

**Operating Systems (CS-220)**

**Project Report**

**Hashir Kaleem Khan K17-3948**

**Project Title:** COMPARISON BETWEEN PROCESS AND THREADS Using 3 Algorithms

**Advisor/Teacher:** Nausheen Shoaib

**INTRODUCTION:**

Processes are an executing instance of the program having a separate virtual address space while the Threads are subset of the process that share its virtual address spaces and system resources. Our Project will demonstrate the basic as well as complex differences between Processes and threads in aspects of memory consumption, context switching timing, resource consumption, dependency, data and code sharing, time for creation and deletion and how the OS treats Processes as well as threads. My Project will implement 3 sorting algorithms which include Quick Sort Merge Sort and Insertion Sort.

# DESCRIPTION:

Project will provide the options for selecting which sorting technique user wants to operate; two files have been attached in which all the sortings have been compiled with respect to threading using openmp and/or pthreads and the other file contains all sortings implemented with Processes.

# HOW I STARTED:

# Firstly, the major task was to gather the information regarding the differences between openmp, pthreads and process. In order to understand deeply what these are and how to attain the maximium possible efficiency and least amount of clock time.

# PROBLEMS FACED:

Developing logic around every sorting algorithm that where the placement of parallelization will be beneficial and where it will be useless.

# *SOLUTION TO EACH PROBLEM:*

I managed to understand and analyze carefully that how the placement would take place by implementing it on different operating systems and managing to get the best results.

# THE ACTUAL WORKING OF THE PROJECT (Methodology):

Basically project will provide difference between process and threads through sorting algorithms which include merge sort, quick sort, and insertion sort.

Every sorting algorithm has been implemented by process and openmp and every algorithm tells the appropriate termination time which helps us to differentiate between them.

# EXPLANATION OF SOURCE CODE:

Two files have been created, one for all the Processes of sorting algorithms and another for the Threads of sorting algorithms. All the source codes are in C++ and in Thread section I have used openmp instead of pthread because of the efficiency and Processes have been done the regular way.

**Thread Code (OpenMP):**

#include<iostream>

#include<iomanip>

#include<stdlib.h>

#include<time.h>

#include<pthread.h>

#include<omp.h>

#include<fstream>

#define MAX 1000000

#define MAX\_THREAD 4

using namespace std;

// merge

void merge(int \*arr, int l, int m, int r)

{

int i, j, k;

int n1 = m - l + 1;

int n2 = r - m;

int L[n1], R[n2]; // temp arrays for partition in two halves

for (i = 0; i < n1; i++)

L[i] = arr[l + i];

for (j = 0; j < n2; j++)

R[j] = arr[m + 1+ j];

// initial indexes

i = 0;

j = 0;

k = l;

while (i < n1 && j < n2)

{

if (L[i] <= R[j])

{

arr[k] = L[i];

i++;

}

else

{

arr[k] = R[j];

j++;

}

k++;

}

// copy remaining items of left array

while (i < n1)

{

arr[k] = L[i];

i++;

k++;

}

// copy remaining items of right array

while (j < n2)

{

arr[k] = R[j];

j++;

k++;

}

}

void mergeSort(int \*arr, int l, int r)

{

if (l < r)

{

int m = l+(r-l)/2;

// Sort first and second halves

#pragma omp parallel sections num\_threads(2) // Two threads are used here

{

#pragma omp section

mergeSort(arr, l, m);

#pragma omp section

mergeSort(arr, m+1, r);

}

merge(arr, l, m, r);

}

}

// quick

int partition (int arr[], int low, int high)

{

int i,j,pivot = arr[high],temp; // pivot

i = low - 1; // Index of smaller element

for(j = low; j <= high - 1; j++)

{

// If current element is smaller than or equal to pivot

if (arr[j] <= pivot)

{

i++;

temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

// swap part

temp = arr[i + 1];

arr[i + 1] = arr[high];

arr[high] = temp;

return (i + 1);

}

void quickSort(int arr[], int low, int high)

{

if (low < high)

{

int pi = partition(arr, low, high);

#pragma omp parallel sections num\_threads(2)

{

#pragma omp section

quickSort(arr, low, pi - 1);

#pragma omp section

quickSort(arr, pi + 1, high);

}

}

}

// insertion

void insertionSort(int \*A, int num)

{

int k;

#pragma omp parallel for shared(A) private(k)

for(int n = 1; n < num; n++)

{

int key = A[n];

k = n;

#pragma omp critical

for(;k>0 && A[k-1]> key;k--)

{

A[k] = A[k-1];

}

A[k] = key;

}

}

// driver

int main()

