GUM: a portable parallel implementation of Haskell

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

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

September 4, 1995

Abstract

GUM is a portable, parallel implementation of the Haskell functional language which has been publicly released with version 0.26 of the Glasgow Haskell Compiler (GHC). GUM is message-based, and portability is facilitated by using the PVM communications-harness available on most multi-processors, including shared-memory and distributed-memory machines. For example GUM is available by FTP for a Sun SPARCserver multiprocessor and for a networks of Sun SPARC workstations.

High message-latency in 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 PEs during execution.

*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 Introduction

GUM (Graph reduction for a Unified Machine model) is a portable, parallel, publicly-released implementation of the purely-functional programming language Haskell. So far as we know, it is the first publicly-released parallel implementation of any functional language, except possibly MultiLisp, despite hundreds of papers, dozens of paper designs, and a handful of real single-site implementations. We believe that this is partly because of the diversity of parallel machine architectures, but also because the implementation task 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 presenting preliminary performance measurements.

GUM has the following features:

- *GUM is portable.* It is message based, and uses PVM [17], 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 (hereinafter called PEs) connected by some kind of network that is accessed through PVM.

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

- *GUM delivers 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 garbage-collect its local heap independently of any other PE, a property we found to be crucial on the GRIP multiprocessor [20].

- *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 2.2.2). 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 [2] — 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 2.3.2). 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 find out.

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 Sun's symmetric multiprocessor SPARCserver.

The remainder of this paper is structured as follows. Section 2 describes how the GUM runtime 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

This section describes GUM's design and implementation. A lot of the description centres around the different types of message that GUM uses; for reference, Appendix A lists the fields of each type of message.

[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.

2.1 Initialisation and Termination

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 closure called *Main.main*).

The program terminates when either the main thread completes, or encounters an error. 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.

2.2 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*. The PE executes the following scheduling loop until it receives a FINISH message.

Main Scheduler:

1. Perform local garbage collection, if necessary.
2. Process any incoming messages from other PEs.
3. If there are runnable threads, run one of them.
4. Otherwise look for work, as described in Section 2.2.2.

The thread scheduler is fair: runnable threads are executed in following a round-robin policy. Each thread is run until its time-slice expires, it completes, space is exhausted or the thread blocks — either on a black hole (Section 2.2.3) or accessing remote data (Section 2.3.2). A fair scheduler facilitates concurrent and speculative execution, at the cost of increasing both space usage and overall run-time [4].

2.2.1 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* (aka 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`.

Sparking a thunk is a relatively cheap operation, consisting only of adding a pointer to the thunk to the PE's *spark pool*[2]. 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.

2.2.2 Finding work

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 useless, if the thunk it refers to has by now been overwritten with its value, in which case it is discarded.

[2] There is a caveat here — see Section 4.1 for details

If the PE finds a useful spark, it turns it into a thread by allocating a fresh TSO and SO[3], and starts executing it.

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 FISH messages are created, they are sent at random to some other PE. If the recipient has no useful spark, or potential task, it increases the age of the FISH, and reissues the FISH to another PE, again chosen at random. There is a limit to 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 reissuing it. 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 issues 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. 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.

2.2.3 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:

[3] 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

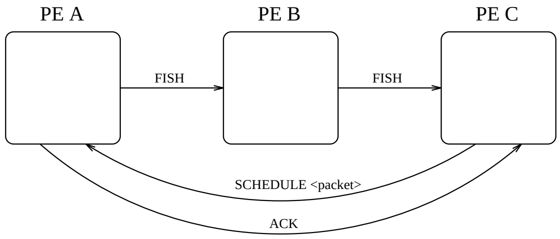

Figure 1: Fish/Schedule/Ack Sequence

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 (its *blocking queue*), and calls the scheduler.
3. Finally, when a thread completes the evaluation of a thunk, it overwrites it 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.

2.3 Memory Management

Parallel graph reduction proceeds on a shared program/data graph, and 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.

As mentioned earlier, GUM is based on the Glasgow Haskell Compiler, and most execution is carried out in precisely the same way as on a uniprocessor. In particular, new heap closures are allocated from a contiguous chunk of free space, and the heap-closure addresses manipulated by the compiled code are simply one-word pointers within the local heap.

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).

2.3.1 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, and compacting garbage collection is a crucial component of our efficient compilation technology.

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[4]. 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. (Appendix B describes how an unused identifier is found, and how the GIT is represented.) 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 → GA table*, and checking it before globalising a heap closure. Naturally, the LA → 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 → 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 → LA table; then, when it needs (B,34) again it checks the GA → 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 → 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 → 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; Appendix B elaborates.

2.3.2 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

[4] 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.

can the local identifier be re-used. Again following standard practice, [14] we use *weighted reference counting* to recover local identifiers, and hence the closures they identify [3, 25].

We augment both the GIT and the GA $\rightarrow$ LA table to deliver 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 garbage collection, 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 other PEs or messages refer to this global address.) All the other entries must be treated as roots for local garbage collection.
2. The local addresses in a PE's GA $\rightarrow$ LA table are treated as roots for local garbage collection, preserving local copies of global closures in the hope that they will prove useful in the future. Potentially this step could be omitted if the PE is short of space.

   After local garbage collection is complete, the GA $\rightarrow$ LA table is scanned. Any entries whose local closure has 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).

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.

If the weight in a GA to be sent is 1 it can no longer be split, and 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.

Distributing Data

Global references are handled by special *Fetch-Me* closures (Figure 2 shows a typical Fetch-Me closure). When a thread enters a Fetch-Me:

- The Fetch-Me closure is converted into a special *Fetch-Me blocking queue*, and the thread is enqueued, i.e. blocked.
- The Fetch-Me blocking queue is globalised, i.e. given a new local GA.
- A FETCH message is sent to the PE that owns the GA in the Fetch-Me.
- 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.

