Parallel Performance Tuning for Haskell

Don Jones Jr.
University of Kentucky
donnie@darthik.com

Simon Marlow
Microsoft Research
simonmar@microsoft.com

Satnam Singh
Microsoft Research
satnams@microsoft.com

Abstract

Parallel Haskell programming has entered the mainstream with support now included in GHC for multiple parallel programming models, along with multicore execution support in the runtime. However, tuning programs for parallelism is still something of a black art. Without much in the way of feedback provided by the runtime system, it is a matter of trial and error combined with experience to achieve good parallel speedups.

This paper describes an early prototype of a parallel profiling system for multicore programming with GHC. The system comprises three parts: fast event tracing in the runtime, a Haskell library for reading the resulting trace files, and a number of tools built on this library for presenting the information to the programmer. We focus on one tool in particular, a graphical timeline browser called ThreadScope.

The paper illustrates the use of ThreadScope through a number of case studies, and describes some useful methodologies for parallelizing Haskell programs.

1. Introduction

Life has never been better for the Parallel Haskell programmer: GHC supports multicore execution out of the box, including multiple parallel programming models: Strategies (Trinder et al. 1998), Concurrent Haskell (Peyton Jones et al. 1996) with STM (Harris et al. 2005), and Data Parallel Haskell (Peyton Jones et al. 2008). Performance of the runtime system has received attention recently, with significant improvements in parallel performance available in the forthcoming GHC release (Marlow et al. 2009). Many of the runtime bottlenecks that hampered parallel performance in earlier GHC versions are much reduced, with the result that it should now be easier to achieve parallel speedups.

However, optimizing the runtime only addresses half of the problem; the other half being how to tune a given Haskell program to run effectively in parallel. The programmer still has control over task granularity, data dependencies, speculation, and to some extent evaluation order. Getting these wrong can be disastrous for parallel performance. For example, the granularity should neither be too fine nor too coarse. Too coarse and the runtime will not be able to effectively load-balance to keep all CPUs constantly busy; too fine and the costs of creating and scheduling the tiny tasks outweigh the benefits of executing them in parallel.

Current methods for tuning parallel Haskell programs rely largely on trial and error, experience, and an eye for understanding the limited statistics produced at the end of a program's run by the runtime system. What we need are effective ways to measure and collect information about the runtime behaviour of parallel Haskell programs, and tools to communicate this information to the programmer in a way that they can understand and use to solve performance problems with their programs.

In this paper we describe a new profiling system developed for the purposes of understanding the parallel execution of Haskell programs. In particular, our system includes a tool called ThreadScope that allows the programmer to interactively browse the parallel execution profile.

This paper contributes the following:

- We describe the design of our parallel profiling system, and the ThreadScope tool for understanding parallel execution. Our trace file format is fully extensible, and profiling tools built using our framework are both backwards- and forward-compatible with different versions of GHC.
- Through several case studies, we explore how to use ThreadScope for identifying parallel performance problems, and describe a selection of methodologies for parallelising Haskell code.

Earlier methodologies for parallelising Haskell code exist (Trinder et al. 1998), but there are two crucial differences in the multicore GHC setting. Firstly, the trade-offs are likely to be different, since we are working with a shared-memory heap, and communication is therefore cheap[1]. Secondly, it has recently been discovered that Strategies interact badly with garbage collection (Marlow et al. 2009), so in this paper we avoid the use of the original Strategies library, relying instead on our own simple hand-rolled parallel combinators.

Our work is at an early stage. The ThreadScope tool displays only one particular view of the execution of Parallel Haskell programs (albeit a very useful one). There are a wealth of possibilities, both for improving ThreadScope itself and for building new tools. We cover some of the possibilities in Section 6.

2. Profiling Motivation

Haskell provides a mechanism to allow the user to control the granularity of parallelism by indicating what computations may be usefully carried out in parallel. This is done by using functions from the Control.Parallel module. The interface for Control.Parallel is shown below:

```
par  :: a → b → b
pseq :: a → b → b
```

[1] though not entirely free, since memory cache hierarchies mean data still has to be shuffled between processors even if that shuffling is not explicitly programmed.

The function par indicates to the GHC run-time system that it may be beneficial to evaluate the first argument in parallel with the second argument. The par function returns as its result the value of the second argument. One can always eliminate par from a program by using the following identity without altering the semantics of the program:

```
  par a b = b
```

A thread is not necessarily created to compute the value of the expression a. Instead, the GHC run-time system creates a *spark* which has the potential to be executed on a different thread from the parent thread. A sparked computation expresses the possibility of performing some speculative evaluation. Since a thread is not necessarily created to compute the value of a, this approach has some similarities with the notion of a *lazy future* (Mohr et al. 1991).

We call such programs semi-explicitly parallel because the programmer has provided a hint about the appropriate level of granularity for parallel operations and the system implicitly creates threads to implement the concurrency. The user does not need to explicitly create any threads or write any code for inter-thread communication or synchronization.

To illustrate the use of par we present a program that performs two compute intensive functions in parallel. The first compute intensive function we use is the notorious Fibonacci function:

```
fib :: Int → Int
fib 0 = 0
fib 1 = 1
fib n = fib (n−1) + fib (n−2)
```

The second compute intensive function we use is the sumEuler function taken from (Trinder et al. 2002):

```
mkList :: Int → [Int]
mkList n = [1..n−1]

relprime :: Int → Int → Bool
relprime x y = gcd x y == 1

euler :: Int → Int
euler n = length (filter (relprime n) (mkList n))

sumEuler :: Int → Int
sumEuler = sum . (map euler) . mkList
```

The function that we wish to parallelize adds the results of calling fib and sumEuler:

```
sumFibEuler :: Int → Int → Int
sumFibEuler a b = fib a + sumEuler b
```

As a first attempt we can try to use par to speculatively spark off the computation of fib while the parent thread works on sumEuler:

```
−− A wrong way to parallelize f + e
parSumFibEuler :: Int → Int → Int
parSumFibEuler a b
  = f ‘par‘ (f + e)
    where
    f = fib a
    e = sumEuler b
```

To create two workloads that take roughly the same amount of time to execute we performed some experiments which show that fib 38 takes roughly the same time to execute as sumEuler 5300. The execution trace for this program as displayed by ThreadScope is shown in Figure 1. This figure shows the execution trace of two Haskell Execution Contexts (HECs), where each HEC corresponds to a processor core. The x-axis is time. The purple portion of each line shows at what time intervals a thread is running and the orange (lighter coloured) bar shows when garbage collection is occurring. Garbage collections are always "stop the world", in that all Haskell threads must stop during GC, but a GC may be performed either sequentially on one HEC or in parallel on multiple HECs; in Figure 1 we are using parallel GC.

We can examine the statistics produced by the runtime system (using the flags +RTS -s -RTS) to help understand what went wrong:

```
SPARKS: 1 (0 converted, 0 pruned)

INIT  time    0.00s  (  0.00s elapsed)
MUT   time    9.39s  (  9.61s elapsed)
GC    time    0.37s  (  0.24s elapsed)
EXIT  time    0.00s  (  0.00s elapsed)
Total time    9.77s  (  9.85s elapsed)
```

The log shows that although a single spark was created, no sparks where "converted", i.e. executed. In this case the performance bug is because the main thread immediately starts to work on the evaluation of fib 38 itself which causes this spark to *fizzle*. A fizzled spark is one that is found to be under evaluation or already evaluated, so there is no profit in evaluating it in parallel. The log also shows that the total amount of computation work done is 9.39 seconds (the MUT time); the time spent performing garbage collection was 0.37 seconds (the GC time); and the total amount of work done amounts to 9.77 seconds with 9.85 seconds of wall clock time. A profitably parallel program will have a wall clock time (elapsed time) which is less than the total time[2].

One might be tempted to fix this problem by swapping the arguments to the + operator in the hope that the main thread will work on sumEuler while the sparked thread works on fib:

```
−− Maybe a lucky parallelization
parSumFibEuler :: Int → Int → Int
parSumFibEuler a b
  = f ‘par‘ (e + f)
    where
    f = fib a
    e = sumEuler b
```

This results in the execution trace shown in Figure 2 which shows a sparked thread being taken up by a spare worker thread.

