3-Partition Problem

Hyeon U Kang hk3021

Parallel and Functional Programming: Fall 2019

Professor: Stephen Edwards

TA: Amanda Liu

Contents

1 Introduction

For the final project for the class, I created a program that solves the 3-Partition Problem utilizing parallelism. The executable takes in an input of a list of integers and returns a triple containing three lists of integers. The program attempts to partition the input list into three different subsets where each subset will have an equal total sum. 3-Partition Problem is an NP-Complete problem that could be solved in close to polynomial time through dynamic programming, but for this project, it was solved using a brute force method to study the impact that parallelism has on the performance.

2 Implementation

My implementation involves finding and examining all of the possible partitions to check for any partitions that divides the input into three subsets that sums up to the same number. The program immediately returns a valid partition when it is found. The program decides that a given input cannot be evenly split after checking through all of the possible partitions.

2.1 Original Implementation

My original implementation was inspired by the powerset function utilizing monad (equation 1). The idea behind the original approach was to find all of the possible scenarios where an element in the input is in one of the three subsets but not in the other two. All of the possible subsets were found by combining three subset equations into one comprehensive list of triples that each possible subset as elements in type format [(subsetFirst, subsetSecond, subsetThird)]. (equation 2,3,4).

$$powersetx = filterM(\backslash\_ \rightarrow [True], [False])x \quad (1)$$
$$subsetFirstx = filterM(\backslash\_ \rightarrow [True], [False], [False])x \quad (2)$$
$$subsetSecondx = filterM(\backslash\_ \rightarrow [False], [True], [False])x \quad (3)$$
$$subsetThirdx = filterM(\backslash\_ \rightarrow [False], [False], [True])x \quad (4)$$

After finding all of the possible partitions, the program iterated through the entire list until it found a partition that adhered to the conditions. Overall, this implementation was correct and returned a partition for any input if a partition was possible. Additionally, the code for this approach was very ”functional” and easy to read, but overall, it was very inefficient. There are many reasons behind this claim. First, this approach requires the program to find all of the possible partitions before determining the output, since the program does not check each partition in the final list while creating them. Therefore, in cases where 3-partition was possible, a lot of time was wasted by waiting to find all of the possible partitions before checking to see if any of them was valid.

Next, the code creates duplicates and unnecessary partitions. Unless the sum of the inputs add up to 0, there is no need for the program to check over or create any partitions that has an empty set as one of its subsets. For example, if the input is [1,2,3], there is no reason to consider the subset ([1,2][][3]), since it would be impossible for all of the subsets to be the same unless the total sum of the input is 0. The final list of all possible partitions also ended with a lot of duplicates. Because the same element of an input list cannot appear in multiple subsets, the ordering of the subsets does not matter. However, this implementation failed to take advantage of this characteristic. For example, with input of [1,2,3], the partitions ([1],[2],[3]) and ([2],[1],[3]) are equivalent. In fact, any permutations of those three subsets will be equivalent. However, this code did not have a way of determining that without more computation overhead.

Finally and most importantly, I realized that this approach will be very difficult make parallel. Even if I was able fix the first problem by creating a function that implements and improves on the filterM function, the code will be very linear and I will not have much room to implement parallelism. The only way for me to make this implementation parallel was to first compute all of the possible partitions first then running the different segments of the list in parallel. This resulted being a massive bottleneck for the program. Therefore, I had to implement an approach that is very similar to approaches done in ”traditional” languages that uses a recursive function to solve the problem.

Threads	Time (s)
1	169.585
2	85.419
3	78.469
4	73.109

Table 1: Number of threads vs performance for an input without a possible 3-partition

2.2 Final Implementation

By changing the skeletal code to a recursive function, I was able to solve two of the problems that I found in my initial implementations. I failed to fix the second error, and I am not sure if the second problem can be solved, since each possible partition must be first be created because it can be checked to be unnecessary or a duplicate. After many failed trials and errors, I have determined that it is just as fast to create and test all of the possible partitions including the duplicates and unnecessary partitions as it is to try to only find a pool of unique and more promising partitions.

