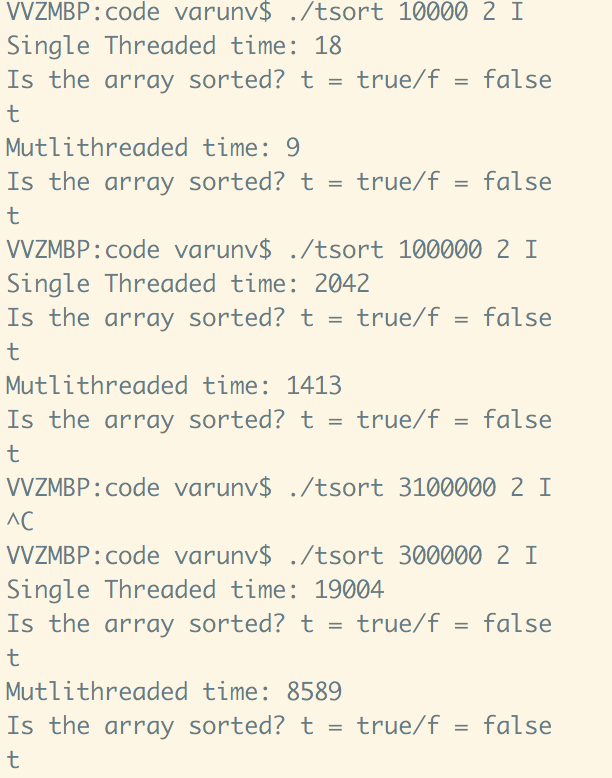
**Varun Ved**

**CSC 139-04**

**Multithreaded Sort Writeup**

**Timings + Screenshots**

**1. Run InsertionSort using two threads with array sizes 10K, 100K and 300K.**

VVZMBP:code varunv$ ./tsort **10000** 2 I

**Single Threaded time: 18**

Is the array sorted? t = true/f = false

t

**Mutlithreaded time: 9**

Is the array sorted? t = true/f = false

t

VVZMBP:code varunv$ ./tsort **100000** 2 I

**Single Threaded time: 2042**

Is the array sorted? t = true/f = false

t

**Mutlithreaded time: 1413**

Is the array sorted? t = true/f = false

t

VVZMBP:code varunv$ ./tsort **300000** 2 I

**Single Threaded time: 19004**

Is the array sorted? t = true/f = false

t

**Mutlithreaded time: 8589**

Is the array sorted? t = true/f = false

t

**2. Run InsertionSort using four threads with an array size of 100K.**

**VVZMBP:code varunv$ ./tsort 100000 4 I**

**Single Threaded time: 1034**

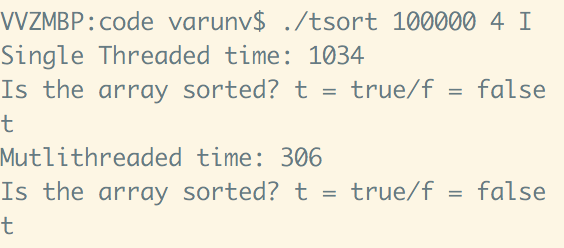
Is the array sorted? t = true/f = false

t

**Mutlithreaded time: 306**

Is the array sorted? t = true/f = false

t



**3. Run QuickSort using two threads with array sizes 1M, 10M and 100M.**

VVZMBP:code varunv$ ./tsort **1000000** 2 Q

**Single Threaded time: 132**

Is the array sorted? t = true/f = false

t

**Mutlithreaded time: 48**

Is the array sorted? t = true/f = false

t

VVZMBP:code varunv$ ./tsort **10000000** 2 Q

**Single Threaded time: 1675**

Is the array sorted? t = true/f = false

t

**Mutlithreaded time: 550**

Is the array sorted? t = true/f = false

t

VVZMBP:code varunv$ ./tsort **100000000** 2 Q

**Single Threaded time: 17322**

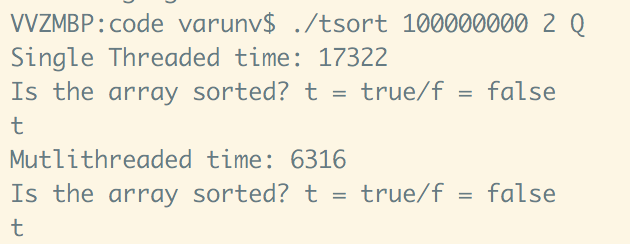
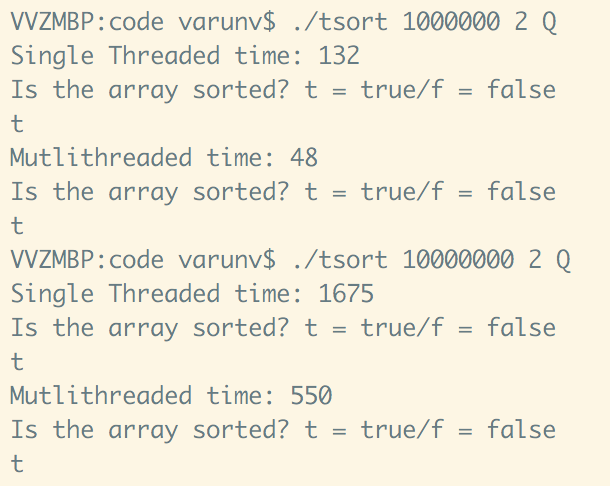
Is the array sorted? t = true/f = false

