Part 1 - Solution:

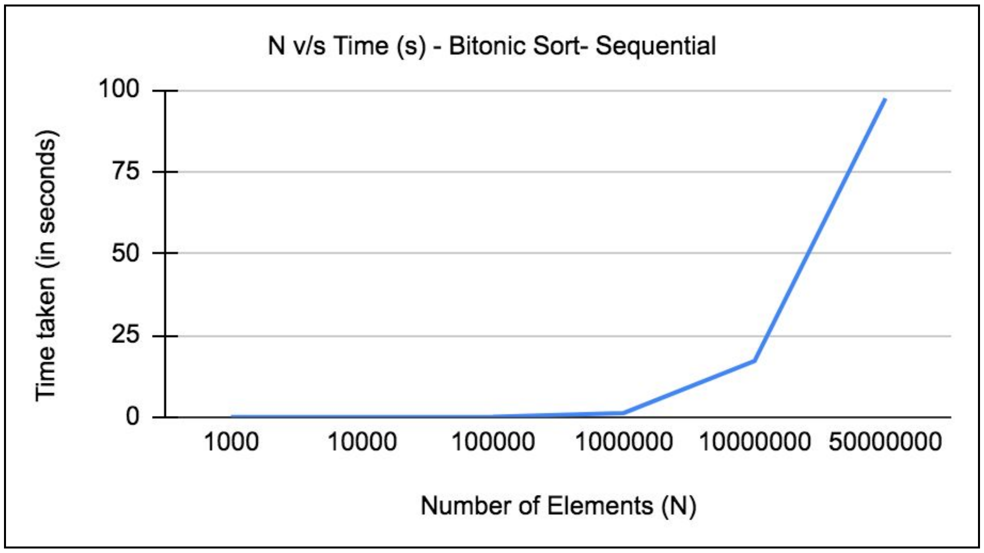
Implemented both Bitonic and Quick Sort algorithms for sequential execution. The number of elements used for sorting are 1000,10000,100000,1000000,10000000, and 50000000 and the machine used for execution of the program is Orbit Systems.

Bitonic Sort:

Below is the table of time taken in seconds to sort different sizes of the list having random numbers in it using the Sequential Method of Bitonic Sort Algorithm.

Number of Elements	Bitonic Sort (in seconds)
1000	0.000333
10000	0.006294
100000	0.091262
1000000	1.21319
10000000	17.1412
50000000	97.4081

Below is the line chart for the above observed performance:



Performance of Bitonic Sort Algorithm - Sequential Method

As seen in the above table and line graph, the time taken to sort the array/list increases as the number of elements increases. There is an exponential increase after the number of elements reaches 1 million i.e. 1000000 taking ~97 seconds for sorting when N is 50000000.

Steps to execute:

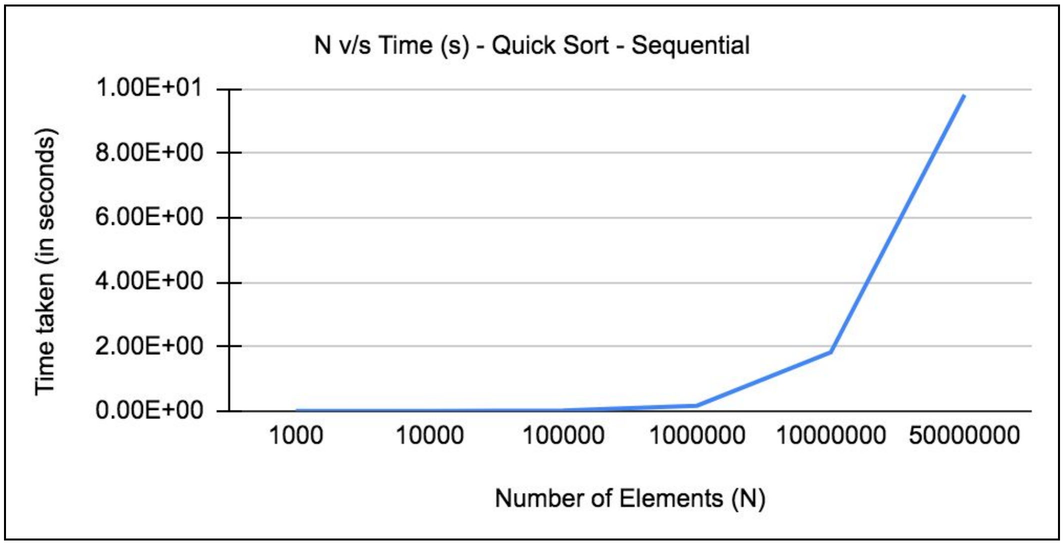
1. Please refer to the bitonic_sequential.cpp code file in Problem 2 folder where the program lies.
2. Use the “make all” command to generate the executable file for this program.
3. Enter ***./bitonic_sequential <Number_of_elements>*** command to execute the program.

Quick Sort (Extra):

Below is the table of time taken in seconds to sort different sizes of the list having random numbers in it using the Sequential Method of Quick Sort Algorithm.

Number of Elements	Quick Sort (in seconds)
1000	0.000094
10000	0.001202
100000	0.013992
1000000	0.162673
10000000	1.81902
50000000	9.81052

Below is the line chart for the above observed performance:



Performance of Quick Sort Algorithm - Sequential Method

As seen in the above table and line graph, the time taken to sort the array/list increases as the number of elements increases. However, this increase is not drastic or sudden. Even for a very high value of N i.e. 50000000, the quick sort algorithm takes only 9.8 seconds to complete. This is very compared to the sequential Bitonic Sort algorithm discussed earlier.

Steps to execute:

1. Please refer to the .cpp code file in Problem 2 folder where the program lies.
2. Use the “make all” command to generate the executable file for this program.
3. Enter ***./quicksort_sequential <Number_of_elements>*** command to execute the program.

