Vinta Reethu
Operating Systems
Indian Institute of Technology Hyderabad
Assignment-1

Programming Assignment-1

Vinta Reethu - ES18BTECH11028

February 1, 2021

Question :

Perform sorting of numbers efficiently using multiple threads.

Working details :

Header files :

```
#include <stdio.h>              // Importing libraries
#include <stdlib.h>
#include <math.h>
#include <pthread.h>
#include <time.h>
```

These are the header files that are added in the written c files. These are necessary for us to perform the operations in this code.

Global declarations :

```
int *A;          // Declaring array globally
int p,n;         // p denotes the number of threads, n denotes the number of
    elements present in the array
```

- These are the global variables that accessed in some of the functions.

Common Functions used in both solutions code :

```
// Function that swap the values present at i'th,j'th indices
void swap(int i,int j)
{
    int temp;
    temp=A[i];
    A[i]=A[j];
    A[j]=temp;
}
```

भारतीय प्रौद्योगिकी संस्थान हैदराबाद
Indian Institute of Technology Hyderabad

- This function is used to swap the values at i'th, j'th indices.

```
int partition(int low,int high)// Function the find the partition
{
    int temp;
    int pivot = A[high];        // Pivot is the high'th element
    int i = low-1;
    for(int j=low;j<high;j++)  // Iterate from low to high
    {
        if(A[j]<pivot)          // If value is less that pivot then swap with i
        {
            i++;
            swap(i,j);          // Swapping
        }
    }
    swap(i+1,high);             // Swapping
    return i+1;                 // Return the partion indice
}
```

- In this function we take take pivot as the A[high], and put pivot at its correct position in the sorted array and put all smaller elements to the left of pivot, and put all greater elements to the right of pivot.

```
void quicksort(int low,int high)  // Function to perfrom quick sort
{
    if(low<high)
    {
        int p=partition(low,high);// Find the partition index
        quicksort(low,p-1);        // Now sort the left portion of the partition
    index
        quicksort(p+1,high);       // Sort the right portion of the partition index
    }
}
```

- This function performs quicksort using the partition function on array **'A'**.

Method 1 code :

```
void merge(int low,int high,int split)// Perform to merge arrays
{
    int left[split-low+1];              // Array to store the values from low to
    split
    int right[high-split];              // Array to store the values from split to
    high

    for(int i=0;i<split-low+1;i++)      // Copying the values
    {
        left[i]=A[i+low];
    }

```

```
    for(int i=0;i<high-split;i++)     // Copying the values
    {
        right[i]=A[i+split+1];
    }

    int i=0;                          // Initial index of left subarray
    int j=0;                          // Initial index of right subarray
    int k=low;                        // Initial index of merged subarray

    while(i<split-low+1 && j<high-split)
    {
        if(left[i]<=right[j])
        {
            A[k++]=left[i++];
        }
        else
        {
            A[k++]=right[j++];
        }
    }
    while(i<split-low+1)
    {
        A[k++]=left[i++];
    }
    while(j<high-split)
    {
        A[k++]=right[j++];
    }
}
```

- One subarray is from low to split index and the other subarray is from split+1 to high. This function merges two subarrays to give the resultant array in the increasing order.

```
void *ParallelSort(void *params)  // Function that performs sorting parallely
{
    int current_segment = *((int *) params); // Retrieve the value of current segment
    int low=current_segment*(n/p);           // Calculating the extreme indices of the current segment
    int high=((current_segment+1)*(n/p))-1;  // Calculating high
    quicksort(low,high);                     // Calling quicksort between the low and high index

    pthread_exit(0);                         // Terminating the current thread
}
```

- In this function we sort each segment using single thread parallely. The arguments for this function is the thread number(current segment). Using this thread number we can get the indices of the subarray as each subarray is of size n/p (here **n = pow(2,n) p=pow(2,p)** and low = current_segment*size_off_each_segment. Similarly we calculate high value. After performing sorting we exit the thread.