t

**Mutlithreaded time: 6316**

Is the array sorted? t = true/f = false

t



**4. Run QuickSort using four threads with an array size of 10M**

VVZMBP:code varunv$ ./tsort **10000000** 4 Q

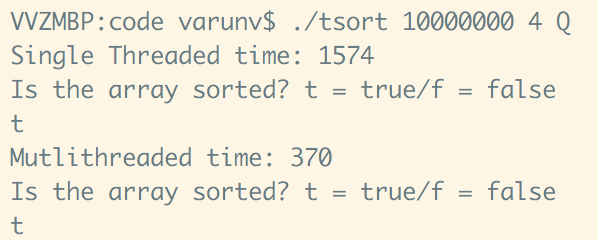
**Single Threaded time: 1574**

Is the array sorted? t = true/f = false

t

**Mutlithreaded time: 370**

Is the array sorted? t = true/f = false

t

**Tables**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| array size | 2 Threads Insertion | 4 threads Insertion | 2 Threads QS | 4 Threads QS |
| 10k | 18/9 |  |  |  |
| 100k | 2042/1413 | 1034/306 |  |  |
| 300k | 19004/8589 |  |  |  |
| 1m |  |  | 132/48 |  |
| 10m |  |  | 1675/550 | 1574/370 |
| 100m |  |  | 17322/6316 |  |

**2 thread insertion**

|  |  |
| --- | --- |
| array size | 2 Threads Insertion |
| 10k | 18/9 |
| 100k | 2042/1413 |
| 300k | 19004/8589 |
| 1m |  |
| 10m |  |
| 100m |  |

**4 thread insertion**

|  |  |
| --- | --- |
| array size | 4 threads Insertion |
| 10k |  |
| 100k | 1034/306 |
| 300k |  |
| 1m |  |
| 10m |  |
| 100m |  |

**2 threaded qs**

|  |  |
| --- | --- |
| array size | 2 Threads QS |
| 10k |  |
| 100k |  |
| 300k |  |
| 1m | 132/48 |
| 10m | 1675/550 |
| 100m | 17322/6316 |

**4 threaded qs**

|  |  |
| --- | --- |
| array size | 4 Threads QS |
| 10k |  |
| 100k |  |
| 300k |  |
| 1m |  |
| 10m | 1574/370 |
| 100m |  |

**Write-up**

As we can see from the table, different algorithms and array sizes reacted differently in relation to their algorithms provided. For insertion sort, doubling the number of threads made it run a bit more than twice as fast for the single threaded time on 100k elements, however on the multithread, it ran 4x as fast! Insertion sort didn't help with threads nearly as much as it did with quicksort. Quicksort however reacted very well to multiple threads as it performed faster than insertion on 10x as much data. However, adding more threads didn't help too much.

**Source-code**

//

//Varun Ved

//CSC 139-04

//tsort.cpp

#include <stdlib.h>

#include <string.h>

#include <sys/timeb.h>

#include <sys/types.h>

#include <stdio.h>

#include <pthread.h>

#include <unistd.h>

#define INT2VOIDP(i) (void\*)(uintptr\_t)(i)

//global vars

//consts

int threadCount;

int arraySize;

char algorithmChoice;

//timing

long gRefTime;

//arrays

int \*singleArray;

int \*threadedArray;

int \*copyArray;

//global util counters

int value;

int indices[16][2];

//thread vars

pthread\_attr\_t threadAttr;

pthread\_t threads[16];

int threadArr[16];

//method signatures- utils

void FixIndices();

int\* Merge(int arr[], int finalArr[]);

void Swap(int& x, int& y);

int Rand(int x, int y);

int IsSorted(int arr[]);

void printArray(int arr[]);

//method signatures- sorts

void InsertionSort(int arr[], int arraySize);

void QuickSort(int arr[], int left, int right);

void \*InsertionSortT(void \*index);

void \*QuickSortT(void \*index);

int Partition(int data[], int p, int r);

