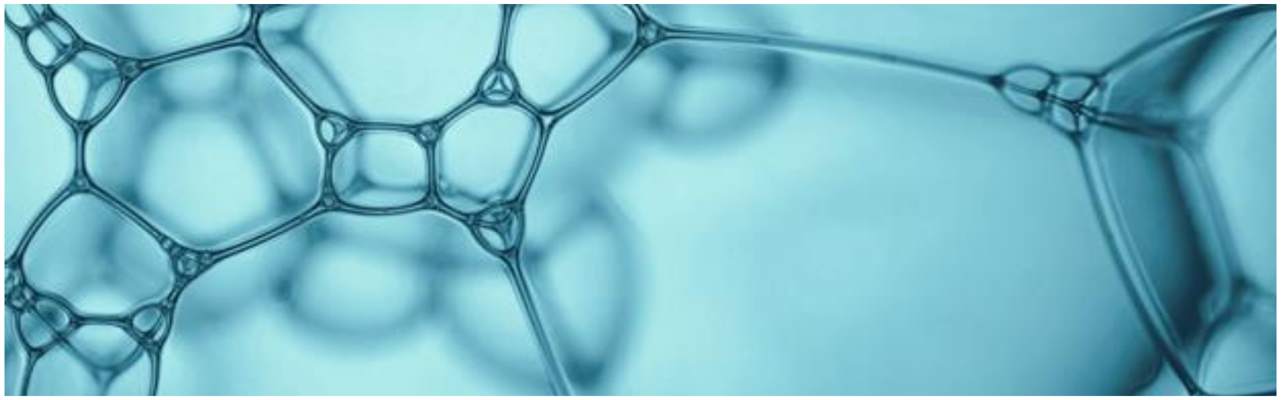


OS Project

Parallel Processing of Algorithms BY

Muhammad Maaz Siddiqui : 21K-3187Muhammad Arif: 21K-3416

DESCRIPTION:

This project is about determining the most efficient programming technique out of the following three: Serial, Open MP, and Multi-threading (pthreads). This is done by experimenting each of these techniques on the following three sorting algorithms: Insertion Sort, Selection Sort, and Cocktail Sort. The serial, obviously, falls under the sequential programming, while the other two are used to achieve Parallel Programming.

OBJECTIVE:

In this project, we aim to discover the most efficient technique which is done by executing programs of which each contain an array, with unsorted elements, and using a timer to record the sorting time of each technique.

Codes:

Selection Sort:

Serial:

```
#include <stdio.h>
#include <time.h>
#include <wait.h>
#include<sys/wait.h>
#include <stdlib.h>
#include<unistd.h>
#include<sys/time.h>
clock_t ticks;

struct timeval stop, start;

void selectionSort(int* A, int n);
void swap(int* a, int* b);

void display(int* arr, int n);
int main(){



  time_t t;

        int number, iter =0, find;
  srand((unsigned) time(&t));



  printf("\nEnter the Size of the Array: ");
```

```
        scanf("%d", &number);

        int *Arr = (int *)malloc( number * sizeof(int));



        for(; iter<number; iter++){
        printf("\nElement No. %d: ", iter + 1);

    Arr[iter] = rand() % 100;

    //scanf("%d", &Arr[iter]);

    printf("%d", Arr[iter]);

    }
```

```
double bstart = clock(); gettimeofday(&start, NULL); selectionSort(Arr, number);
gettimeofday(&stop, NULL);
```

```
    double bstop = clock();
    display(Arr, number);
    FILE* fp;

    fp = fopen("Timings.txt", "a");

             fprintf(fp, "Serial Burst Time: %lf\n",difftime(bstop,bstart));

    fprintf(fp, "Serial Execution Time: %lu\n\n\n", (stop.tv_sec - start.tv_sec) * 1000000 +
stop.tv_usec - start.tv_usec);

    fclose(fp);

}



void display(int* arr, int n){



            printf("\nARRAY: {");
            for(int i = 0; i<n; i++)

            {

            if(i != n - 1)

            {

            printf("%d, ", *(arr + i));

            }

            else

            printf("%d}\n\n", *(arr + i));

            }
```

```
}


void selectionSort(int* A, int n)

{

        for(int startpos =0; startpos < n-1; startpos++){
        int maxpos = startpos;

        for(int i=startpos +1; i< n; ++i){
        if(A[i] < A[maxpos]){

                maxpos = i;

        }

        }

  if(maxpos != startpos)

                swap(&A[startpos], &A[maxpos]);

        }



}
```

```
void swap(int* a, int* b){
        int temp = *a;

        *a = *b;

        *b = temp;

}
```

