The Par Monad for parallel Haskell programming: a tutorial

Markus Aronsson
mararon@student.chalmers.se

Emma Bogren
boemma@student.chalmers.se

Erik Bogren
erikbo@student.chalmers.se

Johan Toft
tojohan@student.chalmers.se

May 7, 2013

Contents

1 Introduction
–What this little tutorial has to offer

If you want to introduce parallelism to your Haskell code, in hope that it will run faster, there are many possible ways to approach this today. This tutorial will introduce you to one of them - the Par monad. Some knowledge about monads would be great, but you will probably be able to follow the idea presented and perhaps even use it yourself anyway after reading this.

The idea of the Par monad was presented by Simon Marlow, Ryan Newton, and Simon Peyton Jones in 2011, and it is entirely implemented as a Haskell library. As you have already figured out, it is based on a monad, but another important topic is I-structures. The Par monad uses I-structures, or IVars more specifically, for communication between tasks. The API offered is pretty simple to use, and resembles an API for concurrency. Concurrency and parallelism are not the same concept, but if you have every tried to program in concurrent Haskell before, writing parallel programs with the Par monad will probably be easier.

This tutorial will introduce you to the API of the Par monad, show you how it works and how to use it. A filter function and a quicksort algorithm will be written in purely sequential code, in order to explain how to make them parallel using the Par monad, and the speedup gained by doing so will be stated. Lastly, we will show examples of how you can create your own combinators, in a way to extend the current API (since it does not contain that many at the moment).

2 Saying hello to the API
–A presentation of the monad and IVars

"Our goal with this work is to find a parallel programming model that is expressive enough to subsume Strategies, robust enough to reliably express parallelism, and accessible enough that non-expert programmers can achieve parallelism with little effort." [1]

2.1 The API

While building parallel programs, where the order of computations are not known to us beforehand, the need for explicit dependencies can arise. In these cases it would be desirable to simply express the information flow throughout the program, leaving the nitty-gritty details of computation scheduling and such to the run-time system. Previously, concurrent Haskell has been used for this, but that also infers that the determinism is lost. Determinism is however a property that we would like to keep, as it makes reasoning about our programs much easier. For the purpose of filling this gap in current Haskell, the Par monad was introduced. The Par monad allows us to express our parallel algorithms in the pure language of Haskell without losing determinism. Furthermore, the Par monad is implemented entirely as a library within Haskell, which means we don't even have to learn any new and scary syntax to use it.

The Par monad

```haskell
newtype Par a

instance Functor     Par
instance Applicative Par
instance Monad       Par

runPar :: Par a  -> a
fork   :: Par () -> Par ()
```

Worth noticing is that the run function, runPar, takes a parallel computation on a type a and returns a result of the same type. While this might not seem as such a big deal at first glance it is actually really useful, since the result is pure it gives us a guarantee of determinism. However, one has to be careful when using the runPar function, since it is quite expensive and should generally be used to schedule large parallel tasks.

While the runPar function introduces a way to gain parallelism, we still need a way as a user to create parallel tasks. This is what the fork function is used for - you pass it an unevaluated computation in the Par monad and it evaluates this in parallel with the current computation. Forking new computations without a mean to retrieve their result is however quite pointless. To resolve this need for communication, the IVar type and its operations was introduced.

2.2 Concerning IVars

Don't let the word IVar scare you away - yes, it is another thing you will have to learn, but we will get you there. IVars are actually quite useful. In short, IVars are the Par monad's implementation of I-structures. So what are I-structures then?

[1] http://community.haskell.org/ simonmar/papers/monad-par.pdf

You can think of I-structures as a kind of array where each element can only be assigned a value once - if you try to 'put' a value more than once into an IVar, it will result in a runtime error. So avoid doing this. When you want to retrieve an element, it will return the same element each time and will never be empty. If the value has not been calculated yet when you try to retrieve it, the calling thread will simply be blocked until the value becomes available. However, since this type of data structure is not possible in purely functional data structures, it has to be part of a monad. It is not possible to "fill in" a variable once it has been created in purely functional code.

IVar

```
  newtype IVar

new    :: Par (IVar a)
put    :: NFData a => IVar a -> a -> Par ()
get    :: IVar a -> Par a
```

The interface for IVars represents the usual lifespan for the objects in the Par monad, where IVars are used for communicating results from forked processes. This is usually accomplished by forking a process that does some works and puts the result in an IVar, the parent process or other processes can then easily use that value by getting the result from the IVar. Worth noticing is the NFData (which stands for "normalform data") context on put, which means that put is fully strict. Hence, all values communicated through IVars are fully evaluated.

