Operating Systems-II
Report-Asgn-1

The programs written will provide the amount of time necessary using gettimeofday() function to sort an array using quick sort in 3 different methods. The methods are:

- Method-I: Main Thread merging all threads after sorting.
- Method-II: Creating separate threads for merging after sorting.
- Method-III (Sequential): Just normal quicksort.

Method-I:

1. Given the inputs n,p. An array of size 2^n random values are generated. By using pthread_create() method we are creating 2^p *threads* and sorting the respective segments using the runner function in pthread_create().

2. The thread creation part looks similar to the below screenshot.

```
void* sort_workers(void* arg) {
    int th_part= *((int*)arg);
    int val = pow(2,n-p);
    int l= th_part*val,h=(th_part+1)*val-1;
    qsort((void*)(arr+l),h-l+1,sizeof(int),cmp);
}
```

```
for (int i = 0; i < num_workers ; i++) {
    int* th_num =malloc(sizeof(int));
    *th_num = i;
    pthread_create(&workers[i], &attr, sort_workers,
                   (void*)th_num);
}
```

3. After that, the main thread waits for all the other threads to complete their sorting process of respective segments, and then it starts merging one by one. Similarly time taken Just to sort the array using quicksort is also done in this file.

Method-II:

1. Similar to Method-I we create an array of size 2^n and assign random values to them. In this method instead of the main thread merging all the threads we again create threads to merge different segments of the array.
2. The algorithm proceeds such that in each round, half of the threads would be reduced and the remaining half will merge their respective segments.

3. For the ease of programming I have defined a structure for maintaining the index and merge size for each thread. The code for creating the first threads will be the same. The code for merging threads will be similar to the below snippet. After creating the threads, num_threads will become half and then the merging process continues...

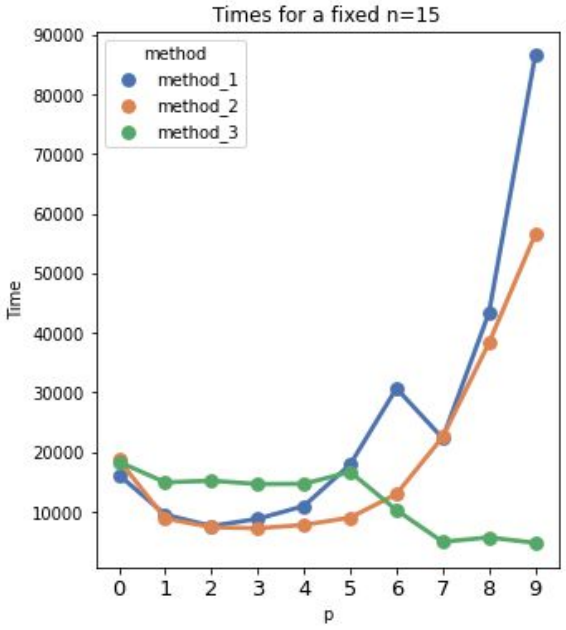
```c
struct args {
    int index;
    int merge_size;
};
typedef struct args args;
```

```c
void* merge_workers(void* arg) { // Function to merge segments 2 at a time
    int index = ((args*)arg)->index; // get index of threads
    int merge_size = ((args*)arg)->merge_size;
    int l = index*merge_size, r = l + merge_size - 1; // range of indices to be merged
    int mid = (l + r - 1)/2;
    merge(l, mid, r);
}
```

```c
num_workers /= 2;
int merge_size = 2*pow(2, n-p);
while(num_workers) { // while no. of segments>1 merge segments
    for (int i = 0; i < num_workers ; i++) {
        args* th_num = malloc(sizeof(*th_num));
        th_num->index = i;
        th_num->merge_size = merge_size;
        pthread_create(&workers[i], &attr, merge_workers, (void*)th_num);
    }
    for (int i = 0; i < num_workers; i++)
        pthread_join(workers[i], NULL); // waiting for threads to merge
    num_workers /= 2;
    merge_size *= 2;
}
```

Analysis:

Graph#1: $N = 15,\ p \in [0, 6]$.

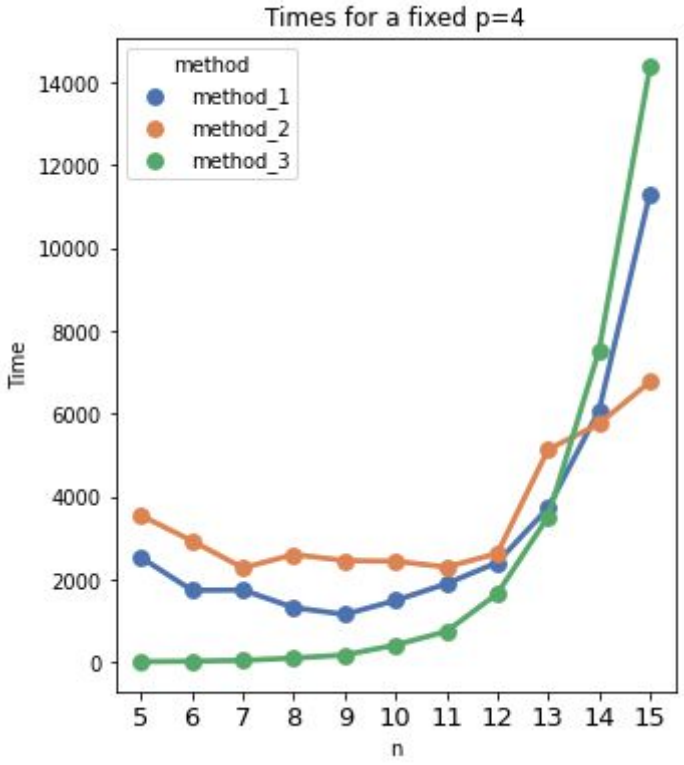


- We can see from the graph that the Sequential Algorithm takes the (Method-3) least time among all (for p>=5) in most of the cases because thread creation for sorting is a costly process. It can be naively implemented without using threads.