Multi-threading (Pthreads):

```
#include <stdio.h>
#include <time.h>
#include <wait.h>
#include<sys/wait.h>
#include <stdlib.h>
#include<unistd.h>
#include<pthread.h>
#include<sys/time.h>

//time_t start,  stop;
struct timeval stop,
start;clock_t ticks;

int linearSearch(int* A, int n, int tos);
void* selectionSort();

void swap(int* a, int* b);
void display(int* arr, int n);
#define MAX_THREAD 4



struct sortingArgs{



    int *arr;
    int n;

};

int main(){



    time_t t;

    int noDuplication = 0;// 0 = false
```

```
srand((unsigned) time(&t));

         pthread_t threads[MAX_THREAD];
         int number, iter =0, find;



struct sortingArgs Args;



printf("\nEnter the Size of the Array: ");
         scanf("%d", &number);

Args.n = number;
int Arr[number];

         //Arr = (int *)malloc( number * sizeof(int));
Args.arr = Arr;

for(; iter<number; iter++){

          printf(/*\nEnter */"\nElement No. %d: ", iter + 1);

          //scanf("%d", &Arr[iter]);
```

```
                    Arr[iter] = rand() % 100;
          printf("%d", Arr[iter]);

}



double bstart = clock();
gettimeofday(&start, NULL);



for(int i = 0; i<MAX_THREAD; i++)

{

                                        pthread_create(&threads[i], NULL, selectionSort, &Args);



}



for(int i = 0; i<MAX_THREAD;i++)

{



          pthread_join(threads[i], NULL);

}



          gettimeofday(&stop, NULL);
double bstop = clock(); display(Arr,
number);
```

```
    FILE* fp;

    fp = fopen("Timings.txt", "a");

              fprintf(fp, "Multi-threading Burst Time: %lf\n",difftime(bstop,bstart));

    fprintf(fp, "Multi-threading Execution Time: %lu\n\n\n",(stop.tv_sec - start.tv_sec) * 1000000 +
stop.tv_usec - start.tv_usec);

    fclose(fp);

}




void* selectionSort(void* input)

{

              struct sortingArgs *Arg = (struct sortingArgs*) input;
    int *A;

    A = Arg->arr;
    int n = Arg->n;



             for(int startpos =0; startpos < n-1; startpos++){
             int maxpos = startpos;

             for(int i=startpos +1; i< n; ++i){
             if(A[i] < A[maxpos]){

             maxpos = i;
```

```
            }

            }

    if(maxpos != startpos)

                          swap(&A[startpos], &A[maxpos]);

            }

}


void swap(int* a, int* b){

            int temp = *a;

            *a = *b;

            *b = temp;

}


void display(int* arr, int n){



            printf("\nARRAY: {");
            for(int i = 0; i<n; i++)

            {

            if(i != n - 1)

            {

            printf("%d, ", *(arr + i));

            }
```

```
        else

        printf("%d}\n\n", *(arr + i));

        }


}
```

Open_MP:

```
#include<stdio.h>
#include<time.h>
#include<sys/time.h>
#include<stdlib.h>
#include<pthread.h>
struct array

{

int  *ptr;
long int size;

}Array,Final_Array;
struct Time

{

double Start,End;

}Execution;

struct timeval stop, start;

void* Cocktail_Sort_Down(void *args )

{

int start=(Array.size/2+1);
int end=Array.size-1;
int i;
while(start<end)

{

for(i=start;i<end;i++)

{

if(Array.ptr[i]>Array.ptr[i+1])

{

int temp=Array.ptr[i];
Array.ptr[i]=Array.ptr[i+1];
Array.ptr[i+1]=temp;
}

}

for(i=end-1;i>start;i--)

{

if(Array.ptr[i]<Array.ptr[i-1])

{

int temp=Array.ptr[i];
Array.ptr[i]=Array.ptr[i-1];
Array.ptr[i-1]=temp;

}

}

start++
;end--;



}

pthread_exit(0);

}

void* Cocktail_Sort_Up(void *args)

{

int start=0;

int end=Array.size/2;
int i;
while(start<end)

{

for(i=start;i<end;i++)

{

if(Array.ptr[i]>Array.ptr[i+1])

{

int temp=Array.ptr[i];
Array.ptr[i]=Array.ptr[i+1];
Array.ptr[i+1]=temp;
}

}

for(i=end-1;i>start;i--)

{

if(Array.ptr[i]<Array.ptr[i-1])

{

int temp=Array.ptr[i];
Array.ptr[i]=Array.ptr[i-1];
Array.ptr[i-1]=temp;

}

}

start++;
end--;

}


pthread_exit(0);

}

void* merge(void *args)

{

Final_Array.size=Array.size;

Final_Array.ptr=(int *) calloc(Final_Array.size,sizeof(int));int
i=0,j=Array.size/2+1,k=0;
```

```
for(;i<(Array.size/2+1);k++)

{

if(Array.ptr[i]<Array.ptr[j] || j>=Array.size)

{

Final_Array.ptr[k]=Array.ptr[i];i++;

}

else

{
```

```
if(j<Array.size)

{

Final_Array.ptr[k]=Array.ptr[j];j++;

}

}

}

while(j<Array.size)

{

Final_Array.ptr[k]=Array.ptr[j];j++;

k++;

}


}

void* Filling(void *args)

{

FILE *ptr;
ptr=fopen("Project_Times.txt","a+");
if(ptr==NULL)

{

printf("Unable to Open File");
exit(1);
```

```
}

else

{

double total= (double)(Execution.End-Execution.Start)/(double)CLOCKS_PER_SEC;double
Total=(double)((stop.tv_sec - start.tv_sec) * 1000000 + stop.tv_usec - start.tv_usec);

fprintf(ptr,"Burst Time Taken By Multithreaded_Cocktail_Sort is %lf\n",total);
fprintf(ptr,"Time Taken By Multithreaded_Cocktail_Sort is %lf\n",Total);
```

```
}

fclose(ptr);
pthread_exit(0);

}

int main(void)

{

srand(time(NULL));

printf("Enter The Size of Array: ");
scanf("%ld",&Array.size);

Array.ptr=(int *) calloc(Array.size,sizeof(int));
printf("Enter The Elements Array\n");

for(int i=0;i<Array.size;i++)

{

*(Array.ptr+i)=rand()%Array.size;

}

pthread_t threads[2];
Execution.Start=clock();
gettimeofday(&start, NULL);

pthread_create(&threads[0],NULL,Cocktail_Sort_Up,NULL);
pthread_create(&threads[1],NULL,Cocktail_Sort_Down,NULL);for(int
i=0;i<2;i++)

{
```

```
pthread_join(threads[i],NULL);

}

pthread_create(&threads[0],NULL,merge,NULL);
pthread_join(threads[0],NULL); gettimeofday(&stop,
NULL); Execution.End=clock();
pthread_create(&threads[1],NULL,Filling,NULL);
printf("Sorted Array: ");

for(int i=0;i<Array.size;i++)
```

```
{
printf("%d ",(Final_Array.ptr[i]));
}
printf("\n");
pthread_join(threads[1],NULL);

}
```