My implementation starts with a triple of empty list. Then, the code traverses through the input list while calling itself three times placing the current element of the input list to each of the lists in the triple. A simplified implementation is demonstrated in equation 5.

$$partition(s : sx)(a, b, c) = partitionsx(s : a, b, c)||partitionsx(a, s : b, c)||partitionsx(a, b, s : c) \tag{5}$$

After traversing through all of the elements in the input list, each leaf in the recursive tree ends up with one of the possible partitions. Then, the program will check to see if each leaf has a valid partition that the code is searching for. If it is, the code returns the partition and halts. If none of the leaves of the recursion ends up being valid, it determines that there is no possible partition.

2.2.1 Parallel Programming

The program was made parallel by making each recursion create a new spark by using the rpar function. I also tested the performance using rseq to see if there is any improvements, but as I expected, recursing with rpar had far better results since each branch of the recursion are mostly independent from each other. The only time separate branches need to share information is when each branch has finished their computations and returns to the parent. Consequently, there was no need for me to make any sequential decisions while coding the program. I also added a depth parameter to the implementation in order to control spark production. By creating three new sparks at each branch, the spark creation became uncontrollable and there were too many sparks being fizzled or garbage collected.

3 Result and Analysis

There were three things that I was hoping to optimize while implementing parallelization to my implementation. First and foremost, I wanted the runtime to decrease as the number of threads increased. Then, I wanted to determine the most optimal depth to stop creating sparks in order to limit the number of sparks being fizzled or garbage collected. Finally, I wanted to make sure that the workload in each thread is similar and the work distribution is balanced.

3.1 Runtime

The runtime of the program is greatly varied depending on if the given input has a valid 3-partition or not. The program can halt when it finds one of the 3-partitions for a given input; however, if a given input does not have a possible 3-partition, the program must go through all of the possible partitions before determining that a 3-partition is impossible. Therefore, even before running tests, I hypothesized that it will take much

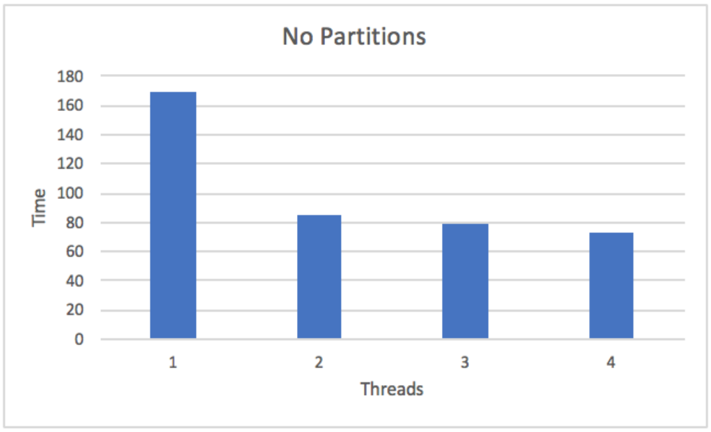

Figure 1: The graph above demonstrates the runtime in respect to different number of threads running when the program gets an input that has no possible 3-partition. Refer to Table 1 for exact figures.

Threads	Time (s)
1	4.336
2	4.318
3	5.939
4	7.567

Table 2: Number of threads vs performance for an input with a possible 3-partition

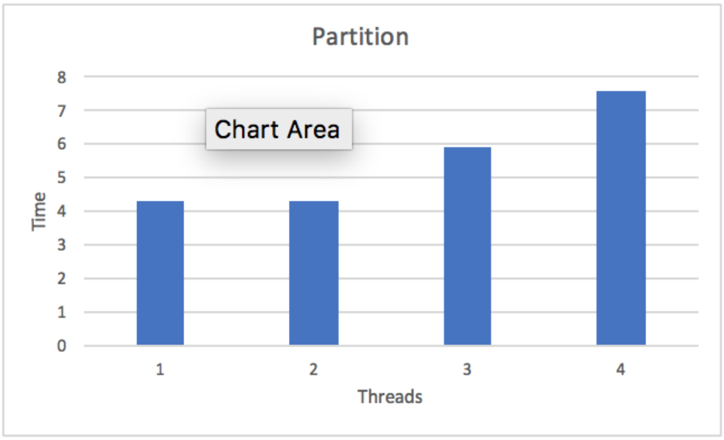

Figure 2: The graph above demonstrates the runtime in respect to different number of threads running when the program gets an input that has a possible 3-partition. Refer to Table 2 for exact figures.

longer to run test cases without a possible 3-partition than test cases with a possible 3-partition given they have the same number of elements.

