LabA

Adam Olsson and Sebastian Frenzel Gabrielsson

April 2020

1 Assignment 1

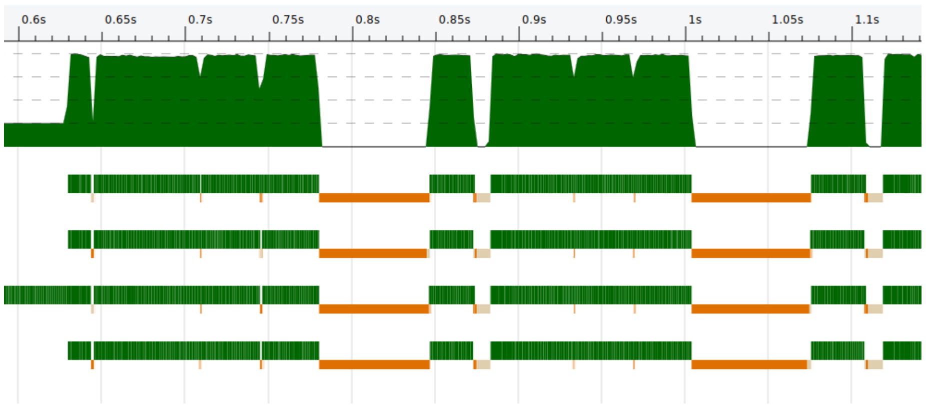
The implementations of parallel jackknife is quite straight forward. In a (pmapA) the first recursive call is parallelized which continues with the second arguement. In the second arg we've used the pseq operator to force the program to evaluate the next recursive call. Had we not done that, it is undefined if wether the recursive call construction of the list would have been evaluated first.

```
pmapA :: (a -> b) -> [a] -> [b]
pmapA f []     = []
pmapA f (x:xs) = par nf1 (pseq nf2 (nf1:nf2))
  where nf1  = f x
        nf2 = pmapA f xs
```

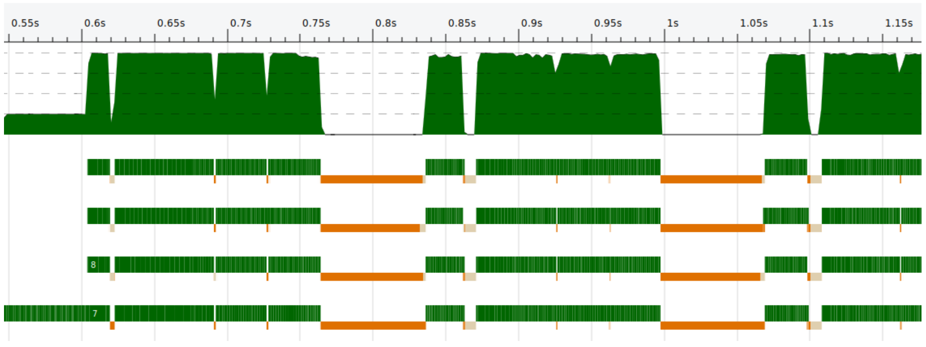
In b (pmapB) we used the Eval monad to wrap all applications of fx. Once the entire list has been iterated through, the jackknifeB function calls *runEval* do run the computation and extract the result from the monad.

```
pmapB :: (a -> b) -> [a] -> Eval [b]
pmapB f []     = return []
pmapB f (x:xs) = do
  nf1 <- rpar (f x)
  nf2 <- pmapB f xs
  return (nf1:nf2)


jackknifeB :: ([a] -> b) -> [a] -> [b]
jackknifeB f = runEval . pmapB f . resamples 500
```

Comparing to the built-par map, we can see that the elapsed time is roughly the same on the same input and cores. The only worth noting is that our implementation creates far less sparks and manages to convert all of them to parallel computations. Additionally, a comparison in thread scope does not reveal anything particular interesting.



(a) pmapB



(b) Built-in pmap

Figure 1: Threadscope comparison between our implementation (a) and the built-in parmap (b).

	pmapB	parMap
CPU Time	23.871s	24.596s
Elapsed Time	6.456s	6.597s
Sparks Created	121000	242000
Sparks GC'd	0	6488
Sparks Converted	121000	230474
Sparks Fizzled	0	5138

In c (pmapC) we build a strategy on a list from a strategy on a single element using the evalList function. The jackknifeC function separates the concern of logic and parallelization with the ‘using‘ function.

```
evalList :: Strategy a -> Strategy [a]
evalList _ []       = return []
evalList s (x:xs) = do
  x' <- s x
  xs' <- evalList s xs
  return (x':xs')

parList :: Strategy a -> Strategy [a]
parList s = evalList (rpar `dot` s)

jackknifeC :: (a -> b) -> [a] -> [b]
jackknifeC f xs = (map f xs) `using` (parList rseq)
```

