Concurrent Merge Sort

Explanation of the implemented Code

Tabel of Content

Function to get shared memory

This function returns a pointer to (type casted to int) shared memory which gets share between the childs of a process or between threads.Shmtget is used to allocate the memory segment and shmat is used to attach that shared segment to the return int pointer. Errors have been handled.

```
ll  *get_shared_mem(int arr_size)
 {

     key_t share_mem_key = IPC_PRIVATE;
     size_t SHM_SIZE = arr_size * 8;
ll      shared_mem_id; ll *shared_mem_at; ll f = 0;
     if ((shared_mem_id = shmget(share_mem_key, SHM_SIZE, IPC_CREAT | 0666)) < 0)
     {
         perror("SHMGET");
         f = 1;
     }
     if ((shared_mem_at = shmat(shared_mem_id, NULL, 0)) == (ll *)-1)
     {
         perror("SHMAT");
         f = 1;
     }

     if (f)
     {
         return NULL;
     }
     return shared_mem_at;
 }
```

Merge Function

We can directly copy the sorted elements in the final array, no need for a temporary sorted array. This is for merging left and right array.It takes argument as array,left index, right index, mid value.

```
void merge(ll *a, ll l1, ll h1, ll h2)
{

    int count, m = 0;
    count = h2 + 1 - l1;
    int sorted[count + 1];
    int i = l1;
    int k = 1 + h1;

    while (h1 >= i && h2 >= k)
    {
        if (0 < (a[k] - a[i]))
        {
            sorted[m++] = a[i++];
        }
        else if ((a[i] - a[k]) == 0)
        {
            sorted[m++] = a[i++];

            sorted[m++] = a[k++];
        }
        else if (0 < (a[i] - a[k]))
        {
            sorted[m++] = a[k++];
        }
    }

    while (0 <= (h1 - i))
    {
        sorted[m++] = a[i++];
    }
    while (0 <= (h2 - k))
    {
        sorted[m++] = a[k++];
    }
    int arr_count = 0;
    arr_count = l1;
    for (i = 0; i < count; ++l1, i++)
    {
        a[l1] = sorted[i];
    }
}
```

Normal Merge Sort

This function implemends normal merge sort

```
void *normal_mergesort(ll *BRR, ll low, ll high)
{
    ll sz = high + 1 - low, mid;
```

```
    if (sz >= 5)
    {
        mid = low + ((high - low) >> 1);
        normal_mergesort(BRR, low, mid), normal_mergesort(BRR, mid + 1, high);
        merge(BRR, low, mid, high);
    }
    else
    {
        selection_sort(BRR, low, high);
    }
    return NULL;
}
```

Multi Process Merge Sort

This function implements that if array has length less than 5 then it performs selection sort . If greater then it splits the array into left and right part and forks and create process to sort left part and in parent it again forks to create process to sort right part . In the parent it waits for child process and join left and right process.

```
void *multiprocess_mergesort(ll *ARR, ll low, ll high)
{
ll      sz = high + 1 - low, mid; if (sz < 5)
    {
        selection_sort(ARR, low, high);
        return NULL;
    }

    mid = low + (high - low) / 2;
    pid_t pid1;
    pid_t pid2;
    pid1 = fork();
    if (pid1 < 0)
    {
        perror("Left Child Process not created\n");
        return NULL;
    }
    else if (!pid1)
    {
        multiprocess_mergesort(ARR, low, mid);
        _exit(1);
    }
    else if (0 < pid1)
    {
        pid2 = fork();
        if (pid2 < 0)
        {
            perror("Right Child Process not created");
            return NULL;
        }
```

```
        else if (!pid2)
        {
            multiprocess_mergesort(ARR, mid + 1, high);
            _exit(1);
        }
        else
        {
            int status;
            waitpid(pid1, &status, 0), waitpid(pid2, &status, 0);
            merge(ARR, low, mid, high);
        }
    }
    return NULL;
}
```

