Assignment 3

Wanyi Su

Student # 301445656

Q1.

Wordcount-5 has 8 unevenly-split files, in which the data files are skewed. We have 8 executors. With no partitioning, the executors, which run relatively small data files and finish in advance, would have to wait for others who run large data files to finish the whole tasks. This doesn’t ensure complete resource utilization on executor’s nodes. This would increase the processing time to an unnecessary longer time.

With repartitioning, we can partition the data files into smaller ones and distribute them across executors. An executor can complete smaller files faster than before and then another partitioned file is passed in and is processed. Therefore, no executors are idle and all executors would keep working on smaller data files with a faster speed. This would result in faster running time of the whole program.

Q2.

Wordcount-3 has 101 relatively evenly-distributed data files. If we do repartition on wordcount-3, we would have too much intense task scheduling, which is unnecessary for those data files. The shuffle cost causes by repartitioning and too many files can be an extra burden to the program. Hence, the repartitioning doesn’t work to wordcount-3.

Q3.

We can firstly use sc.saveAsTextFile() to create an RDD with the original data file, and then partition the data into smaller and not skewed pieces of RDD. Then we start our program and read data from this new file.

Q4.

On lab computers, I got results as below. If we graph the results, we can get a convex curve with a flat valley in between. Therefore, the appropriate range of partition# can be [6, 600]. The optimal repartitions can be 7 – 10.

Euler – on lab computer (with 100000000 sample inputs):

2 par: 36.454s

3 par: 25.452s

5 par: 17.065s

6 par: 14.997s

7 par: 13.723s

8 par: 13.151s

20 par: 14.304s

200 par: 14.096s

600 par: 14.409s

800 par: 15.431s

1000 par: 18.636s

1200 par: 20.38s

2400 par: 25.881s

3000 par: 31.163s

Q5.

(run on cluster)

standard CPython implementation: 20.004s

Spark Python with PyPy 21.405s

No spark with single-threaded pypy: 2.148s

No spark with c: 0.009s

In order to get results, we set sample input to a small number 10 using CPython command.

For speedup:

Compare standard CPython implementation and Spark Python with PyPy.

Spark Python with Pypy and No Spark with single-threaded Pypy -> Comparison is not fair as one is using single thread/core and the other is using multiple to speed up the computation. To check the overhead it would be better to make the computation very small. Like if we run spark-submit euler.py 100 (here n=100 ensures that computation is negligible and the execution time can be taken as Spark overhead)

~~Overhead added by spark can be the difference between running time of Spark Python with Pypy and No Spark with single-threaded Pypy. Here it can be 19.257s (21.405 – 2.148). The reason may be the loading time of Spark when we start it up to run our program.~~

~~Pypy speeds up over the usual Python implementation by about 4s. (pure Python implementation with a same sample input of 10 runs 25.653s on cluster)~~