The execution log for this program shows that a spark was used productively and the elapsed time has dropped from 9.85s to 5.33s:

```
SPARKS: 1 (1 converted, 0 pruned)

INIT  time    0.00s  (  0.00s elapsed)
MUT   time    9.47s  (  4.91s elapsed)
GC    time    0.69s  (  0.42s elapsed)
EXIT  time    0.00s  (  0.00s elapsed)
Total time   10.16s  (  5.33s elapsed)
```

While this trick works, it only works by accident. There is no fixed evaluation order for the arguments to +, and GHC might decide to use a different evaluation order tomorrow. To make the parallelism more robust, we need to be explicit about the evaluation order we intend. The way to do this is to use pseq[3] in combination with par, the idea being to ensure that the main thread works on sumEuler while the sparked thread works on fib:

```
−− A correct parallelization that does not depend on
```

[2] although to measure actual parallel speedup, the wall-clock time for the parallel execution should be compared to the wall-clock time for the sequential execution.

[3] Previous work has used seq for sequential evaluation ordering, but there is a subtle difference between Haskell's seq and the operator we need for sequencing here. The details are described in Marlow et al. (2009).

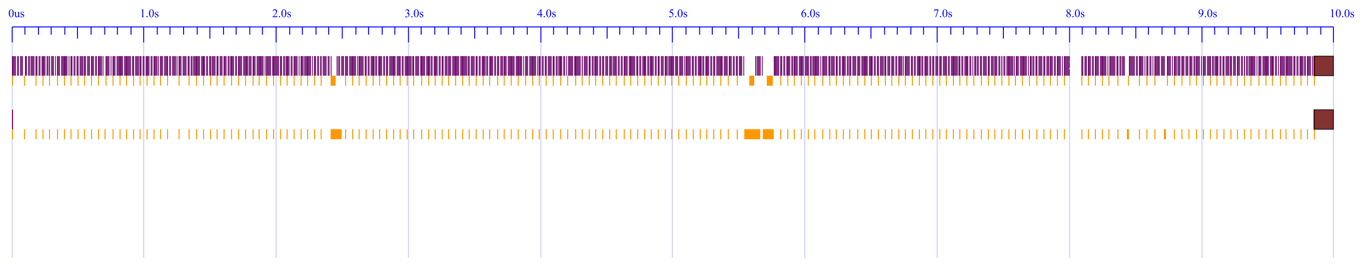


Figure 1. No parallelization of f 'par' (f + e)

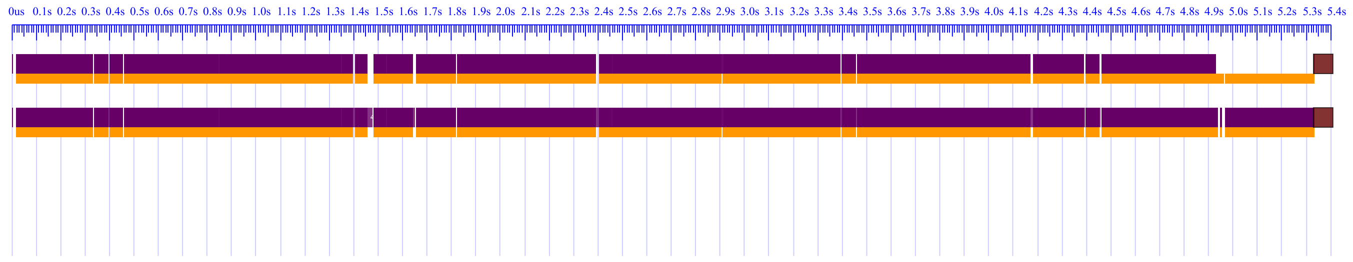


Figure 2. A lucky parallelization of f 'par' (e + f)

```
-- the evaluation order of +
parSumFibEuler :: Int → Int → Int
parSumFibEuler a b
  = f 'par' (e 'pseq' (f + e))
    where
    f = fib a
    e = sumEuler b
```

This version does not make any assumptions about the evaluation order of +, but relies only on the evaluation order of pseq, which is guaranteed to be stable.

This example as well as our wider experience of attempting to write semi-explicit parallel programs shows that it is often very difficult to understand if and when opportunities for parallelism expressed through par are effectively taken up and to also understand how operations like garbage collection influence the performance of the program. Until recently one only had available high level summary information about the overall execution of a parallel Haskell program. In this paper we describe recent improvements to the Haskell run-time which allow a much more detailed profile to be generated which can then be used to help debug performance problems.

3. Case Studies

3.1 Batcher's Bitonic Parallel Sorter

Batcher's bitonic merger and sorter is a parallel sorting algorithm which has a good implementation in hardware. We have produced an implementation of this algorithm in Haskell originally for circuit generation for FPGAs. However, this executable model also represents an interesting software implicit parallelization exercise because the entire parallel structure of the algorithm is expressed in terms of just one combinator called par2:

```
par2 :: (a → b) → (c → d) → (a, c) → (b, d)
par2 circuit1 circuit2 (input1, input2)
  = (output1, output2)
    where
    output1 = circuit1 input1
    output2 = circuit2 input2
```

This combinator captures the idea of two circuits which are independent and execute in parallel. This combinator is used to define other combinators which express different ways of performing parallel divide and conquer operations:

```
two :: ([a] → [b]) → [a] → [b]
two r = halve >→ par2 r r >→ unhalve

ilv :: ([a] → [b]) → [a] → [b]
ilv r = unriffle >→ two r >→ riffle
```

The halve combinator breaks a list into two sub-lists of even length and the unhalve operate performs the inverse operation. The riffile combinator permutes its inputs by breaking a list into two halves and then interleaving the resulting lists. unriffle performs the inverse permutation.

These combinators are in turn used to define a butterfly parallel processing network which describes a merger:

```
butterfly circuit [x,y] = circuit [x,y]
butterfly circuit input
  = (ilv (butterfly circuit) >→ evens circuit) input
```

The evens combinator breaks an input list into adjacent groups of two elements and applies the circuit argument to each group. A column of par-wise processing elements is used to combine the results of two sub-merges:

```
evens :: ([a] → [b]) → [a] → [b]
evens f = chop 2 >→ map f >→ concat
```

The chop 2 combinator breaks a list into sub-lists of length 2. This parallel Batcher's bitonic merger plus the evens function can be used to build a parallel Batcher's bitonic sorter:

```
sortB cmp [x, y] = cmp [x, y]
sortB cmp input
  = (two (sortB cmp) >→ sndList reverse >→ butterfly cmp) input
```

The sndList combinator breaks a list into two halves and applies its argument circuit to the top halve and the identity function to the bottom halve and then concatenates the sub-results into a single list.

A straightforward way to perform a semi-explicit parallelization of the par2 combinator is use par to spark off the evaluation of one of the sub-circuits.

```
par2 :: (a → b) → (c → d) → (a, c) → (b, d)
par2 circuit1 circuit2 (input1, input2)
  = output1 ‘par‘ (output2 ‘pseq‘ (output1, output2))
    where
    output1 = circuit1 input1
    output2 = circuit2 input2
```

This relatively simple change results in a definite performance gain due to parallelism. Here is the log output produced by running a test-bench program with just one Haskell execution context:

```
.\bsortpar.exe +RTS -N1 -l -qg0 -qb -sbsortpar-N1.log
  SPARKS: 106496 (0 converted, 106496 pruned)

  INIT  time    0.00s  (  0.00s elapsed)
  MUT   time    5.32s  (  5.37s elapsed)
  GC    time    0.72s  (  0.74s elapsed)
  EXIT  time    0.00s  (  0.00s elapsed)
  Total time    6.04s  (  6.12s elapsed)
```

Although many sparks are created none are taken up because there is only one worker thread. The execution trace for this invocation is shown in Figure 3.

Running with two threads shows a very good performance improvement:

```
.\bsortpar.exe +RTS -N2 -l -qg0 -qb -sbsortpar-N2.log
  SPARKS: 106859 (49 converted, 106537 pruned)

  INIT  time    0.00s  (  0.00s elapsed)
  MUT   time    4.73s  (  3.03s elapsed)
  GC    time    1.64s  (  0.72s elapsed)
  EXIT  time    0.00s  (  0.00s elapsed)
  Total time    6.36s  (  3.75s elapsed)
```

This example produces very many sparks most of which fizzle but enough sparks are turned into productive work i.e. 6.36 seconds worth of work done in 3.75 seconds of time. The execution trace for this invocation is shown in Figure 4. There is an obvious sequential block of execution between 2.1 seconds and 2.9 seconds and this is due to a sequential component of the algorithm which combines the results of parallel sub-computations i.e the evens function. We can use the parallel strategies library to change the sequential application in the definition of evens to a parallel map operation:

```
evens :: ([a] → [b]) → [a] → [b]
evens f = chop 2 >→  parMap rwhnf f >→  concat
```

This results in many more sparks being converted:

```
.\bsortpar2.exe +RTS -N2 -l -qg0 -qb -sbsortpar2-N2.log
  SPARKS: 852737 (91128 converted, 10175 pruned)

  INIT  time    0.00s  (  0.04s elapsed)
  MUT   time    4.95s  (  3.86s elapsed)
  GC    time    1.29s  (  0.65s elapsed)
  EXIT  time    0.00s  (  0.00s elapsed)
  Total time    6.24s  (  4.55s elapsed)
```

3.2 Soda

Soda is a program for solving word-search problems: given a rectangular grid of letters, find occurrences of a word from a supplied list, where a word can appear horizontally, vertically, or diagonally, in either direction (giving a total of eight possible orientations).