Since this pattern of forking a process which puts a value into a new IVar, and then using get to retrieve that value, is so common, a shorthand function for this pattern comes with the library. There are two versions of it - spawn and spawnP. You use spawn when the task you want to fork returns a monadic value, and spawnP is for forking pure computations rather than monadic.

spawn

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

  spawnP :: NFData a => a -> Par (IVar a)
  spawnP = spawn . return
```

IVars are great for describing data flow graphs, and making explicit what computations are shared down the line. You only have to define the structure of the data flow graph using spawns for nodes and get calls as arrows between nodes.

2.3 Data flow graphs – picture your code

As mentioned above, one could easily create data flow graphs from the code using the Par monad. This is a great way to learn how the API actually works. As an example, take the code

```
import Control.Monad.Par.Scheds.Trace
import Control.Monad.Par (NFData)

-- A silly function that forks computations
foo :: (Num a, NFData a) => a -> a
foo x = runPar $ do v1 <- new
              v2 <- new
```

```
             v3 <- new
             v4 <- new
                     fork $ put v1 (f x)
             fork $ put v2 (g x)
                     a <- get v1
             b <- get v2
             let v = a + b
             fork $ put v3 (h v)
             fork $ put v4 (q v)
             c <- get v3
             d <- get v4
             return (c*d)

-- Some very cool functions perhaps
f x =  ...
g x =  ...
h x =  ...
q x =  ...
```

The above code does not do anything useful - so don't worry if you don't understand what it is doing. It is the patterns in it that are important. You will learn how to make parallel useful programs eventually - actually already in the next chapter, so continue reading.

Four IVars are created, and then two new computations are forked. These can run in parallel, and will put their result in their given IVar when finished. The computation which forked them will wait for the results, and when it gets them it will add them. Then two other computations are forked in the same way. These computations are dependent on the results from the previously forked computations. When they are done, they put their results into their IVars, and the pro- ons can continue.

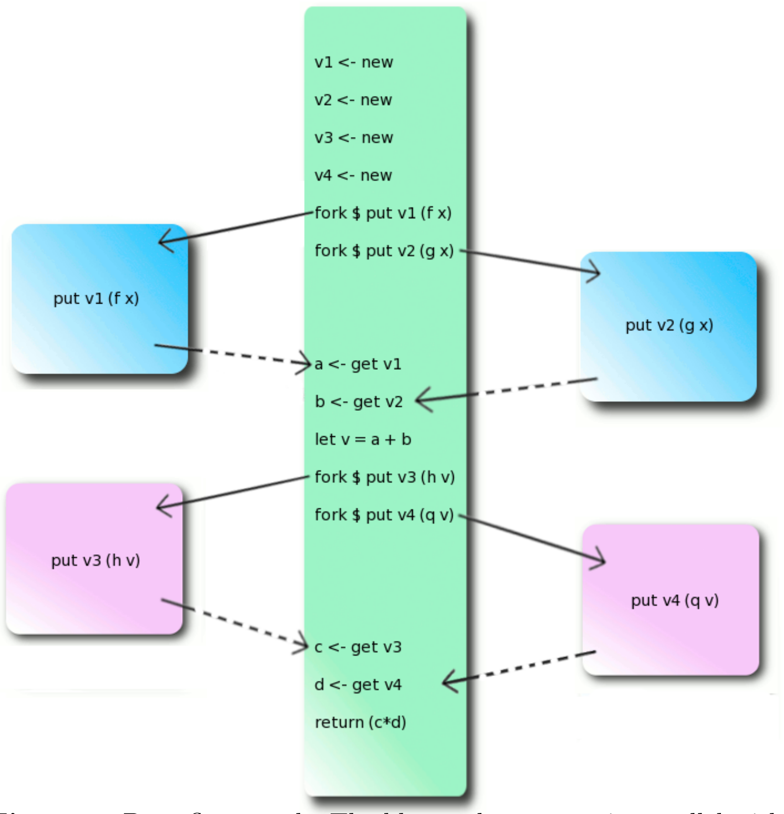

Figure 1: Data flow graph. The blue nodes are run in parallel with each other, and the pink nodes are run in parallel with each other. Arrows shows the forking of new threads, the dotted arrows show when a thread is waiting for an IVar.

This code be represented by the data flow graph in figure 1 below. Data flow graphs can look very different, and they all depend on the code of course. In this example, the main computation is represented by the green box in the middle, and arrows indicate forking of new computations. The new computations are displayed as small boxes, and boxes of the same color can run in parallel. The two blue boxes contains computations that can be run in parallel with each other, since there are no dependencies between them. There are dependencies between these and the code in the pink boxes though, and that is why these cannot run in parallel. The computations in the pink boxes does not depend on each other, so these can also run in parallel.

The arrows show, as mentioned, that a new child is forked, and this will run some given computation. The dotted arrows are created for each "get", and represents the flow of the data from a child computation to the parent computation through the IVar. The child computation will contain a "put" - it will put its result into the IVar when done.

3 Get your hands dirty
–introducing the Par monad into sequential code

Before we get started with the actual coding, it might be a good idea to learn how to compile and run parallel Haskell programs. This is how it's done from a terminal on a UNIX machine:

Compilation instructions