Part 2 - Solution:

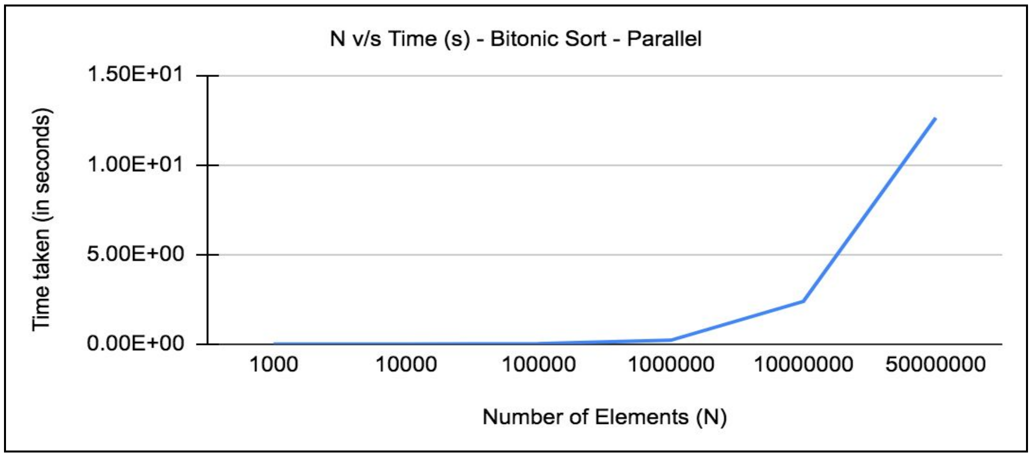
Implemented both Bitonic and Quick Sort algorithms for parallel execution using Pthreads. We have used barriers for synchronization at different stages of the algorithm. Also, we have used pthread_join to join the output of all the threads together for final synchronization and then proceed towards the termination of the program. Using barrier and pthread_join for parallelism and synchronization are very simple parallel programming techniques using pthreads.

Bitonic Sort:

Below is the table of time taken in seconds to sort different sizes of the list having random numbers in it using the Pthreads Parallel Method of Bitonic Sort Algorithm. The number of threads used for program execution is 8 for experimental purposes because the grid node is of 4-cores and thus we decided to move forward with double the number of threads.

Number of Elements	**Bitonic Sort (in seconds)**
1000	0.001935
10000	0.002829
100000	0.020452
1000000	0.213526
10000000	2.380086
50000000	12.62054

Below is the line chart for the above observed performance:



Performance of Bitonic Sort Algorithm - Pthreads Parallel Method

As seen in the above table and line graph, the time taken to sort the array/list increases as the number of elements increases. However, the increase in time is very minute i.e. around 12 seconds when N is 50000000 compared to sequential implementation of bitonic sort where it takes 97 seconds to complete the sorting.

Steps to execute:

1. Please refer to the bitonic_pthreads.c code file in the *Problem 1/Part 2* folder where the program lies.
2. Use the "make all" command to generate the executable file for this program.
3. Enter **./bitonic_pthreads** ***<Number of Threads> <Number_of_elements>*** command to execute the program.
4. Optionally you can use: **./bitonic_pthreads** ***<Number of Threads> <Number_of_elements> -o*** command to view the list as well but we would highly recommend that you enter less number of elements for better display of the numbers.
5. VERY IMPORTANT: Number of threads should be in power of 2 and total elements i.e. N must be evenly divisible by the number of threads. If you don't meet this condition then the list won't be sorted.

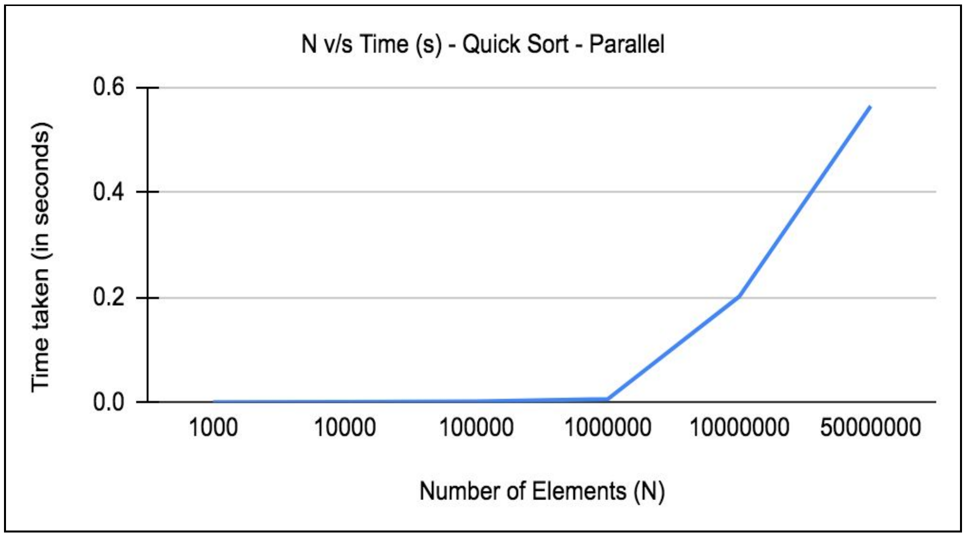
Quick Sort (Extra):

Below is the table of time taken in seconds to sort different sizes of the list having random numbers in it using the Pthreads Parallel Method of Quick Sort Algorithm. The number of threads used for program execution is 8 for experimental purposes because the grid node is of 4-cores and thus we decided to move forward with double the number of threads.

