GUM: a portable parallel implementation of Haskell

PW Trinder K Hammond JS Mattson Jr * AS Partridge †
SL Peyton Jones ‡
Department of Computing Science, Glasgow University
Email: {trinder,kh,simonpj}@dcs.glasgow.ac.uk

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.

The paper appears in the Proceedings of Programming Language Design and Implementation, Philadelphia, USA, May 21-24, 1996.

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 [21], 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 short-cuts 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. 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. The speedups are gained using one of the best available sequential Haskell compilers, 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 3.1).
- *GUM provides a suite of tools for monitoring and visualising the behaviour of programs.* The bottom line for any parallel program is performance, and 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. These tools are outside the scope of this paper, but are discussed in [9].
- *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

* Author's present address: Hewlett-Packard, California Language Laboratory. 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.

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 without synchronising with other PEs, a property we found to be crucial on the GRIP multiprocessor [22].

- *Thread distribution is performed lazily, but data distribution is performed somewhat eagerly.* Threads are never exported to another 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. Other ports are in progress.

The remainder of this paper is structured as follows. Section 2 describes how the GUM run-time system works. Section 3 gives preliminary performance results. Section 4 discusses related work. Section 5 contains a discussion and outlines some directions for development.

2 How GUM works

The first action of a GUM program is to create a PVM manager task, whose job is to control startup and termination. The manager 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 a local 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 (Sections 2.2 and 2.3.4).
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. These techniques are standard practice 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 non-preemptively 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 explicitly in a Haskell program by the `par` combinator. At present these combinators are added by the programmer, though we would of course like this task to be automated. The `par` combinator implements a form of parallel composition. Operationally, when the expression `` x `par` e `` is evaluated, the heap object 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`.

Sparking a thunk is a relatively cheap operation, consisting only of adding a pointer to the thunk to the PE's *spark pool*. 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. A sparked thunk is similar to a *future*, as used in MultiLisp and TERA C [19].

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)[1].
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.

[1] In fact, thunks are only overwritten with black holes when a thread context switches. The advantage of this *lazy black-holing* is that many thunks may have been entered and updated without ever being black-holed.

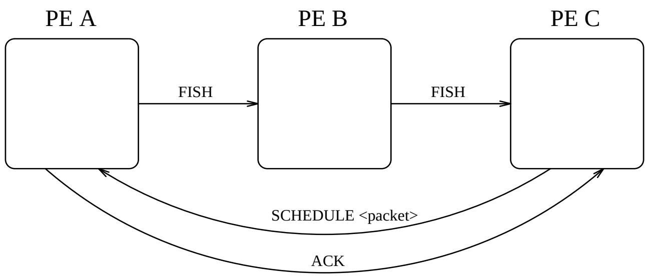

Figure 1: Fish/Schedule/Ack Sequence

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 be under evaluation by another thread. If so, the PE simply discards the spark and tries the next in first-in first-out (FIFO) order. FIFO gives good results for divide-and-conquer programs because large-grain threads are given priority. 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.

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 (again located by a FIFO search), 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 incoming message may have unblocked some thread. A sequence of messages initiated by a FISH is shown in Figure 1.

[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.

2.3 Memory Management

Parallel graph reduction proceeds on a shared program/data graph, so a primary function of GUM is to manage the virtual shared memory in which the graph resides.

2.3.1 Local Addresses

Since GUM is based on the Glasgow Haskell Compiler, most execution is carried out in precisely the same way as on a uniprocessor. In particular:

- Each PE has its own local heap.
- New heap objects are allocated from a contiguous chunk of free space in this local heap.
- The heap-object addresses manipulated by the compiled code are simply one-word pointers within the local heap which we term *local addresses*.
- 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 object from one PE's local heap to another's. For example, when PE C in Figure 1 with plenty of sparks receives a FISH message, it sends one of its sparked thunks to A, the originating PE. When a thunk is moved in this way, the original thunk is (ultimately) overwritten with a *FetchMe object*, containing the *global address* of the new copy on A. Why does the thunk need to be overwritten? It would be a mistake simply to copy it, because then both A and C might evaluate it separately (remember, there might be other local pointers to it from C'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 object on that PE. Such a scheme would, however, prevent the PEs from performing compacting garbage collection, since that changes the local address of most objects. Since compacting garbage collection is a crucial component of our efficient compilation technology we reject this restriction.

Accordingly, we follow standard practice [14] and allocate each globally-visible object 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 object. The GIT is treated as a source of roots for local garbage collection, and is adjusted to reflect the new locations of local heap objects following local garbage collection[3]. We say that a PE *owns* a globally-visible object (that is, one possessing a global address) if the object's global address contains that PE's identifier.

[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.

A heap object 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 object's address. Of course, it is possible that the object 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 object. Naturally, the LA → GA table has to be rebuilt during garbage collection, since objects' local addresses may change.