```
Compiling:
$> ghc -threaded -rtsopts -eventlog --make filename.hs
# -threaded - tells ghc that we want to run the program with multiple threads.
# -rtsopts - tells ghc that we want to be able to give flags to the runtime
    system.
# -eventlog - tells ghc that we want to be able to get a log of the program
    running that we can analyze.
# --make - generates a binary with the same name as the filename.

Running the program:
$> ./filename +RTS -Nx -ls
# +RTS - means that we want to give commands to the run-time system.
# -Nx - specify a number (x) of Haskell Execution Cores (HECs) to use
# -ls - tell the runtime system that we want an eventlog.
```

3.1 A simple example, using the divide-and-conquer pattern

We will begin with a really simple example of how to make a sequential program parallel with the Par monad. The code below describes a function that takes a list and constructs a new list containing all those elements in the first list that satisfies some condition (the function sent in should return true). Yes, it is a filter function. It is not the nicest piece of code, and you can probably think of better ways to write it, but for now we want to keep it simple.

```haskell
-- Sequential function for filtering a list
seq_filter :: (a -> Bool) -> [a] -> [a]
seq_filter _ []    = []
seq_filter f (x:xs)
  | f x            = x : seq_filter f xs
  | otherwise      = seq_filter f xs
```

So, how to make this parallel? We just want to remove all elements that does not satisfy a condition, and the filtering of one part of the input list should not depend on the filtering of another part of it. It seems like we could just split the list in two equally large parts and then just run the parallel filter function on both parts in parallel, in a divide-and-conquer fashion. If that is what we do, then we obtain the following code, using the Par monad:

```haskell
import Control.Monad.Par.Scheds.Trace
import Control.Monad.Par (NFData)
```

```haskell
-- Parallel filtering of a list
par_filter :: (NFData a) => (a -> Bool) -> [a] -> Par [a]
par_filter f []       = return []
par_filter f (x:[])   = if f x then return [x] else return []
par_filter f xs       = do
             let (as,bs) = halve xs
             v1<- spawn $ par_filter f as
             v2 <- spawn $ par_filter f bs
             left  <- get v1
             right <- get v2
             return $ left ++ right
                      where
                halve :: [a] -> ([a], [a])
                halve xs = splitAt (length xs `div` 2) xs
```

We spawn the two tasks of filtering the two parts of the list, to run these in parallel. When the results are received, all we do is concatenating the two filtered lists. This might work, but ... oh no! This function returns the answer that you wanted, but it is inside one of those creepy monads that you didn't think you would have to learn about. Well, do you remember the runPar function? We hope so, since this is one of the most important parts. You will not get a parallel computation without it. You run it on the parallel computation you got from your par_filter function in order to actually run the computation, and the answer you get will be a simple list.

The question is now, did we get any speed-up from making our function parallel? When benchmarking your code, make sure that you have imported the Control.Monad.Par.Scheds.Trace scheduler. At the time this tutorial was written, the standard scheduler in Control.Monad.Par is not working properly, and if you import it you might end up very frustrated thinking that there is something wrong with your code.

When we benchmarked the code on a quad-core computer, with 8 HECs, using a list of 200.000 random integers and the f function

