Master's Programme in Data Science

Spark project

Mikko Saukkoriipi 013877851

November 26, 2020

University of Helsinki
Faculty of Science

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Degree programme
Faculty of Science	Master's Programme in Data Science

Tekijä — Författare — Author

Mikko Saukkoriipi 013877851

Työn nimi — Arbetets titel — Title

Spark project

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidantal — Number of pages
Report	November 26, 2020	13

Tiivistelmä — Referat — Abstract

This is my Spark project document. The given project questions were:

1. For the first data set (data-1.txt), provide the value of the median of the data set. You should provide the exact median, not an approximation, and sorting the complete data set is not an acceptable solution.

2. The second data set (data-2.txt) contains the matrix A. Your task is to calculate $\mathbf{A} * \mathbf{A}^\intercal * \mathbf{A}$ and provide the resulting matrix as your answer in the same format as the input matrix.

Avainsanat — Nyckelord — Keywords

Spark, Quickselect, Matrices

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Contents

1. Introduction

The spark project was more demanding and time-consuming than I expected. In the last period, I had done the course Big Data Platforms where we also used PySpark, and I thought that I know the basics of Spark. Unfortunately, I soon found out that I knew a lot less about Spark programming and it's libraries than I had thought. At the same time, I was fortunate to know Amdalh's law and how the program must be splittable, to be efficient in distributed architecture. I tried to keep this in mind when designing the program, but it was surprisingly tricky, and it will need much more practice.

For this project, I spent approximately 40 hours. The first 8 hours I spent on familiarising myself with how to connect to Ukko2, how to send files with ssh, how to write code in terminal (used vim), and how to write scripts. Before this project, I had only written code in Jupyter Notebook, and this was the first time writing scripts. The rest of the hours I spent on reading Spark manuals and testing on what works and what doesn't.

There was lots of confusion about the allowed tools. On the second last day, it was told that Mllib is permitted, which was a great help. In the future, I would hope for more precise guidance regarding this kind of thing.

I would have hoped that we would have more time to spent on this project. Now we had to write essays simultaneously and do projects and homework for other classes. As a result, I had to reduce the time I spent on reading articles and writing the essays.

Unfortunately, I got sick three days before the deadline, but I did what I could, and I am pretty satisfied with the results.

