Runtime Support for Multicore Haskell

Simon Marlow
Microsoft Research
Cambridge, U.K.
simonmar@microsoft.com

Simon Peyton Jones
Microsoft Research
Cambridge, U.K.
simonpj@microsoft.com

Satnam Singh
Microsoft Research
Cambridge, U.K.
satnams@microsoft.com

Abstract

Purely functional programs should run well on parallel hardware because of the absence of side effects, but it has proved hard to realise this potential in practice. Plenty of papers describe promising ideas, but vastly fewer describe real implementations with good wall-clock performance. We describe just such an implementation, and quantitatively explore some of the complex design tradeoffs that make such implementations hard to build. Our measurements are necessarily detailed and specific, but they are reproducible, and we believe that they offer some general insights.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages; D.3.2 [*Programming Languages*]: Language Classifications—Concurrent, distributed and parallel languages; D.3.3 [*Programming Languages*]: Language Constructs and Features—Concurrent programming structures; D.3.4 [*Programming Languages*]: Processors—Runtime-environments

General Terms Languages, Performance

1. Introduction

At least in theory, Haskell has a head start in the race to find an effective way to program parallel hardware. Purity-by-default means that there should be a wealth of inherent parallelism in Haskell code, and the ubiquitous lazy evaluation model means that, in a sense, *futures* are built-in.

How can we turn these benefits into real speedups on commodity hardware? This paper documents our experiences with building and optimising a parallel runtime for Haskell. Our runtime supports three models of parallelism: explicit thread-based concurrency (Peyton Jones et al. 1996), semi-explicit deterministic parallelism (Trinder et al. 1998), and data-parallelism (Peyton Jones et al. 2009). In this paper, however, we focus entirely on semi-explicit parallelism.

Completely implicit parallelism is still a distant goal; one recent attempt at this in the context of Haskell can be found in Harris and Singh (2007). The semi-explicit GpH programming model, in contrast, has been shown to be remarkably effective (Loidl et al. 1999, 2003). The semantics of the program remains completely deterministic, and the programmer is not required to identify threads, communication, or synchronisation. They merely annotate subcomputations that might be evaluated in parallel, leaving the choice of whether to actually do so to the runtime system. These so-called *sparks* are created and scheduled dynamically, and their grain size varies widely.

Our goal is that programmers should be able to take existing Haskell programs, and with a little high-level knowledge of how the program should parallelise, make some small modifications to the program using existing well-known techniques, and thereby achieve decent speedup on today's parallel hardware. However, when we started benchmarking some existing Parallel Haskell programs, we found that many programs which at first glance appeared to be completely reasonable-looking parallel programs, in fact failed to achieve significant speedup when run with our implementation on parallel hardware.

This led us to a critical study of our (reasonably mature) baseline implementation of semi-explicit parallelism in GHC 6.10. In this paper we report the results of that study, with the following contributions:

- We give a complete description of GHC's parallel runtime, starting with an overview in Section 4, and amplified in the rest of the paper. A major constraint is that we do barely compromise the (excellent) execution speed of sequential code.
- We discuss several major sets of design choices, relating to spark distribution, scheduling, and memory management (Sections 5 and 7); parallel garbage collection (Section 6); the implementation of mutual exclusion on thunks (Section 8); and load balancing and thread migration (Section 9). In each case we give quantitative measurements for a number of optimisations that we made over our baseline GHC 6.10 implementation.
- While we focus mainly on the implementation, our work has had some impact on the programming model: we identify the need for `pseq` as well as `seq` (Section 2.1), and we isolated a signficant difficulty in the "strategies" approach to writing parallel programs (Section 7).
- On the way we developed a useful new profiling and tracing tool (Section 10.1).

Overall, our results are encouraging (Figure 1). The optimisations we describe improve the absolute runtime and scalability of all benchmarks, sometimes dramatically so. Before our work, some programs would speed up on a parallel machine, but others slowed down. Afterwards, using 7 cores of an 8-core machine yielded speedups in the range 3x to 6.6x, which is not bad for a modest investment of programmer effort.

Some of the improvements were described in earlier work (Berthold et al. 2008), along with preliminary measurements, as part of a comparison between shared-heap and distributed-heap parallel execution models. In this paper we extend both the range

of measurements and the range of improvements, while focussing exclusively on shared-heap execution.

All our results are, or will be, repeatable using a released version of the widely-used GHC compiler. Our results do not require special builds of the compiler or libraries: identical results will be obtainable using a standard binary distribution of GHC. At the time of writing, most of our developments have been made public in the GHC source code repository, and we expect to include the remaining changes in the forthcoming 6.12.1 release of GHC, scheduled for the autumn of 2009. The sources to our benchmark programs are available in the public `nofib` source repository.

Speedup on 4 cores			Speedup on 7 cores		
Program	Before	After	Program	Before	After
gray	2.19	2.50	gray	2.61	2.77
mandel	2.94	3.51	mandel	4.50	4.96
matmult	2.56	3.37	matmult	4.07	5.04
parfib	3.73	3.89	parfib	5.94	6.67
partree	0.74	1.99	partree	0.68	3.18
prsa	3.28	3.56	prsa	5.22	5.23
ray	0.81	2.11	ray	0.82	3.48
sumeuler	3.74	3.85	sumeuler	6.32	6.42

Figure 1. Speedup results

2. Background: programming model

The basic programming model is known as Glasgow Parallel Haskell, or GpH (Trinder et al. 1998), and consists of two combinators:

```
par  :: a -> b -> b
pseq :: a -> b -> b
```

The semantics of `par a b` is simply the value of `b`, whereas the semantics of `pseq` is given by

$$\begin{aligned} \texttt{pseq a b} &= \bot, && \text{if } \texttt{a} = \bot \\ &= \texttt{b}, && \text{otherwise} \end{aligned}$$

Informally, `par` stores its first argument as a *spark* in the *spark pool*, and then continues by evaluating its second argument. The intention is that idle processors can find (probably) useful work in the spark pool. Typically the first argument to `par` will be an expression that is shared by another part of the program, or will be an expression that refers to other such shared expressions.

2.1 The need for `pseq`

The `pseq` combinator is used for sequencing; informally, it evaluates its first argument to weak-head normal form, and then evaluates its second argument, returning the value of its second argument. Consider this definition of `parMap`:

```
parMap f []     = []
parMap f (x:xs) = y `par` (ys `pseq` y:ys)
   where y  = f x
         ys = parMap f xs
```

The intention here is to spark the evaluation of `f x`, and then evaluate `parMap f xs`, before returning the new list `y:ys`. The programmer is hoping to express an *ordering* of the evaluation: *first* spark `y`, *then* evaluate `ys`.

The obvious question is this: why not use Haskell's built-in `seq` operator instead of `pseq`? The only guarantee made by `seq` is that it is strict in both arguments; that is, `seq a` $\bot$ `= a` $\bot$ and `seq` $\bot$ `a = a` $\bot$. But this *semantic* property makes no *operational* guarantee about order of evaluation. An implementation could impose this operational guarantee on `seq`, but that turns out to limit the optimisations that can be applied when the programmer only cares about the semantic behaviour. Instead, we provide both `pseq` and `seq` (with and without an order-of-evaluation guarantee), to allow the programmer to say what she wants while leaving the compiler with as much scope for optimisation as possible.

To our knowledge this is the first time that this small but important point has been mentioned in print. The `pseq` operator first appeared in GHC 6.8.1.

2.2 Strategies

In *Algorithms + Strategies = Parallelism* (Trinder et al. 1998), Trinder *et al* explain how to use *strategies* to modularise the construction of parallel programs. In brief, the idea is as follows. A strategy is a function that may evaluate (parts of) its argument and create sparks, but has no interesting results:

```
type Done = ()
done = ()
type Strategy a = a -> Done
```

Strategies compose nicely; that is, we can build complex strategies out of simpler ones:

```
rwhnf :: Strategy a
rwhnf x = x `pseq` done

parList :: Strategy a -> Strategy [a]
parList strat []     = done
parList strat (x:xs) = strat x `par` parList strat xs
```

Finally, we can combine a data structure with a strategy for evaluating it in parallel:

```
using :: a -> Strategy a -> a
using x s = s x `pseq` x
```

Here is how we might use the combinators to evaluate all the element of a (lazy) input list in parallel, and then add them up:

```
psum :: [Int] -> Int
psum xs = sum (xs `using` parList rwhnf)
```

3. Making parallel programs run faster

We now turn our attention from the programming model to the implementation. Our baseline is GHC 6.10.1, a mature Haskell compiler. Its performance on sequential code is very good, so the overheads of parallelism are not concealed by sloppy sequential execution. It has supported parallel execution for several years, but while parallel performance is sometimes good, it is sometimes surprisingly bad. The trouble is that it is hard to know *why* it is bad, because performance is determined by the interaction of four systems — the compiler itself, the GHC runtime system, the operating system, and the physical hardware — each of which is individually extremely complex. The rest of this paper reports on our experience of improving both the absolute performance and its consistency.