The program has a long history as a Parallel Haskell benchmark (Runciman and Wakeling 1993). The version we start with here is a recent incarnation, using a random initial grid with a tunable size. The words do not in fact appear in the grid; the program just fruitlessly searches the entire grid for a predefined list of words. One advantage of this formulation for benchmark purposes is that the program's performance does not depend on the search order, however a disadvantage is that the parallel structure is unrealistically regular.

The parallelism is expressed using parListWHNF to avoid the space leak issues with the standard strategy implementation of parList (Marlow et al. 2009). The parListWHNF function is straightforwardly defined thus:

```
parListWHNF :: [a] -> ()
parListWHNF [] = ()
parListWHNF (x:xs) = x ‘par‘ parListWHNF xs
```

To establish the baseline performance, we run the program using GHC's `+RTS -s` flags, below is an excerpt of the output:

```
  SPARKS: 12 (12 converted, 0 pruned)

  INIT  time    0.00s  (  0.00s elapsed)
  MUT   time    7.27s  (  7.28s elapsed)
  GC    time    0.61s  (  0.72s elapsed)
  EXIT  time    0.00s  (  0.00s elapsed)
  Total time    7.88s  (  8.00s elapsed)
```

We can see that there are only 12 sparks generated by this program: in fact the program creates one spark per word in the search list, of which there are 12. This rather coarse granularity will certainly limit the ability of the runtime to effectively load-balance as we increase the number of cores, but that won't be an issue with a small number of cores.

Initially we try with 4 cores, and with GHC's parallel GC enabled:

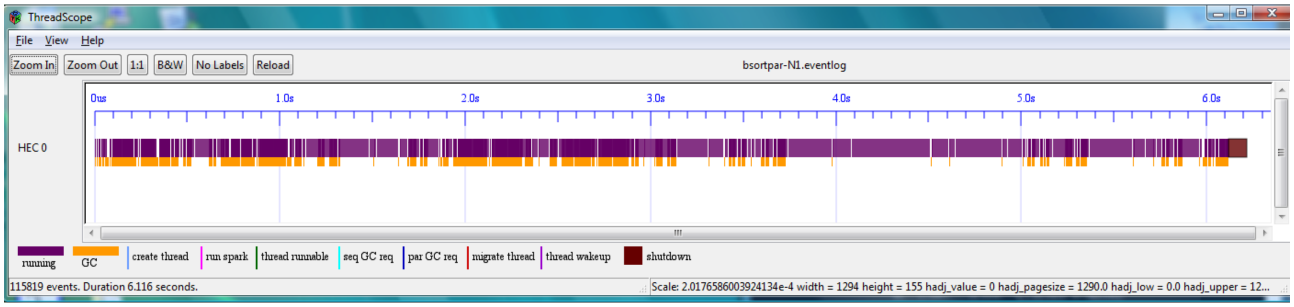
```
  SPARKS: 12 (11 converted, 0 pruned)

  INIT  time    0.00s  (  0.00s elapsed)
  MUT   time    8.15s  (  2.21s elapsed)
  GC    time    4.50s  (  1.17s elapsed)
  EXIT  time    0.00s  (  0.00s elapsed)
  Total time   12.65s  (  3.38s elapsed)
```

Not bad: 8.00/3.38 is a speedup of around 2.4 on 4 cores. But since this program has a highly parallel structure, we might hope to do better.

Figure 5 shows the ThreadScope profile for this version of soda. We can see that while an overall view of the runtime shows a reasonable parallelization, if we zoom into the initial part of the run (Figure 6) we can see that HEC 0 is running continuously, but threads on the other HECs are running very briefly and then immediately getting blocked (zooming in further would show the individual events).

Going back to the program, we can see that the grid of letters is generated lazily by a function mk_grid. What is happening here is that the main thread creates sparks before the grid has been evaluated, and then proceeds to evaluate the grid. As each spark runs, it blocks almost immediately waiting for the main thread to complete evaluation of the grid.

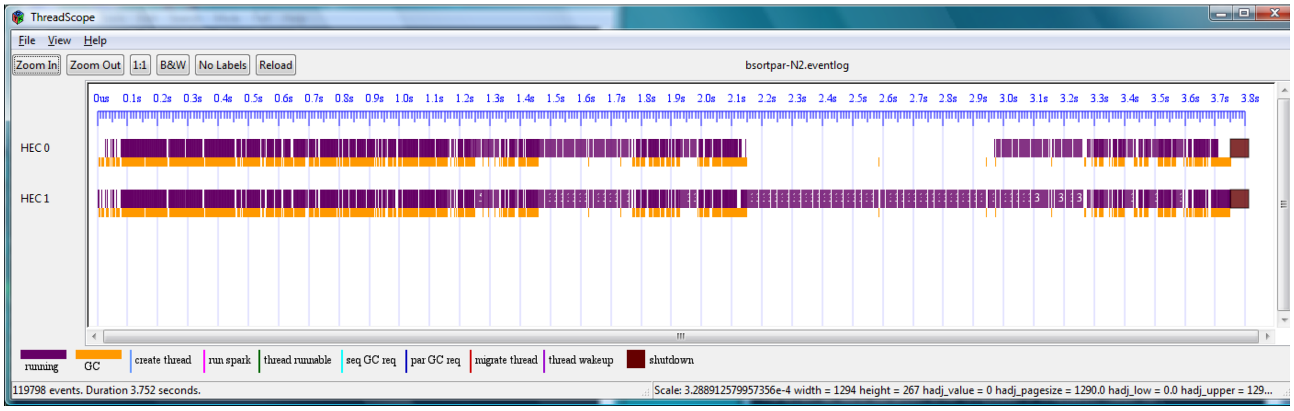
This type of blocking is often not disastrous, since a thread will become unblocked soon after the thunk on which it is blocking is evaluated (see the discussion of "blackholes" in Marlow et al. (2009)). There is nevertheless a short delay between the thread becoming runnable again and the runtime noticing this and moving the thread to the run queue. Sometimes this delay can be hidden if the program has other sparks it can run in the meantime, but that


ThreadScope
File View Help
Zoom In Zoom Out 1:1 B&W No Labels Reload
bsortpar-N1.eventlog
0us 1.0s 2.0s 3.0s 4.0s 5.0s 6.0s
HEC 0
running GC create thread run spark thread runnable seq GC req par GC req migrate thread thread wakeup shutdown

115819 events. Duration 6.116 seconds.

Scale: 2.0176586003924134e-4 width = 1294 height = 155 hadj_value = 0 hadj_pagesize = 1290.0 hadj_low = 0.0 hadj_upper = 12...

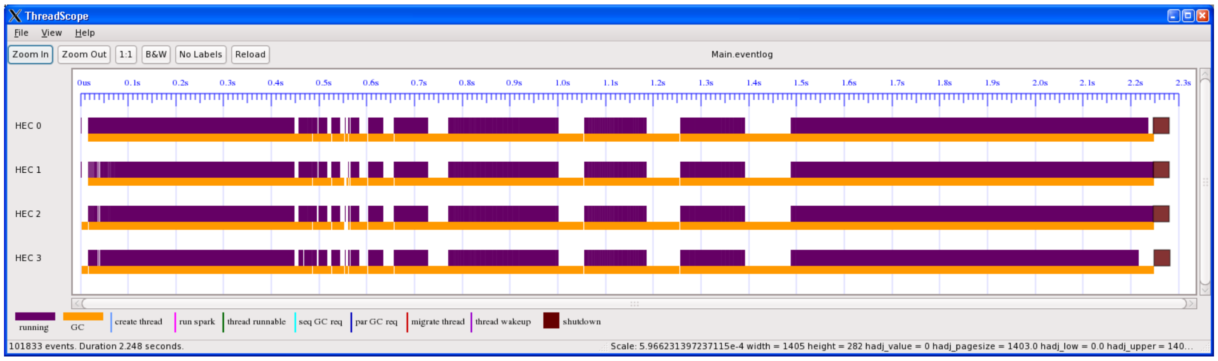
Figure 3. A sequential execution of bsort


ThreadScope
File View Help
Zoom In Zoom Out 1:1 B&W No Labels Reload
bsortpar-N2.eventlog
0us 0.1s 0.2s 0.3s 0.4s 0.5s 0.6s 0.7s 0.8s 0.9s 1.0s 1.1s 1.2s 1.3s 1.4s 1.5s 1.6s 1.7s 1.8s 1.9s 2.0s 2.1s 2.2s 2.3s 2.4s 2.5s 2.6s 2.7s 2.8s 2.9s 3.0s 3.1s 3.2s 3.3s 3.4s 3.5s 3.6s 3.7s 3.8s
HEC 0
HEC 1
running GC create thread run spark thread runnable seq GC req par GC req migrate thread thread wakeup shutdown

119798 events. Duration 3.752 seconds.

