Final Project Report
Bounded Knapsack Problem

Jingyuan Wang(jw3732), Shaohua Tang(st3207)

Fall 2019

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

```
(6501,[("betso",5),("contection",4),("orthoclastic",2),("macromeric
    ",1),("uprightly",5),("conjugium",1),("ardor",3),("Sestian",1)
    ,("noncondensation",2),("intellectible",2),("supposititious",3)
    ,("centripetalism",3),("oversolemn",1),("unestimated",4),("
    stancher",3),("tyrannicide",2),("archpretender",5),("klam",3)
    ,("meteoritic",1),("circulable",4),("productory",2),("
    noncoincident",2),("speckless",2),("Patarin",4),("foresaddle
    ",1),("riffraff",4),("frankness",2),("rationalistical",3),("
```

```
forebowline",1),("anidiomatical",4),("counterturn",1),("
pepticity",4),("proteroglyphic",3),("sponspeck",4),("
subdititious",3),("Janiculum",2),("fergusite",5),("
spermatogenic",2),("qualmy",2),("forgottenness",4),("iodocresol
",2),("esocyclic",2),("Castoridae",1),("wholeheartedly",3),("
homeoblastic",4),("quotationally",5),("intervalvular",3),("
plumbosolvency",4),("podophthalmitic",2),("kern",3),("atropic
",3),("hyposystole",2),("shama",2),("paucifolious",3)])
   483,501,064 bytes allocated in the heap
   808,853,736 bytes copied during GC
   153,080,616 bytes maximum residency (8 sample(s))
       421,080 bytes maximum slop
           145 MB total memory in use (0 MB lost due to
fragmentation)

                                   Tot time (elapsed)  Avg pause
  Max pause
Gen  0      461 colls,     0 par    0.443s   0.490s     0.0011s
    0.0077s
Gen  1        8 colls,     0 par    0.351s   0.638s     0.0798s
    0.2429s

INIT    time    0.000s  (  0.003s elapsed)
MUT     time    0.367s  (  0.387s elapsed)
GC      time    0.794s  (  1.129s elapsed)
EXIT    time    0.000s  (  0.004s elapsed)
Total   time    1.162s  (  1.522s elapsed)

%GC     time       0.0%  (0.0% elapsed)

Alloc rate    1,316,627,219 bytes per MUT second

Productivity  31.6% of total user, 25.4% of total elapsed
```

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:

```
solve = foldr myroll basearray
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

```
a -> a -> a
```

to achieve parallelization. Since our function "myroll" with type

```
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

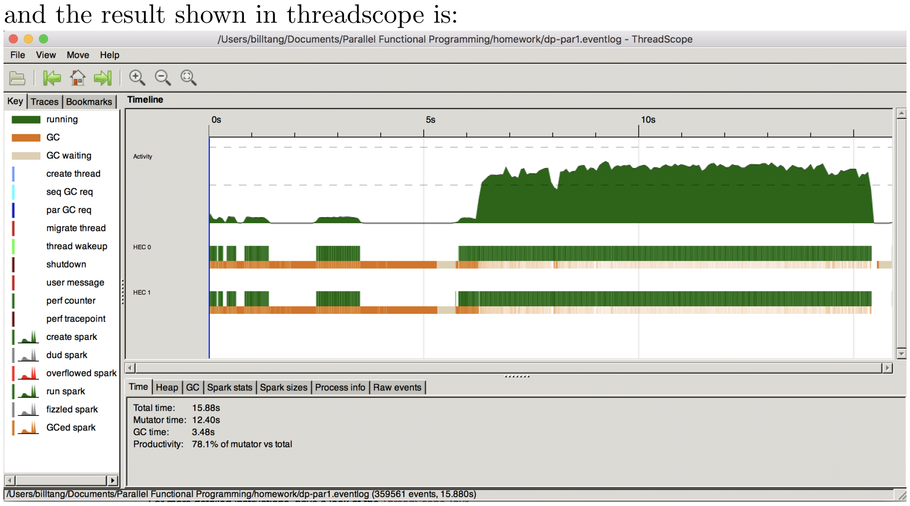
The first thought we have is to use runPar with parMap instead of the pure map function. Therefore, we tried with using parMap with runPar, which we replaced the line

