AISSMS
INSTITUTE OF INFORMATION TECHNOLOGY
ADDING VALUE TO ENGINEERING

Department of Computer Engineering

A Design and Analysis of Algorithm

PROJECT REPORT ON

MULTITHREAD SORTING ALGORITHM (Merge Sort)

SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERINGAISSMS IOIT

BE Computer Engineering

SUBMITTED BY

STUDENT NAME	ERP No:
Onasvee Banarse	**09**
Kaustubh Kabra	**37**
Akash Mete	**50**
Harsh Shah	**65**

2022 -2023

Department of Computer Engineering

CERTIFICATE

This is to certify that the project report
"MULTITHREAD SORTING ALGORITHM (Merge Sort)"

Submitted by

STUDENT NAME	ERP No:
Onasvee Banarse	09
Kaustubh Kabra	37
Akash Mete	50
Harsh Shah	65

is a bonafide students at this institute and the work has been carried out by them under the supervision of **Prof. Prashant Sadaphule** and it is approved for the partial fulfillment of the Department of Computer Engineering AISSMS IOIT.

(**Prof. Prashant Sadaphule**) (**Dr. S.N.Zaware**)

Mini-Project Guide Head of Computer Department

Place: Pune Date:

Abstract

Almost all the modern computers today have a CPU with multiple cores, providing extra computational power. In the new age of big data, parallel execution is essential to improve the performance to an acceptable level. With parallelisation comes new challenges that needs to be considered.

In this work, parallel algorithms are compared and analysed in relation to their sequential counterparts, using the Java platform. Through this, find the potential speedup for multithreading and what factors affects the performance. In addition, provide source code for multithreaded algorithms with proven time complexities.

A literature study was conducted to gain knowledge and deeper understanding into the aspects of sorting algorithms and the area of parallel computing. The data gathered was studied and analysed with its corresponding source code to prove the validity of parallelisation. Multithreading does improve performance, with two threads in average providing a speedup of up to 2x and four threads up to 3x. However, the potential speedup is bound to the available physical threads of the CPU and dependent of balancing the workload.

Contents

1. Introduction

Increasing performance is the basis for improving solutions for a given problem to its fullest potential by utilizing every tool and resource to our advantage. In computer science, the resources and tools could be seen as the hardware and software, while the performance would be the collaboration between the two. To fully utilize and increase performance, not only must the problem, resources and tools be understood, but also when and how to use them at appropriate situations.

The given problem in this work, is one of the biggest fundamental problems, namely sorting. Sorting have always been of interest and solutions have been extensively studied and documented. Numerous solutions have been proposed and accepted, using different designs and techniques, each with its own advantage and disadvantage.

The aim in this work is to evaluate the performance of merge sort and its approaches, as to what advantages and disadvantages they have and why they occur. In addition, this work also aims to evaluate an improvement technique of using extra raw computer power, namely multithreading. Improving the performance by utilizing multithreading, introduces new problems and factors, which needs to be considered. Knowing and understanding these factors, could prove useful when utilized correctly.

For this work, the given implementations are all in Python and C++, and have been thoroughly tested and analysed. However, it needs to be emphasised that focus does not lay on optimization for the given implementations. Instead, the implementations show what is possible in terms of increasing performance, through the results and analysis of this work.

2. Problem Statement

Implement merge sort and multithreaded merge sort. Compare time required by both the algorithms. Also analyse the performance of each algorithm for the best case and the worst case.

In this Project we will focus on following questions:

- How much can an algorithm's efficiency and performance be improved by multithreading?
- What affects the performance and efficiency of algorithms, how are they different in multithreaded environments?
- Is there a threshold for multithreaded algorithms in terms of performance, and what are the reasons for it?
- How does unbalanced workload in multithreaded environments change the performance of an algorithm?

3. Software Requirement Specification

Software Used:

- Python (version 3 or above)-

  Python is a high-level, general-purpose programming language. Its design philosophy emphasizes code readability with the use of significant indentation. The sentiment analysis is performed using python language and packages.

- Cpp –

  C++ is a high-level general-purpose programming language created by Danish computer scientist Bjarne Stroustrup as an extension of the C programming language, or "C with Classes"

- VScode or any IDE–

  Visual Studio Code is a source-code editor made by Microsoft for Windows, Linux and macOS. Features include support for debugging, syntax highlighting, intelligent code completion, snippets, code refactoring, and embedded Git.

