Thread Migration in a Parallel Graph Reducer

André Rauber Du Bois[1], Hans-Wolfgang Loidl[2], and Phil Trinder[1]

[1] School of Mathematical and Computer Sciences,
Heriot-Watt University, Riccarton, Edinburgh EH14 4AS, U.K.
{dubois,trinder}@macs.hw.ac.uk
[2] Ludwig-Maximilians-Universität München,
Institut für Informatik, D 80538 München, Germany
hwloidl@informatik.uni-muenchen.de

Abstract. To support high level coordination, parallel functional languages need effective and automatic work distribution mechanisms. Many implementations distribute potential work, i.e. sparks or closures, but there is good evidence that the performance of certain classes of program can be improved if current work, or threads, are also distributed. Migrating a thread incurs significant execution cost and requires careful scheduling and an elaborate implementation.

This paper describes the design, implementation and performance of thread migration in the GUM runtime system underlying Glasgow parallel Haskell (GpH). Measurements of nontrivial programs on a high-latency cluster architecture show that thread migration can improve the performance of data-parallel and divide-and-conquer programs with low processor utilisation. Thread migration also reduces the variation in performance results obtained in separate executions of a program. Moreover, migration does not incur significant overheads if there are no migratable threads, or on a single processor. However, for programs that already exhibit good processor utilisation, migration may increase performance variability and very occasionally reduce performance.

1 Introduction

The potential of functional languages to support parallelism with minimal programmer intervention has been long recognised [31], but has only recently been realised using sophisticated language implementations, e.g. [4, 9, 29]. Parallel functional languages typically generate massive, but fine-grained, parallelism and a successful implementation must have effective mechanisms to distribute work across the parallel machine. In many models potential work is easily and cheaply distributed, e.g. in graph reduction a *spark* is simply a reference to an unevaluated closure in the graph. On receiving a potential work item an idle processor will create a *thread* to perform the computation, and the thread has an execution state, typically including stack(s) and a set of registers.

It is well known that the performance of certain classes of programs can be improved if, in addition to potential work, threads can be distributed. These are programs with poor load balance leading to under utilisation of some processors:

some processors are idle while others have several threads to execute. Many data parallel programs are vulnerable to this, especially those that generate parallelism only at the start of execution. Like software developers in other parallel languages, we have encountered poor load balance in large-scale GpH programs [19]. As a simple example, Figure 1 shows an overall activity profile of the sumEuler program discussed in section 4. The profile is recorded on 8 processors and shows execution time on the X-axis and the number of threads on the Y-axis. The shades of gray in the figure represent the different states of the threads, and the key states are *running*, i.e. currently executing and *runnable*, i.e. could be executed but residing on a processor currently executing another thread. For much of the execution there are idle processors and runnable threads simultaneously. If these threads can be *migrated* from a heavily-loaded processor to a lightly-loaded processor, runtime can be reduced.

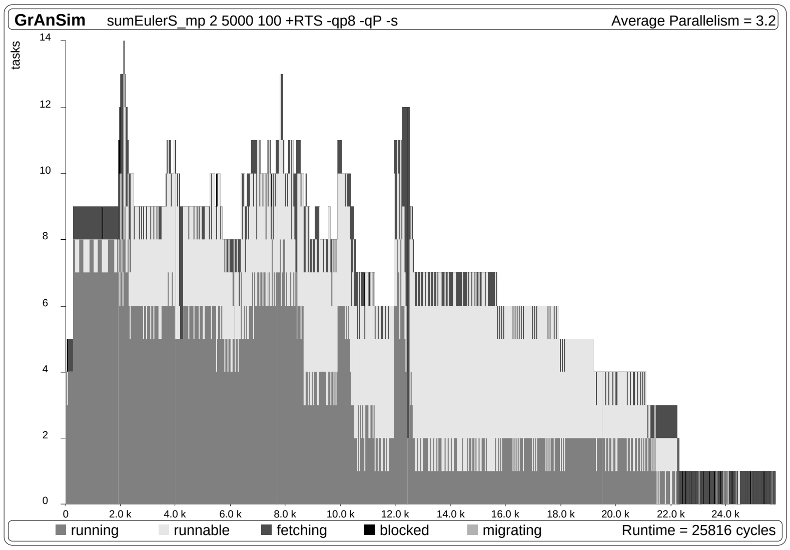

Fig. 1. Activity Profile of sumEuler, a Program with Migratable Threads

Although conceptually simple, engineering effective thread migration in a sophisticated compiled parallel language implementation is challenging. The execution state of a thread has an elaborate representation that must be carefully packed, communicated and unpacked for execution to resume. Moreover the sharing of data and computations by the thread with other threads on the source and destination processors must be preserved. A further consequence is that migrating threads is not only expensive but also destroys any locality of reference the thread enjoyed on the source processor. In consequence thread migration must be carefully scheduled, to be used only when other load balancing mechanisms have failed. It is salutary that relatively few systems have been constructed, examples include [3] and [10]. This paper describes the implementation of thread migration in the GUM runtime system [29] supporting GPH [28].

The remainder of the paper is structured as follows. Section 2 describes GPH and its associated GUM runtime system, including existing work distribution mechanisms. Section 3 presents the design and implementation of the new thread migration mechanism. Section 4 gives preliminary performance measurements. Section 5 covers related work and Section 6 concludes.

