CS4012 Parallelism Report

I have three files, sorting1.hs, sorting2.hs and sorting3.hs, which I will talk about in order. The explanation in sorting1.hs is important because this is where I explain the entire set up of my algorithm, and my real starting point, so bare with me for now.

---

sorting1.hs

First of all, I had to come up with two sorting algorithms, and I chose to come up with quicksort and mergesort functions, as shown below:

```haskell
--MERGESORT
mergesort :: (Ord a) => [a] -> [a]

mergesort [] = []
mergesort [single] = [single]
mergesort lst = merge (mergesort leftHalf) (mergesort rightHalf)
                  where leftHalf = take (div (length lst) 2) lst
                        rightHalf = drop (div (length lst) 2) lst

merge :: (Ord a) => [a] -> [a] -> [a]
merge lList [] = lList
merge [] rList = rList
merge (lhead : ltail) (rhead : rtail)
    | lhead <= rhead = lhead : merge ltail (rhead:rtail)
    | otherwise      = rhead : merge (lhead:ltail) rtail
```

```haskell
--QUICKSORT
quicksort :: (Ord a) => [a] -> [a]

quicksort [] = []
quicksort (pivot:xs) = (quicksort lesserPart) ++ [pivot] ++ (quicksort
greaterPart)
    where
        lesserPart = filter (<= pivot) xs
        greaterPart = filter (> pivot) xs
```

To check if this worked, I did the following in main:

- Passed a list of 30,000 elements (which I got from a random list generator online) to the sort functions
- Calculated the amount of time it took for the sort functions (in this case, quicksort, since mergesort was commented) to run, using function secDiff and the System.Time library and,
- printed out the sorted (hopefully) list

Illustrations of these in my code are shown on the next page.

```haskell
--MAIN
main =
  let lst = [101764, 175658, 28192, 38403, 19196, 171099, 92654, 171566, 82870,
  30865, 90299, 18486, 28894, 17613, 159502, 66050, 91010, 157523, 21177,
  15136, 190825, 13236, 33465, 157371, 81722, 197033, 79503, 159634, 22031,
```

list of length 30,000

```haskell
in
  (do
    timeBefore <- getClockTime
    -- sortedLst <- return $ mergesort $ lst
    sortedLst <- return $ quicksort $ lst
    timeAfter <- getClockTime
    print $ sortedLst
    print $ "Time diff : " ++ show (secDiff timeAfter timeBefore))
```

passing list to quicksort, calculating time diff before and after function using secDiff, and printing the sorted list

```haskell
--TIME HELPER FUNC
secDiff (TOD secs1 psecs1) (TOD secs2 psecs2) = fromInteger ( psecs1 - psecs2)
/ 1e12 + fromInteger (secs1 - secs2)
```

Implementation of secDiff

I complied the program and ran the executable file with option -N1 (one core), just to check if it worked, and the output was relieving, it did!

```
[Shauns-MacBook-Air:1 shaunjose$ ghc -threaded -rtsopts -eventlog sorting1.hs
[Shauns-MacBook-Air:1 shaunjose$ ./sorting1 +RTS -s -N1 -ls
 [1,1,2,2,3,3,4,4,5,5,6,6,7,7,8,8,9,9,10,10,11,11,12,12,13,13,13,14,14,15,15,1
6,17,17,18,18,19,19,20,20,20,21,21,22,22,23,23,24,24,25,25,26,26,27,27,28,28,
29,30,30,31,31,31,32,32,33,33,34,34,35,35,35,36,36,37,37,38,38,39,39,40,40,41
,42,42,43,43,44,44,45,45,46,46,47,47,48,48,49,49,50,50,51,51,52,52,53,53,54,5
4,55,55,56,56,57,57,58,58,59,59,60,60,61,61,62,62,63,63,64,64,65,65,66,66,67,
68,68,69,69,70,70,71,71,72,72,73,73,74,74,75,75,76,76,77,77,78,78,79,79,80,80
,81,82,82,83,83,84,84,85,85,86,86,87,87,88,88,89,89,90,90,90,91,91,92,92,93,9
4,94,95,95,96,96,97,97,98,98,99,99,100,100,101,101,102,102,103,103,104,104,10
05,106,106,107,107,108,108,109,109,110,110,111,111,112,112,113,113,114,114,11
15,115,116,116,117,117,118,118,119,119,120,120,121,121,122,122,123,123,124,12
25,125,126,126,126,127,127,128,128,129,129,130,130,131,131,132,132,133,133,13
34,135,135,136,136,137,137,138,138,139,139,140,140,141,141,142,142,143,143,14
44,145,145,146,146,147,147,148,148,149,149,150,150,151,151,152,152,153,153,15
54,155,155,156,156,156,157,157,158,158,159,159,160,160,161,161,162,162,163,16
64,164,165,165,166,166,167,167,168,168,169,169,170,170,171,171,172,172,173,17
74,174,175,175,176,176,177,177,178,178,179,179,180,180,181,181,182,182,183,18
```

line 1 = compile ; line 2 = run with one core ; line 3 onwards = sorted list (result needed)

