J.T. Timpanaro
jtt2120
Parallel Functional Programming
December 18, 2019

Parallel Dynamic Programming Solution for 0-1 Knapsack Problem

Problem Statement

The Knapsack Problem is an optimization problem. Given a set of items, each with a weight and value, and a capacity, the goal is to choose a subset of items such that the total value is maximized and the total weight is less than the capacity. The 0-1 variation of the problem specifies that only zero or one of each item can be chosen.

The dynamic programming solution to this problem is to define a table as follows:

$tbl[i, j]$ represents the maximum attainable using the first i items with total weight less than or equal to j.

$tbl[0, j] = 0$ for all j.

$tbl[i, j] = tbl[i - 1, j]$ if new items weight is greater than j.

$tbl[i, j] = max(tbl[i - 1, j], tbl[i - 1, j - w] + v$ otherwise, where w is the weight and v is the value of the current item.

Since each row only depends on the previous (no row is self-referential), the opportunity for parallelization comes in the computation of the next row's values. Each row is computed sequentially, but within the row, the values will be computed in parallel.

Implementation

I/O

The *main* function handles the I/O. The program is run with two command line arguments: the test file path and the number of chunks to process in parallel. Test files are of the format: first line is a single Int representing the capacity, second line is whitespace separated Ints representing the weights, and the thrid line is whitespace separated Ints representing the values. The test file is then read into three variables: an Int representing the capacity, an [Int] representing the weights of the items, and an [Int] representing the values of the items. The decision to use the Int type instead of other numeric data types came down to simplicity and performance. The program would work with any numeric data type (if the type signatures were changed); however, since the object of the project is to show the improvement parallelization offers, Int is the best choice as it is simpler and faster than polymorphic and unbounded numeric data types.

The *main* function then calls the *knapsack* function, and prints the result in the format: first line is "Solution: [1,3,4]" where the numbers represent the 1-indexed items selected and the second line is "Total Value: 14" where the number represents the total value of these items.

Dynamic Programming Algorithm

The first function called is *knapsack*. *knapsack* takes three parameters: an Int number of chunks, an Int knapsack capacity, and an [(Int,Int,Int)] list of items. The list of items consists of three-tuples (a,b,c): a is the 1-indexed position of the item in the list, b is the weight of the item, c is the value of the item. *knapsack* is just a wrapper for the real algorithm runner. All *knapsack* does is use the capacity and number of chunks to determine the size of each chunk and create the first row of the dynamic programming table.

knapsack calls *kdp*, which is the runner of the dynamic programming algorithm. *kdp* takes three parameters: an Int chunk size, an [(Int,Int,Int)] list of items, and

a Seq (Int,Int,[Int]) table. I chose to use a Data.Sequence for the table because it performed significantly better than the list version and the Data.Vector version. This is most likely since Sequence outperforms lists in indexing and outperforms Vector in construction. The table only represents one row of the table at a time, since each new row only depends on the directly previous row. Each row consists of tuples (a,b,c): a is the column (total weight allowed), b is the maximum value for weight a, c is the current best solution (list of item numbers). *kdp* is a recursive function. If the items list isn't empty, it takes the first item and maps the *step* function curried with the current table and this item over the table. If the items list is empty, the last entry in the table is taken, and its total value and current solution elements are returned.

The *step* function is what computes the value for Table(Row i+1, Col j) given Table(Row i, Col j). It takes three parameters: the table (or row rather), the current item, and a specific entry in the table (column, value, current solution). It finds the entry in the table at column j-weight(item). If the current items weight isn't larger than the column and the value(entry)+value(item) is greater than the value in the current entry at this column, the value is updated to this new value and the item's number is added to the solution. Otherwise, the current entry remains unchanged.

Parallelization

The paraellization comes in the mapping step of *knapsack*. This mapping is called with *using* and my strategy *parSequenceChunk*. *parSequenceChunk* is a strategy I wrote that is essentially *parListChunk* adapted for sequences. It splits the sequence into chunks of a given size, then calls *parTraversable* over the chunks. The strategy passed to *parTraversable* is just *evalTraversable* with the strategy *rseq*. To summarize, the sequence is split into chunks, each chunk is handled in paraellel,

and within a chunk entries are handled sequentially with the strategy *rseq*.

Performance

Test Information

All performance tests were run on Test 8 (6404180 capacity and 24 items). Each test was run 25 times. Times are shown in seconds. SC represents spark conversion percentage.