Even though I successfully predicted the expected duration for two different classes of inputs, I was unable to predict the the effects parallelism will have on these two different classes. The test results indicate that the number of threads is inversely correlated to the runtime on inputs without a possible 3-partition while it is directly correlated to the runtime on inputs with a possible 3-partition. As you can see in Table 1 and Figure 1, the runtime decreases as the number of threads increases. However, as demonstrated by

Depth	Sparks	Converted	GC'd	Fizzled	Con Rate	Con to Fiz
1	3	2	0	1	67%	2
2	12	8	0	4	67%	2
3	39	21	0	18	54%	1.17
4	132	47	0	85	36%	.55
5	363	35	0	328	10%	.11
6	1096	39	0	1053	4%	.04
7	3279	84	0	3195	3%	.03
8	9846	89	0	9751	1%	<.01
9	29523	98	0	29425	<1%	<.01
10	88641	165	91	88385	<1%	<.01
15	21537774	378	21026107	511289	<1%	<.01
20	595606479	331	594106325	1499823	30	<.01

Table 3: Table demonstrating the sparks statistics for the different depths where the spark creation was stopped.

Table 2 and Figure 2, the runtime increases as the number of threads increases. This results was unexpected since I expected the runtime to decrease monotomically as the number of threads increased. I cannot claim with certainty that I know the reason behind this phenomenon; however, I predict that this resulted from the disjoint nature of each spark. At the first recursion, the work is divided to three different sparks where each spark considers a partition where the element of the input is placed in each of the three subsets. when compared to the list of all possible partitions created in the original implementation, it is equivalent to the first spark going through the first third, second spark recursing through the next third, and the third spark checking the last third. The program turns into a basic divide and conquer problem that is solved more efficiently utilizing parallel programming. Due to each partition having at least 2 duplicates, each of the original sparks will most likely run into at least one valid partition if the program is given an input with a valid partition and is allowed to run completely. Therefore, I predict that the amount of time it takes to find a valid partition should be similar no matter the number of threads that are utilized. Therefore, I predict that the additional runtime as the threads increases stems from increased overhead caused by parallelization.

Unlike the test cases with a possible 3-partition, cases without had expected results. It is only possible to declare that an input cannot be divided to three subsets of equal sum after going through all of the possible partitions, so it is expected for the runtime to decrease as the number of threads increases. The biggest improvement was found when the number of threads increased from one to two. However, it does not seem as beneficial to increase the number of threads after two. I attribute this to increased overhead of running a parallel program and to the fact that my machine only has two processors.

It does not seem beneficial to run this program in parallel when only considering results from test cases with valid partitions. However, the increase in runtime for test cases with possible partitions is very small compared to improvement in runtime for test cases with no valid partitions. One of the most important objectives of 3-Partition Problem is to determine if a given list can be partitioned into two subsets of equal total sum. Therefore, overall, I believe it will be beneficial to utilize multiple cores even if it means sacrificing performance for certain inputs.

3.2 Depth

The number of sparks that got garbage collected and fizzled was the biggest concern when I first implemented parallelism to my code. As Table 3 demonstrates, the number of sparks garbage collected and fizzled was insurmountable when the sparks were created beyond the 10th recursion call. Please note that the results were collected by testing it on a test case that did not have a possible 3-partition so that the program does not halt without going through all the possibilities. After first few initial tests, I realized that I cannot allow the code to create infinitely many sparks and had to put an upper bound. After testing the sparks generations based on the depth in which spark creation ends, I decided depth 4 was the most optimal. The

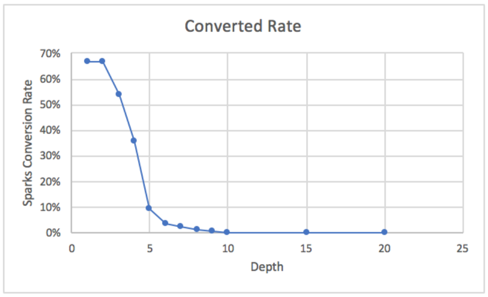

Figure 3: The graph above demonstrates the conversion rate based on depth.