That was easy! But hold on for a second, let's check the rest of our output to see the time it took for quicksort to run:

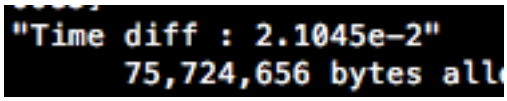
```
"Time diff : 1.0e-6"
```

What!???!
Yes, quicksort is fast. But it takes just that long to run on a list of 30,000 elements using only one core? Something's gotta be wrong.
After much thought, I realised that Haskell is a lazy language and this lazy evaluation is what gives the illusion of a fast quicksort method. It only got evaluated when I printed it out. Obviously, the solution now was to get rid of the laziness in order to calculate the "true" runtime of quicksort. I simply had to add a "!" to force the evaluation, and my code now looked like this:

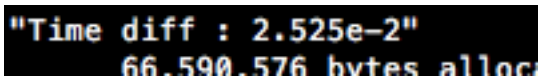
```
(do
  timeBefore <- getClockTime
  -- sortedLst <- return $! mergesort $ lst
  sortedLst <- return $! quicksort $ lst
  timeAfter <- getClockTime
  print $ sortedLst
```

forcing evaluation with !

I complied and ran it with one core once again, and it worked again, also giving a more reasonable runtime for the quicksort function. I did this for mergesort as well (after uncommenting mergesort and commenting quicksort), and received the following runtimes:

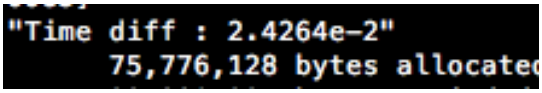


quicksort on one core

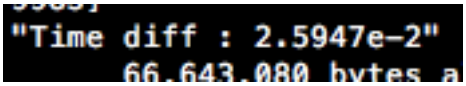


mergesort on one core

That's more like it! Now, before trying to make it parallel, I'm going to do the intuitively obvious thing (if you're silly, like me) -> run this program on 4 cores! I switched the -N1 option to -N4 while running the executable files for each sort function, and received these results:



quicksort on 4 cores



mergesort on 4 cores

You would notice that the runtime is almost the same, or actually, slightly higher than that while using one core. This is because we haven't actually introduced any parallelism in our code, and moreover, trying to use 4 cores includes some unnecessary spark overhead in exchange for no real gain (parallelism) at all!
I was able to confirm this using thread scope, and this is where the event log files came in handy. The thread scope results are shown on the next page.

mergesort on one core

mergesort on 4 cores

quicksort on one core

quicksort on 4 cores

Observe how running it on one core just deals with that single core, but running either sorting algorithm on 4 cores causes unnecessary spark overheads due to all 4 cores, and what's worse is that the other 3 cores do absolutely nothing, except for the tiniest bit of work at the beginning and the end which doesn't seem to compensate for the spark overheads.
It is evident that making this program faster wasn't that easy after all, and I had to actually try to really introduce some parallelism programatically, in order to improve the performance of these sorting algorithms.

Finally, we can move on to the real stuff, after setting up my sorting algorithms, a list of 30,000 elements, introducing timing calculations and forcing the evaluation for the sorting functions in order to calculate the "true" runtime of the functions.

Parallelism was attempted in a separate file, 'sorting2.hs'

sorting2.hs

Parallelising Mergesort

I made a copy of the file sorting1.hs, named it sorting2.hs, and tried to introduce parallelism in my mergesort function at first. I used to Control.Parallel library in order to use functions par and pseq.

It was very tempting for me to type ‘par leftHalf (pseq rightHalf (merge (mergesort leftHalf) (mergesort rightHalf)))’, but after noticing that this would just parallelise the take and drop operations (which leftHalf and rightHalf are set as, respectively), I realised it wouldn’t be such a great idea. Take and drop operations are relatively cheap, and parallelising this would not compensate for the spark overheads of the threads.