Threaded Multi Sort

This function is already well commented for understanding.

```
void *threaded_mergesort(void *T_ARR)
{
/       Extracting the struct out of parameter T_ARR ll l, r, mid;
    struct array *T_arr = (struct array *)T_ARR;

/       Extracting values of Array to be sorted.
    r = T_arr->R_INDEX;
    l = T_arr->L_INDEX;
    ll *arr = T_arr->ARRAY;

/       Checking for single item array. ll sz = r + 1 - l;
    if (sz < 5)
    {
        selection_sort(arr, l, r);
        return NULL;
    }

/       Finding middle index from where to divide the array. mid = l + (r - l) / 2;

    struct array L_arr, R_arr;
/       Allocating struct for left half of the provided array L_arr.R_INDEX = mid;
    L_arr.L_INDEX = l;
    L_arr.ARRAY = arr;

/       Allocating struct for right half of the provided array R_arr.R_INDEX = r;
    R_arr.L_INDEX = mid + 1;
    R_arr.ARRAY = arr;
```

```
/       Creating threads for both half. pthread_t tid1;
    pthread_t tid2;
    pthread_create(&tid1, NULL, threaded_mergesort, &L_arr), pthread_create(&tid2,
NULL, threaded_mergesort, &R_arr);
    //Joining Threads
    pthread_join(tid1, NULL), pthread_join(tid2, NULL);

/       merge the two half. merge(arr, l, mid, r);
}
```

main function

This is the driver code of the problem where it calls different functions for execution . The basic general functions for printing has not been explained as they were understood from their name.

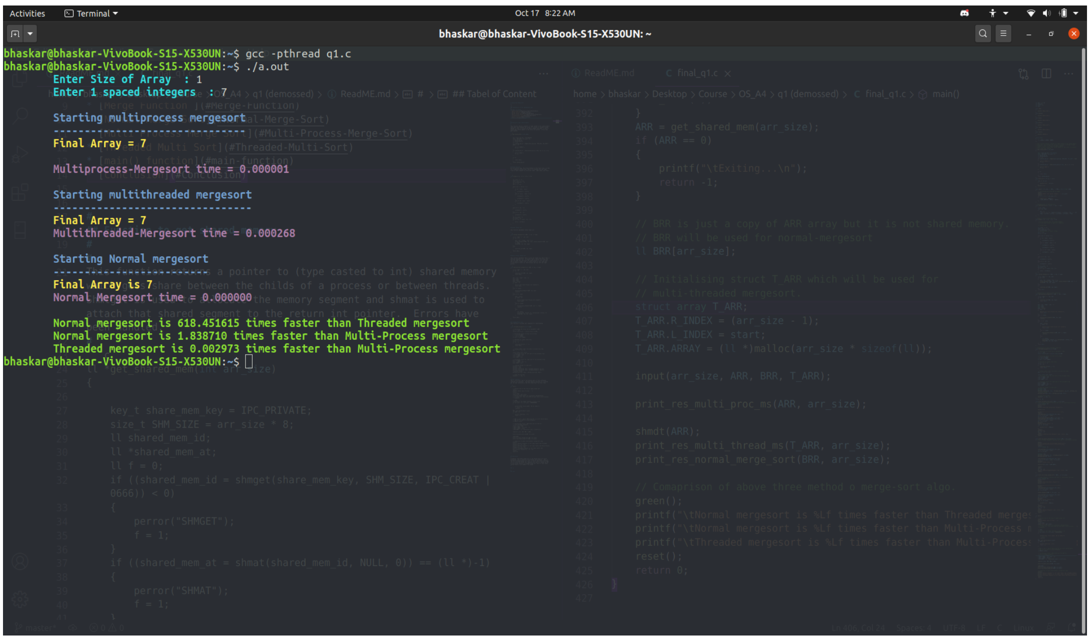
```
int main()
{
/       Getting size of the array to apply merge sort. cyan();
    printf("\tEnter Size of Array : ");
    reset();
    ll arr_size;
    scanf("%lld", &arr_size);
/       Getting shared memory for processses implementation
/       of merge sort. This array ARR will get used in multi-process
/       mergesort.

    ll *ARR;
    //checking if the number entered is valid or not
    if (arr_size < 0 || arr_size > 99999999999)
    {

        printf("Enter valid number. Re-run Code\n");
/           return -1; _exit(1);
    }
    ARR = get_shared_mem(arr_size);
    if (ARR == 0)
    {
        printf("\tExiting...\n");
        return -1;
    }

/       BRR is just a copy of ARR array but it is not shared memory.
/       BRR will be used for normal-mergesort
    ll BRR[arr_size];

/       Initialising struct T_ARR which will be used for
/       multi-threaded mergesort.
```

```
    struct array T_ARR;
    T_ARR.R_INDEX = (arr_size - 1);
    T_ARR.L_INDEX = start;
    T_ARR.ARRAY = (ll *)malloc(arr_size * sizeof(ll));

    input(arr_size, ARR, BRR, T_ARR);

    print_res_multi_proc_ms(ARR, arr_size);

    shmdt(ARR);
    print_res_multi_thread_ms(T_ARR, arr_size);
    print_res_normal_merge_sort(BRR, arr_size);

/      Comaprison of above three method o merge-sort algo. green();
    printf("\tNormal mergesort is %Lf times faster than Threaded mergesort\n", t2 /
t1);
    printf("\tNormal mergesort is %Lf times faster than Multi-Process mergesort\n", t3
/ t1);
    printf("\tThreaded mergesort is %Lf times faster than Multi-Process mergesort\n",
t3 / t2);
    reset();
    return 0;
}
```