2 GpH and GUM

2.1 GpH

GPH(*Glasgow Parallel Haskell*) [28] is a modest and conservative extension of Haskell 98, using the parallel combinator `par` to specify parallel evaluation. The expression `` p `par` e `` (here we use Haskell's infix operator notation) has the same value as `e`. Its dynamic effect is to indicate that `p` could be evaluated by a new parallel thread, with the parent thread continuing evaluation of `e`. Higher-level coordination is provided using *evaluation strategies*: higher-order polymorphic functions that use `par` and `seq` combinators to introduce and control parallelism [28].

2.2 GUM Parallel Virtual Machine

Figure 2 summarises the main components of GUM. Potential parallelism is represented as "sparks", i.e. pointers to graph structures in the heap, which are collected in a spark pool. Sparks are generated by executing the `par` primitive. Threads are generated on a processing element (PE), consisting of a CPU and local memory, if it is idle, i.e. if its thread pool is empty. More threads are added from blocking queues when the required data becomes available. By design the generation of sparks is very cheap, while threads are far more heavy-weight (although still light compared to usual OS threads). Both spark and thread pools are managed as FIFO queues. To create a new thread the PE activates one of its sparks, generating a Thread State Object (TSO), which holds essential information such as registers and a stack pointer. The scheduler then determines how to choose one of the threads from the thread pool to run on the graph reduction engine (Reducer). If a running thread blocks on unavailable data, it is added to the blocking queue of that node. A blocked thread is added to the thread pool when the required data becomes available, either because a local thread produces it or the data arrives from another processor.

Figure 3 illustrates the communications induced by the existing spark distribution mechanism, a *work stealing* scheme. A FISH message requests work and specifies the PE requesting work as well as an age limit, i.e. the maximum number of PEs to visit, in the form `FISH(Source, Age)`. Initially all PEs except for the main PE are idle and without sparks. Idle PEs send a `FISH` message to

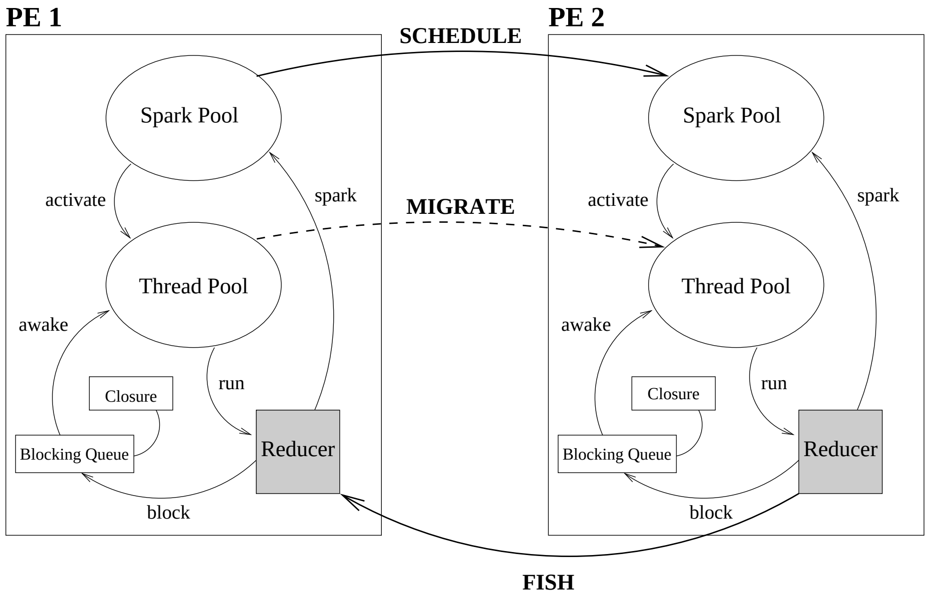

Fig. 2. Interaction of the Components of a GUM PE.

a PE chosen at random, and only ever have one outstanding FISH: in our case PE A sends to PE C. If a FISH recipient has an empty spark pool it increases the age and forwards the FISH to another PE chosen at random, in our case PE D. If a FISH recipient has a spark it sends it to the source PE as a SCHEDULE message: in our case PE D sends to PE A. The age limit of a FISH is used to avoid swamping a lightly loaded machine with FISH messages: if the age reaches this limit (a tunable system parameter) the FISH is returned to the source PE which delays before reissuing another FISH.

3 Implementing Thread Migration

3.1 Scheduler

A central component in GUM is the scheduler, which determines which thread to execute next. To implement thread migration the following scheduling policy is used. It attempts thread migration only if other cheaper work location schemes fail. The new policy is an extension of the existing policy and has been tested for simulated parallel execution [16], but hasn't previously been available in GUM.

1. execute another runnable thread, if available;
2. turn a spark into a thread if no runnable threads are available;
3. try to acquire a remote spark if the processor has no local sparks;
4. try to **migrate** another runnable thread if no remote sparks can be found;

