|  |  |  |
| --- | --- | --- |
|  | **FACULTY of COMPUTING, ENGINEERING & SCIENCE** | Final mark awarded:\_\_\_\_\_ |

**Assessment Cover Sheet and Feedback Form 2016/17**

|  |  |  |  |
| --- | --- | --- | --- |
| Module Code:CS3S601 | Module Title: Concurrent and Parallel Programming | | Module Lecturer: C. W. Morris |
| Assessment Title and Tasks: Threads | | | Assessment No. 1 of 2 |
| No. of pages submitted in total including this page:  **31** | | | Word Count of submission  (if applicable): **7799 (excluding cover sheet)** |
| Date Set:  25/11/16 | | Submission Date: 13/01/17 | Return Date: Usually within 20 working days of submission |

|  |  |
| --- | --- |
| ***Part A: Record of Submission (to be completed by Student)*** | |
| **Extenuating Circumstances**  If there are any exceptional circumstances that may have affected your ability to undertake or submit this assignment, make sure you contact the Advice Centre on your campus prior to your submission deadline. | |
| **Fit to sit policy**:  The University operates a fit to sit policy whereby you, in submitting or presenting yourself for an assessment, are declaring that you are fit to sit the assessment. You cannot subsequently claim that your performance in this assessment was affected by extenuating factors. | |
| **Plagiarism and Unfair Practice Declaration:**  By submitting this assessment, you declare that it is your own work and that the sources of information and material you have used (including the internet) have been fully identified and properly acknowledged as required[[1]](#footnote-1). Additionally, the work presented has not been submitted for any other assessment. You also understand that the Faculty reserves the right to investigate allegations of plagiarism or unfair practice which, if proven, could result in a fail in this assessment and may affect your progress. | |
| **Intellectual Property and Retention of Student Work:**  You understand that the University will retain a copy of any assessments submitted electronically for evidence and quality assurance purposes; requests for the removal of assessments will only be considered if the work contains information that is either politically and/or commercially sensitive (as determined by the University) and where requests are made by the relevant module leader or dissertation supervisor. | |
| **Details of Submission:**  Note that all work handed in after the submission date and within 5 working days will be capped at 40%[[2]](#footnote-2). No marks will be awarded if the assessment is submitted after the late submission date unless extenuating circumstances are applied for and accepted (Advice Centre to be consulted). | |
| You are required to acknowledge that you have read the above statements by writing your student number(s) in the box: | Student Number(s):  **14009730** |

# Concurrent and Parallel Programming: Threads

## Introduction

A process is defined by Silberschatz et al. (1985, p. 82) as “a program in execution”. A process is fundamentally a task that a CPU works on. Normally, a system will run many processes at one time, in a way that appears to be simultaneous. Each process will be allocated some time to execute on the CPU, before being pre-empted (interrupted) by the operating system to hand over control of the CPU to another process.

Silberschatz et al. (1985, p. 82) state that a process stored in memory consists of multiple sections. These sections are the stack (used for storage of that process’s local variables, return addresses and function parameters), the heap (which is used for dynamic allocation of memory during run-time), the data section (which stores global variables), and the text or code section (which holds that process’s executable code). The OS also maintains a process control block for each process in the system. This process control block stores data used to manage the process, such as the process’s state, associated CPU registers and their contents, program counter value, CPU-scheduling and memory-management information etc. (Silberschatz et al., 1985, pp. 83-84).

The word “thread” refers to a “thread of execution”, and it is essentially the smallest sequence of instructions representing a discrete task that can be scheduled by the OS to run on the CPU. Threads are very similar to processes, but there are a few differences. A thread always belongs to a process. A process can have multiple threads, each of which can be scheduled to run on the CPU with some degree of independence. Each thread has its own unique thread ID, program counter, register data and stack (Silberschatz et al., 1985, p.127). However, threads that belong to the same process share the same memory space. As a result, data can be shared between them easily (data in each thread’s stack and heap is available to related threads provided they have a pointer to its location).

The main benefit of splitting up a process into threads is that threads can be scheduled to run on the CPU separately, in a concurrent fashion. This allows a process to make progress with multiple jobs it has to perform, without holding up the rest of the process while waiting for jobs to complete. For example, if a process is used to run a text editor, it could be split into two threads, one to listen for and handle user input, while another displays text on the screen. Silberschatz et al. (1985, p. 129) identify some benefits of multi-threading including:

* Improved responsiveness to the user
* Efficiency when creating, managing and switching between threads (compared to doing the same for separate processes)
* Effective use of multiple CPUs (since each CPU can run a separate thread of a process, whereas a single process must run on only one CPU)

## Designs and code

## Code for shared functions

Some identical functions have been used in each of the three implementations of the coursework. These functions are used to handle memory management for 2D arrays allocated on the heap, and to populate the main array by importing the numbers from a text file containing randomly generated numbers. In order to save space and cut down duplication of code, the definitions of these identical functions have been extracted from cw1Part1.cpp, cw1Part2.cpp and cw1Part3.cpp and placed here instead.