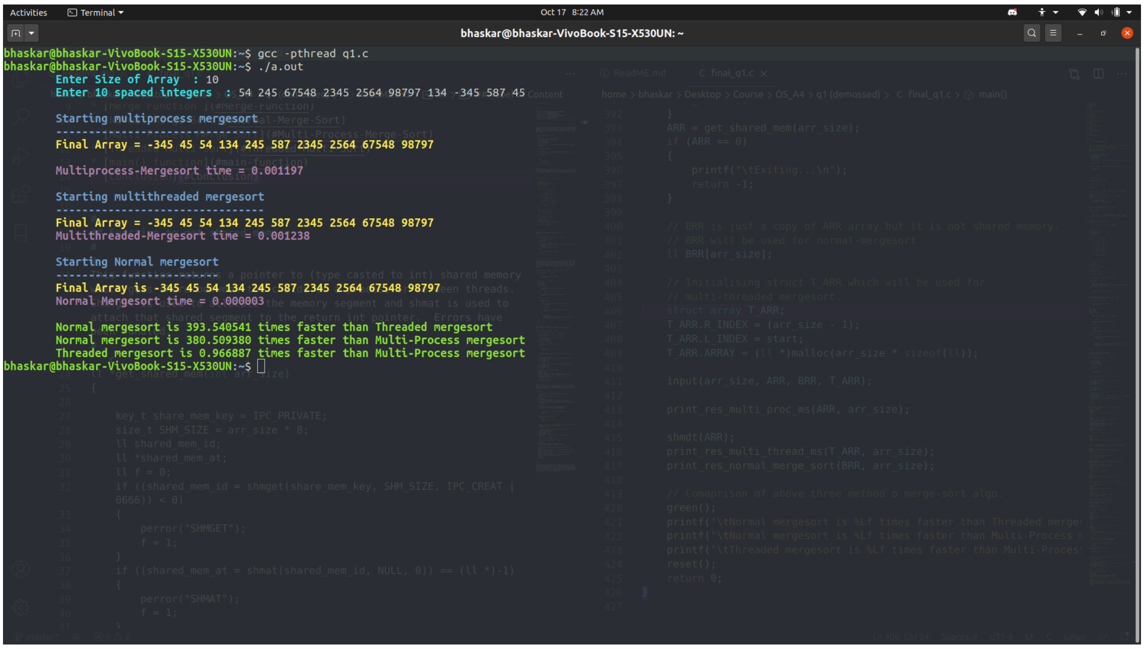
Conclusion

In General (for large n),The normal merge sort is the fastest due to no extra processing and no creation of the threads and processes . Next is the Multi-threaded Merge sort which is faster than Concurrent Merge sort as Concurrent Merge sort requires more time for coping the PCB of the process whereas in Threaded Merge Sort there is some common section which is not needed to be copied. The reason of normal merge sort being faster than the other can be understood due to **Context Switching** and also due to **Cache Misses**.