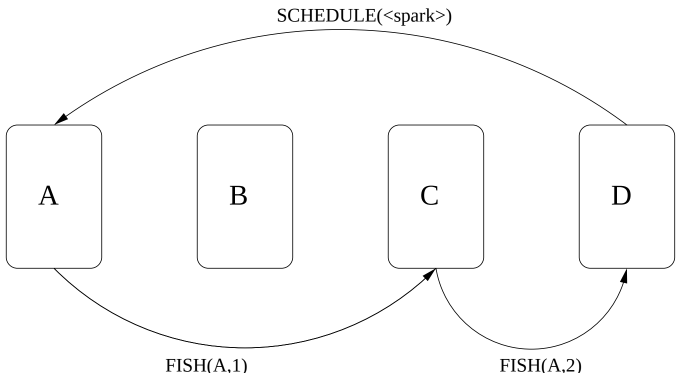

Fig. 3. Spark Distribution in GUM

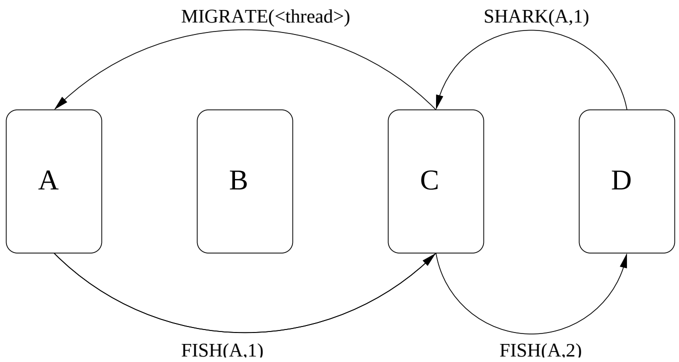

Fig. 4. Thread Migration in GUM

Several scheduling alternatives are possible. Currently the TSO of a migrated thread is added at the end of the (FIFO) runnable queue. Migrated threads could be preferred by inserting at the front of the runnable queue, or more generally by distinguishing them in the queue.

3.2 Communication Protocol

The communication protocol in GUM is very simple and consists of only 6 classes of messages [29]. Thread migration introduces two new messages, both variants of existing messages: a SHARK which is a hungry variant of the FISH message; and MIGRATE which is a variant of the SCHEDULE message and transmits a thread and an associated graph structure between PEs.

Figure 4 illustrates the communications induced by the new thread migration mechanism. As before, a FISH seeks to locate a spark, and in our example the FISH visits PEs C and D unsuccessfully. Now, however, when a FISH reaches its age limit instead of returning to its source PE, it becomes a SHARK message with age 1 and is forwarded to a random PE. When a SHARK arrives, if the PE has a spark it is sent to the source PE in a SCHEDULE message; otherwise if the PE has a runnable thread it is sent to the source PE in a MIGRATE message; otherwise the PE increases the age and forwards the SHARK at random. SHARKs and FISHes have the same age limit, and a SHARK reaching this age limit is returned to the source PE. In the example the SHARK finds a thread, but no spark on PE C, and migrates it to PE A.

At the moment GUM does not propagate load information between PEs. One potential improvement of the load balancing mechanism would be to carry information about spark and thread pool sizes of the visited PEs as part of a FISH message. In our experience even the naive, but cheap mechanism of randomly choosing the target of a FISH works well for most applications. In the presence of thread migration the additional overhead might be justified, though, because in general runnable threads are much rarer than available sparks. Similarly, information about the granularity of threads would be useful in order to choose the largest thread. However, such information is not directly available and hard to obtain automatically [15].

3.3 Graph Packing

The main modification to enable thread migration is to provide mechanisms to pack and unpack threads for communication between PEs. This entails packing a TSO and its associated stack. Since a TSO is a (slightly special) heap object, we simply extend the cases for packing a graph node, to include a case for a TSO. Packing most of the entries in the TSO is uncomplicated, since they are static data rather than pointers. The exceptions are the pointers in the stack that have to be adjusted when unpacking the TSO.

GUM is a parallel extension of the STG-machine [26] and the TSO and stack layout is unchanged. Figure 5 summarises the transfer of a thread, represented as a TSO, from PE 1 to PE 2 (note, that the stack grows downwards). The TSO can be partitioned into a header, containing mostly non-pointer data, and the stack. The stack consists of a continuous sequence of variable sized activation records. Each record starts with one of 4 possible frame types (named in Figure 5): an update frame, a stop frame, a seq frame or an exception frame. The most common type is an update frame, which contains a pointer into the heap, pointing to a graph structure that will be updated with the result of the current evaluation. As shown in Figure 5, the type of the updatee will be either BH, a "black hole" representing a graph structure under evaluation, or a BH_BQ, a "black hole blocking queue" which additionally contains a list of TSOs waiting for the result of this evaluation. An exception frame contains a pointer to the code that has to be executed when catching an exception. A seq frame contains a pointer to the code corresponding to the second part of a sequential composition (the `y` in a `x 'seq' y` construct). A stop frame can only occur at the bottom of the stack and indicates the end of the computation for this thread. All frames are linked together, with one thread register pointing to the top-of-stack frame. The layout of an activation record itself is specified by a bitmask immediately after the

