**Introduction**

Parallelism is a new concept to most CSC2002S students, and we are only just started learning the brief of parallelism. The fork-join model is a parallel design pattern that we formulated as early as 1963. Fork join can be expressed in the following java

pseudocode:Compute (problem) {

If (the problem is small enough) {

Solve the problem using sequential method

} else {

Split the task into 2 subtasks, fork, join and then compose to return a combined result of the 2 subtasks

}

}

**Aim:** This report aims to compare the sequential method with the fork-join method and generate a statistical report that shows the time that each program is needed for a problem. I will be using data smoothing as the problem to compare the time needed for each method to solve the problem. The two programs will have to solve different sample sizes for data smoothing, with them being 100, 1 000, 10 000, 100 000, 1 000 000, 10 000 000 samples. I will be recording the time taking for each sample size at their 15th to 20th tries to compose the most accurate time required for each method to solve the problem. The sequential method will be represented by the sequential.java program. The fork join method will be represented by the ForkJoin.java MedianCalculation.java programs.

**Parallel algorithm:** Parallelism is the process of several sets of instructions simultaneously and combines the result of each individual set to obtain a final result. There are 4 models of computation: SISD, SIMD, MISD, MIMD. The most used model is MIMD with multiple instructions and multiple data. The number of instructions that can be run simultaneously is the number of cores that a computer has. Unfortunately, parallel algorithms are only compatible with computers with multiple cores, but the majority of modern computers are multi-cores.

**Expected speedup:** Ideally, this type of algorithm should reduce the time programs needed to solve a real-life problem that deals with a large amount of data. In my opinion, the speedup of the programs between sequential and parallel can vary. With smaller programs, the parallel algorithm should be slower because threading can be very costly. With a large program, a parallel algorithm should be faster because we’re using multiple cores to solve a problem. I think in the ideal situation if a computer only has 8 cores. The parallel programs can only be 8x faster than the sequential programs. The actually expected speedup will be tested in the programs and I will record and report the result of each sample size.

**Method**

I will be discussing all the functions that I have created in both sequential and parallel programs. The overall design of the main method in both codes are:

public static void main (String[] args) {

readFile

dataSmoothing

writeFile

}

Explanation of the codes:

**Both** sequential and parallel programs have the same tick, tock readFile and writeFile function.

readFile function is used to read each line in the input text file using the scanner and add it to the output array. The first line of the input file will be the size of the file and the following lines will be the actual numbers. To store the numbers into the float array, I will have to replace all the ”,” with “.”.

writeFile function has an if statement to check if the output file exists or not. If it doesn’t exist, create a new file with the output file name and write in that file. If it does exist, I will clear all the content inside the output file and write in that file.

The tick function is to start time recording and the tock is to end the time recording.

The sequential program has a data smooth function to smooth data in a sequential method. The program will receive an array that represents all the numbers in the input text file and the filter size. I will create a for loop that loops through all the numbers in the input text file. Inside the for loop, there is an if statement that finds whether the number is at the border or not. If it is at the border, then the output number after data smoothing will be the same as the input number. If it is not in the border, then I will find the surrounding numbers depending on the filter size and add them to the temporary array. Sort the temporary array and find the midpoint of the temporary array and add it to the output array. The data smooth function will then output the output array that is afloat.

Parallel programs have a function called fjDataSmooth that returns an output array after invoking the folk join pool with the MedianCalculation object. In the MedianCalculation.java class, there is a constructor, compute method and a sequential dataSmooth method. The constructor will receive the values and store them in the local variables. The dataSmooth method is the same as the sequential dataSmooth method. This is used in compute when the sequential cutoff is larger than (high – low). Otherwise, I will be using the fork-join methods, by first splitting the array into left and right subarray, use the fork-join method to calculate the subarrays simultaneously and return with an output array with all the output of the subarrays combined.

**Validation**: I have worked out the output subarray for the first 10 positions, random 10 consecutive positions in the array and the last 10 numbers in the array for 3 text files. The output I have calculated matches with the output of the program. I also have compared the output with other peoples output and the result turn out to be the same. So statistically, it is very unlikely that two programs that are coded differently have the same output result and have bugs in the program. So I am assuming that my programs are correct.

**Timing:** I use the tick and tock method that is explained above in the explanation of the sections of the codes to calculate the time required for programs to calculate inputs. I only calculate the dataSmooth method excluding the time used to read and write to the file. For the most accurate time to compare the data smoothing, I have run the code 20 times using a for loop, then I will report the time taken for the 21st to 25th including the 25th time taken for computing for both programs. I will then average the 5 recorded times and generate an average that represents the time taken for the program to process the problem. This time does not represent the actual time taken but rather the estimated time is taken in the best scenario.