Variable Cores, 1024 Chunks

Cores	Min Time	Max Time	Mean Time	Min SC	Max SC	Mean SC
1	58.289	63.322	59.580	0	0	0
2	37.371	42.363	38.703	99.821	99.890	99.857
3	26.196	28.105	27.050	99.882	99.915	99.902
4	21.881	23.132	22.629	99.882	99.919	99.901

As the number of cores increased, there is a clear speedup. However, there are diminishing returns on this improvement.

Variable Chunks, 8 Cores

Chunks	Min Time	Max Time	Mean Time	Min SC	Max SC	Mean SC
2	47.530	48.122	48.715	65.888	66.667	69.730
16	22.612	24.611	23.552	94.121	95.332	94.384
256	21.507	22.139	21.9304	98.397	98.461	98.449

Once again, as the number of chunks increased, there is a speedup. However, this has even less returns as the number of chunks increase. This makes sense, considering less parallel work is being done as the number of chunks increase, and it is to be expected at some point the performance will decrease as well. The major benefit of more chunks is the spark conversion percentage increases significantly as the number of chunks increases.

Code Listing

```
import Control.Parallel.Strategies
  ( Strategy
  , evalTraversable
  , parTraversable
  , rseq
  , using
  )
import Data.Sequence as DS
  ( Seq
  , ViewR((:>), EmptyR)
  , (!?)
  , (><)
  , chunksOf
  , iterateN
  , replicate
  , viewr
  , zip3
  )
import System.Environment (getArgs)
import System.Exit (die)
import System.IO.Error
  ( catchIOError
  , ioeGetFileName
  , isDoesNotExistError
  , isPermissionError
  , isUserError
  )
```

```
{-
    Read in data, call knapsack function, print result.
-}
main :: IO ()
main =
  do [filename, chunks] <- getArgs
     content <- readFile filename
     let n = read chunks :: Int
         [[c], wts, vals] = map toInt $ map words $ lines content
         (val, sol) = knapsack n c $ Prelude.zip3 [1,2 ..] wts vals
     putStrLn $
       "Solution: " ++ (show sol) ++ "\n" ++ "Total Value: " ++ (show val)
     `catchIOError` \e -> die $ handler e
  where
    toInt = map (\x -> read x :: Int)
    handler e =
     case ioeGetFileName e of
       Just fn
         | isDoesNotExistError e -> fn ++ ": File does not exist."
         | isPermissionError e -> fn ++ ": Permission denied."
       _
         | isUserError e -> "Usage: knapsack <filename> <# parallel chunks>"
         | otherwise -> show e


{-
    Wrapper to generate initial table and call dynamic programming algorithm.
-}
knapsack :: Int -> Int -> [(Int, Int, Int)] -> (Int, [Int])
knapsack n c items =
  kdp (c `div` n) items $
```

```
  DS.zip3
    (DS.iterateN (c + 1) ((+) 1) 0)
    (DS.replicate (c + 1) 0)
    (DS.replicate (c + 1) [])

{-
    Runner for knapsack dp algorithm.
-}
kdp :: Int -> [(Int, Int, Int)] -> Seq (Int, Int, [Int]) -> (Int, [Int])
kdp n (item:items) tbl =
  kdp n items ((fmap (step tbl item) tbl) `using` (parSequenceChunk n rseq))
kdp _ _ tbl =
  case DS.viewr tbl of
    EmptyR -> (0, [])
    _ :> (_, val, sol) -> (val, sol)

{-
    Computes value in next row for specific column.
-}
step ::
     Seq (Int, Int, [Int])
  -> (Int, Int, Int)
  -> (Int, Int, [Int])
  -> (Int, Int, [Int])
step tbl (i, w, v) (j, val, sol)
  | w > j || val > nval = (j, val, sol)
  | otherwise = (j, nval, osol ++ [i])
  where
    Just (_, oval, osol) = tbl !? (j - w)
    nval = oval + v
```

```
{-
    Divide sequence into chunks, then apply evalTraversable in parallel.
    Modified version of Control.Parallel.Strategies.parListChunk.
-}
parSequenceChunk :: Int -> Strategy a -> Strategy (Seq a)
parSequenceChunk n strat xs
  | n <= 1 = parTraversable strat xs
  | otherwise =
    fmap (foldr1 (><)) (parTraversable (evalTraversable strat) (chunksOf n xs))
```