Multi-threaded Merge Sort Analysis

Index:

An Overview Of The Design

The program has 2 main parts, namely Input analysis, which gives a sorted output file on running and Time analysis which is meant to calculate times for both Multithreaded and single threaded sorting algorithms. Input analysis accepts input form the user and then calls the multithreaded algorithms. Time Analysis does not do the same however and will simply give output with time.

STRUCTURE OF THE PROGRAM:

1. Input analysis
    a. Take Input and store in array
    b. Create Threads and assign intervals
    c. Threads will start merge -sorting themselves
    d. Merge sorted threads
    e. Give Output
2. Time Analysis
    a. Create Random Array of random size
    b. Define Threads/Size according to choice
    c. Call time efficient multithreaded algorithm
    d. Call time efficient merge sorted threads
    e. Print Output.

VERY IMPORTANT REMARKS

Remark 1:

The array must be defined on heap, because the array has to be shared amongst all the threads(its a common memory for all of them).

Remark 2:

The input analysis function is the one which will take input and sort. Time analysis will create a random array using rand() function and print results.

Remark 3:

There are many functions in the code for printing arrays etc. They are trivial and not analyzed in this report. The merge sort algorithm is given under "over view of merge sort " please read it to understand the structure of the program.

Remark 4:

To run the Time analysis please uncomment the part pertaining to time analysis and comment the code pertaining to input analysis() in the source file and then compile.

AN OVERVIEW OF MERGE SORT

Sequential Merge Sort:

The algorithm breaks the array into 2 intervals, centered at the midpoint, and then recurses forward. Once the subarrays have been sorted, they are merged using a merge function

```
void sequential_merge_sort(double *a ,unsigned int low,unsigned int high)
{
	unsigned int mid = (low+high)/2;

	if(low<high)
	{
		sequential_merge_sort(a,low,mid);
		sequential_merge_sort(a,mid+1,high);
		sub_array_merge(a,low,mid,high);
	}
}
```

Merging of The Subarrays:

Remove minimum of top of left and right subarray to merge them into one array

```
void sub_array_merge(double *a, unsigned int low,unsigned int mid ,unsigned int high)
{
	unsigned int size_left = mid - low + 1;
	unsigned int size_right = high - mid;
	double *left = (double *)malloc((size_left + 1)*sizeof(double));
	double *right = (double *)malloc((size_right +1)*sizeof(double));

	for (int i = 0 ; i<size_left; i++)
	{
		left[i] = a[low + i];
	}
	for( int i = 0 ; i<size_right; i++)
	{
		right[i] = a[mid + i + 1];
	}
	left[size_left] = (double)(INT_MAX); //sentinel card(so that its not pulled out)
	right[size_right] = (double)(INT_MAX);//sentinel is a maximum card at the end
	int i =0;
```

```
        int j =0;
        for (int k = low ; k <= high ; k++)
        {
            if(left[i] <= right[j])
            {           a[k] = left[i]; //removes top of left
                        i = i+1;
               }
               else
               {        a[k] = right[j];//removes top of right
                        j = j + 1;
               }
        }
}
```

INPUT ANALYSIS:

a. Important Variables.
   i. The Struct Parameters and the Set Parameters Function
   ii. Step -The size of each array in the Thread
b. The Creation of Threads and Their Ending.
c. The Thread Itself.
d. Merge-Sorted Subarrays
e. Merge Solutions to Multiple Threads
f. Output Solution to File

IMPORTANT VARIABLES:

STRUCT PARAMETERS AND SET PARAMETERS FUNCTION

The struct parameters is used to pass arguments to the thread! This is because a thread usually takes only a single argument! But for the thread to take multiple arguments we need to wrap all the arguments into a struct. The arguments will include the array itself , starting point of the part of the array which the thread has to handle along with the thread number and size of the array.

The set parameters function merely sets these parameters according to the scenario.

Struct : Parameters

```
typedef struct
{
        double *array;
        unsigned int index_low;
        unsigned int index_high;
        unsigned int size;
```

```
        unsigned int count;

}parameters;
```

Function Set Parameters:

```
void set_parameters(parameters *par,double *a,unsigned int low,unsigned int high,unsigned
int size,unsigned int count)
{
          par->array = a;
          par->size = size;
          par->index_low = low;
          par->index_high = high;
          par->count = count;
}
```

STEP-THE SIZE OF EACH THREAD

The array will be evenly divided amongst all the threads needed. The exception is however when the number of threads does not divide the total array, in this case the floor of division is taken and the EXTRA PART IS ADDED TO THE LAST THREAD OF EXECUTION.
Code:

```
int step = size/(number_of_threads);
```

Extra Part Handling(final low describes the low index of the final thread)

```
int step = size/(number_of_threads);
int final_low = step*(number_of_threads-1);
int final_high = size-1;
```

THE CREATION OF THREADS :

The thread itself will take the arguments of a particular interval of values and then it will merge sort its particular interval.