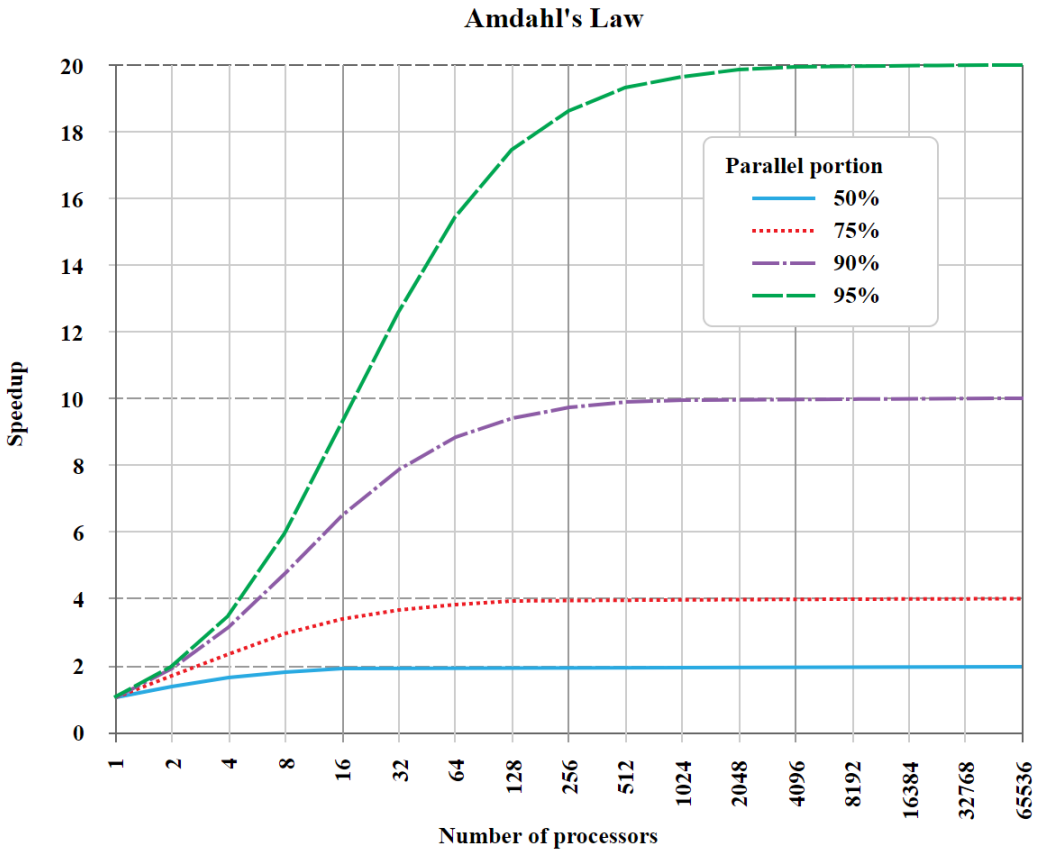

Figure 1.1: Amdahl's Law [2]

2. Finding the median

Finding the median without sorting was a trickier task than I expected. Next, I will introduce the models creating process and the results I got. I planned two different algorithms for this exercise, and the Quickselect style algorithm was implemented in Spark.

2.1 Searching median by counting

My first plan was to start counting the place in the set for the numbers. In other words, count how many numbers there are in the set that are less than the given number. Then continue this process until we found numbers in places k/2 and k/2+1, where k is number of numbers in list. Then finally calculate average of these middle numbers. I did implement this algorithm in Python3, and code is below.

```
import pandas as pd
from numpy import median
```

```
# Read the data
df = pd.read_csv("data-1-sample.txt", header=None)
df = df.rename(columns={"index": "value"})
df1 = df
```

```
# Count number of values less than value in row
def count_less(values, row):
    values = [value for value in values if value <= row[0]]
    cnt_values = len(values)
    return cnt_values

# Make list of all values
values = list(df1[0])

# Count how many values have smaller value
df1["cnt_less"] = df1.apply(lambda row: count_less(values, row),axis=1)

# Select two middle values (works if len is even)
df1 = df1.loc[(df1["cnt_less"] <= df1.shape[0]/2+1) & (df1["cnt_less"] >= df1.shape[0]/2)]

# Show df
df1
```

]:

	0	cnt_less
214	40.480026	500
703	40.497400	501

```
print("True median:", df[0].median())
print("My median:", sum(df1[0])/2)
```

```
True median: 40.48871315055
My median: 40.48871315055
```

This model would have needed two improvements to be efficient. The first im-

provement would be to add a break when middle numbers are found and not to run it through all numbers. The second improvement would be pre-filtering, where to drop part of the numbers. This could be done, for example, by dropping 30% of the smallest and largest numbers. This would have reduced unnecessary calculations.

I did not implement this prototype and its improvements because I started to worry about this model's efficiency. It is easy to distribute, but it does lots of unnecessary calculations. For this reason, I did not write this algorithm to Spark but changed my plan to the Quickselect algorithm.

2.2 Quickselect

I had read about the Quickselect algorithm before, but this was the first time I implemented it. Unfortunately, it was not as efficient as I had hoped for this task. It would have become more efficient with prefiltering and indexing moving of the pivot.

According to Wikipedia [3], Quickselect can reach the performance of $O(nlogn)$ to $O(n)$, but in the worst case, performance is $O(n^2)$. With Quickselect, I was able to get a sample set median in less than 30 seconds, but with the full dataset, it did not finish in 2 hours. Therefore I don't have a median for the full set, but I am sure that the algorithm works. I guess that with a given full-size dataset, the performance is close to $O(n^2)$, which is too slow.

