PARALLEL DIVIDE-AND-CONQUER SORTING

USING PTHREADS IN QUICK SORT ALGORITHM

AUTHOR: SANDILE NGWENYA
ORGANIZATION: UNIVERSITY OF ALBERTA
SUBMISSION DATE: 18 APRIL 2023

TABLE OF CONTENTS

EXECUTIVE SUMMARY

This project aimed to compare the performance of the serial and parallel implementations of the quick sort algorithm. The purpose of the study was to determine whether the parallel implementation would provide any performance benefits compared to the serial version. The methodology involved generating arrays of random integers of varying sizes, and then using both the serial and parallel versions of the quick sort algorithm to sort them. The performance of each implementation was measured in terms of execution time, finding the speed up and efficiency.

The key findings of the study showed that the parallel implementation of the quick sort algorithm outperformed the serial implementation in terms of execution time and efficiency, particularly for larger data sizes. The results demonstrated that the parallel implementation was not able to utilize multiple cores effectively, resulting in a significant reduction in the execution time. Additionally, the scalability of the parallel implementation was demonstrated for the large sets, as the efficiency did not decrease as the number of processes increased.

The limitations of the study included the fact that the number of cores used in the experiment might not have been enough to meet the requirements, which was shown by the repeated or significant change in the linear graph each time it got to two threads. Also, the use of random integers in generating the arrays meant that there were repeated integers, which impacted the expected output negatively. Additionally, the MacBook M1 chip (Silicon) used in this project did not support the OpenMP option that could have been used for the parallel quick sort version.

The recommendation based on the findings is that parallel implementations of the quick sort algorithm should be utilized for large data sizes as it provides better performance than the serial implementation. However, researchers should carefully consider the hardware requirements and implementation options for parallel programming to ensure optimal performance. Further research could explore the use of different parallel programming options and hardware configurations to optimize the performance of the parallel implementation of the quick sort algorithm.

INTRODUCTION

Quick sort is a popular and widely used sorting algorithm that has been extensively studied and implemented in various programming languages and frameworks. It was first introduced by Tony Hoare in 1959 and has since become one of the most commonly used algorithms for sorting large sets of data. Invented by the computer scientist Tony Hoare and published in July 1961, the algorithm's task is to put lists of things into the correct order: numbers in numerical order, words into alphabetical order, or dates into chronological order. (Freiberger). The algorithm is based on the divide-and-conquer strategy, which involves breaking down the problem into smaller sub-problems and solving each of them recursively.

The purpose of this report is to compare the performance of parallel computing quick sort and serial quick sort algorithms. Parallel computing is a form of computation in which multiple processors or cores work together to solve a single problem. It has become increasingly important in recent years, as data sets have grown larger and more complex, and the need for faster and more efficient computing has become more pressing.

Serial quick sort algorithms, on the other hand, are traditional implementations of quick sort that are designed to run on a single processor or core. While they are generally fast and efficient for small to medium-sized data sets, their performance can degrade significantly as the data sets become larger.

The scope of this report is to compare the performance of these two algorithms using a variety of test data sets and performance metrics. I will use both synthetic and real-world data sets to evaluate the performance of the algorithms under different conditions, such as data size, and system architecture: that is different number of threads.

Overall, the goal of this report is to provide a comprehensive analysis of the performance of parallel and serial quick sort algorithms. Focus is strongly going to be looking at the run time, speed up, and efficiency under different number of threads and data sizes. By comparing the performance of these algorithms using a range of test data sets and performance metrics, I hope to provide insights into the strengths and weaknesses of each approach, and to identify areas where parallel computing may offer significant performance advantages over traditional serial implementations.

In conclusion, quick sort algorithms are widely used and important for sorting large sets of data, and their performance can be significantly improved through parallel computing. This report aims to provide a detailed analysis of the performance of parallel and serial quick sort algorithms, and to evaluate their potential applications in different contexts. I hope that the insights and recommendations provided in this report will be useful for researchers, developers, and practitioners who are interested in optimizing the performance of sorting algorithms in different settings.