**Speedup:** To measure the speed up, I will run the program 5 times and record the 1st, 5th, 10th time 15th and 21st to 25th. This theoretically will provide the most accurate speed up of the program. The recorded times will be noted and calculate the average time taken for each recorded time. With this average, I will then provide statistical analysis to further understand the speed up and the nature of parallelism with sequential algorithms.

**Machine architecture:** My computer is a Dell inspiron 5379 with the following specs:

Device name DESKTOP-H3JQ9NN

Processor Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz 2.00 GHz

Installed RAM 8,00 GB (7,84 GB usable)

System type 64-bit operating system, x64-based processor

Pen and touch Pen and touch support with 10 touch points

There are 8 Logical processors in the CPU

The current operating system information are the following:

Edition Windows 10 Home Single Language

Version 20H2

Installed on ‎2021/‎04/‎11

OS build 19042.1165

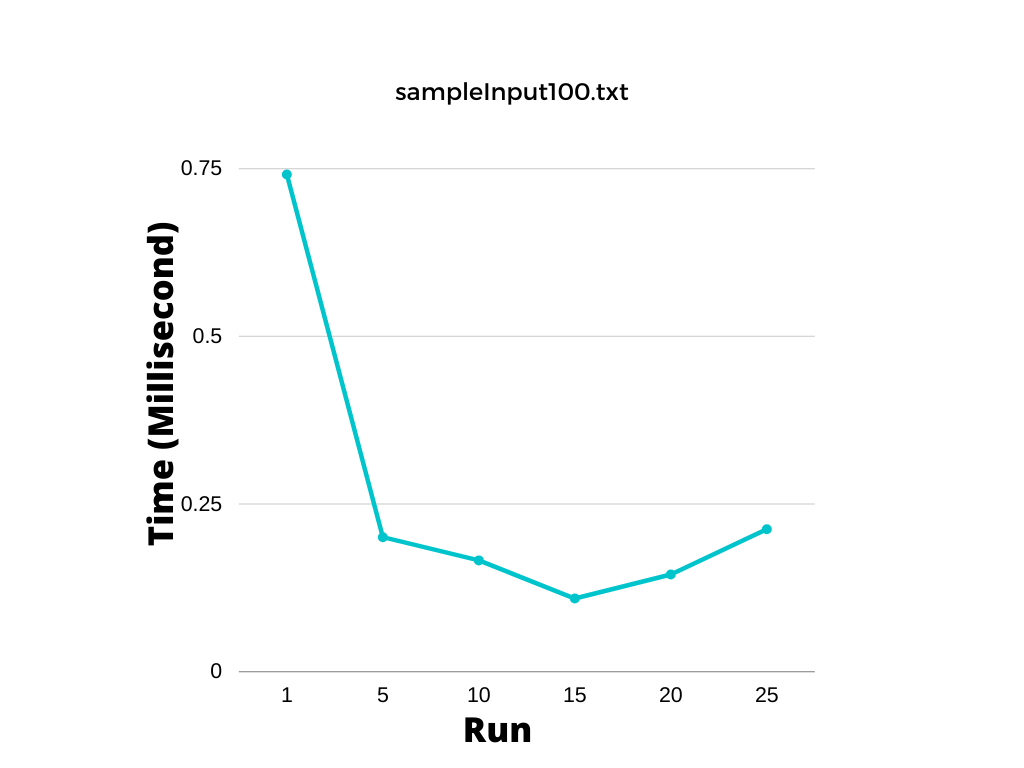
Experience Windows Feature Experience Pack 120.2212.3530.0

**Interesting problem and problems:** Data Smoothing is a very interesting problem to research, especially today with all the COVID-19 cases happening around the world. We could very much use parallelism to data smooth the daily COVID cases to generate a trend so that we can analyse the situation better. The biggest problem I am encountering is to code in a parallel way because I am still relatively new to this concept and still not 100% understanding the coding style of parallelism. This is making my assignment a lot hard as I have used a lot of sources to help me understand the concept before I can start coding. Coding is also very hard because the errors I am getting are very new to me and I need further studies to understand the meanings of the errors.