To whet your appetite, Figure 1 summarises the cumulative improvement of the work we present, for 4 and 7 cores[1]. Each table has 2 columns:

[1] Why did we use only 7 of the 8 cores on our test system? In fact we did perform the measurements for all 8 cores, but found that the results were far less consistent than the 7 core results, and in some cases performance degraded significantly. On closer inspection the OS appeared to be descheduling one or more of our threads, leading to long pauses when the threads needed to synchronise. This effect is discussed in more detail in Section 10.1.

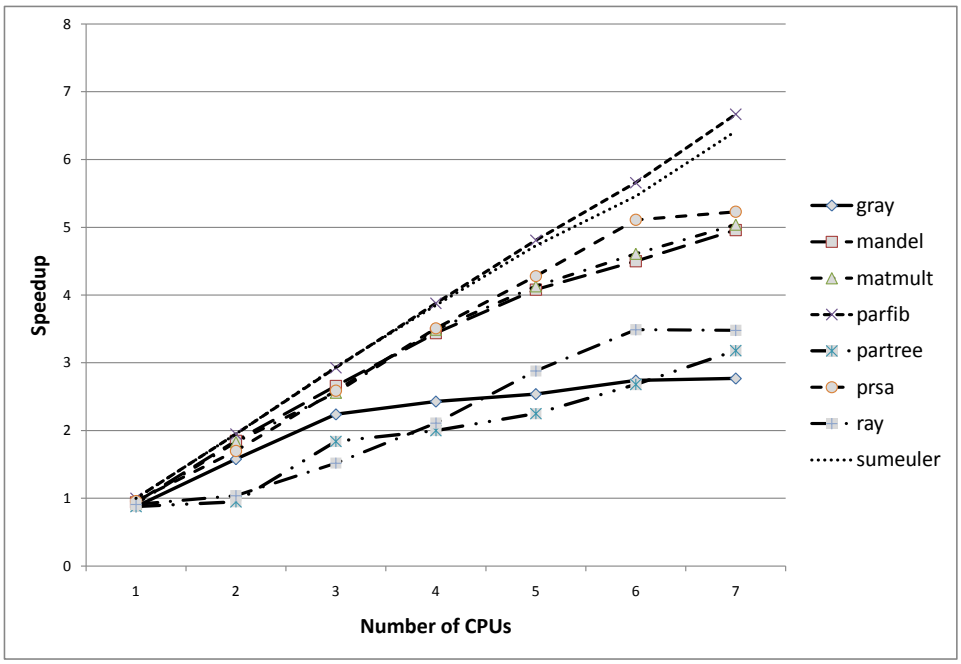

Figure 2. Speedup results

- **Before:** speedup achieved by parallel execution using GHC 6.10.1, compared to the same program compiled sequentially with 6.10.1, with the parallel GC turned off. In GHC 6.10.1 the parallel GC tended to make things worse rather than better, so this column reflects the best settings for GHC 6.10.1.
- **After:** our best speedup results, using the PARGC3 configuration (Section 6).

The improvements are substantial, especially for the most disappointing programs which actually ran slower when parallelism was enabled in 6.10.1. Section 10 gives more details about the experimental setup and the benchmark programs.

Figure 2 shows the scaling results for each benchmark program after our cumulative improvements, relative to the performance of the sequential version. By "sequential" we mean that the single-threaded version of the runtime system was used, in which `par` is a no-op, and there are no synchronisation overheads.

4. Background: the GHC runtime

By way of background, we describe in this section how GHC runs Haskell programs in parallel. In the sections that follow we present various measurements to show the effectiveness of certain aspects of our implementation design. Each of our measurements compare two configurations of GHC. Many of our improvements are cumulative and it proved difficult to untangle the source-code dependencies from each other in order to be able to make each measurement against a fixed baseline, so in each case we will clearly state what the baseline is.

4.1 The basic setup

The GHC runtime system supports millions of lightweight threads by multiplexing them onto a handful of operating system threads, roughly one for each physical CPU. This overall scheme is well-established, but it is easier to sketch than to implement!

Each *Haskell thread* runs on a finite-sized stack, which is allocated in the heap. The state of a thread, together with its stack, is kept in a heap-allocated *thread state object* (TSO). The size of a TSO is around 15 words plus the stack, and constitutes the whole state of a Haskell thread. A stack may grow by copying the TSO into a larger area, and may subsequently shrink again.

Haskell threads are executed by a set of *operating system threads*, which we call *worker threads*. We maintain roughly one worker thread per physical CPU, but exactly *which* worker thread may vary from moment to moment, as we explain in Section 4.2. Since the worker thread may change, we maintain exactly one *Haskell Execution Context* (HEC) for each CPU[2]. The HEC is a data structure that contains all the data that an OS worker thread requires in order to execute Haskell threads. In particular, a HEC contains

- An Ownership Field, protected by a lock, that records which worker thread is currently animating the capability (zero if none is). We explain in Section 4.2 why we do not use the simpler device of a lock to protect the entire HEC data structure.
- A Message Queue, containing requests from other HECs. For example, messages saying "Please wake up thread T" arrive here.
- A Run Queue of threads ready to run.

[2] In the source code of the runtime system, a HEC is called a "Capability". The HEC terminology comes from the lightweight concurrency primitives work (Li et al. 2007b). Others call the same abstraction a "virtual processor" (Fluet et al. 2008a).

- An Allocation Area (Section 6). There is a single heap, shared among all the HECs, but each HEC allocates into its own local allocation area.
- GC Remembered Sets (Section 6.2).
- A Spark Pool. Each invocation of `par` a b adds the thunk a to the (current HEC's) Spark Pool; this thunk is called a "spark".
- A Worker Pool of spare worker threads, and a Foreign Outcall Pool of TSOs that are engaged in foreign calls (see Section 4.2).

In addition there is a global Black Hole Pool, a set of threads that are blocked on black holes (see Section 8).

An active HEC services work using the following priority scheme. Items lower down the list are only performed if there are no higher-priority items available.

1. Service a message on the Message Queue.
2. Run a thread on the Run Queue; we use a simple round-robin scheduling order.
3. If any spark pool is non-empty, create a *spark thread* and start running it (see Section 5.3).
4. Poll the Black Hole Pool to see if any thread has become runnable; if so, run it.

All the state that a HEC needs for ordinary execution of Haskell threads is local to the HEC, so under normal execution a HEC proceeds without requiring any synchronisation, locks, or atomic instructions. Synchronisation is only needed when:

- Load balancing is needed (Section 9).
- Garbage collection is required (Section 6).
- Blocking on black holes (Section 8.1).
- Performing an operation on an `MVar`, or an STM transaction.
- Unblocking a thread on another HEC.
- Throwing an exception to a thread on another HEC, or a blocked thread.
- Allocating large or immovable memory objects; since these operations are relatively rare, we allocate such objects from single global pool.
- Making a (safe) foreign call (Section 4.2).

4.2 Foreign calls

Suppose that a Haskell thread makes a foreign call to a C procedure that blocks, such as `getChar`. We do not want the entire HEC to seize up so, before making the call, the worker thread relinquishes ownership of the HEC, leaving the Haskell thread in a tidy state. The thread is then placed in the Foreign Outcall Pool so that the garbage collector can find it. We maintain a Worker Pool of worker threads for each HEC, each eager to become the worker that animates the HEC. When one worker relinquishes ownership, it triggers a condition variable that wakes up another worker from the Worker Pool. If the latter is empty, a new worker is spawned.

What happens when the original worker thread W completes its call to `getChar` and wants to return? To return, it must re-acquire ownership of the HEC, so it must somehow displace any worker thread X that currently owns the HEC. To do so, it adds a message to the HEC's Message Queue. When X sees this message, it signals W, and returns itself to the worker pool. Worker thread W wakes up, and takes ownership of the HEC. This approach is slightly better than directly giving W ownership of the HEC, because W might be slow to respond, and the HEC does not remain locked for the duration of the handover.

4 cores	
Program	Δ Time (%)
gray	+6.5
mandel	-3.4
matmult	-2.1
parfib	+3.5
partree	-1.2
prsa	-4.7
ray	-35.4
sumeuler	+0.0
Geom. Mean	-5.5

7 cores	
Program	Δ Time (%)
gray	-2.1
mandel	-4.9
matmult	+1.6
parfib	-1.2
partree	-3.7
prsa	-6.7
ray	-66.1
sumeuler	+1.4
Geom. Mean	-14.4

When a spark is pushed onto an already full queue, we have a choice between discarding the new spark or discarding one of the older sparks. Our current implementation discards the newer spark; we do not investigate this choice further in this paper.

