1 Base Code With No Parallel

We start the project by coding up the base code of solving bounded knapsack problem with dynamic programming approach in Haskell. Here, we take the index of the returned array from solve to be the weight. Each element of that array is composed of a tuple with the first element as the maximum total value, and the second to be an array consists of tuples of item names and the quantity.

1.1 Data set and run the code

The command line for starting up the base code is:

```
./dp randomItems.txt 300
```

where randomItems.txt is an automatically generated text file of 10000 items with their information line by line in the format itemName, weight, value, quantitylimit. ItemName is randomly picked from words.txt contained in homework 3; Weight and value are random integers between 2 and 50; Quantity limit is an random integer between 1 and 5; And 300 is the weight limit for the backpack.

1.2 Base code runtime

Below shows the runtime output for the base code from running (code can be found in section 5)

```
./dp randItems.txt 300 +RTS -s
```


2 483,501,064 bytes allocated in the heap
3 808,853,736 bytes copied during GC
4 153,080,616 bytes maximum residency (8 sample(s))
5 421,080 bytes maximum slop
6 145 MB total memory in use (0 MB lost due to fragmentation)

<table>
<thead>
<tr>
<th>Max pause</th>
<th>Tot time (elapsed)</th>
<th>Avg pause</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gen 0</td>
<td>461 cols, 0 par</td>
<td>0.443 s</td>
</tr>
<tr>
<td></td>
<td>0.0077 s</td>
<td>0.490s</td>
</tr>
<tr>
<td></td>
<td>0.0011s</td>
<td>0.2429s</td>
</tr>
<tr>
<td>Gen 1</td>
<td>8 cols, 0 par</td>
<td>0.351 s</td>
</tr>
<tr>
<td></td>
<td>0.638s</td>
<td>0.0798s</td>
</tr>
</tbody>
</table>

From the above output, we can see that the runtime is pretty good for the unparallelized version of knapsack code. In below sections we tried to add parallelism into the code to improve the performance.

2 Analysis

The time complexity of the base code is $O(nw)$, where $n$ is number of items and $w$ is the weight. And corresponding to the code, there are two main loops:

```
1 solve = foldr myroll basearray
2 solu = map getbest [0..]
```

where "solve" fold through all items and "solu" maps the weights. Unfortunately foldr can only be parallelized if the function being folded is associative. In other words, the function must has type

```
1 a -> a -> a
```
to achieve parallelization. Since our function "myroll" with type

```haskell
myroll :: (Ix i, Num i) => Item -> Array Int (Int, [(String, Int)])
  -> Array i (Int, [(String, Int)])
```

is not associative, the foldr part cannot be parallelized. Therefore the only parallelism we can perform is the "map".

## 3 Developing Parallelized Algorithm

### 3.1 parMap

The first thought we have is to use runPar with parMap instead of the pure map function. Therefore, we tried with using parMap with runPar, which we replaced the line

```haskell
solu = map getbest [1..]
```

with

```haskell
solu = map fromJust (filter isJust (runPar \$ parMap getbest [0 ..ttl\$]))
```

And edited a couple places to using Monad. Then we run with two cores and the result shown in threadscope is:

![Threadscope](image)

Obviously the result is not satisfying. Therefore we tried another attempt.

### 3.2 Strategy: parList
This time we tried to used strategy with parList to parallelize the code. We replace

```
1 solu = map getbest [1..]
```

with

```
1 solu = withStrategy (parList rdeepseq) (map getbest [0..ttlwght])
```

And below is the result of running the code with 2 cores.

We can see that the runtime result is much better than the first version with parMap.

### 3.3 Strategy: parBuffer

We tried to further improve the parallel performance. And by looking at "spark states", we notices that in previous result, most of the sparks are overflowed, as shown below

```
parList sparks states
```

<table>
<thead>
<tr>
<th>Time</th>
<th>Heap</th>
<th>GC</th>
<th>Spark stats</th>
<th>Spark sizes</th>
<th>Process info</th>
<th>Raw events</th>
</tr>
</thead>
<tbody>
<tr>
<td>HEC</td>
<td>3831730</td>
<td>281</td>
<td>3815030</td>
<td>0</td>
<td>15335</td>
<td>1085</td>
</tr>
<tr>
<td>HEC</td>
<td>3638789</td>
<td>20</td>
<td>3628128</td>
<td>0</td>
<td>1274</td>
<td>9366</td>
</tr>
<tr>
<td>HEC</td>
<td>3470519</td>
<td>301</td>
<td>7443158</td>
<td>0</td>
<td>16609</td>
<td>10451</td>
</tr>
</tbody>
</table>

Therefore, we tried to use parBuffer, which is supposed to help regulating the number of sparks. So we replace

```
1 solu = withStrategy (parList rdeepseq) (map getbest [0..ttlwght])
```

with

```
1 solu = withStrategy (parBuffer 100 rdeepseq) (map getbest [0..ttlwght])
```
And the runtime result is shown below.