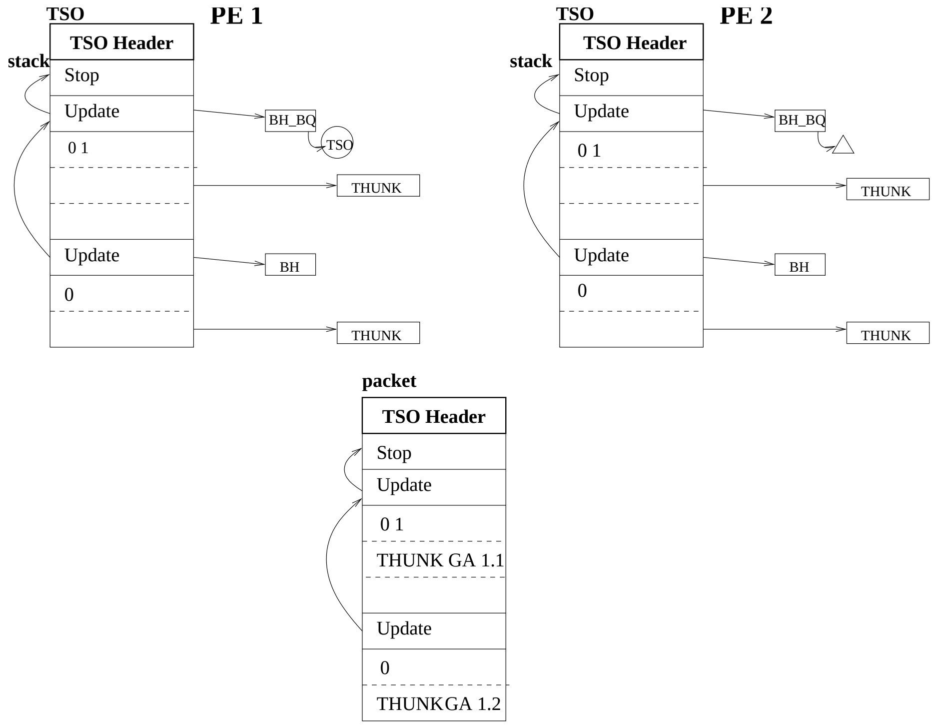

Fig. 5. Transfer of a Thread (TSO) between PEs.

update frame, with 1 representing data and 0 a pointer entry. Since this layout is similar to the one of a partial application closure we can treat the elements of one activation record on the stack in the same way as the available arguments in a partially applied function when packing the stack.

During packing, the overall structure of the stack is maintained, but some modifications are made. As with all graph structures, global addresses (GAs) have to be allocated for pointers, in order to ensure that thunks, or unevaluated closures, are uniquely identified in the virtual shared heap. This can be seen in the packet in Figure 5 where the two thunks are packed with new global addresses GA 1.1 and GA 1.2. As with ordinary graph structures, a mapping of old GAs on the source PE, to new GAs on the target PE, is sent back as a reply to the communication shown here and the BH and BH_BQ closures on PE 1 are converted into FetchMe (FM) and FetchMe blocking queue (FM_BQ) closures. If a thread on PE 1 demands a thunk being evaluated by the migrated TSO, a FETCH request will be sent to PE 2 upon entering the FM or FM_BQ closure.

Note that when unpacking the black hole blocking queue (BH_BQ) on PE 2, a different kind of closure has to be used to represent the TSO that is blocked on the black hole on PE 1. This closure is a "blocked fetch" closure, which already exists in GUM. It normally represents a fetch request from another PE, and contains information about the requesting PE and TSO, so that upon updating the BH_BQ closure, a message with the result data is sent to the original PE. By this mechanism the TSO on PE 1 will continue as soon as the migrated TSO updates the BH_BQ on PE 2.

4 Performance Measurements

The measurements in this section are performed on a high-latency cluster: a Beowulf [27] consisting of Linux RedHat 6.2 workstations with a 533MHz Celeron processor, 128kB cache, 128MB of DRAM and 5.7GB of IDE disk. The workstations are connected through a 100Mb/s fast Ethernet switch with a latency of 142μs, measured under PVM 3.4.2.

Table 1. SumEuler Runtimes(s) with/without Migration

	Mean Runtime		Min. Runtime		Max. Runtime		Avg No Thr Mig	% Range		Performance Improvement
No PEs	**Mig.**	No Mig.	**Mig.**	No Mig.	**Mig.**	No Mig.		**Mig.**	No Mig.	
1	**97.5**	97.5	**97.3**	97.4	**97.7**	97.6	0	**0.4%**	0.2%	0%
2	**51.9**	52.5	**50.2**	50.1	**54.2**	55.4	1	**7.7%**	10%	1.1%
4	**23.5**	26.1	**20.5**	21.4	**26.3**	31.7	4.2	**24.6%**	39.4%	10%
6	**17.0**	23.7	**15.66**	20.4	**18.8**	27.3	3	**18.4%**	29.1%	28%
8	**14.7**	20.8	**12.8**	14.9	**17.8**	27.5	4.0	**34%**	60.5%	29%
10	**11.7**	17.7	**9.5**	14.7	**12.9**	20.2	3.4	**29%**	31%	33%
12	**10.6**	17.5	**9.0**	12.6	**11.9**	20.8	3.4	**27.3%**	46.8%	39%
14	**10.5**	15.1	**8.1**	11.8	**11.8**	19.7	4.6	**35.2%**	52.3%	30%
16	**11.2**	14.3	**9.8**	8.4	**12.5**	19.0	5.2	**24.1%**	74.1%	21%
18	**8.8**	13.5	**7.9**	11.2	**9.2**	14.8	3.4	**14.7%**	26.6%	34%
20	**8.4**	12.8	**5.3**	9.0	**11.1**	14.9	4	**69%**	46.3%	34%
22	**8.5**	13.1	**5.7**	10.9	**11.4**	17.6	4.6	**67%**	51%	35%

4.1 Performance Improvement

Experience has shown that many GpH programs have migratable threads and some under-utilised processors [19], and four programs from two parallel paradigms are discussed in this section.