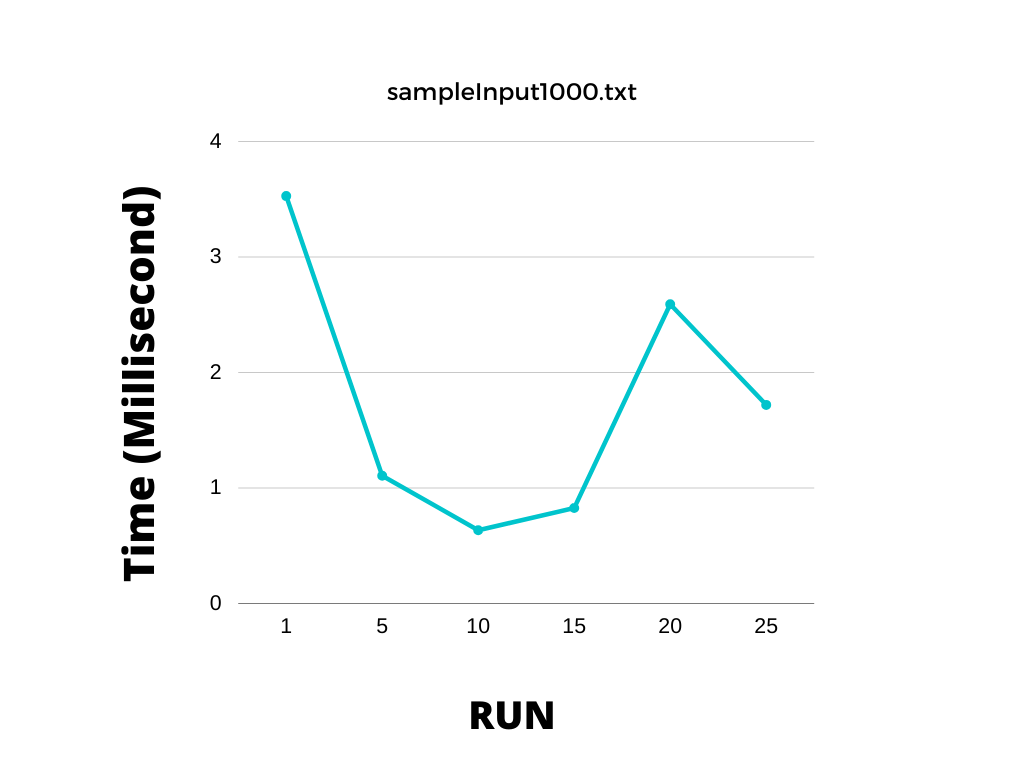
**Results:**

***­­Sequential:***

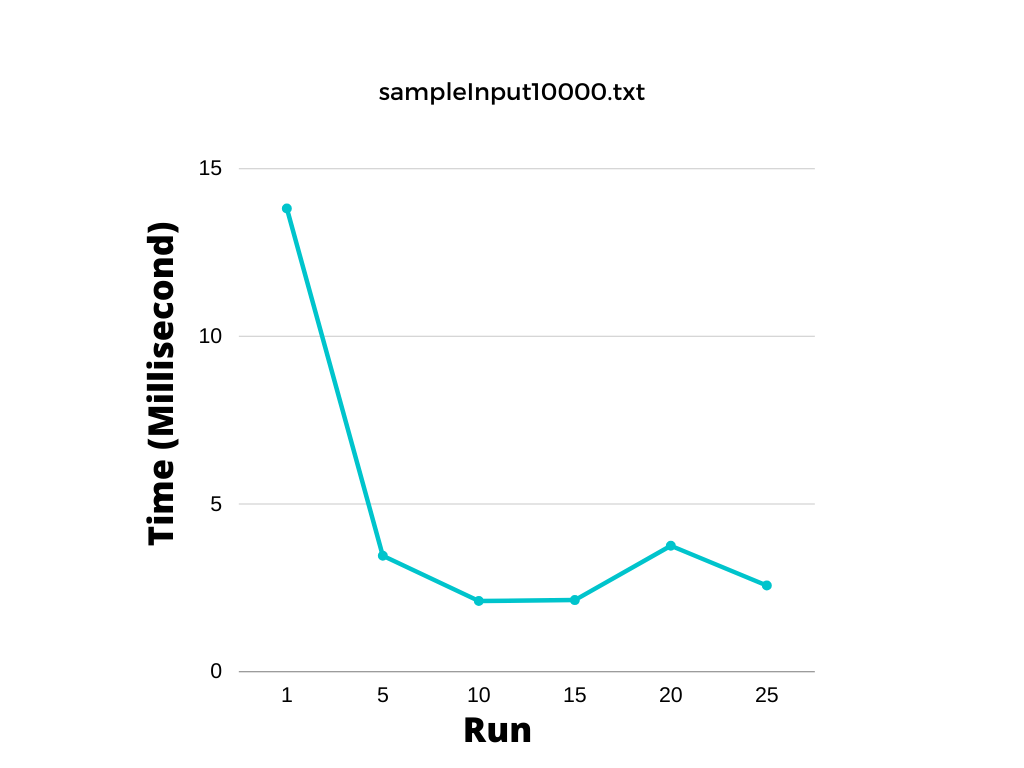
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| sampleInput100.txt filter size 7 | | | | | | |
| times | 1st run | 2nd run | 3rd run | 4th run | 5th run | average |
| 1 | 1.1074 | 0.4558 | 1.0194 | 0.5662 | 0.5580 | 0.7414 |
| 5 | 0.2722 | 0.1623 | 0.1678 | 0.1356 | 0.2648 | 0.2005 |
| 10 | 0.0937 | 0.0931 | 0.0823 | 0.4112 | 0.1491 | 0.1659 |
| 15 | 0.1639 | 0.0861 | 0.0905 | 0.0952 | 0.1094 | 0.1090 |
| 20 | 0.0733 | 0.0692 | 0.0708 | 0.1096 | 0.4014 | 0.14486 |
| 25 | 0.4567 | 0.2078 | 0.2011 | 0.153 | 0.1804 | 0.2123 |



