Question 1 PDF:

MERGE SORT

MergeSort recursively divides the array into two halves, sort the segments and merge them finally. In the given problem first we check whether the size of the arrray segment is less than 5 if it is then we do selection sort on that segment and returns. If not we calculate the middle position and divide the array on the middle element (first part contains the middle element). Then we call MergeSort on both the halves. Same Process takes place until the size of array becomes 1 . Now we keep on the merging the segments to form the sorted array.

```
void mergeSort(int arr[], int l, int r)
{

        if (l < r)
        {
                if(r-l<5)
                {
                        int i, j, min_idx;

                        // One by one move boundary of unsorted subarray
                        for (i = l; i < r; i++)
                        {
                                // Find the minimum element in unsorted
array
                                min_idx = i;
                                for (j = i+1; j < r+1; j++)
                                        if (arr[j] < arr[min_idx])
                                                min_idx = j;

                                // Swap the found minimum element with the
first element
                                swap(&arr[min_idx], &arr[i]);

                        }
                }
                else
                {

                        // Same as (l+r)/2, but avoids
                        // overflow for large l and h
                        int m = l + (r - l) / 2;

                        // Sort first and second halves
                        mergeSort(arr, l, m);
                        mergeSort(arr, m + 1, r);
                        // wait until both are completed

                        merge(arr, l, m, r);
                }
        }
}
```

CONCURRENT MERGE SORT

```
ProcessMergesort function does the mergesort using process. Processes are
created using fork . As mentioned above if  the array size is less than or
equal to 5, It does selection sort. Otherwise it forks two children left
and right processes and each handles the sorting of left and right halves
respectively. The left child process calls the mergesort on the left
partition of the array and right child process calls mergesort on right
child. The two processes will sort the left and right parts respectively.
After they both return we do merge them.
```

```
void forkmergeSort( int * arr , int l , int r)
{
        if(l<r)
        {
                if(r-l<5)
                {
                        int i, j, min_idx;

                        // One by one move boundary of unsorted subarray
                        for (i = l; i < r; i++)
                        {
                                // Find the minimum element in unsorted
array
                                min_idx = i;
                                for (j = i+1; j < r+1; j++)
                                        if (arr[j] < arr[min_idx])
                                                min_idx = j;

                                // Swap the found minimum element with the
first element
                                swap(&arr[min_idx], &arr[i]);

                        }
                }

                else
                {
                        int m = l + (r - l) / 2;

                        int pid1=fork();
                        int pid2;
                        if(pid1==0){
```

```
                                                //sort the left half of the array
                                                forkmergeSort(arr, l, m);
                                                _exit(1);
                                        }
                                        else{
                                                pid2=fork();
                                                if(pid2==0){
                                                        // sort the right half of the array
                                                        forkmergeSort(arr, m+1, r);
                                                        _exit(1);
                                                }
                                                else{
                                                        // wait until both are completed
                                                        int status;
                                                        waitpid(pid1, &status, 0);
                                                        waitpid(pid2, &status, 0);
                                                        merge(arr, l , m , r);
                                                }
                                        }
                                        return;

                                }
                        }

        }
```

THREADED MERGE SORT

Threaded Mergesort is pretty much the same as process mergesort. The threads takes an void struct as arguments which contains need arguments . . As before, if the array size is less than or equal to 5 it does selection sort. Otherwise it creates two threads, left thread and right thread. And the left thread does sorting of left half of array and the right thread does the sorting of right half of the array by calling threadedMergeSort on the respective parts. The main thread waits until both the threads join back and then the Function calls the Merge function.

```
void *threaded_mergeSort(void *a)
{
        struct arg *args =  (struct arg*)a;

        int l = args->l;
        int r = args->r;
        int *arr = args->arr;
        if(l>r) return NULL;

        if( l < r )
        {
                if(r-l<5)
                {
                        int i, j, min_idx;

                        // One by one move boundary of unsorted subarray
                        for (i = l; i < r; i++)
                        {
                                // Find the minimum element in unsorted
array
                                min_idx = i;
                                for (j = i+1; j < r+1; j++)
                                        if (arr[j] < arr[min_idx])
                                                min_idx = j;

                                // Swap the found minimum element with the
first element
                                swap(&arr[min_idx], &arr[i]);
                        }

                }
                else
                {
                        int m = l + (r - l) / 2;
                        // sort the left half of the array
                        struct arg a1;
                        a1.l = l;
                        a1.r = m;
                        a1.arr = arr;
                        pthread_t tid1;
                        pthread_create(&tid1, NULL, threaded_mergeSort,
&a1);

                        //sort right half array
                        struct arg a2;
                        a2.l = m+1;
                        a2.r = r;
                        a2.arr = arr;
                        pthread_t tid2;
                        pthread_create(&tid2, NULL, threaded_mergeSort,
&a2);

                        //wait for the two halves to get sorted
                        pthread_join(tid1, NULL);
                        pthread_join(tid2, NULL);

                        merge(arr , l , m ,r); // call merge when both the
threads join back the main thread.
                        cnttt++;
                }
        }


}
```

Some important functions

1. runSort : The runsort function calls all the three types of mergeSort functions. It records the clock just before calling the process and records the time just after the process returns. By using these we get the run time of the process.shareMem functions gets shared memory of the size mentioned by argument size.

