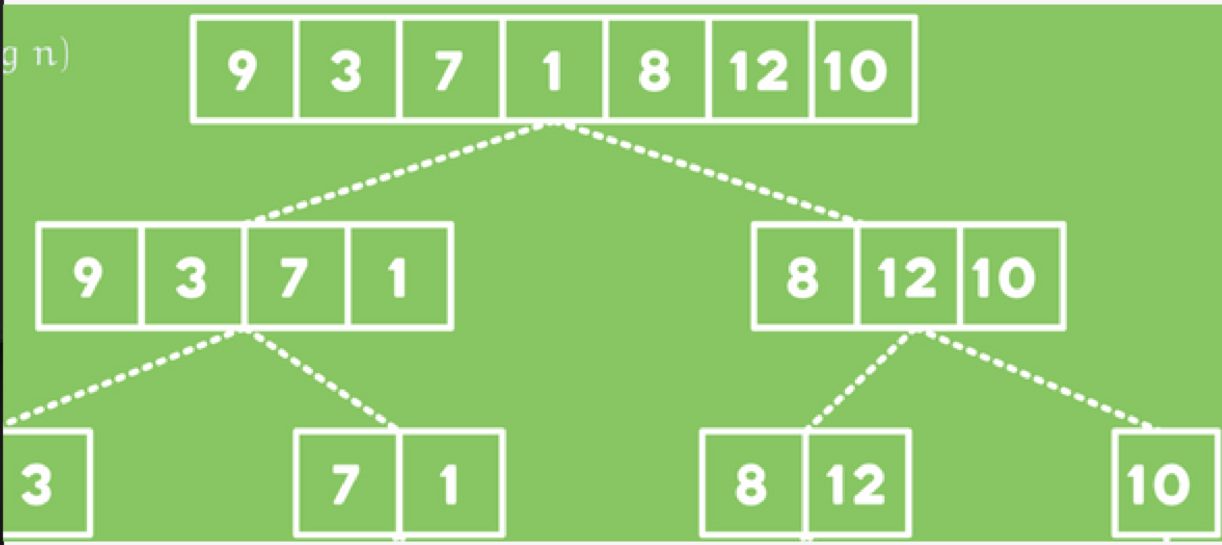
OSN ASSIGNMENT - 4 // CONCURRENCY

OCTOBER 2020

CONCURRENT MERGE_SORT



PREPARED AND PRESENTED BY

AYUSHMAN PANDA
2020121007

MERGE_SORT :

Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. The merge() function
 is used for merging two halves. The merge(arr, l, m, r) is key process
that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the
two sorted sub-arrays into one.

CONCURRENT_ MERGE_SORT() :

The MErge sort can be made concurrent by recurssively making separate process for each half. Separate process will take care of sorting each half and it seems as an efficient and faster way to sort a given array.

THREADED_ MERGE_SORT():

