**OS PROJECT REPORT**

**COMPARING THREADS AND PROCESS**

**21K-3153**

**21K-3372**

**21K-3200**

**Objective:**

Our objective is to compare execution time, or seconds taken by the CPU, to perform 5 sorting algorithms of 100,1000,10000, 100 000 and 1000 000 (one million) elements, using a single thread (process), two threads and 4 threads. We aim to find out which implementation suits which algorithm best and how multithreading affects the speed of the algorithm.

**Introduction:**

We have chosen 5 sorting algorithms, namely

1. Merge Sort
2. Bubble Sort
3. Quick Sort
4. Insertion Sort
5. Selection Sort

and used them to sort 100,1000,10000,100 000 and 1 million integers, recorded the times and plotted comparison graphs.

**Platform and Languages:**We worked on Linux, specifically WSL, and used the Ubuntu distro. Our codes are written in CPP. For threading we used two libraries interchangeably, the POSIX library and the <thread> library included with CPP.

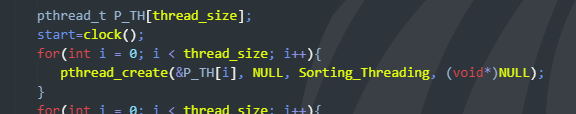
**METHODOLOGY:**

**NOTE:** as we need to merge the arrays after they have been sorted by the threads, we have implemented versions of merge sort to merge the sorted parts back together.

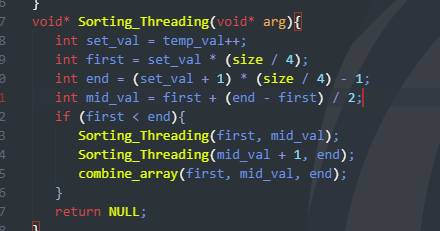
**MERGE SORT:**

We used a simple merge sort code for the merge sort single thread process.

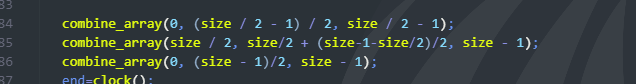
For merge sort multi-threading, we used the POSIX library, made an array of threads based on available cores:



split the data, and made each thread sort a part of the data:



In the end, we merged all the sorted parts together, completing the sorted array:

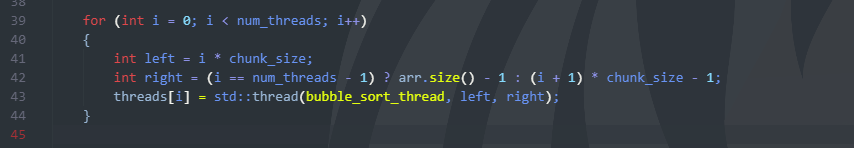


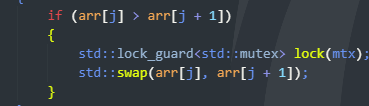
**BUBBLE SORT:**

In Bubble sort threading, we first made a vector of available threads, divided the data by number of threads to split it for each thread:



After that we calculated a start and end point (left and right) for each thread, so that each thread sorts a different part of the data, and then bubble sorted each chunk to make the final array.

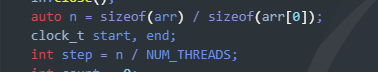


We used the <thread> library to make threads and used mutex to sync threads: 

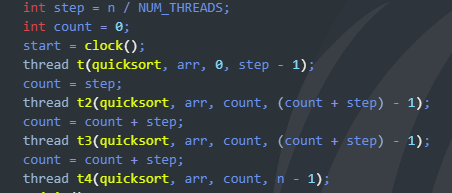
The <lock\_guard> automatically releases the lock as soon as the destructor is called, thus we don’t need to release the lock ourselves.

