Multithreaded Sorting

Topic - Implementation and analysis of multithreaded sorting where multiple threads are used to sort segments of an array and these segments are merged (with and without parallelism)

Submitted By: Kushagra Indurkhya
CS19B1017

Contents

Implementation Details

As stated in the problem statement to study the efficiency of using multithreading in performing sorting of arrays i have implemented 2 multithreaded solutions which take (n,p) as input where 2^n is the length of array and 2^p is the number of threads.

Multithreaded sorting of Segments of Array

➢ The main thread(master) creates 2^p slave threads using `pthread_create` each sorting a segment of length 2^(n-p) of the array

```
for (int i = 0; i < no_of_threads; ++i)

pthread_create(&thread_ids[i],NULL,sort_seg,(void*)&shared_data[i]);
```

Thread Creation

The following attributes are passed in `pthread_create` call :

- `&thread_ids[i]`-thread_ids is an array of type pthread_t and length 2^p to store the IDs of all the threads created.
- `NULL` - to use the default attributes
- `Sort_seg` - function calls the sort function which is implementation of standard merge sort algorithm called on the segment of array in range(start,start+seg_size) and exits with NULL value when the control returns after sorting the segment.

```
void* sort_seg(void* seg_start)
{
    share* data=(share*)seg_start;
    int start=data->start;
    long int* arr=data->arr;
    int seg_size=data->seg;
```

```
    int len=data->len;

    sort(arr,start,start+seg_size-1,len);
    pthread_exit(NULL);

}
```

This function in turn calls the `sort` function which is a recursive merge sort implementation which takes array and left and right indices of the sub array to be sorted , the `merge` function merges 2 sorted subarrays boundary indices of which are passed as arguments to it:

```
void merge(int a,int b,int c,int d,long int* arr)
{
    int temp[d-a+1];
    int i=a,j=c,k=0;
    while(i<=b && j<=d)
    {
        if (arr[i]<arr[j])
            temp[k++]=arr[i++];
        else
            temp[k++]=arr[j++];
    }
    while (i<=b)
        temp[k++]=arr[i++];
    while (j<=d)
        temp[k++]=arr[j++];

    for (int i = a,j=0; i <=d; i++,j++)
        arr[i]=temp[j];
}
```

```
void sort(long int*arr,int l,int r)
{
    if(l>=r)
        return;
```

```
    int m = (l+r-1)/2;
    sort(arr,l,m);
    sort(arr,m+1,r,len);
    merge(l,m,m+1,r,arr,len);

}
```

- `&shared_data[i]-` is the argument passed to the sort_seg function which is a pointer to the i'th element of shared_sata array of a struct data typecasted to a void pointer, the composition of the struct is :

```
long int* arr -> pointer to the array of random long ints
int start -> start index of the segment
int seg     ->  length of the segment (2^(n-p))
```

➢ After all the threads are created and with each thread sorting a segment of the array , the main(master) thread wait for all the slave to terminate this is achieved by using `pthread_join()`

```
    for (int i = 0; i < no_of_threads; i++)
        pthread_join(thread_ids[i], NULL);
```

Here we wait for every thread ID in thread_ids array to finish

Now `arr` has its segments (length=2^(n-p)) sorted, now we merge those segments in two different ways:

Method-1: Sequential merging in main thread

We will use main thread to merge our segments in a sequential manner for example if `no_of_threads=8` and there are 8 segments numbered 1 to 8 then the merging will proceed as follows:

Phase-1	(1)(2)
Phase-2	(1,2)(3)
Phase-3	(1,2,3)(4)
Phase-4	(1,2,3,4)(5)
Phase-5	(1,2,3,4,5)(6)
Phase-6	(1,2,3,4,5,6)(7)
Phase-7	(1,2,3,4,5,6,7,8)

(Segments in '()' are already merged)

This is my implementation of merging two sorted subarrays of length p,q which has both time and space complexity of O(p+q)

```
void merge(int a,int b,int c,int d,long int* arr,int len)
{
    int temp[d-a+1];
    int i=a,j=c,k=0;
    while(i<=b && j<=d)
    {
        if (arr[i]<arr[j])
            temp[k++]=arr[i++];
        else
            temp[k++]=arr[j++];
    }
    while (i<=b)
        temp[k++]=arr[i++];
    while (j<=d)
        temp[k++]=arr[j++];
```

```
    for (int i = a,j=0; i <=d; i++,j++)
        arr[i]=temp[j];
}
```