Figure 3. The effect of adding work-stealing queues vs. GHC 6.10.1

This also explains why we don't simply have a mutex protecting the HEC, which all the spare worker threads are blocked on. That approach would afford us less control in the sense that we often want to hand the HEC to a particular worker thread, and a simple mutex would not allow us to do that.

Foreign calls are not the focus of this paper, but more details can be found in Marlow et al. (2004).

5. Faster sparks

We now discuss the first set of improvements, which relate to the handling of sparks.

5.1 Sharing sparks

GHC 6.10.1 has a private Spark Pool for each HEC, but it uses a "push" model for sharing sparks, as follows. In between running Haskell threads, each HEC checks whether its spark pool has more than one spark. If so, it checks whether any other HECs are idle (a cheap operation that requires no atomic instructions); if it finds an idle HEC it gives one or more sparks to it, by temporarily acquiring ownership of the remote HEC and inserting the sparks in its pool.

To make spark distribution cheaper and more asynchronous we re-implemented each HEC's Spark Pool as a bounded work-stealing queue (Arora et al. 1998; Chase and Lev 2005). A work-stealing queue is a lock-free data structure with some attractive properties: the owner of the queue can push and pop from one end without synchronisation, meanwhile other threads can "steal" from the other end of the queue incurring only a single atomic instruction. When the queue is almost empty, popping also incurs an atomic instruction to avoid a race between the popping thread and a stealing thread.

Figure 3 shows the effect of adding work-stealing queues to our baseline GHC 6.10.1. As we can see from the results, work-stealing for sparks is almost always beneficial, and increasingly so as we add more cores. It is of particular benefit to **ray**, where the task granularity is very small.

5.2 Choosing a spark to run

Because we use a work-stealing queue for our spark pools, stealing threads must always take the *oldest* spark in the pool. However, the HEC owning the spark pool has a choice between two policies: it can take the youngest spark from the pool (LIFO), or it can take the oldest spark (FIFO). Taking the oldest spark requires an atomic instruction, but taking the youngest spark does not.

Figure 4 shows the effect of changing the default (FIFO) to LIFO. In most of our benchmarks this results in worse performance, because the older sparks tend to be "larger".

4 cores		7 cores	
Program	Δ Time (%)	Program	Δ Time (%)
gray	+10.1	gray	+5.3
mandel	+4.5	mandel	+6.2
matmult	-0.4	matmult	-1.2
parfib	+0.2	parfib	+1.0
partree	-7.3	partree	-1.6
prsa	-0.3	prsa	-2.3
ray	+10.0	ray	+18.7
sumeuler	+0.3	sumeuler	+17.9
Geom. Mean	+2.0	Geom. Mean	+5.2

Figure 4. Using LIFO rather than FIFO for local sparks

4 cores		7 cores	
Program	Δ Time (%)	Program	Δ Time (%)
gray	-0.7	gray	+2.3
mandel	-1.5	mandel	+1.3
matmult	-8.2	matmult	-11.3
parfib	+0.6	parfib	+6.4
partree	-0.6	partree	+0.3
prsa	-1.0	prsa	-1.0
ray	-31.2	ray	-24.3
sumeuler	+0.2	sumeuler	-0.3
Geom. Mean	-5.9	Geom. Mean	-3.8

Figure 5. The effect of batching sparks

5.3 Batching sparks

To run a spark a, a HEC simply evaluates the thunk a to head normal form. To do so, it needs a Thread State Object. It makes no sense to create a fresh TSO for every spark, and discard it when the evaluation is complete for the garbage collector to recover.

Instead, when a HEC has no work to do, it checks whether there are any sparks, either in the HEC's local spark pool or in any other HEC's spark pool (check the non-empty status of a spark pool does not require a lock). If there are sparks available, then the HEC creates a *spark thread*, which is a perfectly ordinary thread except that it runs the following steps in a loop:

1. If the local Run Queue or Message Queue is non-empty, exit.
2. Remove a spark from the local spark pool, or if that is empty, steal a spark from another HEC's pool.
3. If there were no sparks to steal, exit.
4. Evaluate the spark to weak-head-normal form.

where "exit" means that the spark thread exits and performs no further work; its TSO will be recovered by a subsequent GC.

A spark thread will therefore evaluate sparks to WHNF one after another until it can find no more sparks, or until there is other work to do, at which point it exits. This is a particularly simple strategy and works well: the cost of creating the spark thread is amortized over multiple sparks, and the spark thread gets out of the way quickly if any other work arrives. If a spark blocks on a black hole, since the spark thread is just an ordinary thread it will block in the usual way, and the scheduler will create another spark thread to continue running the available sparks. We don't have to worry unduly about having too many spark threads, because a spark thread will always exit when there are other threads around. This reasoning does rely on sparks not being too large, however: many large sparks becoming blocked could lead to a large number of running spark threads.

Figure 5 compares the effect of using the spark-batching approach described above to the approach taken in GHC 6.10.1, which was to create a new thread for each activated spark. Our baseline for these measurements is GHC 6.10.1 plus work-stealing-queues (Section 5.1). Batching sparks is particularly beneficial to two of our benchmarks, **matmult** and **ray**, while it is a slight pessimisation for **parfib** on 7 cores. For **ray** the rationale is clear: there are lots of tiny sparks, so reducing the overhead for spark execution has a significant effect. For **parfib** we believe that the reduction in performance shown here is because the program is actually being more effective at exploiting parallelism, which leads to reduced performance due to lack of locality (Section 6); as we shall see later, this performance loss is recovered by proper use of parallel GC.

6. Garbage collection

The shared heap is divided into fixed-size (4kbyte) *blocks*, each with a block descriptor that specifies which generation it belongs to, along with other per-block information. A HEC's Allocation Area simply consists of a list of such blocks.

When any HEC's allocation area is exhausted, a garbage collection must be performed. GHC 6.10.1 offers a parallel garbage collector (see Marlow et al. (2008)), but GC only takes place when all HECs stop together, and agree to garbage collect. We aim to keep this synchronisation overhead to a minimum by ensuring that we can stop a HEC quickly (Section 6.3). In future work we plan to relax the stop-the-world requirement and adopt some form of CPU-independent GC (Section 12.1).

When a GC is required, we have the option of either

- Performing a single-threaded GC. In this case, the HEC that initiated the GC waits for all the other HECs to cease execution, performs the GC, and then releases the other HECs.
- Performing a parallel GC. In this case, the initiating HEC sends a signal to the other HECs, which causes them to become GC threads and await the start of the GC. Once they have all responded, the initiating HEC performs the GC initialisation and releases the other GC threads to perform GC. When the GC termination condition is reached, each GC thread waits at the GC exit barrier. The initiating HEC performs any post-GC tasks (such as starting finalizers), and then releases the GC threads from the barrier to continue running Haskell code.

In a single-threaded program, it is often better to use single-threaded GC for the quick young-generation collections, because the cost of starting up and shutting down the GC threads can outweigh the benefits of doing GC in parallel.

6.1 Avoiding synchronisation in parallel copying GC

Parallel copying GC normally requires each GC thread to use an atomic instruction to synchronise when copying an object, so that objects are not accidentally duplicated. The cost of these atomic instructions is high: roughly 30% of GC time (Marlow et al. 2008). However, as we suggested in that paper, it is possible to relax the synchronisation requirement where immutable objects are concerned. The only adverse effect from making multiple copies of an immutable object is a little wasted space, and we know from measurements that the rate of actual collisions is very low–typically less than 100 collisions per second of GC time–so the amount of wasted space is likely to be minuscule.

Our parallel GC therefore adopts this policy, and avoids synchronising access to immutable objects. Figure 6 compares the two policies: the baseline is our current system in which we only lock mutable objects, compared to a modified version in which we lock every object during GC. As the results show, our optimisation of only locking mutable objects has a significant benefit on overall performance: without it, performance drops by over 7%. The effect

7 cores	
Program	Δ Time (%)
gray	+18.7
mandel	+9.4
matmult	+4.5
parfib	-0.5
partree	+17.3
prsa	+2.5
ray	+4.8
sumeuler	+5.8
Geom. Mean	+7.6

Figure 6. Effect of locking all closures in the parallel GC

4 cores	
Program	Δ Time (%)
gray	-3.9
mandel	-1.4
matmult	-0.5
parfib	-0.5
partree	+5.0
prsa	+2.3
ray	+3.4
sumeuler	-0.3
Geom. Mean	+0.5

7 cores	
Program	Δ Time (%)
gray	-9.7
mandel	-2.1
matmult	+0.0
parfib	+0.4
partree	+0.3
prsa	-0.1
ray	-2.8
sumeuler	-1.7
Geom. Mean	-2.0

Figure 7. Using the heap-limit for context switching

is the most marked in benchmarks that do the most GC: **gray**, but is negligible in those that do very little GC: **parfib**.

