Parallel Merge Sort Report

Project 1

MSCS 679L_256_25S

Marist University
School of Computer Science and Mathematics

Submitted To:
Rasit Topaloglu

By: Arjun Suresh

1. DESCRIPTION OF TEAM MEMBERS

Arjun Suresh:

I'm Arjun, but I go by AJ. I'm a part of the five year masters program and am the Vice President of the Marist Esports Club which recently held a successful charity campaign for Sparrow’s Nest this past year. I am particularly interested in the area of artificial intelligence and how it can be used to enhance software development as well as cloud computing. I have experience in game development as well, and currently work with a group of indie developers to build a game.

2. INTRODUCTION

This project utilizes the merge sort algorithm, adding separate threads to create a parallel merge sort. I specifically picked out the algorithm because I had some interest in learning how the original algorithm itself functioned as I had lost some memory about the understanding of it. I decided it would be a good way to relearn while also learning how to introduce threads and make the process parallel as well.

Regular merge sort has an overall runtime of O(n log n) for all cases. It's a stable sort, meaning the order of equal elements is preserved in the sorted list.

While my own report on the observations I found with my implementation of parallel merge sort serves as a good way to understand the importance and drawbacks of parallelism, there is another paper that goes into further detail on this topic. Arne Maus [1] writes about his ParaMerge algorithm, discussing several innovations in his paper "A Faster, All Parallel Merge Sort Algorithm for Multicore Processors". The ParaMerge algorithm introduces different techniques such as dual ended merging with twin threads and a copy free insertion sort for small subarrays, together it aims to minimize bottlenecks and maximum parallel efficiency as dictated by Aamdahl's law. ParaMerge in turn achieves significant speedups on systems with many cores.

My own project, in contrast, focuses on clarity and ease of modification while operating under the constraints of the libraries. My approach focuses on utilizing recursive splitting with controlled thread depth, which spawns a new thread for the left subarray while processing the right subarray sequentially, while still utilizing a conventional merge strategy. The observations gathered from the project aim to provide an educational understanding as to the importance and use of threads in parallelism, as well as the benefits and drawbacks it can have depending on various input size and CPU power.

3. INSTRUCTIONS

In terms of merge sort itself, I do not use anything rather different. The code follows basic principles and examples already online. The merge sort will separate and divide the unsorted list into smaller sublists until each sublist contains only one element before merging them back together. If there are too many threads, out of depth, then the code would switch over to calling the regular merge sort over the parallel merge sort function.

The program itself requires three command line arguments, though in the code it checks for the program name as well. It needs the input file name, output file name, and thread depth.

The thread depth is an integer value representing how much parallelism the program utilizes. If the thread depth is 0 for example, then the program uses a single threaded regular merge sort. If there is a value of 1, it enables up to 2 threads. If there is a value of 2, it enables up to 4 threads, if there is a value of 3, it enables up to 8 threads, and so on. The maximum number of possible threads is given by (1 << thread_depth).

Before running, you might need to configure command line arguments. In Visual Studio, this is done by right clicking the project in Solution Explorer, selecting Properties → Configuration Properties → Debugging, and then entering the arguments in the Command Arguments field (for example: input.txt output.txt 2). The ‘2’ signifies 4 threads.

If you are running from the command line itself, you can execute the program with the four arguments it expects, the additional one being the name of the program, in this case, **ParallelMergeSort_Final.exe**. The rest of the arguments are formatted and listed below:

To run the program in command line, the four arguments it expects are this:

[Filename, input file name, output file name, thread depth]

[ParallelMergeSort_Final.exe, input_small.txt, output_small.txt, 3]
(3 means 8 threads)

4. MEMORY MANAGEMENT & ERROR CHECKING

The project features dynamic memory allocation as the entire file is initially read into a dynamically allocated buffer. After counting and filtering valid characters, a new array called data is allocated with an exact size equal to the number of valid characters from the file. This ends up minimizing memory waste.

In terms of error checking, there are a multitude of checks through the program. The program checks whether the input file successfully opens, as all file streams are checked to ensure they are opened successfully. It verifies if valid characters are present. If no valid characters are found, it prints an error message and exits. Additionally, the conversion of thread depth argument is handled carefully to ensure it is not a negative integer.