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

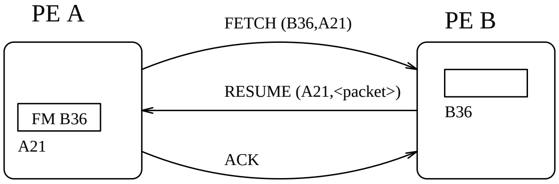

Figure 2: Fetch/Resume/Ack Sequence

RESUME arrives, the originating PE unpacks the graph, redirects the Fetch-Me to point to the root of this graph, and restarts any threads that were blocked on global closures that were transmitted in the packet. Having done this, an ACK message is returned to the PE that sent the RESUME (the following section explains why).

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 FETCHes that need to be sent. This section discusses the algorithms and heuristics that are used to pack graph into GUM packets.

The Design of the Packing Algorithm

Packing arbitrary graph is a non-trivial problem. The basic problem of transferring arbitrary pointer structures, while preserving any sharing and cycles, has already received some attention in a non-functional context [24, 16, 11]. In GUM the problem of preserving the integrity of the graph is exacerbated because the graph on both the sender and the recipient may be changing as a result of normal reduction occurring at the same time that it is being shipped.

There are several possible choices of how much graph to pack [10]: we can either pack a single closure, all reachable graph, or some reachable sub-graph. Since shipping a single closure is likely to be very expensive over a high-latency network, and the graph that is reachable from a closure may easily exceed implementation limits on packet sizes or memory, we have opted to pack only some of the reachable graph, up to a fixed size limit. The current default size is 1K words, other sizes can be selected at run time.

In order to maximise the likelihood that the graph we transmit is actually needed, we pack and unpack graph breadth-first. Thus closures are normally shipped with at least the outer-level of their arguments. Fetch-Mes are used to refer to graph that could not be packed directly. Good strictness analysis could obviously help improve this heuristic, by indicating the extent to which thunks were known to need their arguments.

Packing and unpacking graph is expensive: there are two ways in which we hope to recoup some of this cost. Firstly, by reducing the number of packets that need to be sent, we can use the network bandwidth to reduce the overall latency cost of transmitting the data that the program requires, at the cost of slightly increasing the latency of the each packet. Secondly, if fewer packets are sent, the total context-switch and packet times will be reduced. The former effect is likely to be most significant for medium- to high-latency networks; while the latter applies to all latencies. It remains to be seen whether bulk graph packing is sensible in practice for low latency machines, though initial simulation results do suggest that bulk packing is at least as efficient as single closure packing, even for very low latencies.

Shipping packets of graph introduces complexity into the RTS. Instead of a single closure being copied, a subgraph is being copied, and all of the links from the subgraph back to the original graph must be maintained. In a degenerate case the same closures may be shipped to and fro between PEs. When a packet of graph arrives, the recipient interrogates the GA $\rightarrow$ LA table for each closure to determine whether there is already a local copy of that closure. If so, care must be taken to select the closure requiring the least amount of communication.

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

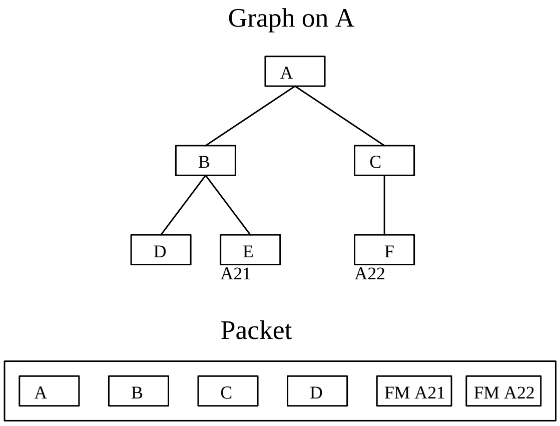

Figure 3: A Full Packet

reachable graph has been packed, or the packet is full. To detect when a packet is full, the packing routine keeps track of how full the packet is, and how many closures remain to be packed. Once the packet is full, any outstanding closures are globalised and packed as Fetch-Mes. Figure 3 shows an example of this.

It has been observed that packing a pointer structure is similar to the graph copying performed by copying garbage collectors [24]. Indeed the packing code performs many of the operations undertaken by the garbage collector to "improve" the graph. For example indirections are short-circuited. A minor beneficial effect of packet-based transfer is that it improves locality by co-locating "nearby" closures.

The packet that the graph is packed into has a short 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 below. 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. To help avoid space leaks, the GA $\rightarrow$ LA table is also used during unpacking to ensure that each PE only has one copy of normal form closures.

As each closure is packed it is made global, if it is not already, and the weighted reference count is divided between the local and packed copy. A few closure types are packed specially:

1. "Black holes", i.e. closures under evaluation, are packed as a Fetch-Me to the black hole.
2. Permanent local closures (PLCs), i.e. globally-shared top-level constants, reside on each PE. The PLC address is simply encoded in the GA/tag.
3. Thunks are converted into *revertable black hole* closures for reasons explained below.
4. If a closure occurs more than once in the graph to be packed, then the second and subsequent occurrences are packed as empty closures, with a special tag in the GA/tag representing an offset into the packet. The offset points to the location of the first occurrence of the closure in the packet.
5. "Primitive arrays" (monolithic data structures) are always packed along with their parent closures. A primitive array cannot be replaced by a Fetch-Me because, for efficiency, some operations index directly off the primitive array without first ensuring that the closure actually is a primitive array.