**Quick Sort:**

The same method applied to quick sort, we made an array of threads and calculated a ‘step’ by dividing the number of elements by the number of threads (ie 100 elements divided by 4 threads would make step be equal to 25). <thread> library is used



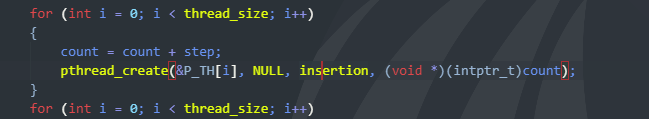
As quicksort requires both a start and end integer, instead of sending 0 and n-1 as start and end, we split elements and calculate different starts and ends for each thread:

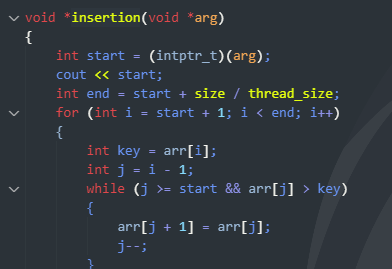


Leading to the entire array being sorted in parallel.

**Insertion Sort:**

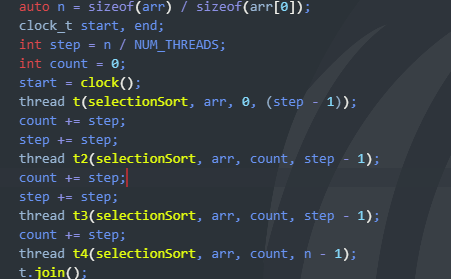
We go back to the POSIX library, but apply the same logic. Split the data, make threads, calculate different start and end points for each thread, and make each thread sort data

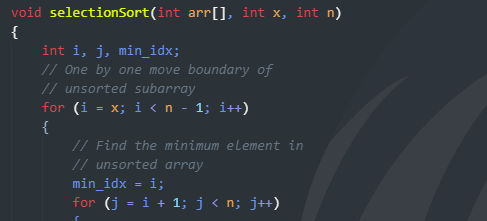




**Selection Sort:**

Same logic as insertion and quick applies in selection, instead of making the selection sort loop begin from 0, we made it into a variable to allow for data splitting



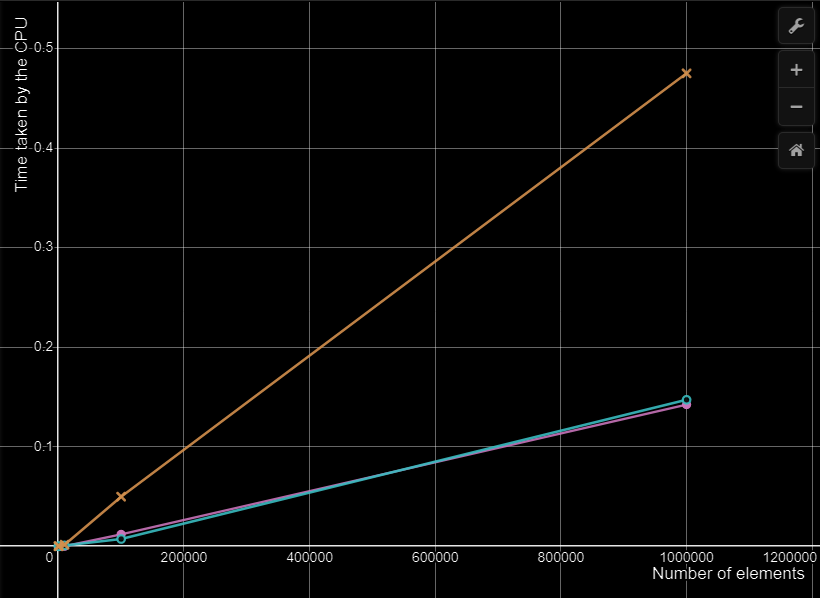


**RESULTS:**

**MERGE SORT:**

|  |  |  |  |
| --- | --- | --- | --- |
| **No of Elements** | **Time taken by process in seconds** | **Time taken by multithreading (2 cores)** | **Time taken by multithreading (4 cores)** |
| 100 | 0.000010 s | 0.000278 seconds | 0.000295 seconds |
| 1000 | 0.000142 | 0.000150 | 0.000305 |
| 10,000 | 0.001805 | 0.001098 | 0.0018 |
| 100,000 | 0.012138 | 0.0075 | 0.05 |
| 1000,000 | 0.142443 | 0.147281 | 0.475 |