Insertion Sort:

Serial:

```
#include <stdio.h> #include <stdlib.h> #include <pthread.h> #include <sys/time.h> #define
SIZE 10



 int n1[SIZE];
 int n2[SIZE];


 struct timeval

 stop2,start2;typedef

 struct dim{

             int start;

             int end;

 } limit;



 typedef struct
    joined{int Start;

    int
    mid;int
    End;

 } join;
```

```
void* merge(void *args) {


    struct joined* params = (join*)args;

    int begin = params->Start, mid = params->mid, end = params-

    >End;int i = begin, j = mid, tpos = begin;


    while (i < mid && j < end)

    {

        if (n1[i] < n1[j])
            n2[tpos++] =
            n1[i++];

        else

            n2[tpos++] = n1[j++];

    }
    while (i < mid)
        n2[tpos++] = n1[i++];



    while (j < end)
        n2[tpos++] = n1[j++];



    return NULL;

}



void* insertion(void *l)

{

            const limit* l1 = (limit*)l;
            int i = l1->start + 1;

            int j, k, move;



            while(i < l1->end)

            {

                        j = i-1;

                        move = n1[i];

                        while(j >= l1->start && move < n1[j])

                        {

                                    n1[j+1] = n1[j];
                                    j--;

                        }
```

```
                    n1[j+1] = move;
                    i++;

          }

          return NULL;

}



void fillarray(int size) {
   int i;
   srand(time(NULL));
   for (i=0; i<size; i++)

      n1[i] = rand() % 100;

}



void print_array(int list[], int size) {
   int i;

   for (i=0; i<size-1; i++)
      printf("%d, ", list[i]);

   printf("%d\n", list[i]);

}
```

```
void file_create(double start,double stop)

{

            FILE *ptr;
            ptr=fopen("PTHREAD_Insertion.txt","a+");
            if(ptr==NULL)

            {

                        printf("Unable to Open File");
                        exit(EXIT_FAILURE);


            }

            els
            e

            {           double total=  (double)(stop-start);
                        fprintf(ptr,"Time Taken(Burst Time): %lf\n",total);

                        fprintf(ptr,"Time Taken(Clock Time): %lu\n",(stop2.tv_sec-

start2.tv_sec)*1000000+stop2.tv_usec-start2.tv_usec);

            }

            fclose(ptr);

}



int main()

{

            fillarray(SIZE);
   print_array(n1, SIZE);

            pthread_t t1,t2;
```

```
limit l1,l2;

l1.start=0;

        l1.end=SIZE/2;



        l2.start=l1.end
        ;l2.end=SIZE;



        join j1;
        j1.Start=l1.start
        ;j1.mid=l2.start;
        j1.End=l2.end;

        //printf("%d %d %d",j1.Start,j1.mid,j1.End);
        pthread_t m1;



        double Start1=clock();
        gettimeofday(&start2,NULL);



        pthread_create(&t1,NULL,insertion,&l1);
```

```
            pthread_create(&t2,NULL,insertion,&l2);



    pthread_join(t1,NULL);
            pthread_join(t2,NULL);



    pthread_create(&m1,NULL,merge,&j1);
            pthread_join(m1,NULL);



            gettimeofday(&stop2,NULL);
            double stop1 = clock();



            print_array(n2,SIZE);
            file_create(Start1,stop1);



            return 0;

}
```

Multi-threading (Pthreads):

```
#include <stdio.h>
#include <stdlib.h>
#include
<sys/time.h>
#include <time.h>
#define SIZE 10

int num1[SIZE];

struct timeval stop2,start2;



void insertion_sort()

{

            int i,j,temp;



    for(i = 1; i < SIZE; i++)

    {

        j = i - 1;

        temp = num1[i];

        while(j >= 0 && num1[j] > temp)

        {

            num1[j + 1] =
            num1[j];j--;

        }
        num1[j + 1] = temp;

    }

}



void fill_array(int size)
    {int i;
    srand(time(NULL));
    for (i=0; i<size; i++)

        num1[i] = rand() % 100;

}



void file_create(double start,double stop)

{

             FILE *ptr;
             ptr=fopen("Sequencial_Insertion.txt","a+");
             if(ptr==NULL)

             {

                          printf("Unable to Open File");
                          exit(1);


             }

             els
             e
        {

                double total=  (double)(stop-start);
                fprintf(ptr,"Time Taken(Burst Time): %lf\n",total);

                fprintf(ptr,".comTime Taken(Clock Time): %lu\n",(stop2.tv_sec-

start2.tv_sec)*1000000+stop2.tv_usec-start2.tv_usec);

        }

        fclose(ptr);

}

int main()

{



        fill_array(SIZE); double
        Start1=clock();



        gettimeofday(&start2,NULL);
        insertion_sort();
        gettimeofday(&stop2,NULL);
        double stop1=clock();

        for(int i=0;i<SIZE;i++)

        {

                printf("%d ",num1[i]);

        }

  putchar('\n')
        ;
```

```
            file_create(Start1,stop1);
            return 0;

}
```