6.2 Remembered Sets

Remembered sets are used in generational GC to track pointers from older generations into younger ones, so that when collecting the younger generation we can quickly find all the pointers into that generation. Whenever a pointer into a younger generation is created, an entry must be added to the remembered set.

There are many choices for the representation of the remembered set and the form of its associated write barrier (Blackburn and Hosking 2004). In GHC, because mutation is rare, we opted for a relatively expensive write barrier in exchange for an accurate remembered set. Our remembered set representation is a *sequential store buffer*, which contains the addresses of objects in the old generation that contain pointers into the new generation. Each mutable object is marked to indicate whether it is dirty or not; diry objects are in the remembered set. The write barrier adds a new entry to the remembered set if and only if the object being mutated is in the old generastion and is not already marked dirty.

In our parallel runtime, each HEC has its own remembered set. The reasons for this are twofold:

- Even though appending to the remembered set is not a common operation, it is common enough that the effect of including any synchronisation would be noticeable. Hence, we must be able to add new entries to the remembered set without atomic instructions.
- Objects that have been mutated by the current CPU are likely to be in its cache, so it is desirable to visit these objects by the garbage collector on the same CPU. This is particularly important in the case of threads: the stack of a thread, and hence its TSO, is itself a mutable object. When a thread executes, the stack will accumulate pointers to new objects, and so if the TSO resides in an old generation it must be added to the remembered set. Having HEC-local remembered sets helps to ensure that the garbage collector traverses threads on the same CPU that was running the thread.

One alternative choice for the remembered set is the card table. Card tables have the advantage that they can be updated by multiple threads without synchronisation, but they compromise on accuracy. More importantly for us, however, would be the loss of locality from using a single card table instead of per-HEC sequential store buffers.

We do not have individual measurements for the benefit of using HEC-local remembered sets, but believe that it is essential for good performance of parallel programs. In GHC 6.10.1 remembered sets were partially localised: they were local during execution, but shared during GC. We subsequentially modified these partially HEC-local remembered sets to be fully localised.

6.3 Pre-emption and garbage collection

Since garbage collection is relatively frequent, and requires all HECs to halt, it is important that they all do so promptly. One way to do this would be to use time-based pre-emption; however that would essentially mandate the use of conservative GC, which we consider an unacceptable compromise. Hence in order to GC, we require that all HECs voluntarily yield at a safe point, leaving the system in a state where all the heap roots can be identified.

The standard way to indicate to a running thread that it should yield immediately is to set its heap-limit register to zero, thus causing the thread to return to the HEC scheduler when it next tries to allocate.

On a register-poor machine, we keep the heap-limit "register" in memory, in a block of "registers" pointed to by a single real machine register. In this case, it is easy for one HEC to set another HEC's heap limit to zero, simply by overwriting the appropriate memory location. On a register-rich machine we can keep the heap limit in a real machine register, but it is then a good deal more difficult for one HEC to zap another HEC's heap limit, since it is part of the register set of a running operating-system thread. We therefore explored two alternatives for register-rich architectures:

- Keep the heap limit in memory. This slows the heap-exhaustion check, but releases an extra register for argument passing.
- Keep the heap limit in a register, and implement pre-emption by setting a separate memory-resident flag. The flag is checked whenever the thread's current allocation block runs out, since it would be too expensive to insert another check at every heap-check point. This approach is cheap and easy, but pre-emption is much less prompt: a thread can allocate up to 4k of data before noticing that a context-switch is required.

Figure 7 measures the benefit of using the heap-limit register to signal a context-switch, versus checking a flag after each 4k of allocation. We see a slight drop in performance at 4 cores, changing to an increase in performance at 7 cores. This technique clearly becomes more important as the number of cores and the amount of garbage collection increases: benchmarks like **gray** that do a lot of GC benefit the most.

6.4 Parallel GC and locality

When we initially developed the parallel GC, our goal was to improve GC performance, so we focused most of our effort on using parallelism to accelerate garbage collection for *single-threaded* programs (Marlow et al. 2008). In this case the key goal is achieving good load-balancing, that is, making sure that all of the GC threads have work to do.

However, there is another factor working in the opposite direction: locality. For parallel programs, when GC begins each CPU already has a lot of data in its cache; in a sequential program only one CPU does. It would obviously make sense for each CPU to

	Δ Time (%)		
Program	PARGC1	PARGC2	PARGC3
gray	-3.9	+52.5	-11.9
mandel	+3.6	+19.0	-9.9
matmult	-14.6	-3.9	-8.1
parfib	-0.5	-1.9	-6.4
partree	+5.0	-60.7	-65.8
prsa	-0.4	+1.2	-4.4
ray	+2.0	-2.6	-18.0
sumeuler	+0.3	-0.5	-1.4
Min	-14.6	-60.7	-65.8
Max	+5.0	+52.5	-1.4
Geometric Mean	-1.2	-5.1	-19.3

Figure 8. The effectiveness of parallel GC (4 cores)

	Δ Time (%)		
Program	PARGC1	PARGC2	PARGC3
gray	+2.8	+106.2	-3.0
mandel	+8.9	+57.5	-5.2
matmult	-19.2	+14.1	-14.1
parfib	+0.9	-1.5	-7.6
partree	-2.1	-70.3	-79.8
prsa	+3.2	+16.7	+8.2
ray	-0.8	+6.2	-5.2
sumeuler	+5.1	+0.0	-2.8
Min	-19.2	-70.3	-79.8
Max	+8.9	+106.2	+8.2
Geometric Mean	-0.5	+3.7	-21.3

Figure 9. The effectiveness of parallel GC (7 cores)

garbage-collect its own data, so far as possible, rather than to allow GC to redistribute it.

Each HEC starts by tracing its own root set, starting from the HEC's private data (Section 4.1). However, our original parallel GC design used global work queues for load-balancing (Marlow et al. 2008). This is a poor choice for locality, because the link between the CPU that copies the data and the CPU that scans it for roots is lost. To tackle this, we modified our parallel GC design to use work-stealing queues. The benefits of this are threefold:

1. Contention is reduced.
2. Locality is improved: a CPU will take work from its own queue in preference to stealing. Local work is likely to be in the CPU's cache, because it consists of objects that this CPU recently copied.
3. We can easily disable load-balancing entirely, by opting not to steal any work from other CPUs. This trades parallelism in the GC for locality.

The use of work-stealing queues for load-balancing in parallel GC is a well-known technique (Flood et al. 2001), however what has not been studied before is the trade-off between whether to do load-balancing at all or not for parallel programs. We will measure our benchmarks in three configurations:

- The baseline is our best system, with parallel GC turned off.
- PARGC1: using parallel GC in the old generation only, with load-balancing.
- PARGC2: using parallel GC in both young and old generations, with load-balancing.
- PARGC3: using parallel GC in both young and old generations, without load-balancing.

PARGC1 and PARGC2 use work-stealing for load-balancing, PARGC3 uses no load-balancing. In terms of locality, PARGC2 will improve locality significantly by traversing most of the data reachable by parallel threads on the same CPU as the thread is executing. PARGC3 will improve locality further by not moving data from one CPU's cache to another in an attempt to balance the work of GC.

Figures 8 and 9 present the results, for 4 cores and 7 cores respectively. There are several aspects to these figures that are striking:

- **partree** delivers an 80% improvement with PARGC3 on 7 cores, with most of the benefit coming with PARGC2. Clearly locality is vitally important in this benchmark.
- **gray** and **mandel** degrade significantly with PARGC2, recovering with PARGC3. Load-balancing appears to be having a significant negative effect on performance here. These are benchmarks that don't achieve full speedup, so it is likely that when a GC happens, idle CPUs are stealing data from the busy CPUs, harming locality more than would be the case if all the CPUs were busy.
- PARGC3 is almost always better than the other configurations.

There are of course other possible configurations. For instance, parallel GC in the old generation only without load balancing, or parallel GC in both generations but with load-balancing only in the old generation. We have performed informal measurements on these and other configuarations and found that on average they performed less well than PARGC3, although for individual benchmarks it is occasionally the case that a different configuration is a better choice.

Future versions of GHC will use PARGC3 by default for parallel execution, although it will be possible to override the default and select any combination of parallel/sequenctial GC for each generation with and without load-balancing.

7. The space behaviour of `par`

The spark pool should ideally contain only useful work, and we might hope that the garbage collector would assist the scheduler by removing useless sparks from the spark pool.