**Graph comparison:**

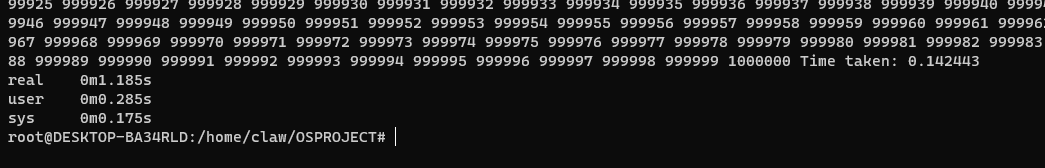


**Purple = process**

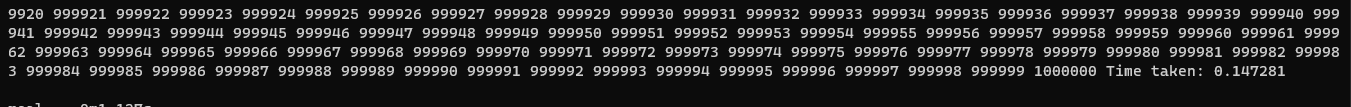
**Blue = 2 cores**

**Yellow = 4 cores**

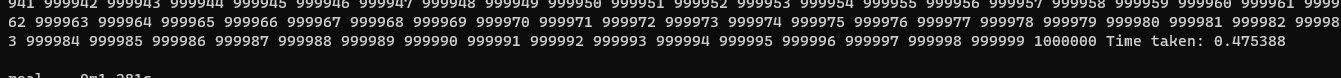
**1 million elements sorted using Merge Sort Process:**

****

**1 million elements sorted using Merge Sort Thread, 2 cores**

****

**1 million elements sorted using Merge Sort Thread 4 cores**

****

**Findings:**

You can see that the single thread and 2 core thread take roughly the same time, while 4 cores take significantly more time.

This is because while threading is faster with 2 cores, the overhead and context switching in threading due to recursion in merge sort negates that speed, and makes execution time of threading and single process almost same.

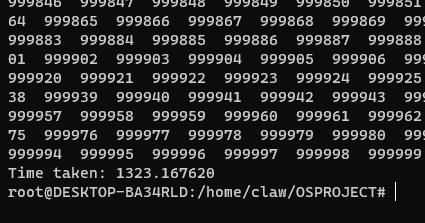
In 4 cores however, due to more cores needing to be context switched and having more overhead due to a lot of recursions, especially with larger elements like 1 million, the overhead adds up and 4 core threading takes much longer. As merge sort works on divide and conquer rule so we have to merge all the parts at the end which is time consuming, and takes longer with 4 cores.

This can be made better by splitting data more efficiently, or better management of threads.

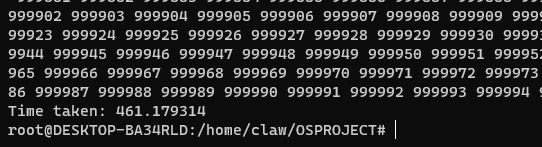
**Bubble Sort:**

|  |  |  |  |
| --- | --- | --- | --- |
| **No of Elements** | **Time taken by process in seconds** | **Time taken by multithreading (2 cores)** | **Time taken by multithreading (4 cores)** |
| 100 | 0.000010 s | 0.000289 seconds | 0.000603 seconds |
| 1000 | 0.000142 | 0.000146 | 0.000516 |
| 10,000 | 0.001805 | 0.262 | 0.633 |
| 100,000 | 12 seconds | 4.481 | 1.105 |
| 1000,000 | 1363 seconds, 22 minutes | 461 | 116 |