To merge the segments in the manner mentioned above i have called the merge function in the following manner:

```
for (int i = 1; i < no_of_threads; i++)

merge(0,(i*seg_size)-1,i*seg_size,(i*seg_size)-1+seg_size,&arr[0],le
n);
```

After (2^p)-1 iterations all the sorted segments are merged and the whole array is sorted

Method-2: Multithreaded Merging

In method-2 i have taken a multithreaded approach in merging the sorted segments of the array in such way that the main thread creates `no_of_threads/2` slave threads where each thread merge two segments and exits then in the next iteration `no_of_threads/4` slave threads are created to merge the merge the merged segments and so on until only one segment remains.
For example if `no_of_threads=8` and there are 8 segments numbered 1 to 8 and corresponding threads numbered (0 to 7) then the merging will proceed as follows:

Phase-1:	1,2	3,4	5,6	7,8
Phase-2:	1,2,3,4		5,6,7,8	
Phase-3:	1,2,3,4,5,6,7,8			

(Different colors indicate different threads)
Thread-0, **Thread-2**, **Thread-4**, **Thread-6**

To create the slave threads to merge the segments in the manner mentioned above i have used `pthread_create` called the in the following manner:

```
1.     merge_thread_data data[no_of_threads];
2.     for (int i = 1; 2*i<=no_of_threads; i*=2)
3.     {
4.         int j=0;
5.         for (int thread_idx = 0; thread_idx < no_of_threads ; thread_idx++)
6.         {
7.             if (thread_idx%(2*i)==0)
8.             {
9.                  data[thread_idx].arr=&arr[0];
10.                 data[thread_idx].arr_len=len;
11.                 data[thread_idx].phase=i;
12.                 data[thread_idx].seg_size=seg_size;
13.                 data[thread_idx].seg_idx=thread_idx;
14. pthread_create(&thread_ids[j],NULL,merge_thread,(void*)&data[thread_idx]);
15.                 j++;
16.             }
17.         }
18.         for (int k = 0; k <j; k++)
19.             pthread_join(thread_ids[k], NULL);
20.     }
```

Here the control variable of outer loop (line 2 to 20) is multiplied by 2 every time it indicates the no of consecutive array segments which are currently merged,in case of the given example this runs for 3 times(3 phases)

Then in the inner loop we check if the segment corresponding to i needs to be processed in this iteration by taking its modulus with 2*i.
(To prevent the time lost in array creation i have reused the same thread array used during sorting).
Variable j is incremented each time a slave thread is created

The following attributes are passed in `pthread_create` call :

- `&thread_ids[i]`-thread_ids is an array of type pthread_t and length 2^p to store the IDs of all the threads created.
- `NULL` - to use the default attributes
- `Merge_thread –` Calls the merge function (same as that of method-1) with corresponding subarray's starting and ending indices And exits after they are merged

```
void* merge_thread(void* shared_data)
{
    merge_thread_data* data=(merge_thread_data*) shared_data;
    long int* arr=data->arr;
    int len=data->arr_len;
    int phase=data->phase;
    int seg_size=data->seg_size;
    int seg_idx=data->seg_idx;

    int a=seg_idx*seg_size;
    int b=a+(phase*seg_size)-1;
    int c=b+1;
    int d=c+(phase*seg_size)-1;

    merge(a,b,c,d,arr,len);

    pthread_exit(NULL);
}
```

- `&data[thread_idx]`–is the argument passed to the merge_thread function which is a pointer to the element of data array of merge_data struct typecasted to a void pointer, the composition of the struct is :

```
struct merge_data
{
long int* arr    -> pointer to the array of random long ints
int seg_idx      -> start index of the segment
int seg_size     -> length of the segment (2^(n-p))
int arr_len      -> length of the array (2^n)
Int phase        -> i ie the phase of merging
}
```

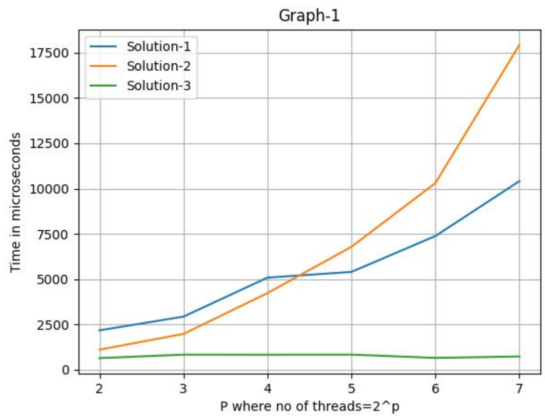
After the threads are created in the inner loop before exiting the outer loop all the created threads are joined (line-18,19).