The first program, sumEuler, is data parallel: computing the sum of the Euler totient function over an integer list [18]. Figure 6 shows mean speedup curves for sumEuler with and without migration calculated from five executions of the program. The speedups reported here and throughout the paper are *relative*, i.e. improvement over the single-PE parallel execution. Table 1 shows, for varying

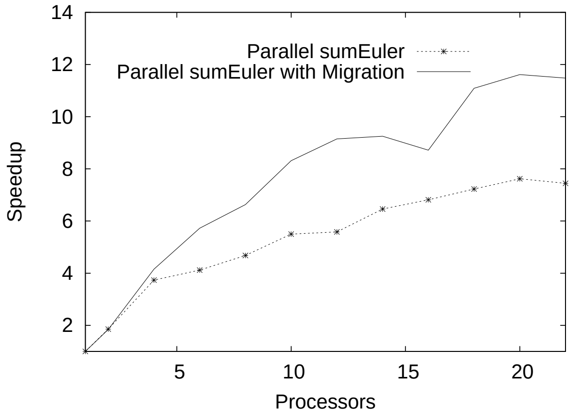

Fig. 6. Speedups for sumEuler

numbers of PEs: the mean, minimum and maximum runtimes, the average number of threads migrated in each execution, the range of runtimes as a percentage of the mean runtime and the percentage reduction in mean runtime.

The sumEuler results show that thread migration improves runtime for all numbers of PEs measured. For small numbers of PEs the improvement is limited by the small numbers of migratable threads, but between 6 and 22 PEs the improvement is approximately 30% with a single exception. The improvement is variable, as discussed in the next section, with the greatest improvement being 39% on 12 PEs.

The second program, Maze, uses a divide-and-conquer algorithm to search a maze for an exit [8]. Figure 7 shows the mean speedup curves and table 2 the performance improvements. The results show that thread migration improves performance on all numbers of PEs; from 4 PEs onwards improvements of approximately 13% are achieved with two exceptions. The improvement is variable with the greatest improvement being 21% on 16 PEs.

A third program, Raytracer, is data parallel: constructing a scene from objects and a viewpoint [11, 17]. Space precludes a full presentation of Raytracer results, but as with the previous programs they show that thread migration improves performance on all numbers of PEs; from 3 PEs onwards improvements of approximately 20% are achieved. The improvement is variable with the greatest improvement being 34% on 5 PEs.

The fourth program, Queens, is data parallel: placing chess pieces on a board. Figure 8 shows the mean speedup curves and table 3 the performance improvements. The results show that while thread migration improves performance on most configurations, it degrades it on two; the 4-PE and 8-PE configurations. There is an enormous amount of variability in the improvement, with a maximum of 37% and minimum of -10%.

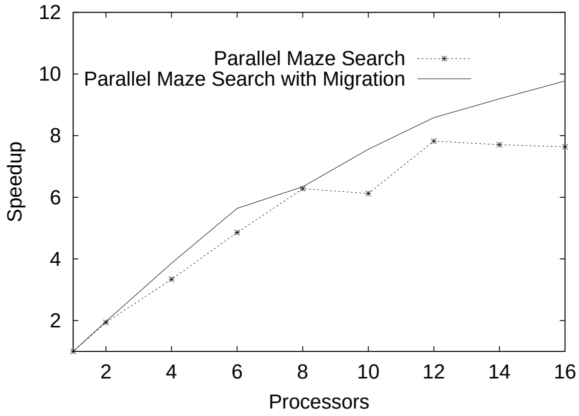

Fig. 7. Speedups for Maze

Thread migration does not consistently or significantly improve the performance of either Queens or Maze up to 8 PEs because both have excellent processor utilisation, achieving a speedup of approximately 6 in each case. If the default GUM work distribution mechanism is achieving good utilisation, it is hard for the more expensive migration mechanism to improve on it. Indeed results for Queens show that the additional communication introduced may reduce performance (e.g. at 4 and 8 PEs), and increase variability. In contrast migration delivers significant and consistent improvements when utilisation is low, e.g. on 8 PEs sumEuler without migration has a speedup of just 4.7 and migration delivers a 29% improvement. Likewise on 7 PEs Raytracer has a speedup of just 4 and migration delivers a 21% improvement. In a similar way, migration improves the performance of both Queens and Maze at higher numbers of PEs as the utilisation delivered by the default load balancing mechanism falls.

In summary, thread migration always improves the runtime of programs with migratable threads and low processor utilisation, like sumEuler, Maze and Raytracer. Thread migration often improves but may degrade programs with migratable threads and good processor utilisation, like Queens. In both cases the runtimes and improvements achieved are variable. The improvements are achieved by migrating a relatively small number of threads: typically around 4. This indicates that the migration policy described in section 3 works adequately, striking a balance between good data locality and even load distribution.

4.2 Variability

We hypothesised that thread migration would reduce the variability in runtimes, as it allows a poor initial distribution of sparks to PEs to be rectified. The '% Range' column in tables 1 and 2 together with measurements of the Raytracer show that migration reduces the range of performance results for programs with low utilisation, like sumEuler, Maze and Raytracer. However, table 3 shows that