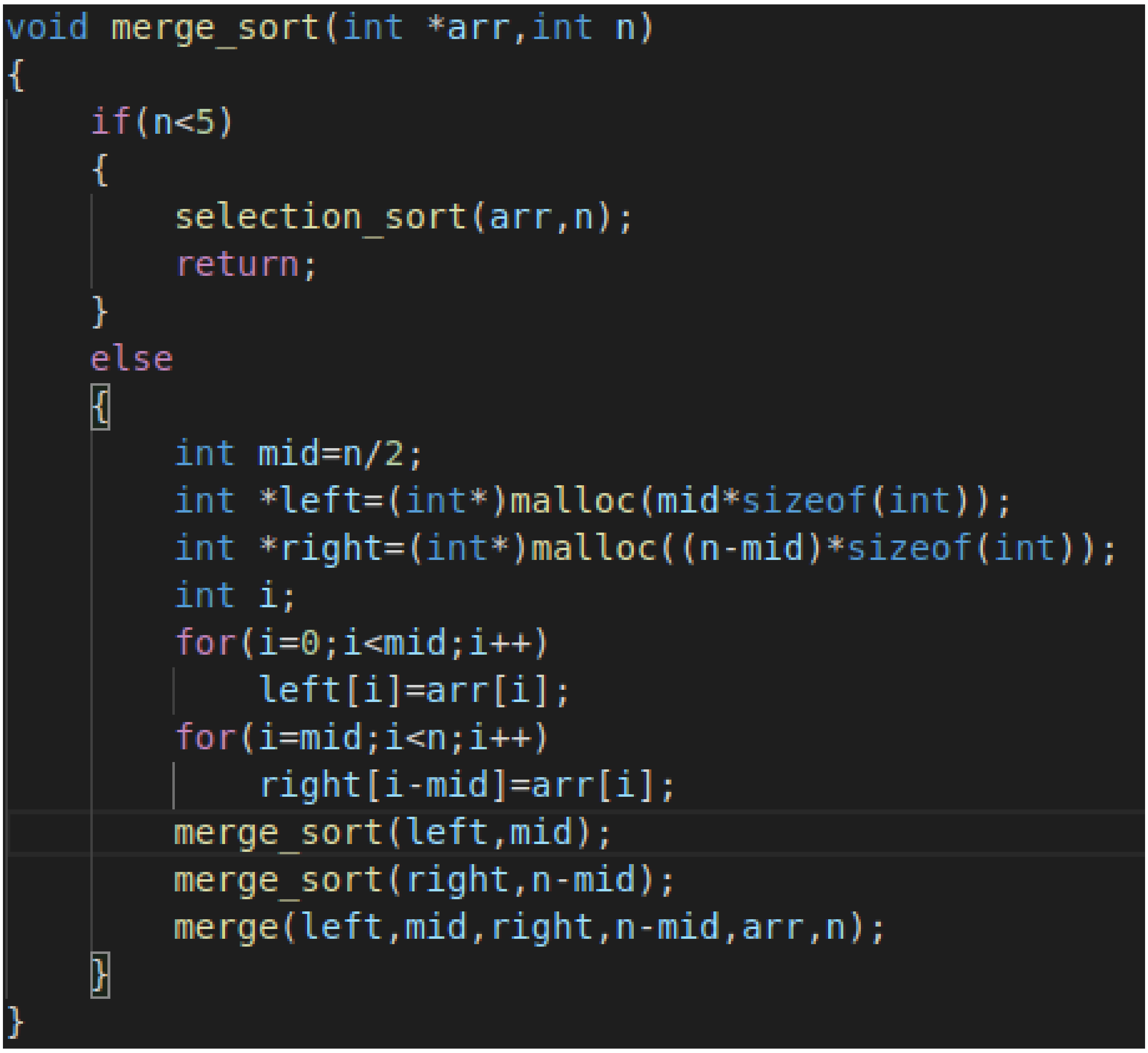
Threaded merge sort can be implemented similarly ,just that in place of creating separate process for each half of the subarray , we create threads and let them handle each subarray separately .This approach too ,seems to work faster than the normal merge sort.

NOTE:

When the number of elements in the array/subarray becomes less than 5 , we perform a selection sort.

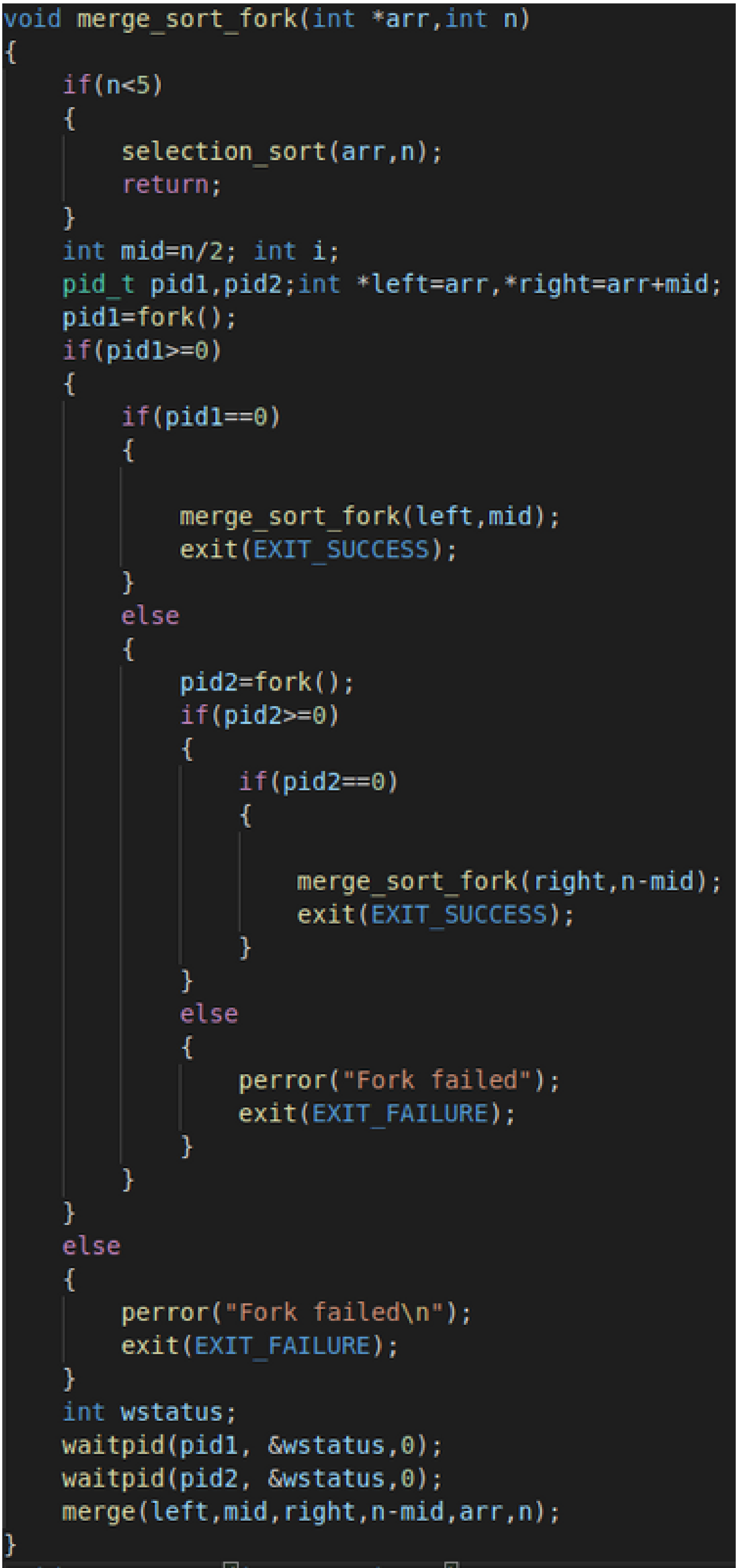
PSEUDO_FUNCTIONS :