Number of Elements	Quick Sort (in seconds)
1000	0.000149
10000	0.000469
100000	0.001801
1000000	0.00601
10000000	0.20171
50000000	0.56521

As seen in the above table and below line graph, the time taken to sort the array/list increases as the number of elements increases. However, the increase in time is very minute i.e. around in milliseconds compared to sequential implementation of quick sort where it takes 12 seconds to complete the sorting for N = 50000000. Also, it is noticeable that the quicksort parallel algorithm is the fattest among all the other algorithms (Bitonic - Sequential, Parallel and Quicksort - Sequential). Thus, the quick sorting algorithm beats bitonic sort in both sequential and parallel methods.

Below is the line chart for the above observed performance:



Performance of Quick Sort Algorithm - Pthreads Parallel Method

Steps to execute:

1. Please refer to the .cpp code file in *Problem 1/Part 2* folder where the program lies.
2. Use the "make all" command to generate the executable file for this program.
3. Enter ***./quicksort_pthreads <Number_of_elements>*** command to execute the program.

Part 3 - Solution:

Sorting is an algorithm to arrange elements of a list into certain order and make data become easier to access and speed up other operations such as searching and merging. In parallel algorithm design there are many general techniques that can be used across a variety of problem areas. Some of these are variants of standard sequential techniques, while others are new to parallel algorithms.Among them Divide and Conquer approach is the technique used in sequential computation which is naturally adaptable to parallel environment.

A divide-and-conquer algorithm splits the problem to be solved into subproblems that are easier to solve than the original problem, solves the subproblems, and merges the solutions to the subproblems to construct a solution to the original problem. The divide-and-conquer paradigm improves program modularity, and often leads to simple and efficient algorithms. It has therefore proven to be a powerful tool for sequential algorithm designers. Divide-and-conquer plays an even more prominent role in parallel algorithm design. Because the subproblems created in the first step are typically independent, they can be solved in parallel. Often the subproblems are solved recursively and thus the next divide step yields even more subproblems to be solved in parallel. As a consequence, even divide-and-conquer algorithms that were designed for sequential machines typically have some inherent parallelism. Hence Sequential Sorting algorithms that follow divide and conquer strategy have inherent parallelism and hence can be easily parallelised. **So algorithms belonging to these categories are Merge Sort , Quick Sort.**

In computer science there are various **sequential algorithms** that are being used commonly like bubble sort, insertion sort etc. Consider the case of Bubble sort where the It is a popular, but inefficient sorting algorithm. It works by repeatedly swapping adjacent elements that are out of order. But it poses difficulty in parallelisation as each element is dependent on the previous element for making comparison and swap it with the previous element, if required. So such dependencies make certain difficult to parallelise.Consider the case of Bubble sort where we can parallelise it using ALGORITHM MODIFICATION.

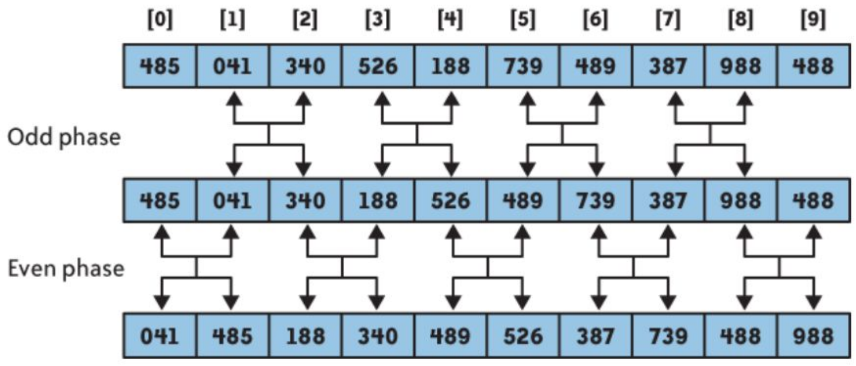
Two instructions can be executed in parallel if they satisfy Bernstein's conditions. Let Pi and Pj are two program segments. For Pi, let Ii be all of the input variables and Oi the output variables, and likewise for j and they satisfy the following conditions.

- $I_j \cap O_i = \varnothing$,
- $I_i \cap O_j = \varnothing$,
- $O_i \cap O_j = \varnothing$,

One of the parallelization ways of sorting algorithms can be modification of the algorithm in a way of those operations in loops and frequently executed instructions satisfy Bernstein's conditions. In this case parallelization can be done through a pipeline mechanism. Every iteration can be performed independently from the previous one.

As conveyed that, in case of bubble sort algorithm, each iteration's operations depend on the previous iteration's result, so there is a need to make some change in order to make possible the instructions' parallel execution. For that purpose we can use Odd-Even transposition method and divide our input list into two imaginary lists – odds and evens.

On one pass through the list, we can compare an odd index and the right adjacent even index element; in the succeeding phase, it compares an even index and the right adjacent odd index element. The odd and even phases are repeated until no exchanges of data are required. This will allow instructions of each loop to be independent.



Pseudo Code:

- *Compare all pairs in the list in parallel*
- *Alternate between odd and even phases*
- *Shared flag, sorted, initialized to true at beginning of each iteration (2 phases), if any processor perform swap, sorted = false*

Program: bubblesort_pthread.cpp

Below is the table of time taken in seconds to sort different sizes of the list having random numbers in it using the Pthreads Parallel Method of Bubble Sort Algorithm. The number of threads used for program execution should be greater than equal to half the size of the input list i.e. Thread Count >= (N+1)/2.