Open-mp:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <time.h>
#include <omp.h>



#define SIZE 10
int num1[SIZE];

struct timeval stop2, start2;



void insertion_sort()

{

    int i, j, temp;



    for (i = 1; i < SIZE; i++) {j =
        i - 1;

        temp = num1[i];

        while (j >= 0 && num1[j] > temp) {
            num1[j + 1] = num1[j];

            j--;

        }

        num1[j + 1] = temp;

    }
```

```
}


void fill_array(int size) {
    int i;
    srand(time(NULL)); for
    (i = 0; i < size; i++)

        num1[i] = rand() % 100;

}


void file_create(double start, double stop)

{

    FILE* ptr;

    ptr = fopen("open_mp_time.txt",
    "a+");if (ptr == NULL) {

        printf("Unable to Open File");
        exit(1);

    }
    else {

        double total = (double)(stop - start);

        fprintf(ptr, "Time Taken(Burst Time): %lf\n", total);

        fprintf(ptr, "Time Taken(Clock Time): %lu\n", (stop2.tv_sec - start2.tv_sec) * 1000000 +
stop2.tv_usec - start2.tv_usec);

    }

    fclose(ptr);

}


void parallel_insertion_sort()

{

    gettimeofday(&start2, NULL);


#pragma omp parallel

    {

        insertion_sort();

    }


    gettimeofday(&stop2, NULL);

}


int main()
```

```
{

    fill_array(SIZE);

    double Start1 =

    omp_get_wtime();

    parallel_insertion_sort();


    double stop1 = omp_get_wtime();



    for (int i = 0; i < SIZE; i++) {
        printf("%d ", num1[i]);

    }

    putchar('\n');



    file_create(Start1,
    stop1);return 0;

}
```

Cocktail Sort

Serial:

```
#include<stdio.h>
#include<time.h>
#include<sys/time.h>
#include<stdlib.h>

void Cocktail_Sort(int *A,int size)

{

        int start=0;

        int end=size-1;
        while(start!=(size/2))

        {

                for(int i=start;i<end;i++)

                {

                        if( *(A+i)>*(A+(i+1)))

                        {

                                int temp=*(A+i);

                                *(A+i)=*(A+(i+1));

                                *(A+(i+1))=temp;
```

```
                    }


                }

                for(int j=end-1;j>=start;j--)

                {

                    if(*(A+j)<*(A+(j-1)))

                    {

                        int temp=*(A+j);

                        *(A+(j))=*(A+(j-1));

                        *(A+(j-1))=temp;

                    }

                }

                start++;
                end--;

        }

}

void Filling(double Start,double End,double Total)

{

        FILE *ptr; ptr=fopen("Project_Times.txt","a+");
        if(ptr==NULL)

        {
	}	printf("Unable to Open File");
		exit(1);
	else

	{

		double total= (double)(End-Start)/(double)CLOCKS_PER_SEC;;
%lf\n",total);	fprintf(ptr,"Burst Time Taken By Sequencial_Cocktail_Sort is

		fprintf(ptr,"Total Time Taken By Sequencial_Cocktail_Sort is

%lf\n",Total);

	}

	fclose(ptr);

}

int main(void)
```

```
{

        srand(time(NULL));int
        n;

        struct timeval stop, start;
        printf("Enter The Size of Array: ");
        scanf("%d",&n);

        int *ptr=(int *) calloc(n,sizeof(int));
        printf("Enter The Elements Array\n");
        for(int i=0;i<n;i++)

        {

                *(ptr+i)=rand()%n;

        }

        double Start=clock();
        gettimeofday(&start, NULL);
        Cocktail_Sort(ptr,n);
        gettimeofday(&stop, NULL);
        double End=clock();
        printf("Sorted Array: "); for(int
        i=0;i<n;i++)

        {

                printf("%d ",(ptr[i]));

        }

        printf("\n");

        double Total=(double)((stop.tv_sec - start.tv_sec) * 1000000 + stop.tv_usec -
start.tv_usec);

        Filling(Start,End,Total);
}
```

Multi-threading (Pthreads):

```
#include<stdio.h>
#include<time.h>
#include<sys/time.h>
#include<stdlib.h>
#include<pthread.h>
struct array

{
        int  *ptr; long
        int size;



}Array,Final_Array;
struct Time

{

        double Start,End;

}Execution;

struct timeval stop, start;

void* Cocktail_Sort_Down(void *args )

{

        int start=(Array.size/2+1);
        int end=Array.size-1;

        int i;
        while(start<end)

        {

                for(i=start;i<end;i++)

                {

                        if(Array.ptr[i]>Array.ptr[i+1])

                        {

                                int temp=Array.ptr[i];
                                Array.ptr[i]=Array.ptr[i+1];
                                Array.ptr[i+1]=temp;
        }


}

for(i=end-1;i>start;i--)

{

        if(Array.ptr[i]<Array.ptr[i-1])

        {

                int temp=Array.ptr[i];
                Array.ptr[i]=Array.ptr[i-1];
                Array.ptr[i-1]=temp;

        }


}
```

```
            start++
            ;end--;



        }

        pthread_exit(0);

}

void* Cocktail_Sort_Up(void *args)

{

        int start=0;

        int end=Array.size/2;
        int i;
        while(start<end)

