Tour d'Algorithms: OpenMP Qualifier

Programming Project I

CS-516

Computer Architecture

SIUE

Submitted by:

Krishna Khanal-800771289

Sushrit Kafle-800792522

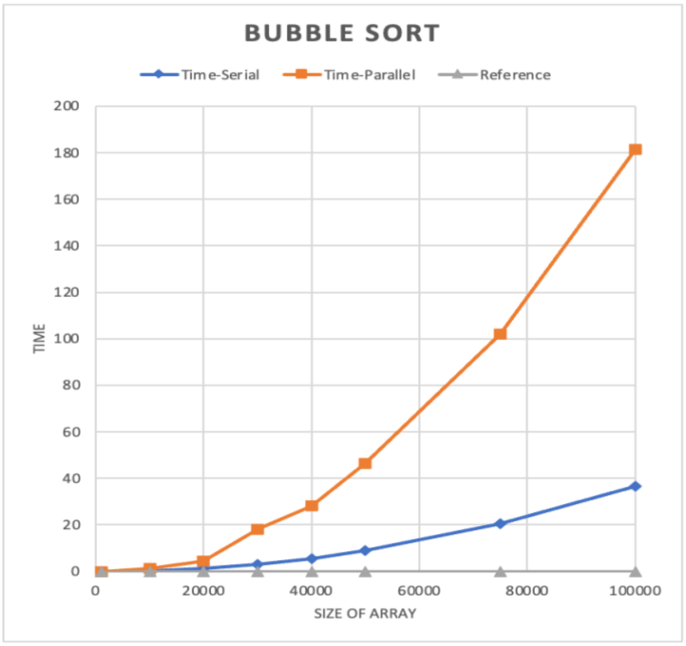
Venkatesh Bollineni-800792618

Table of Contents

1. Bubble Sort

Bubble sort is a simple sorting algorithm that makes multiple passes through an array and swaps the largest value to the end in every pass. It is not very efficient as it has O(N^2) time complexity in worst case scenario.

Array Size	Time-Serial	Time-Parallel	Reference
1000	0.00293031	0.00609241	0.00027
10000	0.303545	1.17596	0.0032585
20000	1.34922	4.54452	0.0069217
30000	3.1086	18.1588	0.0103697
40000	5.61967	28.3545	0.0138108
50000	8.94791	46.4675	0.0174637
75000	20.4141	102.136	0.0258657
100000	36.4615	181.561	0.0338745

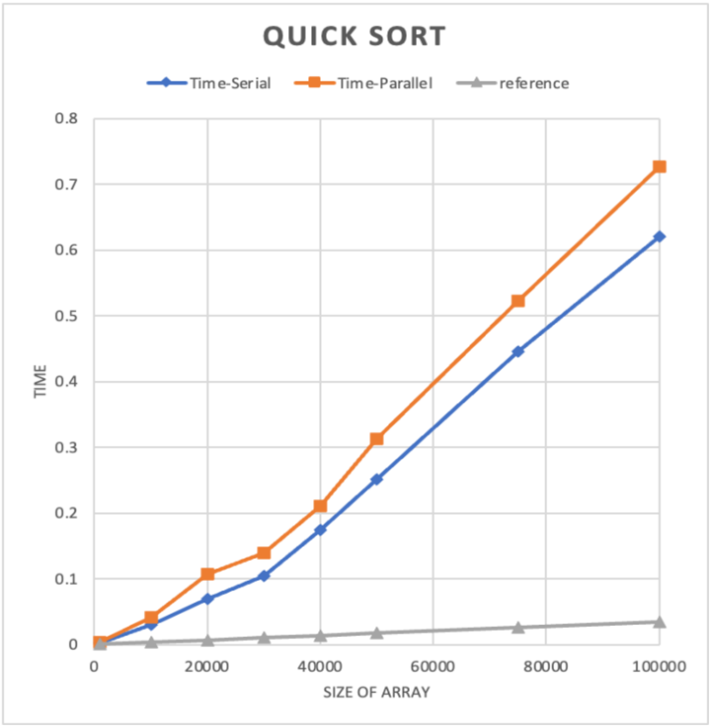


The above graph depicts that the growth rate of bubble sort is very high compared to the reference STL sort method. More specifically, the parallel algorithm using OpenMP has more rapid rate of growth compared to the serial one. Some reasoning for this could possibly be:

1. Bubble sorting requires a lot of **swapping**; it isn't very efficient for parallel threading as this might cause multiple threads to try to excess the same value at the same time which slows the execution time.
2. As creating and managing multiple threads take time and consumes system resources leading to **overhead issues**, especially on smaller array (as in figure) it might have outweighed the gains of parallel execution.
3. In parallel bubble sort, multiple threads are trying to **modify the same variable** "swapped" simultaneously leading to overhead issues.
4. As data dependencies between threads leads to pipeline stalls and reduced instruction throughput. This might have created some **CPU architectural issues**.

2. Quick Sort

Array Size	Time-Serial	Time-Parallel
1000	0.00184566	0.0035025
10000	0.0304134	0.0421863
20000	0.0691406	0.107098
30000	0.105088	0.139171
40000	0.174498	0.210313
50000	0.251446	0.312272
75000	0.446387	0.523399
100000	0.621064	0.726467



Graph Explanation:

The x-axis represents the size of the array and the y-axis represents the execution time. The input size of the array ranging from 1000 to 100000 with minimum and maximum execution time as 0.00184566 and 0.621064 respectively.

Because of "increasing but diminishing rate of time taken with larger input", the curve generated is curved shaped just like the logarithmic curve. The average-case time complexity of the quick sort is O(nlogn) where n is the number of elements in the array. This complexity is shown by the curved line in the graph.

From the obtained data, it is seen that the parallel implementation is slower than the serial implementation for the smaller size of the array. It is obvious that the execution time will be low for lower size of the array. But with increase in array size, due to the logarithmic nature of the quick sort's time complexity, this will slow down the rate of increase of the execution. This is the expected behavior and is the overhead due to parallelism.

Dividing the task into smaller sub-tasks, parallel execution and then combining all the results as the single result (or simply thread creation, synchronization, and combining results) are the overheads during the parallelization of the task.

The graph representing the reference execution time for the quicksort algorithm would demonstrate the ideal performance achievable by the algorithm under optimal conditions. Similar to both serial and parallel execution, the reference implementation has also the similar rise in the graph but the rate of rise is considerably slower as this implementation is supposed to be highly optimized and efficient. This shows the algorithms best performance achieved by quicksort algorithm for given hardware and implementation constraints as compared to both serial and parallel.

The parallel implementation is slower as compared to serial because of the following reasons:

1. For smaller dataset, different threads access memory concurrently, load data into their respective cache causing **cache contention** and this leads to delay.
2. Quicksort has **limited Instruction Level Parallelism**. Because of this reason inefficient use of CPU resources might occur causing one thread waiting for memory access to complete while other do not get independent instruction.
3. **OS thread management overhead**: The cost of creating and managing threads introduce overhead and might outweigh the potential speedup gained from parallel execution for such small datasets.
4. Parallel algorithm in quicksort do not **scale linearly.** Because of CPU cores and memory shared resources contention and the overhead, adding more threads might not result in increase in speed proportionately.

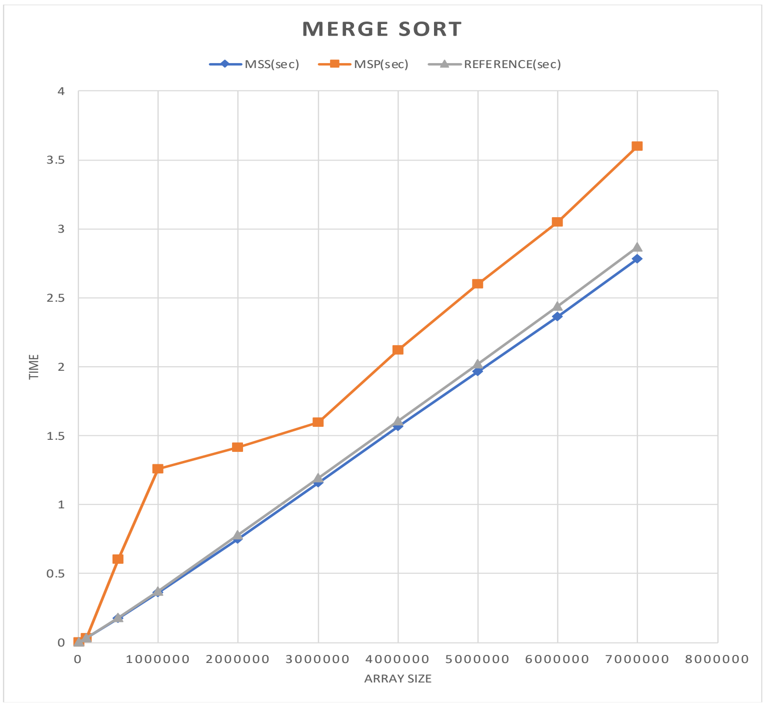
Once the overhead point is reached, then the execution time rise at the slower rate and this can be seen by observing the curve of the graph at array size 20000.