Now, there are a lot of caveats when we increase the number of threads as given in the question. Below are the important ones observed:

1. For bubble sort, since we are doing odd and even based sorting, we always require the number of threads to be more than half of the input array size.
2. Now, there is obviously a limit to the number of threads for a given hardware that can be spawned and we encountered that limit while experimenting various input sizes and thread counts which should be more than half of size.
3. At around thread count = 10000, the system crashes and gives an error code "Segmentation Fault".
4. Thus, for the given hardware i.e. ORBIT system node, we decided to create upto 5000 threads so that it can process upto 10000 elements. After this, any increase in input array size would require more threads to work on but due to hardware limitations the program crashes.
5. The total execution time remains the same for Pthreads implementation for bubble sort algorithm for any input size and any threads. It's approximately 4.19 seconds and the reason might be because of context switching between large numbers of threads while cores are only 4. Also, due to less memory and synchronization between threads while sorting, the time remains the same for all the cases. Below is the table:

Number of Elements	Number of Threads	Time Taken (in seconds)
10	5	4.19788
100	50	4.19816
1000	500	4.19813
2500	1250	4.19791
5000	2500	4.19809

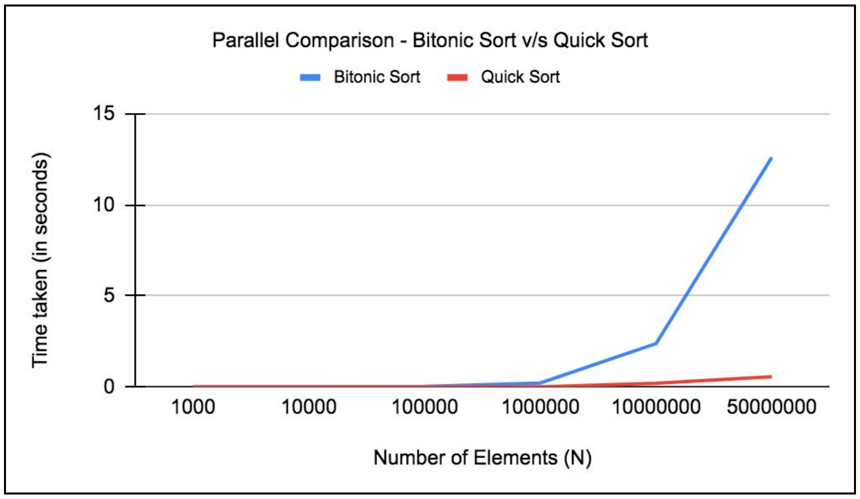
Steps to execute:

1. Please refer to the bubblesort_pthread.cpp code file in the Problem 1/Part 2 folder where the program lies.
2. Use the “make all” command to generate the executable file for this program.
3. Enter ***./bubblesort_pthread <Number of Threads> <Number_of_elements>*** command to execute the program.
4. Optionally you can use: ***./bubblesort_pthread <Number of Threads> <Number_of_elements> -o*** command to view the list as well but we would highly recommend that you enter less number of elements for better display of the numbers.

Part 4 - Solution:

Bitonic Sort v/s Quicksort - Parallel Method:

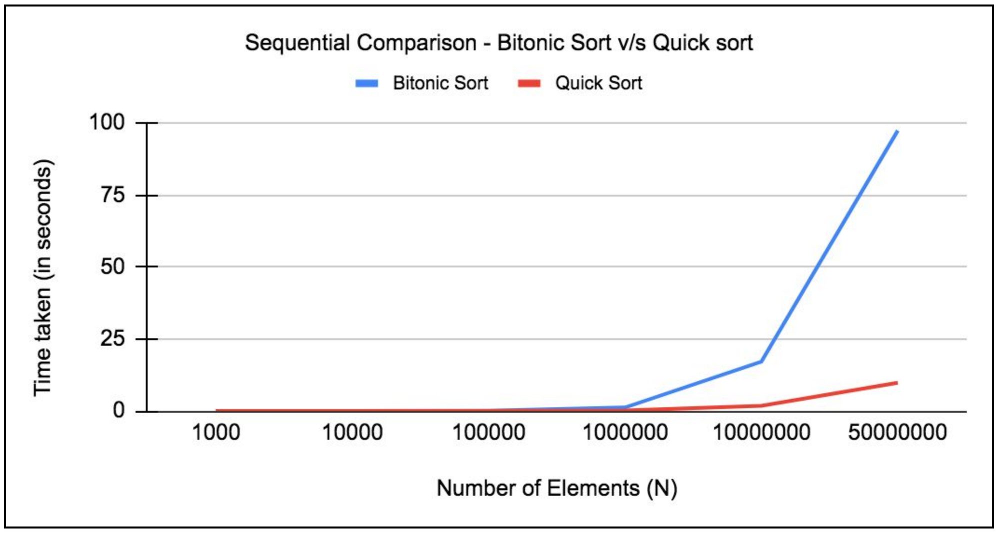
1. As seen in Part 2 of the problem, we discussed Bitonic and Quicksort parallel program execution using pthreads along with the time taken for each of them to sort different sizes input array.
2. Based on the experimental observations, we can visually compare the performance between these two algorithms as shown below.
3. The quicksort algorithm using pthreads outperformed the bitonic sorting algorithm. Upto array of sizes N=1000000, the performance is nearly the same.
4. But then as we keep on increasing the sizes the quick sort performance i.e. time taken to sort elements is very less compared to the bitonic sorting algorithm.



Parallel Performance Comparison - Bitonic Sort v/s Quicksort

Bitonic Sort v/s Quicksort - Sequential Method:

1. As seen in Part 1 of the problem, we discussed Bitonic and Quicksort sequential program execution along with the time taken for each of them to sort different sizes input array.
2. Based on the experimental observations, we can visually compare the performance between these two algorithms as shown below.