One sure-fire way to do so is to remove any *fizzled sparks*. A spark has *fizzled* if the thunk to which it refers it has already been evaluated, so that running the spark would terminate immediately. Indeed, we expect most sparks to fizzle. The `par` operation creates *opportunities* for parallel evaluation but, if the machine is busy, few of these opportunities are taken up. For example, consider

```
x `par` (y `pseq` (x+y))
```

This sparks the thunk `x` (adding it to the spark pool), evaluates `y`, and then adds `x` and `y`. The addition operation forces both its arguments, so if the sparked thunk `x` has not been taken up by some other processor, the addition will evaluate it. In that case, the spark has fizzled.

Clearly a fizzled spark is useless, and the garbage collector can (and does in GHC 6.10) discard them, but which *other* sparks should the garbage collector retain? Two policies immediately spring to mind, that we shall call ROOT and WEAK:

- ROOT: Treat (non-fizzled) sparks as *roots* for the garbage collector. That is, retain all such sparks and the graph they point to.

	Total time(s)	MUT time(s)	GC time(s)	Total sparks	Fizzled Sparks
Strat.	10.7	5.1	5.7	1000000	0
No Strat.	6.4	5.2	1.2	1000000	999895

Figure 10. Comparison of `ray` using strategies vs. no strategies

- WEAK: Only retain (non-fizzled) sparks that are reachable from the roots of the program.

The problem is, neither of these policies is satisfactory. WEAK seems attractive, because it lets us discard sparks that are no longer required by the program. However, the WEAK policy is completely incompatible with strategies. Consider the `parList` strategy:

```
parList :: Strategy a -> Strategy [a]
parList strat []     = done
parList strat (x:xs) = strat x `par` parList strat xs
```

Each spark generated by `parList` is a thunk for the expression "`strat x`"; this thunk is not shared, since it is created uniquely for the purposes of creating the spark, *and hence can never fizzle*. Hence, the WEAK policy will discard all sparks created by `parList`, which is obviously undesirable.

So, what about the ROOT policy? This is the policy that is used in existing implementations of GpH, including GUM (Trinder et al. 1996) and GHC. However, it leads to the converse problem: *too many* sparks are retained, leading to space leaks. Consider the expression

```
sum (parList rnf (map expensive [1..100000]))
```

With the ROOT policy we will retain all of the sparks created by `parList`, and hence lose no parallelism. But if there are not enough processors to evaluate all of the sparks, they will never be garbage collected, *even after the* `sum` *is complete*! They remain in the spark pool, retaining the list elements that they point to. This can lead to serious space leaks[3].

To quantify the effect, we compared two versions of the **ray** benchmark. The first version uses `parBuffer` from the standard strategies library, applied to the `rwhnf` strategy, while the second uses a modified version of `parBuffer` which avoids the space leak (we will explain how the modified version works in Section 7.2). We ran both versions of the program on a single CPU, to illustrate the degenerate case of having too few CPUs to use the available parallelism. Figure 10 gives the results; MUT is the amount of "mutator time" (execution time excluding garbage collection), GC is the time spent garbage collecting. We can see that with the strategies version, no sparks fizzle, and the GC time suffers considerably as a result[4].

Implementations using the ROOT policy have been around for quite a long time, and yet the problem has only recently come to light. Why is this? We are not sure, but postulate that the applications that have been used to benchmark these systems do not suffer unduly from the space leaks, perhaps because the amount of extra space retained is small, and there is little or no speculation involved. If there are enough CPUs to use all the parallelism, then no space leaks are observed; the problem comes when we want to write a single program that works well when run both sequentially and in parallel.

Are there any other policies that we should consider? Perhaps we might try to develop a policy along the lines of "discard sparks that share no graph with the main program". This is clearly an improvement on the ROOT policy because it lets us discard sparks that share nothing with the main program. However, it is quite difficult to establish whether there is any sharing between the spark and the main program, since this entails establishing a "reaches" property, where each closure in the graph is marked if it can reach certain other closures (namely the main program). This is exactly the opposite of the property that a garbage collector normally establishes, namely "is reachable from", and is therefore at odds with the way the garbage collector normally works. It requires a completely new traversal, perhaps by reversing all the pointers in the graph.

Even if we could implement this strategy, it does not completely solve the problem. A spark may share data with the main program, but that is not enough: it has to share *unevaluated* data, and that unevaluated data must be part of what the spark will evaluate. Moreover, perhaps we still want to discard sparks that are retaining a lot of unshared data, but still refer to a small amount of shared data, on the grounds that the cost of the space leak outweighs the benefits of any possible parallelism.

7.1 Improving space behaviour of sparks

One way to improve the space behaviour of sparks is to use the WEAK policy for garbage collection. This guarantees, by construction, that the spark pool does not leak any space whatsoever. However, this choice would affect the programming model. In particular we can no longer use the strategies abstraction as it stands, because every strategy combinator involves sparking unique, unshared thunks, which WEAK will discard immediately. It is for this reason that GHC still uses the ROOT policy: if we were to switch to WEAK, then existing code using Strategies would lose parallelism.

We can continue to write parallel programs without space leaks under the ROOT policy, as long as we observe the rule that all sparks must be eventually evaluated. Then we can be sure that any unused sparks will fizzle, and in this case there is no difference between ROOT and WEAK. The following section describes how to write programs in this way.

Nonetheless, we believe that in the long term the implementation should use the WEAK policy. The WEAK policy has one distinct advantage over ROOT, namely that it is possible to write programs that use *speculative* parallelism without incurring space leaks. A speculative spark is by its nature one that may or may not be eventually evaluated, and in order to ensure that such sparks are eventually garbage collected if they turn out not to be required, we need to use WEAK.

7.2 Avoiding space leaks with ROOT

We can still use strategy-like combinators with ROOT, but they are no longer compositional. In the case of `parList`, if we simply want to evaluate each element to weak-head-normal-form, we use a specialised version of `parList`:

```
parListWHNF :: Strategy [a]
parListWHNF []     = done
parListWHNF (x:xs) = x `par` parListWHNF xs
```

Now, as long as the list we pass to `parListWHNF` is also evaluated by the main program, the sparks will all be garbage collected as usual. The rule of thumb is to always put a variable on the left of `par`.

Reducing the granularity with `parListChunk` is a common technique. The idea is for each spark to evaluate a fixed-size chunk of the list, rather than a single element. To do this without incurring a space leak means that the sparked list chunks must be concatenated into a new list, and returned to the caller:

```
parListChunkWHNF :: Int -> [a] -> [a]
parListChunkWHNF n = concat
```

[3] In fact, this space leak was reported to the GHC team as a bug, `http://hackage.haskell.org/trac/ghc/ticket/2185`.

[4] Why don't all the sparks fizzle in the second version? In fact the runtime does manage to execute a few sparks while it is waiting for IO to happen.

```
                . (`using` parListWHNF)
                . map (`using` seqList)
                . chunk n
```

where `chunk :: Int -> [a] -> [[a]]` splits a list into chunks of length `n`.

A combinator that we find ourselves using often is `parBuffer`, which behaves like `parList` except that it does not traverse the whole list eagerly; it sparks a fixed number of elements initially, and then sparks subsequent elements as the list is consumed. This formulation works particularly well with programs that produce output as a lazy list, since it allows us to retain the constant-space property of the program while taking advantage of parallelism. The disadvantage is that we have to pick a buffer size, and the best choice of buffer size might well depend on how many CPUs we have available.

Our modified version of `parBuffer` that avoids space leaks is `parBufferWHNF`:

```
parBufferWHNF :: Int -> [a] -> [a]
parBufferWHNF n xs = return xs (start n xs)
  where
    return (x:xs) (y:ys) = y `par` (x : return xs ys)
    return xs [] = xs

    start !n [] = []
    start 0 ys = ys
    start !n (y:ys) = y `par` start (n-1) ys
```

7.3 Compositional strategies revisited

We can recover a more compositional approach to strategies by changing their type. The existing `Strategy` type is defined thus:

```
type Strategy a = a -> Done
```

Suppose that instead we define `Strategy` as a projection, like this:

```
type Strategy a = a -> a
```

then a `Strategy` can do some evaluation and sparking, and return a new `a`. In order to use this new kind of `Strategy` effectively, we need a new version of the `par` combinator:

```
spark :: Strategy a -> a -> (a -> b) -> b
spark strat a f = x `par` f x
  where x = strat a `pseq` a
```

The `spark` combinator takes a strategy `strat`, a value `a`, and a continuation `f`. It creates a spark to evaluate `strat a`, and then passes a new object to the continuation with the same value as `a`. When evaluated, this new object will cause the spark to fizzle and be discarded. Now we can recover compositional `parList` and `seqList` combinators:

```
parList :: Strategy a -> Strategy [a]
parList strat xs = foldr f [] xs
  where f x xs = spark strat x $ \x -> xs `pseq` x:xs

seqList :: Strategy a -> Strategy [a]
seqList strat xs = foldr seq ys ys
  where ys = map strat xs
```

and indeed this works quite nicely. Note that `parList` requires linear stack space; it is also possible to write a version that only requires linear heap space, but that requires two traversals of the list.