In this way all the sorted segments are merged.

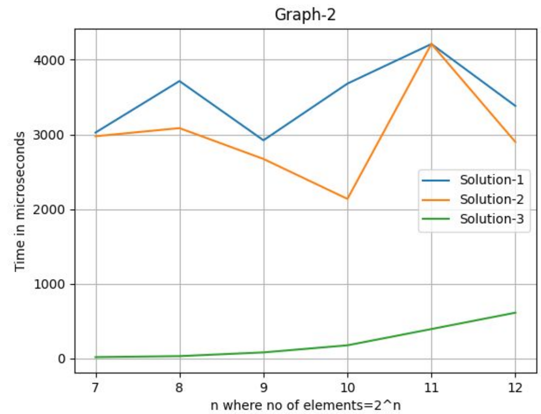
Time Graph Analysis

To analyse the efficiency of the above described methods , running three solutions (Solution-1 Method-1, Solution-2 Method-2, Solution-3 is the plain usage of merge sort algorithm) with varying n,p gave the following results:

```
stark@Jarvis:/mnt/d/WSL/os_assignment/submission$ ./visualize.sh
✓ Compiled
✓ Inputs Generated
                              Graph-1
(n p)    SOl-1                SOL-2                 SOL-3
(12 2)   2179    SORTED       1112    SORTED        646     SORTED
(12 3)   2935    SORTED       1981    SORTED        836     SORTED
(12 4)   5087    SORTED       4233    SORTED        832     SORTED
(12 5)   5403    SORTED       6788    SORTED        838     SORTED
(12 6)   7384    SORTED       10305   SORTED        657     SORTED
(12 7)   10412   SORTED       17932   SORTED        736     SORTED
                              Graph-2
(n p)    SOl-1                SOL-2                 SOL-3
(7 4)    3025    SORTED       2976    SORTED        18      SORTED
(8 4)    3715    SORTED       3085    SORTED        31      SORTED
(9 4)    2922    SORTED       2672    SORTED        81      SORTED
(10 4)   3681    SORTED       2137    SORTED        176     SORTED
(11 4)   4211    SORTED       4213    SORTED        394     SORTED
(12 4)   3382    SORTED       2899    SORTED        613     SORTED
✓ Graphs Plotted
✓ Cleanup
Script Exited
```



Graph-1 Varying p along x-axis while keeping n fixed to 14



Graph-2 Varying n along x-axis while keeping p fixed to 4

Graph-1

Observations:

1. Time taken by Both solution-1 and solution-2 increase as we increase the number of threads
2. Solution-2 performs better than Solution-1 for p=2,3,4.
3. Solution-3 performs better than both solutions-1 and 2.

Explanations:

1. As we increase the number of threads the ,the thread overhead (caused by thread creation, deletion, scheduling, management and context switches) is more than the time saved by the use of multithreading in sorting the segments
2. Initially when no of threads is less the overhead caused by thread creation , deletion and context switches (in merging) is compensated by the speedup provided by the multithreading, As we increase the number of threads the time taken by the thread overhead is more than the time saved by the use of multithreading in merging the segments.
3. In a relatively small array the thread overhead is much more than the speedup provided by multiple threads(both in sorting and sorting+merging) hence plain merge sort performs better than both.

(Note- observation-2,3 are corresponding to the graph above , its not true for every execution as many other factors also control the running time)

Graph-2

Observations:

1. Solution-2 in general performs better than Solution-1.
2. Solution-3 performs better than both solutions-1 and 2.
3. The trends in solution-1 & 2 are erratic