### Prototypes

template <typename type> type\*\* setup2DArrayOnHeap(void); //templated function to setup a 2D array on heap (must allocate arrays on heap due to their large size)

template <typename type> void delete2DArray(type\*\* array); //templated function to release memory used for a given 2D array (must be called for each array)

float\*\* setupMainArray(void); //used for importing and converting data for main array from array.txt and storing it in mainArray

### Pseudocode

**setup2DArrayOnHeap function:**

**Allocate an array of pointers on the heap (of size equal to required number of rows)**

**For each element (pointer) in array**

**Allocate an array of required type on the heap (of size equal to required number of columns)**

**Set the element to point to first element of this new array**

**delete2DArray function:**

**For each row of 2D array**

**Deallocate memory used for that row**

**Deallocate memory used for the array of pointers to first element of each row (representing a column)**

**setupMainArray function:**

**Open text file for reading**

**Call setup2DArray to set up array to store numbers**

**Call setup2DArray to set up array to temporarily store string data read in from text file**

**For each row of array**

**Set loop counter to 0**

**Read a character from the file**

**While character is not a newline character**

**If character is not a space**

**Append the character to the end of the corresponding element in row**

**Else**

**Increment loop counter**

**Read another character from the file**

**For each row of string array**

**For each element in row**

**Convert element in string array to a float**

**Store converted element in the number array in corresponding element**

**Call delete2DArray to deallocate the memory for the temporary string array**

### Definitions

//set up 2D array on the heap which matches the array dimensions ARRAY\_HEIGHT and ARRAY\_WIDTH

template <typename type>

type\*\* setup2DArrayOnHeap(void)

{

//sets up a double-pointer and points it to a dynamically allocated array of pointers

type\*\* newArray = new type\*[ARRAY\_HEIGHT];

//sets each element of newArray to point to a dynamically allocated array of variables of chosen type

for (int i = 0; i < ARRAY\_HEIGHT; i++)

newArray[i] = new type[ARRAY\_WIDTH];

//return double-pointer to array created

return newArray;

}

template <typename type>

void delete2DArray(type\*\* array)

{

//iterate through every row of 2D array and free memory used for each row using array deallocation operator

for (int i = 0; i < ARRAY\_HEIGHT; i++)

delete[] array[i];

//free memory used to hold the column data for the 2D array

delete[] array;

}

float\*\* setupMainArray(void)

{

//import data for main 2D array from text file

ifstream inFile;

inFile.open("array.txt");

//main array to hold the numbers to be operated on by threads

float\*\* mainArray = setup2DArrayOnHeap<float>();

//temporary storage for numbers as strings before they are converted to floats

string\*\* numbersAsStrings = setup2DArrayOnHeap<string>();

//imports data from array.txt into numbersAsStrings 2D array

for (int i = 0; i < ARRAY\_HEIGHT; i++)

{

char character = inFile.get();

int columnIndex = 0;

//looks at file char by char

//checks for newline and space chars to delimit input

while (character != '\n')

{

if (character != ' ')

numbersAsStrings[i][columnIndex] += character;

else

columnIndex++;

character = inFile.get();

}

}

//convert strings imported from array.txt to floats (using string to float function) and store them in main array

for (int i = 0; i < ARRAY\_HEIGHT; i++)

for (int j = 0; j < ARRAY\_WIDTH; j++)

mainArray[i][j] = stof(numbersAsStrings[i][j]);

//deallocate memory used for numbersAsStrings as it is no longer needed

delete2DArray<string>(numbersAsStrings);

return mainArray;

}

## cw1Part1.cpp

### Pseudocode

**Declare global constants storing:**

**Array width**

**Array height**

**Horizontal distance between adjacent points in each row of array**

**main function:**

**Record current CPU time program has used**

**Call setupMainArray function to set up main array (storing numbers read in from text file)**

**Call setup2DArrayOnHeap function to set up the distance and angle result arrays**

**For each row of main array**

**For each element in row**

**Set next element to compare with as neighbour directly to the right of current element**

**If current element is last in its row**

**Set next element to compare with as the first element in that row instead**

**Calculate vertical distance between current element and next element by subtracting one’s value from the other’s value**

**Calculate distance between current and next elements using Pythagoras’ Theorem (treating distance as hypotenuse of right-angled triangle). Length of hypotenuse = square root (vertical distance squared + horizontal distance squared)**

**Store the calculated distance in the element of the distance array which corresponds to the current element’s position in the main array**

**Calculate angle of slope between the current and next elements in main array using trigonometry. Angle = Arc sine (vertical distance / hypotenuse length)**

**Store angle in appropriate element of angle array**

**Call delete2DArray to deallocate memory used for main array and two results arrays**

**Calculate CPU time used by program by subtracting start CPU time from current CPU time**

**Convert CPU time to seconds and output it**

### Code

#include <iostream>

#include <fstream>

#include <unistd.h>

#include <string>

#include <cmath>

#include <time.h>

using namespace std;