        {

            for(i=start;i<end;i++)

            {

                if(Array.ptr[i]>Array.ptr[i+1])

                {

                    int temp=Array.ptr[i];
                    Array.ptr[i]=Array.ptr[i+1];
                    Array.ptr[i+1]=temp;

                }
}

for(i=end-1;i>start;i--)

{

        if(Array.ptr[i]<Array.ptr[i-1])

        {

                int temp=Array.ptr[i];
                Array.ptr[i]=Array.ptr[i-1];
                Array.ptr[i-1]=temp;}



         }



}

start++;
end--;
```

```
pthread_exit(0);

 }

 void* merge(void *args)

 {

        Final_Array.size=Array.size;

        Final_Array.ptr=(int *) calloc(Final_Array.size,sizeof(int));int
        i=0,j=Array.size/2+1,k=0;



        for(;i<(Array.size/2+1);k++)

        {

               if(Array.ptr[i]<Array.ptr[j] | | j>=Array.size)

               {

                      Final_Array.ptr[k]=Array.ptr[i];i++;



               }

               else

               {      if(j<Array.size)

                      {

                             Final_Array.ptr[k]=Array.ptr[j];j++;

                      }
```

```
                }

        }

        while(j<Array.size)

        {

                Final_Array.ptr[k]=Array.ptr[j];j++;

                k++;


        }


}

void* Filling(void *args)
```

```
{

        FILE *ptr; ptr=fopen("Project_Times.txt","a+");
        if(ptr==NULL)

        {

                printf("Unable to Open File");
                exit(1);


        }

        else

        {
                double total= (double)(Execution.End-

Execution.Start)/(double)CLOCKS_PER_SEC;

                double Total=(double)((stop.tv_sec - start.tv_sec) * 1000000 + stop.tv_usec

- start.tv_usec);

                fprintf(ptr,"Burst Time Taken By Multithreaded_Cocktail_Sort is

%lf\n",total);

                fprintf(ptr,"Time Taken By Multithreaded_Cocktail_Sort is %lf\n",Total);
        }

        fclose(ptr);
        pthread_exit(0);



}

int main(void)
```

```
{

        srand(time(NULL));

        printf("Enter The Size of Array: ");
        scanf("%ld",&Array.size);

        Array.ptr=(int *) calloc(Array.size,sizeof(int));
        printf("Enter The Elements Array\n");

        for(int i=0;i<Array.size;i++)

        {

                *(Array.ptr+i)=rand()%Array.size;

        }

        pthread_t threads[2];
        Execution.Start=clock();
        gettimeofday(&start, NULL);

        pthread_create(&threads[0],NULL,Cocktail_Sort_Up,NULL);
        pthread_create(&threads[1],NULL,Cocktail_Sort_Down,NULL);for(int
        i=0;i<2;i++)

        {

                pthread_join(threads[i],NULL);

        }

        pthread_create(&threads[0],NULL,merge,NULL);
        pthread_join(threads[0],NULL); gettimeofday(&stop,
        NULL); Execution.End=clock();
        pthread_create(&threads[1],NULL,Filling,NULL);
        printf("Sorted Array: ");

        for(int i=0;i<Array.size;i++)

        {

                printf("%d ",(Final_Array.ptr[i]));

        }

        printf("\n");
        pthread_join(threads[1],NULL);

}
```

open-mp:

```
#include <stdio.h>
#include <time.h>
#include <sys/time.h>
#include <stdlib.h>
#include <omp.h>



void Cocktail_Sort(int *A, int size) {int
    start = 0;

    int end = size - 1;

    while (start != (size / 2)) {
        #pragma omp parallel for

        for (int i = start; i < end; i++) {if
            (A[i] > A[i + 1]) {

                int temp = A[i];
                A[i] = A[i + 1];

                A[i + 1] = temp;

            }

        }



        #pragma omp parallel for

        for (int j = end - 1; j >= start; j--) {if
            (A[j] < A[j - 1]) {

                int temp = A[j];
    A[j] = A[j - 1];

    A[j - 1] = temp;

}
```

```
        }



        start++;
        end--;

    }

}



void Filling(double Start, double End, double Total) {FILE
    *ptr;

    ptr = fopen("Project_Times.txt", "a+");if
    (ptr == NULL) {

        printf("Unable to Open File");
        exit(1);

    } else {

double total = (End - Start) / CLOCKS_PER_SEC;

        fprintf(ptr, "Burst Time Taken By Sequential_Cocktail_Sort is %lf\n", total); fprintf(ptr, "Total
        Time Taken By Sequential_Cocktail_Sort is %lf\n", Total);

    }

    fclose(ptr);

}



int main(void) {
    srand(time(NULL));int
    n;
    struct timeval stop, start;
    printf("Enter The Size of Array: ");
    scanf("%d", &n);

    int *ptr = (int *)calloc(n, sizeof(int));
    printf("Enter The Elements of the Array\n");for
    (int i = 0; i < n; i++) {

        ptr[i] = rand() % n;

