Parallel Sorting Using Multi-Threading

Ananya Mantravadi
CS19B1004

Method 1:

This program takes input the value of n and p from a text file, inp.txt. It first creates an array of size 2^n using rand() inbuilt function which can safely generate random long values without causing an overflow.

```
//function to initialise an array with random long values
void arr_init(long arr[],int size)
{
  for(int i=0;i<size;i++){
    arr[i] = (long)rand()*(long)rand()/10;
  }
}
```

This initial array before is then printed out as output. We then define the number of segments to be equal to the number of threads, i.e, 2^p. Then the size of each segment would be 2^(n-p).

```
typedef struct seg {
long *arr;
int start;
int end;
} seg;
```

We define the above structure that will keep track of the starting and ending points of each segment of the array while having access to the array, with which each segment of the array be sorted separately by each thread. We declare an array of structures, `seg* segments,` and initialize it with appropriate values.

I chose the Selection Sort algorithm for sorting each of the segments, which is of O(n^2):

```
//function to sort each segment using threads
void *seg_sort(void* arg)
{
  int i,j;
  long temp;
  seg* a = (seg*) arg;
  for(i=a->start; i<a->end; i++){
         int index = indexOfMinimum(a->arr,i,a->end);
         temp = a->arr[i];
       a->arr[i] = a->arr[index];
       a->arr[index] = temp;
       }
     pthread_exit(NULL);
}
```

Next, we create threads and assign each thread to a segment to perform `seg_sort`, which is the sequential algorithm.

```
//we create 2^p threads
pthread_t* threads;
threads = (pthread_t*)malloc(no_threads*sizeof(pthread_t));

//creating threads and passing the data for sorting of one segment at a
time
for(i=0; i<no_threads; i++){
      pthread_create(&threads[i], NULL, seg_sort, (void*)(&segments[i]));
}

//wait for the threads to terminate
for(i=0; i<no_threads; i++){
      pthread_join(threads[i], NULL);
}
```

At this stage, we have sorted segments of the array we started with. We then use the main thread to perform the merging of the segments, which is of O(sum of size of each segment). We iterate through consecutive segments, compare the segment elements and store the smaller values:

```
// function to merge the sorted subarrays
void merge_sort(long arr[],long mergesort[],int start,int end)
{
  int i=0,j=start,k=0;
       while (i<start && j<end){
             if(arr[i] < arr[j]){
                 mergesort[k] = arr[i];
                 i++;
             }
             else{
                 mergesort[k] = arr[j];
                 j++;
             }
             k++;
       }

       if (i>=start){
             while (j<end){
                 mergesort[k] = arr[j];
                 j++;
                 k++;
             }
       }

       if (j>=end){
             while (i<start){
                 mergesort[k] = arr[i];
                 i++;
                 k++;
             }
       }
}
```

This will finish sorting the entire array. The time taken for computation in microseconds has been calculated using clock():

```
// note the time when computation begins
clock_t begin = clock();
// note the time when computation ends
clock_t end = clock();
clock_t clockTimeSpent = end - begin;
unsigned long long time_spent = 1000000*((double)clockTimeSpent/(unsigned
long long)CLOCKS_PER_SEC);
```

Method 2:

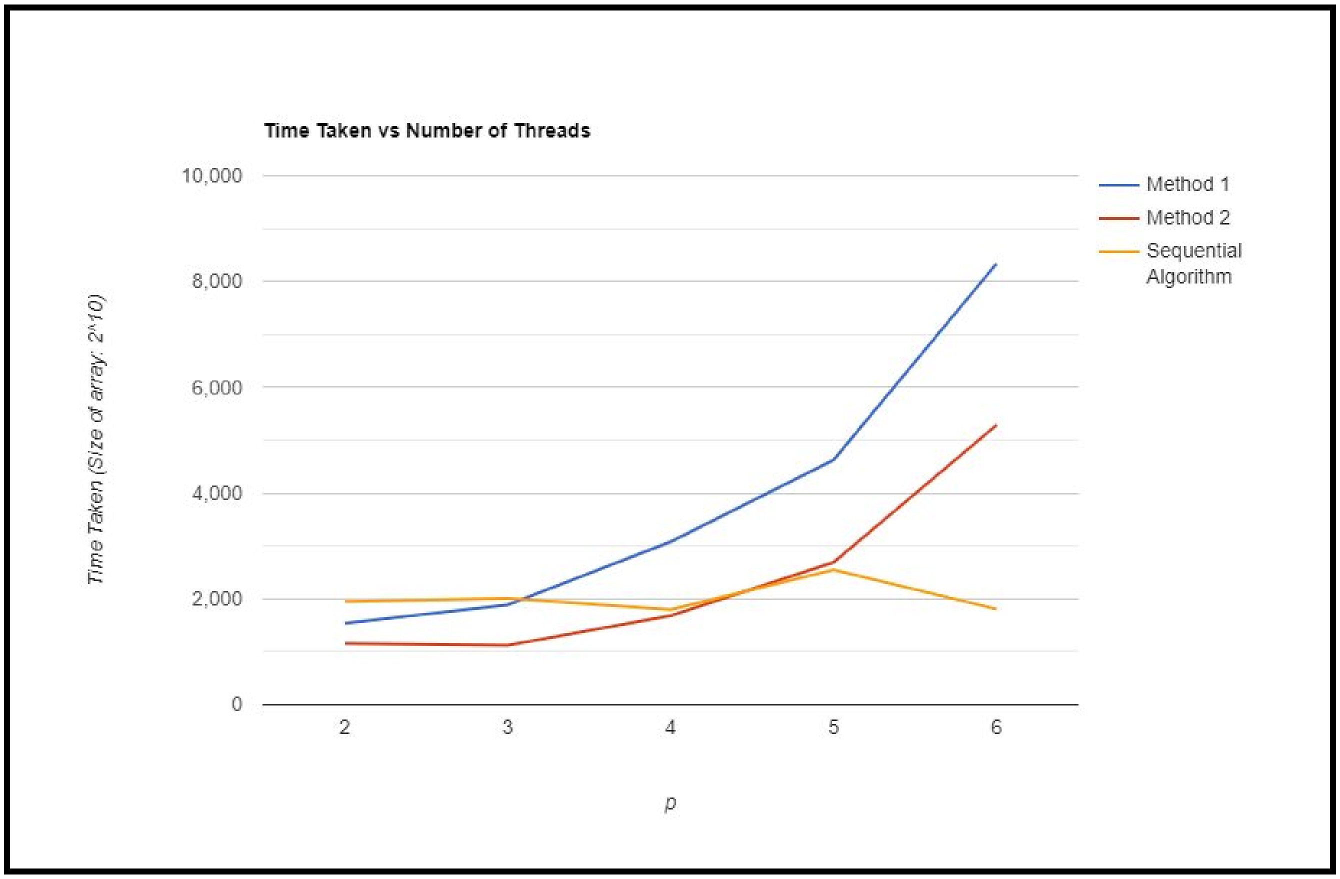
The majority of the initial part of this program is similar to Method 1, till the part where individual segments are sorted by 2^p threads created. Then we run a while loop till all the segments are merged, i.e, till we are left with only one sorted segment which is the final sorted array.

```
int no_segments = no_threads/2;
int k = 2;

while(no_segments>=1){
//create half the number of threads as the number of segments
 pthread_t* threads;
 threads = (pthread_t*)malloc((no_segments)*sizeof(pthread_t));

//assigning threads the right segments to sort and merge
 for(i=0; i<no_segments; i++){
    pthread_create(&threads[i], NULL, merge_sort,(void*)(&segments[k*i]));
 }

//wait for the threads to terminate
 for(i=0; i<no_segments; i++){
       pthread_join(threads[i], NULL);
 }
//after the threads merge the segments, the number of segments becomes
half
 no_segments = no_segments/2;
 k = k*2;
}
```

Analysis of the output

Graph 1

Time taken by Method 1, 2, and sequential algorithm for different values of the number of threads (size of array = constant = 2^10)