```haskell
-- A costly operator
f :: (Num a, Ord a) => a -> Bool
f x = nfib 5 `deepseq` x < 10
  where
    nfib :: Integer -> Integer
    nfib n | n < 2 = 1
    nfib n = nfib (n-1) + nfib (n-2)
```

we compared the run times for the parallel filter and the sequential filter. The speedup gained was 0.53. But that is not even a speed-up, it is a slowdown! Why? There cannot possibly be anything wrong with the code, since it is so very beautiful and it uses the Par monad correctly. It must be something wrong with this whole concept - Simon Marlow and the others probably did a pretty lousy job. No, it's actually the code.

There is nothing wrong with the code - it is running, but there is one problem. Imagine spawning the filtering of the two sublists. These will in turn spawn the filtering of two sublists each, which will in turn spawn two filtering computations each etc etc. What you will end up with are an awful lot of subtasks, and they will all be quite small - even if the f function is very costly. We say that the function has a fine granularity. The tasks will actually be so small that the overheads from spawning tasks will outweigh the parallelism, and therefore we will gain no speed-up at all, but rather a slowdown.

Is all hope lost now? No, of course not - that would be some anti-climax. An even more costly f function would of course make the tasks larger, but what if we wanted to keep the function that we have now? We can actually do something about the granularity without changing the f. We could start off by chunking the input list in sublist of a given length, and then simply run the sequential filter function on each sublist in parallel. With a good size of the sublists, the tasks spawned will be large enough to outweigh the overheads, but also small enough to make use of parallelism.

```haskell
import Control.Monad.Par (parMap)

-- Parallel filter using chunking
par_filter2 :: (NFData a) => Int -> (a -> Bool) -> [a] -> Par [a]
par_filter2 n f xs = fmap concat $ parMap (seq_filter f) $ chunk n xs

-- Function chunking up a list into
-- sublists of size n
chunk :: Int -> [a] -> [[a]]
chunk n xs = as : chunk n bs
      where (as, bs) = splitAt n xs
```

This looks quite different from the previous parallel filter function - where are all the spawns? Actually, they are built in the parMap function that is used on the chunks. This is a function in the API for the Par monad, and it spawns one filtering task for each chunk. Then we concatenate the solutions, using fmap since we want to concatenate values that are inside a monad. What is the speed-up for this piece of code then? On a quad-core computer, with 8 HECs, and with chunk size 25.000, using 200.000 random integers, it is 3.83. This is actually quite good, and is better compared to the first solution without the chunking. Instead of chunking up the list, one could also keep the par_filter function with the divide-and-conquer pattern and just add a threshold to gain more coarse-grained tasks. We will see an example of how to use thresholds soon.

3.2 Parallel sorting using a threshold

After seeing the previous example, we are now ready for a more complicated problem. You have probably seen some kind of sorting function in your days and maybe in particular the quicksort function? If not, here is a short explanation.

The quick sort algorithm is what is known as a divide and conquer algorithm - it uses a so called pivot element (in our case it will be the head of the list) to divide the list into two parts - the elements less than x and the elements greater than or equal to x. It then sorts these two parts recursively, and this is the divide in divide and conquer. It then concatenates the two parts with the x in the middle in what is called the conquer stage.

```haskell
-- Sequential quicksort
quickSort :: Ord a  => [a] -> [a]
quickSort []        = []
quickSort (x:xs) = quickSort [y | y <- xs, y < x] ++
                              [x] ++
                   quickSort [y | y <- xs, y >= x]
```

We will now try to make a parallel quick sort using the Par Monad. It is a divide-and-conquer algorithm, and we saw an example of how to use this pattern for making a parallel filter function in section 3.2, and the solution here will look much like it. Since quicksort divides the list into two parts it seems only natural that we should run these recursive calls in parallel.

```haskell
import Control.Monad.Par.Scheds.Trace
import Control.Monad.Par (NFData)

-- Parallel quicksort
parQuickSort :: (Ord a, NFData a) => [a] -> Par [a]
parQuickSort [] = return []
parQuickSort (x:xs) = do  p1 <- spawn (parQuickSort (filter (<x) xs))
                          p2 <- spawn (parQuickSort (filter (>=x) xs))
                          left    <- get p1
                          right  <- get p2
                          return $ left ++  (x:right)
```

In this first attempt at a parallel quicksort function, we use the filter function (the one from Prelude - not our own anymore) to create lists containing elements lesser than the pivot element or greater than the pivot element. As was said earlier we make use of the fact that quicksort does two recursive calls when calling the spawn function two separate times. Notice how similar this parallel implementation looks to the sequential implementation and also how little code was actually added to make the algorithm parallel.

Benchmarking this code on a quad-core computer, using 8 HECs and sorting a list of 20000 random integers, the speedup is 1.33. It is a bit faster, but after reading the previous section, we know that we could probably make it run even faster if we change the granularity. We now introduce a threshold (d), which makes the search run in parallel to a certain point. After that, the sequential quicksort is used. This is once again in order to make the tasks more coarse-grained.