Vinta Reethu
Operating Systems
Indian Institute of Technology Hyderabad
Assignment-1

भारतीय प्रौद्योगिकी संस्थान हैदराबाद
Indian Institute of Technology Hyderabad

Main function :

```
FILE *fileptr,*opfileptr;
fileptr = freopen("inp.txt","r",stdin); // Opening the inp.txt file
if(fileptr == NULL)                     // If there is error while open the file then this block gets executed
    {
        printf("Couldn't open the mentioned file \n");
        exit(1);
    }
```

- Declaring file pointers and opening the **inp.txt** file using **freopen()**. If there is any error opening the inp.txt file we report and exit.

```
struct timeval start_time,end_time;     // Stores time when declared
```

- Declaring structures to store the time using **gettimeday()**.

```
fscanf(fileptr,"%d", &n);          // Scanning n,p values from the input file
fscanf(fileptr,"%d", &p);
```

- Scanning **n,p** from inp.txt file

```
n=pow(2,n);                        // 2^n
p=pow(2,p);                        // 2^p
```

- Computing $2^n, 2^p$ and storing it in n,p.

```
A=(int*)malloc(sizeof(int)*n);     // Allocating memory for the array
```

- Allocating memory for the array A.

```
for(int i=0;i<n;i++)               // Generating the random values for the array
    {
        A[i]=rand()%100;
    }
fclose(fileptr);                   // Closing the inp.txt file
```

- Initialising array with random values
- Closing the inp.txt file.

Vinta Reethu
Operating Systems
Indian Institute of Technology Hyderabad
Assignment-1

```c
opfileptr = freopen("output.txt","w+",stdout); // Creating and opening output.txt to put our result
if(opfileptr==NULL)                  // If there is error while open the file then this block gets executed
{
    printf("Couldn't open file named output.txt \n");
    exit(1);
}

for(int i=0;i<n;i++)                 // Printing the array to the output file
{
    printf("%d ",A[i]);
}
printf("\n");
```

- Creating and opening **output.txt** file. Printing the array A values into the output file.

```c
    gettimeofday(&start_time, NULL); // Save the current time
```

- Saving the current time.

```c
pthread_t threads[p];             // Declaring thread array
pthread_attr_t attr[p];           // Contents are used at thread creation time to determine attributes for the new thread

for(int i=0;i<p;i++)
{
    pthread_attr_init(&attr[i]);
}
```

- Creating thread array and initialising attributes.

```c
// Creating p(here p is 2^p) threads
for(int i=0;i<p;i++)
{
    int *current_thread = malloc(sizeof(i));
    *current_thread=i;
    pthread_create(&threads[i],&attr[i],ParallelSort,current_thread); //Creating the thread passing the function
}
for (int i = 0; i < p; i++)
{
    pthread_join(threads[i], NULL); // Joining the terminated threads
}
for(int i=1;i<p;i++)
{
    merge(0,((i+1)*(n/p)-1),((i)*(n/p)-1)); // Calling the merge function
}
```

- Creating p threads and passing **ParallelSort** function.

- Once the threads terminate, they are joined using **pthread_join** function.
- Next, Merging the individual array segments.

```
gettimeofday(&end_time, NULL);    // Save the current time

for(int i=0;i<n;i++)              // Printing the sorted array values into output.txt file
{
    printf("%d ",A[i]);
}
printf("\n");
// Calculating the elapsed time
double timetaken = (end_time.tv_sec-start_time.tv_sec + (end_time.tv_usec-start_time.tv_usec)*0.000001); // Finding out the difference between start and end time
printf("Time Taken: %0.3f Microseconds",timetaken*1e6);
```

- Saving the current time.
- Printing the final sorted array into **output.txt** file.
- Calculating the elapsed time and printing it to the output.txt file.

Global declaration Method 2 code :

```
typedef struct indices // Structure defined to stores indices
{
    int low;
    int high;
    int mid;
}indices;
```

- Declaring the indices to pass as an argument to merge function which has low,high,mid attributes.

Method 2 code :

```
void *merge(void *params)          // Perform to merge arrays
{
    indices index = *((indices *) params); // Unwrapping
    int low=index.low;
    int high=index.high;
    int mid=index.mid;
    int left[mid-low+1];           // Array to store the values from low to mid
    int right[high-mid];           // Array to store the values from mid to high

    for(int i=0;i<mid-low+1;i++)   // Copying the values
    {
```

```
            left[i]=A[i+low];
    }

    for(int i=0;i<high-mid;i++)        // Copying the values
    {
        right[i]=A[i+mid+1];
    }

    int i=0;                           // Initial index of left subarray
    int j=0;                           // Initial index of right subarray
    int k=low;                         // Initial index of merged subarray

    while(i<mid-low+1 && j<high-mid)
    {
        if(left[i]<=right[j])
        {
            A[k++]=left[i++];
        }
        else
        {
            A[k++]=right[j++];
        }
    }
    while(i<mid-low+1)
    {
        A[k++]=left[i++];
    }
    while(j<high-mid)
    {
        A[k++]=right[j++];
    }
    pthread_exit(0);                   // Terminating the current thread
}
```