Normal Merge Sort :



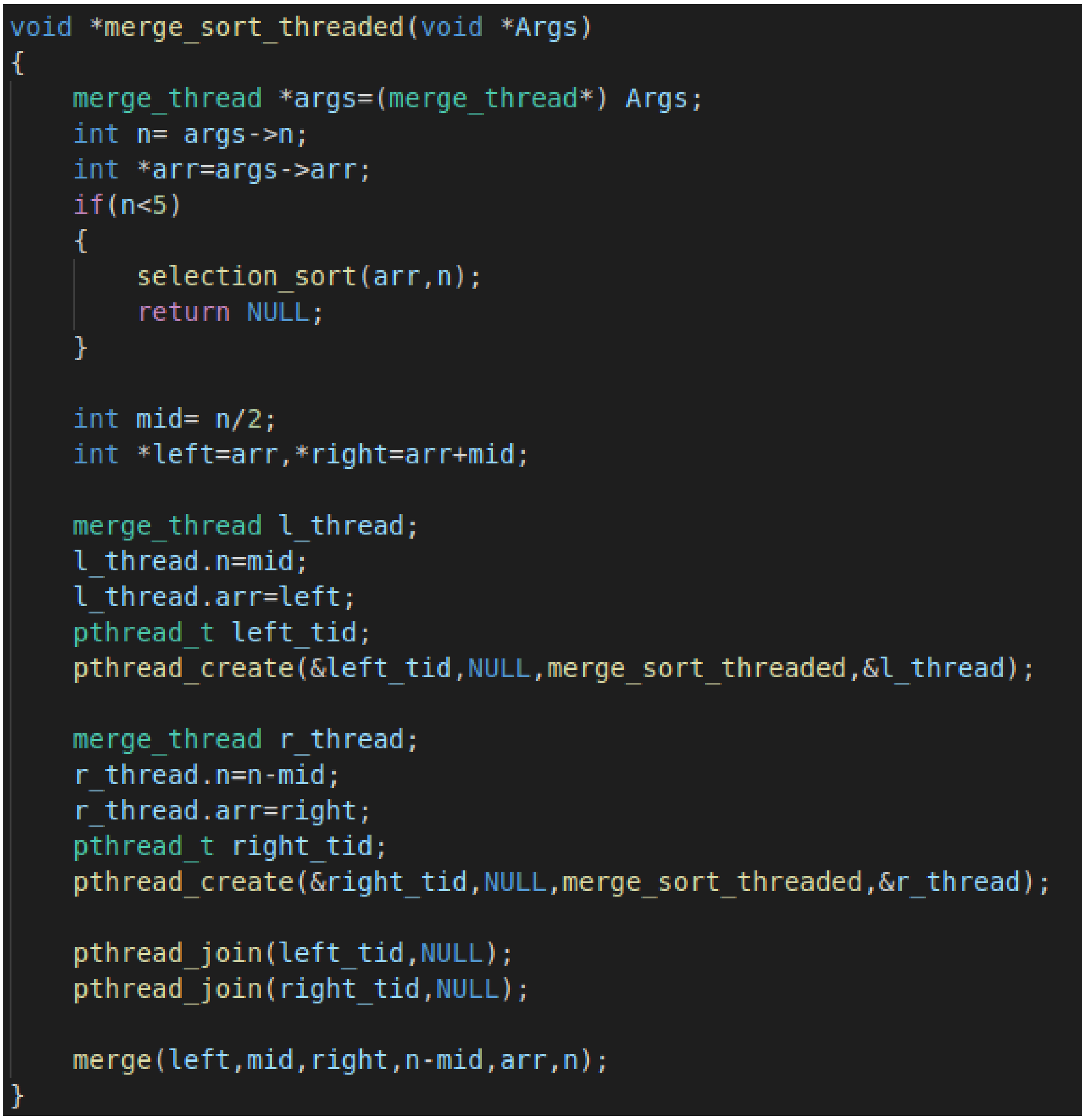
```
void merge_sort(int *arr,int n)
{
    if(n<5)
    {
        selection_sort(arr,n);
        return;
    }
    else
    {
        int mid=n/2;
        int *left=(int*)malloc(mid*sizeof(int));
        int *right=(int*)malloc((n-mid)*sizeof(int));
        int i;
        for(i=0;i<mid;i++)
            left[i]=arr[i];
        for(i=mid;i<n;i++)
            right[i-mid]=arr[i];
        merge_sort(left,mid);
        merge_sort(right,n-mid);
        merge(left,mid,right,n-mid,arr,n);
    }
}
```