Table 2. Maze Runtimes(s) with/without Migration

	Mean Runtime		Min. Runtime		Max. Runtime		Avg No Thr Mig	% Range		Performance Improvement
No PEs	**Mig.**	No Mig.	**Mig.**	No Mig.	**Mig.**	No Mig.		**Mig.**	No Mig.	
1	**162.9**	162.0	**162.6**	159.7	**163.7**	163.5	0	**0.6%**	0 %	0%
2	**82.7**	83.5	**82.5**	83.3	**82.9**	83.5	1	**0.4%**	0.2%	0.9%
4	**42.2**	48.5	**40.4**	43.1	**44.1**	58.2	0.8	**8.7%**	31.1%	13%
6	**28.9**	33.3	**27.9**	30.8	**30.3**	35.1	2	**8.3%**	12.9 %	13%
8	**25.7**	25.8	**23.8**	24.9	**28.3**	29.1	5.8	**17.5%**	16.2%	0%
10	**21.5**	26.4	**19.8**	24.5	**24.1**	29.6	2.4	**20%**	19.3%	18%
12	**18.9**	20.7	**16.3**	16.7	**22.0**	24.6	4.2	**30.1%**	38.1%	8%
14	**17.7**	21.0	**16.0**	18.0	**20.3**	24.7	4	**24.2%**	31.9%	15%
16	**16.6**	21.2	**12.2**	16.4	**20.3**	25.3	4.4	**48.7%**	41.9%	21%

migration may increase the variability of some programs with good utilisation, like Queens on small numbers of processors. In these cases, we have observed increased communication in programs with migration and suspect the variability is due to both increased communication and potential blocking after having migrated threads. Depending on the amount of sharing with other graphs on the original PE, threads on a PE may send data requests and become blocked, thereby reducing the gain in performance due to migration. However, none of the programs with poor utilisation seems to suffer from an increased amount of data transfer caused by migration.

Table 3. Queens Runtimes(s) with/without Migration

	Mean Runtime		Min. Runtime		Max. Runtime		Avg No Thr Mig	% Range		Performance Improvement
No PEs	**Mig.**	No Mig.	**Mig.**	No Mig.	**Mig.**	No Mig.		**Mig.**	No Mig.	
1	**405.4**	405.6	**405.3**	405.2	**405.7**	405.9	0	**0.0%**	0.1%	0%
2	**225.0**	227.8	**219.6**	227.8	**227.7**	227.9	0	**3.6%**	0.0%	1%
4	**128.4**	116.7	**114.9**	116.6	**169.6**	116.8	1.5	**42.6%**	0.1%	-10%
6	**91.7**	104.9	**78.03**	103.1	**117.6**	108.7	4.8	**43.1%**	5.3%	12%
8	**78.9**	75.1	**72.2**	74.9	**102.1**	75.3	2.5	**37.8%**	0.5%	-5%
10	**67.3**	73.2	**63.2**	67.1	**68.8**	103.0	1.33	**8.3%**	49%	8%
12	**44.2**	70.9	**38.8**	62.9	**70.8**	73.1	1.33	**72.3%**	14.3%	37%
14	**39.0**	50.5	**39.0**	38.9	**39.1**	78.9	0.84	**0.2%**	79.2%	22%
16	**44.9**	50.0	**38.9**	39.0	**73.9**	72.3	1.6	**77.9%**	66.6%	10%

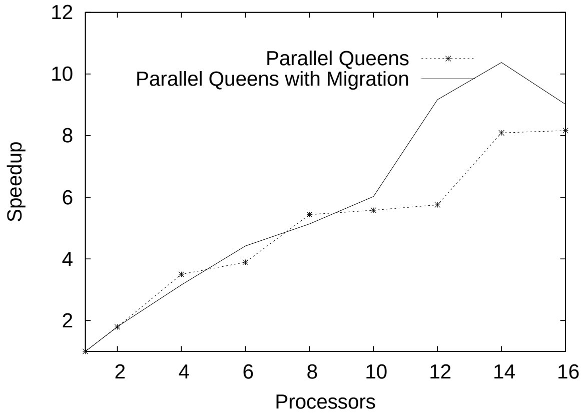

Fig. 8. Speedups for Queens

A second factor is that programs with a short runtime allow little time for migration to correct a poor initial load distribution. This can be seen in the relatively high variability of sumEuler on 20 and 22 PEs in table 1. However, migration still helps even in these cases: the mean, minimum and maximum runtimes are always smaller than for the program without migration.

4.3 Overheads

To investigate the overheads of thread migration two simple programs without migratable threads have been measured: one data-parallel the other divide-and-conquer. For each program table 4 reports the paradigm, the mean runtime with and without migration, the average number of threads migrated in each execution and the percentage change in runtime, all measured on 7 PEs. The results show that the migration mechanism has no significant overhead if there are no migratable threads. Moreover, tables 1, 2 and 3 together with measurements of the Raytracer demonstrate that execution with migration enabled on a single PE does not incur significant additional overheads compared to parallel execution without migration.

Table 4. Migration Overheads

Program Name	Paradigm	Mean Runtime		Avg No	% Change
		With Migration	Without Mig.	Thr Mig	in Runtime
ParFib 35	Div & Conq.	**5.6**	5.7	0	+2%
ParMap	Data Par.	**25.2**	25.3	0	+1%

5 Related Work

Thread migration suffered from bad press due to early news of excessive overhead. Therefore, comparable systems that provide light-weight, and typically fine-grained, threads tend to avoid migration [5, 7, 13, 14].

In the context of earlier versions of parallel graph reducers, early experiments on the GRIP system, implemented on a special-purpose distributed memory machine, indicated, that despite the availability of several sparking strategies to control and balance parallelism, thread migration is still needed for some applications to guarantee high utilisation [10].

The Cid system [24] extends C with primitives for creating ("forking") new threads and synchronising ("joining") them on shared variables, managed in a virtual shared heap. The Cid systems holds runnable threads in two different queues: one for the threads that must be executed on the current processor, one for threads that might be migrated to another processor. The cost for handling the messages is reduced by using the technique of active messages [30] where the message carries a pointer to a function, the "handler", that is called when receiving the message. The effectiveness of Cid's load balancing and latency tolerance mechanisms is assessed in [25].