Unfortunately, the overflow problem is not solved, as shown below

```
parBuffer sparks states
```

<table>
<thead>
<tr>
<th>Time</th>
<th>Heap</th>
<th>GC</th>
<th>Spark stats</th>
<th>Spark sizes</th>
<th>Process info</th>
<th>Raw events</th>
</tr>
</thead>
<tbody>
<tr>
<td>HEC</td>
<td>Total</td>
<td>Converted</td>
<td>Overflowed</td>
<td>Dud</td>
<td>GC’d</td>
<td>Fizzled</td>
</tr>
<tr>
<td>HEC 0</td>
<td>3010000</td>
<td>263</td>
<td>3001480</td>
<td>0</td>
<td>1721</td>
<td>6535</td>
</tr>
<tr>
<td>HEC 1</td>
<td>3009398</td>
<td>262</td>
<td>3001179</td>
<td>0</td>
<td>972</td>
<td>6986</td>
</tr>
</tbody>
</table>

3.4 Strategy: Replace parList with parListChunk

From reading the chapter 3 in Parallel and Concurrent book, we learned that "If you see some overflowed sparks, it is probably a good idea to create fewer sparks; replacing parList with parListChunk is a good way to do that.” Therefore, we decided to replace the parList function with parListChunk and give the chunk number to be the weight limit input:

```haskell
solute = withStrategy (parListChunk ttlwght rdeepseq) (map getbest [0..ttlwght])
```

And we can see from the output that the number of overflowed sparks is largely decreased.
And below is the output of running this version with 2 cores.

<table>
<thead>
<tr>
<th>Technique</th>
<th>Memory</th>
<th>Converted</th>
<th>Spark stats</th>
<th>Spark sizes</th>
<th>Process info</th>
<th>Raw events</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>HEC</td>
<td>Total</td>
<td>Converted</td>
<td>Overflown</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>HEC 0</td>
<td>20000</td>
<td>235</td>
<td>3284</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>HEC 1</td>
<td>19997</td>
<td>210</td>
<td>3203</td>
</tr>
</tbody>
</table>

### 3.5 Strategy: Comparison between parList, parListChunk and parBuffer

Using N2 with weight limit 300, here’s the comparison table (Memory: Maximum heap size)

<table>
<thead>
<tr>
<th>Technique</th>
<th>Memory</th>
<th>Converted</th>
<th>Spark stats</th>
<th>Spark sizes</th>
<th>Process info</th>
<th>Raw events</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>HEC</td>
<td>Total</td>
<td>Converted</td>
<td>Overflown</td>
</tr>
<tr>
<td>parList</td>
<td>876M</td>
<td>301</td>
<td>7443158</td>
<td>0</td>
<td>16609</td>
<td>10451</td>
</tr>
<tr>
<td>parBuffer</td>
<td>1.2G</td>
<td>525</td>
<td>6002659</td>
<td>0</td>
<td>2693</td>
<td>13521</td>
</tr>
<tr>
<td>parListChunk</td>
<td>1.1G</td>
<td>445</td>
<td>6487</td>
<td>0</td>
<td>5</td>
<td>33060</td>
</tr>
</tbody>
</table>

### 4 Conclusion

For this knapsack Haskell project, we finished the base code with optimized dynamic programming techniques, and moved on more sophisticated research on parallelizing the code. After thinking about which part can be parallelized, we found out that no parallelization can be done with "foldr" function, leading to our decision about focusing on the "map" function.

Our first attempt starts with using parMap with runEval, as the Sudoku example on the slides; however, no spark is detected, which leads us to the next
attempt, using parList. With parList, a huge number of sparks are detected, while most of them are overflowed, but not converted. And this leads to our second attempt with parBuffer, which from reading the slides, this function should help regulating the number of sparks. However, the result shown above tells us that in our case, parBuffer doesn’t really help that much.

Then we moved on to our next option, parListChunk. According to the textbook and documentation, it is designed to help with the overflow spark situation by limiting the total number of sparks. And with the result shown above, the overflow situation has been alleviated.

To sum up, the original sequential solution already exhibits pretty good runtime, while we can see how parallelization helps with the performance overall.