- Argument for the function is of void type as we are passing it inside pthread_create. When we unwrap the argument, we get a structure that stores the indices of the arrays to be merged.

```
void Merging() // Function that performs parallel merging
{
    int size_of_segment=n/p;              // Size of segment is tracked with this variable
    int current_thread_count=p;           // Variable that keeps track of number of threads to be executed in the current level of merging
    while(size_of_segment!=n)             // Iterate untill the size of segment is equal to n
    {
        current_thread_count=current_thread_count/2;        // Every round the number of threads will be halved
        size_of_segment=size_of_segment*2;                  // Every round the size of segment will be halved

        pthread_t threads[current_thread_count];            // Declaring thread array of size current_thread_count
```

```
        pthread_attr_t attr[current_thread_count];          // Contents are used at thread creation time to determine attributes for the new thread

        for(int i=0;i<current_thread_count;i++)             // Setting the attributes
        {
            pthread_attr_init(&attr[i]);
        }
        for(int i=0;i<current_thread_count;i++)             // Creating and passing the function to the thread
        {
            indices index;
            index.low=i*size_of_segment;
            index.high=index.low+size_of_segment-1;
            index.mid=index.low+(index.high-index.low)/2;
            indices *current_index = malloc(sizeof(indices));
            *current_index=index;
            pthread_create(&threads[i],&attr[i],merge,current_index); //Creating the thread passing the function
        }
        for (int i = 0; i < current_thread_count; i++)
        {
            pthread_join(threads[i], NULL); // Joining the terminated threads
        }
    }
}
```

- Here we perform parallel merging until the size of segment is n.
- Every time the number of threads are halved.
- Each time we create number of threads equal to that of current_thread_count and initialise them.
- After creating them we pass merge function to perform merging on that segment. Later once the task is done we join the threads.

```
void *ParallelSort(void *params)        // Function that performs sorting parallely
{
    int current_thread = *((int *) params); // Retrieve the value of current segment
    int low=current_thread*(n/p);           // Calculating the extreme indices of the current segment
    int high=((current_thread+1)*(n/p))-1;  // Calculating high
    quicksort(low,high);                    // Calling quicksort between the low and high index
    pthread_exit(0);                        // Terminating the current thread
}
```

- In this function we sort each segment using single thread parallely. The arguments for this function is the thread number(current segment). Using this thread number we can get the indices of the subarray as each subarray is of size n/p (here **n = pow(2,n) p=pow(2,p)** and low = current_segment*size_off_each_segment. Similarly we calculate high value. After performing sorting we exit the thread.

Main function :

```
FILE *fileptr,*opfileptr;
fileptr = freopen("inp.txt","r",stdin); // Opening the inp.txt file
if(fileptr == NULL)                     // If there is error while open the file
    then this block gets executed
    {
        printf("Couldn't open the mentioned file \n");
        exit(1);
    }
```

- Declaring file pointers and opening the **inp.txt** file using **freopen()**. If there is any error opening the inp.txt file we report and exit.

```
struct timeval start_time,end_time;      // Stores time when declared
```

- Declaring structures to store the time using **gettimeday()**.

```
fscanf(fileptr,"%d", &n);          // Scanning n,p values from the input file
fscanf(fileptr,"%d", &p);
```

- Scanning **n,p** from inp.txt file

```
n=pow(2,n);                        // 2^n
p=pow(2,p);                        // 2^p
```

- Computing $2^n, 2^p$ and storing it in n,p.

```
A=(int*)malloc(sizeof(int)*n);   // Allocating memory for the array
```

- Allocating memory for the array A.

```
for(int i=0;i<n;i++)               // Generating the random values for the array
    {
        A[i]=rand()%100;
    }
fclose(fileptr);                   // Closing the inp.txt file
```

- Initialising array with random values
- Closing the inp.txt file.