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

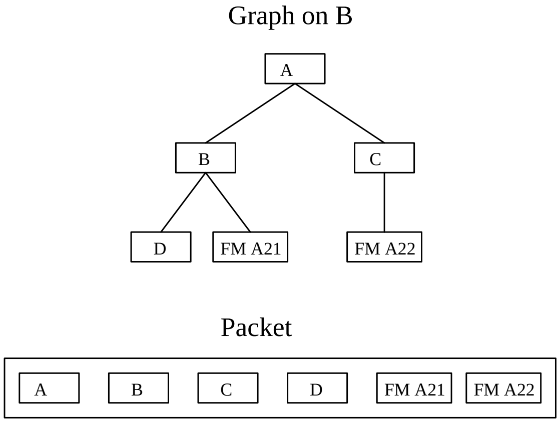

Figure 4: Unpacked Packet

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.

Thunks

Non-normal form closures, or thunks, are treated specially to ensure that exactly one copy of the thunk exists. When shipping a thunk between PEs we want to convert the closure on the source PE into a Fetch-me to the new location of the thunk on the destination PE. However, there are two problems with this:

1. the new global address of each thunk in a packet isn't known until the packet has arrived at its destination; and
2. since there is only one copy of the thunk, we can't destroy the local copy until we know that the shipped copy has been successfully unpacked. Unpacking may be impossible if the remote PE runs out of heap, for example.

These problems are resolved by using revertable black holes and by sending ACK messages to acknowledge every packet that contains one or more thunks. A revertable black hole acts like a normal black hole, except that it also records the original value of the closure. Should a subsequent thread enter the revertable black hole, i.e. demand the thunk that has just been shipped away, it becomes blocked. By turning a thunk into a revertable black hole, we can ensure that it can be restored in the unlikely event that the packet is rejected by the destination. One reason for rejecting the packet would be the onset of global garbage collection, which is not currently implemented. If the packet is rejected a NACK message is sent, identifying the thunks that couldn't be unpacked.

For example, if a thunk with GA A42 is packed on PE A for shipment to PE B, it is converted into a revertable black hole closure on A. When PE B unpacks the thunk, it allocates it a new GA, e.g. B59. After unpacking the entire packet B sends an ACK message to A containing a pair of GAs for every thunk shipped in the packet, e.g. (A42, B59). PE A overwrites the revertable black hole identified by A42 by a Fetch-Me to B59: the thunk's new location, and reawakens any blocked threads. Typically these threads are very short-lived: they enter the Fetch-me, send a FETCH, and block awaiting the response.

We have observed programs where a thunk is repeatedly copied from PE to PE, generating a chain of Fetch-Mes. In the worst case a FETCH message might conceivably chase a thunk endlessly round a cycle of PEs. This behaviour has not been observed.

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 RTS, some of the profiling tools that are provided with sequential Haskell still yield information that can be used to tune parallel program. In addition, we have also produced special tools that give per-PE or per-thread activity profiles over time.

3.1 Clocks

Choosing the time base against which to record measurements in a distributed, multiprogrammed environment is hard. Even maintaining accurate time synchronisation between distributed PEs is expensive and complex [13]. The time behaviour of a program running under GUM is further complicated because of the use of PVM. Each PE task is typically a Unix process, and at the mercy of the Unix process scheduler. In some configurations, such as a network of workstations, there is also competing network traffic that affects overall performance.

Elapsed-time, including initialisation and termination time is the most stringent measure of performance. For small programs, initialisation and termination time is significant, and it is useful to record them. A problem with elapsed-time profiles is that they record activity even when the program has been descheduled by the Unix process scheduler and some other process is running, artificially elongating the profile on the time axis, and inflating the average parallelism figure. An alternative is to use *virtual* or user time, i.e. to record only the time when the program is actually running. This avoids the elongation effect but exaggerates performance by ignoring real-time communication costs, or the effect of executing multiple PE tasks on a single physical processor.

Our solution is a cheap compromise. To mitigate the effects of the process scheduler and competing communication traffic, we recommend that performance measurement is carried out on a machine that is running *only* the parallel Haskell program being measured. Initialisation time is reported separately and either virtual time or real elapsed time is used for different profiles, as appropriate. The profiles on different PEs are synchronised by recording the value of the real-time clock at a known point during initialisation, and using this time to correct the elapsed clock times reported by different PE tasks. Many networks guarantee that the real-time clocks on the PEs differ by only a small amount.

3.2 Sequential Tools

GC Statistics

The garbage collection statistics reported by the sequential RTS are useful in GUM. For each PE the residency can be plotted over time. GUM garbage collection statistics also report both elapsed and virtual time for initialisation, mutation and garbage collection for each PE. Initialisation records the time between the PE task starting and the start of Mutation. Mutation time is the time from initialisation to termination. It includes the time spent performing reduction, communicating or idle, but excludes time spent garbage collecting.

