Report - Merge Sort

Normal Merge Sort runs faster than both Concurrent Merge Sort using Processes and Concurrent Merge Sort using Threads for most tests. That is for small sizes tested. Threaded might be faster for heavy computations

Conclusions

Normal Merge Sort runs quicker because of the following reasons:

1. There are overheads in creating Processes and Threads, which makes Normal Merge Sort than the other two, since it does not have these overheads.
2. There are less context switches in Normal Merge Sort, compared to the other concurrent versions with processes and threads. Context Switches slow the sort. It takes up CPU time.
3. Multithreading is creating an overhead of synchronization of the threads. So Normal and Process based Merge Sort run faster than it.
4. It takes more time also due to overheads in context switches and in handling threads which slows it down by a lot
5. Same goes for Process based Merge Sort due to more context switches.

Performance output

In my system the outputs are as follows:

```
Running Concurrent MergeSort for n = 50
-10 -9 -9 -8 -8 -8 -7 -7 -6 -6 -5 -5 -4 -4 -3 -3 -2 -2 -1 -1 1 1 2 2 3 3 4
4 5 5 6 6 6 7 7 7 8 8 8 9 9 9 10 10 10 46 47 48 49 50
time = 0.001070
Running Threaded MergeSort for n = 50
-10 -9 -9 -8 -8 -8 -7 -7 -6 -6 -5 -5 -4 -4 -3 -3 -2 -2 -1 -1 1 1 2 2 3 3 4
4 5 5 6 6 6 7 7 7 8 8 8 9 9 9 10 10 10 46 47 48 49 50
time = 0.001343
Running Normal MergeSort for n = 50
-10 -9 -9 -8 -8 -8 -7 -7 -6 -6 -5 -5 -4 -4 -3 -3 -2 -2 -1 -1 1 1 2 2 3 3 4
4 5 5 6 6 6 7 7 7 8 8 8 9 9 9 10 10 10 46 47 48 49 50
time = 0.000016
Normal Merge Sort ran:
        [ 67.545122 ] times faster than Concurrent Merge Sort
        [ 84.788885 ] times faster than Threaded Merge Sort

Running Concurrent MergeSort for n = 200
0 2 7 8 26 30 33 39 45 49 53 55 58 59 69 87 91 94 99 101 117 121 123 132
136 140 142 147 149 151 155 158 175 182 183 187 189 190..
time = 0.002982
Running Threaded MergeSort for n = 200
0 2 7 8 26 30 33 39 45 49 53 55 58 59 69 87 91 94 99 101 117 121 123 132
136 140 142 147 149 151 155 158 175 182 183 187 189 190..
time = 0.003126
Running Normal MergeSort for n = 200
```

```
0 2 7 8 26 30 33 39 45 49 53 55 58 59 69 87 91 94 99 101 117 121 123 132
136 140 142 147 149 151 155 158 175 182 183 187 189 190..
time = 0.000047
Normal Merge Sort ran:
        [ 64.120789 ] times faster than Concurrent Merge Sort
        [ 67.220120 ] times faster than Threaded Merge Sort


Running Concurrent MergeSort for n = 500
0 5 6 9 10 12 13 14 16 17 20 21 26 28 29 31 37 38 39 40 43 45 46 48 49 50
52 61 63 67 69 71 72 78 79 84 86 87 88 90 91 93 94 95 96..
time = 0.010866
Running Threaded MergeSort for n = 500
0 5 6 9 10 12 13 14 16 17 20 21 26 28 29 31 37 38 39 40 43 45 46 48 49 50
52 61 63 67 69 71 72 78 79 84 86 87 88 90 91 93 94 95 96..
time = 0.008830
Running Normal MergeSort for n = 500
0 5 6 9 10 12 13 14 16 17 20 21 26 28 29 31 37 38 39 40 43 45 46 48 49 50
52 61 63 67 69 71 72 78 79 84 86 87 88 90 91 93 94 95 96..
time = 0.000115
Normal Merge Sort ran:
        [ 94.435861 ] times faster than Concurrent Merge Sort
        [ 76.743495 ] times faster than Threaded Merge Sort



Running Concurrent MergeSort for n = 1000
33 38 55 68 105 110 116 137 141 163 179 199 213 276 310 319 324 334 350 390
402 417 418 441 449 452 469 472 475 521 527 591 599 607..
time = 0.030103
Running Threaded MergeSort for n = 1000
33 38 55 68 105 110 116 137 141 163 179 199 213 276 310 319 324 334 350 390
402 417 418 441 449 452 469 472 475 521 527 591 599 607..
time = 0.021973
Running Normal MergeSort for n = 1000
33 38 55 68 105 110 116 137 141 163 179 199 213 276 310 319 324 334 350 390
402 417 418 441 449 452 469 472 475 521 527 591 599 607..
time = 0.000255
Normal Merge Sort ran:
        [ 118.045846 ] times faster than Concurrent Merge Sort
        [ 86.167509 ] times faster than Threaded Merge Sort
```