So, I instead decided to parallelise the “merge sort leftHalf” and “merge sort rightHalf” (which keeps breaking up the list into smaller lists) as shown.

```
mergesort [] = []
mergesort [single] = [single]
mergesort lst = let mergesort1 = mergesort $ take (div (length lst) 2) lst
                    mergesort2 = mergesort $ drop (div (length lst) 2) lst
                in
                  par mergesort1 (pseq mergesort2 (merge mergesort1
                  mergesort2) )
```

Parallelising mergesort :- the splitting-lists part

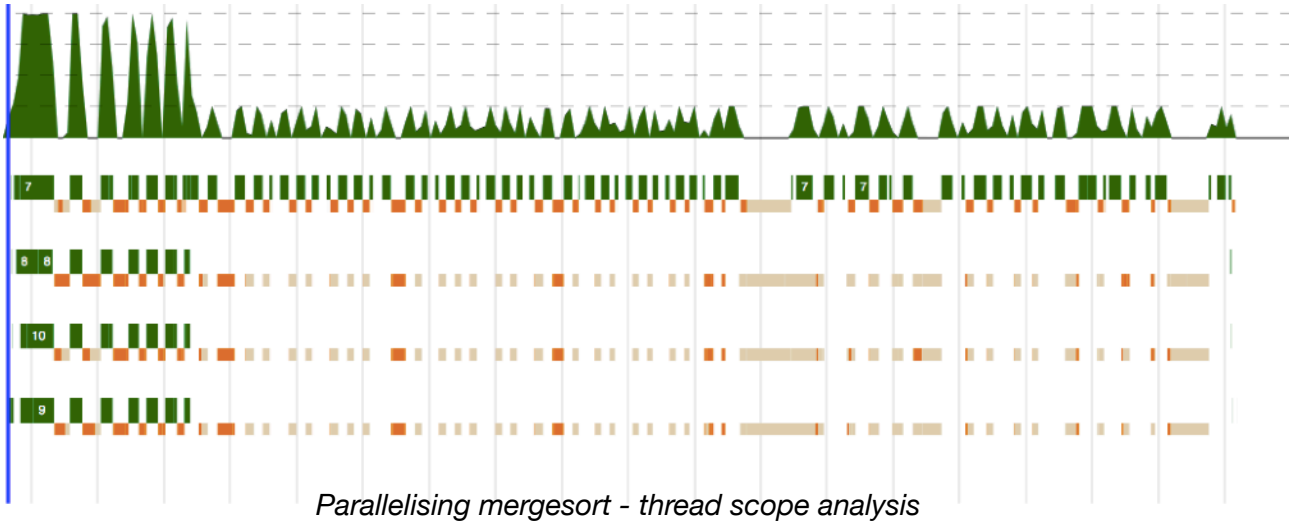
I compiled and ran it (using 4 cores; -N4), and the result was pleasing!

```
"Time diff : 1.3205e-2"
      69,868,000 bytes allocated in the heap
      16,847,264 bytes copied during GC
       1,503,864 bytes maximum residency (8 sample(s))
         108,008 bytes maximum slop
               9 MB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0        41 colls,    41 par    0.065s   0.028s     0.0007s    0.0038s
  Gen  1         8 colls,     7 par    0.021s   0.008s     0.0010s    0.0021s

  Parallel GC work balance: 54.92% (serial 0%, perfect 100%)

  TASKS: 10 (1 bound, 9 peak workers (9 total), using -N4)

  SPARKS: 30952 (21 converted, 0 overflowed, 0 dud, 26511 GC'd, 4420 fizzled)
```

Parallelising mergesort stats

Our first speedup! Notice the decrease in runtime compared to the sequential version of mergesort. Mergesort ran almost twice as fast this time, and also notice the “Work balance” (54.92%), which shows that there was definitely some parallelism in my program.

ThreadScope confirmed that I was breaking up the lists in parallel.



Parallelising mergesort - thread scope analysis

As can be seen, the rest of the work was done on one core, as the latter part was the “merge” function running, which I hadn’t parallelised at all. i only parallelised the breaking-up-lists portion, which happened in the beginning, and that’s what ThreadScope is showing. This meant I could do a lot better if I parallelised the merge function as well.

Looking at my code for the merge function;

```
16  merge :: (Ord a) => [a] -> [a] -> [a]
17  merge lList [] = lList
18  merge [] rList = rList
19  merge (lhead : ltail) (rhead : rtail)
20      | lhead <= rhead = lhead : merge ltail (rhead:rtail)
21      | otherwise      = rhead : merge (lhead:ltail) rtail
```

I see that the most taxing job merge does is call merge again with a head and tail of a list. Now, the head and tails of both lists are already more or less evaluated, so merge really has just one job -> call merge with the evaluated lists.
However, if both the lists are more or less evaluated, there was no point of parallelising it, as that would include unnecessary overheads with no considerable increase in parallelisation, causing an overall slowdown.