    }


double Start = omp_get_wtime(); gettimeofday(&start, NULL); Cocktail_Sort(ptr, n);
gettimeofday(&stop, NULL);
```

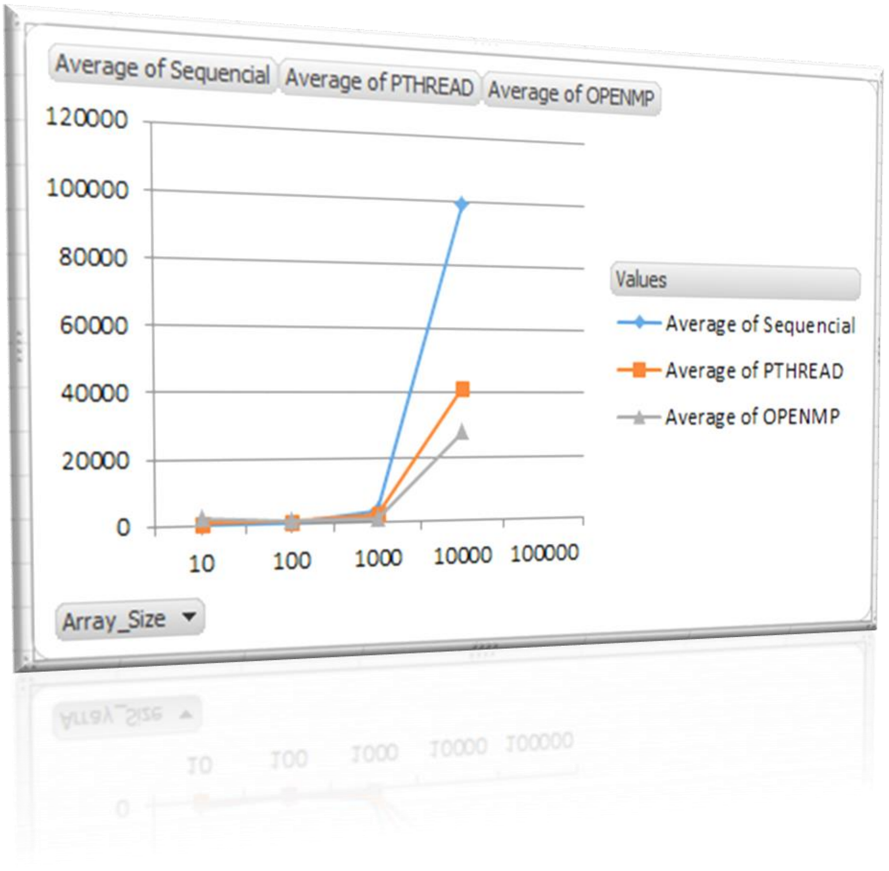
```
double End = omp_get_wtime();
printf("Sorted Array: ");

for (int i = 0; i < n; i++) {
    printf("%d ", ptr[i]);

}

printf("\n");