N=1



N=10

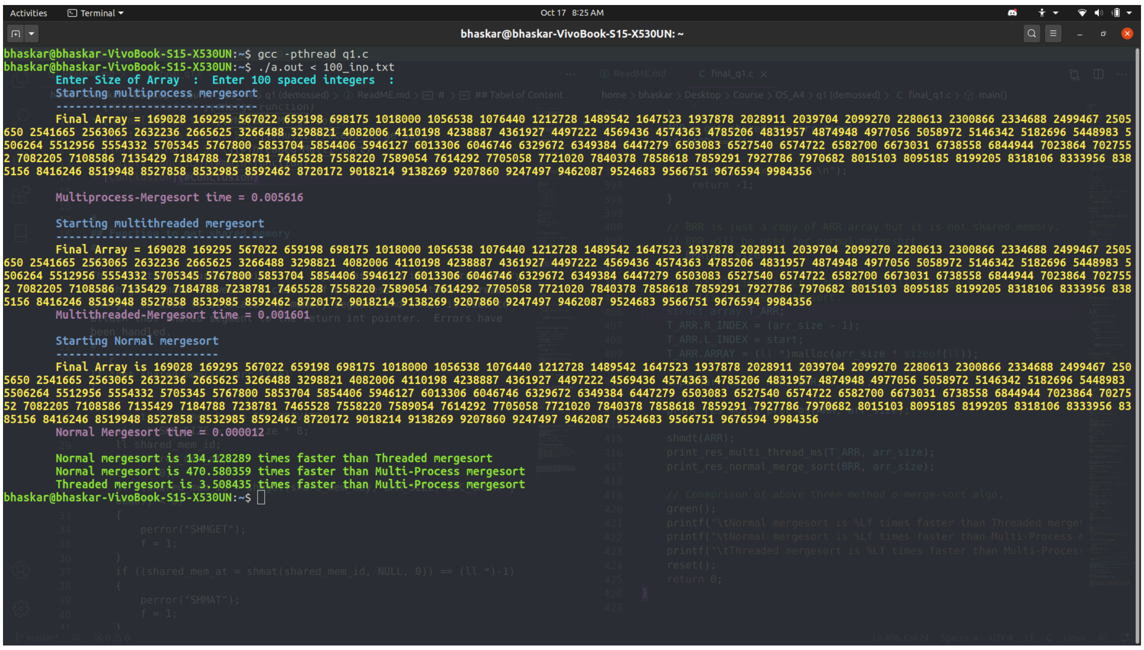

Activities
Terminal
bhaskar@bhaskar-VivoBook-S15-X530UN: ~
bhaskar@bhaskar-VivoBook-S15-X530UN:~$ gcc -pthread q1.c
bhaskar@bhaskar-VivoBook-S15-X530UN:~$ ./a.out
Enter Size of Array : 10
Enter 10 spaced integers : 54 245 67548 2345 2564 98797 134 -345 587 45
Starting multiprocess mergesort

Final Array = -345 45 54 134 245 587 2345 2564 67548 98797
Multiprocess-Mergesort time = 0.001197
Starting multithreaded mergesort

Final Array = -345 45 54 134 245 587 2345 2564 67548 98797
Multithreaded-Mergesort time = 0.001238
Starting Normal mergesort

Final Array is -345 45 54 134 245 587 2345 2564 67548 98797
Normal Mergesort time = 0.000003
Normal mergesort is 393.540541 times faster than Threaded mergesort
Normal mergesort is 380.509380 times faster than Multi-Process mergesort
Threaded mergesort is 0.966887 times faster than Multi-Process mergesort
bhaskar@bhaskar-VivoBook-S15-X530UN:~$