Cilk [3] is similar to Cid, in that it extends C with constructs for creating and synchronising light-weight threads. Its processors also use a sophisticated work-stealing scheduler to obtain new parallelism. While Cilk provides thread migration, it is optimised for local execution of a thread, since that is the most common case. Thus, still high overhead is associated with migration.

The Filaments system [20] is in many aspects similar to our system: it emphasises fine-grained parallelism on a distributed shared heap, with dynamic and implicit management of work and data; the programmer is only required to expose parallelism. It is implemented as an extension of C. Two levels of threads can be distinguished: filaments, with a code pointer and arguments but without a stack, and server threads, with an attached stack, acting as a scheduler over a set of filaments. In balancing the load of the system, it employs a sophisticated adaptive data placement mechanism, that tries to minimise access to remote data and uses information gained from dynamic monitoring of data access. Overall, the system focuses on the placement of data rather than threads.

The Ariadne Threads system [21] is C-based and implements user-level threads and explicit thread migration on shared and distributed memory machines. In contrast to Filaments, placement decisions focus on threads, rather than data, by migrating a thread to the location of its data, rather than vice versa.

The Amber system [6] uses a virtual shared memory model, implemented on the Topaz operating system and is programmed in C++. It provides library calls to realise dynamic clustering of threads at runtime, via explicit thread migration. In contrast to Ariadne it puts limitations on the total number of threads per node, but gains reduced packing and thread management overhead.

Another virtual shared memory system focusing on thread migration is Millipede [12]. It uses kernel-threads and is very flexible, allowing explicit migration at almost any point in the execution. It uses dynamic mechanisms migrating both threads and data to maximise data locality. Stack packing is simplified by guaranteeing that stacks will occupy the same place on all processors. This reduces packing costs but wastes some memory space.

A lot of research was done on process migration in the area of operating systems in the late 70s [23]. The objective of introducing process migration into an operating system was to improve load balancing and fault tolerance. Thus, process migration should be transparent to the user and is not under the control of the programmer. Many operating systems were designed to support process migration, some well known examples are Mach [2] and MOSIX [1].

We are primarily interested in thread migration as a way to obtain even load balance and thereby improve performance in a system for parallel computation. Alternative applications of this technique, with different design requirements, are the use of migration in persistent systems [22] and for mobile computing [23].

6 Conclusions

We have presented a design and implementation of thread migration for the GUM runtime system underlying GpH. The design exploits the uniform representation of heap objects in the STG-machine: the packing of a TSO and its stack only requires an extension of the default packing mechanism to handle a new kind of heap closure. The design also makes minimal extensions to the relatively simple GUM communication protocol, adding only two new kinds of messages. Here we profit from both data and threads being represented by heap objects.

Performance measurements of six programs on a high-latency Beowulf cluster show that thread migration in GUM can improve the performance, and reduce the variability in performance, of data-parallel and divide-and-conquer programs with low processor utilisation. In summary: sumEuler is 35% faster on 22 PEs, Maze is 21% faster on 16 PEs, Raytracer is 21% faster on 7 PEs, and Queens is 10% faster on 16 PEs. Measurements of these and two other programs show that migration does not incur significant overheads if there are no migratable threads, or on a single PE. However, it is hard for migration to improve programs that already have good utilisation, and migration may both increase variability and occasionally reduce performance. For example neither Maze nor Queens is significantly improved by migration until 10 or more PEs are used.

In future work it may be possible to better characterise programs and architectures where thread migration may be beneficial. GUM's thread migration mechanism and load management policies could easily be improved, e.g. replacing the random targeting of FISH and SHARK messages with a more focused approach; and possibly recording partial load information in all messages to maintain a time-stamped partial load information on each PE.

Acknowledgements

The authors would like to thank the following funding agencies for supporting this research: UK's EPSRC council (grant GR/R88137), the Austrian Academy of Sciences (fellowship APART 624), the European Community (IST-2001-33149) and the UK CVCP ORS Scheme.