A PE may also hold copies of globally-visible heap objects 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 the following three tables. In practice the tables are coalesced into a single data structure.

- 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).

2.3.3 Garbage collection

This scheme has the obvious problem that once an object 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, e.g. [14], we use *weighted reference counting* [2, 24] to recover local identifiers, and hence the objects they identify.

We augment both the GIT and the GA → 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 → 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 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 the local addresses in its 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 objects in the hope that they will prove useful in the future.

   If instead 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 object has (for some other reason) been identified as live by the garbage collector are redirected to point to the object's new location. Any entries whose object 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. The new GA maps to the same local address in the GIT. The new and old GAs are aliases for the same heap object, which is unfortunate because it means that some sharing is not preserved. 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* objects. When a thread enters a Fetch-Me the following steps are carried out:

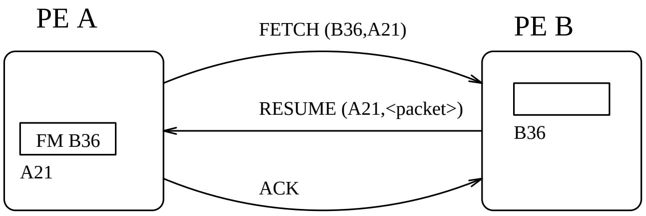

Figure 2: Fetch/Resume/Ack Sequence

1. The Fetch-Me object 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 temporary alias for it.

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

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

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

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 object will also join the queue attached to the Fetching object.

On receipt of a FETCH message, the target PE packages up the appropriate object, 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 object, and redirects the Fetching object 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 an object 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. The objective is to increase throughput over high-latency networks by sending fewer, larger messages. Packing arbitrary graph is a non-trivial problem, and [9] discusses related work and the algorithm and heuristics used in GUM.

Packing proceeds object by object, in a breadth-first traversal of the graph. As each object is packed it is given a global address, if necessary, and its location in the packet is recorded in a table, so that sharing and cycles within the packet are preserved. We stop packing when either all reachable graph has been packed, or the packet is full. Once the

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

packet is full, the links from packed objects to local heap objects are packed as Fetch-Mes.

Unpacking traverses the packet, reconstructing the graph in a breadth-first fashion. As each object is unpacked the GA $\rightarrow$ LA table is interrogated to find existing local copies of the object. 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 object, care is taken to choose the more defined object. In particular, an incoming normal-form object 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 object.

While objects representing (normal form) values are freely copied, care is taken to ensure that there is only ever one copy of a thunk, which represents a potentially unbounded amount amount of work. An ACK message is sent after unpacking a packet to indicate the new location of any thunks in the packet.

3 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.

3.1 Single Processor Efficiency

The first experiment investigates the underlying costs of parallel evaluation, compared with sequential evaluation, on a single processor. Section 3.3 investigates parallel overheads on multiple processors. Unless the overhead on a single processor is small, we cannot hope to achieve good *absolute* speed-ups, i.e. speed-ups relative to a good sequential implementation. The single-processor overhead 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 object 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. This toy program is a good stress-test for the second overhead, because when compiled for sequential execution all of the values used in the main loop are held in registers. However, in its parallel form, the compiler is obliged to insert code to build a heap object 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 (Section 3.2) 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.

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 reports parallel overheads using `parfact` on one processor from two Sparc-based architectures. The Sun multiprocessor has six Sparc 10 processors and communicates by shared memory segments. The Sun 4/15s are connected to a common ethernet segment. A memory-resident 4Mb heap is used on both machines. The figures reported are the mean of at least three runs of the program on a lightly-loaded, but not necessarily completely idle, machine. Unix scheduling introduces some variability into the results, and hence they are reported to only 2 significant figures. The GUM runtimes are presented as a percentage efficiency, i.e. the sequential runtime divided by the GUM runtime.

Single-processor Efficiency

Platform	**seq. runtime**	**seq-par efficiency**	**par-worst**	**par-best**
Sun 4/15	43.2s	93%	35%	92%
SunMP	35.9s	90%	36%	92%

The **seq. runtime** column gives the elapsed or wall-clock runtime in seconds when the fully optimised sequential version of the program is run under the standard (sequential) runtime system.

The **seq-par efficiency** column of the table gives the efficiency when the sequential version of the program is run under GUM on a single processor. The GUM runtimes are elapsed time, but exclude the startup time because it is a small (e.g. 0.6s elapsed), fixed period independent of the program being run. The increased runtime shows that the overhead imposed by GUM on all code, including sequential, is in the region of 10%.

The **par-worst** column of the table gives the efficiency 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 heap object to spark in every recursive call. The overheads are 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 a heap object 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 efficiency 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 choosing a cut-of value of 8192 produces a good thread-size for GUM on both architectures. The behaviour of `parfact` with cut-off 8192 is more typical of parallel programs in that it creates some heap objects in order to spark them, but also has large sections of sequential code.