N=100

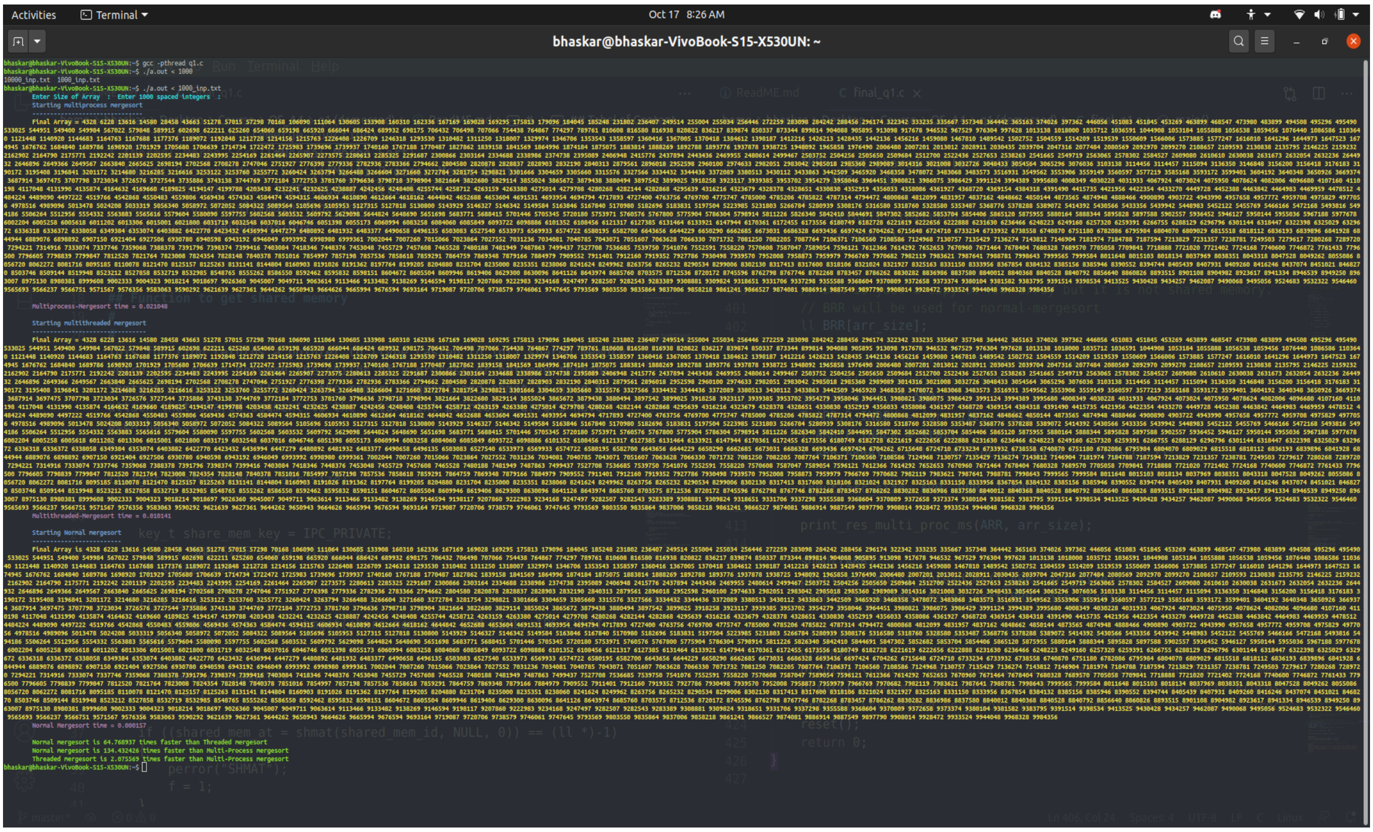

Activities
Terminal
Oct 17 8:25 AM
bhaskar@bhaskar-VivoBook-S15-X530UN: ~
bhaskar@bhaskar-VivoBook-S15-X530UN:~$ gcc -pthread q1.c
bhaskar@bhaskar-VivoBook-S15-X530UN:~$ ./a.out < 100_inp.txt
Enter Size of Array : Enter 100 spaced integers :
Starting multiprocess mergesort