4. Hardware Specification

The detailed hardware used for the project are:

Item	Description
System	HP OMEN 15 series
Processor	AMD Ryzen 5 4600H
RAM	8 GB
System Type	64-bit operating system, x64-based processor
SSD	256 GB Solid State Drive
HDD	1 TB Hard Disk Drive
Graphics	NVIDIA 4 GB Graphic Card
Operating System	Windows 11 Operating System

5. Theory

1. Asymptotic computational complexity

Algorithms are described and analysed with respect to asymptotic computational complexity. There are three major asymptotic notations that are used for analysis. These describe asymptotic boundaries for algorithm's upper and lower bounds as growth functions. With this notation it is possible to describe mathematically and prove algorithms as worst, best and average cases due to their asymptotic boundaries.

For a function **f(n)** the asymptotic behaviour is the growth of f(n) as n gets large. Small input values are not considered. Our task is to find how much time it will take for large value of the input.

For example, f(n) = c * n + k as linear time complexity. f(n) = c * n^2 + k is quadratic time complexity.

The analysis of algorithms can be divided into three different cases. The cases are as follows −

- **Best Case** − Here the lower bound of running time is calculated. It describes the behaviour of algorithm under optimal conditions.
- **Average Case** − In this case we calculate the region between upper and lower bound of the running time of algorithms. In this case the number of executed operations is not minimum and not maximum.
- **Worst Case** − In this case we calculate the upper bound of the running time of algorithms. In this case maximum number of operations are executed.

2. Time complexity

Time complexity is the means to describe an algorithm's execution time for a problem with given input. The time complexity of an algorithm is defined with the previously mentioned cases and that they are dependent on the internal structure and order of the given input. Consider that an algorithm sorts of input

that is already sorted. This case can be the best case of an algorithm since it only needs to check if the list is sorted. However, when an input is sorted in reversed order, some algorithms time complexity will spike, and be defined as their worst case.

3. Space Complexity

Memory usage is another important aspect of sorting algorithms, which in some cases can be more important than the execution time. Ideally, one wants to keep the value for this parameter small, but extra memory can often improve execution times, which leads to a time-space trade-off. One cannot have both, and therefore needs to find a reasonable compromise for what is best for the intended usage. For this reason, in-place sorting algorithms are often slower than its counterpart.

4. Growth functions

Growth functions are used as a common way to describe algorithms time complexity, space complexity, computational resources, and so forth. In time complexity, growth functions describe the runtime of an algorithm in relation to its purpose, including sorting, searching, insertion, deletion and so forth.

4.1 Asymptotic Notations

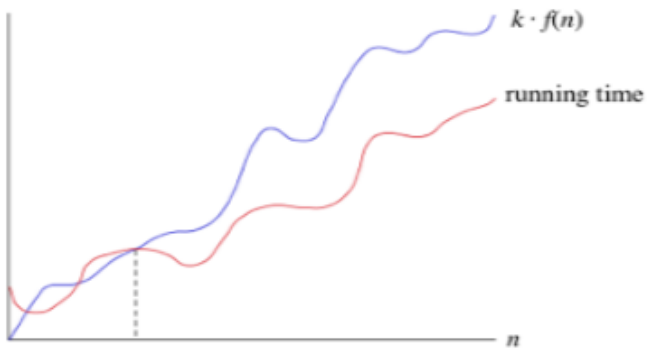
Asymptotic Notation is a way of comparing function that ignores constant factors and small input sizes. Three notations are used to calculate the running time complexity of an algorithm:

1. **Big-oh notation:**

   Big-oh is the formal method of expressing the upper bound of an algorithm's running time. It is the measure of the longest amount of time. The function **f (n) = O (g (n))** [read as "f of n is big-oh of g of n"] if and only if exist positive constant c and such that

   $$f(n) \leqslant k.g(n) f(n) \leqslant k.g(n) \text{ for } n > n0 n > n0 \text{ in all case}$$

Hence, function g (n) is an upper bound for function f (n), as g (n) grows faster than f (n)