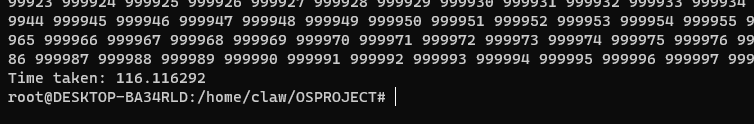
**1 million elements sorting using bubble sort process:**



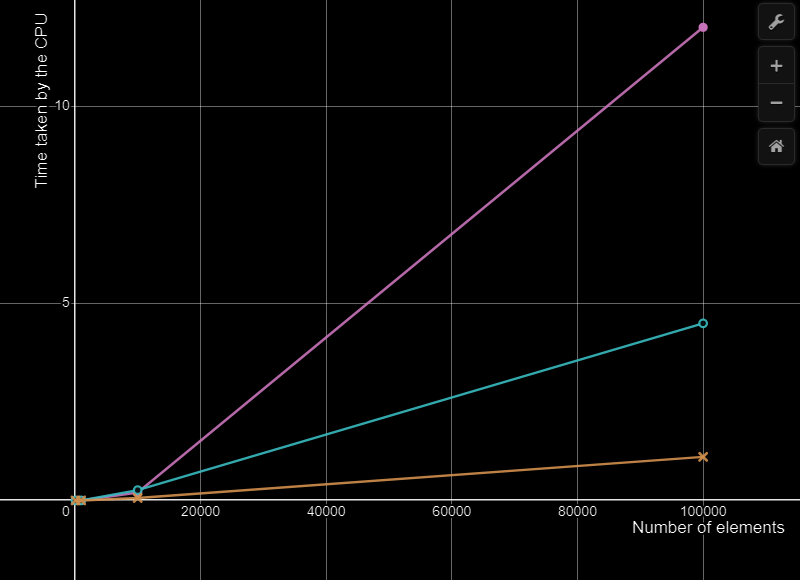
**1 million elements sorting using multithreading bubble sort (2 cores) (average of 3 times taken):**

****

**1 million elements sorting using multithreading bubble sort (4 cores) (average of 3 times taken):**

****

**Graph:**

****

**Purple = process**

**Blue = 2 cores**

**Yellow = 4 cores**

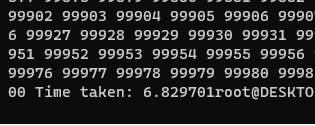
**Findings:**

Threading here makes a significant improvement to speed, with 4 core threading being the fastest, taking 116 seconds to sort a million numbers, while the Process takes 20 minutes! The splitting of data for each thread to sort separately makes the algorithm much faster.

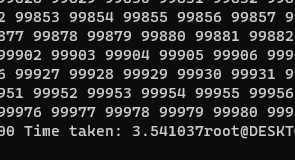
**QUICK SORT:**

|  |  |  |  |
| --- | --- | --- | --- |
| **No of Elements** | **Time taken by process in seconds** | **Time taken by multithreading (2 cores)** | **Time taken by multithreading (4 cores)** |
| 100 | 0.000016 s | 0.001 | 0.0008 |
| 1000 | 0.001635 | 0.001 | 0.0011 |
| 10,000 | 0.002213 | 0.002 | 0.0005 |
| 100,000 | 13 | 6.8297 | 3.5410 |

