Dataset Description & Problem Implemented

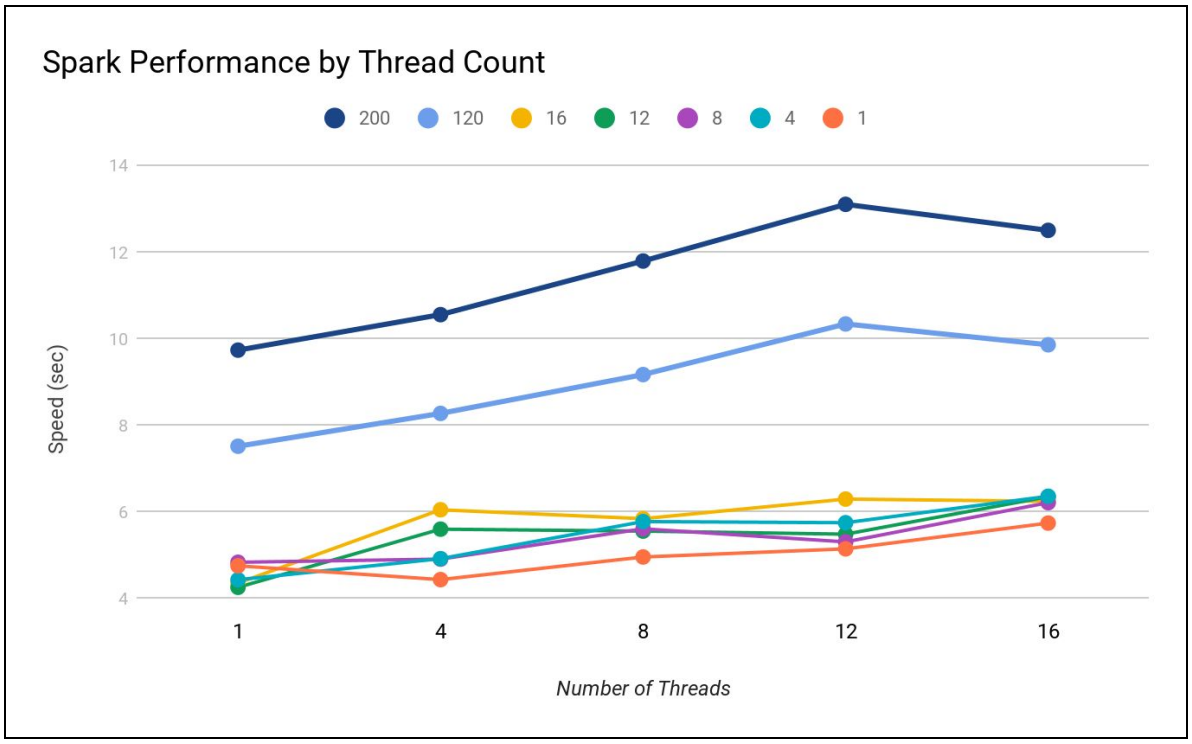
We chose to mine climate data in JSON using Python (PySpark specifically).

The original .dat file containing GHCN climate data from NOAA was 163 MB, and using a python script (DatToJson.py), we turned this into JSON data for easier processing at a size of ~95 MB (TemperatureData.json).

country	year	Avg
MK	1900	12.95
EI	1900	9.29
LH	1900	6.11
JE	1900	10.69
BR	1900	21.13
NL	1900	9.24
TD	1900	27.24
AG	1900	16.54
MX	1900	17.83
AU	1900	6.71
RQ	1900	25.53
ZI	1900	19.2
WS	1900	25.96
TX	1900	14.4
CM	1900	24.24
SW	1900	2.82
FI	1900	1.2
RS	1900	0.83
TW	1900	22.51
EG	1900	21.22

The data was presented as a list of countries (countries.txt) and measuring stations with measurements for each year and month. We decided to produce a single average yearly temperature for each country, averaging monthly data and averaging data from each climate monitoring station (main code in DatSpark.py).

We measured the **speed of processing** against the **number of threads**. The results are presented in the following graph.



The legend indicates the *partition count*; because the default value for partition count is 200 and we found in research that setting the partition count to the number of cores you have available helps improve processing speed, we wanted to play with this number as well and see how it affected speed performance in conjunction with changing the number of threads.

Analysis

Clearly, reducing the partition count to be closer to the number of cores being used made a significant impact on speed performance, with the average difference of 16 vs. 120 and 16 vs. 200 partitions being 3.27s and 5.784s, respectively.

With those outliers addressed, we focus on comparing data points with partition counts closer to the number of threads. In all cases except those where the partition is 1, the single threaded process performed at the fastest speed.

In the higher partition counts (12 & 16), the performance of 4 threads did better than that of 8 threads, potentially due to overhead at that processing level.

In the mid-range partition counts (4, 8 & 12), 12 threads outperformed 8 threads and in one case (with partition count of 12) even outperformed 4 threads.

Generally, the performance of 16 threads did the worst of all thread counts, even when the partition count was set equal to 16. However, with the much larger partition counts, we see better performance of 16 threads than 12 threads; it still does not outperform the lower thread counts though.

There does not seem to be any correlation between the number of threads and partition count that affects processing speed.

Conclusion

It appears that our program did not benefit from parallel computing. We assume this is because the dataset we are operating on (~95 MB) is not big enough, and the overhead from partitioning the data exceeds the benefit from running concurrently on multiple cores.

Challenges

Initially we installed Apache Spark 1.4.0 simply because it was the version used in the Github documentation provided; this was fine until trying to run the main DatSpark.py program that imported SparkSession and expr; updating to 2.4.5 was quick and easy, and solved the issue. Something interesting to note on this was that many of the configuration steps that needed to be done for 1.4.0 per the Github resource seemed to be automatically taken care of on the other nodes. This would be useful to note for future classes, as it would save a significant amount of set-up time and potentially mitigate problems caused by mistakes in configuration.

When first processing the JSON data, we ran out of java heap space, and assumed this was because of the file size. Ultimately, however, it was because of how PySpark reads JSON data; PySpark was reading the original "multiline" JSON format (standard) as one object and was not able to partition. To solve, we broke up the JSON so that there was one object per line.

During the first attempt at running the program on the cluster, we encountered a TimeException error, but it was quickly fixed by configuring the spark.network.timeout to be 600s in the command line when running spark-submit.

Even though it was a non-essential task, we tried to write our dataframe output to a file of some sort to make the results more manageable, and keeping in mind that it will be useful for the extension of this project for the paper. This was more difficult than expected; due to conversion of dataframe/file types we kept encountering various errors, but intend to solve this for the extension.

Next Steps

In our paper as an extension of this project, we intend to find the average yearly *change* in temperature for each country to determine which countries are warming or cooling and at which rate over the last century. In doing that, we will sanitize the data for bad inputs and represent data in fahrenheit. We also want to look into improving the performance and speed of our application, document and present which changes in the data structure and program make improvements. Additionally, we would like to try using python graphing libraries to transform our data findings into visualizations and measuring multiple performance aspects (e.g. speed, energy, etc.).