Question1 Report

Normal merge sort runs faster than concurrent merge sort using process and concurrent merge sort using threads.

Reason:

1. There are performance overheads while creating threads and processes. Because creating them is takes more time so it will be delayed in time.
2. There are less context switches in normal merge sort as compared to concurrent merge sort process because this creates new processes which will lead to more context switches the info have to store in pcb and in in concurrent merge sort using thread also have context switches, but they don't have extra pcbs.
3. If the size of the array is very big then thread process will become faster because threads run in parallel on multiple cores which improves performance.
4. If the array size is big then in process thing also become faster as more processes created and more CPU slices are allocated.
5. The ratios depend upon the size of input and relative overheads. For smaller size the overheads will dominate so the lesser the performance of concurrency using process and threads than normal, for bigger size performance of concurrency using process and threads better than normal.

Performance Output:

In my system the output as follows:

```
11
11 10 9 8 7 6 5 4 3 2 1
Running concurrent_mergesort for n = 11
1 2 3 4 5 6 7 8 9 10 11
time = 0.001304
Running threaded_concurrent_mergesort for n = 11
1 2 3 4 5 6 7 8 9 10 11
time = 0.000501
Running normal_mergesort for n = 11
1 2 3 4 5 6 7 8 9 10 11
time = 0.000017
normal_mergesort ran:
    [ 76.134624 ] times faster than concurrent_mergesort
    [ 29.239236 ] times faster than threaded_concurrent_mergesort


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 = 0.001841
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 = 0.002155
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 = 0.000018
normal_mergesort ran:
    [ 103.126699 ] times faster than concurrent_mergesort
    [ 120.751198 ] times faster than threaded_concurrent_mergesort

200
199 198 197 196 195 194 193 192 191 190 189 188 187 186 185 184 183 182 181
180 179 178 177 176 175 174 173 172 171 170 169 168 167 166 165 164 163 162
161 160 159 158 157 156 155 154 153 152 151 150 149 148 147 146 145 144 143
142 141 140 139 138 137 136 135 134 133 132 131 130 129 128 127 126 125 124
123 122 121 120 119 118 117 116 115 114 113 112 111 110 109 108 107 106 105
104 103 102 101 100 99 98 97 96 95 94 93 92 91 90 89 88 87 86 85 84 83 82
81 80 79 78 77 76 75 74 73 72 71 70 69 68 67 66 65 64 63 62 61 60 59 58 57
56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32
31 30 29 28 27 26 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 = 200
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 25 26 27
28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52
53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77
78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101
102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120
121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139
140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158
159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177
178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196
197 198 199
time = 0.003092
Running threaded_concurrent_mergesort for n = 200
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 25 26 27
28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52
53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77
78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101
102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120
121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139
140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158
159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177
178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196
197 198 199
time = 0.003572
Running normal_mergesort for n = 200
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 25 26 27
28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52
53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77
78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101
102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120
121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139
140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158
```

```
159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177
178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196
197 198 199
time = 0.000044
normal_mergesort ran:
    [ 70.065080 ] times faster than concurrent_mergesort
    [ 80.933198 ] times faster than threaded_concurrent_mergesort
```

Code breakdown:

threadedmergesort:

tms() is my threaded mergesort fn in that I am creating two threads one is for left half array and other is for right half array.and calling the tms()fn to split the array. I am checking whether the passing array size is greater than 5 if not then doing selectionsort . if size is greater than 5 i am initailising the struct things as arrays start index and end index and the arr and passing it to thread and the thread calls tms() and splits it and dothis process again.After creation of threads it waits for them to join back.after that merge()will be called to merge the left half and right half.

```
void *tms(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(r-l+1<=5)
    {
        selectionSort(arr,l,r);
        pthread_exit(0);
    }

    int m = l + (r - l) / 2;
    //sort left half array
    pthread_t tid1,tid2;

    struct arg a1;
    a1.l = l;
    a1.r = m ;
    a1.arr = arr;
   // pthread_t tid1;
    pthread_create(&tid1, NULL, tms, &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, tms, &a2);


    //wait for the two halves to get sorted

    pthread_join(tid1, NULL);

    pthread_join(tid2, NULL);

    merge(arr,l,m,r);
    pthread_exit(0);
    //merge(arr,l,r);
}
```

concurrent procces mergesort:

mpms() is my fn which does merge sort by creating process, In this the main process creates two childs one will take care of left half array and other will take care of right half. In child process it calls mpms() with the start and end index of array and after executing that doing exit(1) exiting child process. I am checking the array size is greater than 5 if less than 5 do selectionsort , after left and right child exits the two arrays are merged by merge().

```
void mpms(int *a,int l , int r)
{
    if (l < r)
    {
        if (r - l + 1 <= 5)
        {
            selectionSort(a, l, r);
            return;
        }

        int m = l + (r - l) / 2;
        int pid1 = fork();
        int pid2;
        if (pid1 == 0)
        {

                mpms(a, l, m);
                _exit(1);


        }
        else
        {
            pid2 = fork();
            if (pid2 == 0)
```

```
            {

                    mpms(a, m + 1, r);
                    _exit(1);



            }
            else
            {
                int status;
                waitpid(pid1, &status, 0);
                waitpid(pid2, &status, 0);
            }
        }
        merge(a, l, m, r);
        return;

    }
}
```

normal mergesort:

mergesort() is my normal merge sort fn in that I am checking that array size is less than five or not if less then doing selectionsort after that arrays are merged by merge().

```
void mergeSort(int arr[], int l, int r)
{
    if (l < r)
    {
        if (r - l + 1 <= 5)
        {
            selectionSort(arr, l, r);
            return ;
        }


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

            mergeSort(arr, l, m);


            mergeSort(arr, m + 1, r);


        merge(arr, l, m, r);

    }
}
```

merging fn does merging of the left half and right half:

```
void merge(int arr[], int l, int m, int r)
```

shareMem functions gets shared memory of the size mentioned by argument size.

```
int *shareMem(size_t size)
{
    key_t mem_key = IPC_PRIVATE; //It is the numeric key to be assigned
tothe returned shared memory segment
    int shm_id = shmget(mem_key, size, IPC_CREAT | 0666); //creating and
granting read and write access

    return (int *)shmat(shm_id, NULL, 0); //shmat() returns the address of
the attached shared memory, NULL, the system chooses a suitable(unused)
page - aligned address to attach the segment.
    //0 is read-only
}
```

All callings all from int main()