{

fstream fin, fout;

int \*mArray = new int [MAX]; //merge array

int \*qArray = new int [MAX]; //quick array

int \*iArray = new int [MAX]; //insertion array

int i, n;

char order;

clock\_t t1, t2;

pthread\_t threads[MAX\_THREAD];

cout << "Enter 1 for Merge Sort" << endl

<< "Enter 2 for Quick Sort" << endl

<< "Enter 3 for Insertion Sort :";

cin >> n;

system("CLS");

switch (n)

{

case 1:

{

fin.open("data.txt", ios::in);

for(i=0; i<MAX; i++)

{

fin >> mArray[i];

}

fin.close();

fout.open("Merge\_Output.txt", ios::out | ios::app);

t1 = clock();

mergeSort(mArray, 0, MAX - 1);

t2 = clock();

double t\_time = (double)(t2-t1)/CLOCKS\_PER\_SEC;

fout << "Sorted Array: " << endl;

for(i=0; i<MAX; i++)

{

fout << mArray[i] << " ";

}

fout << endl;

fout << "\t\tTime Elapsed: " << t\_time << endl;

fout.close();

break;

}

case 2:

{

fin.open("data.txt", ios::in);

for(i=0; i<MAX; i++)

{

fin >> qArray[i];

}

fin.close();

fout.open("Quick\_Output.txt", ios::out | ios::app);

t1 = clock();

quickSort(qArray, 0, MAX-1);

t2 = clock();

double t\_time = (double)(t2-t1)/CLOCKS\_PER\_SEC;

fout << "Sorted Array: " <<endl;

for(i=0; i<MAX; i++)

{

fout << qArray[i] << " ";

}

fout << endl;

fout << "\t\tTime Elapsed: " << t\_time << endl;

fout.close();

break;

}

case 3:

{

fin.open("data.txt", ios::in);

for(i=0; i<MAX; i++)

{

fin >> iArray[i];

}

fin.close();

fout.open("Insertion\_Output.txt", ios::out | ios::app);

t1 = clock();

insertionSort(iArray, MAX);

t2 = clock();

double t\_time = (double)(t2-t1)/CLOCKS\_PER\_SEC;

fout << "Sorted Array: " << endl;

for(i=0; i<MAX; i++)

{

fout << iArray[i] << " ";

}

fout << endl;

fout.precision(10);

fout << "\t\tTime Elapsed: " << t\_time << endl;

fout.close();

break;

}

default:

{

cout << "Invalid command entered" << endl;

break;

}

}

cout << "Output saved in File. Do you want to try more algorithms? (y/n): ";

cin >> order;

if(order == 'y' || order == 'Y')

{

main();

}

else

{

system("CLS");

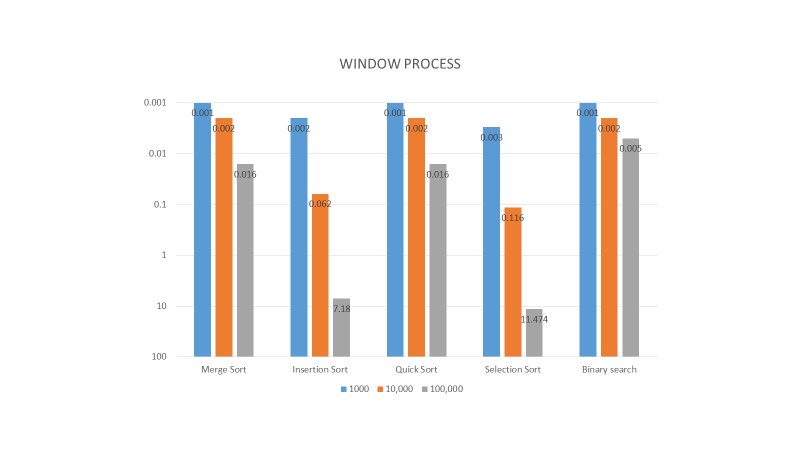
cout << "Thanks, See You !!" << endl;

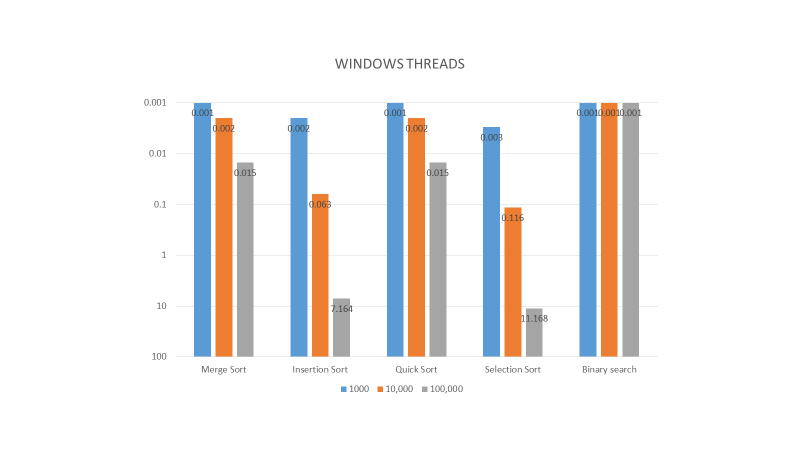
}

return 0;