Code:

```
for(unsigned int j = 0; j<size ; j+=step)
                        {
                parameters *par = (parameters *)malloc(sizeof(parameters));

               set_parameters(par,a,j,j+step-1,size,count);

pthread_create(&thread[count],NULL,sub_array_sort,(void*)par);
                                count = count +1;}
```

The thread then terminates using the join statement after the completion of the program using the following code :

```
for (int i =0 ; i<number_of_threads ; i++)
     {
              pthread_join(thread[i],NULL);
     }
```

THE THREAD ITSELF:

The thread itself performs a sequential merge on itself. That, is the thread is recursive in nature, it calls itself with smaller parameters and then builds the subarrays one by one. Each thread Performs a merge sort on its own subarray. To do that it calls the merge -subarray function(defined after this)

Code:

```
void *sub_array_sort(void * param)
{

     parameters* par = (parameters*)param;
     unsigned int low = par->index_low;
     unsigned int high = par->index_high;
     unsigned int size = par->size;
     unsigned int count = par->count;
     double * a = par->array;



     //now its time to call the merge sort function to sort the array !
     if(low < high){
               unsigned int mid = ((low + high)/2);
               parameters *left_part = (parameters*)malloc(sizeof(parameters));
               parameters *right_part = (parameters*)malloc(sizeof(parameters));

            set_parameters(left_part,a,low,mid,size,count);

            sub_array_sort((void *)left_part);

            set_parameters(right_part,a,mid+1,high,size,count);

            sub_array_sort((void *)right_part);

            sub_array_merge(a,low,mid,high);


        }

return NULL;
}
```

THE MERGING OF SORTED SUBARRAYS

The merging of sorted Subarrays is simple. Pick the minimum of the first 2 elements of the arrays and then place them in the final array. This is implemented in the merge -subarray function

Code:

```
void sub_array_merge(double *a, unsigned int low,unsigned int mid ,unsigned int high)
{

	unsigned int size_left = mid - low + 1;
	unsigned int size_right = high - mid;

	double *left = (double *)malloc((size_left + 1)*sizeof(double));
	double *right = (double *)malloc((size_right +1)*sizeof(double));

	for (int i = 0 ; i<size_left; i++)
	{
		left[i] = a[low + i];
	}
	for( int i = 0 ; i<size_right; i++)
	{
		right[i] = a[mid + i + 1];
	}



	left[size_left] = (double)(INT_MAX);
	right[size_right] = (double)(INT_MAX);
	int i =0;
	int j =0;

	for (int k = low ; k <= high ; k++)
	{
	    if(left[i] <= right[j])
	    {
			a[k] = left[i];
			i = i+1;
		}
		else
		{
			a[k] = right[j];
			j = j + 1;
		}
	}

}
```

THE MERGING OF SORTED THREADS :

As you can expect this version of merging is simple and just merges the sorted threads using the merge subarray function defined previously

Code:

```
void merge_sorted_threads(double *a , unsigned int size, int number_of_threads)
{
	int step = size/number_of_threads;

	int low =0;
	int high = 0;
	int mid =0;
	int cnt = 1;
```

```
        while(1)
        {
           high =(cnt+1)*step -1;
           low = 0;
           mid = cnt*step -1;
           if(high>=size)
           {break;}
           sub_array_merge(a,low,mid,high);
           cnt = cnt +1;
        }


}
```

OUTPUT TO FILE:

The file named SortedArray.txt will store the sorted Array. Simple file handling using file-pointers is enough to achieve this.

```
    FILE *f;
    f = fopen("SortedArray.txt","w");
    fprintf(f,"SORTED ARRAY \n");
    for (int i =0 ;i<size; i++)
    {
              fprintf(f,"%.2f ",a[i]);
       }
       fclose(f);
```

TIME ANALYSIS:

This assumes that the number of threads divides the total size of the array and then performs the SAME MERGING ALGORITHM , here the only difference is a few print statements and the usage of the clock() statement to find the time required.

7. Time Analysis
    iv. Calculating Time using Clock function
    v. Optimized Multithreaded Merge-Sort
    vi. Optimized Merge-Sorted Threads.