5 All Codes

5.1 Base code

```haskell
import System.IO(readFile)
import System.Exit(die)
import System.Environment(getArgs, getProgName)
import Data.Array
import Data.Maybe

data Item = Item { name :: String, weight :: Int, value :: Int,
bound :: Int } deriving (Eq, Show)

type SumValue = Int

main :: IO ()
main = do args <- getArgs
  case args of
    [ filename, weightlimit ] -> do
      contents <- readFile filename
      print (knapsack (processFile contents) (read
weightlimit::Int))
    _ -> do
      pn <- getProgName
      die $ "Usage: " ++ pn ++ " <filename> <weight limit
number>"

processFile :: String -> [Item]
processFile s = map secondProcess $ lines s where
  secondProcess ch = Item (read (w !! 0)) (read (w !! 1)::Int) (read (w !! 2)::Int) (read (w !! 3)::Int) where
    w = words ch

knapsack :: [Item] -> Int -> (Int, [(String, Int)])
knapsack items ttlwght = (solve items) ! ttlwght
where
  solve = foldr myroll basearray
  infl = repeat (0, [])
  basearray = listArray (0, ttlwght) infl
  myroll item s = listArray (0,ttlwght) solu
  where
```
solu = map getbest [0..]
getbest w = maximum $ hd:tl
  where
    hd = s!w
    iname = name item
    iv = value item
    iw = weight item
    ib = bound item
    tl = [combine (iv * x, (iname, x)) (s!(w-iw*x)) | x <- [1..ib], iw*x < w]
  combine (a,b) (c,d) = (a+c,b:d)

5.2 parMap

import System.IO(readFile)
import System.Exit(die)
import System.Environment(getArgs, getProgName)
import Control.Monad.Par
import Data.Array
import Data.Maybe

data Item = Item { name :: String, weight :: Int, value :: Int, bound :: Int } deriving (Eq, Show)

main :: IO ()
main = do args <- getArgs
  case args of
    [filename, weightlimit] -> do
      contents <- readFile filename
      print (knapsack (processFile contents) (read weightlimit :: Int))
      return ()
    _ -> do
      pn <- getProgName
      die $ "Usage: " ++ pn ++ " <filename> <weight limit number>"

processFile :: String -> [Item]
processFile s = map secondProcess $ lines s where
  secondProcess ch = Item (read (w !! 0)) (read (w !! 1) :: Int) (read (w !! 2) :: Int) (read (w !! 3) :: Int) where
    w = words ch

knapsack :: [Item] -> Int -> (SumValue, [(ItemName, Count)])
knapsack items ttlwght = (solve items) ! ttlwght
  where
    solve = foldr myroll basearray
    infl = repeat (0, [])
    basearray = listArray (0, ttlwght) infl
    myroll item s = listArray (0, ttlwght) solu
      where
        solu = map fromJust (filter isJust (runPar $ parMap getbest [0..ttlwght])))
getbest \( w = \text{Just} \) maximum $ \text{hd:tl}$

where
- \( \text{hd} = s!w \)
- \( \text{name} = \text{name item} \)
- \( \text{iv} = \text{value item} \)
- \( \text{iw} = \text{weight item} \)
- \( \text{ib} = \text{bound item} \)
- \( \text{tl} = [\text{combine} (\text{iv} \ast x, (\text{name}, x)) (s!(\text{w-iw}\ast x)) \mid x \leftarrow [1..\text{ib}], iw\ast x < \text{w}] \)

combine \((a,b)\) \((c,d)\) = \((a+c,b: d)\)

5.3 parList

```haskell
import System.IO(readFile)
import System.Exit(die)
import System.Environment(getArgs, getProgName)
import Data.Array
import Data.Maybe
import Control.Parallel.Strategies
import Data.List

data Item = Item { name :: String, weight :: Int, value :: Int, bound :: Int } deriving (Eq, Show)

main :: IO ()
main = do args <- getArgs
  case args of
  [filename, weightlimit] -> do
    contents <- readFile filename
    print (knapsack (processFile contents) (read weightlimit :: Int))
    return ()
  _ -> do
    pn <- getProgName
    die $ "Usage: " ++ pn ++ " <filename> <weight limit number>"

processFile :: String -> [Item]
processFile s = map secondProcess $ lines s where
  secondProcess ch = Item (read (w !! 0)) (read (w !! 1) :: Int) (read (w !! 2) :: Int) (read (w !! 3) :: Int) where
    w = words ch

knapsack :: [Item] -> Int -> (Int, [[(String, Int)]])
knapsack items ttlwght = (solve items) ! ttlwght
  where
    solve = foldr myroll basearray
    infl = repeat (0, [])
    basearray = listArray (0, ttlwght) infl
    myroll item s = listArray (0, ttlwght) solu
      where
        solu = withStrategy (parList rdeepseq) (map getbest [0.. ttlwght])
    getbest \( w = \text{maximum} \) $ hd:tl
    where
```