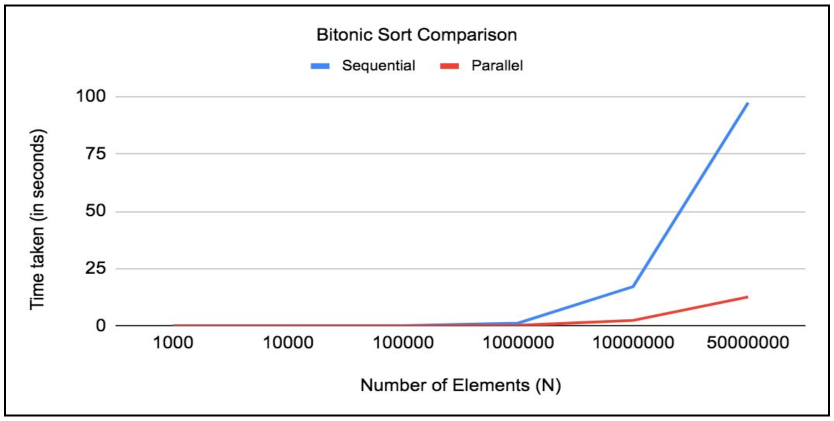


Sequential Performance Comparison - Bitonic Sort v/s Quicksort

3. The sequential quicksort algorithm outperformed the bitonic sorting algorithm. Upto array of sizes N=1000000, the performance is nearly the same.
4. But then as we keep on increasing the sizes the quick sort performance i.e. time taken to sort elements is very less compared to the bitonic sorting algorithm. This was the case even in parallel implementation of both algorithms. Thus, the Quicksort algorithm for both parallel and sequential methods performs better for very large values of N.

Bitonic Sort - Sequential v/s Parallel Method:

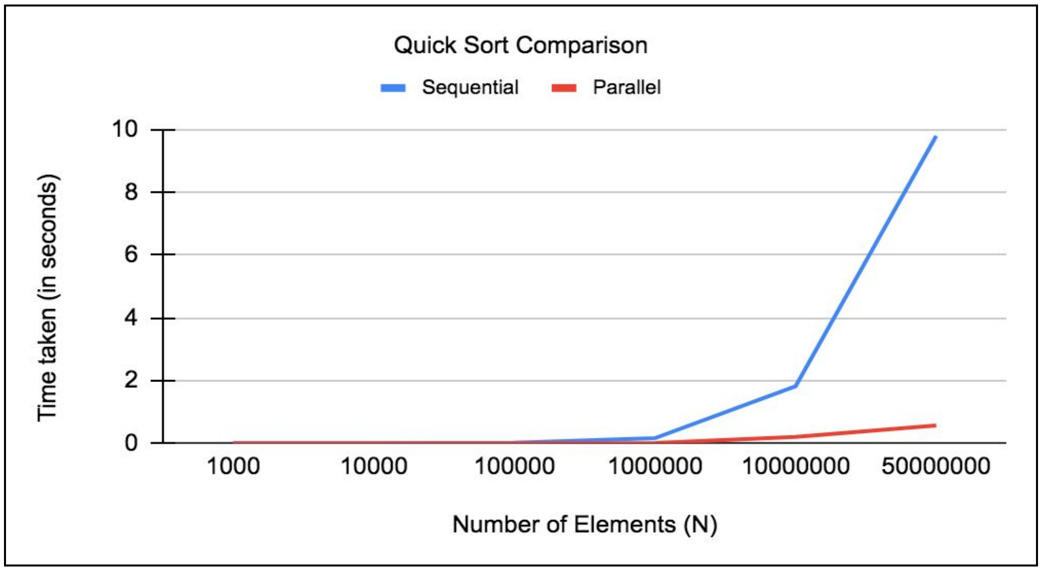
1. As seen in Part 1 and Part 2 of the problem, we discussed Bitonic sequential and parallel program execution along with the time taken for each of them to sort different sizes of input arrays.
2. Based on the experimental observations, we can visually compare the performance between these two algorithms as shown below.
3. The bitonic algorithm using pthreads outperformed the sequential bitonic algorithm. Upto array of sizes N=1000000, the performance is nearly the same.
4. But then as we keep on increasing the sizes the parallel bitonic performance i.e. time taken to sort elements is very less compared to the sequential bitonic sorting algorithm.



Bitonic Sort Performance Comparison - Sequential v/s Parallel

Quicksort - Sequential v/s Parallel Method:

1. As seen in Part 1 and Part 2 of the problem, we discussed Quicksort sequential and parallel program execution along with the time taken for each of them to sort different sizes of input arrays.
2. Based on the experimental observations, we can visually compare the performance between these two algorithms as shown below.
3. The Quicksort algorithm using pthreads outperformed the sequential Quicksort algorithm. Upto array of sizes N=1000000, the performance is nearly the same.
4. But then as we keep on increasing the sizes the parallel Quicksort performance i.e. time taken to sort elements is very very less compared to the sequential Quicksort sorting algorithm. Infact, the time taken is less than all other algorithms that we have implemented.

