Comparision:

Normal merge sort runs quicker than remaining two processes:

1. There are performance overheads in Threads and Processes, which is due to creation of threads and child at each iteration in Sorting function. Creating these takes significant time, hence delaying those methods.
2. Number of Context switches are low in Normal merge sort compared to child creating method,because in latter we are creating more processes which will increase number of processes to be scheduled by the CPU hence increase in context switches. Although in thread method there are not extra PCB switches but there are less costly thread switches which again makes normal mergesort a slightly faster algorithm.
3. When number of numbers in the array significantly large then thread process can also become faster because of the fact that the threads can run parallely on mutliple cores, which will better the performance.
4. Even incase of processes since many processes are created they will be allocated more CPU slices which will faster the execution, there is also context switching overhead but it depends on the balance between the two and the algorithms in CPU scheduling.
5. The overall ratios depend on the number of inputs and the relative overheads. At smaller inputs Normal merge sort is clear winner but when the input size increases threading method can perform better.

Findings: TESTCASE-1

```
25
24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Running concurrent_mergesort for n = 25
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
time = 0.010430
Running threaded_concurrent_mergesort for n = 25
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
time = 0.004081
Running normal_mergesort for n = 25
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
time = 0.000547
normal_mergesort ran:
        [ 19.061007 ] times faster than concurrent_mergesort
        [ 7.459048 ] times faster than threaded_concurrent_mergesort
```

TESTCASE-2

```
11
23 54 7 2 8 90 87 45 23 98 11
Running concurrent_mergesort for n = 11
2 7 8 11 23 23 45 54 87 90 98
time = 0.001605
Running threaded_concurrent_mergesort for n = 11
```

```
2 7 8 11 23 23 45 54 87 90 98
time = 0.003163
Running normal_mergesort for n = 11
2 7 8 11 23 23 45 54 87 90 98
time = 0.000024
normal_mergesort ran:
        [ 68.194746 ] times faster than concurrent_mergesort
        [ 134.398202 ] times faster than threaded_concurrent_mergesort
```

Explaination:

Threaded Merge Sort:

Incase of threaded merge sort I created a struct which stores the start, end parameters which will be used when passing to the fuctions. So in the function merge sort for passing the funtions sort(arr,start,mid); and sort(arr,mid+1,end); generally we need to replace the corresponding parts with the in struct as shown below. After that it is same as normal merge sort except that the syntax for calling a function is different in case of threads compared to others. But coming to merge it is same for all merge sort algorithms. We do selection sort if the size is less than 5. This is done by checking the size of the array passed in the current iteration end-start+1. Note that start and end are only to explain the terms they shouldnt be confused by the variables in the code.

```
struct arg{
     int l;
     int r;
     int* arr;
};
```

```
struct arg a1;
a1.l = l;
a1.r = ind;
a1.arr = arr;
pthread_t tid1;
pthread_create(&tid1, NULL, threaded_mergeSort, &a1);
```

Processes merge sort:

It is exacly same as the normal merge sort except that the calling of the functions in the sort is not done by the parent process instead we do by forking a new child for each call and we wait for the children to complete their job. Again this has several advantages and disdvantges as discussed earlier. After that we call the merge funtion which is same for all algorithms and if the size of the array of the current iteration is smaller than 5 then we just selection sort as given in the questiuon.

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

```
     mergeSort(arr, low, pi);
     _exit(1);
}
else{
     pid2=fork();
     if(pid2==0){
          mergeSort(arr, pi + 1, high);
          _exit(1);
     }
     else{
          int status;
          waitpid(pid1, &status, 0);
          waitpid(pid2, &status, 0);
          merging(arr,low,pi,high);
     }
}
```

In merge function I will take the two sorted arrays and I will sort them by comapring two indices at a time from arrays, after each iteration I will get one of the position fixed and will be added to the sorted array and the other will continue, In this at the end if more elements remain then I will add to the sorted array so that I will sorted array at the end. In this way we can get sorted combined array. shareMem function gets the size mentioned as the argument. In all the alogithms the selection sort is done using the following code. Which is quite self explainatory.

```
void selectionSort(int arr[], int l,int r)
{
    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]);
    }
}
```