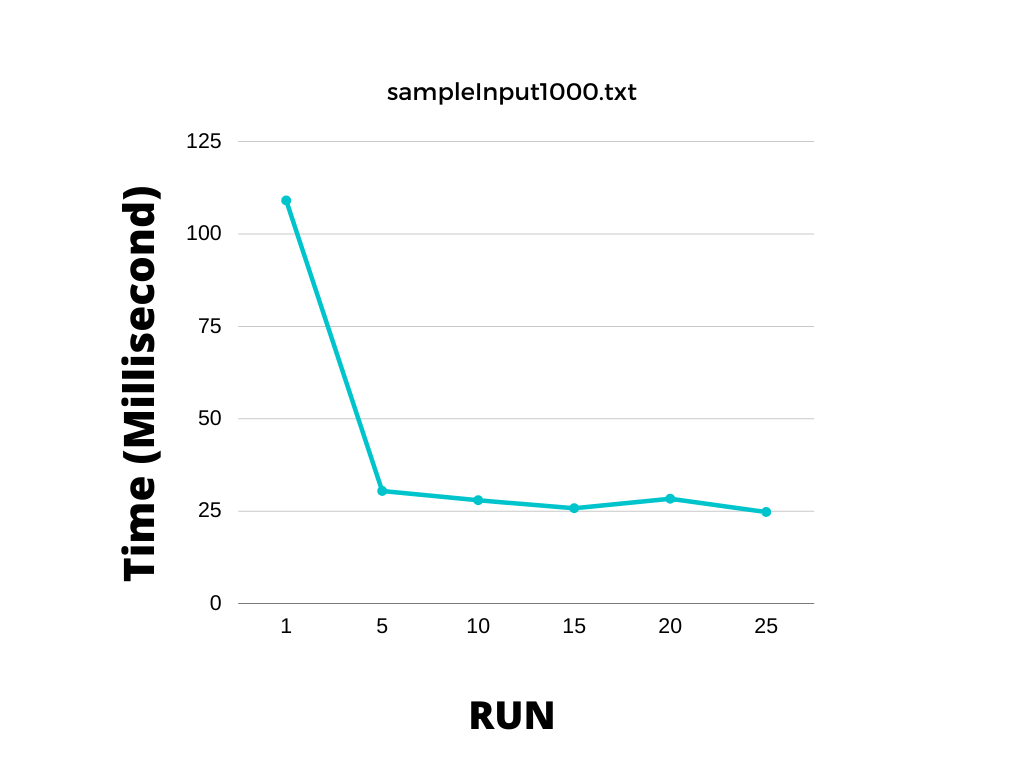
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| sampleInput1000.txt filter size 7 | | | | | | |
| times | 1st run | 2nd run | 3rd run | 4th run | 5th run | average |
| 1 | 3.0526 | 4.1928 | 3.1998 | 4.2474 | 2.9491 | 3.5283 |
| 5 | 2.6697 | 0.7865 | 0.7159 | 0.6961 | 0.6679 | 1.1072 |
| 10 | 0.6583 | 0.5655 | 0.6347 | 0.6986 | 0.6207 | 0.6356 |
| 15 | 1.9314 | 0.5603 | 0.5492 | 0.555 | 2.4050 | 0.8272 |
| 20 | 2.0575 | 3.5049 | 1.1647 | 1.3232 | 4.9082 | 2.5917 |
| 25 | 3.1874 | 1.3753 | 1.3958 | 1.6494 | 0.9946 | 1.7205 |