A heap profile generated when running a linear equation solver over four SUN 4 SPARC processors connected by NFS is given below. Processor numbers (c0001, 80001 etc) refer to PVM task-ids. The ratio between elapsed and user time shows how much of the elapsed time the PE task was descheduled.

```
Processor c0001 shutting down, 260 Threads run

133,383,480 bytes allocated in the heap
71 garbage collections  (0 major, 71 minor)

  INIT  time    0.05s  (  2.10s elapsed)
  MUT   time  110.43s  (162.46s elapsed)
  GC    time    1.78s  (  2.50s elapsed)
  Total time  112.26s  (167.06s elapsed)

  %GC time       1.6%  (1.5% elapsed)

  Alloc rate    1,207,308 bytes per MUT second

  Productivity  98.4% of total user,
                66.1% of total elapsed

Processor 80001 shutting down, 2201 Threads run

 58,968,324 bytes allocated in the heap
     72,508 bytes max residency (1.7%, 1 sample(s))
31 garbage collections  (1 major, 30 minor)

  INIT  time    0.04s  (  1.62s elapsed)
  MUT   time   66.77s  (162.97s elapsed)
  GC    time    1.17s  (  2.12s elapsed)
  Total time   67.98s  (166.71s elapsed)

  %GC time       1.7%  (1.3% elapsed)
```

```
  Alloc rate    882,627 bytes per MUT second

  Productivity  98.2% of total user,
                40.1% of total elapsed

Processor 100001 shutting down, 2989 Threads run

106,720,188 bytes allocated in the heap
57 garbage collections  (0 major, 57 minor)

  INIT  time    0.11s  (  2.08s elapsed)
  MUT   time   72.71s  (164.21s elapsed)
  GC    time    0.60s  (  0.99s elapsed)
  Total time   73.42s  (167.28s elapsed)

  %GC time       0.8%  (0.6% elapsed)

  Alloc rate    1,465,534 bytes per MUT second

  Productivity  99.0% of total user,
                43.5% of total elapsed

Processor 40003 shutting down, 3194 Threads run

 36,266,232 bytes allocated in the heap
18 garbage collections  (0 major, 18 minor)

  INIT  time    0.12s  (  2.16s elapsed)
  MUT   time   45.72s  (164.26s elapsed)
  GC    time    0.47s  (  1.00s elapsed)
  Total time   46.31s  (167.42s elapsed)

  %GC time       1.0%  (0.6% elapsed)

  Alloc rate    791,148 bytes per MUT second

  Productivity  98.7% of total user,
                27.3% of total elapsed
```

Cost Centre Profiling

If the programmer is interested only in the ratios of time spent in individual functions, and the space consumed by those functions, then it is easy to use sequential cost-centre based profiling to provide that information [23]. More sophisticated profiling information, such as space usage over time, needs to be adapted to the parallel environment.

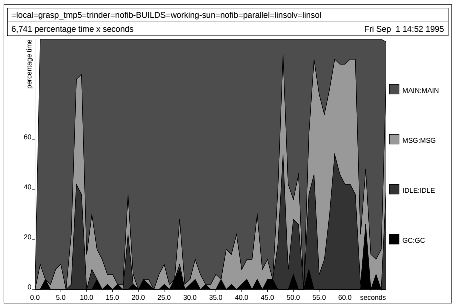

Figure 5: PE 0 (100001) Activity Profile

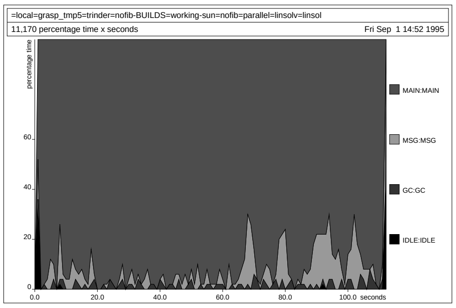

Figure 6: PE 1 (c0001) Activity Profile

3.3 PE Activity Profiles

GUM uses cost-centre profiling internally to record the activity of each PE during reduction. PE activity profiling is performed in virtual time because the area under a PE activity graph represents the percentage time the PE spent on each activity. Any elongation of the time axis caused by using elapsed-time would distort the percentage.

PE activity profiles for two of the PEs in the linear equation solver run described above are given in Figures 5 and 6. Profiles are generated for each PE, and have 4 cost-centres:

- **Main** — time spent performing reduction.
- **Msg** — time spent communicating.
- **GC** — time spent garbage collecting.
- **Idle** — time spent with no reduction, mes-

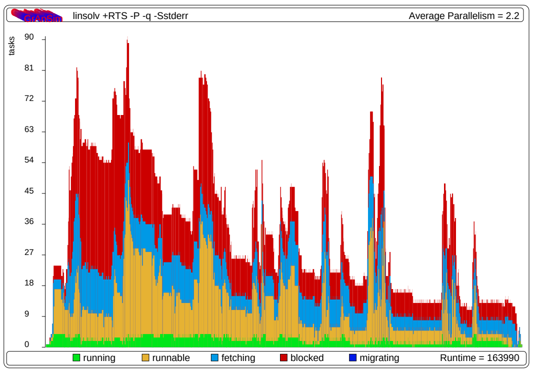

Figure 7: Thread Activity Profile

sages or garbage collection to perform.

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.

3.4 Thread Activity Profiles

An alternative style of profiling available from GUM is provided by the visualisation tools originally produced for the GranSim simulator. GranSim was constructed to investigate aspects of parallel graph reduction [9], and it can be configured to simulate the running of programs compiled for GUM on most parallel machines.

When a program is run under GranSim, a file is generated containing records that record significant events and when they occurred. Examples of significant events recorded by GranSim include when messages are sent and when threads start, block, or terminate. After a simulation run, the event file is processed to produce various kinds of information, such as the average thread length, or profiles of thread activity against time.

A runtime option in GUM programs permits the generation of a subset of the GranSim event records for each PE. The records for all PEs involved in a run can be merged and then processed by the standard GranSim tools to generate a thread activity profile. For example, the thread activity profile for the linsolv run is given in Figure 7. The number of running, runnable, fetching and blocked threads is plotted against time.

These profiles show elapsed time, and so can be used to report actual performance of GUM programs. Elapsed times are increased if the machine is running other jobs, so for performance evaluation it is usually best to make sure that the GUM run is the sole job on the machine.

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. We plan to report the performance of GUM on useful parallel programs in a future paper.

4.1 Divide-and-conquer factorial

This experiment is designed to test the ability of GUM to cope with fine-grain tasks, and to find the minimum acceptable grain-size for two different architectures. It also gives some idea of how GUM can perform with an 'ideal' parallel program.