Vinta Reethu
Operating Systems
Indian Institute of Technology Hyderabad
Assignment-1


```
opfileptr = freopen("output.txt","w+",stdout); // Creating and opening output.txt to put our result
if(opfileptr==NULL)              // If there is error while open the file then this block gets executed
{
    printf("Couldn't open file named output.txt \n");
    exit(1);
}

for(int i=0;i<n;i++)             // Printing the array to the output file
{
    printf("%d ",A[i]);
}
printf("\n");
```

- Creating and opening **output.txt** file. Printing the array A values into the output file.

```
    gettimeofday(&start_time, NULL); // Save the current time
```

- Saving the current time.

```
pthread_t threads[p];            // Declaring thread array
pthread_attr_t attr[p];          // Contents are used at thread creation time to determine attributes for the new thread

for(int i=0;i<p;i++)
{
    pthread_attr_init(&attr[i]);
}
```

- Creating thread array and initialising attributes.

```
for(int i=0;i<p;i++)
    {
        int *current_thread = malloc(sizeof(i));
        *current_thread=i;
        pthread_create(&threads[i],&attr[i],ParallelSort,current_thread); // Creating the thread passing the function
    }

for (int i = 0; i < p; i++)
    {
        pthread_join(threads[i], NULL); // Joining the terminated threads
    }
```

- Creating threads and passing ParallelSort function.
- Once the threads terminate, they are joined using **pthread_join** function.

Vinta Reethu
Operating Systems
Indian Institute of Technology Hyderabad
Assignment-1

```
Merging();                              // Calling merging function that performs merging
```

- Calling function to perform merging.

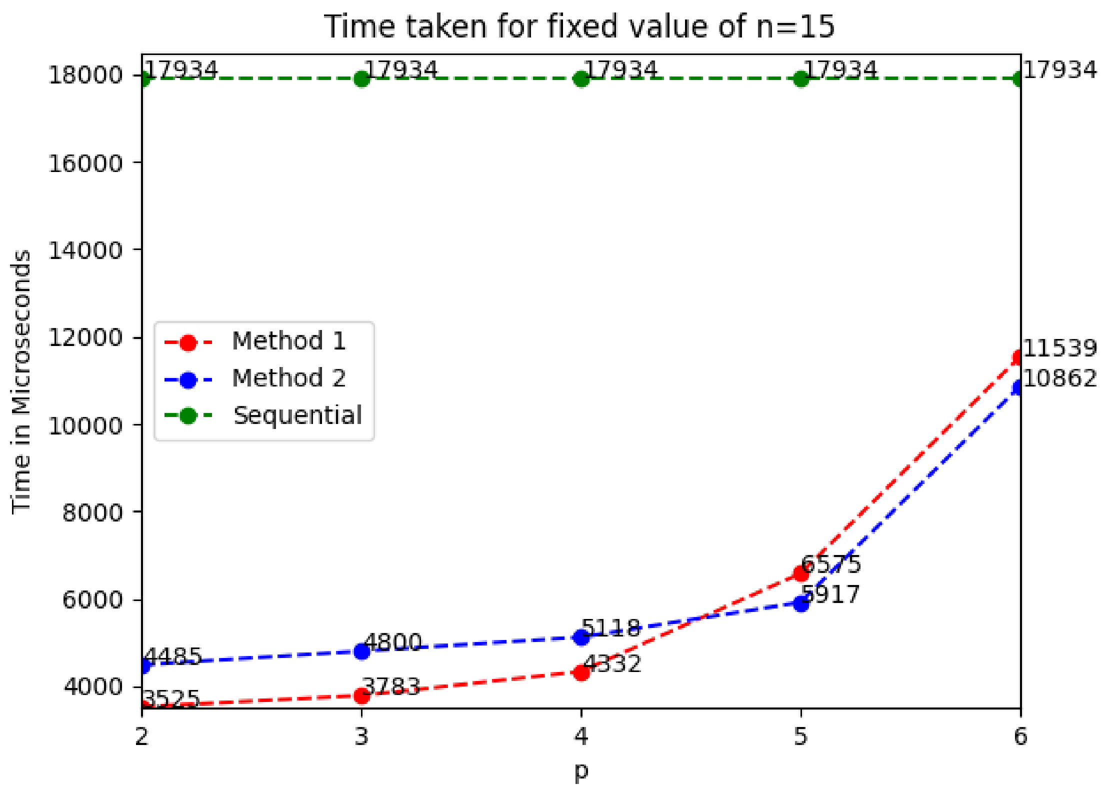
```
gettimeofday(&end_time, NULL);    // Save the current time

for(int i=0;i<n;i++)                    // Printing the sorted array values into output.
    txt file
{
     printf("%d ",A[i]);
}
printf("\n");
// Calculating the elapsed time
double timetaken = (end_time.tv_sec-start_time.tv_sec + (end_time.tv_usec-
    start_time.tv_usec)*0.000001); // Finding out the difference between start and
    end time
printf("Time Taken: %0.3f Microseconds",timetaken*1e6);
```

- Saving the current time.
- Printing the final sorted array into **output.txt** file.
- Calculating the elapsed time and printing it to the output.txt file.