Finally, in d (pmapD) we've used the Par Monad. The pattern of creating a reference pointer and forking the method have been extracted to the *spawn* function. pmapD then maps the forking of new processes over the list and returns a list of IVars. The IVar's value will eventually contain the result of each function application over the list respectively. Finally, the pmapD maps *get* over the list to wait for all parallel processes finish.

```
spawn :: NFData a => Par a -> Par (IVar a)
spawn p = do
  i <- new
  fork $ do x <- p; put i x
  return i

pmapD :: NFData b => (a -> b) -> [a] -> [b]
pmapD f xs = runPar $ do
  ibs <- mapM (spawn . return . f) xs
  mapM get ibs

jackknifeD :: NFData b => ([a] -> b) -> [a] -> [b]
jackknifeD f = pmapD f . resamples 500
```

2 Assignment 2

The first implementation of mergeSort uses the Par Monad. The main thread is forked during the divide step in mergeSort and then wait until both forked processes are finished. This is recursively done throughout the computation.

```
mergeSort1 :: (Ord a, NFData a) => [a] -> [a]
mergeSort1 [] = []
mergeSort1 [x] = [x]
mergeSort1 xs = runPar $ do
    i <- new
    j <- new
    fork $ put i $ mergeSort1 as
    fork $ put j $ mergeSort1 bs
    as' <- get i
    bs' <- get j
    return $ merge as' bs'
  where (as,bs) = splitAt (length xs `div` 2) xs
```

In the second implementation of mergeSort were unable to reach any satisfying speed-ups. The issue was that almost all sparks created were fizzled. The ratio between fizzled and converted spark could be everywhere between 1:3 to 1:1000. The things we tried were:

- Introduce granularity
- Before first divide step, only parallelize the left part of the computation and run the right half using rdeepseq
- For each recursive divide step, parallelize the left half of the computation and run rdeepseq on the right part of the computation
- Implement mergeSort using skeleton structure
- Implement mergeSort using a Tree data type

However, all these tries gave the same result. We are not sure for why so many sparks are fizzled but are starting to think that the hardware is simply fast enough that sparking computations might have too much overhead (even though sparking is cheap).

An example run using the following code:

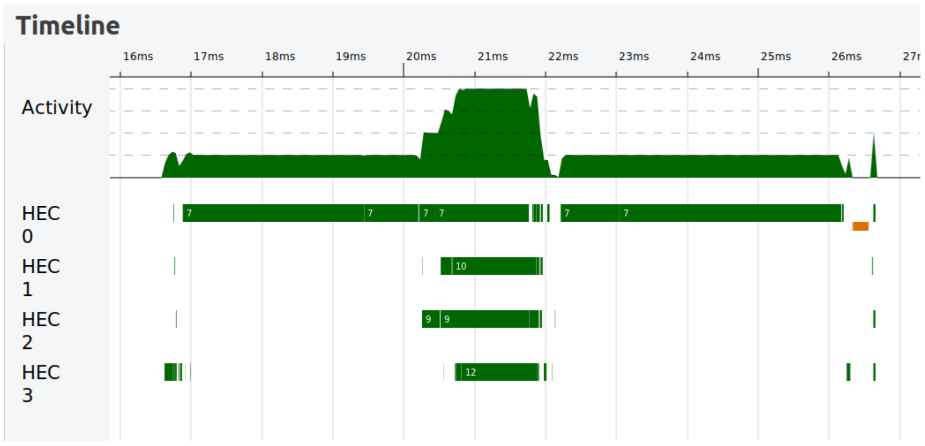
```
mergeSort2 :: (Ord a, NFData a) => Int -> [a] -> [a]
mergeSort2 _ []  =  []
mergeSort2 _ [x] =  [x]
mergeSort2 d xs | d <= 0 = merge l1 r1
                | otherwise = merge l1 r1 `using` strat
  where half    = (length xs `div` 2)
        (l0,r0) = splitAt half xs
        d'      = d-1
        l1      = mergeSort2 d' l0
        r1      = mergeSort2 d' r0
        strat x = do r l1; r r1; return x
          where r | half < 2  = rdeepseq
                  | otherwise = rpar

crud = zipWith (\x a -> sin (x / 300)**2 + a) [0..]

main :: IO()
main = do
  let xs =  (take 6000 (randoms (mkStdGen 211570155)) :: [Float] )
  let rs = crud xs

  print $ sum $ mergeSort2 10 rs  -- rpar and rseq
```

The code was compiled using:
ghc -O2 -threaded -rtsopts -eventlog -feager-blackholing filename.hs

Ran with:
./filename +RTS -N4 -A100M -s -ls

Resulting sparks:
SPARKS: 2046(16 converted, 0 overflowed, 0 dud, 0 GC'd, 2030 fizzled)

Resulting threadscope eventlog:


Timeline
16ms
17ms
18ms
19ms
20ms
21ms
22ms
23ms
24ms
25ms
26ms
27
Activity
HEC 0
HEC 1
HEC 2
HEC 3
7
10
9
12