**Project Proposal Report**

**Topic**

“**Parallel Processing of Algorithms**”

**BY**

**Mohtashim**

STUDENT ID:**20K-0462**

**Ameer Ali**

STUDENT ID:**20K-0498**

**Ali Naeem**

STUDENT ID:**20K-1751**

Project Description

This project is about determining the most efficient programming technique out of the following three: Serial, OpenMp, 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.

# Project 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));

}

}

**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);

}

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);

}

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);

}

else

{

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;

}

**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;;

fprintf(ptr,"Burst Time Taken By Sequencial\_Cocktail\_Sort is %lf\n",total);

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);

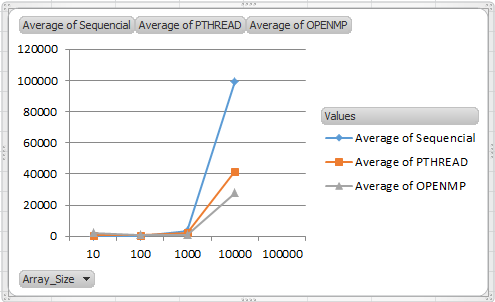
}

Project 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)**

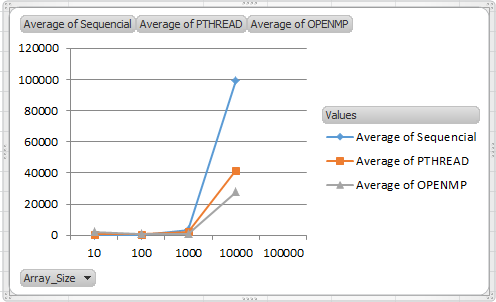


|  |  |  |  |
| --- | --- | --- | --- |
| **Array\_Size** | **Sequencial** | **PTHREAD** | **OPENMP** |
| 10 | 3 | 709 | 2382 |
| 100 | 58 | 598 | 817 |
| 1000 | 3622 | 2288 | 1017 |
| 10000 | 99434 | 41588 | 27861 |
| 100000 | 9248780 | 2157261 | 2216668 |

**(Burst Time)**

|  |  |  |  |
| --- | --- | --- | --- |
| **Array\_Size** | **Serial** | **PTHREAD** | **OPENMP** |
| 10 | 7 | 647 | 2425 |
| 100 | 63 | 459 | 960 |
| 1000 | 3583 | 598 | 1017 |
| 10000 | 99172 | 78097 | 27861 |
| 100000 | 9247398 | 4290962 | 2216668 |

**Selection Sort**

**(Clock Time)**

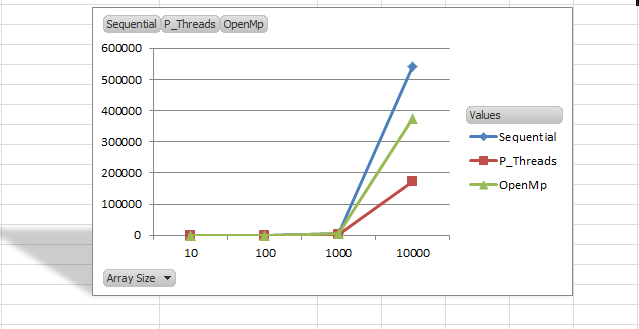
|  |  |  |  |
| --- | --- | --- | --- |
| **Array\_Size** | **Sequencial** | **PTHREAD** | **OPENMP** |
| 10 | 1 | 148 | 12 |
| 100 | 35 | 340 | 59 |
| 1000 | 2606 | 14240 | 2436 |
| 10000 | 210689 | 631466 | 194323 |
|  |  |  |  |

**(Burst Time)**

|  |  |  |  |
| --- | --- | --- | --- |
| **Array\_Size** | **Serial** | **Pthread** | **OpenMP** |
| 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)**

|  |  |  |  |
| --- | --- | --- | --- |
| **Array\_Size** | **Serial** | **Pthread** | **OpenMp** |
| 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 tend 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.