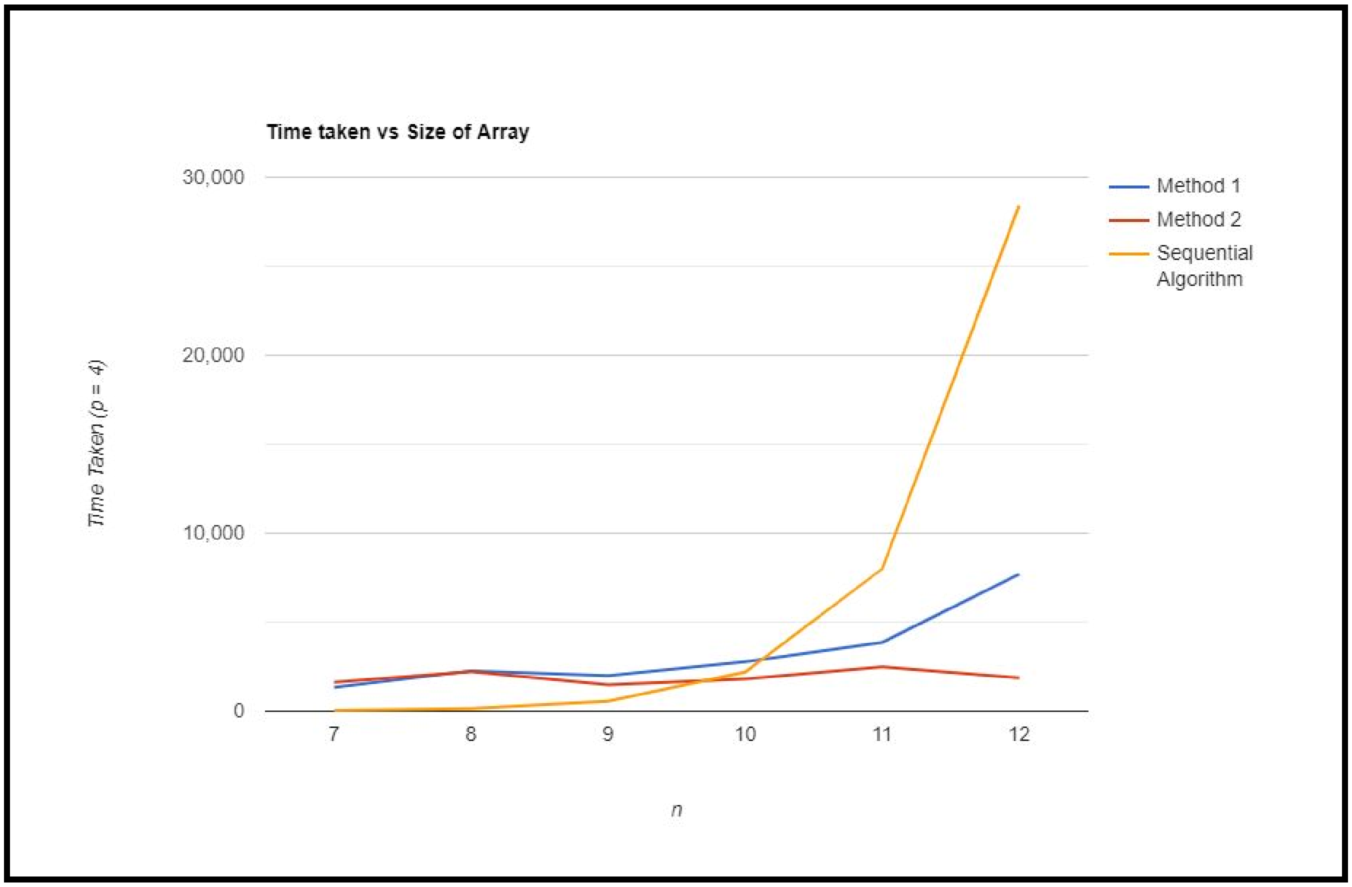
Firstly, we observe that the sequential algorithm (selection sort in this case) gives almost the same values as it depends only on the size of the array which is now constant. Hence, the curve representing this algorithm is a line parallel to X-Axis.

Method 2 will always be faster than method 1 (for larger values of n). This is because Method 1 does not take significant advantage of parallelism as the main thread still has the major task of merging all the segments in a sorted manner. Merging after segment sorting is executed by just the main thread in method 1 while merging in method 2 is executed by creating new threads in each phase. Since there is a sharing of workload, the overhead of maintaining more threads is dominated by faster computation when the size of the array becomes larger.

When the number of threads created in both method 1 and method 2 is low in number, (2, 3), the time taken to sort with multi-threading is lesser than the simple sorting algorithm. The threads are able to share computation optimally and help in speeding up the process. But when the number of threads increases, these methods take more time to sort the same array size - there is an overhead in the creation of threads and assigning them to each segment for sorting. There are more threads that sort smaller segments and this leads to a slower overall computation as compared to the sequential algorithm.

Graph 2

Time taken by Method 1, 2, and sequential algorithm for different sizes of array (number of threads p = constant = 4)



For smaller sizes of the array (2^7, 2^8, 2^9), we observe that sequential algorithm is faster since there is no overhead of creation, assignment of tasks, and termination of threads to deal with which is more expensive. But as the size of the array becomes significantly larger, simple sorting proves slower. Since it is of O(n^2), it quickly grows with n. Multi-threading divides the computation among a constant number of threads and thus saves time. The overhead of maintaining threads is overcome by computation of larger arrays through these threads.

Just like we have observed in graph 1, method 2 will always be faster than method 1 for larger arrays, as there are more threads at each phase for merging the sorted segments as opposed to just the main thread taking up the whole merging process.

Conclusion:
From the above two graphs with their analysis, we can see that **Method 2** will be the most efficient for sorting when the size of array becomes large (> 2^10). It is also important to have optimal number of threads involved to avoid the overhead of maintaining these threads.