Well, there was no harm in trying it out. Sometimes, things “just work”, and so I parallelised merge. I even went so far as to limit the parallelisation of merge and call the sequential version after a point, to reduce the spark overhead. The new parallel and sequential merge functions can be seen on the next page:

```haskell
smerge :: (Ord a) => [a] -> [a] -> [a]
smerge lList [] = lList
smerge [] rList = rList
smerge (lhead : ltail) (rhead : rtail)
    | lhead <= rhead = lhead : smerge ltail (rhead:rtail)
    | otherwise      = rhead : smerge (lhead:ltail) rtail

merge :: (Ord a) => Integer -> [a] -> [a] -> [a]
merge _ lList [] = lList
merge _ [] rList = rList
merge 0 lList rList = smerge lList rList
merge d (lhead : ltail) (rhead : rtail)
    | lhead <= rhead = let restOfLst = merge (d-1) ltail (rhead: rtail)
                       in
                         par restOfLst (lhead : restOfLst)
    | otherwise      = let restOfLst = merge (d-1) (lhead: ltail) rtail
                       in
                         par restOfLst (rhead : restOfLst)

```

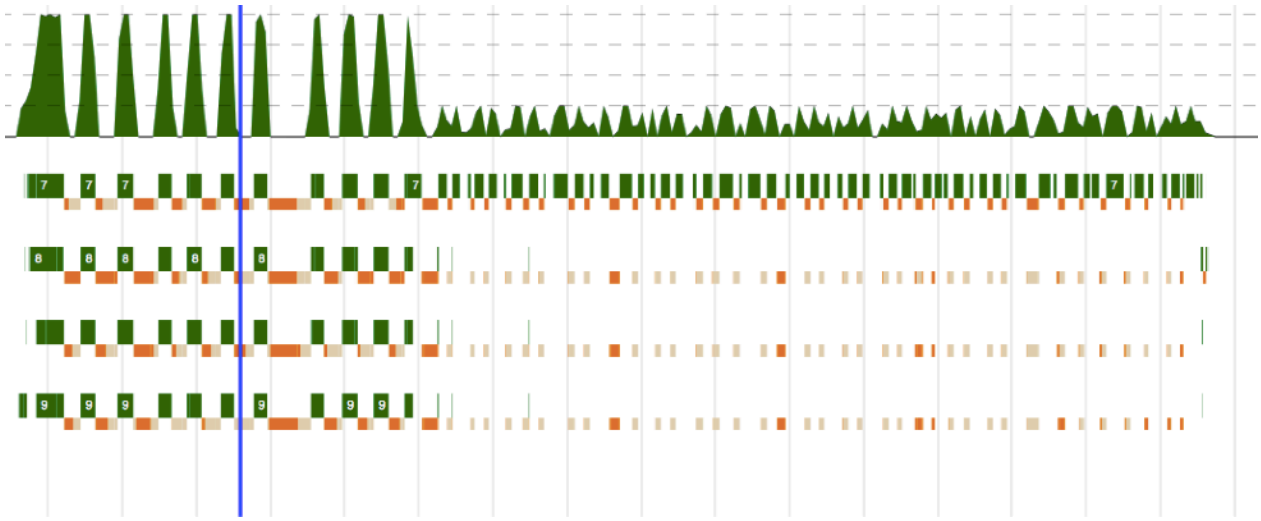
Attempt to parallelise the merge function

The stats were actually surprising. It seems the spark overhead was humongous, and the time was worse than even the sequential version of mergesort.

```
"Time diff : 4.651e-2"
      75,622,584 bytes allocated in the heap
```

Result of parallelising the merge function

You probably guessed that I invested the event log file in ThreadScope, because of my seemingly increasing obsession with it, and it showed me mostly no parallelism, but also some additional spark overhead for the latter part of the runtime, which represented the merge part of the code.



Parallelising merge function :- spark overheads later on with no real parallelism. A total loss

Unfortunately, I had no further sound ideas of how I could parallelise merge, and so, I switched back to my sequential version of merge, let mergesort stay parallel and stuck with it, after achieving close to a 50% increase in speed.

Parallelising Quicksort

It was time to leave mergesort aside for a little bit, and work on quicksort. I was lucky enough to get an increase in speed with mergesort on my first attempt, but sadly, quicksort was a bit harder to understand and parallelise.

Since mergesort's efficiency increased when I parallelised the recursive mergesort part of the code, my first instinct was to do the same for quicksort, and so I parallelised the recursive quicksort calls, making my code look like this:

```
--QUICKSORT
quicksort :: (Ord a) => [a] -> [a]

quicksort [] = []
quicksort (pivot:xs) = let quicksort1 = (quicksort lesserPart)
                           quicksort2 = (quicksort greaterPart)
                       in
                         par quicksort1 (pseq quicksort2 (quicksort1 ++
                         [pivot] ++ quicksort2))
                   where
                       lesserPart = filter (<= pivot) xs
                       greaterPart = filter (> pivot) xs
```

