Program 2 Report

CSC 425- 01

Sean Taylor

Table of Contents

Cover Page …………………………………………………………………….…………. 1

Table of contents …………………………………………………………………………. 2

Introduction …………………………………………………………………...………….. 3

Literature review …………………………………………………………………………. 3

Description …………………………………………………………………………….…. 3

Hardware …………………………………………………………………………...… 3

Software …………………………………………………………………………….... 3

Screenshot of Program ………………………………………………………….……. 4

Multithreading Outcomes ……………………………………………………………. 4

What happened at 500,000? ………………………………………………….……… 5

Trying this on a different computer …………………………………………………. 8

Changing Thread Size ……………………………………………………….…….... 10

Summary ………………………………………………………………………………... 11

References ………………………………………………………………………………. 11

Introduction

This report is about how to maximize threads in a program and the advantages you can gain. By taking advantage of multithreading and your computer’s multiple processors you are able to make your program run faster than beforehand. In this paper I will show you the advantages of using multithreading for Merge and Quick Sort. I will also compare those results to Java 8’s latest sorting method, Java Parallel Sort.

Literature Review

For this project I did not use any online resources to assist me. Since most of the program was provided for us I did not need much outside reference on how to complete this program. The only outside resource I used was Professor Seyed and her Partition Method for Parallel Quick Sort.

Description

Hardware

|  |  |
| --- | --- |
| Category | Value |
| Computer make and model | Dell Inspiron 13-7352  Intel Core i5-5200U |
| Memory Size | 8 GB |
| Processor Speed | 2.20 GHz |
| L1 Cache Size | 256 KB |
| Number of Processors | 4 |
| Max number of threads possible | 4 |

Software

|  |  |
| --- | --- |
| JDK Version | 8.0.1110.14 |
| JRE Version | Jre1.8.0\_101 |