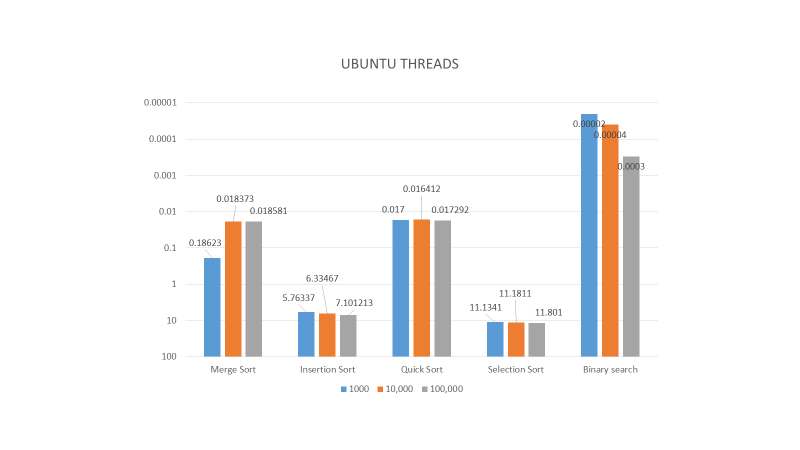
}

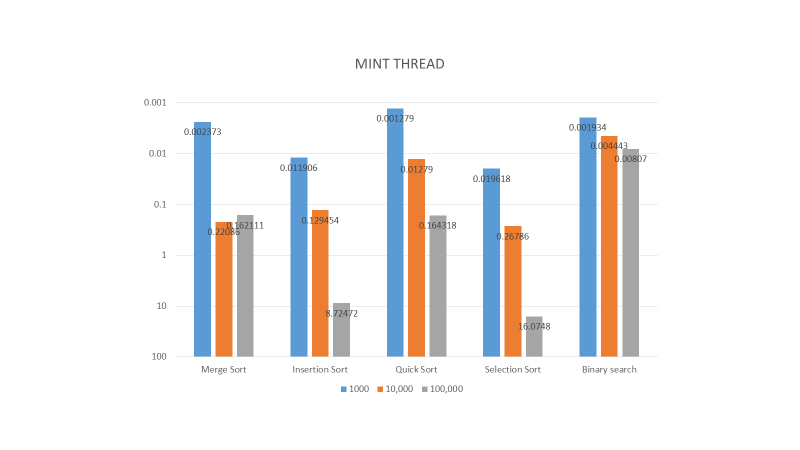
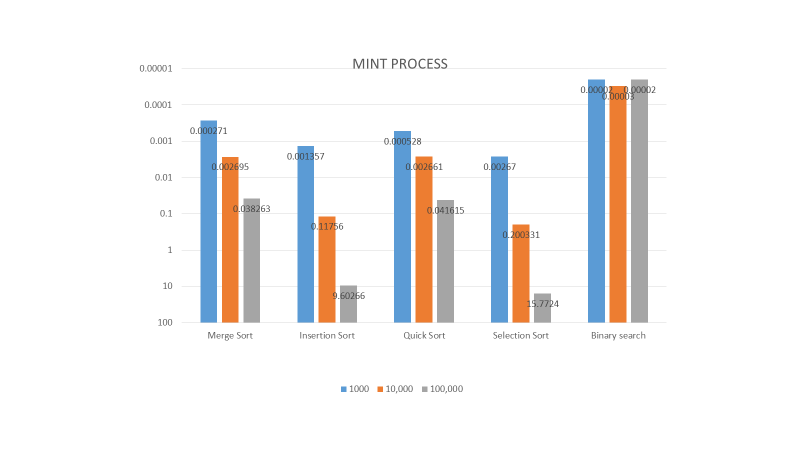
**GRAPHS on Different Operating Systems:**

****









**CONCLUSION**

As observed from the above graphs, I can conclude that upon increasing data size, the performance of threads is better than the performance of process.

**REFERENCES:**

<https://www.ibm.com/support/knowledgecenter/SSGH2K_13.1.2/com.ibm.xlc1312.aix.doc/compiler_ref/prag_omp_parallel.html>

<https://www.youtube.com/watch?v=nE-xN4Bf8XI&list=PLLX-Q6B8xqZ8n8bwjGdzBJ25X2utwnoEG>

<https://www.openmp.org/>