Scale: 3.288912579957356e-4 width = 1294 height = 267 hadj_value = 0 hadj_pagesize = 1290.0 hadj_low = 0.0 hadj_upper = 129...

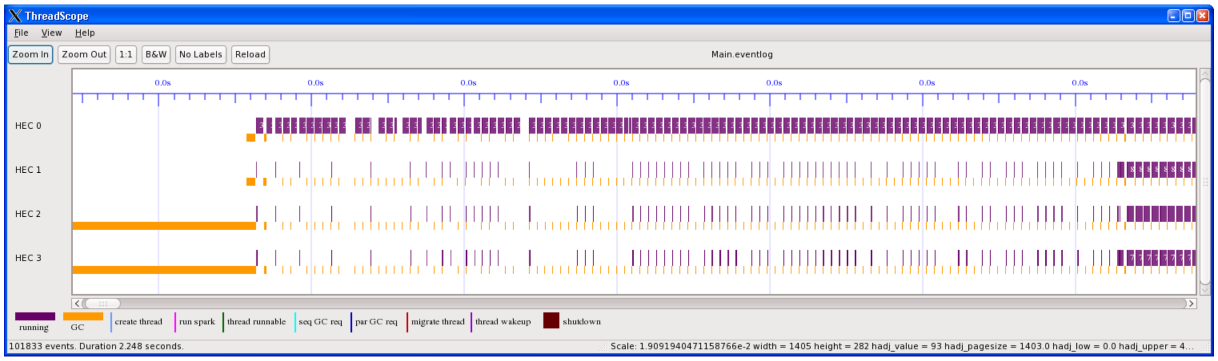
Figure 4. A parallel execution of bsort


ThreadScope
File View Help
Zoom In Zoom Out 1:1 B&W No Labels Reload
Main.eventlog
0us 0.1s 0.2s 0.3s 0.4s 0.5s 0.6s 0.7s 0.8s 0.9s 1.0s 1.1s 1.2s 1.3s 1.4s 1.5s 1.6s 1.7s 1.8s 1.9s 2.0s 2.1s 2.2s 2.3s
HEC 0
HEC 1
HEC 2
HEC 3
running GC create thread run spark thread runnable seq GC req par GC req migrate thread thread wakeup shutdown

101833 events. Duration 2.248 seconds.

Scale: 5.966231397237115e-4 width = 1405 height = 282 hadj_value = 0 hadj_pagesize = 1403.0 hadj_low = 0.0 hadj_upper = 140...

Figure 5. Soda ThreadScope profile


ThreadScope
File View Help
Zoom In Zoom Out 1:1 B&W No Labels Reload
Main.eventlog
0.0s 0.0s 0.0s 0.0s 0.0s 0.0s 0.0s
HEC 0
HEC 1
HEC 2
HEC 3
running GC create thread run spark thread runnable seq GC req par GC req migrate thread thread wakeup shutdown

101833 events. Duration 2.248 seconds.

Scale: 1.9091940471158766e-2 width = 1405 height = 282 hadj_value = 93 hadj_pagesize = 1403.0 hadj_low = 0.0 hadj_upper = 4...

Figure 6. Soda ThreadScope profile (zoomed initial portion)

is not the case here. There are also costs associated with blocking the thread and waking it up again, which we would like to avoid if possible.

One way to avoid this is to evaluate the whole grid before creating any sparks. This is achieved by adding a call to rnf:

```
    -- force the grid to be evaluated:
    evaluate (rnf grid)
```

The effect on the profile is fairly dramatic (Figure 7). We can see that the parallel execution doesn't begin until around 500ms into the execution: creating the grid is taking quite a while. The program also runs slightly faster in parallel now (a 6% improvement, or a parallel speedup of 2.5 compared to 2.4):

```
SPARKS: 12 (11 converted, 0 pruned)

INIT  time    0.00s  (  0.00s elapsed)
MUT   time    7.62s  (  2.31s elapsed)
GC    time    3.35s  (  0.86s elapsed)
EXIT  time    0.00s  (  0.00s elapsed)
Total time   10.97s  (  3.18s elapsed)
```

which we attribute to less blocking and unblocking of threads. We can also see that this program now has a significant sequential section - around 15% of the execution time - which limits the maximum speedup we can achieve with 4 cores to 2.7, and we are already very close to that at 2.5.

To improve parallelism further with this example we would have to parallelize the creation of the initial grid; this probably isn't hard, but it would be venturing beyond the realms of realism somewhat to optimize the creation of the input data for a synthetic benchmark, so we conclude the case study here. It has been instructional to see how thread blocking appears in the ThreadScope profile, and how to avoid it by pre-evaluating data that is needed on multiple CPUs.

Here are a couple more factors that may be affecting the speedup we see in this example:

- The static grid data is created on one CPU and has to be fetched into the caches of the other CPUs. We hope in the future to be able to show the rate of cache misses (and similar characteristics) on each CPU alongside the other information in the ThreadScope profile, which would highlight issues such as this.
- The granularity is too large: we can see that the HECs finish unevenly, losing a little parallelism at the end of the run.

3.3 minimax

Minimax is another historical Parallel Haskell program. It is based on an implementation of alpha-beta searching for the game tic-tac-toe, from Hughes' influential paper "Why Functional Programming Matters" (Hughes 1989). For the purposes of this paper we have generalized the program to use a game board of arbitrary size: the original program used a fixed 3x3 grid, which is too quickly solved to be a useful parallelism benchmark nowadays. However 4x4 still represents a sufficient challenge without optimizing the program further.

For the examples that follow, the benchmark is to evaluate the game tree 6 moves ahead, on a 4x4 grid in which the first 4 moves have already been randomly played. This requires evaluating a maximum of roughly 500,000,000 positions, although parts of the game tree will be pruned, as we shall describe shortly.

We will explore a few different parallelizations of this program using ThreadScope. The function for calculating the best line in the game is alternate:

```
alternate depth player f g board
 = move : alternate depth opponent g f board'
 where
   move@(board',_) = best f possibles scores
   scores          = map (bestMove depth opponent g f) possibles
   possibles       = newPositions player board
   opponent        = opposite player
```

This function calculates the sequence of moves in the game that give the best outcome (as calculated by the alpha-beta search) for each player. At each stage, we generate the list of possible moves (newPositions), evaluate each move by alpha-beta search on the game tree (bestMove), and pick the best one (best).

Let's run the program sequentially first to establish the baseline runtime:

```
14,484,898,888 bytes allocated in the heap

INIT  time    0.00s  (  0.00s elapsed)
MUT   time    8.44s  (  8.49s elapsed)
GC    time    3.49s  (  3.51s elapsed)
EXIT  time    0.00s  (  0.00s elapsed)
Total time   11.94s  ( 12.00s elapsed)
```

One obvious way to parallelize this problem is to evaluate each of the possible moves in parallel. This is easy to achieve with a parListWHNF strategy:

```
scores = map (bestMove depth opponent g f) possibles
           `using` parListWHNF
```

And indeed this does yield a reasonable speedup:

```
14,485,148,912 bytes allocated in the heap

SPARKS: 12 (11 converted, 0 pruned)

INIT  time    0.00s  (  0.00s elapsed)
MUT   time    9.19s  (  2.76s elapsed)
GC    time    7.01s  (  1.75s elapsed)
EXIT  time    0.00s  (  0.00s elapsed)
Total time   16.20s  (  4.52s elapsed)
```