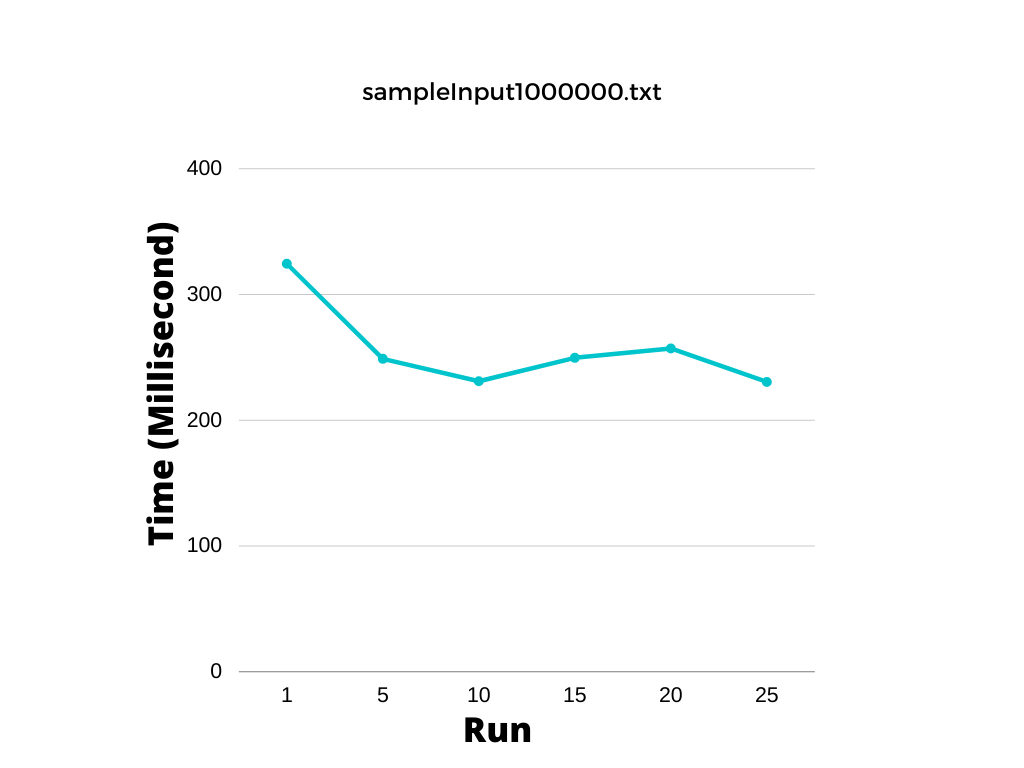
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| sampleInput10000.txt filter size 7 | | | | | | |
| times | 1st run | 2nd run | 3rd run | 4th run | 5th run | average |
| 1 | 11.5295 | 14.5526 | 12.3034 | 18.2075 | 12.4700 | 13.8126 |
| 5 | 4.4554 | 3.1613 | 3.0867 | 3.6251 | 2.9417 | 3.4540 |
| 10 | 1.8159 | 2.4853 | 2.2247 | 1.8671 | 2.1291 | 2.1044 |
| 15 | 2.0561 | 2.1043 | 2.2689 | 1.9916 | 2.2454 | 2.1333 |
| 20 | 2.3394 | 9.6814 | 2.1966 | 2.3575 | 2.1941 | 3.7538 |
| 25 | 2.5255 | 3.1623 | 1.9871 | 3.3035 | 1.8625 | 2.5682 |