Here is `parListChunk` in the new style:

```
parListChunk :: Int -> Strategy a -> Strategy [a]
parListChunk n strat xs = ys `pseq` concat ys
  where ys = parList (seqList strat) $ chunk n xs
```

4 cores		7 cores	
Program	Δ Time (%)	Program	Δ Time (%)
gray	-2.1	gray	+7.5
mandel	-1.0	mandel	+1.8
matmult	-7.8	matmult	+4.4
parfib	-1.0	parfib	-8.6
partree	+5.2	partree	+9.5
prsa	-0.2	prsa	-0.1
ray	+1.5	ray	+0.2
sumeuler	-1.2	sumeuler	+1.7
Geom. Mean	-0.9	Geom. Mean	+1.9

Figure 11. The effect of eager black-holing

8. Thunks and black holes

Suppose that two Haskell threads, A and B, begin evaluation of a thunk t simultaneously. Semantically, it is acceptable for them both to evaluate t, since they will get the same answer (Harris et al. 2005); but operationally it is better to avoid this duplicated work. The obvious way to do so is to lock every thunk when starting evaluation, but that is expensive; measurements in the earlier cited work demonstrate an increase in execution time of around 50%. So we considered several variants that trade a reduced overhead against the risk of duplicated work:

EagerBH: immediately on entry, thread A overwrites t with a *black hole*. If thread B sees a black hole, it blocks until A performs the update (Section 8.1). The "window of vulnerability", in which a second thread might start a duplicate evaluation, is now just a few instructions wide. The cost compared to sequential execution is an extra memory store on every thunk entry.

RtsBH: enlists the runtime system, using the scheme described in Harris et al. (2005). The idea is to walk a thread's stack whenever it returns to the scheduler, and "claim" each of the thunks under evaluation using an atomic instruction. If a thread is found to be evaluating a thunk already claimed by another thread, then we suspend the current execution and put the thread to sleep until the evaluation is complete. Since every thread will return to the scheduler at regular intervals (say, to do garbage collection), this ensures that we cannot continue to evaluate the same thunk in multiple threads indefinitely. The overhead is much less than locking every thunk because most thunks are entered, evaluated, and updated during a single scheduler timeslice.

PostCheck: As Harris et al. (2005) points out, if two threads both succeed in *completing* the evaluation of the same thunk, and its value itself contains more thunks, there is a danger that an unbounded amount of work can be duplicated. The **PostCheck** strategy adds a test just before the update to check whether the thunk has already been updated by another thread. This test does not use an atomic instruction, but reduces the chance of further duplicate work taking place.

In our earlier work (Harris et al. 2005) we measured of the overheads of locking every thunk, but said nothing about the overheads or work-duplication of the other strategies.

GHC 6.10.1 implements **RtsBH** by default. Figure 11 shows the additional effect of **EagerBH** on our benchmark programs, for 4 and 7 cores. As you might expect, the effect is minor, because **RtsBH** catches almost all the cases that **EagerBH** does, except for very short-lived thunks which do not matter much anyhow. Figure 12 shows how many times **RtsBH** catches a duplicate computation in progress, both with and without adding **EagerBH**. As we can see, without **EagerBH** there are occasionally a substantial number of duplicate evaluations (eg. in **ray**), but **EagerBH** reduces

Program	RtsBH	RtsBH + EagerBH
gray	0	0
mandel	3	0
matmult	5	5
parfib	70	1
partree	3	3
prsa	45	0
ray	1261	0
sumeuler	0	0

Figure 12. The number of duplicate computations caught

4 cores	
Program	Δ Time (%)
gray	+2.9
mandel	+4.2
matmult	+42.1
parfib	-0.3
partree	+3.3
prsa	-0.5
ray	-5.4
sumeuler	-0.2
Geom. Mean	+5.0

7 cores	
Program	Δ Time (%)
gray	+2.9
mandel	+1.2
matmult	-0.0
parfib	+7.5
partree	+7.0
prsa	-1.5
ray	-14.4
sumeuler	+1.2
Geom. Mean	+0.3

Figure 13. Disabling thread migration

that number to almost zero. In **ray**, although we managed to eliminate a large number of duplicate evaluations using **EagerBH**, the effect on overall execution time was negligible: this program creates 10^6 tiny sparks, so 1200 duplicate evaluations has little impact. In fact, with the fine granularity in this benchmark, it may be that the cost of suspending the duplicate evaluation and blocking the thread outweighs the cost of just duplicating the computation.

To date we have not measured the effect of **PostCheck**. We expect it to have no effect on these benchmarks, especially in combination with **EagerBH**. However, we have experienced the effect of unbounded duplicate work in other programs; one good example where it can occur is in this version of `parMap`:

```
parMap :: (a -> b) -> [a] -> [b]
parMap f [] = []
parMap f (x:xs) = fx `par` (pmxs `par` (fx:pmxs))
   where fx = f x
         pmxs = parMap f xs
```

This function sparks both the head and the tail of the list, instead of traversing the whole list sparking each element as in the usual `parMap`. The duplication problem occurs if two threads evaluate the `pmxs` thunk: then the tail of the list is duplicated, possibly resulting in a large number of useless sparks being created.

8.1 Blocking on a black hole

When a thread A tries to evaluate a black hole, it must block until the thread currently evaluating the black hole (thread B) completes the evaluation, and overwrites the thunk with (an indirection to) its value. In earlier implementations (before 6.6) we arranged that thread B would attach its TSO to the thunk, so that thread A could re-awaken B when it performed the update. But that requires expensive synchronisation on every update, in case the thunk by now has a sleeping thread attached to it.

Since thunk updates are very common, but collisions (in which a sleeping thread attaches itself to a thunk) are very rare, GHC 6.10 instead optimises for the common case. Instead of attaching itself to the thunk, the blocked thread B simply polls the thunk, waiting for the update. Since a thunk can only be updated once, an update can therefore be performed without any synchronisation whatsoever, provided that writes are not re-ordered. Our earlier work (Harris et al. 2005) discusses these synchronisation issues in much more detail.

GHC 6.10 maintains a single, global Black Hole Pool, which the HECs poll when they are otherwise idle, and at least once per GC. We have considered two alternative designs: (a) privatising the Black Hole Pool to each HEC; and (b) using the thread scheduler directly, by making the blocked thread sleep and retry the evaluation when it reawakens. We have not yet measured these alternatives but, since contention is rare (Figure 12), they will probably only differ in extreme cases.

9. Load balancing and migration

In this section we discuss design choices concerning which HEC should run which Haskell threads.

9.1 Sharing runnable threads

In the current implementation, while we (now) use work-stealing for *sparks*, we use work-pushing for *threads*. That is, when a HEC detects that it has more than one thread in its Run Queue and there are other idle HECs, it distributes some of the local threads to the other HECs. The reason for this design is mostly historical; we could without much difficulty represent the Run Queue using a work-stealing queue and thereby use work-stealing for the load-balancing of threads.

We measured the effect that automatic thread migration has on the performance of our parallel benchmarks. Figure 13 shows the effect of disabling automatic thread migration, against a baseline of the PARGC3 configuration (Section 6.4). Since these are parallel, rather than concurrent, programs, the only way that multiple threads can exist on the Run Queue of a single CPU is when a thread becomes temporarily blocked (on a blackhole, Section 8.1), and then later becomes runnable again. As we can see from the results, often allowing migration makes no difference. Occasionally it is essential: for example **matmult** on 4 cores. And occasionally, as in **ray**, allowing migration leads to worse performance, possibly due to lost locality.

Whether to allow migration or not is a runtime flag, so the programmer can experiment with both settings to find the best one.

9.2 Migrating on wakeup

A blocked thread can be woken up for various reasons: if it is blocked on a black hole, it is woken up when some HEC notices that the black hole has now been evaluated (Section 8.1); if it is blocked on an empty `MVar`, then it can be unblocked when another thread performs a `putMVar` operation on that `MVar`.

When a thread is woken up, if it was previously running on another HEC, we have a choice: it can be placed on the Run Queue of the current HEC (hence migrating it), or we could arrange to awaken it on the HEC it was previously running on. In fact, we could wake it up on any HEC, but typically these two options are the most profitable.

Moving the thread to the current HEC might be advantageous if the thread is involved in a series of communications with another thread on this HEC: context-switching between two threads on the same HEC is particularly cheap. However, locality might also be important: the thread might be referencing data that is in the cache of the other HEC.