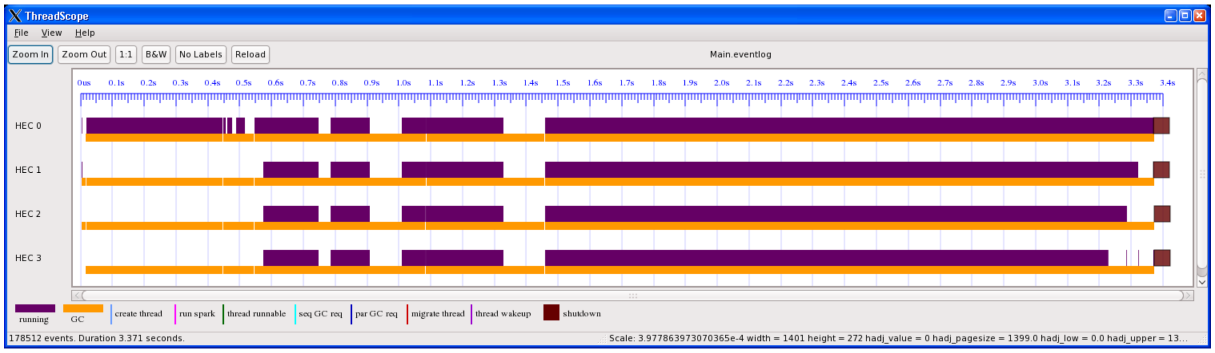
A speedup of 2.7 on 4 processors is a good start! However, looking at the ThreadScope profile (Figure 8), we can see that there is a jagged edge on the right: our granularity is too large, and we don't have enough work to keep all the processors busy until the end. What's more, as we can see from the runtime statistics, there were only 12 sparks, corresponding to the 12 possible moves in the 4x4 grid after 4 moves have already been played. In order to scale to more CPUs we will need to find more parallelism.

The game tree evaluation is defined as follows:

```
bestMove :: Int → Piece → Player → Player → Board
           → Evaluation
bestMove depth p f g
  = mise f g
  . cropTree
  . mapTree static
  . prune depth
  . searchTree p
```

Where searchTree lazily generates a search tree starting from the current position, with player p to play next. The function prune prunes the search tree to the given depth, and mapTree static applies a static evaluation function to each node in the tree. The function cropTree prunes branches below a node in which the game has been won by either player. Finally, mise performs the alpha-beta search, where f and g are the min and max functions over evaluations for the current player p.

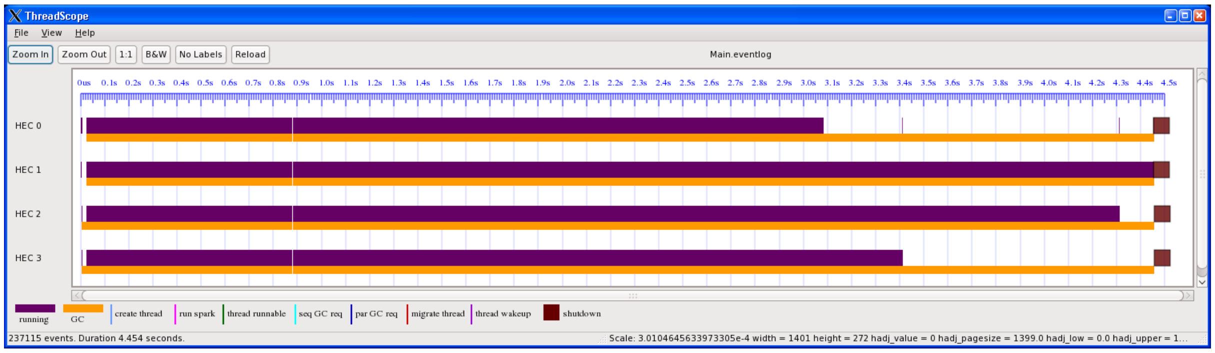
We must be careful with parallelization here, because the algorithm is relying heavily on lazy evaluation to avoid evaluating parts


ThreadScope
File View Help
Zoom In Zoom Out 1:1 B&W No Labels Reload
Main.eventlog
HEC 0
HEC 1
HEC 2
HEC 3
running GC create thread run spark thread runnable seq GC req par GC req migrate thread thread wakeup shutdown

178512 events. Duration 3.371 seconds.

Scale: 3.977863973070365e-4 width = 1401 height = 272 hadj_value = 0 hadj_pagesize = 1399.0 hadj_low = 0.0 hadj_upper = 13…

Figure 7. Soda ThreadScope profile (evaluating the input grid eagerly)


ThreadScope
File View Help
Zoom In Zoom Out 1:1 B&W No Labels Reload
Main.eventlog
HEC 0
HEC 1
HEC 2
HEC 3
running GC create thread run spark thread runnable seq GC req par GC req migrate thread thread wakeup shutdown

237115 events. Duration 4.454 seconds.

Scale: 3.0104645633973305e-4 width = 1401 height = 272 hadj_value = 0 hadj_pagesize = 1399.0 hadj_low = 0.0 hadj_upper = 1…

Figure 8. Minimax ThreadScope profile

of the game tree. Certainly we don't want to evaluate beyond the prune depth, and we also don't want to evaluate beyond a node in which one player has already won (cropTree prunes further moves after a win). The alpha-beta search will prune even more of the tree, since there is no point exploring any further down a branch if it has already been established that there is a winning move. So unless we are careful, some of the parallelism we add here may be wasted speculation.

The right place to parallelize is in the alpha-beta search itself. Here is the sequential code:

```
mise :: Player → Player → Tree Evaluation → Evaluation
mise f g (Branch a [])  = a
mise f g (Branch _ l) = foldr f (g OWin XWin) (map (mise g f) l)
```

The first equation looks for a leaf, and returns the evaluation of the board at that point. A leaf is either a completed game (either drawn or a winning position for one player), or the result of pruning the search tree. The second equation is the interesting one: foldr f picks the best option for the current player from the list of evaluations at the next level. The next level evaluations are given by map (mise g f) l, which picks the best options for the *other* player (which is why the f and g are reversed).

The map here is a good opportunity for parallelism. Adding a parListWHNF strategy should be enough:

```
mise f g (Branch _ l) = foldr f (g OWin XWin)
                          (map (mise g f) l ‘using‘ parListWHNF)
```

However, this will try to parallelize every level of the search, leading to some sparks with very fine granularity. Also it may introduce too much speculation: elements in each list after a win do not need to be evaluated. Indeed, if we try this we get:

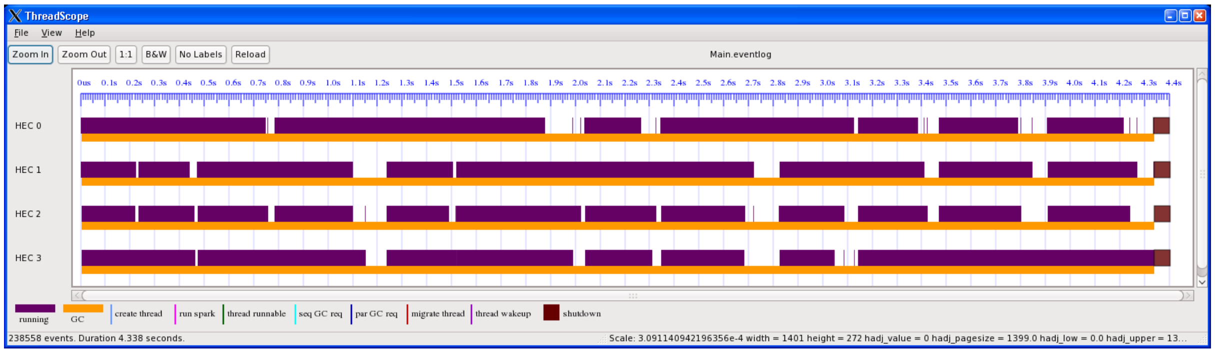
```
22,697,543,448 bytes allocated in the heap

SPARKS: 4483767 (639031 converted, 3457369 pruned)

INIT  time    0.00s  (  0.01s elapsed)
MUT   time   16.19s  (  4.13s elapsed)
GC    time   27.21s  (  6.82s elapsed)
EXIT  time    0.00s  (  0.00s elapsed)
Total time   43.41s  ( 10.95s elapsed)
```

We ran a lot of sparks (600k), but we didn't achieve much speedup over the sequential version. One clue that we are actually speculating useless work is the amount of allocation. In the sequential run the runtime reported 14GB allocated, but this parallel version allocated 22GB[4].

In order to eliminate some of the smaller sparks, we can parallelize the alpha-beta to a fixed depth. This is done by introducing a new variant of mise, parMise, that applies the parListWHNF strategy up to a certain depth, and then calls the sequential mise beyond that. Just using a depth of one gives quite good results:

```
SPARKS: 132 (120 converted, 12 pruned)

INIT  time    0.00s  (  0.00s elapsed)
MUT   time    8.82s  (  2.59s elapsed)
GC    time    6.65s  (  1.70s elapsed)
EXIT  time    0.00s  (  0.00s elapsed)
Total time   15.46s  (  4.30s elapsed)
```

[4] CPU time is not a good measure of speculative work, because in the parallel runtime threads can sometimes be spinning while waiting for work, particularly in the GC.