Code Break-Down

processMergeSort function does Merge sort with forking. 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 calls processMergeSort on left half and right child calls processMergeSort on right half. Now the parent waits for both process to finish and in the parent the arrays are merged back. The comments in the code explain each step in detail.

```
void processMergeSort(int A[],int l, int r){

    if(r-l > threshhold){  //if array size greater than threshold do
mergesort
        int mid= (r+l)/2;


        merge(A,l,mid,r);
        pid_t left,right; //process id of left and right process
        left=fork(); //fork a left child
        if(left==-1){  //error handling

            perror("Fork:");
            _exit(-1);

        }
        else if(left==0){
            //this is left child, this process will sort left side
            processMergeSort(A,l,mid);
            _exit(0); //exit after processing
        }
        else{
            //this is parent,
            right=fork(); // make a right child
            if(right==-1){
                perror("Fork right:");
                _exit(-1);

            }
            else if(right==0){
                //this is right child, process right side
                processMergeSort(A,mid+1,r);
                _exit(0); //exit after doing it
            }
        }
        //this is parent, wait for both of them to complete...
        signed status;
        waitpid(left,&status,0);
        waitpid(right,&status,0);
        //merge array after both of them complete
        merge(A,l,mid,r);
    }
    else{ //else do selection sort

        selectionSort(A,l,r);

    }

}
```

threadedMergeSort function does the threaded Merge Sort. It takes arguments as structure which is self-explanatory. 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. Then it waits for them to join back. Then we call merge. The comments in code explain each step.

```
void *threadedMergeSort(void* a){

    struct arg *args = (struct arg*) a; //arguments passed as structure
    int l = args->l; //left index
    int r = args->r; //right index
    int *A = args->arr; //array to sort
    if(r-l > threshhold){ // if array size greater than threshold
        int mid= (r+l)/2;
        struct arg leftargs, rightargs; //args to right thread and left
thread
        pthread_t ltid,rtid; //left thread, right thread
        leftargs.arr=A;
        leftargs.l=l;
        leftargs.r=mid;
        rightargs.l=(mid+1);
        rightargs.arr=A;
        rightargs.r=r;
        pthread_create(&ltid,NULL,threadedMergeSort,&leftargs); //sort left
side in left thread
        pthread_create(&rtid,NULL,threadedMergeSort,&rightargs);//sort
right side in right thread
        pthread_join(rtid,NULL);//join them, that is wait for them to
complete
        pthread_join(ltid,NULL);
        merge(A,l,mid,r);
        pthread_exit(0); //exit this thread


    }
    else{
        selectionSort(A,l,r);
        pthread_exit(0);//exit this thread
    }
}
```

This function does normal merge sort but if array size is 5 or less it switches to selection sort. The code is self-explanatory.

```
//the normal mergesort function - with selection sort when array size
becomes less than or equal to threshold
void normalMergeSort(int A[], int l, int r){
    if(r-l > threshhold){
        int mid= (r+l)/2;
        normalMergeSort(A,l,mid);
```

```
        normalMergeSort(A,mid+1,r);
        merge(A,l,mid,r);
    }
    else{
        selectionSort(A,l,r);
    }
}
```

This function is the Merging Function, nothing special happens here.

```
void merge(int A[],int l, int mid, int r)
```

The runSort function is called from main() and calls all the three types of sorts and times them and compares them it also creates three arrays for sorting. The comments in the code explain this function.

```
void runSort(int n)
```

shareMem functions gets shared memory of the size mentioned by argument size. The comments explain each step.

```
//gets shared memory
int * shareMem(size_t size){
    key_t mem_key = IPC_PRIVATE; //It is the numeric key to be assigned to
the 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
}
```

The selection sort is done by the following function. Nothing special there too.

```
void selectionSort( int A[], int l, int r )
```

The main calls runsorts function