Screen Shot of Program

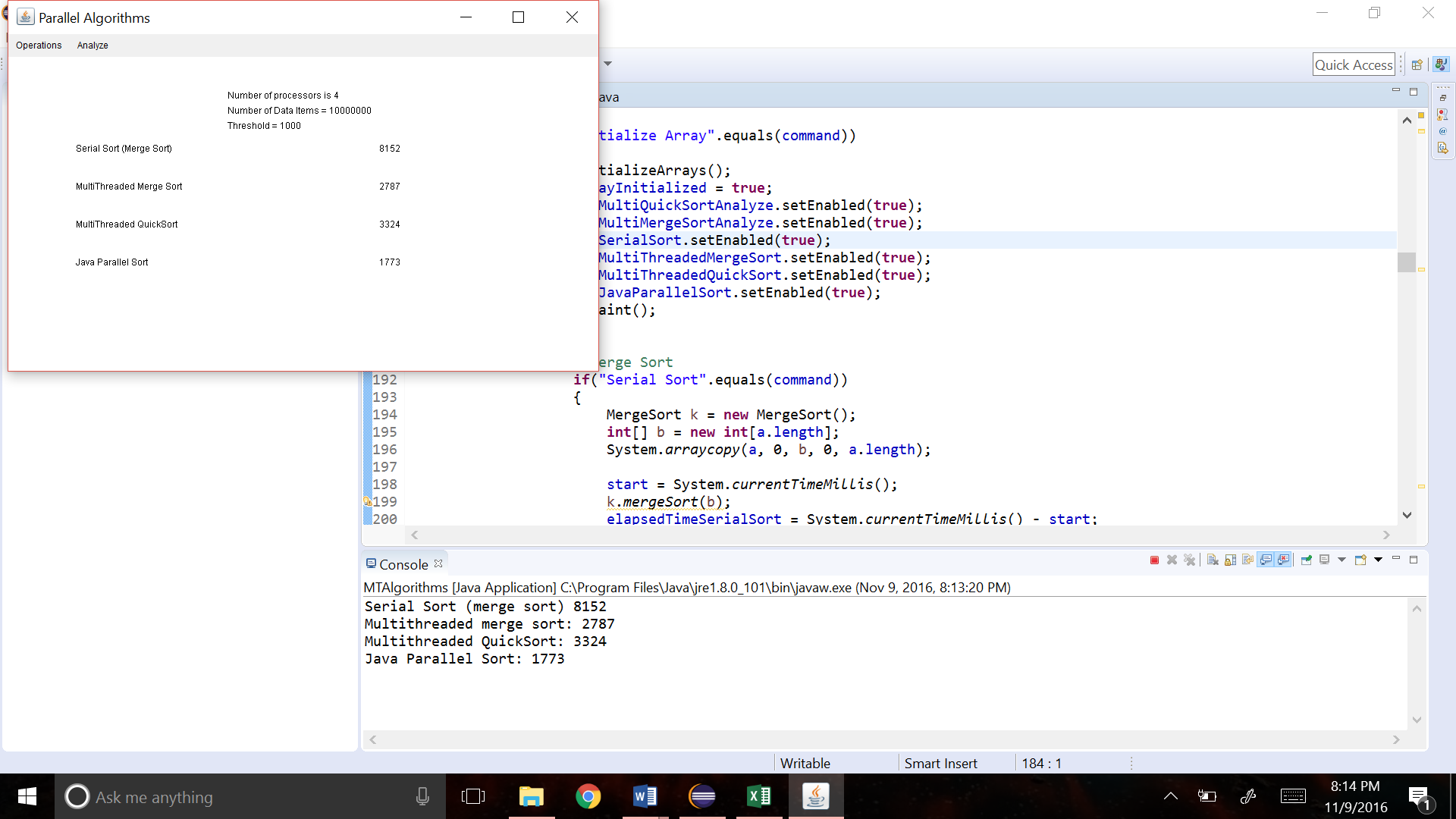


Figure 1

Multithreading Outcomes

Figure 2: Comparing the Averages of All Methods

As you can see in Figure 2, depending on the size of the thread, multithreading was the fastest way to run the program. Ignoring the spike around 500,000, which I will discuss later, Parallel Merge Sort and Parallel Quick Sort were almost ¼ the speed of Serial (Merge) Sort and faster than Java Parallel Sort. By implementing multithreading into a program, you can reduce the speed by a factor of 4, which also the number of processors and max threads in this computer. This can be incredibly useful if you have programs that would take a very long time. A program that used to take an hour to complete, can now be done in 15 minutes. Thus saving you valuable time waiting for the results.

What happened at 500,000?

Although not shown in Figure 2, there was a large spike for Parallel Merge sort when the threshold was 100,000. I ran this program multiple times, but there always was an extremely large spike in time when the threshold was 100,000, see Figure 3. I was also having trouble when the threshold was set to 10,000. There was always an extremely large spike in completion time, so much so that the program was never able to finish in a reasonable amount of time before I would have to force quite the program. When I ran the program without 100,000 the graph looks much more likely, see Figure 4.

As you can see in Figure 2, you will see that Parallel Quick Sort was having a similar issue with a threshold size of 500,000. I tested the program at first with a size of 100,000 but the program was not finishing so I had to force quit every time. I ran the program multiple times, but each time there was trouble with 500,000, see Figure 5. I tried running the threshold in different order, but it did not matter it always had an extremely high execution time. Figure 6 shows the time trend without 500,000 and it appears to be much more accurate.

The biggest trend I noticed with both methods of sorting was that if the threshold was set between 10,000 and 900,000 the execution time would be unreasonable. I have no explanation as to why this happened. The next section will talk about how these results vary on different computers.

Figure 3: Merge Sort

Figure 4: Without 100,000 – Merge Sort

Figure 5: Quick Sort

Figure 6: Without 500,000 – Quick Sort

Trying this on a different computer

Figures 7 and 8 describe the results when you run the program on a different computer. As you can see there is a spike in execution time when you set the threshold between 10,000 and 500,000 for Merge Sort and through 1,000,000 for Quick Sort. Comparing Figures 7 and 8 to 3 and 5 you see that it does not matter what computer this program is ran on. There was always a spike in execution time, increasing the process time, number of processors and number of threads made no difference on the outcome for these threshold sizes (See Figure 9). After trying the program on multiple computers with various different threshold size I was not able to find what was causing the jump in execution time. If I had more time I would look into this issue further.

Figure 7: Merge Sort (Parallel) ran on a different computer

Figure 8: Quick Sort (Parallel) ran on a different computer

|  |  |
| --- | --- |
| Category | Value |
| Computer make and model | Intel Core i7-4790S CPU |
| Memory Size | 16 GB |
| Processor Speed | 3.20 GHz |
| L1 Cache Size | 512 KB |
| Number of Processors | 8 |
| Max number of threads possible | 8 |

Figure 9: Hardware for different computer

Changing Thread Sizes

Figures 10 and 11 show the results of changing the maximum number of threads for each of the parallel sorts. For Merge Sort, you see that the optimum number of threads is between 4 and 6. For Quick Sort, the optimum number of threads is between 4 and 6, preferably 6. If you have too many or too few number of threads it will cause the program to not be able to run at maximum efficiency. By changing around the thread size you are able to find what will let the program run at maximum efficiency. This can become very useful when running extremely large programs.

Figure 10: Changing thread size for Merge Sort (Parallel)

Figure 11: Changing thread size for Quick Sort (parallel)

Summary

By taking advantages of a computer’s multiple processors you are able to greatly decrease the execution time of a program. Compared to Java’s parallel sort method, using parallel merge and quicksort can be much faster when using the right number of threads and thresholds. With multithreading you can cut your execution time by a factor of 4 or great, depending on how many processors your computer has. Thus saving you valuable execution time in the long run.

References

1. Professor Seyed