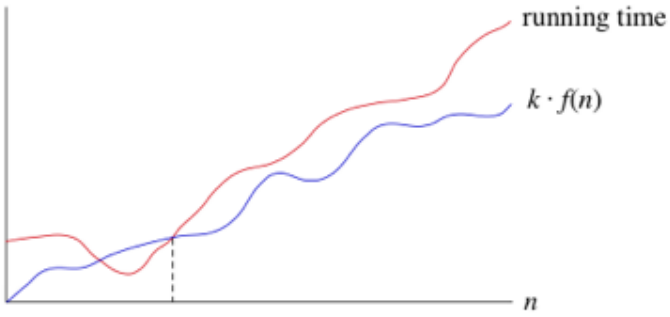
ASYMPTOTIC UPPER BOUND

Hence, the complexity of **f(n)** can be represented as O (g (n))

2. Omega () Notation:

The function f (n) = Ω (g (n)) [read as "f of n is omega of g of n"] if and only if there exists positive constant c and n_0 such that

$$F(n) \geq k^* g(n) \text{ for all } n,\ n \geq n_0$$



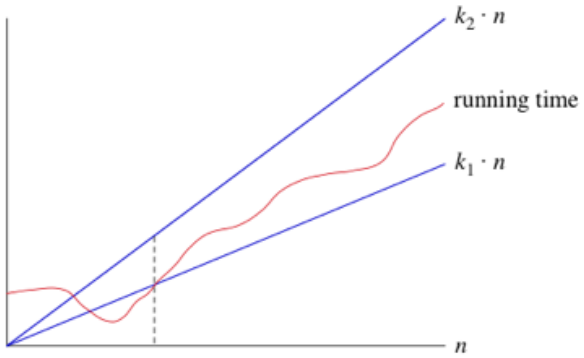
ASYMPTOTIC LOWER BOUND

Hence, the complexity of **f (n)** can be represented as Ω (g (n))

3. Theta (θ):

The function f (n) = θ (g (n)) [read as "f is the theta of g of n"] if and only if there exists positive constant k_1, k_2 and k_0 such that

$$k_1 * g(n) \leq f(n) \leq k_2\, g(n) \text{ for all } n,\ n \geq n_0$$



ASYMPTOTIC TIGHT BOUND

Hence, the complexity of f (n) can be represented as θ (g(n)).

5. Merge sort

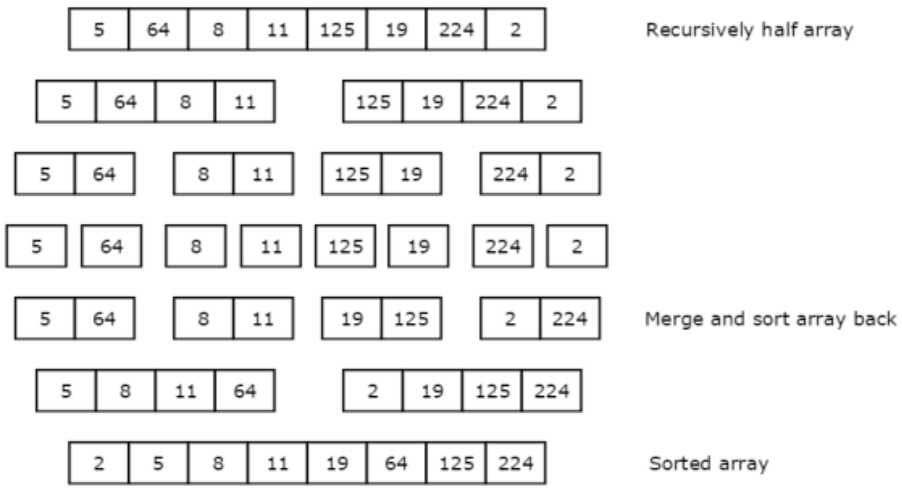
Definition Splits data set in half, sorts each half recursively, and merges them back together to a sorted set.

Time Complexity	
Worst case	O (n log n)
Best case	O (n log n)
Average case	O (n log n)
Space complexity	
Worst case	O(n) auxiliary

5.1 Algorithm

Merge sort is a stable divide and conquer algorithm and is one of the fastest. It recursively divides the data set into subarrays until each subarray consists of a single element. It then merges each subarray back, sorting each new subarray as it builds its way back to a single sorted array. Regardless of the shape of the data set to be sorted, merge sort performs the same number of steps, and will therefore have the same time complexity for all cases, O (n log n). Even though it is an efficient algorithm in terms of sorting, it has a drawback in that it uses O(n) extra memory when sorting. This makes it inefficient if memory usage is a key aspect when choosing a sorting algorithm to use. Due to it being a divide and conquer algorithm, it is possible to implement it in parallel.