238558 events. Duration 4.338 seconds.

Scale: 3.091140942196356e-4 width = 1401 height = 272 hadj_value = 0 hadj_pagesize = 1399.0 hadj_low = 0.0 hadj_upper = 13...

Figure 9. Minimax ThreadScope profile (with parMise 1)

Though as we can see from the ThreadScope profile (Figure 9), there are some gaps. Increasing the threshold to two works nicely:

```
SPARKS: 1452 (405 converted, 1046 pruned)

INIT  time    0.00s  (  0.03s elapsed)
MUT   time    8.86s  (  2.31s elapsed)
GC    time    6.32s  (  1.57s elapsed)
EXIT  time    0.00s  (  0.00s elapsed)
Total time   15.19s  (  3.91s elapsed)
```

We have now achieved a speedup of 3.1 on 4 cores against the sequential code, and as we can see from the final ThreadScope profile (Figure 10) all our cores are kept busy.

We found that increasing the threshold to 3 starts to cause speculation of unnecessary work. In 4x4 tic-tac-toe most positions are a draw, so it turns out that there is little speculation in the upper levels of the alpha-beta search, but as we get deeper in the tree, we find positions that are a certain win for one player or another, which leads to speculative work if we evaluate all the moves in parallel.

Ideally GHC would have better support for speculation: right now, speculative sparks are not garbage collected when they are found to be unreachable. We do plan to improve this in the future, but unfortunately changing the GC policy for sparks is incompatible with the current formulation of Strategies (Marlow et al. 2009).

3.4 Thread Ring

The thread ring benchmark originates in the Compter Language Benchmarks Game[5] (formerly known as the Great Computer Language Shootout). It is a simple concurrency benchmark, in which a large number of threads are created in a ring topology, and then messages are passed around the ring. We include it here as an example of profiling a Concurrent Haskell program using ThreadScope, in contrast to the other case studies which have investigated programs that use semi-explicit parallelism.

The code for our version of the benchmark is given in Figure 11. This version uses a linear string of threads rather than a ring, where a number of messages are pumped in to the first thread in the string, and then collected at the other end.

Our aim is to try to make this program speed up in parallel. We expect there to be parallelism available: multiple messages are being pumped through the thread string, so we ought to be able to pump messages through distinct parts of the string in parallel.

First, the sequential performance. This is for 500 messages and 2000 threads:

```
INIT  time    0.00s  (  0.00s elapsed)
MUT   time    0.18s  (  0.19s elapsed)
GC    time    0.01s  (  0.01s elapsed)
EXIT  time    0.00s  (  0.00s elapsed)
Total time    0.19s  (  0.21s elapsed)
```

[5] `http://shootout.alioth.debian.org/`

```
import Control.Concurrent
import Control.Monad
import System
import GHC.Conc (forkOnIO)

thread :: MVar Int → MVar Int → IO ()
thread inp out = do
  x ← takeMVar inp
  putMVar out $! x+1
  thread inp out

spawn cur n = do
  next ← newEmptyMVar
  forkIO $ thread cur next
  return next

main = do
  n ← getArgs >>= readIO.head
  s ← newEmptyMVar
  e ← foldM spawn s [1..2000]
  f ← newEmptyMVar
  forkIO $ replicateM n (takeMVar e) >>= putMVar f . sum
  replicateM n (putMVar s 0)
  takeMVar f
```

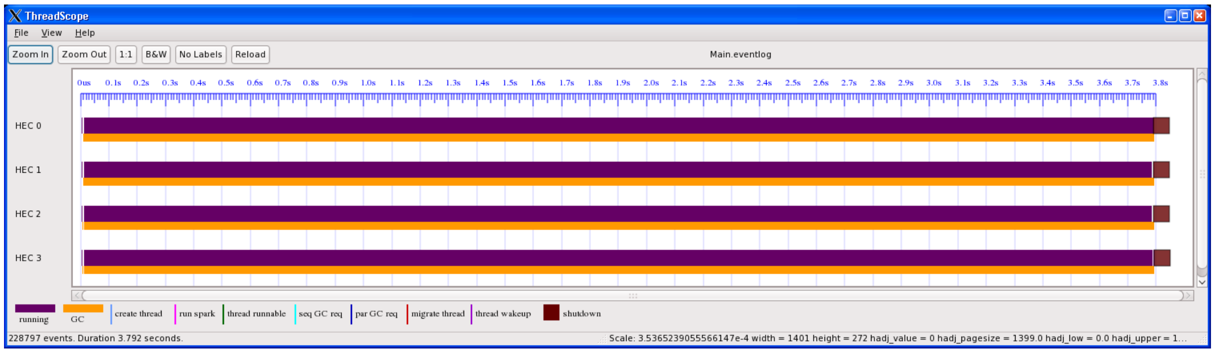
Figure 11. ThreadRing code

Next, running the program on two cores:

```
INIT  time    0.00s  (  0.00s elapsed)
MUT   time    0.65s  (  0.36s elapsed)
GC    time    0.02s  (  0.01s elapsed)
EXIT  time    0.00s  (  0.00s elapsed)
Total time    0.67s  (  0.38s elapsed)
```

Things are significantly slower when we add a core. Let's examine the ThreadScope profile to see why - at first glance, the program seems to be using both cores, but as we zoom in we can see that there are lots of gaps (Figure 12).

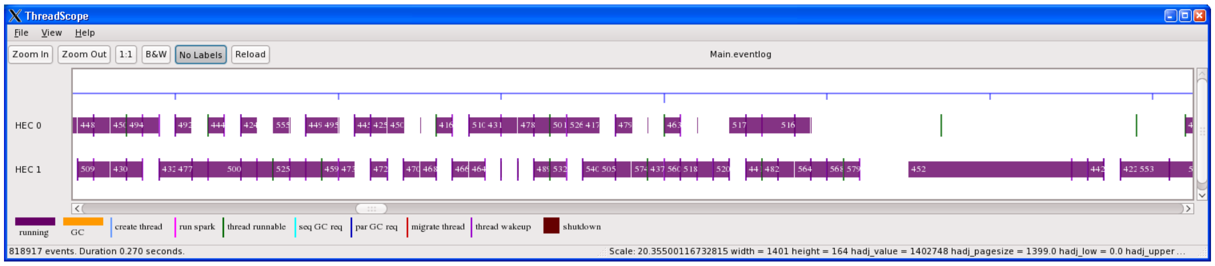
In this program we want to avoid communication between the two separate cores, because that will be expensive. We want as much communication as possible to happen between threads on the same core, where it is cheap. In order to do this, we have to give the scheduler some help. We know the structure of the communication in this program: messages are passed along the string in sequence, so we can place threads optimally to take advantage of that. GHC


ThreadScope
File View Help
Zoom In Zoom Out 1:1 B&W No Labels Reload
Main.eventlog
HEC 0
HEC 1
HEC 2
HEC 3
running GC create thread run spark thread runnable seq GC req par GC req migrate thread thread wakeup shutdown

228797 events. Duration 3.792 seconds.

Scale: 3.5365239055566147e-4 width = 1401 height = 272 hadj_value = 0 hadj_pagesize = 1399.0 hadj_low = 0.0 hadj_upper = 1…

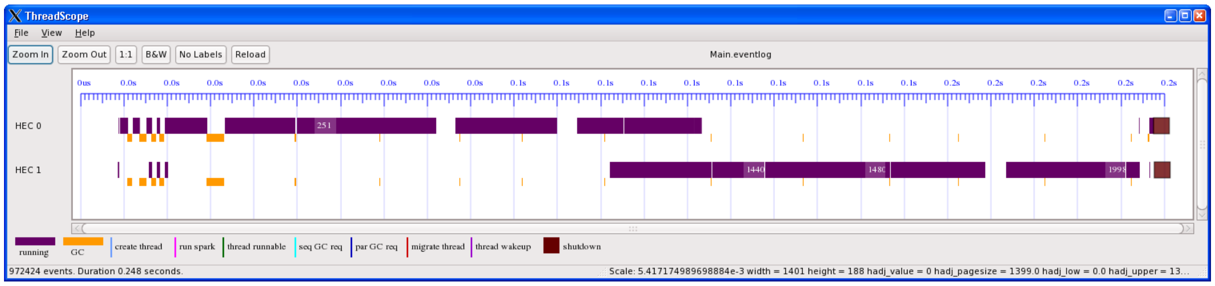
Figure 10. Minimax ThreadScope profile (with parMise 2)


ThreadScope
File View Help
Zoom In Zoom Out 1:1 B&W No Labels Reload
Main.eventlog
HEC 0
HEC 1
running GC create thread run spark thread runnable seq GC req par GC req migrate thread thread wakeup shutdown

818917 events. Duration 0.270 seconds.