Concurrent Merge Sort :



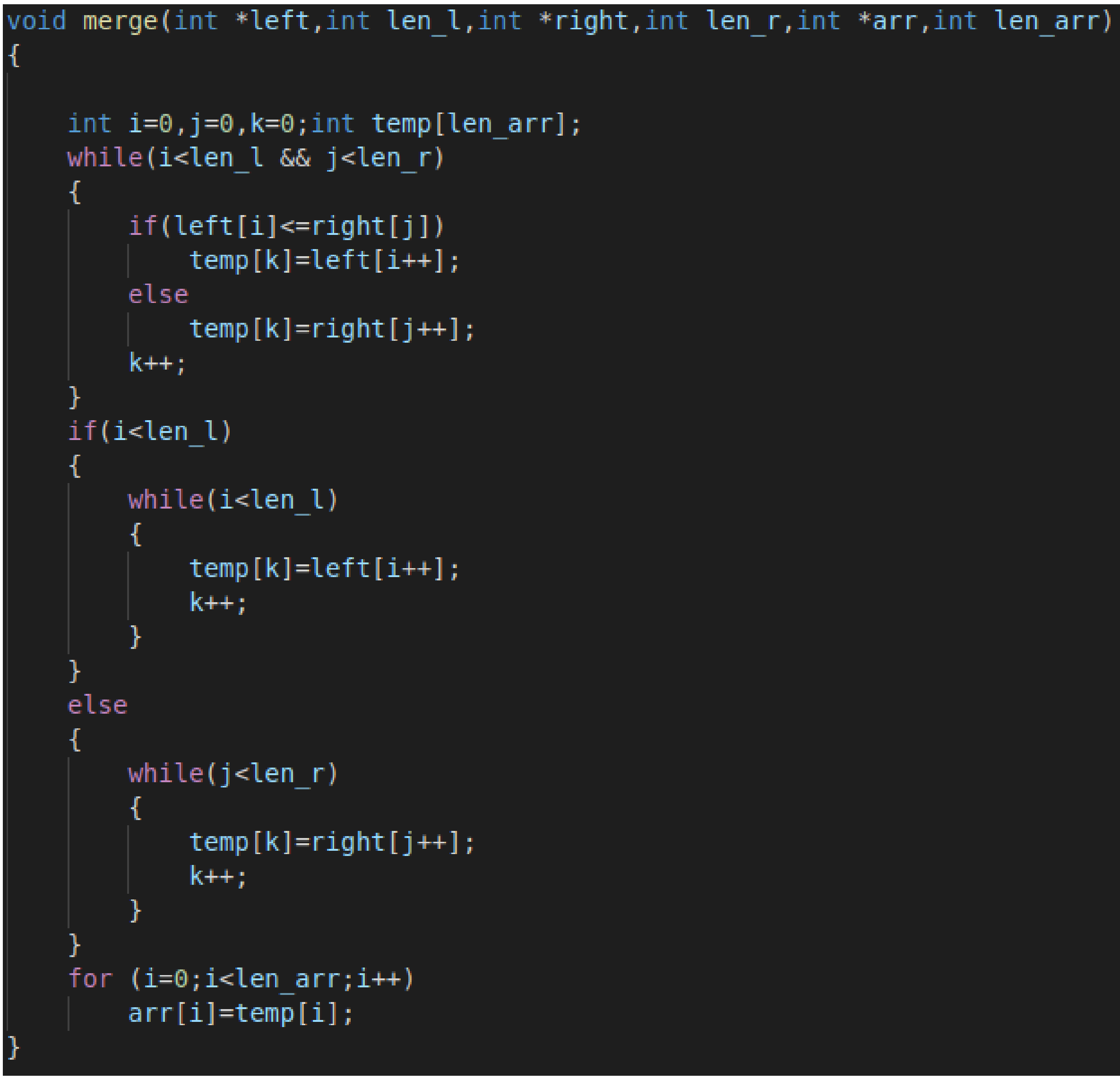
```
void merge_sort_fork(int *arr,int n)
{
    if(n<5)
    {
        selection_sort(arr,n);
        return;
    }
    int mid=n/2; int i;
    pid_t pid1,pid2;int *left=arr,*right=arr+mid;
    pid1=fork();
    if(pid1>=0)
    {
        if(pid1==0)
        {

            merge_sort_fork(left,mid);
            exit(EXIT_SUCCESS);
        }
        else
        {
            pid2=fork();
            if(pid2>=0)
            {
                if(pid2==0)
                {

                    merge_sort_fork(right,n-mid);
                    exit(EXIT_SUCCESS);
                }
            }
            else
            {
                perror("Fork failed");
                exit(EXIT_FAILURE);
            }
        }
    }
    else
    {
        perror("Fork failed\n");
        exit(EXIT_FAILURE);
    }
    int wstatus;
    waitpid(pid1, &wstatus,0);
    waitpid(pid2, &wstatus,0);
    merge(left,mid,right,n-mid,arr,n);
}
```