double Total = (stop.tv_sec - start.tv_sec) * 1000000 + stop.tv_usec - start.tv_usec;Filling(Start,
End, Total);
```

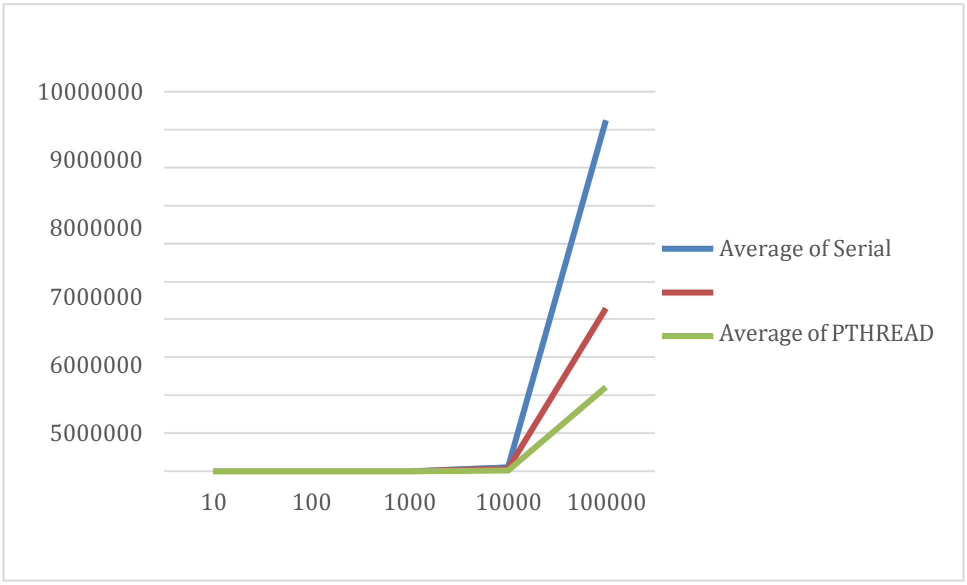
Conclusion:

After executing the programs, the sorting times of each technique implemented on a sorting algorithm is compared. The technique which produces the lowest time is considered as the best technique, and the one which produces the highest time is considered as the worst. This is done for all three sorting algorithms.

Comparative Graphs

Insertion Sort (Clock Time)

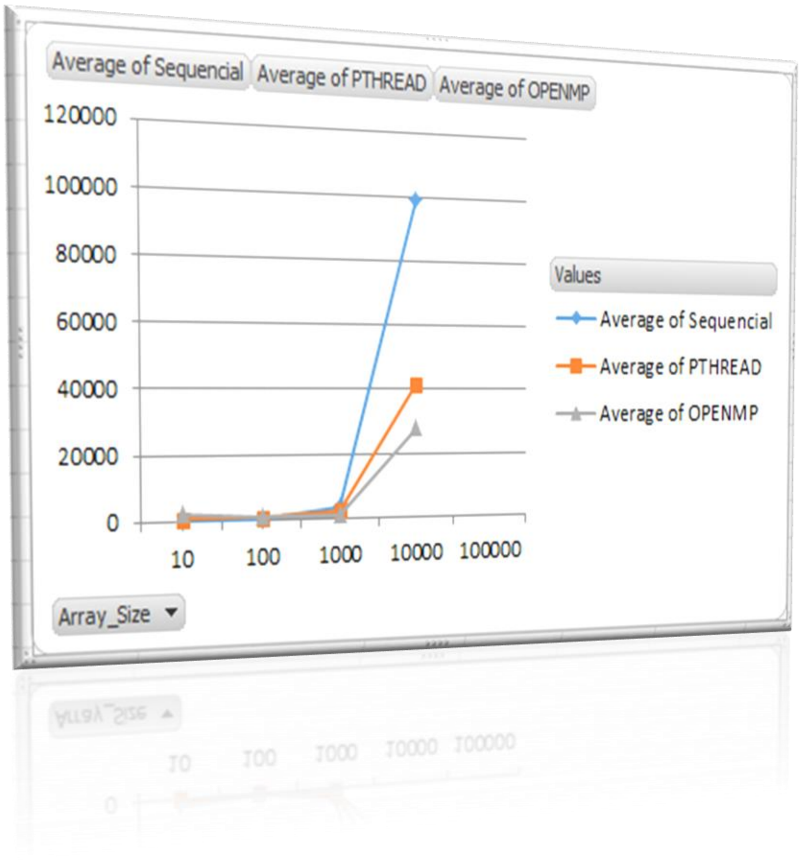

Average of Sequencial
Average of PTHREAD
Average of OPENMP
120000
100000
80000
60000
40000
20000
0
10
100
1000
10000
100000
Values
Average of Sequencial
Average of PTHREAD
Average of OPENMP
Array_Size



Burst Time)

10	3	709	*2382*
100	58	598	*817*
1000	3622	2288	*1017*
10000	99434	41588	*27861*
100000	*9248780*	*2157261*	*2216668*

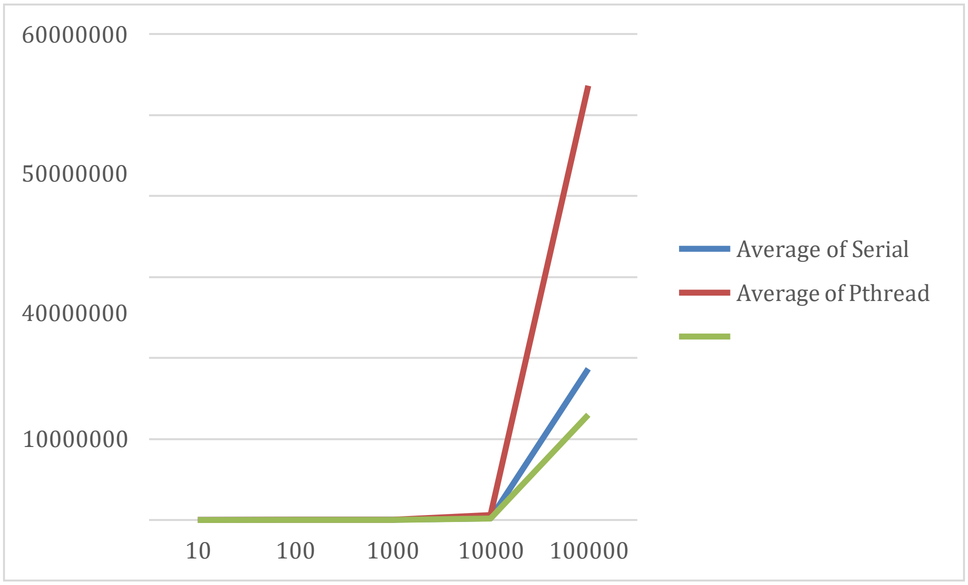
10	7	647	*2425*
100	63	459	*960*
1000	3583	598	*1017*
10000	99172	78097	*27861*
100000	*9247398*	*4290962*	*2216668*



Selection Sort (Clock Time)