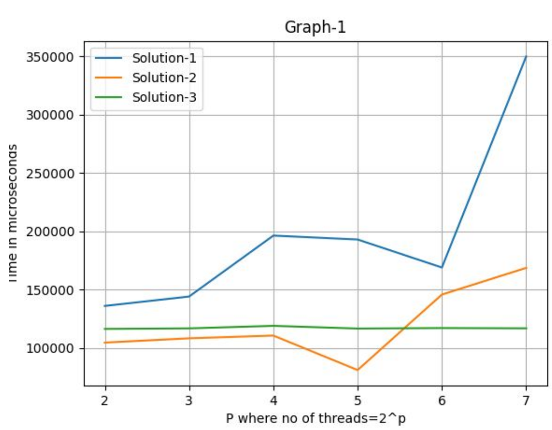
Explanations:

1. As we increase the number of elements the work given to each thread is increases (large segments merging) hence the thread overhead is compensated and justified also the no of user threads fixed here is not very large than the no of kernel threads available hence parallelism is used aptly without unnecessary thread creation, management, and scheduling overhead while in solution 1 multithreading is used only in sorting and no parallelism is used for merging hence solution 2 performs better than solution 1.
2. In the range of n from 7 to 12 no of elements are from 128 to 4096 which is relatively a small size here the thread management overhead cannot be compensated by the speedup provided by multithreading (both in sorting and sorting+merging) hence plain merge sort performs better than both.
3. The trends in solution-1 and solution-2 are erratic as the time taken by both the solutions is dependent on various other factors such as the availability of cpu resources, execution time of threads and other processes in the queue.

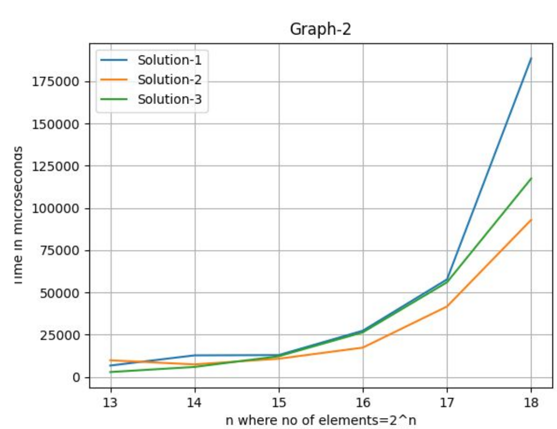
Exploring further

Checking on larger values of n:

```
stark@Jarvis:/mnt/d/WSL/os_assignment/submission$ ./visualize.sh
✓ Compiled
✓ Inputs Generated
                                Graph-1
(n p)   SOl-1                   SOL-2                   SOL-3
(18 2)  136042  SORTED          104628  SORTED          116357  SORTED
(18 3)  144101  SORTED          108285  SORTED          116822  SORTED
(18 4)  196365  SORTED          110637  SORTED          118975  SORTED
(18 5)  192995  SORTED          81076   SORTED          116669  SORTED
(18 6)  168995  SORTED          145792  SORTED          117090  SORTED
(18 7)  349977  SORTED          168669  SORTED          116861  SORTED
                                Graph-2
(n p)   SOl-1                   SOL-2                   SOL-3
(13 4)  6805    SORTED          9911    SORTED          2964    SORTED
(14 4)  12822   SORTED          7487    SORTED          5979    SORTED
(15 4)  12991   SORTED          10843   SORTED          12294   SORTED
(16 4)  27424   SORTED          17411   SORTED          26455   SORTED
(17 4)  57769   SORTED          41735   SORTED          56063   SORTED
(18 4)  188471  SORTED          92993   SORTED          117437  SORTED
✓ Graphs Plotted
✓ Cleanup
Script Exited
```



Graph-1 Varying p along x-axis while keeping n fixed to 18



Graph-2 Varying n along x-axis while keeping p fixed to 4

- ➢ In graph-1 we observe that solution-2 is performing better than plain mergesort and solution-1 when p is varying from 2 to 5.
- ➢ In graph-2 we observe that solution-2 is performing better than both solution-1 & 3 for n ranging from 15-18.

Conclusion

After running all 3 solutions on different sets of values various times i conclude the following:

- ➢ For smaller arrays (approximately of size <= 20000) using simple merge sort without parallelism is efficient.
- ➢ For large arrays (order of 10^5) using multithreading for sorting as well as merging (Solution-2) is most efficient.
- ➢ For small arrays, if using solution-1 or 2 the value of p should be kept minimum - {1,2} so that the number of threads are {2,4},this can sometimes give even better performance than plain mergesort.
- ➢ For large arrays value of p can be kept around {4,5} for optimal performance.