There are two kinds of parallelism overhead incurred in GUM:

- The parallel runtime system imposes a more-or-less fixed percentage overhead on every program regardless of its use of parallelism; and
- There are overheads introduced by every spark site in the program.

Divide-and-conquer factorial is a good 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, when it is written and compiled for parallel execution, 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 will be quite 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. By varying the cut-off parameter we get some idea of how well GUM copes with various size threads.

```
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
       )
```

Note that to prevent the compiler from making the entire program into a CAF we have arranged to read the argument to parfact and the cut-off parameter from the command line. In all cases, the argument supplied on the command line was 8399608; the cut-off parameter was varied.

We report all speedups in this paper relative to a fast sequential version of each program compiled using GHC with full optimisation. To obtain the sequential version of `parfact` we simply replaced the definition of `parfact` in the above code by:

```
parfact x c = pf 1 x
```

The following table compares the run time of this program when run under different conditions on three different Sparc-based platforms.

Platform	**seq**	**seq-par**	**par**
SparcClassic	42.7	+47%	+95%
SunMP	35.9	+10%	+70%
Sparc 10	39.7	+11%	+62%

The columns of this table are:

- **seq**: gives the runtime in seconds of the sequential version of the program when compiled with the full optimising sequential compiler.
- **seq-par**: gives the percentage increase over the **seq** runtime when the sequential version of the program is compiled for parallel execution but only run on a single processor.
- **par**: gives the percentage increase over the **seq-par** runtime when the parallel version of the program is compiled for parallel execution and run on a single processor with a cut-off value of 1.

The **seq-par** column of the table shows that the overhead imposed by the runtime system on all code, including sequential, varies by a surprising amount considering that all three machines are based on the same architecture; we suspect that this is due to the different cache sizes in the machines. The overhead is less than about 50% in the worst-case, and is around 10% for a typical parallel platform.

The **par** column figures show the cost of the extra closure creation caused by the spark sites. These figures are quite high, but it should be

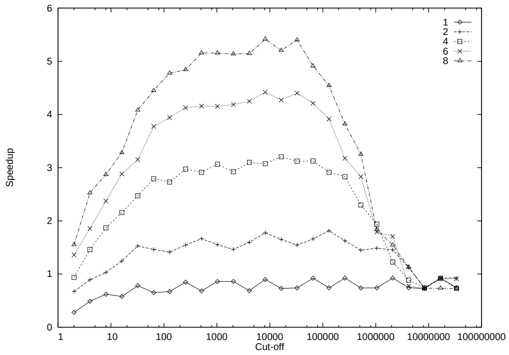

Figure 8: `parfact` speedups on Ethernetted SparcClassics

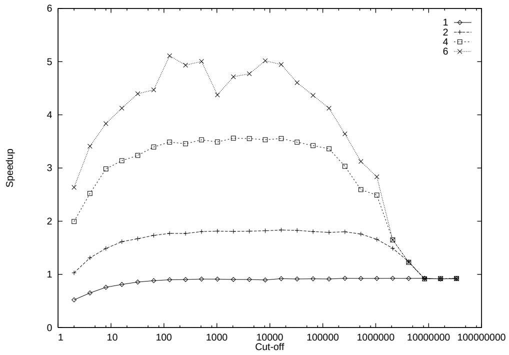

Figure 9: `parfact` speedups on SunMP

remembered that this is a limiting 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.

Figure 8 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 9 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 average speedups obtained over 4 runs. There are two factors which may cause the runtime to vary from one run to another even with the same parameters.

- 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.
- There is a degree of chaos in the results, since a single change in the placement of a spark at runtime can affect the overall runtime. This is most significant when there are few threads (that is, when the cut-off value is high) because the current system does not permit migration of running threads: one processor may end up with a number of runnable threads, while another has none. (We have already verified this phenomenon using the GranSim simulator, and plan to add thread migration to the GUM system.)

Things to note about the results shown in Figures 8 and 9:

- 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 the cut-off is varied 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 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.

4.2 Investigation of load distribution

This experiment was designed to investigate GUM's ability to distribute threads over the machine. The following program, `loadtest`, creates 50 equal-sized parallel threads (the size and number of threads are specified by command-line arguments):

```
module Main(main) where

import Parallel

nfib :: Int -> Int
nfib n | n <= 1 = 1
       | otherwise = n1 + n2 + 1
         where
            n1 = nfib (n-1)
            n2 = nfib (n-2)

parmap :: (a->b) -> [a] -> [b]
parmap f [] = []
parmap f (x:xs)
 = fxs `par` (fx `seq` (fx:fxs))
   where fx = f x; fxs = parmap f xs

main
 = getArgs exit (\[a1,a2] ->
    let
     x = fst (head (readDec a1))
     processes = fst (head (readDec a2))
     ts n = parmap
            (\ten -> sum [nfib z |
                     z <- [(x-ten)..x]])
            (take n (repeat 10))
     loadtest = sum (ts processes)
    in
     appendChan stdout (show loadtest)
                exit done
    )
```

The sequential version of this program is obtained by replacing the call to `parmap` by a call to `map`.