I had plans on how to improve my QuickSelect algorithm. The first improvement would be to do pre-filtering, similar way as with my counting algorithm. The second improvement would be to make pivot smarter with indexing. Also, I started to think if QuickSelect even is efficient in distributed infrastructure with multiple cores. Unfortunately, I did not have more time to dig deeper into this problem.

median.py ● matrix.py ●

C: > Users > saukk > Desktop > Distributed data infrastucture > Spark project > median.py > ...

```python
from pyspark import SparkContext, SparkConf
import os
import sys
import math

# Datasets path on shared group directory on Ukko2.
dataset = "/wrk/group/grp-ddi-2020/datasets/data-1-sample.txt"
#dataset = "/wrk/group/grp-ddi-2020/datasets/data-1.txt"

conf = (SparkConf()
        .setAppName("mpsaukko")   ##change app name to your username
        .setMaster("spark://10.251.52.13:7077")
        .set("spark.cores.max", "10")  ##dont be too greedy ;)
        .set("spark.rdd.compress", "true")
        .set("spark.executor.memory", "32G")
        .set("spark.broadcast.compress", "true"))
sc = SparkContext(conf=conf)

# Read the data
data = sc.textFile(dataset)
data = data.map(lambda s: float(s))
data.cache()

# Define functio find_median
def find_median(data):
  count = data.count()
  middle = round(count / 2) + 1

  # If number of numbers in list is odd
  if (count % 2 != 0):
    return quickselect(data, middle)

  # If number of numbers in file is even, then calculate average of two middle numbers
  else:
    middle1 = quickselect(data, middle)
    middle2 = quickselect(data, middle - 1)
    return (middle1 + middle2) / 2


def quickselect(array, k):

  # Select pivot number
  pivot = array.takeSample(False, 1, seed=0)
  pivot = pivot[0]

  # Create two arrays. One with values more than pivot and one with values greater than pivot.
  numbers_less = array.filter(lambda x: x < pivot)
  numbers_greater = array.filter(lambda x: x > pivot)

  # If middle value found, then return pivot
  if k == (numbers_less.count() + 1):
    return pivot

  # If k<= number_less, then run quickselect again.
  if k <= (numbers_less.count()):
    return quickselect(numbers_less, k)

  # If k > number less, then run quickselect again.
  if k > (numbers_less.count() + 1):
    return quickselect(numbers_greater, k - (numbers_less.count() + 1))

# Find and print median
median = find_median(data)
print(" ")
print("--------- Median is: " + str(median) + " ''''''''''''")
print(" ")
```

Figure 2.1: My Spark median finding code

[25]:
```
import pandas as pd
from numpy import median
```

[26]:
```
# Read the data
A = pd.read_csv("data-2-sample.txt", sep=" ", header=None)

# Drop last column
A = A.iloc[:, :-1]

# Find A transpose
AT = A.T

# Calculate AAT
AAT = A.dot(AT)

# Calculate AATA
AATA = AAT.dot(A)

# Show first row
AATA.head(1)
```

Out[26]:

9	...	990	991	992	993	994	995	996	997	998	999
0552e+11	...	1.273313e+11	1.309680e+11	1.288276e+11	1.265437e+11	1.259945e+11	1.266185e+11	1.279121e+11	1.272910e+11	1.232555e+11	1.258357e+11

Figure 2.2: Python code to confirm correct result for matrix computation sample set

3. Matrix multiplication

The matrix multiplication task was more straightforward than the median calculation. The median calculation task restriction to avoid sorting in, by all means, made the designing of the algorithm challenging, but with matrix multiplication, there was no such issue. Now we did not have such limitations, but the limits came from the tools that Spark was able to offer. My first plan was to calculate multiplications with spark.sql queries, but later it was revealed that Mllib is also allowed to use. After this, I changed the plan and implemented the whole algorithm based on Mllib.

3.1 Matrix multiplication with SQL

My first plan was to do this task bases on Spark SQL queries. Matrix multiplication can be calculated with Spark SQL with code below (code is not tested).

results = spark.sql("SELECT A.row row, B.col col, SUM(A.val * B.val) val FROM A JOIN B ON A.col = B.row GROUP BY A.row, B.col")

As can be seen, this multiplication made this way is pretty straight forward. But there is a problem with efficiency because it can not be distributed to Spark workers efficiently. Fortunately, it was revealed that we are allowed to use Mllib [1], and so I abandoned this idea.

3.2 Matrix multiplication with MlLib

I had no previous experience with matrix computations on Spark nor Mllib, which revealed a more challenging task than expected. Mllib offers all the needed tools for this task, but it requires a sophisticated combination of the Mllib matrices and operations.