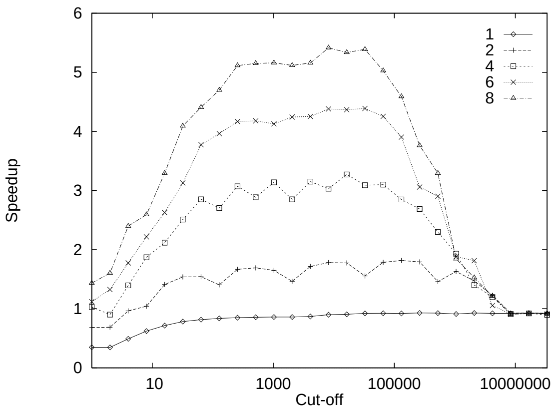

Figure 3: `parfact` speedups on Ethernetted Sun 4/15s

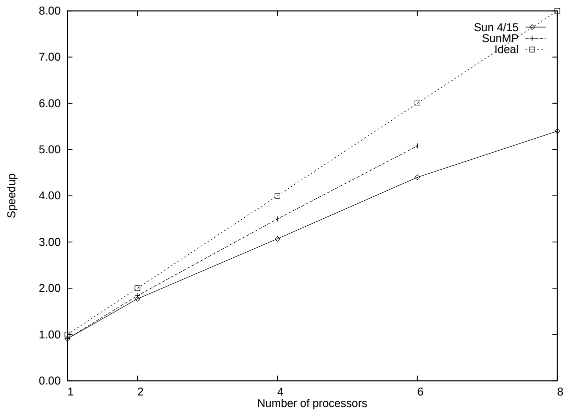

Figure 5: Linearity of `parfact` speedups

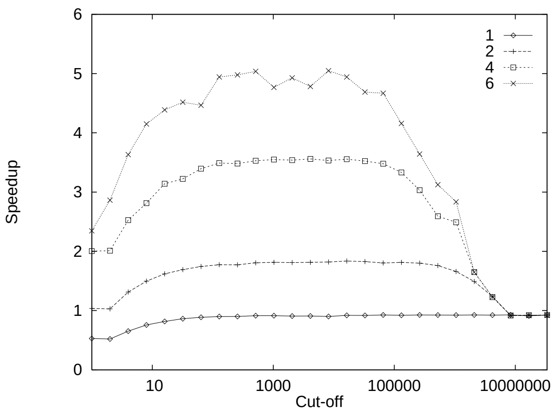

Figure 4: `parfact` speedups on SunMP

3.2 Granularity Investigation

In the following experiment we investigate the minimum acceptable grain-size for two different architectures, again using `parfact`. Figure 3 shows the absolute speedups obtained for `parfact` with different cut-off values and different numbers of processors on the network of Sun 4/15s. Figure 4 shows results from the same experiments run on the Sun multiprocessor.

The speedups shown in these figures are median speedups obtained over 4 runs. The maximum speedup curves (not given here) are smooth, but the 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.

The peak speedup achieved on the SunMP with 6 processors was 5.1, at a cut-off value of 128. For the Ethernetted Sun 4/15s, the peak speedup with 6 processors was 4.4, at a cut-off value of 8192. 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, a good thread size is independent of the number of processors. Furthermore, a good speedup is not dependent on getting exactly the right thread size: good speedups are achieved for cut-off values between 128 and 8192.

3.3 Multiprocessor Efficiency

The results from Section 3.2 assure us that GUM is not grossly inefficient on a single processor. With multiple PEs, what are the overheads of communication, scheduling multiple threads and managing global data? The linearity of speedup as more processors are added gives a good measure of the parallel efficiency of an implementation. Figure 5 plots the speedups obtained for the `parfact` program with good granularity on both architectures, i.e. using a cut-off of 8192.

As expected, efficiency falls with more processors because of greater communication and increased volumes of global data. Because of the high communication cost, the ethernetted Sun 4/15s lose efficiency more quickly than the SunMp. With six processors efficiency in the Sun 4/15 network has fallen from 92% to 73%, and in the SunMp from 92% to 85%.