5.2 Example



The data set is recursively split in halves, until each element is on its own. Through the recursion, the subsets are merged back together, being sorted in the same instance. In this example, after the data set have been split down to single elements, elements 5 and 64 are merged into a new subset. In this merge sequence, the elements are compared to each other and placed in a sorted order. In the next step, this subset merges with another one, in this case, a subset containing elements 8 and 11. Once again, the elements are compared and placed in a sorted order while merging. With only two subsets remaining, a final merge is done, resulting in sorted data set.

6. Parallel computing

Most modern computers today consist of two or more cores within the CPU. This makes it possible to run several instructions simultaneously. The design of these multicore processor varies. A homogeneous multi-core system has cores that are identical, while heterogeneous multi-core systems have cores that are not identical. Multi-core systems can share cache-memory or have individual memory. Some may implement message passing or have a shared memory intercommunication. The performance of multi-core systems is still constricted to how algorithms utilize the resources available. Without an efficient algorithm that take use of the extra resources a multi-core system provides, there will not be any substantial gain in performance.

7. Multithreading

Multithreading is the ability of a program or an operating system to enable more than one user at a time without requiring multiple copies of the program running on the computer. Multithreading can also handle multiple requests from the same user.

Each user request for a program or system service is tracked as a thread with a separate identity. As programs work on behalf of the initial thread request and are interrupted by other requests, the work status of the initial request is tracked until the work is completed. In this context, a user can also be another program. Fast central processing unit (CPU) speed and large memory capacities are needed for multithreading. The single processor executes pieces, or threads, of various programs so fast, it appears the computer is handling multiple requests simultaneously.

8. Python Multithread Implementation

Starting a New Thread

To spawn another thread, you need to call following method available in thread module −

thread.start_new_thread (function, args[, kwargs])

This method call enables a fast and efficient way to create new threads in both Linux and Windows. The method calls return immediately, and the child thread starts and calls function with the passed list of *args*. When function returns, the thread terminates. Here, *args* is a tuple of arguments; use an empty tuple to call function without passing any arguments. *kwargs* is an optional dictionary of keyword arguments.

The Threading Module

The threading module exposes all the methods of the *thread* module and provides some additional methods −

- **threading.activeCount()** − Returns the number of thread objects that are active.
- **threading.currentThread()** − Returns the number of thread objects in the caller's thread control.
- **threading.enumerate()** − Returns a list of all thread objects that are currently active.

In addition to the methods, the threading module has the *Thread* class that implements threading. The methods provided by the *Thread* class are as follows

- **run()** − The run() method is the entry point for a thread.
- **start()** − The start() method starts a thread by calling the run method.
- **join([time])** − The join() waits for threads to terminate.
- **isAlive()** − The isAlive() method checks whether a thread is still executing.
- **getName()** − The getName() method returns the name of a thread.
- **setName()** − The setName() method sets the name of a thread.

6. Code and Output

Test environment

The computer specification used for the experiment was:

CPU: AMD Ryzen 5 4600H, 12MB cache, 6 cores, 12 threads

RAM: 8GB DDR4 RAM

OS: Windows 10 - 64-Bit Edition, installed on an SSD

Python: Version 3.9, x64

Code

Parallel merge sort versus sequential and built in. Using multiprocessing Process/Pipe.

$ python [3] parallelMergesort.py [500000]

This code exemplifies the Process/Pipe paradigm of parallel design. Merge sort says simply:

- sort the left side of the list.
- sort the right side of the list.
- merge the results.

So, a parallel version comes from the realization that the two sorts are independent of one another.

- sort the left and right sides in parallel
- merge the results.

Process objects are instantiated to perform the sorting on the left and right sub lists. Pipes are used to transmit the sorted sublists back up the execution tree. This merge sort is not standard; it is NlogN in memory and does not sort the argument lyst as a side effect, but rather returns a sorted version of the lyst. See:

http://docs.python.org/library/multiprocessing.html
http://docs.python.org/py3k/library/multiprocessing.html

Code Functions:

1. main () function:
   This is the main method, where we
   -generate a random list
   -time a sequential merge sort on the list.
   -time a parallel merge sort on the list.
   -time Python's built-in sorted on the list.
2. merge (left, right) function:
   Returns a merged and sorted version of the two already-sorted lists.
3. mergesort(list) function:

   The recursive merge sort. Returns a sorted copy of list.