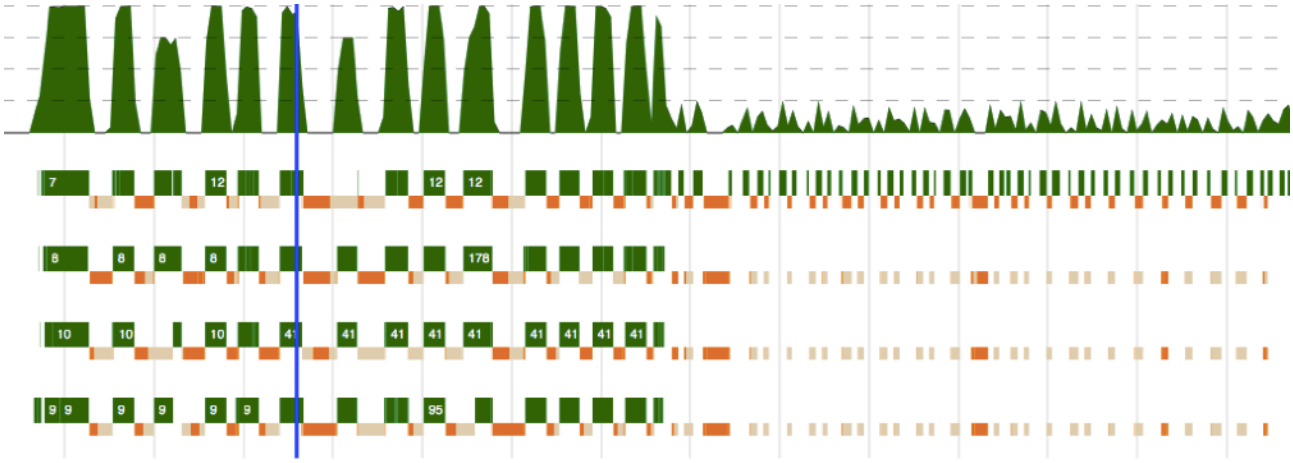
Parallelising quicksort :- Focussing on the recursive calls

I compiled and ran this once with 4 cores, and achieved the following stats ;

```
"Time diff : 3.4783e-2"
      82,873,248 bytes allocated in the heap
      19,917,376 bytes copied during GC
       2,015,864 bytes maximum residency (8 sample(s))
          62,640 bytes maximum slop
              10 MB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0        36 colls,    36 par    0.059s   0.019s     0.0005s    0.0015s
  Gen  1         8 colls,     7 par    0.023s   0.008s     0.0010s    0.0019s

  Parallel GC work balance: 56.39% (serial 0%, perfect 100%)
```

Result of parallelising recursive quicksort function calls

That is a major slowdown, what an absolute disaster. Notice how the work balance was 56.39%, showing a good amount of parallelism! Before repeatedly hitting 'control + Z', I wanted to understand what was happening clearly, using our handy analysis app, ThreadScope.



ThreadScope :- parallelising recursive quicksort function calls

It was probably the spark overheads shown, that were barely being compensated by parallelising the quicksort calls, and it increased my runtime significantly.

The only way out now was to think from scratch. If paralleling quicksort calls didn't work, my only other way to increase efficiency was to work on paralleling the partition. I switched back to my sequential version of quicksort, and then parallelised partition, to give y newly modified code for the quicksort function:

```
--QUICKSORT
quicksort :: (Ord a) => [a] -> [a]

quicksort [] = []
quicksort (pivot:xs) = par lesserPart (pseq greaterPart ((quicksort lesserPart)
++ [pivot] ++ (quicksort greaterPart)))
    where
        lesserPart = filter (<= pivot) xs
        greaterPart = filter (> pivot) xs
```

Parallelising quicksort :- Focussing on the partitions

This was parallelising partitions, as I placed the lesserPart and greaterPart evaluations on different threads, and used the filter to calculate both the parts of the list (effectively parallelising the filter part -> thus parallelising the partitioning aspect of quicksort).
I compiled the program, ran it with 4 cores, and finally, achieved a major speedup:

```
"Time diff : 9.45e-3"
     104,346,288 bytes allocated in the heap
      26,359,416 bytes copied during GC
       3,710,432 bytes maximum residency (8 sample(s))
         111,136 bytes maximum slop
              15 MB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0        92 colls,    92 par    0.060s   0.042s     0.0005s    0.0050s
  Gen  1         8 colls,     7 par    0.037s   0.025s     0.0031s    0.0088s

  Parallel GC work balance: 4.01% (serial 0%, perfect 100%)
```

Result of parallelising recursive quicksort partitions

(9 is a bigger number, but the e-3 is what makes it smaller as opposed to e-2, showing a decrease in runtime and hence a speedup)

What really intrigued me about this was the work balance. Notice that it's just 4.01%. Whereas when we parallelised the quicksort calls, the work load was 56.39%, which hints that it was way more parallel than this version. Even after this, it was the second version that ran faster.

Strange. Right?

Maybe it isn't.

Perhaps it was the overly massive parallelism that was causing the slowdown in my first attempt, due to the probably massive number of spark overheads.