```haskell
-- Parallel quicksort using a threshold
parQuickSort2 :: (Ord a, NFData a) => Int -> [a] -> Par [a]
parQuickSort2 0 xs = return $ quickSort xs
parQuickSort2 d [] = return []
parQuickSort2 d (x:xs) = do p1 <- spawn (parQuickSort2 (d-1) (filter (<x) xs))
                            p2 <- spawn (parQuickSort2 (d-1) (filter (>=x) xs))
                            left <- get p1
                            right  <- get p2
                            return $ left ++  (x:right)
```

When benchmarking this code, different values for the threshold was tested to find the best one. The sequential quicksort was run in 11.78 ms, and the graph below shows the times for the parallel quicksort for the different values of the threshold.

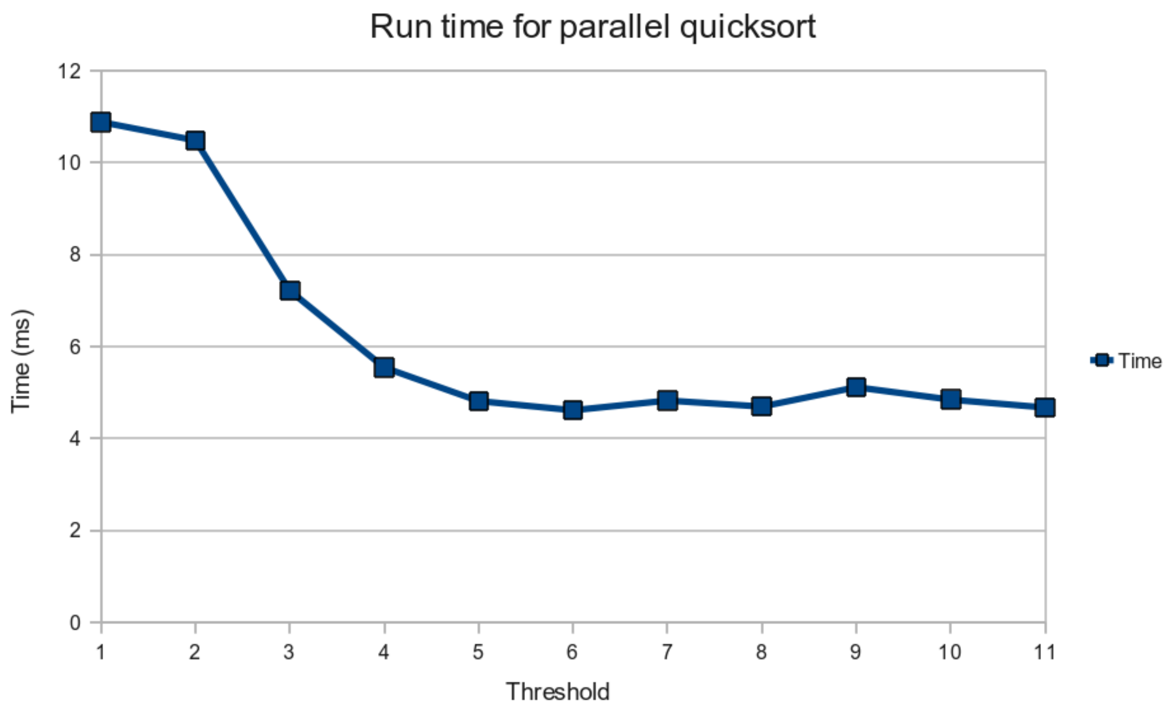

Figure 2: This graph illustrates the speedup of the parallel quicksort when applying varying thresholds. A threshold of 6 seems to be the way to go. The results were achieved on a quad-core CPU running 8 threads.

When running our code, we also used the -ls flag to generate an .eventlog file. This can be opened with the Threadscope tool. The graphs below were generated - the first is for the sequential quicksort, and the second is for the parallel quicksort with threshold 6, which was benchmarked to be the value that gave the best speed-up for the algorithm. The best speed-up gained was 2.56 (the parallel quicksort ran in 4.61ms). This is better than for the parallel algorithm without the threshold, as expected.

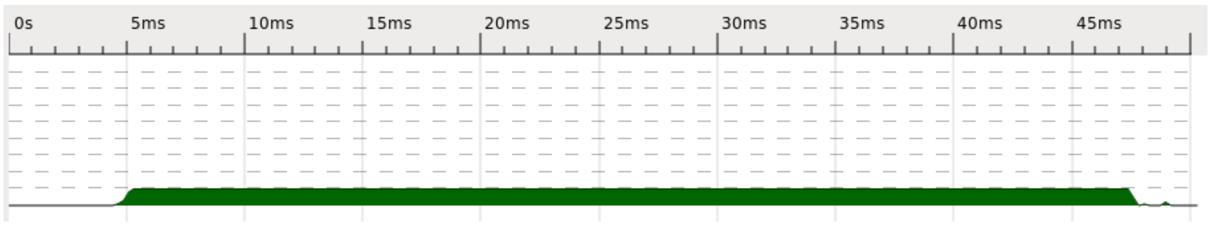

Figure 3: Sequential quicksort as seen in Threadscope. The striped lines shows us that there are 8 available threads but since this was running sequential code, only one thread was used.

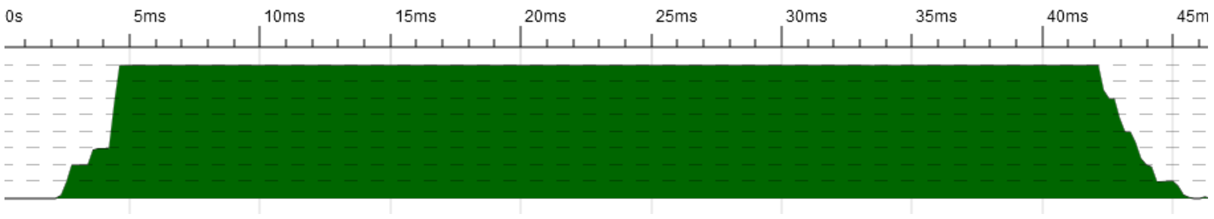

Figure 4: Parallel quicksort as seen in Threadscope. Now it should be obvious that we are running on a total of 8 threads. The picture shows that the code is able to run in parallel on all threads pretty much during the whole execution of the program.

Threadscope gives us a graphical view of the execution of our program; the green stuff is when a HEC is doing work and yellow stuff is garbage collection. So in the first picture, a single HEC is doing all the work, which is what we expect from a sequential computation. But what we want is a graph like the second one, where all HECs are working almost all the time, in order to make the run time shorter.

It is important to note that runtime of a parallel program might vary a lot for each execution of the code depending on what other tasks are running on the system. This is why it is a good idea to benchmark your code - if you just run the code once, the run time could be very different from the mean value of it obtained from benchmarking. Another thing worth mentioning about the quicksort algorithm is that it is actually a bit tricky to parallelize in a good way. The input data is not necessarily split evenly in each recursion - in some runs there can be bad pivot elements which infers that the work is divided very unevenly between the cores. This will slow down the execution.

4 Writing your own combinators –possible extensions of the API

The library Control.Monad.Par does not offer many parallel combinators for you to use at the moment. The only solution is that you write them yourself, but we will help you with the implementation of some of the most commonly used combinators, like zipWith and fold, and the filter function seen from chapter 2.1. One of the functions that the library actually does offer is the parallel map function

parMap