In GHC we take locality seriously, so our default is not to migrate awoken threads to the current CPU. For parallel programs, it is never worthwhile to change this setting, at least with the current implementation of black holes, since it is essentially random which

HEC awakens a blocked thread. If we were to change the implementation of black holes such that a thread can tell when an update should wake a blocked thread (perhaps by using a hash table to map the address of black holes to blocked threads), then there might be some benefit in migrating the blocked thread to the CPU on which the value it was waiting for resides.

10. Benchmarks and experimental setup

Our test system consists of 2 quad-core Intel Xeon(R) E5320 processors at 1.6GHz. Each pair of cores shares 4MB of L2 cache, and there is 16GB of system memory. The system was running Fedora 9. Although the OS was running in 64-bit mode, we used 32-bit binaries for our measurements (programs compiled for 64-bit tend to place more stress on the memory system and garbage collector resulting in less parallelism). In all cases we ran the programs five times and took the average wall-clock execution time.

Our benchmarks consist of a selection of small-to-medium-sized Parallel Haskell programs:

- **parfib**: the ubiquitous parallel fibonacci function, included here as a sanity test to ensure that our implementation is able to parallelise micro-benchmarks. The parallelism is divide-and-conquer-style, using explicit `par` and `pseq`.
- **sumeuler**: the sum of the value of Euler's function applied to each integer up to a given bound. This is a map/reduce style problem: applications of the Euler function can be performed in parallel, and the results must be summed (Trinder et al. 2002). The parallelism is expressed using `parListChunk` from the strategies library.
- **matmult**: A naive matrix-multiply algorithm. The matrix is represented as a `[[Int]]`. The parallelism is expressed using `parListChunk`.
- **ray**: A ray-tracer benchmark[5]. The parallelism is expressed using `parBuffer`, and is quite fine-grained (each pixel to be rendered is a separate spark).
- **gray**: Another ray-tracing benchmark, this time taken from an entry[6] in the ICFP'00 programming contest. Only the rendering part of the program has been parallelised, using a `parBuffer` as above. According to time profiling, the program only spends about 50% of its time in the renderer, so we expect this to limit the parallelism we can achieve. The parallelism is expressed using a single `parBuffer` in the renderer.
- **prsa**: A parallel RSA message encoder, encoding a 500KB message. Parallelism is again expressed using `parBuffer`.
- **partree**: A parallel map and fold over a tree. The program originates in the GUM benchmark suite, and in fact appears to be badly written: it is quadratic in the size of the tree. Nevertheless, it does appear to run in parallel, so we used the program unmodified for the purposes of benchmarking.
- **mandel**: this is a mandelbrot-set program originating in the `nofib` benchmark suite (Partain 1992). It generates a lazy list of pixel data (for a 1024x1024 scene), in a similar way to the ray tracer, and it was parallelised in the same way with the addition of `parBuffer`. The difference in this case is that the parallelism is more coarse-grained: each scan-line of the result is a separate spark.

[5] This program has a long history. According to comments in the source code, it was "taken from Paul Kelly's book, adapted by Greg Michaelson for SML, converted to (parallel) Haskell by Kevin Hammond".

[6] from the Galois team

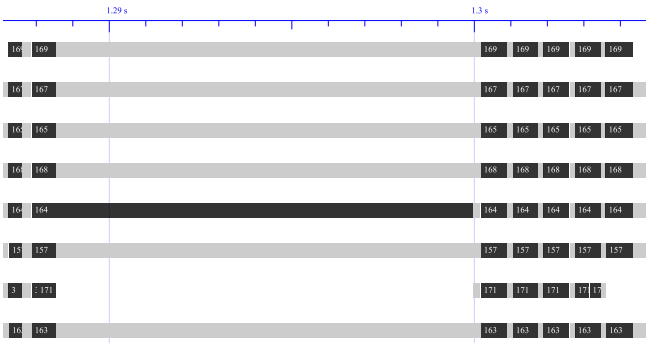

Figure 14. A slow synchronisation

These programs are all small, are mostly easy to parallelise, and are not highly optimised, so the results we report here should be interpreted as suggestive rather than conclusive. Nevertheless, our goal has not been to optimise the *programs*, but rather to optimise the *implementation* to make existing programs parallelise better. Furthermore, smaller benchmarks have their uses:

- Small benchmarks show up in high relief interesting differences in the behaviour of our runtime and execution model. These differences would be less marked had we used only large programs.
- We know that most of these programs *should* parallelise well, so any lack of parallelism is more likely to be as a result of choices made in the language implementation than in the program itself. Indeed, the lack of linear speedup in Figure 1 shows that we still have plenty of room for improvement.

10.1 Profiling

To help understand the behaviour of our benchmark programs we developed a graphical viewer called ThreadScope for event information generated by the runtime system. The viewer is modeled after circuit waveform viewers with a profile drawn with time on the x-axis and HEC number on the y-axis. In each HEC's timeline, the bar is coloured solid black when the HEC is running Haskell code with the thread ID inside the box, and gray when it is garbage collecting. Events, such as threads blocking or being woken up, are indicated by labels annotating the timeline when the view is zoomed enough.

This visualisation of the execution is immensely useful for being able to quickly identify problem areas. For example, when we ran our benchmarks on all 8 cores of our 8-core machine, we experienced inexplicable drops in performance. Figure 14 shows one problem area as seen in the profile viewer. In the middle of the picture there is a long period where one HEC has initiated a GC, and is waiting for the other HECs to stop. The initiating/waiting HEC has a white bar, the HECs that have already stopped and are ready to GC are shown in gray. One HEC is running (black with thread ID 164) during this period, indicating that it is apparently running Haskell code and has not responded to the call for a GC. In fact, the OS thread running this HEC has been descheduled by the OS, so does not respond for a relatively long period. The same pattern repeats many times during the execution, having a significant impact on the overall runtime.

This experience does illustrate that our runtime is particularly sensitive to problems such as this due to the relatively high frequency of full synchronisations needed for GC, and that tackling independent GC (Section 12.1) should be a high priority.

11. Related work

The design space for language features to support implict parallelism and the underlying run-time system is very large. Here we identify just a few systems that make different design decisions and trade-offs from the GHC run-time system.

Like GHC the Manticore (Fluet et al. 2008b,a) system also supports implicit and explicit fine-grained parallelism which in turn has been influenced by previous work on data parallel languages like NESL (Blelloch et al. 1994) and Nepal/DPH (Chakravarty et al. 2001). Unlike NESL or Nepal/DPH, GHC also implements support for explicit concurrency as does Manticore. Many of the underlying implementation choices made for GHC and Manticore are interchangeable e.g. Manticore uses a partially shared heap whereas GHC uses a totally shared heap. Manticore however presents quite a different programming model based on parallel data structures (e.g. tuples and arrays) which provide a fork-join pattern of computation as well as a parallel case expression which can introduce non-determinism. Neither GHC nor Manticore support implicit parallelism without the need for user annotations which has been implemented in other functional languages like Id (Nikhl 1991), pH (Nikhl and Arvind 2001) and Sisal (Gaudiot et al. 1997).

STING (Jagannathan and Philbin 1992) is a system that supports multiple parallel language constructs for a dialect of SCHEME through three layers of process abstraction as well as special support for specifying scheudling policies. We also intend to modify GHC's infrasturcture to allow different scheduling policies to be composed together in a flexible manner (Li et al. 2007a).

GpH (Trinder et al. 1998) extended Haskell98 to introduce the parallel (`par`) and sequential (`seq`) coordination primitives and provides strategies for controlling the evaluation order. Unlike semi-implicit parallelism annotations in Haskell which identify opportunities for parallelsim, in Eden (Loogen et al. 2005) one explicitly creates processes which are always executed concurrently. GUM (Trinder et al. 1996) targets distributed systems and are based on message passing.

In our work we profiled the execution time of Haskell threads and garbage collection. However, we will also need to perform space profiling and the work on the MLton project on semantic space profiling (Spoonhower et al. 2008) represents an interesting approach for a strict language.

Erlang (Armstrong et al. 1996) provides isolated threads which communicate through a mailbox mechanism with pattern matching used to select messages of interest. These design decisions have a subtantial effect on the design of the run-time system. Eden (Loogen et al. 2005) provides special annontations to control parallel evaluation of processes.

Cilk (Blumofe et al. 2001) is an imperative programming language based on C which also supports fine grain parallelism in a fork-join manner by spawning off parallel invocations of procedures. Like GHC Cilk also performs work-stealing for load balancing. The spawn feature of Cilk, expressions bound with `pcase` in Manticore and sparks in GHC can all be considered to be instances of futures.