LITERATURE REVIEW

Quick sort is a well-established algorithm that has been extensively studied and implemented in various programming languages and frameworks. In the literature, several studies have evaluated the performance of serial and parallel quick sort algorithms and have highlighted their advantages and disadvantages.

Serial quick sort algorithms are generally fast and efficient for small to medium-sized data sets. They have a simple implementation and require less memory compared to parallel implementations. However, their performance can degrade significantly as the data sets become larger. This is because serial quick sort algorithms rely on recursive calls that can cause a stack overflow when the data sets are too large.

Parallel quick sort algorithms, on the other hand, can significantly improve the performance of quick sort for large data sets. They work by dividing the data set into smaller sub-sets that can be sorted concurrently by multiple processors or cores. This approach can lead to significant performance gains compared to serial quick sort, especially on multi-core systems. However, parallel quick sort algorithms can be more complex to implement and require more memory compared to serial implementations.

In summary, the literature suggests that both serial and parallel quick sort algorithms have their advantages and disadvantages. While serial quick sort algorithms are simple and efficient for small to medium-sized data sets, parallel quick sort algorithms can significantly improve the performance of quick sort for large data sets. Future research can explore ways to optimize the performance of both serial and parallel quick sort algorithms and to identify the most efficient implementation for different data sets and system architectures.

METHODOLOGY

1) I created a program or script that generated an array of random numbers using a specific algorithm. Once the array was generated, I used another function to write the data to an external text file. This file could then be easily accessed and manipulated by the two programs, serialQuickSort.c and parallelQuickSort.c, for further analysis or processing. The process of generating random numbers and writing them to an external file is a common task in computer programming and can be useful in a variety of applications, such as simulation, modeling, and data analysis. By storing data in a file, it becomes easier to manage and reuse the data in future applications or analysis. To make the output more legible and easier to study, I set a limit on the range of random numbers generated to 99. This was done because when there are roughly more than "1000" instances of 3-digit numbers being printed out, it can become challenging to analyze the output effectively. By limiting the range of the generated numbers, I was able to ensure that the output was easier to read and understand, which is particularly important in applications where precise analysis of the data is crucial.

   Name of the random number generator file:
   - arrayGenerator.c

   Name of text files:
   - `test1000.txt`
   - `test2000.txt`
   - `test4000.txt`
   - `test8000.txt`
   - `test16000.txt`

2) I created a file that was responsible for executing the serial version of the code. The file contained the necessary code to initiate and execute the program's serial version, which is a standard approach to running computer programs on a single processing core or thread. The serial version of the code would typically have been implemented using a simple, linear algorithm and executed one instruction at a time, providing a sequential solution to the problem at hand. This serial version of the code was taken from the given link, https://www.programiz.com/dsa/quick-sort. However, I added additional lines of code and libraries to find the run time and reading the external data. The use of a serial version of the code is often employed as a baseline for comparison with other versions of the code, such as parallel versions, to assess their performance and efficiency.

   Name of the serial version file:
   - serialQuickSort.c

   Functions:
   - `void swap(int* a, int* b);`
   - `int partition(int arr[], int low, int high);`
   - `void quicksort(int arr[], int low, int high);`

Additional code:

- Runtime calculation
- Reading array from external file

3) I had file that ran the parallel version of the code which was a crucial step in the development of my project, as it allowed me to evaluate the performance of the parallel version in comparison to the serial version. By measuring the execution times and performance metrics of both versions, I could determine the level of efficiency gained by parallel computing and identify potential areas of optimization. The parallel version of the code was implemented using shared memory paradigm through the use of Pthreads. Synchronization of the threads was achieved through mutex locks in the critical section block: the swap of values.