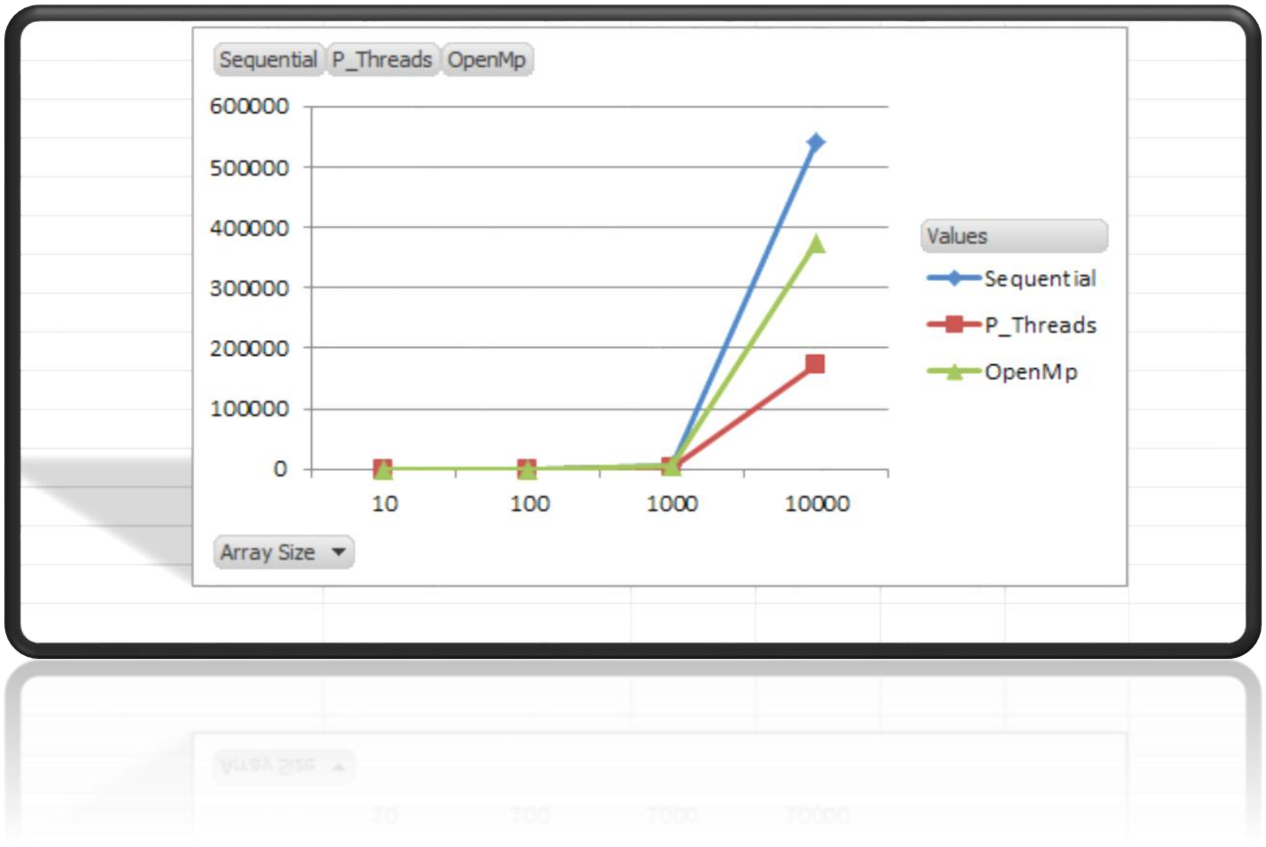
10	1	148	*12*
100	35	340	*59*
1000	2606	14240	*2436*
10000	210689	631466	*194323*

(Burst Time)



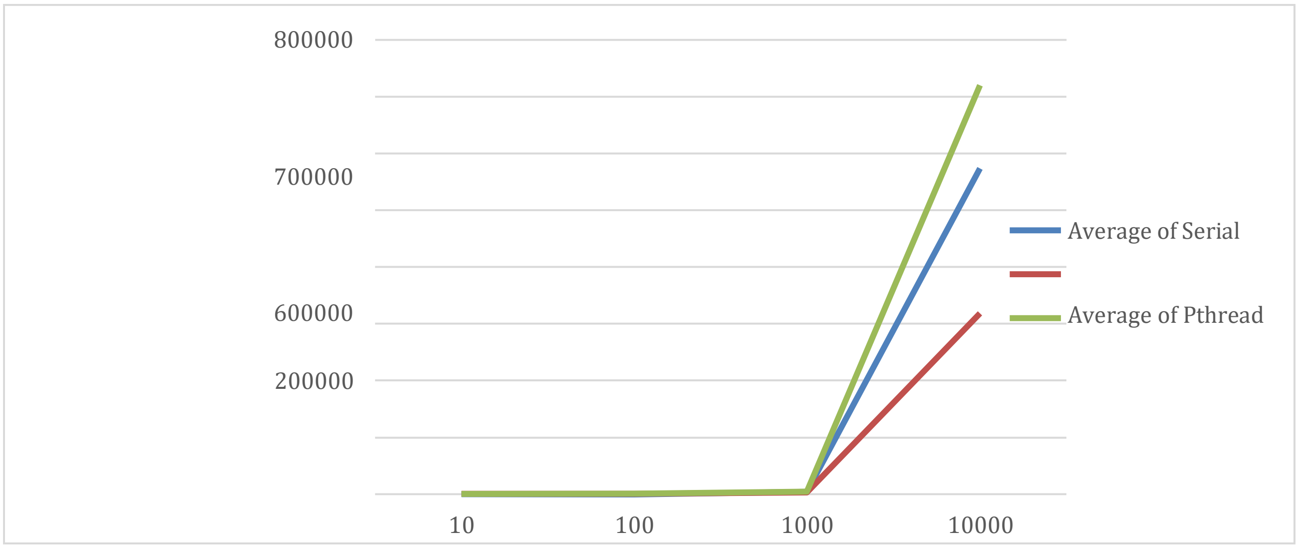
10	5	105	*16*
100	38	172	*63*
1000	2610	4932	*2443*
10000	156670	557027	*140868*
100000	*18648623*	*53594473*	*12976536*

Cocktail Sort (Clock Time)



Time Taken	*Time Taken*	*Time Taken*	*Array Size*
1	450	125	*10*
32	731	269	*100*
4546	1929	39144	*1000*
541007	71912	419953	*10000*
Sequential	***P_Threads***	***OpenMp***	

(Burst Time)



10	3	752	*209*
100	34	699	*619*
1000	4553	3218	*3964*
10000	*573569*	*318419*	*719846*

The above results show that the primary factor of efficiency is the size of the data. The smaller the data the better sequential performs. On the other hand, the larger the data the better PTHREAD and OPENMP performs. The second factor is the CPU burst time; the results show that depending on the algorithms, parallel programming tends to be faster but on the other hand, creation of multiple threads or processes may need more CPU time as compared to the sequential programming.