Final Array = 169028 169295 567022 659198 698175 1018000 1056538 1076440 1212728 1489542 1647523 1937878 2028911 2039704 2099270 2280613 2300866 2334688 2499467 2505
650 2541665 2563065 2632236 2665625 3266488 3298821 4082006 4110198 4238887 4361927 4497222 4569436 4574363 4785206 4831957 4874948 4977056 5058972 5146342 5182696 5448983 5
506264 5512956 5554332 5705345 5767800 5853704 5854406 5946127 6013306 6046746 6329672 6349384 6447279 6503083 6527540 6574722 6582700 6673031 6738558 6844944 7023864 702755
2 7082205 7108586 7135429 7184788 7238781 7465528 7558220 7589054 7614292 7705058 7721020 7840378 7858618 7859291 7927786 7970682 8015103 8095185 8199205 8318106 8333956 838
5156 8416246 8519948 8527858 8532985 8592462 8720172 9018214 9138269 9207860 9247497 9462087 9524683 9566751 9676594 9984356
Multiprocess-Mergesort time = 0.005616
Starting multithreaded mergesort

Final Array = 169028 169295 567022 659198 698175 1018000 1056538 1076440 1212728 1489542 1647523 1937878 2028911 2039704 2099270 2280613 2300866 2334688 2499467 2505
650 2541665 2563065 2632236 2665625 3266488 3298821 4082006 4110198 4238887 4361927 4497222 4569436 4574363 4785206 4831957 4874948 4977056 5058972 5146342 5182696 5448983 5
506264 5512956 5554332 5705345 5767800 5853704 5854406 5946127 6013306 6046746 6329672 6349384 6447279 6503083 6527540 6574722 6582700 6673031 6738558 6844944 7023864 702755
2 7082205 7108586 7135429 7184788 7238781 7465528 7558220 7589054 7614292 7705058 7721020 7840378 7858618 7859291 7927786 7970682 8015103 8095185 8199205 8318106 8333956 838
5156 8416246 8519948 8527858 8532985 8592462 8720172 9018214 9138269 9207860 9247497 9462087 9524683 9566751 9676594 9984356
Multithreaded-Mergesort time = 0.001601
Starting Normal mergesort

Final Array is 169028 169295 567022 659198 698175 1018000 1056538 1076440 1212728 1489542 1647523 1937878 2028911 2039704 2099270 2280613 2300866 2334688 2499467 250
5650 2541665 2563065 2632236 2665625 3266488 3298821 4082006 4110198 4238887 4361927 4497222 4569436 4574363 4785206 4831957 4874948 4977056 5058972 5146342 5182696 5448983
5506264 5512956 5554332 5705345 5767800 5853704 5854406 5946127 6013306 6046746 6329672 6349384 6447279 6503083 6527540 6574722 6582700 6673031 6738558 6844944 7023864 70275
52 7082205 7108586 7135429 7184788 7238781 7465528 7558220 7589054 7614292 7705058 7721020 7840378 7858618 7859291 7927786 7970682 8015103 8095185 8199205 8318106 8333956 83
85156 8416246 8519948 8527858 8532985 8592462 8720172 9018214 9138269 9207860 9247497 9462087 9524683 9566751 9676594 9984356
Normal Mergesort time = 0.000012
Normal mergesort is 134.128289 times faster than Threaded mergesort
Normal mergesort is 470.580359 times faster than Multi-Process mergesort
Threaded mergesort is 3.508435 times faster than Multi-Process mergesort
bhaskar@bhaskar-VivoBook-S15-X530UN:~$

N=1000



N=10000