```
parMap :: (Traversable t, NFData b) => (a -> b) -> t a -> Par (t b)
parMap f xs = mapM (spawnP . f) xs >>= mapM get
```

While this works well for larger tasks, the performance gain gets smaller for shorter tasks. This is due to the granularity of the map function. Since it currently forks each element of the list, the overhead of forking can exceed the execution cost of a task when the tasks are small enough. To solve this we can compute chunks of the list in parallel, using the sequential map for each chunk.

Chunked parmap

```
import qualified Data.Traversable as T
import qualified Data.Foldable    as F
import Control.Monad.Par.Scheds.Trace
import Control.Monad.Par (parMap, NFData)
import Data.Traversable (Traversable)
import Data.Foldable   (Foldable)
```

```haskell
-- Parallel map over chunks
parMapChunk :: (NFData b) => Int -> (a -> b) -> [a] -> Par [b]
parMapChunk n f xs = fmap concat $ parMap (map f) $ chunk n xs

chunk :: Int -> [a] -> [[a]]
chunk _ [] = []
chunk n xs = as : chunk n bs
  where (as, bs) = splitAt n xs
```

This is similar to the way the filter function was constructed in section 2.1. - we just swap the inner map function to a filter function instead. This pattern of mapping a function over parallel chunks, and then combining the result, can be abstracted to a general function. Parallel map and filter could then be defined in terms of the more general function instead.

```haskell
-- General chunking function mapping a function in parallel to each chunk
parMapOpChunk :: (NFData a) => Int -> ([a] -> [b]) -> [a] -> Par [b]
parMapOpChunk n f xs = fmap concat $ parMap f $ chunk n xs

parFilterChunk n f = parMapOpChunk n . filter f
parMapChunk    n f = parMapOpChunk n . map    f
```

Another useful function from the Prelude module is the zipWith function. As the previous functions the zipWith function is also well-suited to be parallelized. However, zipping is slightly more complicated as we have to traverse two lists at the same time. In our implementation we kept the second list as a state, which we would pass on along while iterating the first list. Using an accumulative map function allowed us to do this without resorting to monads.