4. def mergeSortParallel(list, conn, procNum) function :

   mergSortParallel receives a list, a Pipe connection to the parent, and procNum. Merge sort the left and right sides in parallel, then merge the results and send over the Pipe to the parent.

5. def isSorted(list):
   Return whether the argument list is in non-decreasing order.

Output:

Case 1: For 10,000 elements

```
Number of Elements in list:100000
Sequential mergesort: 0.675306 sec
Parallel mergesort: 1.692440 sec
Built-in sorted: 0.043060 sec

Process finished with exit code 0
```

Case 2: For 50,000 elements

```
Number of Elements in list:50000
Sequential mergesort: 0.273073 sec
Parallel mergesort: 1.376139 sec
Built-in sorted: 0.019020 sec

Process finished with exit code 0
```

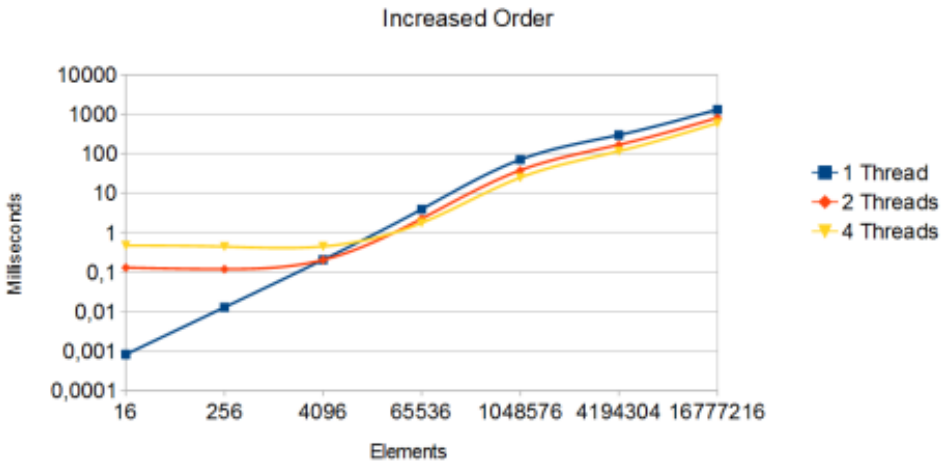
Case 3: For 5,00,000 elements

```
Number of Elements in list:500000
Sequential mergesort: 3.789763 sec
Parallel mergesort: 2.573457 sec
Built-in sorted: 0.229069 sec

Process finished with exit code 0
```

Result:

Following graphs, the average execution times for merge sort on different data sets, where T represents the number of threads used in execution.



7. Conclusion

In this project it is shown that utilizing multithreading as a source for more raw computer power, indeed increases the performance of an algorithm. However, the increase in starts off when the data being processed is sufficiently big enough that the overhead for thread creation is a smaller factor than the overall execution time.

Overall, the results shows that in general, executing two threads gives an increase in performance of up to 2x the original execution time. Using four threads, provides an increase of 2.5x to 3x the original execution time.

This project has focused on parallel execution using sorting algorithms. However, there are many other types of algorithms that could be studied further, such as searching, dynamic programming, mathematical calculations, or other types of data structures. There are many aspects of the Python platform that could be further researched.

8. References

[1] Wang, D., Zhang, X., Men, T., Wang, M., & Qin, H. (2012, March). An implementation of sorting algorithm based on Java multi-thread technology. In Computer Science and Electronics Engineering (ICCSEE), 2012 International Conference on(Vol. 1, pp. 629-632). IEEE.

[2] Mahafzah, B. A. (2013). Performance assessment of multithreaded quicksort algorithm on simultaneous multithreaded architecture. The Journal of Supercomputing, 66(1), 339-363.

[3] Qian, X. J., & Xu, J. B. (2011, May). Optimization and implementation of sorting algorithm based on multi-core and multi-thread. In Communication Software and Networks (ICCSN), 2011 IEEE 3rd International Conference on (pp. 29-32). IEEE.

[4] Aater Suleman. (2011) Parallel programming: How to choose the best task-size?
http://www.futurechips.org/software-for-hardware-guys/parallelprogramming-choose-task-size.html
(Accessed 2015, June)

[5] Thread-based parallelism:
https://docs.python.org/3/library/threading.html

[6] An Intro to Threading in Python:
https://realpython.com/intro-to-python-threading/