CALCULATING TIME USING CLOCK FUNCTION:

The clock() function can calculate time at a particular instant and then we have to subtract at appropriate positions and divide by clocks per sec to find the value in seconds.

Code:

```
                    start_seq = clock();
```

```
                sequential_merge_sort(b,0,size-1);
                end_seq = clock();
                printf("Threads = %d | ",num_of_threads);
                printf("Size = %d ",size);

        double time_spent_seq = ((double)(end_seq - start_seq))/CLOCKS_PER_SEC;
                printf("| SEQUENTIAL TIME : %lf ",time_spent_seq);

                start = clock();
                multithreaded_merge_time_only(a,size,thread,num_of_threads);
                end = clock();

                double time_spent = ((double)(end -start))/CLOCKS_PER_SEC;
                printf("| TIME SPENT MUTLITHREADED |: %lf \n",time_spent);
                count = count +1;
```

OPTIMIZED MULTTHREADED MERGE SORT:

The only change here is that the merge sort does not take cases where number of threads does not divide the number of elements in the array. EVERYTHING ELSE IS THE SAME.This reduces the case checking and final merging problems.

Code:

```
void multithreaded_merge_time_only(double *a ,unsigned int size , pthread_t *thread , int
number_of_threads)
{

       if(size%number_of_threads ==0)
    {

                        int step = size/(number_of_threads);

                        int count = 0;


                        for(unsigned int j = 0; j<size ; j+=step)
                        {
                        parameters *par = (parameters *)malloc(sizeof(parameters));

                                set_parameters(par,a,j,j+step-1,size,count);

pthread_create(&thread[count],NULL,sub_array_sort,(void*)par);
                                count = count +1;
                        }

                        for (int i =0 ; i<number_of_threads ; i++)
                        {
                                pthread_join(thread[i],NULL);
                        }

                        merge_sorted_threads_time_only(a,size,number_of_threads);
```

```
	}
}
```

MERGE SORTED THREADS (TIME EFFICIENT)

This is basically merging sorted threads by forming pairs and then recursively calling the function again.This is much faster than merging sequentially which was done in the merge-sorted threads earlier.

Code:

```
void merge_sorted_threads_time_only(double *a , unsigned int size, int number_of_threads)
{
	int step = size/(number_of_threads);
	if(number_of_threads <=1)
	{
		return;
	}
	if(number_of_threads%2 ==0)
	{
		int low = 0;
		int high = 0;
		int mid = 0;
		int cnt  = 1;
		    while(1)
		    {
			high = (cnt+1)*step -1;
			mid = cnt*step -1;
			low = (cnt -1)*step;
			if(high>=size)
			{
				break;
			}
		       sub_array_merge(a,low,mid,high);
		       cnt = cnt +1;
		     }
		number_of_threads = number_of_threads/2;
		merge_sorted_threads_time_only(a,size,number_of_threads);
```

```
    }
}
```

GRAPH AND PLOT ANALYSIS

THREAD ANALYSIS

We will now analyze the time taken for execution using Plots across number of threads used and several cycles of the CPU. The plot for single threaded has a certain variance (as it should) because the CPU's availability changes.

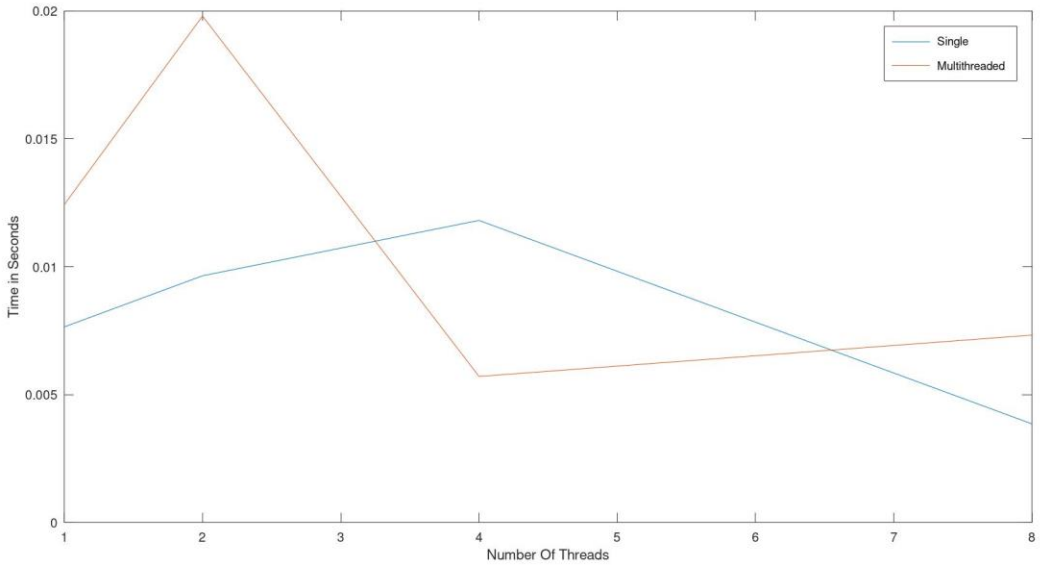
Output Data:

Threads = 1 | SEQUENTIAL TIME: 0.007646 || TIME SPENT MUTLITHREADED |: 0.012432

Threads = 2 | SEQUENTIAL TIME: 0.009646 || TIME SPENT MUTLITHREADED |: 0.019803

Threads = 4 | SEQUENTIAL TIME: 0.011812 || TIME SPENT MUTLITHREADED |: 0.005709

Threads = 8 | SEQUENTIAL TIME: 0.003854 || TIME SPENT MUTLITHREADED |: 0.007325



Plot Analysis:

The Multi -threaded algorithm is performing its best at number of threads = 4. Where it beats the single threaded speed by its maximum margin. This has happened partly because the threads are running in parallel and because the threads have been executed within less time.

SIZE ANALYSIS:

The Number of threads used is 16. (Slightly larger than what is preferred for the computer. Best performance comes at threads =6).

OUTPUT DATA:

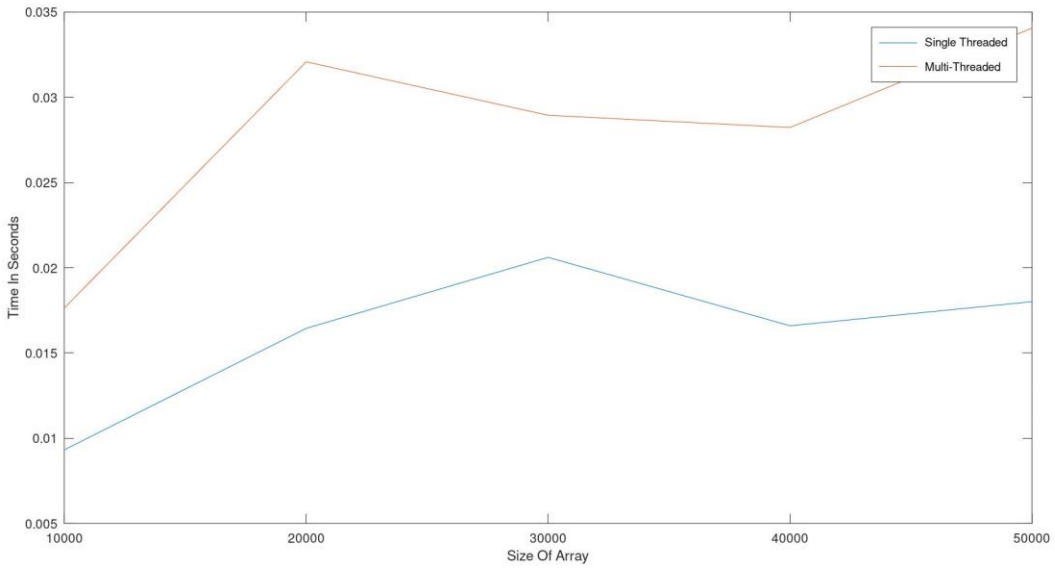
Threads = 16 | Size = 10000 | SEQUENTIAL TIME : 0.009316 | TIME SPENT MUTLITHREADED |: 0.017635

Threads = 16 | Size = 20000 | SEQUENTIAL TIME : 0.016444 | TIME SPENT MUTLITHREADED |: 0.032080

Threads = 16 | Size = 30000 | SEQUENTIAL TIME : 0.020610 | TIME SPENT MUTLITHREADED |: 0.028944

Threads = 16 | Size = 40000 | SEQUENTIAL TIME : 0.016594 | TIME SPENT MUTLITHREADED |: 0.028227

Threads = 16 | Size = 50000 | SEQUENTIAL TIME : 0.018007 | TIME SPENT MUTLITHREADED |: 0.034059



Plot Analysis:

The Number of Threads staying constant, clearly as the size of the array is going to increase, so does the time for execution. The margin between multithreaded and Single threaded clearly has a minimum again at size = 30000. The overhead for 16 threads continues to increase later. 16 threads does not give as good performance as 6 or even 4 threads.