As I had no previous experience in this kind of work, I spent most of the time reading the Mllib and Spark manuals. Different Mllib matrix classes are meant for different tasks and have different properties, and it was important to find the most efficient combination. With my first Mllib based code, I got the correct result for the

sample set in 1 minute. After changing the used matrices, reducing the number of steps, adding caching and tuning block matrices block sizes, I squeezed this time to 20 seconds.

Unfortunately, I was not able to get results for the full dataset. With 10 cores and 16gb of memory, it will end up with a no memory left error. This is a surprising result because all the steps are made with Spark's native tools. Spark offers so-called sparse and dense matrices, but our only option was to use a dense matrix. After further research, I found there are better libraries available for large-scale matrix multiplications. Unfortunately, we did not have a chance to use those tools in this task.

C: > Users > saukk > Desktop > Distributed data infrastucture > Spark project > matrix.py > ...

```python
from pyspark import SparkContext, SparkConf, mllib
import os
import sys
from pyspark.sql import SparkSession
from pyspark.mllib.linalg.distributed import IndexedRowMatrix, CoordinateMatrix, MatrixEntry, RowMatrix

# Select the dataset
#dataset = "/wrk/group/grp-ddi-2020/datasets/data-2-sample.txt"
dataset = "/wrk/group/grp-ddi-2020/datasets/data-2.txt"

conf = (SparkConf()
        .setAppName("mpsaukko")
        .setMaster("spark://10.251.52.13:7077")
        .set("spark.cores.max", "8")  ##dont be too greedy ;)
        .set("spark.rdd.compress", "true")
        .set("spark.executor.memory", "64G")
        .set("spark.broadcast.compress", "true"))

sc = SparkContext(conf=conf)
spark = SparkSession.builder.config(conf=conf).getOrCreate()

# Read the data
A = sc.textFile(dataset)
A = A.map(lambda s : [float(x) for x in s.split()])

# Zip index values with cell values
A = A.zipWithIndex().map(lambda x: (x[1], x[0]))

# Print step 1 ready. With full set 1min.
print(" ")
print("Step 1 ready")
print(" ")

# Conver A to IndexedRowMatrix
A = IndexedRowMatrix(A)

# Convert A to blockmatrices and set block size.
#A = A.toBlockMatrix(1000, 1000) # Works with sample set. Data to 1 block.
A = A.toBlockMatrix(100, 1000)

# Cache A, because it is used multiple times
A.cache()

# Print step 2 ready. With full set 3mins.
print(" ")
print("Step 2 ready")
print(" ")

# Next multiplications. We need to calculate A*AT*A.
# Size of the A is 1000000 x 1000 matrix so size of the A*AT would be 100000*100000.
# For better performance use matrix multiplication rule (A*AT)*A = A*(AT*A)

# Calculate A transpose
AT = A.transpose()

# Make firts multiplication AT*A
ATA = AT.multiply(A)

# Print step 3 ready. With full set 9mins.
print(" ")
print("Step 3 ready")
print("ATA rows and cols:")
print("Rows:", ATA.numRows() , "Cols:", ATA.numCols())
print(" ")

# Make second multiplication A*ATA
AATA = A.multiply(ATA)

# Print step 4 ready
print(" ")
print("Step 4 ready")
print(" ")

# Convert AATA to indexRowMatrix
AATA = AATA.toIndexedRowMatrix()

# Get first row from AATA
first_row = AATA.rows.filter(lambda x: x.index == 0).first().vector.toArray()

# Print first row
print(first_row)
```

Figure 3.1: My Spark matrix multiplication code

4. Conclusions

It was an exciting project, and I learned lots of new and practical skills while writing these codes. Unfortunately, the number of exercises, essays, and projects during the last two has been total horror. Unfortunately, I could not get answers in either of the exercises with a full-size dataset. Nevertheless, I am happy that I could get both codes to work with sample data, especially when considering the limited amount of time I was able to spend on this project.

References

[1] Apache spark mllib, 2020. [Online; accessed 26-November-2020].

[2] Wikipedia contributors. Amdahl's law — Wikipedia, the free encyclopedia, 2020. [Online; accessed 26-November-2020].

[3] Wikipedia contributors. Quickselect — Wikipedia, the free encyclopedia, 2020. [Online; accessed 24-November-2020].