```haskell
-- Parallel zipWith function
parZipWith :: (Traversable t, Foldable f, NFData c)
                   => (a -> b -> c) -> t a -> f b -> Par (t c)
parZipWith comb t f = let step (x:xs) y = (xs, spawnP $ comb y x) in
          T.sequence . snd $ T.mapAccumL step (F.toList f) t >>= T.mapM get
```

Some other useful functions are the fold functions. This was however a bit more tricky than the previous operators, and some assumptions has to be made for any noticable parallelism to occur. Since folding is quite sequential in its definition we had to assume an associative operator to be able to apply a divide-and-conquer pattern to the folding. We recursively combine pairs of elements, and this is repeated until a single element remains, which is the reduced value.

```haskell
-- Parallel fold, requires an associative operator
parFold :: (Foldable f, NFData a) => (a -> a -> a) -> a -> f a -> Par a
parFold g a f = reduceRec g a (F.toList f)
      where
      reduceRec :: (NFData a) => (a -> a -> a) -> a -> [a] -> Par a
      reduceRec f a (x:[]) = return x
      reduceRec f a xs     = reduceRec f a $ runPar $ do
            T.sequence (reduce (\a b -> spawnP $ f a b) a xs) >>= T.mapM get

      reduce :: (a -> a -> b) -> a -> [a] -> [b]
      reduce f a (x:y:ys) = f x y : reduce f a ys
      reduce f a (x:xs)   = f x a : []
      reduce _ _ []       = []
```

If you have a bunch of parallel combinators like these, then you could simply take your sequential code and exchange the sequential combinators to the corresponding parallel combinators, and use the runPar function on the output. This is really simple, and the code looks very much like the sequential version. But on the other hand, the modularity is not that good, since the dynamic behaviour is not separated from the algorithm. It is just squeezed into it, but that is just something you have to live with if you are going to use the Par monad.

Appendix A

All the code:

A.1 Benchmarking the quicksort algorithm

Quicksort

```
import System.Random
import Control.Monad.Par.Scheds.Trace
import Control.Monad.Par (parMap)
import Control.Monad.Par (NFData)
import Criterion.Main

-- Create a list of n random integers
randomInts :: Int -> [Int]
randomInts n = take n . randoms $ mkStdGen 255

-- A list of 20000 integers
list :: [Int]
list = randomInts 20000

-- Use the Criterion library to run benchmarks
main :: IO ()
main = defaultMain [bench "quickSeq" $ nf (quickSort) list,
                    bench "quickNoThreshold" $ nf (runPar . parQuickSort) list,
                    bench "quickThreshold" $ nf (runPar . parQuickSort2 1) list,
                    bench "quickThreshold" $ nf (runPar . parQuickSort2 2) list,
                    bench "quickThreshold" $ nf (runPar . parQuickSort2 3) list,
                    bench "quickThreshold" $ nf (runPar . parQuickSort2 4) list,
                    bench "quickThreshold" $ nf (runPar . parQuickSort2 5) list,
                    bench "quickThreshold" $ nf (runPar . parQuickSort2 6) list,
                    bench "quickThreshold" $ nf (runPar . parQuickSort2 7) list,
                    bench "quickThreshold" $ nf (runPar . parQuickSort2 8) list]


-- Sequential quicksort
quickSort :: Ord a  => [a] -> [a]
quickSort []      = []
quickSort (x:xs)  = quickSort [y | y <- xs, y < x] ++
                              [x] ++
                     quickSort [y | y <- xs, y >= x]

-- Parallel quicksort, no threshold
parQuickSort :: (Ord a, NFData a) => [a] -> Par [a]
parQuickSort [] = return []
parQuickSort (x:xs) = do  p1 <- spawn (parQuickSort (filter (<x) xs))
                     p2 <- spawn (parQuickSort (filter (>=x) xs))
                     left    <- get p1
                     right   <- get p2
                     return $ left ++  (x:right)

-- Parallel quicksort using a threshold
parQuickSort2 :: (Ord a, NFData a) => Int -> [a] -> Par [a]
parQuickSort2 0 xs = return $ quickSort xs
parQuickSort2 d [] = return []
parQuickSor2t d (x:xs) = do p1 <- spawn (parQuickSort2 (d-1) (filter (<x) xs))
                      p2 <- spawn (parQuickSort2 (d-1) (filter (>=x) xs))
                      left <- get p1
                      right  <- get p2
                     return $ left ++  (x:right)
```

A.2 Benchmarking the filter function