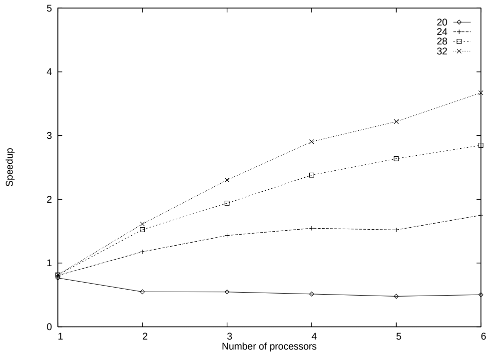

Figure 10: `loadtest` speedups on Ethernetted SparcClassics

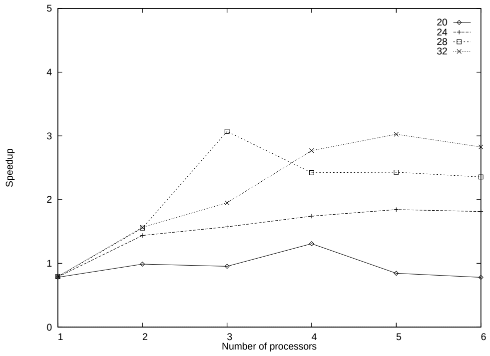

Figure 11: `loadtest` speedups on SunMP

Note that the size of the threads is exponential in the value of the command-line argument because `nfib` is used as the busy-work for each thread.

Figure 10 shows the results of running `loadtest` with different size threads on different numbers of processors of the Ethernetted SparcClassics. Figure 11 shows the results of the same experiments run on the SunMP.

Interesting points from these results:

- With one exception, speedup increases as thread size increases.
- For the Ethernetted SparcClassics with very small threads, adding processors results in a (further) slowdown; however, the amount of the slowdown seems to be limited.

- Surprisingly, the SunMP system gives lower speedups than the Ethernetted system. We have not yet investigated why this is, but it could be that our load distribution strategy is not very well suited to machines with such a low latency as the SunMP. It should also be noted that the absolute times for the SunMP runs are nevertheless almost all better than for the Ethernetted system, due to the higher performance of the individual processors on the SunMP.

4.3 The effect of packet size

The following program, `bulktest`, was designed to verify that the bulk fetching mechanism is operating correctly. It can also be used to determine the optimal value for the packet size for programs which use all of the reachable data. It simply generates a list of Ints (length set by a command-line argument) on one processor, and consumes the list (summing it) on another processor.

```
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
   )
```

Figure 12 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.

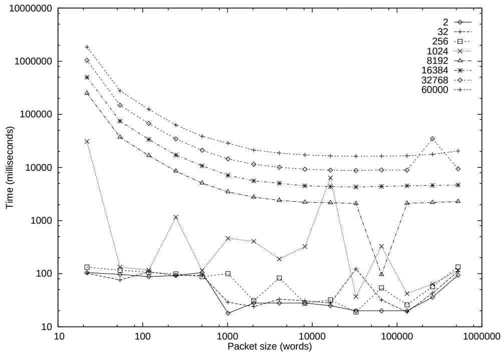

Figure 12: `bulktest` runtimes on Ethernetted SparcClassics

Interesting points from Figure 12:

- 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

5.1 GRIP

As will be obvious to those familiar with our previous work, GUM is a lineal descendent of the GRIP runtime system [7], though several simplifications have been made. While we no longer have two kinds of memory, we have retained the 2-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. We have also retained a similar message-passing structure, though the number of messages has been significantly reduced from about 20 in GRIP, to about 6 for GUM. The load distribution mechanism is also similar to, but simpler than, the most refined versions used on GRIP, which used distributed load information to maintain an even load distribution.

The GRIP (Graph Reduction in Parallel) architecture supported a virtual shared memory model for graph reduction, using two levels of physical memory: PE memory and Intelligent Memory Units (IMUs) [7]. Every closure had a global address field, but only closures in the IMUs had a Global Address, and could be shared between PEs. A GA comprised an (IMU, index) pair. Most closures were *local*, i.e. never referenced globally, in which case the GA field was empty. Because closures typically occupy only 4 or 5 words, global address information incurred a space overhead of between 20% and 25%.

In order to support locally independent garbage collection, GRIP required that there were no pointers from global heap closures to those held in local memory. As described earlier, GUM instead uses tables of "in-pointers" that each represent a global reference to a local heap closure. A similar scheme is used in other systems for distributed machines such as Concurrent Clean [5].

On GRIP, unlike GUM, the spark pool was also divided in two. Each PE maintained a pool of local sparks, and the intelligent memory units (IMUs) each maintained separate pools of global sparks. Idle PEs obtained work from the IMU pools. If the system became underloaded the IMUs then refilled their pools by demanding sparks from PEs. While this scheme prevented PEs from processing FISH messages unnecessarily, because of the restriction that there could be no pointers from global to local heap, when a closure was moved from a local spark pool to a global spark pool, *all* of its reachable graph was also exported. This could lead to significant performance losses, particularly on machines with a high-latency network. We avoid the problem using GUM's on-demand scheme.

On GRIP, we used a synchronous fetch strategy, where PEs simply entered a busy-waiting loop rather than context-switching while fetching a node. While this simplified the run-time system by preventing collisions between threads on the same PE and eliminating awkward context-switches, simulation results show that this strategy generally has a negative impact on performance even at the low latencies of the GRIP communication system.

Finally, on GRIP each closure was fetched as it was needed rather than a group of related closures being fetched eagerly, as in GUM. Kesseler has described a similar scheme that is used for Concurrent Clean [12]. While we expected GUM-style "bulk fetching" to reduce delays when using machines with high-latency networks, initial simulation results using GranSim appear to show that the greatest benefits are actually achieved for low- and medium latency networks. While this may be an artefact of the programs studied (simple parallel divide-and-conquer programs), it may also be that at very high latencies, it is simply not worth attempting to parallelise most programs.

6 Further Work

6.1 Performance Tuning

The GUM implementation has been debugged functionally, but its performance has not yet been tuned. Some obvious areas to investigate are:

- What are the optimal packet sizes for our initial target machines? The GranSim simulator [9] might be a quick way to determine these.

- How good is the load management strategy? There are several issues that need to be explored here. Firstly, we need to ensure that not too much time is spent processing FISH messages.