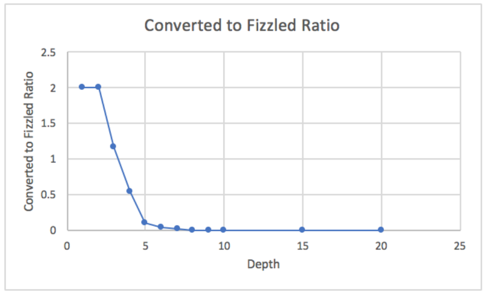

Figure 4: The graph above demonstrates the number of fizzled sparks based on depth.

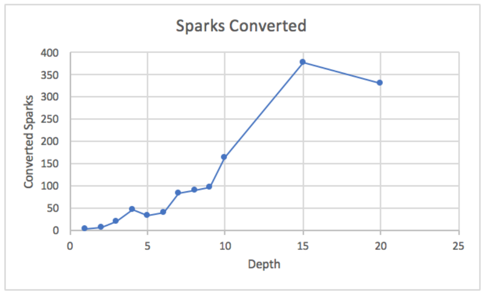

Figure 5: The graph above demonstrates the number of converted sparks based on depth.

reason is because the number of converted sparks almost doubled from depth 3 (Figure 5), and at the same time, the converted to fizzled ratio is still above .5. Also, as Figure 3 demonstrates, the conversion rate drops harshly from depth 4 to depth 5. The biggest concern was that number fizzled sparks quadrupled but that seemed to be normal trend as the depth increased. Finally when I ran a few test cases, the difference between depth 3 and 4 was minimal with a slight edge to depth 4.

3.3 Workload Balance

This program does not have much overhead before the parallel analysis starts, and the parallel analysis tend to be disjoint without any coordination until the very end after a spark is finished computing. The only time a thread must wait for other threads are cases where all three branches from a recursive call fails to find a valid partition. If any one of the three branches finds a valid partition, the program immediately returns it. Because there is minimal coordination between the threads, all of the threads are kept busy throughout the runtime. According Figures 6 and 7 which were exported from threadscope, it seems the workload balance was pretty good throughout the runtime. Due to promising results from threadscope, I did not spend as much time trying to implement rseq to my code. I tried to implement it in here and there, but overall, rseq implementations ended up being much slower than rpar implementation.

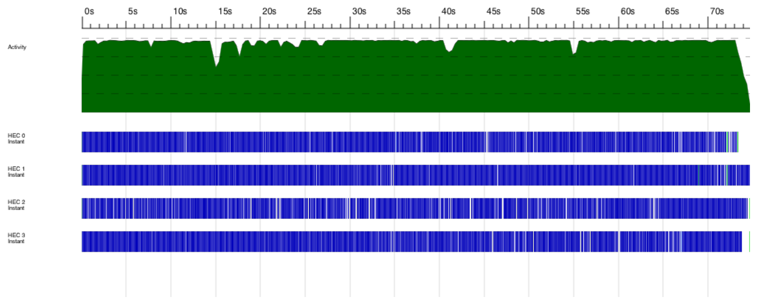

Figure 6: The graph above was exported from threadscope and it represents the instant events within each thread.

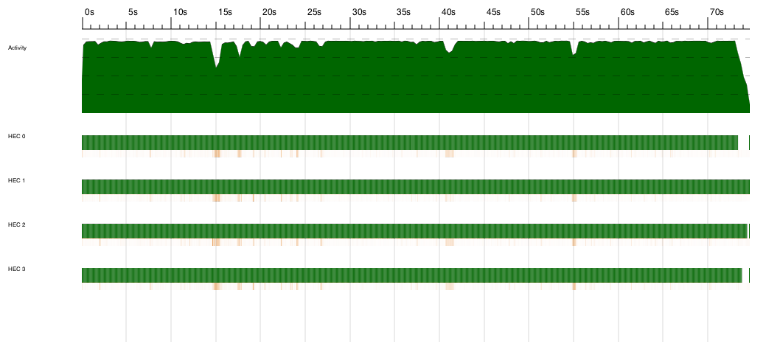

Figure 7: The graph above was exported from threadscope representing HEC traces.