References

1. A. Barak and O. La'adan. The MOSIX Multicomputer Operating System for High-Performance Cluster Computing. *Future Generation Computer Systems*, 13(4–5):361–372, 1998.
2. R. Baron, R. Rashid, E. Siegel, A. Tevanian, and M. Young. Mach-1: An Operating Environment for Large-Scale Multiprocessor Applications. *IEEE Software*, 2(4):65–67, July 1985.
3. R.D. Blumofe, C.F. Joerg, C.E. Leiserson, K.H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *PPoPP'95 — Symp. on Principles and Practice of Parallel Programming*, pages 207–216, Santa Barbara, USA, 1995.
4. S. Breitinger, R. Loogen, Y. Ortega Malln, and R. Pea Marí. Eden — The Paradise of Functional Concurrent Programming. In *EuroPar'96 — European Conf. on Parallel Processing*, LNCS 1123, pages 710–713, Lyon, France, 1996. Springer.
5. T. Bülck, A. Held, W. Kluge, S. Pantke, C. Rathsack, S-B. Scholz, and R. Schröder. Experience with the Implementation of a Concurrent Graph Reduction System on an nCUBE/2 Platform. In *CONPAR'94 — Conf. on Parallel and Vector Processing*, LNCS 854, pages 497–508. Springer, 1994.
6. J.S. Chase, F.G Amador, E.D. Lazowska, H.M Levy, and R.J. Littlefield. The Amber System: Parallel Programming on a Network of Multiprocessors. In *Symp. on Operating Systems Principles*, pages 147–158, Litchfield Park, AZ, USA, 1989.
7. D.E. Culler, S.C. Goldstein, K.E. Schauser, and T. von Eicken. TAM — A Compiler Controlled Threaded Abstract Machine. *J. of Parallel and Distributed Computing*, 18:347–370, June 1993.
8. A. R. Du Bois, R. Pointon, H-W. Loidl, and P. W. Trinder. Implementing Declarative Parallel Bottom-Avoiding Choice. In *14th Symposium on Computer Architecture and High Performance Computing*, pages 82–89, Vitoria, Brazil, october 2002. IEEE Press.
9. K. Hammond and S.L. Peyton Jones. Some Early Experiments on the GRIP Parallel Reducer. In *IFL'90 — Intl. Workshop on the Parallel Implementation of Functional Languages*, pages 51–72, Nijmegen, The Netherlands, June 1990.
10. K. Hammond and S.L. Peyton Jones. Profiling Scheduling Strategies on the GRIP Multiprocessor. In *IFL'92 — Intl. . Workshop on the Parallel Implementation of Functional Languages*, pages 73–98, RWTH Aachen, Germany, September 1992.
11. Impala. Impala – (IMplicitly PArallel LAnguage Application Suite). <URL:`http://www.csg.lcs.mit.edu/impala/`>, July 2001.
12. A. Itzkovitz, A. Schuster, and L. Shalev. Thread Migration and its Applications in Distributed Shared Memory Systems. *J. of Systems and Software*, 42(1):71–87, 1998.
13. M. H. G. Kesseler. *The Implementation of Functional Languages on Parallel Machines with Distributed Memory.* PhD thesis, Wiskunde en Informatica, Katholieke Universiteit van Nijmegen, The Netherlands, 1996.

14. H. Kingdon, D.R. Lester, and G. Burn. The HDG-machine: a Highly Distributed Graph-Reducer for a Transputer Network. *Computer Journal*, 34(4):290–301, 1991.
15. H-W. Loidl. *Granularity in Large-Scale Parallel Functional Programming*. PhD thesis, University of Glasgow, March 1998.
16. H-W. Loidl and K. Hammond. Making a Packet: Cost-Effective Communication for a Parallel Graph Reducer. In *IFL'96 — Intl. Workshop on the Implementation of Functional Languages*, LNCS 1268, pages 184–199, Bonn/Bad-Godesberg, Germany, September 1996. Springer.
17. H-W. Loidl, U. Klusik, K. Hammond, R. Loogen, and P.W. Trinder. GpH and Eden: Comparing Two Parallel Functional Languages on a Beowulf Cluster. In *SFP'00 — Scottish Functional Programming Workshop*, volume 2 of *Trends in Functional Programming*, pages 39–52, University of St Andrews, Scotland, July 2000. Intellect.
18. H-W. Loidl, P.W. Trinder, and C. Butz. Tuning Task Granularity and Data Locality of Data Parallel GpH Programs. *Parallel Processing Letters*, 11(4):471–486, December 2001.
19. H-W. Loidl, P.W. Trinder, K. Hammond, S.B. Junaidu, R.G. Morgan, and S.L. Peyton Jones. Engineering Parallel Symbolic Programs in GPH. *Concurrency — Practice and Experience*, 11:701–752, 1999.
20. D.K. Lowenthal, V.W. Freeh, and G.R. Andrews. Using Fine-Grain Threads and Run-Time Decision Making in Parallel Computing. *J. of Parallel and Distributed Computing*, 37:42–54, 1996.
21. E. Mascarenhas and V. Rego. Ariadne: Architecture of a Portable Threads System Supporting Thread Migration. *Software — Practice and Experience*, 26(3):327–356, March 1996.
22. B. Mathiske, F. Matthes, and J.W. Schmidt. On Migrating Threads. In *Intl. Workshop on Next Generation Information Technologies and Systems*, Naharia, Israel, June 1995.
23. D. Milojicić, F. Douglis, and R. Weeler. *Mobility: Processes, Computers, and Agents*. Addison-Wesley, Reading, MA, USA, 1999.
24. R.S. Nikhil. Parallel Symbolic Computing in Cid. In *Workshop on Parallel Symbolic Computing*, LNCS 1068, pages 217–242, Beaune, France, Oct. 1995. Springer.
25. R.S. Nikhil and A. Singla. Automatic Granularity Control and Load-Balancing in Cid. Technical report, DEC Research Labs, December 1994.
26. S.L. Peyton Jones. Implementing Lazy Functional Languages on Stock Hardware: the Spineless Tagless G-machine. *J. of Functional Programming*, 2(2):127–202, July 1992.
27. D. Ridge, D. Becker, P. Merkey, and T. Sterling. Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs. In *IEEE Aerospace Conference*, pages 79–91, 1997.
28. P.W. Trinder, K. Hammond, H-W. Loidl, and S.L. Peyton Jones. Algorithm + Strategy = Parallelism. *J. of Functional Programming*, 8(1):23–60, January 1998.
29. P.W. Trinder, K. Hammond, J.S. Mattson Jr., A.S. Partridge, and S.L. Peyton Jones. GUM: a Portable Parallel Implementation of Haskell. In *PLDI'96 — Conf. on Programming Language Design and Implementation*, pages 79–88, Philadephia, USA, May 1996.
30. T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *ISCA'92 — Intl. Symp. on Computer Architecture*, pages 256–266, Gold Coast, Australia, May 1992. ACM Press.
31. P. Wegner. *Programming Languages, Information Structures and Machine Organisation*. McGraw-Hill, New York, 1971.