9
5.4 parBuffer

```haskell
import System.IO(readFile)
import System.Exit(die)
import System.Environment(getArgs, getProgName)
import Data.Array
import Data.Maybe
import Control.Parallel.Strategies
import Data.List

data Item = Item { name :: String, weight :: Int, value :: Int, bound :: Int } deriving (Eq, Show)
type SumValue = Int

main :: IO ()
main = do args <- getArgs
  case args of
    [filename, weightlimit] -> do
      contents <- readFile filename
      print (knapsack (processFile contents) (read weightlimit :: Int))
      return ()
    _ -> do
      pn <- getProgName
      die $ "Usage : " ++ pn ++ " <filename> <weight limit number>"

processFile :: String -> [Item]
processFile s = map secondProcess $ lines s where
  secondProcess ch = Item (read (w !! 0)) (read (w !! 1) :: Int) (read (w !! 2) :: Int) (read (w !! 3) :: Int) where
    w = words ch

knapsack :: [Item] -> Int -> (Int, [(String, Int)])
knapsack items ttlwght = (solve items !! ttlwght)
  where
    solve = foldr myroll basearray
    infl = repeat (0, [1])
    basearray = listArray (0, ttlwght) infl
    myroll item s = listArray (0, ttlwght) solu
      where
        solu = withStrategy (parBuffer 100 rdeepseq) (map getbest [0.. ttlwght])
        getbest w = maximum $ hd:tl
          where
            hd = s!w
            iname = name item
```

39  \( \text{hd} = s!w \)
40  \( \text{iname} = \text{name item} \)
41  \( \text{iv} = \text{value item} \)
42  \( \text{iw} = \text{weight item} \)
43  \( \text{ib} = \text{bound item} \)
44  \( \text{tl} = \left[ \text{combine} \left( \text{iv} \times x, (\text{name, x}) \right) \left( s!\left(w-\text{iw} \times x\right) \right) \mid x \leftarrow [1..ib], \text{iw} \times x < w \right] \)
45  \( \text{combine} \left( a, b \right) \left( c, d \right) = \left( a+c, b:d \right) \)
5.5 parListChunk

```haskell
41  iv = value item
42  iw = weight item
43  ib = bound item
44  tl = [combine (iv * x, (iname, x)) (s!(w-iw*x)) | x <- [1..ib], iw*x < w]
45       combine (a,b) (c,d) = (a+c,b:d)
```

```haskell
5.5 parListChunk

import System.IO(readFile)
import System.Exit(die)
import System.Environment(getArgs, getProgName)
import Data.Array
import Data.Maybe
import Control.Parallel.Strategies
import Data.List

9   data Item = Item { name :: String, weight :: Int, value :: Int, bound :: Int } deriving (Eq, Show)
10  type SumValue = Int
11
12  main :: IO ()
13  main = do args <- getArgs
14     case args of
15     [ filename, weightlimit ] -> do
16     contents <- readFile filename
17     print (knapsack (processFile contents) (read weightlimit :: Int))
18     return ()
19   _ -> do
20     pn <- getProgName
21     die $ "Usage: " ++ pn ++ " <filename> <weight limit number>"
22
23  processFile :: String -> [Item]
24  processFile s = map secondProcess $ lines s where
25  secondProcess ch = Item (read (w !! 0)) (read (w !! 1) :: Int) (read (w !! 2) :: Int) (read (w !! 3) :: Int) where
26  w = words ch
27
28  knapsack :: [Item] -> Int -> (Int, [(String, Int)])
29  knapsack items ttlwght = (solve items) ! ttlwght
30  where
31    solve = foldr myroll basearray
32    infl = repeat (0, [])
33    basearray = listArray (0, ttlwght) infl
34    myroll item s = listArray (0,ttlwght) solu
35  where
36    solu = withStrategy (parListChunk ttlwght rdeepseq) (map getbest [0..ttlwght])
37    getbest w = maximum $ hd:tl
38  where
39    hd = s!w
40    iname = name item
41    iv = value item
42    iw = weight item
```
ib = bound item

tl = [combine (iv * x, (iname, x)) (s!(u-iv*x)) | x <- [1..ib], iv*x < w]

combine (a,b) (c,d) = (a+c,b:d)