**OS PROJECT (IT308)**

**ID 1: 201501123 Ashray Bharambe**

**ID 2: 201501128 Shirish Jain**

**Multithread\_Synchronization\_MergeSort\_Barrier**

**Context :** Almost all of the modern computers today have a CPU with multiple cores, providing extra computational power. In the new age of big data, parallel execution is essential to improve the performance to an acceptable level.

**What is parallel programming & Why parallel programming ?**

Parallel Programming is an emerging computer science field that studies the opportunity of splitting data into small chunks and process them on multiple processors simultaneously which provides a faster execution time. Parallel programming is useful in sorting, image processing, network processing and many other memory intensive tasks.

For parallel program execution to be effective and fruitful, we need to run programs on computers with multiple CPUs or cores. Most people have computers with multiple processors and they are interested in high speed execution. This opens the way for parallel programming.

Having a sequential program running on a machine with multiple processors does not mean it will run faster than on a single processor computer. But why? Let’s say we have five processors on a machine, a running sequential program would leave four CPUs untouched and run on only one processor. This will execute in the same amount of time it would take on a single processor machine. On the other hand, a parallel program designed to run on multiple processors can perform the same task in less time.

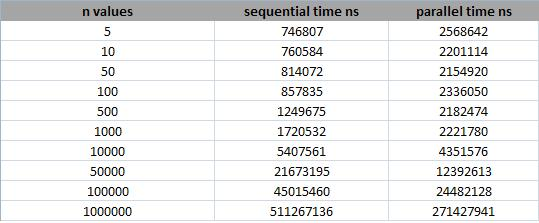
**Objective:** In this project, parallel algorithm is compared and analysed in relation to it’s sequential counterpart. Through this, find the potential speedup for multithreading and what factors affects the performance. In addition, provide source code for multithreaded algorithms with proven time complexities

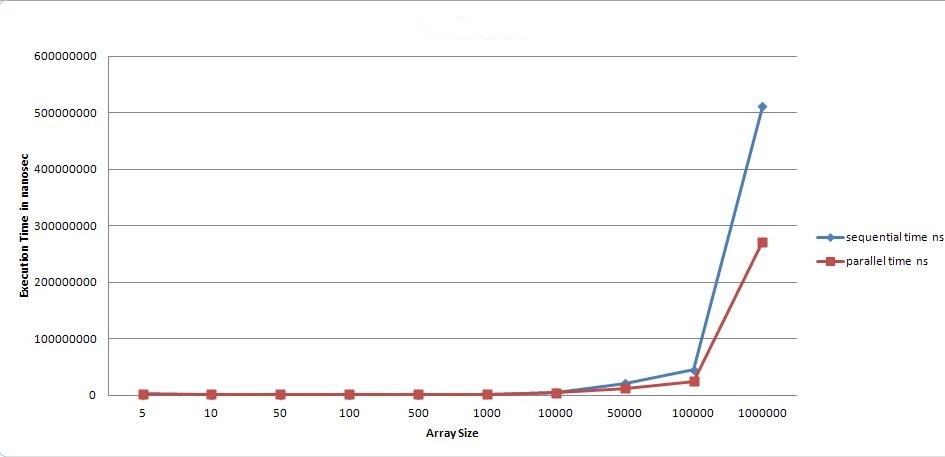
**Using Barrier Synchronization**

In cases where you must wait for a number of tasks to be completed before an overall task can proceed, **barrier synchronization** can be used. POSIX threads specifies a synchronization object called a **barrier**, along with barrier functions. The functions create the barrier, specifying the number of threads that are synchronizing on the barrier, and set up threads to perform tasks and wait at the barrier until all the threads reach the barrier. When the last thread arrives at the barrier, all the threads resume execution.

**RESULTS**

Test cases and the results of execution time for different values for n





**CONCLUSION**

* When we split a large array into equal parts and apply efficient sorting functions on sub arrays in parallel, then parallel execution results in faster processing. It takes less time to merge all sorted arrays that have been processed quickly in separate thread in parallel
* For small values of "n" the sequential execution was faster than the parallel. Why? Because the creation time for two threads in the parallel program plus the merging time is more than just the merging time in the sequential program.
* For large values (>=100000) of “n” the parallel execution was much better than the sequential due to the use of multithreading and parallel programming.