  Secondly, FISH messages currently follow an entirely random path when searching for work. While some randomness is necessary to avoid successive FISH messages following the same path, and therefore never finding the work that is available, it might be more efficient for each PE to record where it last found work, and to send FISH messages alternately randomly and to this work-source. Finally, the FISH message could also record information about the PEs visited, e.g. their load, to provide global information about the system. For example, load information could be used to decide whether a failed FISH message should delay before being reissued.

- When it receives a FETCH from a heavily loaded PE, a lightly loaded PE could first evaluate any thunks that were demanded before shipping the packet.

- Currently all closures are globalised during the packing process, which involves adding them to the GA table on the packing PE. It would be possible to avoid globalising some closures, and so both speed packing and reduce the size of the GA tables, at the cost of losing some sharing.

- Currently local addresses in a PE's GA $\rightarrow$ LA table are treated as roots for local garbage collection. The GA $\rightarrow$ LA scan could be made optional, depending on how much space the PE has available. Effectively local copies of global closures could be sacrificed to retrieve more space.

6.2 Shortcomings

There are currently a number of shortcomings in the GUM system which may occasionally cause performance problems or even outright inability to cope with some programs. None of these are fundamental problems; we have just left them as the last few things to tidy up:

- At present it is impossible to transmit closures that don't fit into a single packet. This can be a problem if the program contains large data structures, such as packed strings. One solution would be to split large closures into multiple packets, as on GRIP.

- GUM does not currently support task migration: consequently, once a spark is converted into a thread on a PE, it is only ever evaluated on that PE. There is good evidence that task migration is sometimes needed to obtain good overall performance [4, 9]. Since some internal thread objects, such as the thread stack, may not fit into a single packet, this may also require the use of a multi-packet protocol.

- Currently, each top-level constant (or *PLC*) is evaluated on every PE that needs its value. This is an exception to our rule that thunks are never duplicated. For small PLCs, this may not matter, since it is probably cheaper to re-evaluate the PLC than to send a FETCH/RESUME message pair to obtain its value. Larger PLCs are more problematic, since re-evaluation can reduce parallelism, and cause space leaks. Parallel Haskell programmers must therefore be careful when generating large PLCs. If support for speculation was added to GUM, then a scheme like Aharoni's [1] might be incorporated. The scheme speculatively reduces the PLC, and if the reduction has not terminated in a short interval the PE communicates to obtain the value.

- Garbage collection could be developed in several ways. For example, some of the desperation measures used to reclaim space on GRIP, such as reverting foreign closures into Fetch-Mes [7], could also be beneficial in GUM. At present, it is also not possible to reclaim graph that forms a cycle between 2 or more PEs. Circumventing this would involve implementing some kind of global stop-and-copy garbage collection: a third level of garbage collection. Full global garbage collection might even balance the weights held by each PE.

- More sophisticated statistics-gathering and visualisation tools might be developed. For example to profile heap usage on each PE over time.
- Support might be provided for the more sophisticated parallel annotations we have proposed elsewhere [9], e.g. *parLocal* to generate local sparks or *parAt* to generate a spark on a remote PE.

6.3 Extensions

There are two obvious extensions that could be made to parallel Haskell.

Firstly, there is a concurrent variant of Haskell, which is typically used to implement user interfaces as a network of communicating sequential processes [6]. An obvious extension is to integrate concurrent and parallel Haskell, creating networks of communicating parallel processes.

Secondly, speculative evaluation [15] might also be supported. Some interesting programs rely on speculation to achieve performance, so this could open up some new applications. Care must be taken, however, to avoid the high overheads that are often associated with speculative techniques.

Concurrency

In Concurrent Haskell, threads are created by `fork` e_1 e_2, analogous to `par` e_3 e_4. The semantic difference is that, while the first argument of `par`, i.e. e_3, may be ignored, the first argument of a `fork`, i.e. e_1, *must* be evaluated. Concurrent threads communicate and synchronise via special variables, *MVars* [6].

In terms of the run-time system, the threads executing e_2, e_3 and e_4 are *mandatory*, but the thread executing e_1 is *advisory*. Currently GUM has only one mandatory thread: the main thread. A fair scheduler is required to ensure that all mandatory threads are evaluated.

Some changes to GUM that would be necessary to support concurrent and parallel Haskell simultaneously are:

- A more sophisticated termination algorithm: *all* mandatory threads must complete before the program can terminate, rather than just the main thread.
- New message types would be required to read and write *MVars*.

Speculation

GUM already supports rudimentary speculative evaluation. In `par` e_1 e_2, if e_2 is not strict in e_1, then its evaluation is speculative. Such speculation is unwise under GUM if e_1 consumes many resources, or sparks additional tasks. A speculative RTS may terminate speculative tasks if they are discovered to be unnecessary, and may revert the graph into the state prior to the execution of the speculative task [15, 18].

Some of the machinery required to support speculation is already in place in GUM. For example, speculative threads could generate revertable black-holes rather than normal black-holes. These can then be reverted to their original state if the thread which is evaluating them is found to be unnecessary.

7 Summary

This paper has described a highly portable parallel implementation of Haskell, built on the PVM communications harness. 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 retargeted to other message-passing libraries such as MPI, or even GRIP's own operating system (GLOS).

We also expect to need to tune our system, especially for shared-memory systems, and perhaps introduce new parallel hints that can be exploited by some classes of architecture.

References

[1] Aharoni G, Barak A and Ronen A, "A Competitive Algorithm for Managing Sharing in the Distributed Execution of Functional Programs ", Submitted to *Journal of Functional Programming* (1995).

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

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

[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] van Eekelen MCJD, Nöcker EGJMH, Plasmeijer MJ, and Smetsers JEW, "Concurrent Clean", *Technical Report 89–18*, University of Nijmegen, (October 1989).