Scale: 20.35500116732815 width = 1401 height = 164 hadj_value = 1402748 hadj_pagesize = 1399.0 hadj_low = 0.0 hadj_upper …

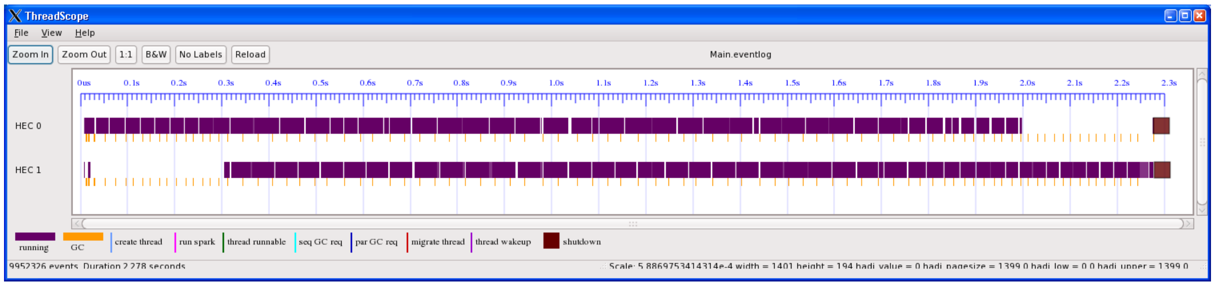
Figure 12. ThreadRing profile (no explicit placement; zoomed in)


ThreadScope
File View Help
Zoom In Zoom Out 1:1 B&W No Labels Reload
Main.eventlog
HEC 0
HEC 1
running GC create thread run spark thread runnable seq GC req par GC req migrate thread thread wakeup shutdown

972424 events. Duration 0.248 seconds.

Scale: 5.417174989698884e-3 width = 1401 height = 188 hadj_value = 0 hadj_pagesize = 1399.0 hadj_low = 0.0 hadj_upper = 13…

Figure 13. ThreadRing profile (with explicit placement)


ThreadScope
File View Help
Zoom In Zoom Out 1:1 B&W No Labels Reload
Main.eventlog
HEC 0
HEC 1
running GC create thread run spark thread runnable seq GC req par GC req migrate thread thread wakeup shutdown

9952326 events. Duration 2.278 seconds.

Scale: 5.8869753414314e-4 width = 1401 height = 194 hadj_value = 0 hadj_pagesize = 1399.0 hadj_low = 0.0 hadj_upper = 1399.0

Figure 14. ThreadRing profile (explicit placement and more messages)

provides a way to place a thread onto a particular core (or HEC), using the forkOnIO operation. The placement scheme we use is to divide the string into linear segments, one segment per core (in our case two).

This strategy gets us back to the same performance as the sequential version:

```
INIT  time    0.00s  (  0.00s elapsed)
MUT   time    0.23s  (  0.19s elapsed)
GC    time    0.02s  (  0.02s elapsed)
EXIT  time    0.00s  (  0.00s elapsed)
Total time    0.26s  (  0.21s elapsed)
```

Why don't we actually see any speedup? Figure 13 shows the ThreadScope profile. The program has now been almost linearized; there is a small amount of overlap, but most of the execution is sequential, first on one core and then the other.

Investigating the profile in more detail shows that this is a scheduling phenomenon. The runtime has moved all the messages through the first string before it propagates any into the second string, and this can happen because the total number of messages we are using for the benchmark is less than the number of threads. If we increase the number of messages, then we do actually see more parallelism. Figure 14 shows the execution profile for 2000 messages and 2000 threads, and we can see there is significantly more overlap.

4. Profiling Infrastructure

Our profiling framework comprises three parts:

- Support in GHC's runtime for tracing events to a log file at runtime. The tracing is designed to be as lightweight as possible, so as not to have any significant impact on the behaviour of the program being measured.
- A Haskell library ghc-events that can read the trace file generated by the runtime and build a Haskell data structure representing the trace.
- Multiple tools make use of the ghc-events library to read and analyze trace files.

Having a single trace-file format and a library that parses it means that it is easy to write a new tool that works with GHC trace files: just import the ghc-events package and write code that uses the Haskell data structures directly. We have already built several such tools ourselves, some of which are merely proof-of-concept experiments, but the ghc-events library makes it almost trivial to create new tools:

- A simple program that just prints out the (sorted) contents of the trace file as text. Useful for checking that a trace file can be parsed, and for examining the exact sequence of events.
- The ThreadScope graphical viewer.
- A tool that parses a trace file and generates a PDF format timeline view, similar to the ThreadScope view.
- A tool that generates input in the format expected by the GtkWave circuit waveform viewer. This was used as an early prototype for ThreadScope, since the timeline view that we want to display has a lot in common with the waveform diagrams that gtkwave displays and browses.

4.1 Fast runtime tracing

The runtime system generates trace files that log certain events and the time at which they occurred. The events are typically those related to thread activity; for example, "HEC 0 started to run thread 3", or "thread 5 blocked on an MVar". The kinds of events we can log are limited only by the extra overhead incurred by the act of logging them. Minimizing the overhead of event logging is something we care about: the goal is to profile the actual runtime behaviour of the program, so it is important that, as far as possible, we avoid disturbing the behaviour that we are trying to profile.

In the GHC runtime, a pre-allocated event buffer is used by each HEC to store generated events. By doing so, we avoid any dynamic memory allocation overhead, and require no locks since the buffers are HEC-local. Yet, this requires us to flush the buffer to the filesystem once it becomes full, but since the buffer is a fixed size we pay a near-constant penalty for each flush and a deterministic delay on the GHC runtime.

The HEC-local buffers are flushed independently, which means that events in the log file appear out-of-order and have to be sorted. Sorting of the events is easily performed by the profiling tool after reading in the log file using the ghc-events library.

To measure the speed at which the GHC runtime can log events, we used a C program (no Haskell code, just using the GHC runtime system as a library) that simply generates 2,000,000 events, alternating between "thread start" and "thread stop" events. Our program generates a 34MB trace file and runs in 0.31 seconds elapsed time:

```
INIT  time    0.00s  (  0.02s elapsed)
MUT   time    0.22s  (  0.29s elapsed)
GC    time    0.00s  (  0.00s elapsed)
EXIT  time    0.00s  (  0.00s elapsed)
Total time    0.22s  (  0.31s elapsed)
```

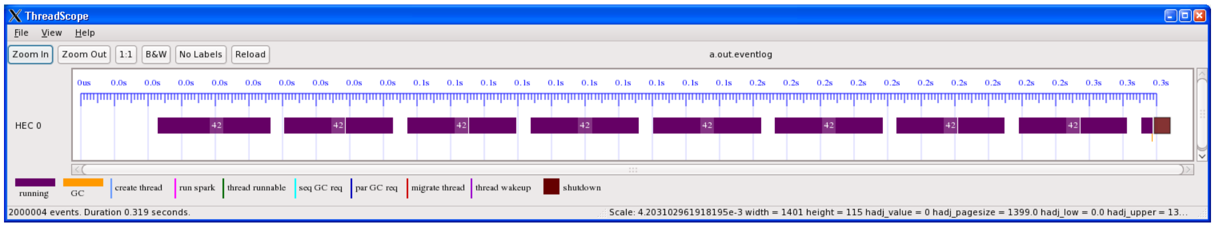
which gives a rough figure of 150ns for each event on average. Looking at the ThreadScope view of this program (Figure 15) we can clearly see where the buffer flushes are happening, and that each one is about 5ms long.

An alternative approach is to use memory-mapped files, and write our events directly into memory, leaving the actual file writing to the OS. This would allow writing to be performed asynchronously, which would hopefully reduce the impact of the buffer flush. According to strace on Linux, the above test program is spending 0.7s writing buffers, so making this asynchronous would save us about 30ns per event on average. However, on a 32-bit machine where we can't afford to reserve a large amount of address space for the whole log file, we would still have to occasionally flush and remap new portions of the file. This alternative approach is something we plan to explore in the future.