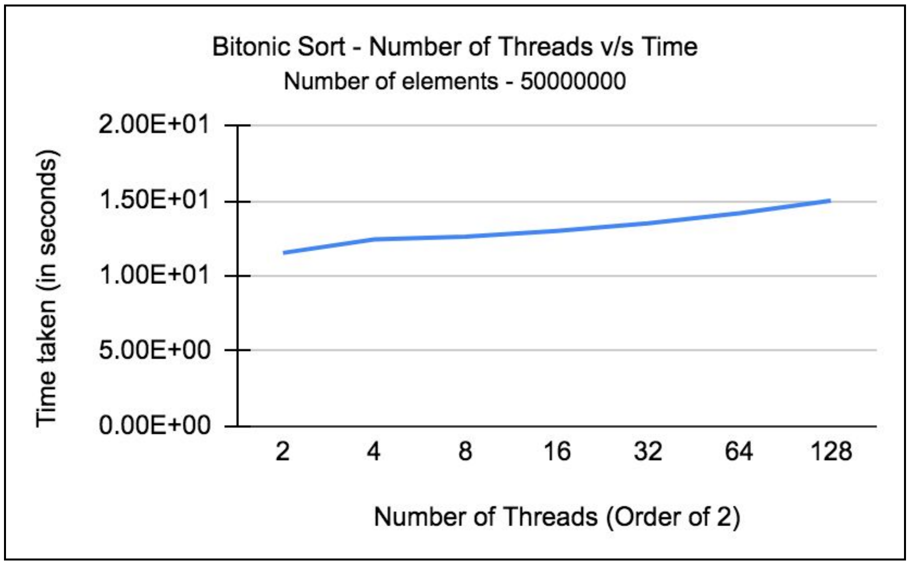


Quicksort Performance Comparison - Sequential v/s Parallel

Bitonic Sort: Number of Threads v/s Time Taken to sort 50000000 elements
Below is the table of time taken for different numbers of threads to sort 50000000 elements.

Number of Threads	Time taken by Bitonic Sort (in seconds)
2	11.52766
4	12.42585
8	12.61535
16	12.98719
32	13.49339
64	14.16353
128	15.00281

Below is the graph of the above table to better visualize the effect of increasing the number of threads in bitonic sort. Since, bitonic sort requires a number of threads in power of 2 and number of elements i.e. 50000000 must be evenly divisible by number of threads, we have increased the number of threads upto 128 only.



From the above graph it can be observed that as we increase the threads beyond a certain level then instead of increase , performance starts decreasing possibly due to increase in the context switching between the threads on the limited number of cores/hardware threads.

Part 5 - Solution:

a) Which of the two algorithms when run sequentially has the best performance in order of complexity?
Solution:

Consider the Bitonic Sort:

In order to form a sorted sequence of length n from two sorted sequences of length n/2, there are log(n) comparator stages required (e.g. the 3 = log(8) comparator stages to form sequence i from d and d'). The number of comparator stages T(n) of the entire sorting network is given by:

$$T(n) = \log(n) + T(n/2)$$

The solution of this recurrence equation is

$$T(n) = \log(n) + \log(n)-1 + \log(n)-2 + \ldots + 1 = \log(n) \cdot (\log(n)+1) / 2$$

Each stage of the sorting network consists of n/2 comparators. On the whole, these are $\Theta(n \cdot \log(n)^2)$ comparators.

The above complexity for the bitonic sort is for the worst, average and the best case scenario.

Consider the Quick Sort:

The complexity depends on the ordering of elements:

a. In case the elements to be sorted are in random orders then complexity is around:

Best Case: O(n logn)

Average Case: O(n logn)

b. In case the elements are ordered already then the quick sort sequential algorithm will take the worst case time of O(N^2)

Worst Case: O(n^2)

So from above discussion assuming that elements to be sorted are in random order then in that case the quick sort seems to be better.

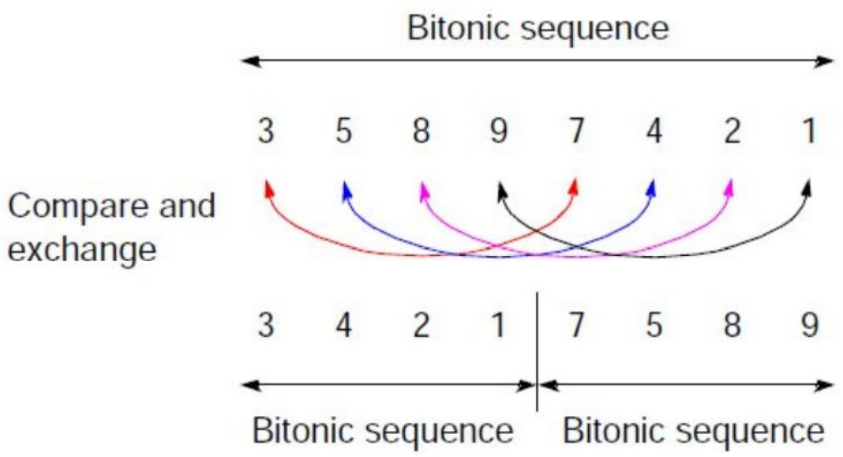
b) Which of the two algorithms when run in parallel has the best performance in order of complexity? Why, how?
Solution:

Bitonic Sort

Bitonic sort is one of the fastest sorting networks. A sorting network is a special kind of sorting algorithm, where the sequence of comparisons is not data-dependent. This makes sorting networks suitable for implementation in hardware or in parallel processor arrays.It falls into the group of sorting networks, which means that the sequence and direction of comparisons is determined in advance and is independent of input sequence.