[6] Finne SO, and Peyton Jones, SL, "Concurrent Haskell", (1995).

[7] 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).

[8] Hammond K, Mattson JS, and Peyton Jones SL, "Automatic Spark Strategies and Granularity for a Parallel Functional Language Reducer", *Proc. CONPAR '94*, Linz, Austria, Springer-Verlag LNCS 854, (September 1994), pp. 521–532.

[9] 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).

[10] Hammond K, Loidl H-W, Mattson JS, Partridge AS, Peyton Jones SL, and Trinder PW, "GRAPHing the Future", *Proc. 6th. Intl. Workshop on Implementation of Functional Languages*, Glauert JRW (ed.), University of East Anglia, (September 1994).

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

[12] 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.

[13] Lamport L, "Time, Clocks and the Ordering of Events in a Distributed System", CACM **21**(7), (July 1978).

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

[15] 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.

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

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

[18] Partridge AS, "Speculative Evaluation in Parallel Implementations of Lazy Functional Languages", PhD Thesis, University of Tasmania, (1991).

[19] Peyton Jones SL, Clack C, Salkild J, and Hardie, M, "GRIP – a high-performance architecture for parallel graph reduction", *Proc FPCA '87*, Portland, Oregon, Kahn G (ed.), Springer-Verlag LNCS 274, (1987).

[20] Peyton Jones SL, Clack C, Salkild J, "High-performance parallel graph reduction", Proc *PARLE*, Odijk E, Rem M, Syre J-C (Eds) Springer Verlag LNCS 365 (June 1989)

[21] Peyton Jones SL, Hall CV, Hammond K, Partain WD and Wadler PL, "The Glasgow Haskell Compiler: a Technical Overview", *Proc. JFIT '93*, Keele, (March 1993).

[22] Peyton Jones SL, "Implementing Lazy Functional Languages on Stock Hardware: the Spineless Tagless G-Machine", *Journal of Functional Programming*, **2(2)**, (April 1992), pp. 127- -202.

[23] 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.

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

[25] 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).

Appendix A Table of Message Type in GUM

The messages required to support parallel graph reduction under GUM are enumerated below.

FETCH a remote closure

- RemoteGA: global address of closure being fetched.
- LocalGA: global address (on this PE) of closure where the fetched graph is to be unpacked
- Load: of requesting PE. Currently unused.

RESUME the thread waiting on the 'fetched' closure.

- RemoteGA: global address where the packet is to be unpacked.
- Size: size in bytes of the following data packet.
- Data: Packet of graph, format described in Section 2.4.

ACK that a RESUME or SCHEDULE has been processed successfully. Also overwrite any thunks shipped.

- Task: (pvm) task identifier of PE.
- NGAs: number of global addresses pairs in the following table.
- GAGAMap: a sequence of global address pairs, use is described in Section 2.4

FISH for work.

- DestPE: current target PE of this FISH message.
- OrigPE: originating PE of the FISH.
- Age: of the fish. Old fish die, as explained in Section 2.2.2.
- History: to record 'global' load information. Currently unused.
- Hunger: to record how desperate the PE/FISH is for work. Currently unused.

SCHEDULE a netted spark.

- OrigPE: the target of the SCHEDULE.
- Size: the size in bytes of the following data packet.
- Data: Packet of graph, format described in Section 2.4.

FREE some global addresses.

- PE: the target of the FREE.
- Size: number of pairs of GAs in the following field.
- Data: a sequence of weight, local-identifier pairs.

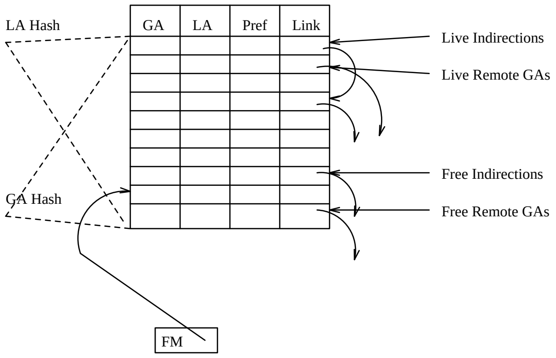

Figure 13: GALA-pair Table

Appendix B Physical Data Structures

Although logically the GIT, GA$\rightarrow$LA, and LA$\rightarrow$GA tables are separate entities, they are implemented by a single data structure, the GALA-pair table. The GALA-pair table is a collection of GALA-pairs, with hashed access from both LA and GA. A GALA pair comprises a GA, including a weight, a LA, and a boolean indicating whether the GA is the preferred GA.

The GALA-pair table has the following properties.

- All GAs are unique in the GALA-pair table: i.e. a GA can only have a single LA. Conversely a LA can have many GAs, this copes with the situation when weight has been exhausted. At least one GA will be preferred, i.e. the GA with the most weight.
- The GA returned by a LAGA lookup is a preferred GA.
- It is not true that only one GALA-pair is the preferred GA, because an indirection with a preferred GA may be shorted to a closure with a preferred GA.

The data structure is depicted in Figure 13. Fetch-Me closures point directly to the GALA-pair that identifies the remote closure. Lookup in the LA$\rightarrow$GA table is implemented by hashing the LA. Similarly lookup in the GA$\rightarrow$LA table is implemented by hashing the GA. The GA$\rightarrow$LA table can be enumerated by following the chain of *Live Remote GAs*. GA$\rightarrow$LA entries are allocated from, and deallocated to, the chain of *FreeRemoteGAs*.

GIT lookup is implemented by hashing the GA. The GIT can be enumerated by following the chain of *Live Indirections*. Local identifiers are allocated from, and deallocated to, the chain of *Free Indirections*.