**CODE**

#define \_XOPEN\_SOURCE 600

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <string.h>

#include <stdbool.h>

#include <time.h>

#define numThreads 2048

int array[numThreads\*2];

int dup\_array[numThreads\*2];

typedef struct

{

int left;

int right;

}parameters;

parameters \*para[numThreads];

int arraySize=0;

pthread\_barrier\_t barr; // barrier variable

pthread\_mutex\_t lock; // define a mutual exclusion lock for threads.

void readArray(){

int i=0;

time\_t t;

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

arraySize = rand() % 1001;

for(i=0;i<arraySize;i++){

array[i] = rand() % 1001;

}

int ind =0;

for(ind=0;ind<arraySize;ind++){

dup\_array[ind] = array[ind];

}

arraySize=i;

printf("arraySize = %d \n",arraySize);

}

bool checkArraySize(void){

int n=arraySize;

int i=0;

while(n!=1){

i++;

n=n>>1;

}

n=arraySize;

if(((n>>i)<<i)==n){

return true;

}

else{

long long req\_size = (long long)1 << (i+1);

int ind = 0;

for(ind = arraySize;ind < req\_size;ind++){

array[ind] = INT\_MAX;

}

arraySize = req\_size;

int itr = 0;

return true;

}

}

void \*mergeSort(void \*para){

int left,right,mid,i,j,k;

int \*leftArray,\*rightArray;

parameters \*para\_=para;

left=para\_->left;

right=para\_->right;

mid=(left+right)/2;

leftArray=(int \*)calloc(sizeof(int),right-left+1);

rightArray=(int \*)calloc(sizeof(int),right-left+1);

for(i=left;i<=mid;i++){

leftArray[i-left]=array[i];

}

for(j=mid+1;j<=right;j++){

rightArray[j-mid-1]=array[j];

}

i=left; j=mid+1; k=left;

while(i<=mid&&j<=right){

if(leftArray[i-left]<=rightArray[j-mid-1]){

array[k]=leftArray[i-left];

i++; k++;

}else{

array[k]=rightArray[j-mid-1];

j++; k++;

}

}

while(i<=mid){

array[k]=leftArray[i-left];

i++; k++;

}

while(j<=right){

array[k]=rightArray[j-mid-1];

j++; k++;

}

free(leftArray);

free(rightArray);

pthread\_barrier\_wait(&barr);

}

int main(int argc,char \*argv[]){

int i,j,k;

pthread\_t tid[numThreads]; // to store the id of each threads.

pthread\_mutex\_init(&lock,NULL); // initiate the mutual exclusion lock.

readArray();

if(checkArraySize()){

i=arraySize/2; // number of threads in this merge layer

int height = 1;

clock\_t start\_time,end\_time;

start\_time = clock();

while(1){

j=arraySize/i; // number of elements to be sorted in single thread

printf("Layer %d : Number of threads = %d , Number of elements in each thread = %d \n",height,i,j);

if(pthread\_barrier\_init(&barr,NULL,i+1)) // initiate barrier with a count of i+1;

{

printf("Could not create a barrier\n");

return -1;

}

for(k=0;k<i;k++){

para[k]=(parameters\*)malloc(sizeof(parameters));

para[k]->left=k\*j;

para[k]->right=(k+1)\*j-1;

if(pthread\_create(&tid[k],NULL,mergeSort,para[k])){

printf("Could not create thread\n");

return -1;

}

}

pthread\_barrier\_wait(&barr);// wait for all threads being finished.

pthread\_barrier\_destroy(&barr);

for(k=0;k<i;k++)free(para[k]);

if(i==1)break; // merge sort has been finished if i==1

else i=i/2; // merge sort for next layer

height++;

}

end\_time = clock();

printf("Time taken by merge sort using mulithreaded programming : %f \n",((double)(end\_time-start\_time))/CLOCKS\_PER\_SEC);

}

printf("\n");

pthread\_mutex\_destroy(&lock); // destroy the mutual exclusion lock.

}