Report on Sorting an array using multi-threading in C:

Using Method 1:

1.After creating a file, starting writing the code with necessary header files. Especially, “pthread.h” for thread functions and “sys/time.h” for ‘gettimeofday’ function.

2.Create a struct containing an array (long datatype) and int type value, for passing parameters during the creation of threads.

3.Now declare necessary global variables and functions.

4.Now, in the main function, using a file pointer, open the file “inp.txt” in read mode and using fscanf command, store the values in the file into n, p (int datatypes).

5.Here, 2^n denotes the size of the array to be sorted and 2^p denotes number of threads to be created.

Therefore, number of segments will be 2^n and each segment size is 2^(n-p).

6.After writing necessary error conditions, dynamically allocate the memory for an array pointed by struct pointer.

7.Using rand function, store the values in the above array.

8.Then print the above unsorted array into “output.txt” which is opened in write mode using a file pointer.

9.with the help of gettimeofday function, store the time in “start”(declared using struct timeval).

10.Then using ‘pthread_create’ function, create required number of threads with ‘sort’ function and pass the arguments for the function as struct pointer.

11. Here, created each thread i.e, 2^n threads will sort 2^n segments correspondingly using ‘sort’ function.

12.In the sort function, with the help of quick sort function, each segment will be sorted.

13.Then, all threads are joined using ‘pthread_join’ function.

14.Now, after each segment is sorted, using main thread, each sorted segment will be merged and results in sorted array.

That is, initially, main thread merges 1st and 2nd segment as 1st segment using the function ‘Merge’.

Here, Merge is a function that merges two sorted segments into one whole sorted segment.

Then, again after series of merges i.e, 1st,2nd; 1st, 3rd; ……1st ,2^n; by main thread, a final sorted array emerges.

15.with the help of gettimeofday function, store the time in “end” (declared using struct timeval).

16.Now, print the sorted array into “output.txt” which is opened in write mode using a file pointer.

17.Then, using start and end, calculate and print the time into “output.txt”.

Using Method 2:

1.Similar to Method 1, after creating a file, starting writing the code with necessary header files. Especially, “pthread.h” for thread functions and “sys/time.h” for ‘gettimeofday’ function.

2.Create a struct containing an array (long datatype) and int type value, for passing parameters during the creation of threads.

3.Now declare necessary global variables and functions.

4.Now, in the main function, using a file pointer, open the file “inp.txt” in read mode and using fscanf command, store the values in the file into n, p (int datatypes).

5.Here, 2^n denotes the size of the array to be sorted and 2^p denotes number of threads to be created.

Therefore, number of segments will be 2^n and each segment size is 2^(n-p).

6.After writing necessary error conditions, dynamically allocate the memory for an array pointed by struct pointer.

7.Using rand function, store the values in the above array.

8.Then print the above unsorted array into “output.txt” which is opened in write mode using a file pointer.

9.with the help of gettimeofday function, store the time in “start” (declared using struct timeval).

10.Then using ‘pthread_create’ function, create required number of threads with ‘sort’ function and pass the arguments for the function as struct pointer.

11. Here, created each thread i.e, 2^n threads will sort 2^n segments correspondingly using ‘sort’ function.

12.In the sort function, with the help of quick sort function, each segment will be sorted.

13.Then, all threads are joined using ‘pthread_join’ function.

14.Now, the main thread repeatedly creates threads which are half the number of segments each time.

That is, since initially there are 2^n segments, so the main thread creates 2^n-1 threads (half the number of segments), using ‘pthread_create’ with ‘merge’ as their function.

In the merge function, with the help of Merge function, any two segments in sorted order are merged into a single segment sorted one.

Therefore, each thread will merge two segments i.e, 1st ,2nd will be merged by a thread; 3rd ,4th will be merged by a thread and so on.

After that, 2^n-1 segments will be present. Then again, main thread creates 2^n-2 threads (half of segments) and merges into 2^n-2 segments and so on.

This process, continues till the number of segments present is 1, which implies whole array is sorted.

15.with the help of gettimeofday function, store the time in “end”(declared using struct timeval).