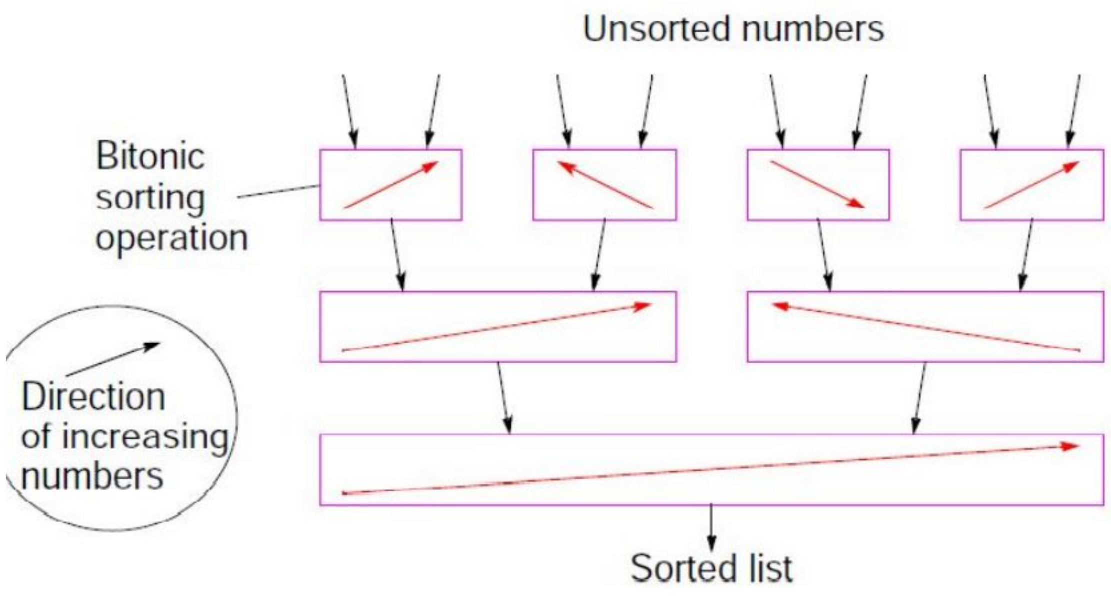
The basis of bitonic sort is the bitonic sequence, a list having specific properties that are exploited by the sorting algorithm. A sequence is considered bitonic if it contains two sequences, one increasing and one decreasing, such that: a1 < a2 < . . . < ai−1 < ai > ai+1 > ai+2 > . . . > an for some value i (0 ≤ i ≤ n). A sequence is also considered bitonic if the property is attained by shifting the numbers cyclically (left or right).

A special characteristic of bitonic sequences is that if we perform a compare-and-exchange operation with elements ai and ai+n/2 for all i (0 ≤ i ≤ n/2) in a sequence of size n, we obtain two bitonic sequences in which all the values in one sequence are smaller than the values of the other.



In addition to both sequences being bitonic sequences, all values in the left sequence are less than all the values in the right sequence. Hence, if we apply recursively these compare-and-exchange operations to a given bitonic sequence, we will get a sorted sequence.

To sort an unsorted sequence, we first transform it in a bitonic sequence. Starting from adjacent pairs of values of the given unsorted sequence, bitonic sequences are created and then recursively merged into (twice the size) larger bitonic sequences. At the end, a single bitonic sequence is obtained, which is then sorted (as described before) into the final increasing sequence.



The bitonic sort algorithm can be split into k phases (with n = 2^k):

Phase 1: convert adjacent pair of values into increasing/decreasing sequences, i.e., into 4-bit bitonic sequences(as shown in the initial phase above)

Phases 2 to k-1: sort each m-bit bitonic sequence into increasing/decreasing sequences and merge adjacent sequences into a 2m-bit bitonic sequence, until reaching a n-bit bitonic sequence

Phase k: sort the n-bit bitonic sequence

(It takes tune of k steps at each phase)

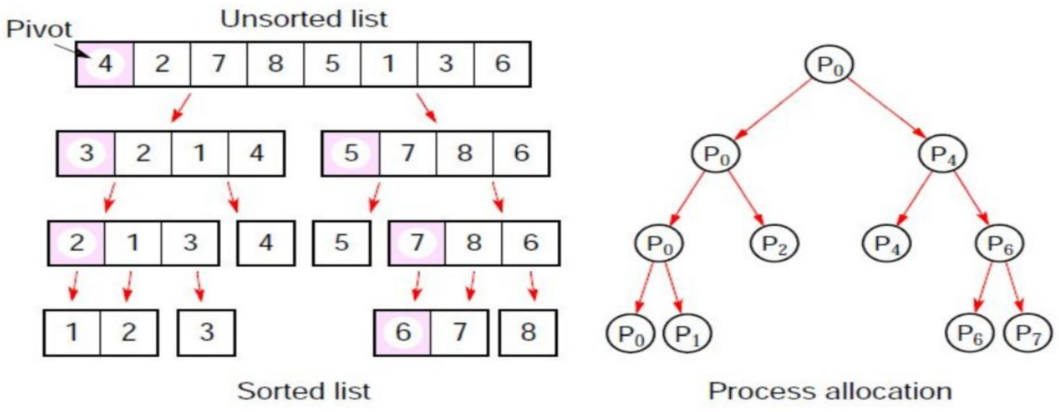
Given an unsorted list of size n = 2 k , there are k phases, with each phase i taking i parallel steps. Therefore, in total it takes X k i=1 i = k(k + 1) 2 = log(n)(log(n) + 1) 2 parallel steps to obtain the final sorted list in a parallel implementation, which corresponds to a time complexity of O(log(n) ^ 2)

In simple terms if we see that at Log N total steps , we are doing around Log N steps for conversion into two ascending and descending (bitonic sequences) and at last only ascending or descending sequence based on the problem in hand. So Log N * Log N

So in case of parallel algorithm , we can say that complexity almost equivalent to Log(n) ^ 2

Quick Sort

In quick sort, the idea is to take advantage of the tree structure of the algorithm to assign work to processes. As in each step we tend to place the pivot element in the right place and then partition the lists. In a tree like structure as shown we create multiple parallel processes.



But Allocating processes in a tree structure leads to two fundamental problems:

In general, the partition tree is not perfectly balanced (selecting good pivot candidates is crucial for efficiency)

The process of assigning work to processes seriously limits the efficient usage of the available processes (the initial partition only involves one process, then the second partition involves two processes, then four processes, and so on)

If we ignore communication time and consider that pivot selection is ideal, i.e., creating sublists of equal size, then it takes log X (n) i=0 n 2 i ≈ 2n steps to obtain the final sorted list in a parallel implementation, which corresponds to a time complexity of O(n). The worst-case pivot selection degenerates to a time complexity of O(n 2).