//getters and setters

long GetMilliSecondTime(struct timeb);

long GetCurrentTime(void);

void SetTime(void);

long GetTime(void);

void SetThreadCount(int);

int GetThreadCount();

void SetArraySize(int);

int GetArraySize();

void SetAlgorithmChoice(char);

char GetAlgorithmChoice();

/\*

\* @param int

\*/

void SetThreadCount(int passedThreadCount)

{

threadCount = passedThreadCount;

}

/\*

\* @param void

\* @returns int

\*/

int GetThreadCount(void)

{

return threadCount;

}

/\*

\* @param int

\*/

void SetArraySize(int passedArraySize)

{

arraySize = passedArraySize;

}

/\*

\* @param void

\* @returns int

\*/

int GetArraySize(void)

{

return arraySize;

}

/\*

\* @param char

\*/

void SetAlgorithmChoice(char passedAlgorithmChoice)

{

algorithmChoice = passedAlgorithmChoice;

}

/\*

\* @param char

\* @returns char

\*/

char GetAlgorithmChoice(void)

{

return algorithmChoice;

}

long GetMilliSecondTime(struct timeb timeBuf)

{

long mliScndTime;

mliScndTime = timeBuf.time;

mliScndTime \*= 1000;

mliScndTime += timeBuf.millitm;

return mliScndTime;

}

long GetCurrentTime(void)

{

long crntTime = 0;

struct timeb timeBuf;

ftime(&timeBuf);

crntTime = GetMilliSecondTime(timeBuf);

return crntTime;

}

void SetTime(void)

{

gRefTime = GetCurrentTime();

}

long GetTime(void)

{

long crntTime = GetCurrentTime();

return (crntTime - gRefTime);

}

/\*

\* @param int[]

\* @param int[]

\* @return int[]

\*/

int\* Merge(int arr[], int finalArr[]){

int lowest;

int increment = 0;

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

lowest = -1;

for (int j = 0; j < threadCount; j++){

if (indices[j][0] <= indices[j][1]){

if (lowest < 0)

{

//printf("low \n");

lowest = arr[indices[j][0]];

}

if (lowest >= arr[indices[j][0]]){

//printf("high \n");

lowest = arr[indices[j][0]];

increment = j;

}

}

//("%d" ,increment);

}

indices[increment][0] += 1;

finalArr[i] = lowest;

}

return finalArr;

}

/\*

\*

\*/

void FixIndices(){

value = arraySize/threadCount;

//printf("%d", value);

for (int i = 0; i < threadCount; i++){

indices[i][0] = i\*value;

indices[i][1] = (i+1)\*value-1;

}

}

/\*

\* @param int[]

\* @param int

\*/

void InsertionSort(int arr[], int index){

int j, temp, left, right;

left = indices[index][0] ;

right = indices[index][1];

for (int i = left; i <= right; i++){

j = i;

while (j > left && arr[j] < arr[j - 1]){

temp = arr[j];

arr[j] = arr[j - 1];

arr[j - 1] = temp;

j--;

}

}

}

/\*

\* @param int

\*/

void \*InsertionSortT(void \*index){

int j, temp, left, right;

left = indices[(size\_t)index][0] ;

right = indices[(size\_t)index][1];

for (int i = left; i <= right; i++){

j = i;

while (j > left && threadedArray[j] < threadedArray[j - 1]){

temp = threadedArray[j];

threadedArray[j] = threadedArray[j - 1];

threadedArray[j - 1] = temp;

j--;

}

}

pthread\_exit(0);

}

/\*

\* @param void

\*/

void \*QuickSortT(void \*index){

int left, right;

left = indices[(size\_t)index][0];

right = indices[(size\_t)index][1];

QuickSort(threadedArray, left, right);

pthread\_exit(0);

}

/\*

\* @param int[]

\* @param int

\* @param int

\* @returns int

\*/

void QuickSort(int data[], int p, int r){

int q;

if(p >= r) return;

q = Partition(data, p, r);

QuickSort(data, p, q-1);

QuickSort(data, q+1, r);

}

/\*

\* @param int[]

\* @param int

\* @param int

\* @return int

\*/

int Partition(int data[], int p, int r){

int i, j, x, pi;

pi = Rand(p, r);

Swap(data[r], data[pi]);

x = data[r];

i = p-1;

for(j=p; j<r; j++){

if(data[j] < x){

i++;

Swap(data[i], data[j]);

}

}

Swap(data[i+1], data[r]);

return i+1;

}

/\*

\* @param int&

\* @param int&

\*/

void Swap(int& x, int& y){

int temp = x;

x = y;

y = temp;

}

/\*

\* @param int

\* @param int

\* @return int

\*/

int Rand(int x, int y){

int range = y-x+1;

int r = rand() % range;

r += x;

return r;

}

/\*

\* @param int[]

