglia, (September 1994).

[7] Foisy C, and Chailloux E, "Caml Flight: a Portable SPMD Extension of ML for Distributed Memory Multiprocessors", *Proc. High Performance Functional Computing '95*, Denver, Colorado, (April 1995), pp. 83–96.

[8] Goldberg B, and Hudak P, "Alfalfa: Distributed Graph Reduction on a Hypercube Multiprocessor", *Proc. Workshop on Graph Reduction*, Fasel RMKJF (ed.), Santa Fé, NM, Springer Verlag LNCS 279, (1986), pp. 94–113.

[9] Goldberg BF, "Buckwheat: Graph Reduction on a Shared Memory Multiprocessor", *Proc. ACM Conf. on Lisp and Functional Programming*, Snowbird, Utah, (1988), ppp. 40–51.

[10] Goldberg B, *Multiprocessor Execution of Functional Programs*, PhD thesis, Department of Computer Science, Yale University, (April 1988).

[11] Goldman R, and Gabriel RP, "Qlisp: Parallel Processing in Lisp", *IEEE Software*, pp. 51–59, (1989).

[12] Hammond K, and Peyton Jones SL, "Profiling Scheduling Strategies on the GRIP Multiprocessor", *Proc 4th. Intl. Workshop on Parallel Implementation of Functional Languages*, Kuchen H (ed.), Aachen University, (September 1992).

[13] Hammond K, "Parallel Functional Programming: an Introduction", *Proc. PASCO '94*, Linz, Austria, World Scientific, (September 1994), pp. 181–193.

[14] Hammond K, Loidl H-W, and Partridge AS, "Visualising Granularity in Parallel Programs: A Graphical Winnowing System for Haskell", *Proc HPFC'95 — High Performance Functional Computing*, Denver, Colorado, (April 1995).

[15] Herlihy M, and Liskov B, "A value transmission method for abstract data types", *ACM TOPLAS* **4**(4), (1982), pp. 527–551

[16] Hicks J, Chiou D, Ang BS, and Arvind, "Performance Studies of Id on the Monsoon Dataflow System", *Journal of Parallel and Distributed Computing*, **18**, (1993), pp. 273–300.

[17] Kesseler M, "Reducing Graph Copying Costs – Time to Wrap it up", *Proc. PASCO '94 — First Intl. Symposium on Parallel Symbolic Computation*, Hagenberg/Linz, Austria, World Scientific, (September 1994), pp. 244–253.

[18] Kingdon H, Lester D, and Burn GL, "The HDG-Machine: a Highly Distributed Graph Reducer for a Transputer Network", *The Computer Journal*, **34(4)**, (April 1991), pp. 290–302.

[19] Kranz DA, Halstead RH, and Mohr E, "Mul-T: a High-Performance Parallel Lisp", *Proc. PLDI '89*, Portland, Oregon, (1989), pp. 81–90.

[20] Kranz DA, Johnson K, Agarwal A, Kabiatowicz J, and Lim B-H, "Integrating Message-Passing and Shared-Memory: Early Experience", *Proc. 4th ACM Symp. on Principles and Practice of Parallel Programming (PPOPP)*, San Diego, CA, (May 1993), pp. 54–63.

[21] Langendoen K, "Graph Reduction on Shared Memory Multiprocessors", PhD Thesis, University of Amsterdam, 1993.

[22] Lester D "An Efficient Distributed Garbage Collection Algorithm", *Proc. PARLE '89*, LNCS 365, Springer Verlag, (June 1989).

[23] McBurney DL, and Sleep MR, "Transputer Based Experiments with the ZAPP Architecture", *Proc. PARLE '87*, LNCS 258/259, Springer Verlag, (1987), pp. 242–259.

[24] McGraw J, Skedzielewski S, Allan S, Odehoeft R, Glauert JRW, Kirkham C, Noyce W, and Thomas R, "SISAL: Streams and Iteration in a Single-Assignment Language: Reference Manual Version 1.2", Manual M-146, Rev. 1, Lawrence Livermore National Laboratory, (March 1985).