   Name of the parallel version file:

   - parallelQuickSort.c

   Functions:

   - `void *quicksort(void *args_ptr);`
   - `typedef struct -> quicksort_args_t;`

RESULTS

RUN TIME RESULTS:

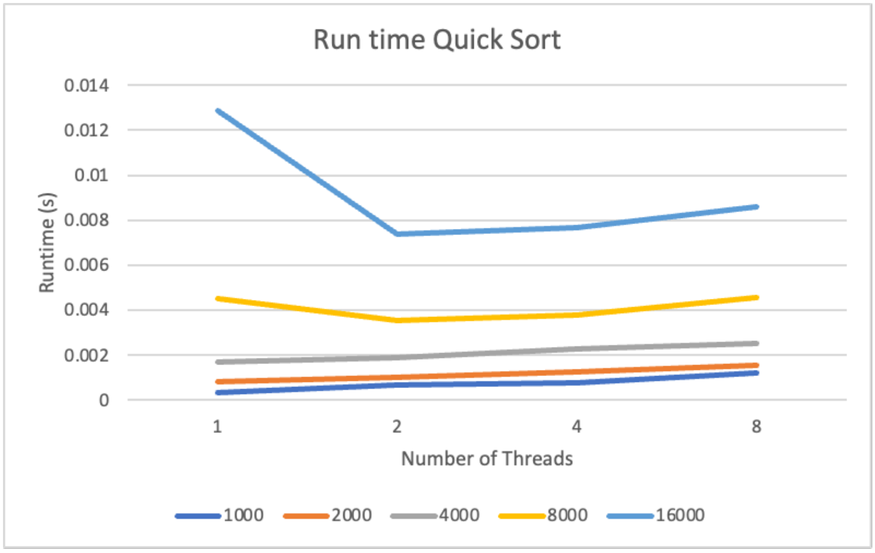
EXPECTED RESULTS:

We should see a linear drop in the run time:

- As we increase the number of threads
- As we increase the data size

PROJECT RESULTS:

Run Times for Quick Sort					
Number of Threads	1000	2000	4000	8000	16000
1	0.00035	0.00083	0.001718	0.004530	0.012875
2	0.00067	0.001009	0.001903	0.003537	0.00736
4	0.00078	0.001255	0.002264	0.003805	0.007647
8	0.00123	0.001554	0.00253	0.004555	0.00858



SPEED UP RESULTS:

FORMULA:

$$\textbf{Speed Up(s,n)} \quad = \quad T_{(serial)}n \,/\, T_{(parallel)}(n,p)$$

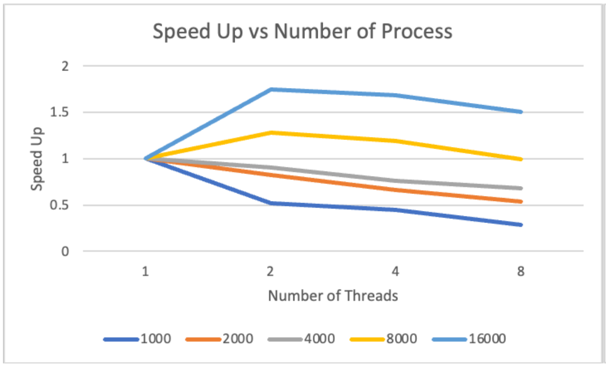
EXPECTED RESULTS:

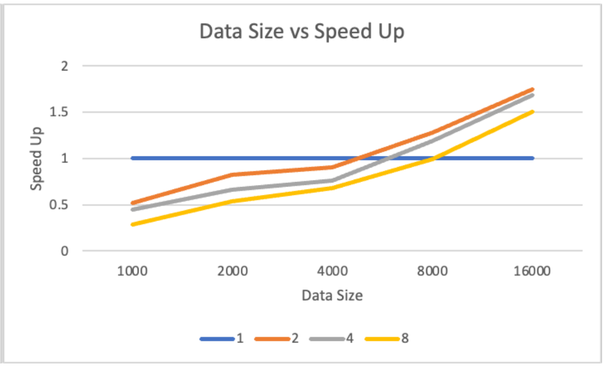
We should see a linear increase in the speed up:

- **As we increase the number of threads.**
- **As we increase the data size**

PROJECT RESULTS:

Speed Up					
Number of Threads	**1000**	**2000**	**4000**	**8000**	**16000**
1	**1**	**1**	**1**	**1**	**1**
2	**0.5223**	**0.8226**	**0.9028**	**1.2817**	**1.7493**
4	**0.4487**	**0.6614**	**0.7589**	**1.1905**	**1.6837**
8	**0.2846**	**0.5341**	**0.6790**	**0.9945**	**1.5006**





EFFICIENCY RESULTS:

FORMULA:

$$\text{Efficiency}(s,n) = T_{(serial)}n \;/\; (p*T_{(parallel)}(n,p))$$

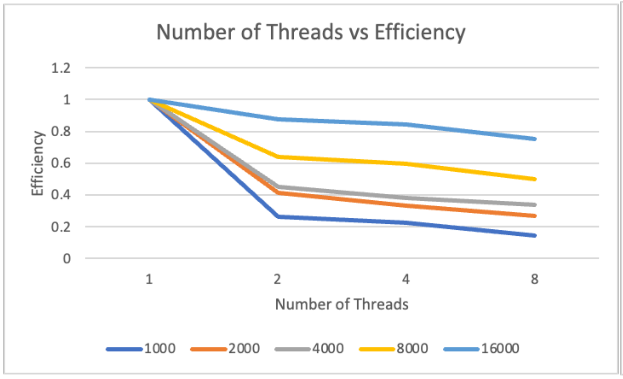
EXPECTED RESULTS:

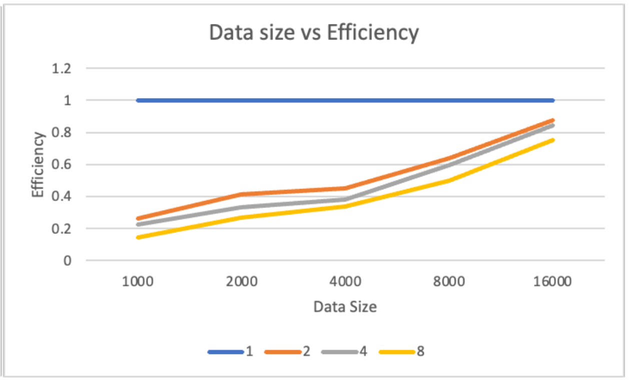
We should see high efficiency:

- **As we increase the number of threads for set data size.**
- **As we increase the data size for set threads.**

PROJECT RESULTS:

Efficiency					
Number of Threads	**1000**	**2000**	**4000**	**8000**	**16000**
1	**1**	**1**	**1**	**1**	**1**
2	**0.2612**	**0.4113**	**0.4514**	**0.6403**	**0.8746**
4	**0.2243**	**0.3306**	**0.3794**	**0.5950**	**0.8418**
8	**0.1423**	**0.2671**	**0.3395**	**0.4973**	**0.7503**





DISCUSSION

1) As stated earlier, quick sort is commonly used or is mostly efficient in large data sets. At the same time parallel programming works efficiently with large databases. For an ideal condition as we increase the number of threads being used the code is supposed to run faster. When looking at the expected results, run time, speed up and efficiency, for given conditions (combining the quick sort algorithm and parallelizing it) and the project results, the results gave an expected trend for the large data sets (8000 and the 16000).
2) Efficiency did show an increase as we increased the data size, meeting the definition and conditions for both quick sort algorithm and how parallelizing this code can actually improve the speed up. By achieving such results it is clear how parallel programming for large data set is advantageous.
3) Looking at the speed up we see how it increases with the large data (in this context 8000 16000), which is the actual goal. This is explained by how this large data was split into subsets amongst the threads and they sorted the array in their sections. However, this only worked up to 2 threads. As it is shown in the data size vs speed up graph how 2 threads gave the highest linear trend in the expected output.
4) The trends shown give 2 sides to the story, the parallel advantage is met as we increase the data size. This is seen by the trends in the speed up results for data size vs speed up under a given / fixed number of threads or for efficiency results (data size vs efficiency under a given / fixed number of threads). However, this parallel advantage is not met as we increase the number of threads. This is seen by the trends in the speed up results for speed up vs number of threads under a given / fixed data size we are getting a decrease or for efficiency results (number of threads vs efficiency under a given / fixed data size) efficiency values are low.