\* @returns int

\*/

int IsSorted(int arr[]){

int correct;

for (int i = 1; i < arraySize; i++){

if (arr[i] >= arr[i-1]){

correct = 1;

}

else correct = -1;

}

return correct;

}

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

//parse args

if (argc < 3 || argc > 4)

{

printf("Too many or too few arguments passed, please enter arguments in the \n array size (1-100000), number of threads (1-16), the sorting algorithm (I or Q)");

exit(0);

}

//read and set array size

int tempArraySize = atoi(argv[1]);

if ((tempArraySize <= 1 || tempArraySize >=1000000000) && (sizeof(tempArraySize) != sizeof(size\_t)))

{

printf("Your array size is out of range, exiting");

exit(0);

}

else

{

SetArraySize(tempArraySize);

//printf("%d", arraySize);

}

//read and set thread count

int tempThreadCount = atoi(argv[2]);

if ((tempThreadCount <= 1 || tempThreadCount >=16) && (sizeof(tempThreadCount) != sizeof(size\_t)))

{

printf("Your thread count size is out of range, exiting");

exit(0);

}

else

{

SetThreadCount(tempThreadCount);

//printf("%d", threadCount);

}

//read and set thread count

char\* tempSortingAlgorithm = argv[3];

char sortc = tempSortingAlgorithm[0];

if ((sortc == 'I') || (sortc == 'Q'))

{

SetAlgorithmChoice(sortc);

//printf("%c", algorithmChoice);

}

//arraySize

singleArray = new int[arraySize];

threadedArray = new int[arraySize];

copyArray = new int[arraySize];

//1. Fill the array with random numbers

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

singleArray[i] = Rand(0, arraySize);

threadedArray[i] = Rand(0, arraySize);

}

//2. Based on the number of threads (T), compute the indicies for dividing the array into

//T equal parts. For example, if the array size (N) is 1000 and T is 2, the indicies will be 0,

//499, 500, 999.

FixIndices();

//3. Sort sequentially by applying the sorting algorithm to each part of the array, and

//then combining the sorted parts into one sorted array using an O(n) algorithm

char sortingAlgorithm = GetAlgorithmChoice();

if (sortingAlgorithm == 'I')

{

SetTime();

for (int i = 0; i < threadCount; i++){

InsertionSort(singleArray, i);

}

Merge(singleArray, copyArray);

gRefTime = GetTime();

}

else if (sortingAlgorithm == 'Q')

{

SetTime();

QuickSort(singleArray, 0, arraySize);

for (int i = 0; i < threadCount; i++){

QuickSort(singleArray, indices[i][0], indices[i][1]);

}

Merge(singleArray, copyArray);

gRefTime = GetTime();

}

printf("Single Threaded time: %ld\n", gRefTime);

// 4. Apply an O(n) algorithm to check if the array has been sorted correctly, and print a

// message indicating correct/incorrect sorting

int sortedVal = IsSorted(singleArray);

char sortedChar;

if (sortedVal == 1)

{

sortedChar = 't';

}

else sortedChar = 'f';

printf("Is the array sorted? t = true/f = false \n");

printf("%c\n", sortedChar);

//5. refill array with random numbers

FixIndices();

SetTime();

// 6. Sort using multi-threading. This step should be done the same way you did

// sequential sorting with the only difference that the sorting of each part is done in a separate

// thread. Combining the sorted parts into one sorted array should be done in the main (parent)

// thread after all child threads have completed. So, the parent thread must wait for all child

// threads.

for (int i = 0; i < threadCount; i++){

int pthreadC;

pthread\_attr\_init(&threadAttr);

threadArr[i] = i;

//for all the p\_threads in our array, go ahead and create a pthread

if (sortingAlgorithm == 'I')

{

pthreadC = pthread\_create(&threads[i], &threadAttr, InsertionSortT, (void \*) i);

}

else if(sortingAlgorithm == 'Q')

{

pthreadC = pthread\_create(&threads[i], &threadAttr, QuickSortT, (void \*) i);

}

if(pthreadC)

{

printf("Thread failed, %d\n", pthreadC);

exit(0);

}

}

//join threads after children have completed

for (int i = 0; i < threadCount; i++){

pthread\_join(threads[i],NULL);

}

Merge(threadedArray,copyArray);

//end time

gRefTime = GetTime();

printf("Mutlithreaded time: %ld\n", gRefTime);

//7. Apply an O(n) algorithm to check if the array has been sorted correctly, and print a

//message indicating correct/incorrect sorting.

int isSortMT = IsSorted(copyArray);

char sortedCharMT;

if (isSortMT == 1)

{

sortedCharMT = 't';

}

else sortedCharMT = 'f';

printf("Is the array sorted? t = true/f = false \n");

printf("%c\n", sortedCharMT);

}