16.Now, print the sorted array into “output.txt” which is opened in write mode using a file pointer.

17.Then, using start and end, calculate and print the time into “output.txt”.

Performance of their algorithms:

1.Basing on the algorithm, Method2 is faster than Method 1.

2.Because in Method2, after each segment sorting is done, the main thread creates more and more threads to merge the segments in parallel manner which obviously takes less time when compared to method1.

3. That is, in method 1, after each segment sorting is done, the whole segment merging part is done by main thread itself one merge after another which takes a lot of time while compared to method 2 in which merging is done parallelly.

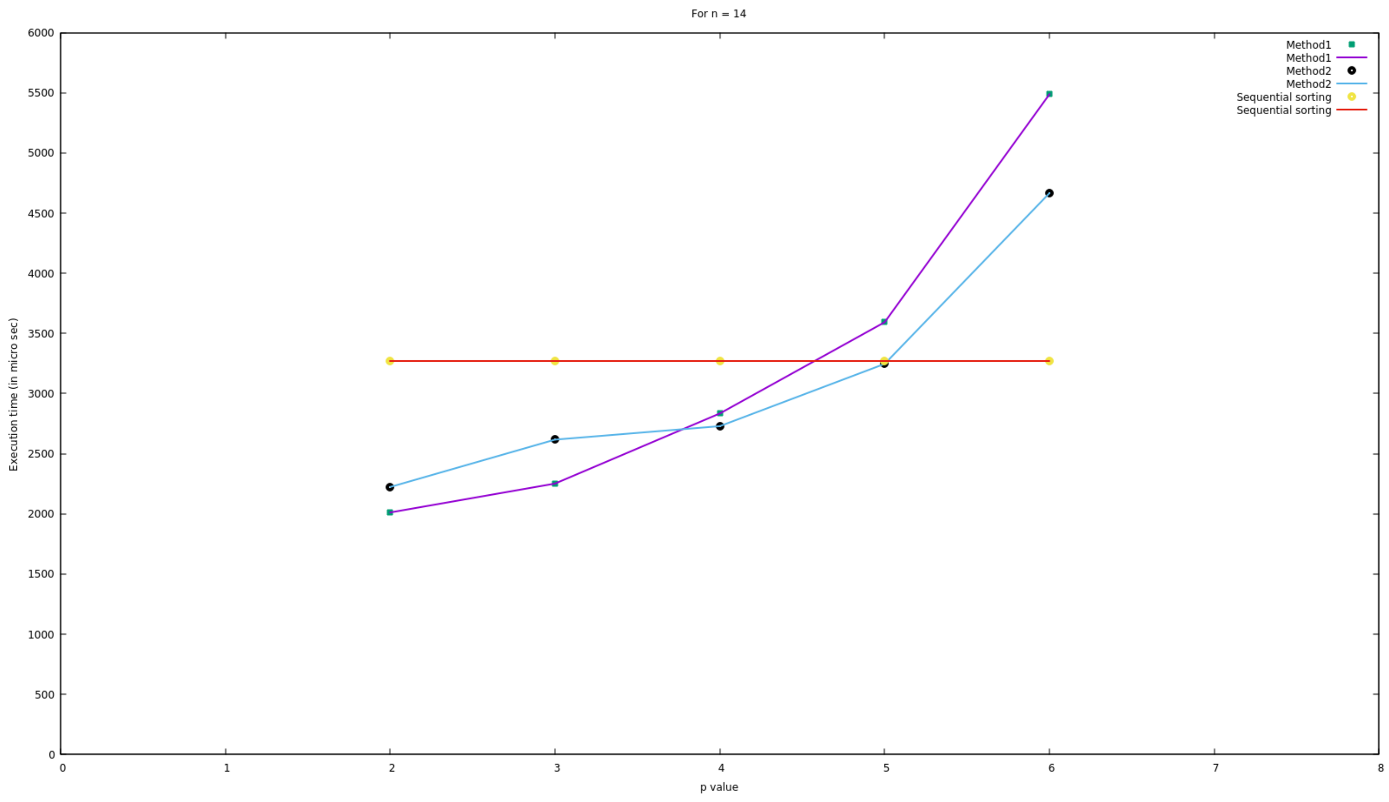
Analysis on their outputs:

1.For lower values of n while p is fixed, method1 is faster and efficient than method2, but for higher values of n while p is fixed, method 2 becomes more faster than method1.