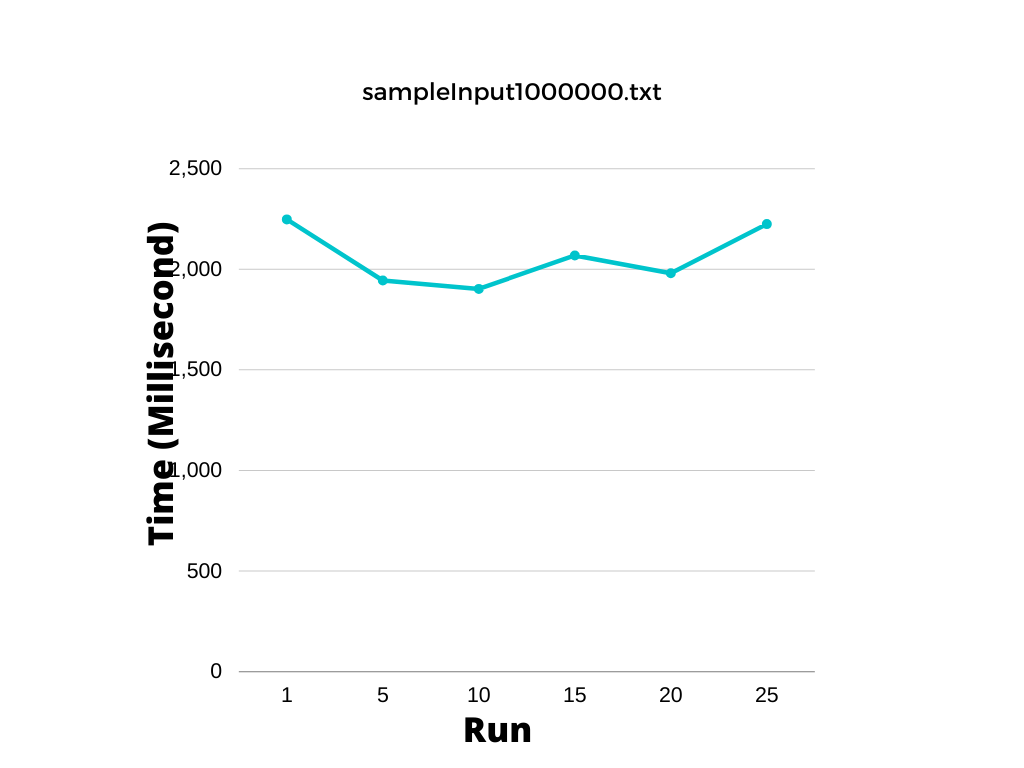
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| sampleInput100000.txt filter size 7 | | | | | | |
| times | 1st run | 2nd run | 3rd run | 4th run | 5th run | average |
| 1 | 174.1482 | 142.6782 | 84.3048 | 84.6624 | 59.4596 | 109.0506 |
| 5 | 19.2912 | 69.9182 | 20.8717 | 25.3313 | 16.9724 | 30.4770 |
| 10 | 20.3259 | 19.5674 | 19.7184 | 18.2721 | 61.8489 | 27.9465 |
| 15 | 19.7618 | 20.8603 | 44.6726 | 23.0652 | 20.5808 | 25.7881 |
| 20 | 22.3686 | 24.0998 | 49.5156 | 21.7822 | 23.9855 | 28.3503 |
| 25 | 21.0545 | 37.8597 | 19.5664 | 23.1639 | 22.2085 | 24.7704 |



|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| sampleInput1000000.txt filter size 7 | | | | | | |
| times | 1st run | 2nd run | 3rd run | 4th run | 5th run | average |
| 1 | 274.5307 | 413.6702 | 276.1097 | 293.7264 | 363.7801 | 324.3634 |
| 5 | 177.4878 | 286.8003 | 214.1314 | 309.8877 | 255.3729 | 248.7360 |
| 10 | 173.7387 | 278.3552 | 201.3174 | 294.5569 | 206.5220 | 230.8980 |
| 15 | 241.6548 | 323.4347 | 200.3747 | 289.4549 | 192.7891 | 249.5416 |
| 20 | 252.9872 | 265.9088 | 213.2775 | 254.9264 | 297.8470 | 256.9894 |
| 25 | 223.7819 | 258.9375 | 215.4724 | 183.2447 | 270.4318 | 230.3737 |



|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| sampleInput10000000.txt filter size 7 | | | | | | |
| times | 1st run | 2nd run | 3rd run | 4th run | 5th run | Average |
| 1 | 2687.7470 | 1932.5208 | 2302.1304 | 2208.2160 | 2108.5874 | 2247.8403 |
| 5 | 2126.3699 | 1846.0446 | 2052.6646 | 1690.3236 | 2005.4578 | 1944.1721 |
| 10 | 2012.0107 | 1897.4105 | 1731.9047 | 1876.5608 | 1994.0396 | 1902.3853 |
| 15 | 2080.7476 | 1774.7831 | 2015.2861 | 2268.7214 | 2204.1494 | 2068.7375 |
| 20 | 2283.4912 | 1717.0476 | 1812.6257 | 2044.1785 | 2042.1611 | 1979.9008 |
| 25 | 2059.6750 | 2393.4392 | 2316.5989 | 1889.6475 | 2462.505 | 2224.3731 |



**Data analysis:**

I have the sequential.java program on 4 different sized text files with them being 100, 10 000, 1 000 000, 10 000 000 elements in each text file. In each of the 4 different runs, we can see that there is a significant decrease in the average observed time taken from as the number of run increases. There is a problem that I have observed through the test runs, the average time taken for each run increases after the 10th to 15th run. This is something that is out of my expectations, the reason for this is still unknown and still need further research to understand this phenomenon. One of my predictions for this phenomenon is outliers, which could very possibly affect the result of the average.

With the average of each time, I run the program, I have created a line graph to show the trend of the time taken for the data smoothing process. As we can see, the trend, in general, is downward sloping indicating that as more runs are executed, on average, less time is required to generate an output. With each runs, the most significant decrease in the time taken is in the first 5 runs, because the slopes are much steeper.