```
import  Control.DeepSeq
import  Control.Monad.Par(parMap,NFData)
import  Control.Monad.Par.Scheds.Trace
import  Criterion.Main
import  qualified  Data.Traversable  as  T
import  qualified  Data.Foldable     as  F

-- Sequential function for filtering a list
seq_filter :: (a -> Bool) -> [a] -> [a]
seq_filter _ []   = []
seq_filter f (x:xs)
  | f x            = x : seq_filter f xs
  | otherwise      = seq_filter f xs

-- Parallel filtering of a list
par_filter :: (NFData a) => (a -> Bool) -> [a] -> Par [a]
par_filter f []       = return []
par_filter f (x:[])   = if f x then return [x] else return []
par_filter f xs       = do let (as,bs) = halve xs
                           v1<- spawn $ par_filter f as
                           v2 <- spawn $ par_filter f bs
                           left  <- get v1
                           right <- get v2
                           return $ left ++ right
                 where
      halve :: [a] -> ([a], [a])
      halve xs = splitAt (length xs `div` 2) xs

-- Parallel filter using chunking
par_filter2 :: (NFData a) => Int -> (a -> Bool) -> [a] -> Par [a]
par_filter2 n f xs = fmap concat $ parMap (seq_filter f) $ chunk n xs

-- Function chunking up a list into
-- sublists of size n
chunk :: Int -> [a] -> [[a]]
chunk _ [] = []
chunk n xs = as : chunk n bs
      where (as, bs) = splitAt n xs

-- Costly operator
f :: (Num a, Ord a) => a -> Bool
f x = nfib 5 `deepseq` x < 10
  where
    nfib :: Integer -> Integer
    nfib n | n < 2 = 1
    nfib n = nfib (n-1) + nfib (n-2)

-- Chunk size
n = 25000 :: Int
-- The list to be filtered
list = [1..200000] :: [Int]

-- Benchmarking sequential filter,
-- par_filter , and par_filter2
main :: IO ()
main = do
  defaultMain [bench "filter" $ nf (seq_filter f) list ,
               bench "parFilter" $ nf (runPar . par_filter f) list ,
               bench "parFilter2" $ nf (runPar . par_filter2 n f) list]
```

A.3 Implemented combinators

Combinators

```
import qualified Data.Traversable as T
import qualified Data.Foldable    as F
import Control.Monad.Par.Scheds.Trace
import Control.Monad.Par (parMap, NFData)
import Data.Traversable (Traversable)
import Data.Foldable  (Foldable)

-- Parallel map over chunks
--parMapChunk :: (NFData b) => Int -> (a -> b) -> [a] -> Par [b]
--parMapChunk n f xs = fmap concat $ parMap (map f) $ chunk n xs

-- Chunks a list up in sublists of size n
chunk :: Int -> [a] -> [[a]]
chunk _ [] = []
chunk n xs = as : chunk n bs
where (as, bs) = splitAt n xs

-- General chunking function mapping a function in
-- parallel to each chunk
parMapOpChunk :: (NFData a) => Int -> ([a] -> [b]) -> [a] -> Par [b]
parMapOpChunk n f xs = fmap concat $ parMap f $ chunk n xs

-- Using the general function to create parFilterChunk
-- and parMapChunk
parFilterChunk n f = parMapOpChunk n . filter f
parMapChunk    n f = parMapOpChunk n . map    f

-- Parallel zipWith function
parZipWith :: (Traversable t, Foldable f, NFData c)
                   => (a -> b -> c) -> t a -> f b -> Par (t c)
parZipWith comb t f = let step (x:xs) y = (xs, spawnP $ comb y x) in
           (T.sequence . snd $ T.mapAccumL step (F.toList f) t) >>= T.mapM get

-- Parallel fold, requires an associative operator
parFold :: (Foldable f, NFData a) => (a -> a -> a) -> a -> f a -> Par a
parFold g a f = reduceRec g a (F.toList f)
      where
      reduceRec :: (NFData a) => (a -> a -> a) -> a -> [a] -> Par a
      reduceRec f a (x:[]) = return x
      reduceRec f a xs     = reduceRec f a $ runPar $ do
            T.sequence (reduce (\a b -> spawnP $ f a b) a xs) >>= T.mapM get

      reduce :: (a -> a -> b) -> a -> [a] -> [b]
      reduce f a (x:y:ys) = f x y : reduce f a ys
      reduce f a (x:xs)   = f x a : []
      reduce _ _ []     = []
```