PERFORMANCE RECORDS:

The outputs are as follows:

```
Enter the value of n: Running concurrent_mergesort for n = 30
62 83 115 126 136 192 235 240 249 259 267 286 293 368 427 429 477 526 530
563 586 590 611 662 667 721 772 782 823 835
time = 0.002559
Running threaded_concurrent_mergesort for n = 30
62 83 115 126 136 192 235 240 249 259 267 286 293 368 427 429 477 526 530
563 586 590 611 662 667 721 772 782 823 835
time = 0.001897
Running normal_Mergesort for n = 30
62 83 115 126 136 192 235 240 249 259 267 286 293 368 427 429 477 526 530
563 586 590 611 662 667 721 772 782 823 835
time = 0.000024
normal_MergeSort ran:
   [ 106.733853 ] times faster than concurrent_Mergesort
   [ 79.116250 ] times faster than threaded_concurrent_Mergesort


Enter the value of n: Running concurrent_mergesort for n = 50
11 27 59 67 123 198 211 263 277 293 362 368 386 492 540 542 558 569 635 667
736 782 815 835 886 926 956 1037 1190 1229 1284 1393 1421 1426 1429 1429
1522 1530 1567 1649 1672 1824 1870 1883 1915 1921 2302 2362 2373 2419
time = 0.002576
Running threaded_concurrent_mergesort for n = 50
11 27 59 67 123 198 211 263 277 293 362 368 386 492 540 542 558 569 635 667
736 782 815 835 886 926 956 1037 1190 1229 1284 1393 1421 1426 1429 1429
1522 1530 1567 1649 1672 1824 1870 1883 1915 1921 2302 2362 2373 2419
time = 0.001021
Running normal_Mergesort for n = 50
11 27 59 67 123 198 211 263 277 293 362 368 386 492 540 542 558 569 635 667
736 782 815 835 886 926 956 1037 1190 1229 1284 1393 1421 1426 1429 1429
1522 1530 1567 1649 1672 1824 1870 1883 1915 1921 2302 2362 2373 2419
time = 0.000012
normal_MergeSort ran:
   [ 216.131830 ] times faster than concurrent_Mergesort
   [ 85.705631 ] times faster than threaded_concurrent_Mergesort

Enter the value of n: Running concurrent_mergesort for n = 1000
12 27 59 336 364 492 540 545 846 886 925 1087 1313 1393 1421 1530 1729 1873
2305 2362 2399 2567 2651 2754 2777 2862 3058 3069 3135 3367 3368 3426 3526
3584 3750 3784 3895 3926 3929 4022 4043 4067.......
time = 0.008097
```

```
Running threaded_concurrent_mergesort for n = 100
12 27 59 336 364 492 540 545 846 886 925 1087 1313 1393 1421 1530 1729 1873
2305 2362 2399 2567 2651 2754 2777 2862 3058 3069 3135 3367 3368 3426 3526
3584 3750 3784 3895 3926 3929 .......
time = 0.004673
Running normal_Mergesort for n = 100
12 27 59 336 364 492 540 545 846 886 925 1087 1313 1393 1421 1530 1729 1873
2305 2362 2399 2567 2651 2754 2777 2862 3058 3069 3135 3367 3368 3426 3526
3584 3750 3784 3895 3926 3929 4022 .......
time = 0.000023
normal_MergeSort ran:
   [ 349.072456 ] times faster than concurrent_Mergesort
   [ 201.465528 ] times faster than threaded_concurrent_Mergesort
```

OBSERVATIONS:

```
• Here for most of the inputs (input size < 1000), The Normal Merge sort
has run faster than Merge sort implemented using creating new process and
threads.
```

REASONS:

```
• The NORMAL MERGE SORT is the efficient for the smaller size inputs since
the implementation using threads and process has the overhead
  of creating various threads and processes respectively.  The overhead
created is much costlier than the time saved due to multiprocessing
  and multithreading.
• Number of Context switches are low in Normal merge sort compared to child
creating method, because in child creating method, we are creating more
  processes which will increase number of processes  which in turn
increases the number of context switches. Although in thread method there
are not
  extra PCB switches but there are less costly thread switches which again
makes normal merge sort a slightly faster algorithm regarding context
  switches .
• As the size of the inputs increases the implementation using threads will
be beneficial because the multithreading and multiprocessing cores
  brings in concurrency and parallelism to the execution. The
implementation using process will also beneficial since it introduces
concurrency
  to the execution.
• For very large inputs the threaded Merge sort runs quicker than Merge
sort implemented using process, Since the context switches would be proven
  costlier than the context switches among the threads (the context
switches among threads is less costly than context switches among the
process).
```