If anything, this is proof that spark overheads can dominate sometimes, especially when you're working on a large list with about 30,000 elements in it, and maybe this was what caused a major slowdown in my first attempt of parallelisation.

Introducing just a bit of parallelism however, just for the partitioning aspect of quicksort, happened to work wonders, because of the positive tradeoff between an increase in efficiency due to a bit of parallelism, with an insignificant number of spark overheads.

This was what I stuck with, and this is all there is to 'sorting2.hs', as I believed I achieved a good amount of progress with respect to both the sorting algorithms.

This seemed like a good milestone and a nice checkpoint.

The only thing I can think of at this point is restricting the number spark overheads, a term I've over used in this report, and rightfully so, since it is so critical when it comes to the runtime over large lists and data.

I tried to play around with it and restrict it with the intention of speeding up my program, in sorting3.hs

sorting3.hs

Restricting Spark Overheads in Mergesort

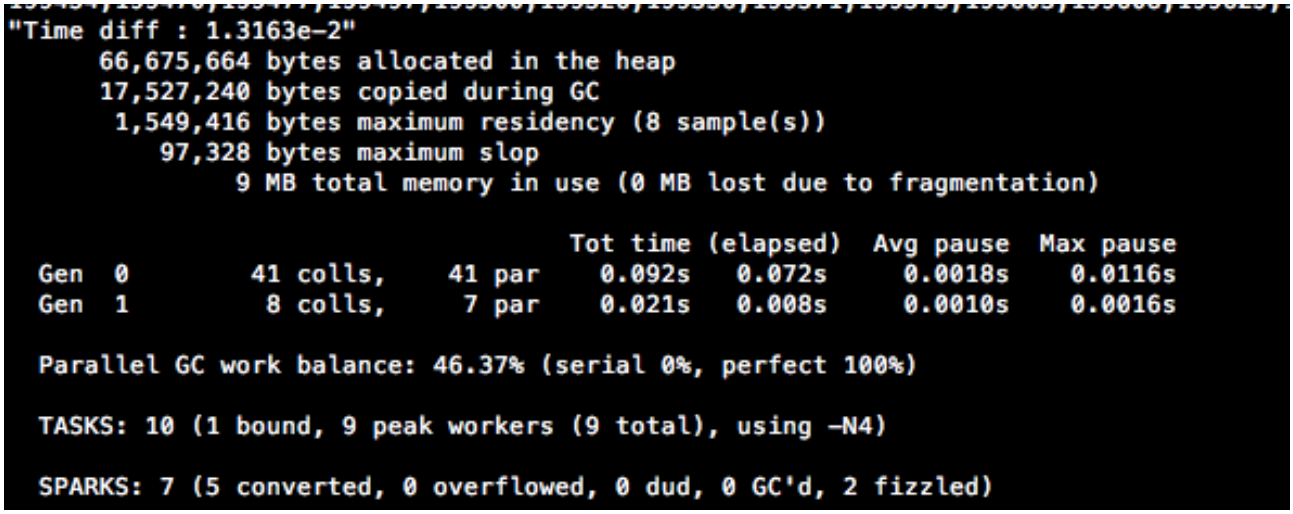
First of all, I want to make it clear that I wouldn't really call this an improvement, as I received similar results of efficiency, but in rather different ways. So it's more of a different technique rather than an improvement in this case.

```haskell
--MERGESORT

--SEQUENTIAL VERSION
smergesort :: (Ord a) => [a] -> [a]

smergesort [] = []
smergesort [single] = [single]
smergesort lst = merge (smergesort leftHalf) (smergesort rightHalf)
                 where leftHalf = take (div (length lst) 2) lst
                       rightHalf = drop (div (length lst) 2) lst

--PARALLEL VERSION
mergesort :: (Ord a) => Integer -> [a] -> [a]

mergesort 0 lst = smergesort lst --do sequential once you get 0
mergesort _ [] = []
mergesort _ [single] = [single]
mergesort d lst = let mergesort1 = (mergesort (d-1)) $ take (div (length lst)
2) lst
                      mergesort2 = (mergesort (d-1)) $ drop (div (length lst)
                      2) lst
                  in
                    par mergesort1 (pseq mergesort2 (merge mergesort1
                    mergesort2) )
```