4 Conclusion

By utilizing parallel computations, I was able to improve the runtime of the 3-Partition Problem by more than a factor of 2. Although there was an anomaly for cases with valid 3-partitions, on average, the program tends to run faster when parallelism is implemented. 3-Partition Problem was an excellent introduction to parallel functional programming due to the disjoint nature between the different sparks allowing for fast and efficient workload partition. At the same time, the implementation created unbounded number of sparks allowing me to experience and understand the negative consequences of constantly producing sparks without any considerations. If I had more time for the project, I like to try running the program in a stronger machine with more cores and create a more efficient partition algorithm that finds all of the possible partitions without any duplicates.

5 Code

```haskell
import Control.Parallel.Strategies
import System.Environment(getArgs)
import System.IO
import System.Exit(die)
import System.IO.Error(catchIOError, isUserError, isDoesNotExistError, ioeGetFileName, isPermissionError)

main :: IO ()
main = do
          [filename] <- getArgs
          h <- openFile filename ReadMode
          contents <- hGetContents h
          let list = inputToList contents
          putStr "Input: "
          print list
          printPartition $ threePartition list 4

      `catchIOError` \ e -> die $ case ioeGetFileName e of

       Just fn  | isDoesNotExistError e -> fn ++ ": No such file"
                | isPermissionError e -> fn ++ ": Permission denied"
       _        | isUserError e -> "Usage: partition <filename>"
                | otherwise -> show e

inputToList :: String -> [Integer]
inputToList = map read . words

printPartition :: Show a => Maybe ([a],[a],[a]) -> IO ()
printPartition Nothing = do putStrLn "There is no possible partition"
printPartition (Just x) = do putStr "Possible partition: "
                             print x

threePartition :: Integral a => [a] -> Integer -> Maybe ([a],[a],[a])
threePartition list x | length list < 3 = Nothing
                      | foldr (+) 0 list `mod` 3 > 0 = Nothing
                      | otherwise = threePartitionHelper x list ([],[],[])

threePartitionHelper :: Integral a => Integer -> [a] -> ([a], [a], [a]) -> Maybe ([a],[a],[a])
threePartitionHelper 0 s t = threePartitionNP s t
threePartitionHelper _ [] t = if check t then Just t
                              else Nothing
                              where check (a, b, c) = if x == y && y == z then True
                                                      else False
                                                      where x = foldr (+) 0 a
                                                            y = foldr (+) 0 b
                                                            z = foldr (+) 0 c
threePartitionHelper d (s:sx) (a, b, c) =  runEval $ do
                                            first <- rpar $ threePartitionHelper (d-1) sx (s:a, b, c)
                                            second <- rpar $ threePartitionHelper (d-1) sx (a, s:b, c)
                                            third <- rpar $ threePartitionHelper (d-1) sx (a, b, s:c)
                                            return $ output first second third
                                           where output Nothing Nothing Nothing = Nothing
                                                 output (Just i) _ _ = Just i
                                                 output _ (Just j) _ = Just j
                                                 output _ _ (Just k) = Just k

threePartitionNP :: Integral a => [a] -> ([a], [a], [a]) -> Maybe ([a],[a],[a])
threePartitionNP [] t = if check t then Just t
                        else Nothing
                        where check (a, b, c) = if x == y && y == z then True
                                                else False
                                                where x = foldr (+) 0 a
                                                      y = foldr (+) 0 b
                                                      z = foldr (+) 0 c
threePartitionNP (s:sx) (a, b, c) = output first second third
                                    where output Nothing Nothing Nothing = Nothing
                                          output (Just i) _ _ = Just i
                                          output _ (Just j) _ = Just j
                                          output _ _ (Just k) = Just k
                                          first = threePartitionNP sx (s:a, b, c)
                                          second = threePartitionNP sx (a, s:b, c)
                                          third = threePartitionNP sx (a, b, s:c)
```