3.4 The effect of packet size

To investigate the effect of packet size we use a program that generates a list of integers on one processor, and consumes the list (summing it) on another. Figure 6 shows the absolute runtimes for `bulktest` when run on a pair of Sun 4/15 workstations connected to the same segment of Ethernet. The x-axis shows varying packet sizes, while the multiple

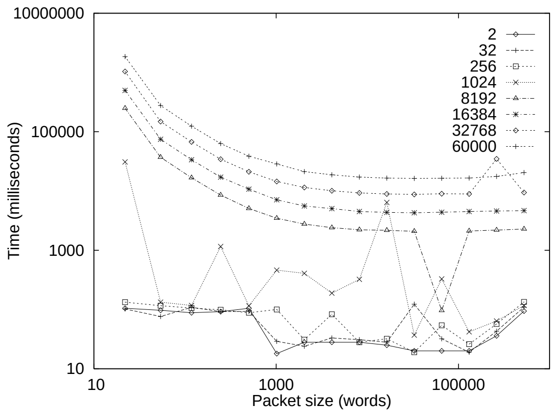

Figure 6: `bulktest` runtimes on Ethernetted Sun 4/15s

plots are for different list lengths, as set by a command-line argument.

We make the following observations on the results. 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`.

4 Related Work

There has been much work on parallel functional programming. Hammond [8] 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. Less closely related are the variety of implementations for strict functional languages, including Lisp derivatives such as Qlisp [7] or Mul-T [12], and dataflow languages such as Sisal [16].

4.1 Shared-Memory Implementations

Shared-memory implementations of non-strict functional languages 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 [6], 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, an expensive operation on a shared-memory machine. 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. As a consequence absolute speedups achieved were 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 [18].

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 which may lead to starvation, and should be used only on programs that are coarse-grained or continuously generate parallelism. On the Sequent Balance, GAML achieves relative speedups of between 3.3 and 5.8 for small programs [17].

WYBERT [13] is based on the FAST/FCG sequential compiler, and runs on a 4-processor Motorola HYPERmodule. Rather than defining a general primitive for parallelism, the implementation uses an explicit divide-and-conquer skeleton. This limits the programs that can be run to those suiting a single, albeit common, paradigm. The cost of locking is avoided 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. Relative speedups are fairly good: between 2.4 and 4 on 4 processors.

4.2 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 [5]. Unfortunately, the communication overhead on this system was high, and performance results were disappointing: relative speedups of around 4 to 8 being achieved for 32 processors.

Like WYBERT, ZAPP [15] 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 HDG-Machine [11] 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 [14]. 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 speedup for naive Fibonacci was 3.6 on 4 transputers.

Concurrent Clean runs on transputers and networks of Macintoshes [20]. 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 [10].

4.3 GRIP

GUM's design is a development of our earlier work on the GRIP multiprocessor [22]. 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. Objects were fetched from global memory singly on demand rather than using GUM-style bulk fetching. While GUM no longer has two kinds of memory, we have retained the two-level separation of local and global heap that permits each PE to garbage collect independently. Another potential advantage is that purely local objects might be held in faster unshared memory on a shared-memory machine.

On GRIP, in addition to the spark pool on each PE, special hardware maintained a distributed global spark pool. GRIP's scheme had the advantage that PEs never processed FISH messages unnecessarily, but, because local memory was unshared, a sparked thunk could only be exported after *all* graph reachable from it had also been exported. The work-stealing scheme used in GUM avoids this problem. Relative speedups on GRIP were generally good, for example, a parallel ray tracer achieved speedups of around 14 on 16 processors.

5 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 (and onwards) of the Glasgow Haskell compiler. Further information is available on the internet at http://www.dcs.gla.ac.uk/fp/software/ghc.html.

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. A port of GUM to a CM5, a distributed-memory machine with 512 Sparc processors, is nearing completion at Los Alamos National Laboratory. We hope that experiments with GUM on shared-memory and distributed-memory machines will reveal how realistic a single implementation is for both classes of architecture.

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.

The development of GUM is being driven by users who want parallel Haskell to make their programs run faster. The two main users are a 30K-line natural language processor, and a user writing complex database queries. Both are in the preliminary stages of parallelising their applications.

While we have initially targeted PVM because of its wide availability this is not a fixed decision and our implementation is designed to be easily re-targeted to other message-passing libraries such as MPI. Indeed the CM5 implementation uses the CMMD message-passing library native to the machine. We would like to measure the speedups can be obtained by using machine-specific communication primitives, particularly on shared-memory machines.

The GUM implementation could be improved in many ways. The load management strategy could be made less naive. 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 [23]. 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] 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.

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

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

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

[9] Hammond K, Mattson JS, Partridge AS, Peyton Jones SL, and Trinder PW "GUM: a portable Parallel Implementation of Haskell", *Proc 7th. Intl. Workshop on Implementation of Functional Languages*, Bastad, Sweden (September 1995).

[10] 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.

[11] 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.

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

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

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

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

[16] 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).

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

[18] 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).

[19] Mohr E, Kranz DA, and Halstead RH, "Lazy Task Creation – a Technique for Increasing the Granularity of Parallel Programs", *IEEE Transactions on Parallel and Distributed Systems*, **2(3)**, (July 1991), pp. 264–280.

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

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

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

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

[24] 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).

5.1 Appendix: Parallel Factorial

```
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)
```