***Parallel:***

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| sampleInput100.txt filter size = 7 | | | | |
| times | Sequential Cutoff 10 | Sequential Cutoff 20 | Sequential Cutoff 50 | Sequential Cutoff 100 |
| 1 | 4.9369 | 4.7111 | 4.4220 | 4.6711 |
| 5 | 0.4510 | 0.4279 | 1.3123 | 0.9923 |
| 10 | 0.9420 | 0.5885 | 0.6291 | 0.6653 |
| 15 | 0.6562 | 0.5758 | 0.6564 | 1.0111 |
| 20 | 0.4909 | 0.3689 | 0.1575 | 0.4785 |
| 25 | 0.5088 | 0.4228 | 0.4607 | 0.3567 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| sampleInput1000.txt filter size = 7 | | | | |
| times | Sequential Cutoff 50 | Sequential Cutoff 100 | Sequential Cutoff 200 | Sequential Cutoff 500 |
| 1 | 14.4995 | 11.1752 | 10.8865 | 8.9662 |
| 5 | 10.0294 | 5.8130 | 2.9428 | 1.5240 |
| 10 | 8.7492 | 5.1577 | 3.1082 | 1.1094 |
| 15 | 4.5937 | 3.9525 | 2.8113 | 2.7510 |
| 20 | 4.2240 | 2.7327 | 3.9278 | 2.0001 |
| 25 | 2.0071 | 3.5581 | 2.2176 | 1.9315 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| sampleInput10000.txt filter size = 7 | | | | |
| times | Sequential Cutoff 500 | Sequential Cutoff 1000 | Sequential Cutoff 2000 | Sequential Cutoff 5000 |
| 1 | 112.6104 | 60.6512 | 36.7662 | 67.3170 |
| 5 | 4.4193 | 14.2410 | 6.8660 | 5.1472 |
| 10 | 4.4625 | 3.2962 | 3.4438 | 4.1144 |
| 15 | 3.2416 | 2.2638 | 1.4268 | 3.3251 |
| 20 | 3.9416 | 2.4936 | 1.9112 | 2.3009 |
| 25 | 2.4455 | 2.8907 | 1.2701 | 1.4449 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| sampleInput100000.txt filter size = 7 | | | | |
| times | Sequential Cutoff 5000 | Sequential Cutoff 10000 | Sequential Cutoff 20000 | Sequential Cutoff 50000 |
| 1 | 168.2134 | 172.9096 | 140.6765 | 124.3699 |
| 5 | 60.7977 | 21.1820 | 19.9345 | 27.2039 |
| 10 | 38.6248 | 33.1890 | 20.5587 | 11.1330 |
| 15 | 36.3201 | 21.5776 | 18.8241 | 8.9227 |
| 20 | 38.3977 | 25.2858 | 14.8288 | 7.3272 |
| 25 | 40.2965 | 27.8099 | 14.0080 | 7.8833 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| sampleInput1000000.txt filter size = 7 | | | | |
| times | Sequential Cutoff 50000 | Sequential Cutoff 100000 | Sequential Cutoff 200000 | Sequential Cutoff 500000 |
| 1 | 700.4124 | 433.5927 | 392.7662 | 300.6199 |
| 5 | 441.4858 | 273.6655 | 176.5276 | 100.1500 |
| 10 | 555.2097 | 281.2962 | 137.1830 | 87.7927 |
| 15 | 423.1468 | 229.0145 | 124.0126 | 92.5187 |
| 20 | 456.3014 | 190.8815 | 140.2837 | 99.3189 |
| 25 | 517.3791 | 210.1661 | 130.9201 | 80.0162 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| sampleInput10000000.txt filter size = 7 | | | | |
| times | Sequential Cutoff 500000 | Sequential Cutoff 1000000 | Sequential Cutoff 2000000 | Sequential Cutoff 5000000 |
| 1 | 6628.3115 | 7218.3343 | 3293.8537 | 2107.1730 |
| 5 | 7912.4334 | 6814.7069 | 1774.2640 | 938.6947 |
| 10 | 6270.6017 | 5510.1445 | 1893.5893 | 960.3525 |
| 15 | 7110.4043 | 555.3300 | 2257.1634 | 1083.8702 |
| 20 | 5902.8092 | 6712.6072 | 2011.7366 | 1108.9218 |
| 25 | 6690.2392 | 6662.7220 | 2018.7272 | 1020.6584 |

I have conducted testing results for sample input sizes of 100, 10 000, 1 000 000 and 10 000 000. The time used to compile the sample inputs is reduces as more runs are introduced. The problem that I have observed is that for parallel codes, the sample size 100 and 10 000, parallel are slower than the sequential. This suggests that parallelism is not great for a small sample size such as 100. This is most likely due to the nature of parallelism where it only comes effective as the sample size increases.

