**TEST FILE USED PRIOR TO WAR&PEACE…**

**Test file to start with:**

aaAaa b

op

**\*\* This is what was used to build the program prior to running the larger file; projectRun.py used WarAndPeace**

**FINDING THE AVERAGES:**

TOTAL NUMBER OF CHARACTERS: 3227581 (used to solve for pc)

RESULTS:

112013085818 nanoseconds

112013085.818 microseconds (μs)

112.013085818 seconds

112.01 seconds (rounded)

INFORMATION IN DATA STREAM: 743699.526 (SINGLES)

INFORMATION IN DATA STREAM: 160299.742 (PAIRS)

INFORMATION IN DATA STREAM: 52336.552 (TRIPLES)

\*\*\* Now going to grab the average time for 0 threads

\*\*\*\*\* 0 THREADS

RUN 1: 112.013085818 seconds

RUN 2: 119.011069424 seconds

RUN 3: 110.80892328 seconds

AVERAGE = 341.8330785 / 3 = 113.9443595 seconds

Results: (for a Pool of 2…)

70348807890 nanoseconds

70348807.89 microseconds (μs)

70.34880789 seconds **BIG JUMP!!**

70.35 seconds (rounded)

INFORMATION IN DATA STREAM: 743699.526 (SINGLES)

INFORMATION IN DATA STREAM: 160299.742 (PAIRS)

INFORMATION IN DATA STREAM: 52336.552 (TRIPLES)

**\*\*\*AVERAGES\*\*\***

\*\*\*\*\* 2 THREADS

RUN 1: 70.34880789 seconds

RUN 2: 68.556980977 seconds

RUN 3: 68.56214363 seconds

AVERAGE = 207.4679326 / 3 = 69.15597754 seconds

\*\*\*\*\* 4 THREADS

RUN 1: 48.660048846 seconds

RUN 2: 47.938612678 seconds

RUN 3: 47.875360353 seconds

AVERAGE = 144.4740219 / 3 = 48.15800729 seconds

\*\*\*\*\* 8 THREADS

RUN 1: 47.571287324 seconds

RUN 2: 47.750695475 seconds

RUN 3: 47.074456033 seconds

AVERAGE = 142.3964388/ 3 = 47.46547961 seconds

\*\*\*\*\* 16 THREADS

RUN 1: 47.45276913 seconds

RUN 2: 47.535462656 seconds

RUN 3: 47.589933512 seconds

AVERAGE = 142.578165298/ 3 = 47.526055099333333333333333333333 seconds

\*\*\*\*\* 32 THREADS

RUN 1: 48.099712915 seconds

RUN 2: 48.104893957 seconds

RUN 3: 48.147236941 seconds

AVERAGE = 144.351843813/ 3 = 48.117281271 seconds

\*\*\*\*\* 60 THREADS (64 didn’t work…)

RUN 1: 50.140585623 seconds

RUN 2: 50.53693208 seconds

RUN 3: 50.303801896 seconds

AVERAGE = 150.981319599/ 3 = 50.327106533 second

**GRAPH:**

Graph will show EXECUTION TIME in seconds, as a FUNCTION OF THE NUMBER OF THREADS…

SUMMARY:

**The min (fastest result)** was with 8 and 16 threads, with the **absolute fastest being 8 threads**.

**The biggest improvement** was seen **between 0 and 2 threads**, **improving by a factor of ~ 1.6**

The second biggest improvement was seen between 2 and 4 threads, improving by a factor of ~ 1.4

Prior to the results, the expectation was that there would be ***consistent improvement*** due to having more threads to help with the load. As the threads increased… this was proven to be wrong, since after 8 threads, the **execution time only increased.**

Taking a look at the power point slides from class, the guess as to why comes from **extremes:** at one end there were not enough threads, and at the other end there were too many. The **benefit of having multiple threads to handle the problem started to be outmatched by the cost of having to create & use multiple threads.**

Threads for concurrency (x values) were chosen by the pattern 2n were n was in range 1 through 6. Creating a graph for n we see:

There seems to be a limit as the execution time approaches 47. It does not go below this point. I.e.: **Lim f(n) as n -> 3= 47**

The graph resembles the shape of : **f(x) = (-lnx)**

**Troubles in the process:**

There was an issue with trying to use 64 threads. This would not run (not sure as to why). 60 threads were used as the max instead.

**CLOSING REMARKS:**

If you are curious as to see the process (transforming from imperative to functional), see CodeDocumentation.docx

If you want to see how files transformed, all files can be found in characterProbability.zip

See projectRun.py for code source, or finalCodeVersion.docx

All code was written with PyCharm community addition

That is all.