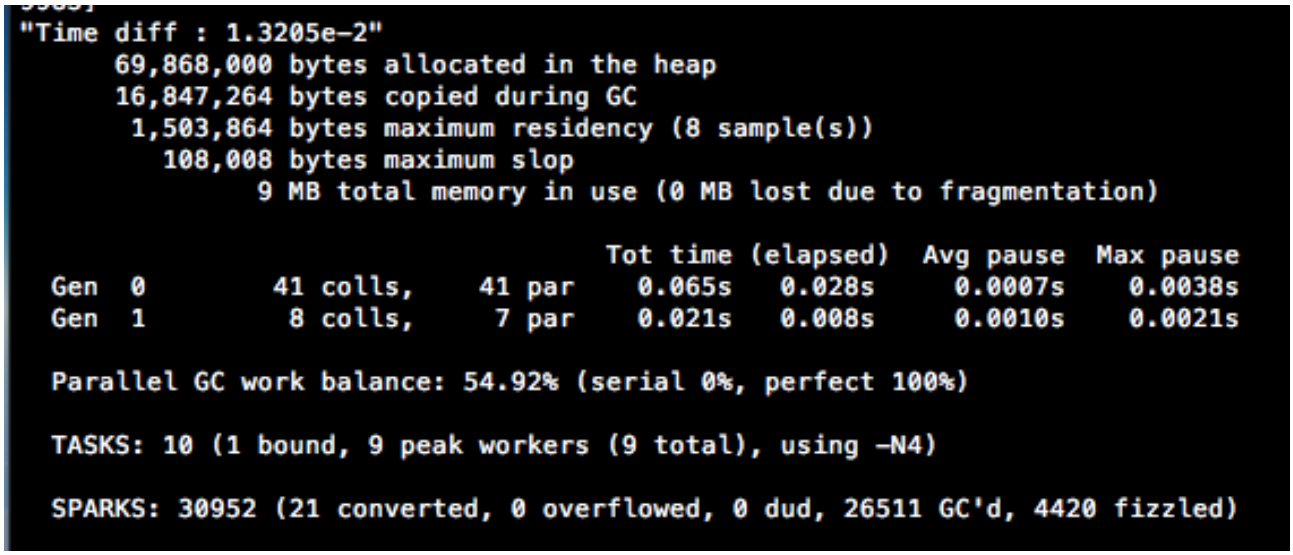
Restricting Spark overheads in mergesort

What I did to my code is simple. I just included the sequential version of mergesort, included an integer argument to my parallel mergesort, and kept subtracting until it went to 0, which is when I redirected to the sequential version. This reduced the parallelism, but that in turn also reduced the number of sparks.

Compare the stats of running mergesort in sorting3.hs and sorting2.hs



sorting3.hs : mergesort stats



sorting2.hs : mergesort stats

Notice how the parallelism (work balance) in sorting3.hs is slightly lower, but the Sparks are significantly reduced (10 as opposed to 30952). There's barely a difference in the efficiency of these two, sorting2 is fast because of the parallelism but sorting3 catches up because of the reduced number of sparks.

3 seemed to be the "magic number" to pass to mergesort, as when I tried it with 2 or 4, the efficiency seemed to reduce. Making it 14 would be the same as sorting2, as log 30,000 to the base 2 = 14, which is the amount of times mergesort splits the list.

Moving on to Quicksort…

Restricting Spark Overheads in Quicksort

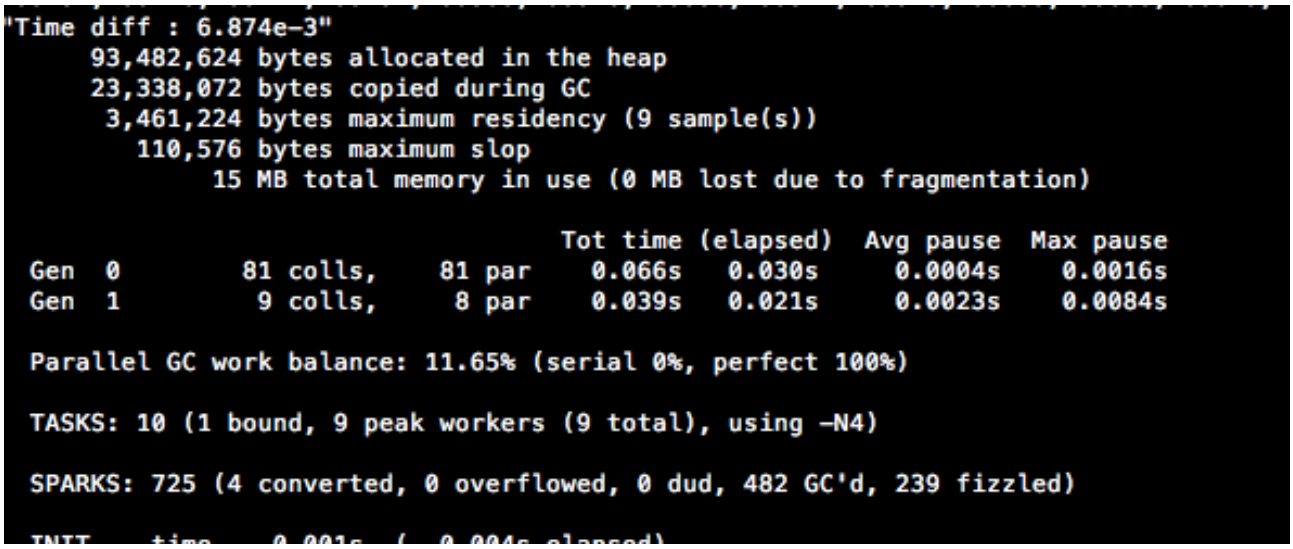
I did the same with quicksort as I did with mergesort. First, I included the sequential quicksort and added the “number of rounds” integer argument to the parallel quicksort. You’d expect my code to be something like this:

```
37  --QUICKSORT
38
39  --SEQUENTIAL
40  squicksort :: (Ord a) => [a] -> [a]
41
42  squicksort [] = []
43  squicksort (pivot:xs) = (squicksort lesserPart) ++ [pivot] ++ (squicksort
    greaterPart)
44      where
45          lesserPart = filter (<= pivot) xs
46          greaterPart = filter (> pivot) xs
47
48  --PARALLEL
49  quicksort :: (Ord a) => Integer -> [a] -> [a]
50
51  quicksort 0 lst = squicksort lst
52  quicksort _ [] = []
53  quicksort d (pivot:xs) = par lesserPart (pseq greaterPart ((quicksort (d-1))
    lesserPart ++ [pivot] ++ (quicksort (d-1)) greaterPart))
54      where
55          lesserPart = filter (<= pivot) xs
56          greaterPart = filter (> pivot) xs
57
58
```