<u>So from the above detailed description it can be seen that Quick sort is supposed to perform better than Bitonic sort in view of the time complexity assuming randomised input sequence for sorting.</u>

c) Do you results agree with the experimental results and analysis conducted in Part1-Part4? If not, or if not entirely, could you justify your answer with a few distinct arguments?
Solution:

Number of Elements	Sequential Implementation		Parallel Implementation	
	Bitonic Sort	Quick Sort	Bitonic Sort	Quick Sort
1000	0.000333	9.40E-05	1.94E-03	0.000149
10000	0.006294	0.001202	2.83E-03	0.000469
100000	0.091262	0.013992	2.05E-02	0.001801

1000000	1.21319	0.162673	2.14E-01	0.00601
10000000	17.1412	1.81902	2.38E+00	0.20171
50000000	97.4081	9.81052	1.26E+01	0.56521

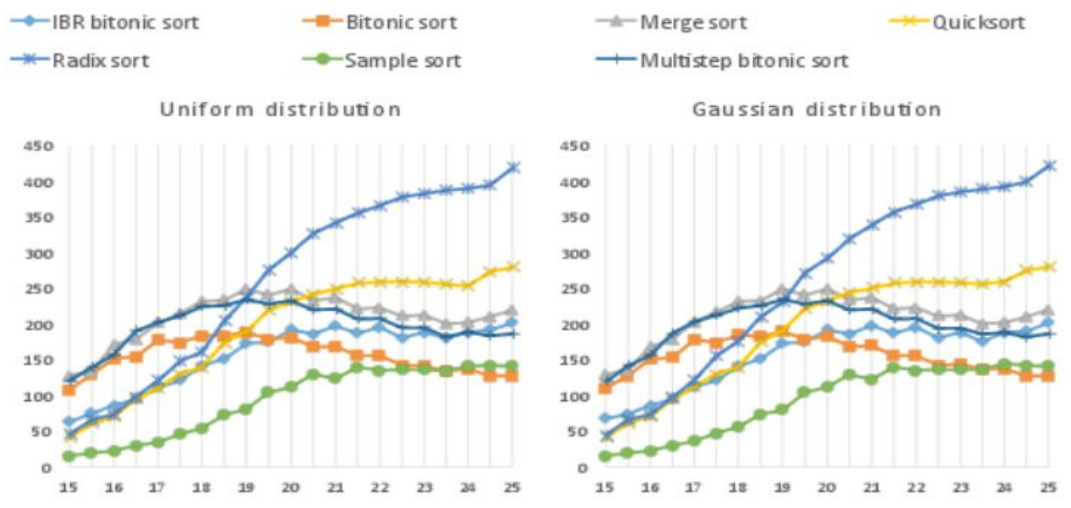
The above are the results obtained by varying the number of elements to the sequential and parallel implementation of both Bitonic Sort and the Parallel Quick-Sort.

As explained in the theoretical analysis, that in case of randomised inputs, in case of sequential and parallel sorting Quicksort fares well and that is the result we have obtained as shown in the above table.

Using Reference: TechnicalReport_v2 (arxiv.org)
Comparison of parallel sorting algorithms Darko Božidar and Tomaž Dobravec Faculty of Computer and Information Science, University of Ljubljana, Slovenia
Environment used:
For parallel computing they used the GPU GeForce GTX670 with 2GB of memory.

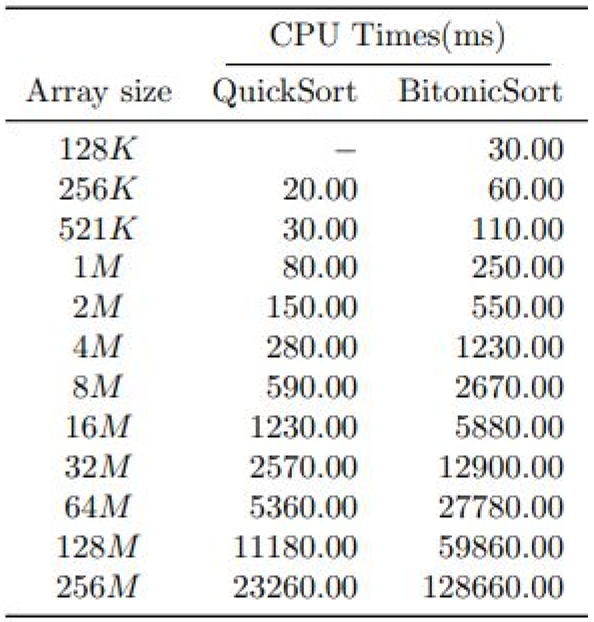
It can be seen their comparison of the Various Parallel sorting algorithms:
That parallel Quicksort fared well in comparison to the bitonic sort in sufficient data quantity.
Note M/s is the rate of sorting in a second.

Parallel, 32 bit, keys only
The sort rate of parallel algorithms when sorting sequences of 32-bit keys.
X-axis: Binary logarithm of the sequence length, Y-axis: sort rate in M/s.



Reference:
1506.01446.pdf (arxiv.org)
Technical Paper : The implementation and optimization of Bitonic sort algorithm Qi Mu. Liqing Cui. Yufei Song
Concluded:
Quick sort is much faster than bitonic sort on the CPU,this is due to the differences in time complexity.

Table in the paper:



Array size	CPU Times(ms) QuickSort	CPU Times(ms) BitonicSort
$128K$	–	30.00
$256K$	20.00	60.00
$521K$	30.00	110.00
$1M$	80.00	250.00
$2M$	150.00	550.00
$4M$	280.00	1230.00
$8M$	590.00	2670.00
$16M$	1230.00	5880.00
$32M$	2570.00	12900.00
$64M$	5360.00	27780.00
$128M$	11180.00	59860.00
$256M$	23260.00	128660.00