After copying the valid characters into the data array, the original buffer is deleted to free allocated memory. Once the processing is fully complete, the memory allocated for the data array is also freed. Similarly, in the merge function, temporary memory is allocated to hold merged segments and is deleted after use.

5. ANALYSIS

Gathering the performance data and tracking the thread execution time was more challenging than the merge sort itself. There are many resources online for merge sort, but for thread timing execution that was a lot more difficult.

I used an atomic timer, using a global timer, declaring it as a static member of a ThreadTimer struct. The counter is then incremented continuously in a dedicated timer thread. The use of the atomic variable ensures all the threads can safely read and update the timer concurrently without race conditions.

The timer thread continuously increments the global timer. This primarily happens before and after key operations like sorting a segment or merging two sorted halves. The code captures the current timer value using ThreadTimer::getTime(). It allows me to calculate the time taken by subtracting the start time from the end time.

For performance, I used two different computers to test a few different input files. For Table 1, it is tested using a PC with an AMD Ryzen 7 7800X3D 8-Core Processor, with 32GB of RAM. For Table 2, it is tested using a lenovo thinkpad with a i7-1355U 1.7Ghz Processor with 16 GB of RAM.

I primarily test with three different input files, input_small, input_medium, and input_large with different sizes of characters.

Table 1: Powerful PC

Number of Threads Tried	Input Size Tried	Total Units (Total Time)	Characters per unit time
1	**95**	**73695**	**0.0012891**
1	**11540**	**10885135**	**0.00106016**
1	**75362**	**68874533**	**0.00109419**
2	**95**	**81272**	**0.00116891**
2	**11540**	**12566694**	**0.0009183**
2	**75362**	**79426132**	**0.000948831**
4	**95**	**82859**	**0.00114653**
4	**11540**	**12488019**	**0.000924086**
4	**75362**	**82989849**	**0.000908087**
8	**95**	**84382**	**0.00112583**
8	**11540**	**12396772**	**0.000930887**
8	**75362**	**82515860**	**0.000913303**

Table 2: Lower Grade Laptop

Number of Threads Tried	Input Size Tried	Total Units (Total Time)	Characters per unit time
1	**95**	**297108**	**0.000319749**
1	**11540**	**76126235**	**0.00015159**
1	**75362**	**580502217**	**0.000129822**
2	**95**	**285017**	**0.000333313**
2	**11540**	**77249285**	**0.000149386**
2	**75362**	**516282351**	**0.000145971**
4	**95**	**323410**	**0.000293745**
4	**11540**	**70720129**	**0.000163178**
4	**75362**	**509322547**	**0.000147965**
8	**95**	**344220**	**0.000275986**
8	**11540**	**72589655**	**0.000158976**
8	**75362**	**414454739**	**0.000181834**

Observations:

For a tiny dataset such as 95 characters, multiple threads seem to introduce overhead that outweighs the parallel benefit. As a result, single threaded runs may finish faster or nearly as fast as multi-threaded ones because thread creation and synchronization cost more time than the data needs to be sorted.

For a medium and large dataset, increasing the thread count generally improves performance as more data can be processed in parallel. For moderately sized inputs, going from 1 to 2 or 4 threads tends to speed up the sorting decently. The powerful processor makes it more difficult to see clear improvements, however. Going from 4 to 8 threads can yield further improvements, but it depends on the input size as sometimes it can give you diminishing returns.

The lower grade laptop, with its weaker CPU, gives much clearer insights into the performance of parallelism. More threads appear to be more useful as the input size increases as the diminishing returns are a detriment to smaller input sizes. A clear example of this can be seen when viewing Table 2’s 75362 character input sizes. The total units decrease as the threads increase.

6. CITATIONS

[1] A. Maus, “A faster all parallel Mergesort algorithm for multicore processors,” *NIKT*, Aug. 2018.
[2] Adenzila. Answer to “How can I measure the execution time of one thread?”, *Stack Overflow*. 16 Aug., 2016. <https://stackoverflow.com/questions/38977879/how-can-i-measure-the-execution-time-of-one-thread>.