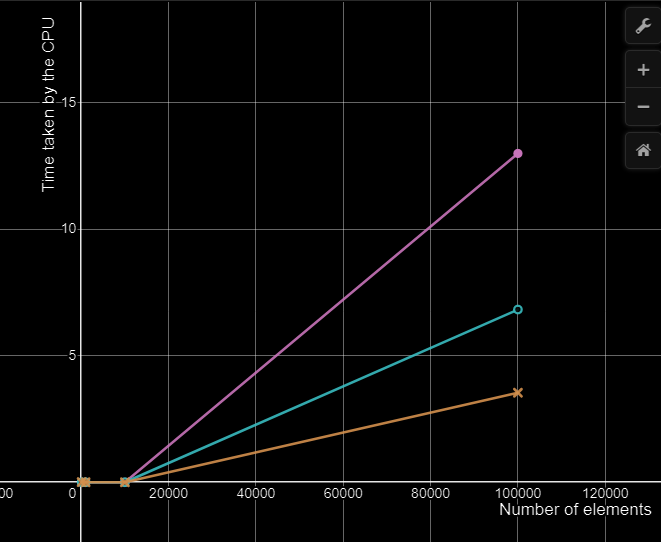
**1 hundred thousand elements sorting using multithreading quick sort (2 cores) (average of 3 times taken):**

****

**1 hundred thousand elements sorting using multithreading quick sort (4 cores) (average of 3 times taken):**

****

**Graph:**

****

**Purple = process**

**Blue = 2 cores**

**Yellow = 4 cores**

**Findings:**

Quick Sort threading also makes a significant improvement to speed, sorting 100 000 integers using one thread took 13 seconds, while using 4 core multi-threading with data split, it was reduced to 3 seconds! A whole 10 second difference.

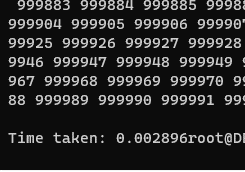
Quick Sort is our second faster algorithm due to its recursive calls, context switching and constant recalculation of beginning and end points.

If not for those, it could have been the faster algorithm, but the overhead slows it down.

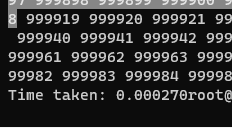
**Insertion Sort:**

|  |  |  |  |
| --- | --- | --- | --- |
| **No of Elements** | **Time taken by process in seconds** | **Time taken by multithreading (2 cores)** | **Time taken by multithreading (4 cores)** |
| 100 | 0. 00001 | 0.0004 | 0.0002 |
| 1000 | 0.00004 | 0.00015 | 0.0005 |
| 10,000 | 0.0746 | 0.0286 | 0.000275 |
| 100,000 | 0.000257 | 0.0002 | 0.000358 |
| 1000,000 | 0.002811 | 0.000359 | 0.000409 |

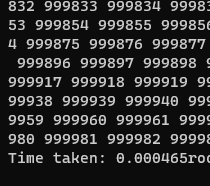
**1 million elements sorting using insertion sort process:**



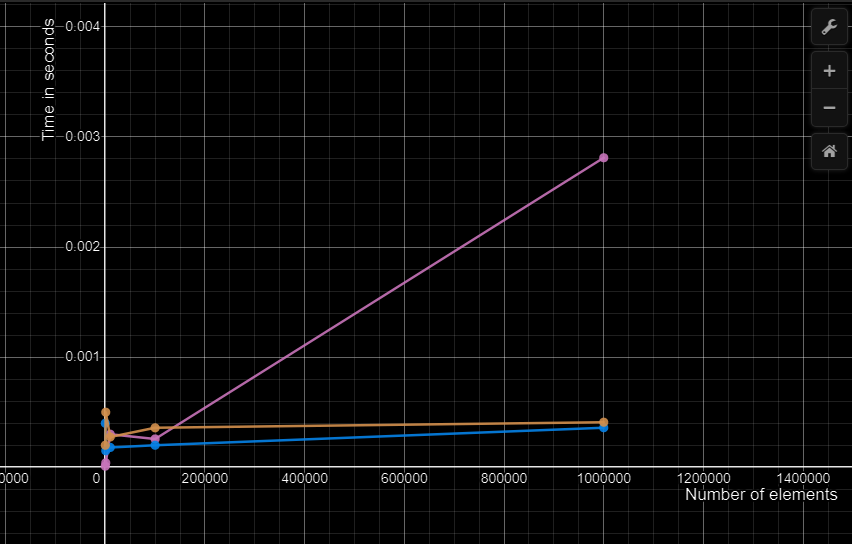
**1 million elements sorting using insertion sort threading (2 cores) (average of 3 times taken):**

****

**1 million elements sorting using insertion sort threading (4 cores) (average of 3 times taken):**

****

**Graph:**

****

**Purple = process**

**Blue = 2 cores**

**Yellow = 4 cores**

**Findings:**

Insertion Sort is the fastest algorithm we tested, with it working even better on larger numbers like 1 million integers of 100 000.