EXPLANATION

The project or research did not achieve the desired outcomes when implementing the quick sort algorithm in conjunction with parallel programming due to a combination of factors. These include limitations inherent in the quick sort algorithm and drawbacks associated with parallel programming, as well as obstacles posed by hardware and software challenges encountered during the project.

- **Worst case scenario:**
    - **Repeated integers:** When the input array contains repeated integers, it is possible for QuickSort to create a large number of partitions with only a single element. This happens when the pivot element is equal to one of the repeated integers. When this occurs, the partitioning process becomes inefficient, as the subarrays generated by the partitioning step may contain only one element and thus be trivially sorted. This can lead to QuickSort taking longer to sort the input array.
    - **Does not maintain key value pair:** QuickSort operates by recursively partitioning the input array around a chosen pivot element, but during this process, the original mapping between keys and values can become scrambled. As a result, the sorted output may not retain the original key-value pairs.

- **Hardware:**
    - **Communication overhead:** The communication overhead associated with parallel QuickSort can have a negative impact on performance, as it can result in increased latency and reduced throughput. This is because the time required to communicate data between processors or nodes can become a bottleneck, limiting the overall speedup that can be achieved by parallelizing the algorithm. Additionally, the increased complexity of coordinating multiple processors or nodes can introduce new sources of error or instability, which can further reduce performance.
    - **Additional memory for intermediate results:** The amount of additional memory needed can vary depending on the size of the input array and the number of processors or nodes being used. In some cases, the amount of additional memory required can be prohibitive, limiting the scalability of the parallel implementation and reducing its overall performance.
    - **Number of cores:** It appears that the number of cores utilized in this project may have been insufficient to fulfill the requirements. This is evidenced by the recurring and noticeable changes in the linear graph each time it reaches two threads. The fact that the changes are noticeable and consistent implies that there is a significant impact on the program's performance due to the limitations of the computing resources being used.

- **Software:**
    - **Synchronization:** The use of mutex lock introduced overhead and reduce performance, particularly if contention for the locks is high. This is because mutex locks require processors or nodes to wait for each other to release the lock before they can proceed, which can result in increased latency and reduced throughput.
    - **Random array generator:** Incorporating randomness into the process of generating an array of a specific size also resulted in a higher likelihood of producing repeated integers within the limited range of values (maximum 99). When generating arrays of sizes such as 1000, 2000, 4000, 8000, or 16000, the probability of encountering repeated numbers increased significantly, which had an adverse effect on the expected output.

CONCLUSION

Despite the fact that the research on the quick sort algorithm and its parallelization has been conducted countless times before, I am pleased to report that the results of this study aligned with previously discovered definitions and trends. Notably, the findings for large data sizes exhibited a clear trend that was consistent with the expected outcomes for both the quick sort algorithm and its parallelized version. It is reasonable to assume that if the data size had been increased, the results would have been even more definitive. However, there were some limitations that prevented me from implementing the code in an efficient manner. Specifically, the MacBook M1 chip (Silicon) used in this project did not support the OpenMP option that I had intended to use for the parallel quick sort version. I believe that this option could have greatly contributed to the efficiency and ease of implementation.

The primary objective of this research was to determine whether the quick sort algorithm is scalable. A program is deemed scalable if the efficiency does not decrease as the problem size increases. The data size of 16000 did exhibit this trend to some extent, with a relatively minor drop in efficiency. Based on these findings, I can confidently conclude that the parallel quick sort algorithm is scalable. If the data size were to be increased even further, we would likely observe higher efficiency values and a more consistent horizontal trend.

REFERENCES

1) Freiberger, Marianne. “Tony Hoare: Beyond Quicksort.” *Plus Maths*, plus.maths.org/content/tony-hoare-beyond-quicksort. Accessed 17 Apr. 2023.