To see how much impact event logging has on real execution times, we took a parallel version of the canonical Fibonacci function, parfib, and compared the time elapsed with and without event logging enabled for 50 executions of parfib on an Intel(R) Core(TM)2 Duo CPU T5250 1.50GHz, using both cores. The program generates about 2,000,000 events during the run, and generates a 40MB log file.

```
parfib eventlog
./Main 40 10 +RTS -N2 -l -RTS
Avg Time Elapsed  Standard Deviation
20.582757s        0.789547s

parfib without eventlog
./Main 40 10 +RTS -N2 -RTS
Avg Time Elapsed  Standard Deviation
17.447493s        1.352686s
```

Considering the significant number of events generated in the traces and the very detailed profiling information made available by these traces, the overhead does not have an immense impact at approximately 10-25% increase in elapsed time. In the case



```
2000004 events. Duration 0.319 seconds.
Scale: 4.203102961918195e-3 width = 1401 height = 115 hadj_value = 0 hadj_pagesize = 1399.0 hadj_low = 0.0 hadj_upper = 13...
```

Figure 15. Synthetic event benchmark

of parfib, the event representing the creation of a new spark is dominant, comprising at least 80% of the the events generated. In fact, it is debatable whether we should be logging the creation of a spark, since the cost of logging this event is likely to be larger than the cost of creating the spark itself - a spark creation is simply a write into a circular buffer.

For parallel quicksort, far fewer sparks are created and most of the computation is spent in garbage collection; thus, we can achieve an almost unnoticeable overhead from event tracing. The parallel quicksort example involved sorting a list of 100,000 randomly generated integers and was performed in the same manner as parfib where we compare with event logging and without, yet in this test we perform 100 executions on an Intel(R) Core(TM) 2 Quad CPU 3.0Ghz.

```
parquicksort eventlog
./Main +RTS -N4 -l -RTS
Avg Time Elapsed  Standard Deviation
14.201385s        2.954869

parquicksort without eventlog
./Main +RTS -N4 -RTS
Avg Time Elapsed  Standard Deviation
15.187529s        3.385293s
```

Since parallel quicksort spent the majority of the computation doing useful work, particularly garbage collection of the created lists, a trace file of only approximately 5MB and near 300,000 events was created and the overhead of event tracing is not noticeable.

The crux of the event tracing is that even when a poorly performing program utilizes event tracing, the overhead should still not be devastating to the program's performance, but best of all on a program with high utilization event tracing should barely affect the performance.

4.2 An extensible file format

We believe it is essential that the trace file format is both backwards and forwards compatible, and architecture independent. In particular, this means that:

- If you build a newer version of a tool, it will still work with the trace files you already have, and trace files generated by programs compiled with older versions of GHC.
- If you upgrade your GHC and recompile your programs, the trace files will still work with any profiling tools you already have.
- Trace files do not have a shelf life. You can keep your trace files around, safe in the knowledge that they will work with future versions of profiling tools. Trace files can be archived, and shared between machines.

Nevertheless, we don't expect the form of trace files to remain completely static. In the future we will certainly want to add new events, and add more information to existing events. We therefore need an extensible file format. Informally, our trace files are structured as follows:

- A list of *event types*. An event-type is a variable-length structure that describes one kind of event. The event-type structure contains
  - A unique number for this event type
  - A field describing the length in bytes of an instance of the event, or zero for a variable-length event.
  - A variable-length string (preceded by its length) describing this event (for example "thread created")
  - A variable-length field (preceded by its length) for future expansion. We might in the future want to add more fields to the event-type structure, and this field allows for that.
- A list of *events*. Each event begins with an event number that corresponds to one of the event types defined earlier, and the length of the event structure is given by the event type (or it has variable length). The event also contains
  - A nanosecond-resolution timestamp.
  - For a variable-length event, the length of the event.
  - Information specific to this event, for example which CPU it occurred on. If the parser knows about this event, then it can parse the rest of the event's information, otherwise it can skip over this field because its length is known.

The unique numbers that identify events are shared knowledge between GHC and the ghc-events library. When creating a new event, a new unique identifier is chosen; identifiers can never be re-used.

Even when parsing a trace file that contains new events, the parser can still give a timestamp and a description of the unknown events. The parser might encounter an event-type that it knows about, but the event-type might contain new unknown fields. The parser can recognize this situation and skip over the extra fields, because it knows the length of the event from the event-type structure. Therefore when a tool encounters a new log file it can continue to provide consistent functionality.

Of course, there are scenarios in which it isn't possible to provide this ideal graceful degradation. For example, we might construct a tool that profiles a particular aspect of the behaviour of the runtime, and in the future the runtime might be redesigned to behave in a completely different way, with a new set of events. The old events simply won't be generated any more, and the old tool won't be able to display anything useful with the new trace files. Still, we expect that our extensible trace file format will allow us to smooth over the majority of forwards- and backwards-compatibility issues that will arise between versions of the tools and

GHC runtime. Moreover, extensibility costs almost nothing, since the extra fields are all in the event-types header, which has a fixed size for a given version of GHC.

5. Related Work

GranSim (Loidl 1998) is an event-driven simulator for the parallel execution of Glasgow Parallel Haskell (GPH) programs which allows the parallel behaviour of Haskell programs to be analyzed by instantiating any number of virtual processors which are emulated by a single thread on the host machine. GranSim has an associated set of visualization tools which show overall activity, per-processor activity, and per-thread activity. There is also a separate tool for analyzing the granularity of the generated threads. The GUM system (Trinder et al. 1996) is a portable parallel implementation of Haskell with good profiling support for distributed implementations.

The timeline view in ThreadScope is very similar to that of the trace viewer for Eden (Loogen et al. 2005).

6. Conclusions and Further work

We have shown how thread-based profile information can be effectively used to help understand and fix parallel performance bugs in both Parallel Haskell and Concurrent Haskell programs, and we expect these profiling tools to also be of benefit to developers using Data Parallel Haskell in the future.

The ability to profile parallel Haskell programs plays an important part in the development of such programs because the analysis process motivates the need to develop specialized strategies to help control evaluation order, extent and granularity as we demonstrated in the minmax example.

Here are some of the future directions we would like to take this work:

- Improve the user interface and navigation of ThreadScope. For example, it would be nice to filter the display to show just a subset of the threads, in order to focus on the behaviour of a particular thread or group of threads.
- It would also be useful to understand how threads interact with each other via MVars e.g. to make it easier to see which threads are blocked on read and write accesses to MVars.
- The programmer should be able to generate events programmatically, in order to mark positions in the timeline so that different parts of the program's execution can easily be identified and separated in ThreadScope.
- It would be straightforward to produce graphs similar to those from the GpH and GranSim programming tools (Trinder et al. 2002; Loidl 1998), either by writing a Haskell program to translate the GHC trace files into the appropriate input for these tools, or by rewriting the tools themselves in Haskell.
- Combine the timeline profile with information from the OS and CPU. For example, for IO-bound concurrent programs we might like to see IO or network activity displayed on the timeline. Information from CPU performance counters could also be superimposed or displayed alongside the thread timelines, providing insight into cache behaviour, for example.
- Have the runtime system generate more tracing information, so that ThreadScope can display information about such things as memory usage, run queue sizes, spark pool sizes, and foreign call activity.

Acknowledgments

The authors would like to acknowledge the work of the developers of previous Haskell concurrent and parallel profiling systems which have provided much inspiration for our own work. Specifically work on GpH, GranSim and Eden was particularly useful.

We wish to thank Microsoft Research for funding Donnie Jones' visit to Cambridge in 2008 during which he developed an early prototype of event tracing in GHC.

References

Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM. ISBN 1-59593-080-9. doi: http://doi.acm.org/10.1145/1065944.1065952.

John Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, April 1989.

H-W. Loidl. *Granularity in Large-Scale Parallel Functional Programming*. PhD thesis, Department of Computing Science, University of Glasgow, March 1998.

Rita Loogen, Yolanda Ortega-Malln, and Ricardo Pea-Mar. Parallel functional programming in Eden. *Journal of Functional Programming*, 3 (15):431–475, 2005.

Simon Marlow, Simon Peyton Jones, and Satnam Singh. Runtime support for multicore Haskell. In *ICFP'09: The 14th ACM SIGPLAN International Conference on Functional Programming*, Edinburgh, Scotland, 2009.

E. Mohr, D. A. Kranz, and R. H. Halstead. Lazy task creation – a technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3), July 1991.

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 *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2008)*, 2008.

Colin Runciman and David Wakeling. Profiling parallel functional computations (without parallel machines). In *Glasgow Workshop on Functional Programming*, pages 236–251. Springer, 1993.

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)*. Philadelphia, May 1996.

P.W. Trinder, K. Hammond, H.-W. Loidl, and Simon Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8 (1):23–60, January 1998. URL `http://research.microsoft.com/Users/simonpj/Papers/strategies.ps.gz`.

P.W. Trinder, H.-W. Loidl, and R. F. Pointon. Parallel and Distributed Haskells. *Journal of Functional Programming*, 12(5):469–510, July 2002.