The simple reason for it being faster is because insertion sort has less overhead. It’s faster than even Quick Sort as quick sort has recursive calls and Insertion doesn’t.

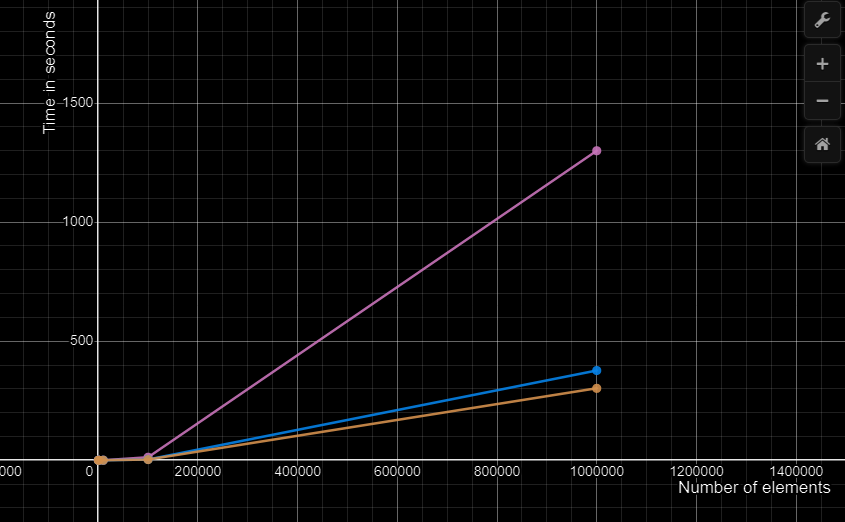
To sort a million numbers, Insertion sort only too 0.0003 seconds (2 cores) and 0.0004 seconds (4 cores)!

However, threading also made a difference in speed, as the single thread process took 0.002 seconds, being a hundredth of a second slower than multi thread.

**Selection Sort:**

|  |  |  |  |
| --- | --- | --- | --- |
| **No of Elements** | **Time taken by process in seconds** | **Time taken by multithreading (2 cores)** | **Time taken by multithreading (4 cores)** |
| 100 | 0. 00001 | 0.000145 | 0.0007 |
| 1000 | 0.0014 | 0.000251 | 0.0009 |
| 10,000 | 0.1319 | 0.0238 | 0.01041 |
| 100,000 | 12.7 | 3.447 | 2.794 |
| 1000,000 | 1300, 20 minutes | 377 ,6 minutes | 302,5 minutes |

**Graph:**

****

**Purple = process**

**Blue = 2 cores**

**Yellow = 4 cores**

**Findings:**

Selection sort is our slowest yet, with each iteration of sorting a million numbers taking minutes. This is because selection sort is always quadratic and does not have adjustments to make to its algorithm like other algorithms do. (For e.g., Insertion sort is linear in best case) Selection sort will always be quadratic in every case.

Multi-threading does make the algorithm faster though, where it took 20 minutes to sort 1 million numbers for a single threaded process, 4 core multi-threading did it in 5 minutes.

**Conclusion:**

Multi-threading does make sorting algorithms faster, but only if optimized correctly. Data should be split evenly the overhead due to recursive calls and context switching should be kept in mind and the type of threading used should be considered.

Sometimes, a single threaded process might be best (i.e. merge sort), and multi-threading might be slower, or sometimes and 4 thread multi thread process might be faster (i.e. Insertion sort). Each algorithm has different ways of implementation regarding different types of data.