2.similarly, for lower values of p while n is fixed, method1 is faster and efficient than method2, but for higher values of p while n is fixed, method 2 becomes more faster than method1.

3.This may occur because as method 2 creates more and more number of threads repeatedly, so, it might be taking more time, which can be clearly observed for lower values of n and p as it takes very less execution time.

Graph1:



Firstly, this graph is created based on the inputs n,p given to the method1, method2 and sequential sorting program(here, quick sort) and also on the output i.e, time taken for the execution of the corresponding program.

In this, the input n is fixed constant to ‘14’(i.e, size of array is 2^14).

And in the graph, on the x-axis: p is taken as parameter varying from 2 to 6(i.e, 2,3,4,5,6).

And on the y-axis: execution time is noted in micro seconds.

Analysis:

Coming to sequential sorting, since the array size is fixed, therefore, clearly the time taken for sorting will be same for any number of threads (i.e, p value).

Hence, it is a straight line parallel to x-axis, always having constant execution time for a given size of array.

Coming to method1 and method 2, clearly, as p increases, the execution time also goes on increasing.

Now, comparing method1 and method2, for lower values of p (i.e, p<4) method 1 is faster than method2.

But for p>=4, method 2 has become more faster than method1.

Also, As the p value increases, clearly, the slope of method1 curve is increasing more compared to that slope of method2 curve.

Now, by comparing method1 & method 2 with sequential sorting, clearly for lower values of p (i.e,p<5), method1&2 are relatively faster than sequential sorting, while, as p goes on increasing, method1&2 are taking more time for execution compared to sequential sorting.

Hence,

Slope:

Sequential sorting – always constant

Method1 – always positive

Method2 – always positive

(As p increases, slope of Method1 curve > slope of Method2 curve)

Curve nature:

Sequential sorting – constant

Method1 – increasing

Method2 – increasing

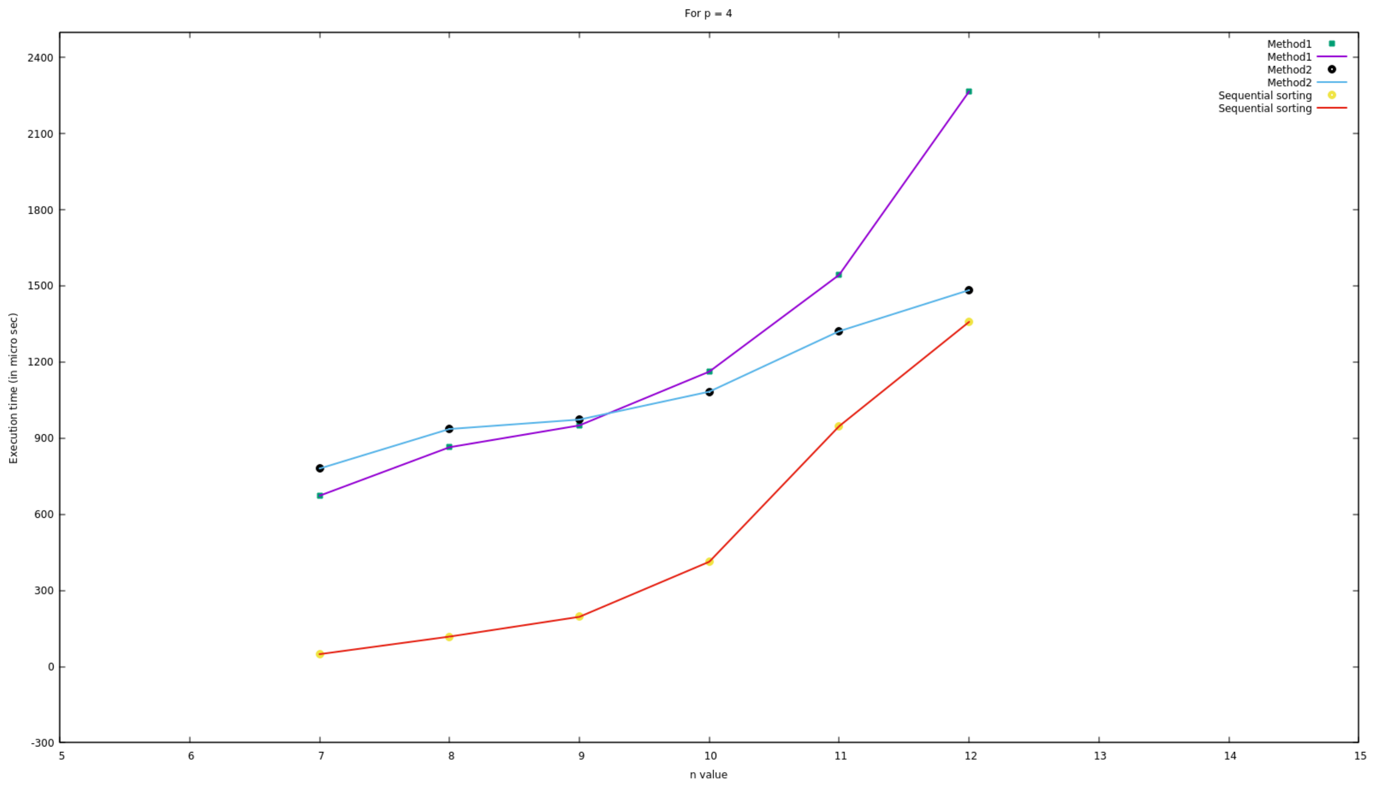
For low values of p:

Method1 > Method2 > sequential sorting (Performance comparison)

For higher values of p:

Sequential sorting > Method2 > Method1 (performance comparison)

Graph2:



Firstly, this graph is created based on the inputs n,p given to the method1, method2 and sequential sorting program(here, quick sort) and also on the output i.e, time taken for the execution of the corresponding program.

In this, the input p is fixed constant to ‘4’(i.e, number of threads is 2^4).

And in the graph, on the x-axis: n is taken as parameter varying from 7 to 12(i.e, 7,8,9,10,11,12).

And on the y-axis: execution time is noted in micro seconds.

Analysis:

Coming to method1, method 2 and sequential sorting, clearly, as n increases, the execution time also goes on increasing.

Now, comparing method1 and method2, for lower values of n (i.e, n<10) method 1 is faster than method2.

But for n>=10, method 2 has become more faster than method1.

Also, As the n value increases, clearly, the slope of method1 curve is increasing more compared to that slope of method2 curve.

Now, comparing method1 & method 2 with sequential sorting, clearly for all values of n (i.e, n<=12), sequential sorting is much efficient and faster than that of method1&2.

But on further observation (as we have seen for n=14,p=4 in graph 1), for much more higher values of n, sequential sorting becomes less efficient and slower compared to that of method1&2.

Hence,

Slope:

Sequential sorting – always positive

Method1 – always positive

Method2 – always positive

(as n increases, slope of Method1 > slope of method2)

(for much higher values of n, slope of sequential sorting > slope of method1 > slope of method2)

Curve nature:

Sequential sorting – increasing

Method1 – increasing

Method2 – increasing

For low values of n:

Sequential sorting > Method1 > Method2 (Performance comparison)

For higher values of n:

Sequential sorting > Method2 > Method1 (performance comparison)

For much higher values of n (as we have seen for n=14, p=4 in graph1):

Method2 > Method1 > sequential sorting (performance comparison)

Conclusion:

Therefore, for a fixed higher n value and varying p values, as p goes on increasing method2 becomes more and more efficient and faster than method1 and sequential sorting.

Also, for a fixed p value and varying n values, as n goes on increasing, method 2 becomes more faster than method1 and on further observation, we can say that, for much higher values of n, method 2 becomes even becomes more efficient than sequential sorting as well.