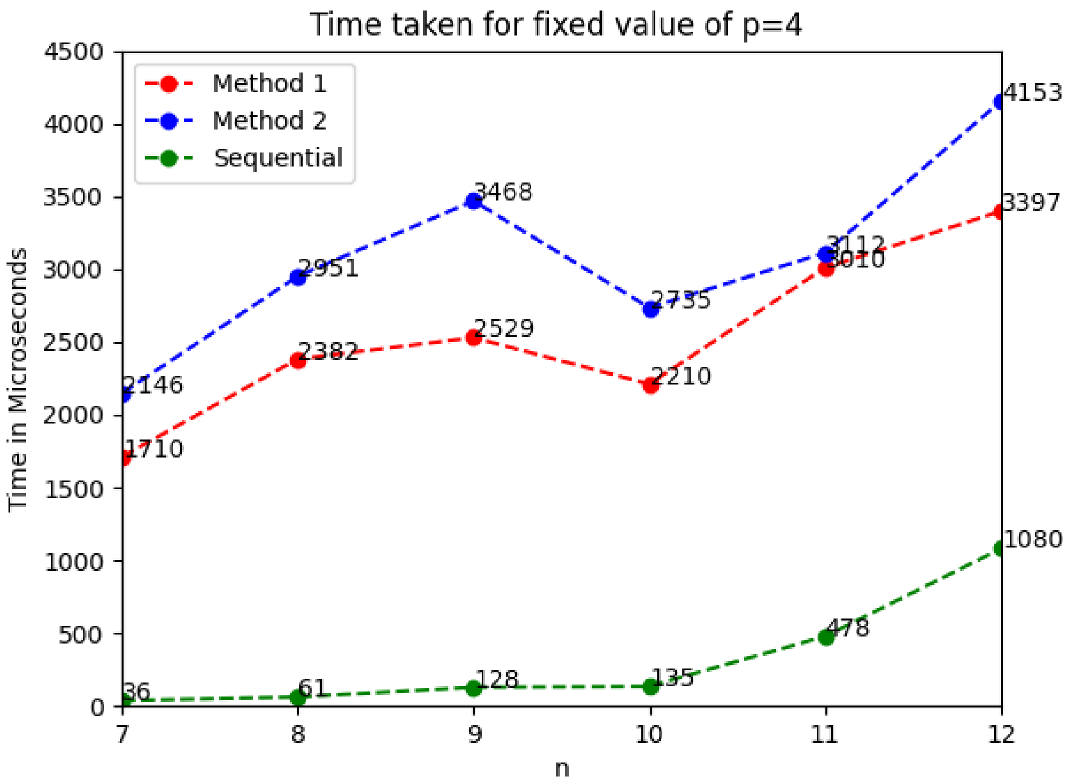
Vinta Reethu
Operating Systems
Indian Institute of Technology Hyderabad
Assignment-1

Output Analysis :



- The above graph is drawn by fixing the value of n to 15 and varying the value of p from 2 to 6.
- As we are keeping the number of elements in the array to be constant and only varying number of threads, the time taken by sequential is constant and is parallel to x-axis.
- It can be seen that out of the 3 methods, sequential algorithm takes takes the highest amount of time.
- Between method-I, method-II we can observe that till p=4 method-I performs better and after p=4 method-II starts performing better.
- As the value of p increases the number of segments increase, size of each segment decreases. Due to the presence of more threads the sorting can be performed in less time, but coming to merging it will take more time. In method-I we perform merging without threads hence the amount of time taken increases.
- From p=5 the number of threads are increasing and it is helping to use more threads for merging process so it starts performing better than method-I.

Vinta Reethu
Operating Systems
Indian Institute of Technology Hyderabad
Assignment-1



- The above graph is drawn by fixing the value of p to 4 and varying the value of n from 7 to 12.
- Here we are increasing n value, so the time taken by sequential also increases with n.
- It can be seen that method-II takes the highest amount of time.
- In both of the methods we can observe a dip at n=10 value.
- Time taken by method-II is high than expected as we are creating and joining threads in each level, as creating threads is a costly process the time taken by the overall method-II is more.
- It can be observed that the time taken by creating new threads in each level of method-II is greater than the time taken to merge the array by a single threads in method-II.
- Because of low value of p, parrallelisation in merging isn't helping here.