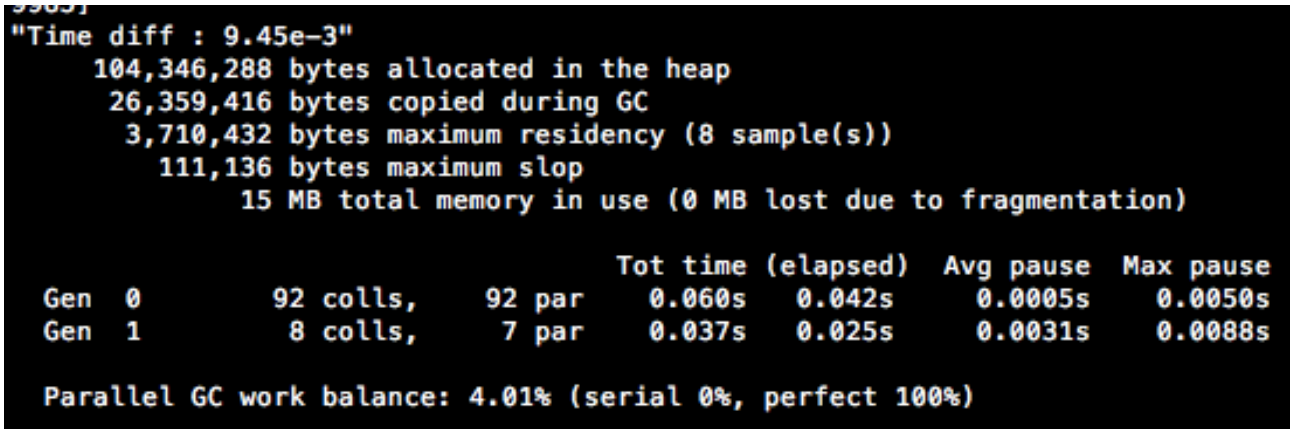
Restricting Spark overheads in quicksort

Just like 3 was the “magic number” for mergesort, passing 10 as the argument to quicksort seemed to work pretty well and yielded the slightest improvement in efficiency (if not negligible), of about 0.003s, which could be volatile. The efficiency could be considered to be about the same once again, but there’s something really interesting about the stats of quicksort in sorting3 compared with the stats of quicksort in sorting2.

This is discussed on the next page, for the purpose of ease of comparing the two stats



sorting3.hs : quicksort stats



sorting2.hs : quicksort stats

We discussed earlier that the runtime could be considered to be the same.

What's interesting is that, in sorting3.hs, where I tried to restrict the sparks by restricting the number of "rounds" or recursive calls that were made to the parallel version of quicksort, my parallelism (work balance) actually seemed to increase! In fact, it's almost triple the work balance of quicksort in sorting2.hs, whereas I initially thought it should have been the other way around.

Without a doubt, there are bits and pieces in my code screaming out to me, and perhaps might be more annoying to you. I think it's apt to end with a few changes I wish I could have made, in order to improve this project.

Further improvements:

1. These algorithms are just something I came up with. Sorting algorithms could probably be made much more efficient. For example, we could make quick sort choose a random element as a pivot instead of the first element.

2. The next improvement my files SCREAM OUT is that long 30,000 sized list pasted on one line. It should be replaced by a random list generating function.

3. Lastly, I wish I found a way to parallelise the merge function such that I got a speedup, if possible