Threaded Merge Sort :



```
void *merge_sort_threaded(void *Args)
{
    merge_thread *args=(merge_thread*) Args;
    int n= args->n;
    int *arr=args->arr;
    if(n<5)
    {
        selection_sort(arr,n);
        return NULL;
    }

    int mid= n/2;
    int *left=arr,*right=arr+mid;

    merge_thread l_thread;
    l_thread.n=mid;
    l_thread.arr=left;
    pthread_t left_tid;
    pthread_create(&left_tid,NULL,merge_sort_threaded,&l_thread);

    merge_thread r_thread;
    r_thread.n=n-mid;
    r_thread.arr=right;
    pthread_t right_tid;
    pthread_create(&right_tid,NULL,merge_sort_threaded,&r_thread);

    pthread_join(left_tid,NULL);
    pthread_join(right_tid,NULL);

    merge(left,mid,right,n-mid,arr,n);
}
```

The Merge Function :



```c
void merge(int *left,int len_l,int *right,int len_r,int *arr,int len_arr)
{

    int i=0,j=0,k=0;int temp[len_arr];
    while(i<len_l && j<len_r)
    {
        if(left[i]<=right[j])
            temp[k]=left[i++];
        else
            temp[k]=right[j++];
        k++;
    }
    if(i<len_l)
    {
        while(i<len_l)
        {
            temp[k]=left[i++];
            k++;
        }
    }
    else
    {
        while(j<len_r)
        {
            temp[k]=right[j++];
            k++;
        }
    }
    for (i=0;i<len_arr;i++)
        arr[i]=temp[i];
}
```

Report :

For n=5

Normal Merge sort =
0.000008 secs

Concurrent Merge sort=
0.000337 secs

Threaded Merge sort=
0.000629 secs

For n=10

Normal Merge sort =
0.000026 secs

Concurrent Merge sort=
0.000424 secs

Threaded Merge sort=
0.006501 secs

For n=500

Normal Merge sort =
0.000278 secs

Concurrent Merge sort=
0.000373 secs

Threaded Merge sort=
0.013007 secs

For n=1000

Normal Merge sort =
0.000539 secs

Concurrent Merge sort=
0.000411 secs

Threaded Merge sort=
0.024299 secs

Conclusion :

We see that , normal merge_sort is faster than concurrent merge sort and threaded merge sort in cases where number of elements are comparatively lesser (approx. <1000)

When number of elements increases more than around 1000, the concurrent merge sort sorts faster than normal merge sort , but normal merge sort ,still remains faster than threaded merge sort.

The normal merge sort ,sorts faster because ,it doesn't have to create additional processes and threads like concurrent and threaded merge sort.