**Discussion**

**Is it worth using parallelization (multithreading) to tackle this problem in Java?**

This depends on the sample size, for a small sample below 100000, sequential would be much more effective than parallel coding mainly because sequential do not have to create threads, unlike parallel. As we can see from the diagram, the average of sequential is much more than parallel codes. For sample size over 100000, we can see that the parallel method is much more efficient than sequential codes, and as the size of the sample size increases, the range between parallel and sequential increases.

Although this also variation between the parallel and sequential also have to take the filter size into consideration. But for this experiment, I am only using the filter size of 7 only. My assumption for as the sample size increases, the sequential codes will have to take longer to compile relative to the parallel codes and the point where parallel methods are more efficient than sequential will decrease and the filter size increases.

**For what range of data set sizes and filter sizes does your parallel program perform well?**

I think the optimal sample size is from 10 000 and up because this is the point where the difference between the sequential and parallel are almost the same, indicating as the sample size increases, parallel will be much more efficient. We can see from the graph and table, the average of parallel is in general lower than sequential in every run for Sequential cut off of half of the sample size. Therefore, lower time indicates less time to solve the problem therefore more efficient

**What is the maximum speedup obtainable with your parallel approach? How close is this speedup to the ideal expected?**

As we can see from the parallel code for sample size 10 000 000, we can see a speedup of about 2 times, I have an 8-core laptop therefore the ideal or optimal speedup should be 8x. Due to the scalability, the typical success will decrease relative to the linear speedup. It is close, but not identical. There could be more efficient methods that could speed up the process even further, ideally the same as the linear speedup. If the sample size continue to increase, the speed up of the program will increase even more because as we can see from the table pattern, the speed up can still increase.

**What is an optimal sequential cutoff for this problem? (Note that the optimal sequential cutoff can vary based on dataset size.)**According to my data, parallel codes are only effective after 100 000, I have tested 4 times with 4 different sequential cutoffs. I will be comparing their optimal result, average time excluding 1st and 5th run because my programs need to be warmed up therefore I will skip the warming up process.

According to my program, the best sequential cut offs are as the following:

sampleInput100000.txt => Sequential cut off = 50000

sampleInput1000000.txt => Sequential cut off = 500000

sampleInput10000000.txt => Sequential cut off = 5000000

This sequential cut off does not represent the real sequential cut-off, there could be variations due to outliers, for example, more statistical testing are required to find the exact sequential cutoff, but according to the table. The sequential cut off above is considered correct.

**What is the optimal number of threads on your architecture?**The optimal threads are dependent on the computer and sample size, for my computer, the best number of threads should be 8 or 16 for my programs for programs that is much larger than the sample inputs that I am provided. The reason for this is my computer has 8 cores and each core can work on one thread. Creating a new thread is very expensive and I do not want too many threads because that could make my program slower. But for the current sample sizes, I think 2 threads will be better because 10 000 000 is not large enough to introduce more thread to the load.

**Conclusion:**

**Amdahl’s Law:** In these experiments, we can see the work of Amdahl’s law where the speedup increases at a decreasing rate as the number of processors increases. I have 8 core computers but I don’t get the theoretic speed up of 8x, rather I got a 4x speedup indicating that Amdahl is right about the increase in the number of processors don’t necessarily mean that the number of speed up would increase proportionally. This can also be seen in the sampleInput1000.txt, I made 8 threads for it but the performance is not as expected, it is slower than the sequential codes, increasing the number of cores do not necessarily mean we will get a proportional increase in the speed of the program

**Parallelism** is a very powerful tool to boost a computer’s performance, especially in this era of computers where each core cannot improve as fast as the computers decades ago, parallelism solves this problem by letting computers use multiple cores to improve efficiency. The use of multiple cores proved that Moore’s law is wrong, where multiple cores can improve the efficiency of a computer.

**Reliability:** Parallelism is much more complicated than sequential codes because of shared data and working with multiple cores. It would be highly reliable if a computer scientist knows how to handle the data properly without overwriting, duplication. Then we will get a program that is faster than sequential programs most of the time. For an inexperienced programmer, like me, I think it would be better to continue learning about parallelism and understand how data are operated, avoid programming parallel programs for companies until I am more knowledgeable about parallelism.

**Interesting fact:** The reason why CPU’s do get better as time moves on is because of the heating and power problems. The heating problems are obvious because when we play games, the computer often starts to warm up and the CPU should not reach a temperature of 75 degrees, so the OS often have to decrease the performance of the phone to reduce temperature. A modern gaming computer can use up to 500Watts of power, which is equivalent to 100 5W lights. The future of computers could be quantum computers, although researchers suggest that quantum computers won’t be available to the general public for the next few decades. Therefore we can only work with multiple cores and improving each core to improve the efficiency of the CPU