[25] Maranget L, "GAML: a Parallel Implementation of Lazy ML" Proc. FPCA '91, Springer Verlag LNCS 523, p.102–123, (1991).

[26] Mattson JS, *An Effective Speculative Evaluation Technique for Parallel Supercombinator Graph Reduction*, PhD thesis, Dept. of Computer Science and Engineering, University of California, San Diego, (1993).

[27] Newcomer JM, "Efficient Binary I/O of IDL objects", *ACM SIGPLAN Notices* **22**(11), (November 1987), pp. 35–43.

[28] Nöcker EGJMH, Smetsers JEW, van Eekelen MCJD and Plasmeijer MJ, "Concurrent Clean", *Proc. PARLE '91*, Springer Verlag LNCS 505/506, pp. 202–220, (1991).

[29] Oak Ridge National Laboratory, University of Tennessee, "Parallel Virtual Machine Reference Manual, Version 3.2", (August 1993).

[30] Peyton Jones SL, Clack C, Salkild J, "High-performance parallel graph reduction", *Proc PARLE '89*, Springer Verlag LNCS 365 (June 1989).

[31] Peyton Jones SL, Gordon AD, and Finne SO, "Concurrent Haskell", To appear in *Proc. ACM Symposium on Principles of Programming Languages*, St Petersburg Beach, Florida, (January 1996).

[32] Reppy JH, "CML: a Higher-Order Concurrent Language", *Proc. PLDI '91*, Toronto, Canada, (June 1991), pp. 293–305.

[33] Sansom PM, "Time Profiling in a Lazy Functional Language", *Proc. 1993 Glasgow Workshop on Functional Programming*, Ayr, Springer-Verlag WICS, (July 1993), pp. 227–239.

[34] Toyn I, and Dix AJ, "Efficient Binary Transfer of Pointer Structures", *Software Practice and Experience* **24**(11), (November 1994), pp. 1001–1023.

[35] Watson P and Watson I, "An Efficient Garbage Collection Scheme for Parallel Computer Architectures", *Proc PARLE*, deBakker JW, Nijman L and Treleaven PC (eds), Eindhoven, Netherlands (June 1987).

7 Appendix A: Parallel Haskell Programs

This appendix contains the parallel Haskell programs used to investigate aspects of GUM's performance, as described in Section 4.

7.1 Appendix A.1: Parallel Factorial

The following is the parallel `parfact` program. To obtain the sequential version, we simply replaced the definition of the `parfact` function by: `parfact x c = pf 1 x`.

```
module Main(main) where

import Parallel

pfc :: Int -> Int -> Int -> Int
pfc x y c
   | y - x > c = f1 `par`
                 (f2 `seq` (f1+f2))
   | x == y    = x
   | otherwise = pf x m + pf (m+1) y
   where
      m  = (x+y) `div` 2
      f1 = pfc x m c
      f2 = pfc (m+1) y c

pf :: Int -> Int -> Int
pf x y
   | x < y     = pf x m + pf (m+1) y
   | otherwise = x
   where
      m  = (x+y) `div` 2

parfact x c = pfc 1 x c

main
 = getArgs exit ( \[a1, a2] ->
    let x = fst (head (readDec a1))
        c = fst (head (readDec a2))
    in
        appendChan stdout
                   (show (parfact x c))
                   exit done
       )
```

7.2 Appendix A.2: Bulk Test

```
module Main(main) where

import Parallel
```

```
bulktest x
   = sxs `par` ((force xs) `seq` sxs)
     where
        xs = take x (repeat (1::Int))
        sxs = sum xs

-- force returns only when the argument
-- list has been completely evaluated.

force :: [Int] -> ()
force [] = ()
force (x:xs) = x `seq` (force xs)

main
 = getArgs exit ( \[a1] ->
    let x = fst (head (readDec a1)) in
     appendChan stdout
                (show (bulktest x))
                exit done
   )
```