12. Conclusion and future work

While we have achieved some significant improvements in parallel efficiency, our work clearly has some way to go; several benchmarks do not speed up as much as we might hope. Our focus in the future will therefore continue to be on using profiling tools to identify problem areas, and using those results to direct our attention to appropriate areas of the runtime system and execution model.

The work on implicit parallelization described in Harris and Singh (2007) may benefit from the recent changes to the GHC run-time and we are considering re-running the benchmarks to measure any improvements. In particular we expect benchmarks that perform a lot of garbage collection to benefit from the parallel garbage collector.

It is clear from our investigation of the programming model in Section 7 that we should change the GC policy for sparks from ROOT to WEAK, but we must also revisit the Strategies abstraction and develop a new library that works effectively under WEAK.

12.1 Independent GC

Stop-the-world GC will inevitably become more of a bottleneck as the number of cores increases. There are known techniques for doing CPU-independent GC (Doligez and Leroy 1993), and these techniques are used in systems such as Manticore (Fluet et al. 2008a).

We fully intend to pursue CPU-independent GC in the future. However this is unlikely to be an easy transition. CPU-independent GC replaces direct sharing by physical separation and explicit communication. This leads to trade-offs; it isn't a straightforward win. More specifically, CPU-independent GC requires a local-heap invariant, namely that there are no pointers between local heaps, or from the global heap into any local heap. Ensuring and maintaining this invariant introduces new costs and complexities into the runtime execution model.

On the other hand, as the number of cores in modern CPUs increases, the illusion of shared memory begins to break down. We are already experiencing severe penalties for losing locality (Section 6.4), and it is likely that these will only get worse in the future. Hence, moving to more explicitly-separate heap regions is a more honest reflection of the underlying memory architecture, and is likely to allow the implementation to make intelligent decisions about data locality.

Acknowledgments

We wish to thank Jost Berthold, who helped us identify some of the bottlenecks in the parallel runtime implementation, and built the first implementation of work-stealing queues for spark distribution during an internship at Microsoft Research in the summer of 2008. We also wish to thank Phil Trinder and Tim Harris for their helpful comments on an earlier draft of this paper.

References

J. R. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent programming in ERLANG (2nd ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1996.

Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *In Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA), Puerto Vallarta*, pages 119–129, 1998.

J. Berthold, S. Marlow, A. Al Zain, and K. Hammond. Comparing and optimising parallel Haskell implementations on multicore. In *IFL'08: International Symposium on Implementation and Application of Functional Languages (Draft Proceedings)*, Hatfield, UK, 2008.

Stephen M. Blackburn and Antony L. Hosking. Barriers: friend or foe? In *ISMM '04: Proceedings of the 4th international symposium on Memory management*, pages 143–151, New York, NY, USA, 2004. ACM. ISBN 1-58113-945-4. doi: http://doi.acm.org/10.1145/1029873.1029891.

G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M Zagha. Implementation of a portable nested data-parallel language. *JDPC*, 21(1):4–14, 1994.

R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, and C. E. Lieserson. Cilk: an efficient multithreaded runtime system. *SIGNPLAN Not.*, 30(8):207–216, 2001.

M. T. Chakravarty, G. Keller, R. Leshinskiy, and W. Pfannenstiel. Nepal - Nested Data Parallelism in Haskell. *LNCS*, 2150, Aug 2001.

David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *SPAA '05: Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 21–28, New York, NY, USA, 2005. ACM. ISBN 1-58113-986-1. doi: http://doi.acm.org/10.1145/1073970.1073974.

Damien Doligez and Xavier Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ml. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 113–123, New York, NY, USA, 1993. ACM. ISBN 0-89791-560-7. doi: http://doi.acm.org/10.1145/158511.158611.

Christine Flood, Dave Detlefs, Nir Shavit, and Catherine Zhang. Parallel garbage collection for shared memory multiprocessors. In *Usenix Java Virtual Machine Research and Technology Symposium (JVM '01)*, Monterey, CA, 2001. URL `citeseer.ist.psu.edu/flood01parallel.html`.

Matthew Fluet, Mike Rainey, and John Reppy. A scheduling framework for general-purpose parallel languages. *SIGPLAN Not.*, 43 (9):241–252, 2008a. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1411203.1411239.

Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. Implicitly-threaded Parallelism in Manticore. *International Conference on Functional Programming*, pages 119–130, 2008b.

J. L. Gaudiot, T. DeBoni, J. Feo, W. Bohm, W. Najjar, and P. Miller. The Sisal model of functional programming and its implementation. In *As '97*, pages 112–123, Los Altimos, CA, March 1997. IEEE Computer Society Press.

Tim Harris and Satnam Singh. Feedback directed implicit parallelism. In *ICFP '07: Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, pages 251–264, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-815-2. doi: http://doi.acm.org/10.1145/1291151.1291192.

Tim Harris, Simon Marlow, and Simon Peyton Jones. Haskell on a shared-memory multiprocessor. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 49–61. ACM Press, September 2005. ISBN 1-59593-071-X. doi: http://doi.acm.org/10.1145/1088348.1088354. URL `http://www.haskell.org/~simonmar/papers/multiproc.pdf`.

S. Jagannathan and J. Philbin. A customizable substrate for concurrent languages. In *ACM Conference on Programming Languages Design and Implementation (PLDI'92)*, pages 55–81. ACM Press, June 1992.

P. Li, Simon Marlow, Simon Peyton Jones, and A. Tolmach. Lightweight concurrency primitives for GHC. In *Haskell '07: Proceedings of the 2007 ACM SIGPLAN workshop on Haskell*, pages 107–118. ACM Press, September 2007a.

Peng Li, Simon Marlow, Simon Peyton Jones, and Andrew Tolmach. Lightweight concurrency primitives for GHC. Haskell '07: Proceedings of the ACM SIGPLAN workshop on Haskell workshop, June 2007b. URL `http://www.haskell.org/~simonmar/papers/conc-substrate.pdf`.

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. URL `http://www.cee.hw.ac.uk/\~{}dsg/gph/papers/ps/cpe.ps.gz`.

H.-W. Loidl, F. Rubio, N. Scaife, K. Hammond, S. Horiguchi, U. Klusik, R. Loogen, G. J. Michaelson, R. Pe na, S. Priebe, Á J. Rebón, and P. W. Trinder. Comparing parallel functional languages: Programming and performance. *Higher Order Symbol. Comput.*, 16(3):203–251, 2003. ISSN 1388-3690. doi: http://dx.doi.org/10.1023/A:1025641323400.

Rita Loogen, Yolanda Ortega-Mallén, and Ricardo Pena. Parallel Functional Programming in Eden. *Journal of Functional Programming*, 15(3):431–475, 2005.

Simon Marlow, Simon Peyton Jones, and Wolfgang Thaller. Extending the haskell foreign function interface with concurrency. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 57–68, Snowbird, Utah, USA, September 2004. URL `http://www.haskell.org/~simonmar/papers/conc-ffi.pdf`.

Simon Marlow, Tim Harris, Roshan P. James, and Simon Peyton Jones. Parallel generational-copying garbage collection with a block-structured heap. In *ISMM '08: Proceedings of the 7th international symposium on Memory management*. ACM, June 2008. URL `http://www.haskell.org/~simonmar/papers/parallel-gc.pdf`.

R. S. Nikhl. ID language reference manual. *Laboratory for Computer Science, MIT*, Jul 1991.

R. S. Nikhl and Arvind. *Implicit Parallel Programming in pH*. Morgan Kaufmann Publishers, San Francisco, CA, 2001.

WD Partain. The `nofib` benchmark suite of Haskell programs. In *Functional Programming, Glasgow 1992*, Workshops in Computing, pages 195–202. Springer Verlag, 1992.

S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proc. of POPL'96*, pages 295–308. ACM Press, 1996.

Simon Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M. T. Chakravarty. Harnessing the multicores: Nested data parallelism in Haskell. In *FSTTCS 2009: IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, 2009.

Daniel Spoonhower, Guy E. Blelloch, Robert Harper, and Phillip B. Gibbons. Space profiling for parallel functional programs. In *ICFP '08: Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, pages 253–264, New York, NY, USA, 2008. ACM.

P. W. Trinder, H.-W. Loidl, and R. F. Pointon. Parallel and distributed Haskells. *J. Funct. Program.*, 12(5):469–510, 2002. ISSN 0956-7968. doi: http://dx.doi.org/10.1017/S0956796802004343.

PW Trinder, K Hammond, JS Mattson, AS Partridge, and SL Peyton Jones. GUM: a portable parallel implementation of haskell. In *ACM Conference on Programming Languages Design and Implementation (PLDI'96)*. ACM Press, Philadelphia, May 1996.

PW Trinder, K Hammond, H-W Loidl, and SL Peyton Jones. Algorithm + strategy = parallelism. *Journal of Functional Programming*, 8:23–60, January 1998.