- Among Method-I and Method-II we can see the trend that, as number of threads increases and as the array size is large (2^{15}) Method-II is working more efficiently than Method-I, as different segments are merged by different threads rather than a single main thread merging all threads.

- Up to some point both the Method-I and II are decreasing as we increase "P" as the merging is done faster in both cases. But after p=4 we can see that there is significant increase in time in Method-I as the Merging process is done wholly by only a single thread. Whereas It increased in Method-II because it's always efficient to create the number of threads<=no. Of cores. My PC consisted of 4 cores. Hence p<=4 was ideal. For p>4 the time increased in Method-II too.

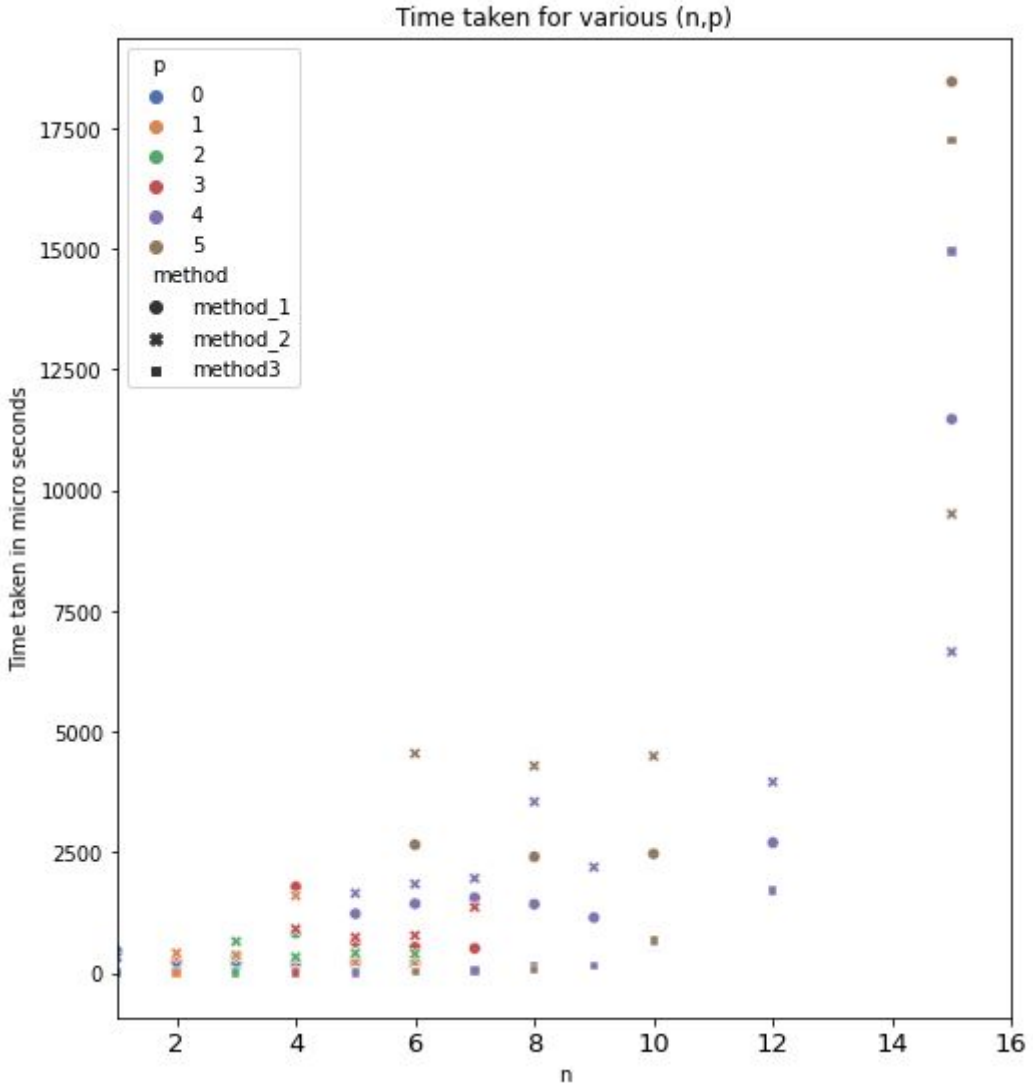
Graph#2: $p = 4,\ n \in [5, 11]$



- Same as the case of "n" fixed, here too Sequential Algorithm (Method-3) takes the least time among all up to a point where n=12, because thread creation for sorting is a costly process. After that we can see that there is a significant decrease in time in Methods-I and II.

- We see that Method-II takes more time than Method-I up to n=10, as initially we are creating less number of threads and there are less number of elements. As no. of threads is fixed in the first round (4 threads in first round) **.** In consequent rounds the waiting time for the other thread to merge is greater than the time for the whole thread to merge the array. Hence we can see that for small values of "n, Method-II takes a large amount of time compared to Method-I. Whereas after n>=14 Method-II takes less time than Method-I.

Graph#3 : Time variance for various (n,p)



- Time taken for various (n,p) are plotted as a scatter plot. The Various (n,p) values are mentioned here.

- We can see from the graph that the Sequential method takes least time and in general up to n=12. If the number of array elements is very large and there are sufficient number of Threads created so that the CPU has a perfect balance between maintaining the number of Threads and Cores Method-II will be performing well else Method-I performs better.

Code for Plotting Graphs:

https://colab.research.google.com/drive/1_m80pxFXtkjC4Yd0k3RCQ11NF5l3YTIv?usp=sharing

********** THE END **********