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

with

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

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



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

3.2 Strategy: parList

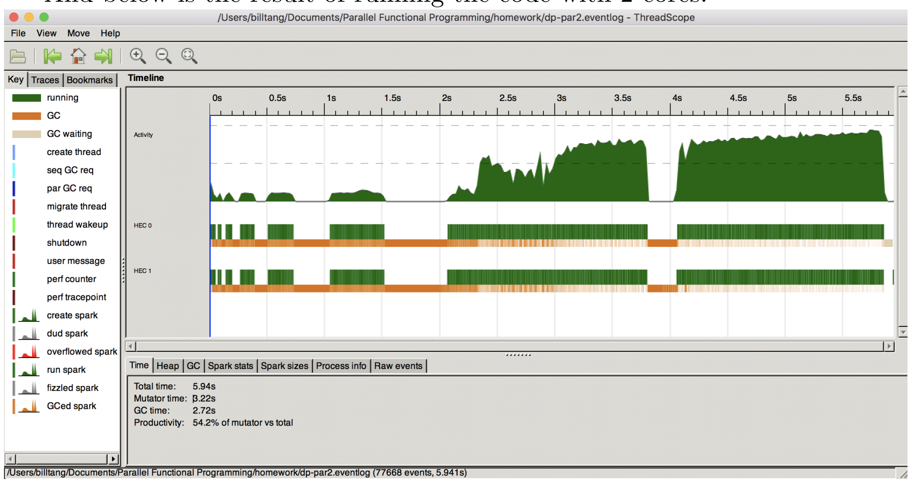
This time we tried to used strategy with parList to parallelize the code. We replace

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

with

```
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

HEC	Total	Converted	Overflowed	Dud	GC'd	Fizzled
Total	7470519	301	7443158	0	16609	10451
HEC 0	3638789	20	3628128	0	1274	9366
HEC 1	3831730	281	3815030	0	15335	1085

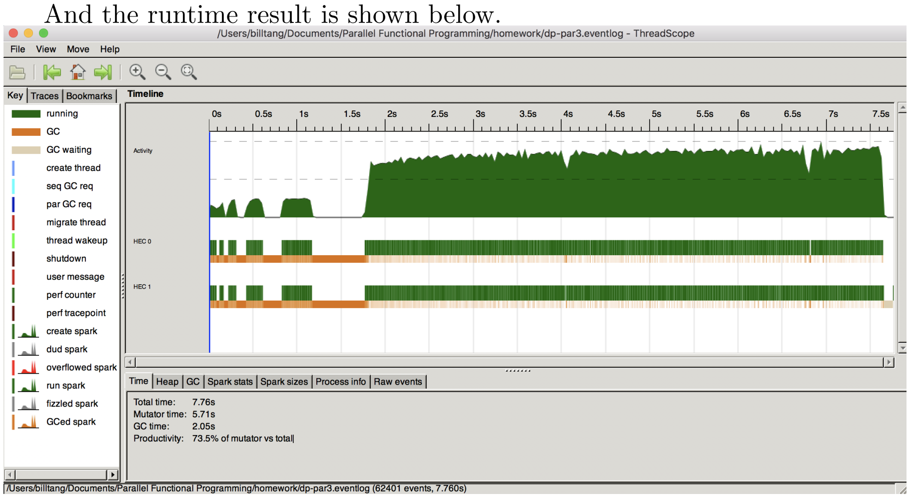
Therefore, we tried to use parBuffer, which is supposed to help regulating the number of sparks. So we replace

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

with