3. Merge Sort

The strategy behind the merge sort is divide and conquer technic. We divide the array into two halves and further divide into subarrays until one element left. Then the sorted subarrays are merged into a single array. Time complexity of merge sort is O(n log n).

The data table below shows that for the given various array sizes we have obtained elapsed time for mss, msp and reference sorting. Based on the results we have noticed that elapsed time for sequential sorting has taken larger time than the parallel and reference sorting.

ARRAY SIZE	MSS(sec)	MSP(sec)	REFERENCE(sec)
10000	0.003466	0.0034767	0.00322
100000	0.033579	0.0333949	0.0332
500000	0.175244	0.600849	0.177801
1000000	0.362794	1.2579	0.372076
2000000	0.750273	1.41429	0.779765
3000000	1.15795	1.59563	1.19501
4000000	1.56515	2.11976	1.60512
5000000	1.96548	2.60233	2.02132
6000000	2.36439	3.04663	2.43936
7000000	2.78007	3.60037	2.86861

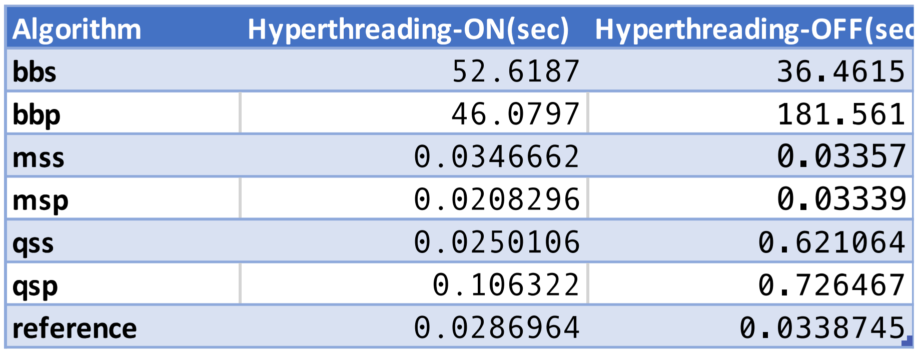


In parallel merge sorting, we have used pragma omp parallel sections and divided the two subarrays into two sections and each section runs on separate threads where they run in parallel so solved simultaneously. Finally merged the two subarrays into a single array. We expected that parallel sorting would be faster than serial sorting, but it was slightly slower. We think this is because using all the CPU's cores can cause problems with thread management, cache, or capability.

We have noticed that serial sorting has executed faster than the parallel and reference sorting. The merge sort line graph is shown below which is between array size and run time for the mss, msp and reference sorting. we have compiled these sorting programs in the home server and noticed the observations as below.

4. Hyperthreading

All the sorting algorithms were run on *jalapeno.cs.siue.edu* with hyperthreading on for 100,000 size array and we received the following time.



Algorithm	Hyperthreading-ON(sec)	Hyperthreading-OFF(sec
bbs	52.6187	36.4615
bbp	46.0797	181.561
mss	0.0346662	0.03357
msp	0.0208296	0.03339
qss	0.0250106	0.621064
qsp	0.106322	0.726467
reference	0.0286964	0.0338745

There are significant improvements in elapsed time in hyperthreading especially for parallel algorithms. The execution time for bbp is very low compared to hyperthreading off.

Thus, hyperthreading allows better utilization of multiple logical cores for parallel execution.