```
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

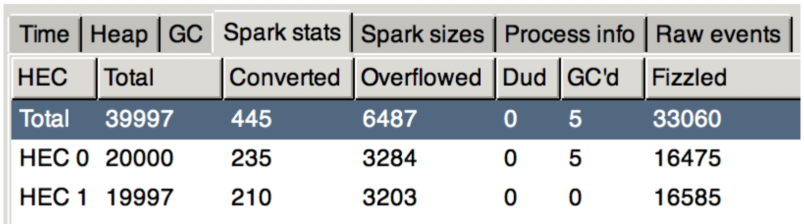
HEC	Total	Converted	Overflowed	Dud	GC'd	Fizzled
Total	6019398	525	6002659	0	2693	13521
HEC 0	3010000	263	3001480	0	1721	6535
HEC 1	3009398	262	3001179	0	972	6986

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:

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

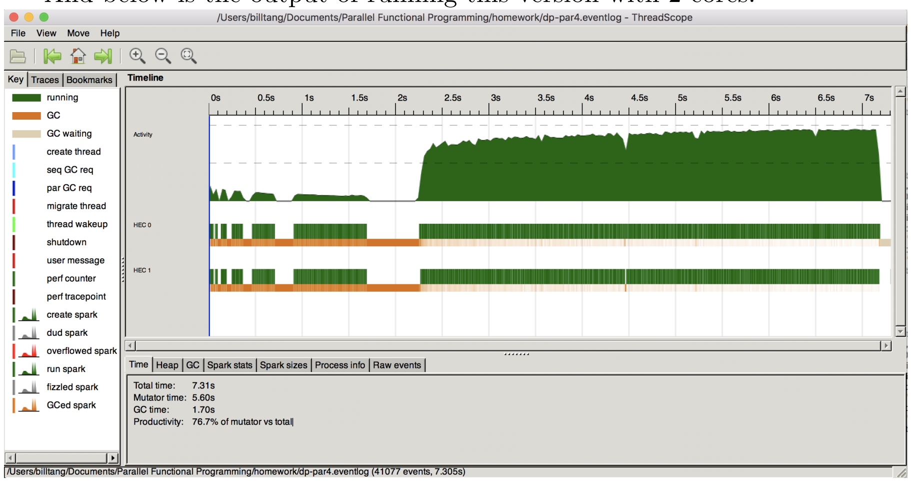
And we can see from the output that the number of overflowed sparks is largely decreased.



Time	Heap	GC	Spark stats	Spark sizes	Process info	Raw events

HEC	Total	Converted	Overflowed	Dud	GC'd	Fizzled
Total	39997	445	6487	0	5	33060
HEC 0	20000	235	3284	0	5	16475
HEC 1	19997	210	3203	0	0	16585

And below is the output of running this version with 2 cores.



3.5 Strategy: Comparison between parList, parListChunk and parBuffer

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

Techneque	Memory	Sparks					Time
		Converted	Overflowed	Dud	GC'ed	Fizzled	
parList	876M	301	7443158	0	16609	10451	5.94
parBuffer	1.2G	525	6002659	0	2693	13521	7.76
parListChunk	1.1G	445	6487	0	5	33060	7.31

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

```
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))
              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 = 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)
type SumValue = Int
type ItemName = String
type Count = 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 -> (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 = Just $ 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)
```

5.3 parList

```
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, [])
  basearray = listArray (0, ttlwght) infl
  myroll item s = listArray (0,ttlwght) solu
   where
    solu = withStrategy (parList rdeepseq) (map getbest [0..ttlwght])
    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)
```

5.4 parBuffer

```
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, [])
    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
```

```
            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)
```

5.5 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

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, [])
    basearray = listArray (0, ttlwght) infl
    myroll item s = listArray (0,ttlwght) solu
      where
        solu = withStrategy (parListChunk ttlwght rdeepseq) (map getbest [0..ttlwght])
        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)
```