//height and width of 2D array of points

#define ARRAY\_WIDTH 1000

#define ARRAY\_HEIGHT 50000

//horizontal distance between heights stored in each row of main array

//remains the same for every point and its immediate neighbours in row

//so long as this value is constant, its actual number value is unimportant

#define HORIZONTAL\_POINT\_DIST 50

#define DEGREES\_PER\_RADIAN 57.2958

float\*\* setupMainArray(void); //used for importing and converting data for main array from array.txt and storing it in mainArray

template <typename type> type\*\* setup2DArrayOnHeap(void); //templated function to setup a 2D array on heap (must allocate arrays on heap due to their large size)

template <typename type> void delete2DArray(type\*\* array); //templated function to release memory used for a given 2D array (must be called for each array)

int main()

{

//create a clock object and set it equal to current processor time used by this process (measured in clock ticks)

clock\_t t = clock();

//double-pointers used to point to 2D arrays

//the 2D arrays created have been set up on the heap due to their large size

float\*\* mainArray = setupMainArray();

float\*\* distanceArray = setup2DArrayOnHeap<float>();

float\*\* angleArray = setup2DArrayOnHeap<float>();

//calculate distance results and populate corresponding array

for (int i = 0; i < ARRAY\_HEIGHT; i++)

{

for (int j = 0; j < ARRAY\_WIDTH; j++)

{

//set height value to compare with as that of next element in row

int nextColumn = j + 1;

//special case:

//if we are looking at last element in row, "wrap around" and compare it with first element in that row

if (j == ARRAY\_WIDTH - 1)

nextColumn = 0;

//calculate vertical distance between points being compared

float verticalDist = mainArray[i][nextColumn] - mainArray[i][j];

//Pythagoras' Theorem to calculate Euclidean distance between the points (hypotenuse of the triangle)

float hypotenuse = sqrt(verticalDist \* verticalDist + HORIZONTAL\_POINT\_DIST \* HORIZONTAL\_POINT\_DIST);

distanceArray[i][j] = hypotenuse;

//calculate angle of slope from one point to the next using basic trigonometry (theta = sin-1(opposite / hypotenuse))

angleArray[i][j] = DEGREES\_PER\_RADIAN \* asin(verticalDist / hypotenuse);

}

}

//release memory used for arrays before finishing program

delete2DArray<float>(mainArray);

delete2DArray<float>(distanceArray);

delete2DArray<float>(angleArray);

//set value of clock object to current processor time used minus processor time used at start of process

t = clock() - t;

//output time taken for program to complete (converting clock ticks to seconds)

cout << "The program took " << ((float)t / (float)CLOCKS\_PER\_SEC) << " seconds from start to finish.\n";

return 0;

}

## cw1Part2.cpp

### Pseudocode

**Declare global constants storing:**

**Array width**

**Array height**

**Horizontal distance between adjacent points in each row of array**

**Number of rows to process with each processRows function call**

**Declare new type (Arrays) that will hold pointers to each of the three arrays (main array, distance array and angle array)**

**main function:**

**Record current CPU time program has used**

**Call setupMainArray function to set up main array (storing numbers read in from text file)**

**Call setup2DArrayOnHeap function to set up the distance and angle result arrays**

**Create object of “Arrays” type**

**Store pointers to each array inside the “Arrays” object**

**Set current row of array to 0**

**While current row is less than the number of rows in the array**

**Call processRows function, passing it:**

**A pointer to the structure object storing pointers to the three arrays**

**The current row of the array**

**The number of rows it has to process**

**Increment current row by the number of rows processed by the processRows function**

**Call delete2DArray to deallocate memory used for main array and two results arrays**

**Calculate CPU time used by program by subtracting start CPU time from current CPU time**

**Convert CPU time to seconds and output it**

**processRows function:**

**If current row is greater than number of rows in the array**

**Output an error message and quit the program (because it is impossible to process any more rows)**

**Set number of rows processed to 0**

**While the current row is less than the number of rows in the array, and the number of rows left to process is not equal to 0**

**For each element in row**

**Set next element to compare with as neighbour directly to the right of current element**

**If current element is last in its row**

**Set next element to compare with as the first element in that row instead**

**Calculate vertical distance between current element and next element by subtracting one’s value from the other’s value**

**Calculate distance between current and next elements using Pythagoras’ Theorem (treating distance as hypotenuse of right-angled triangle). Length of hypotenuse = square root (vertical distance squared + horizontal distance squared)**

**Store the calculated distance in the element of the distance array which corresponds to the current element’s position in the main array**

**Calculate angle of slope between the current and next elements in main array using trigonometry. Angle = Arc sine (vertical distance / hypotenuse length)**

**Store angle in appropriate element of angle array**

**Decrement number of rows to process**

**Increment number of rows processed**

**Increment current row**

**Return number of rows processed**

### Code

#include <iostream>

#include <fstream>

#include <unistd.h>

#include <string>

#include <cmath>

#include <time.h>

using namespace std;

//height and width of 2D array of points

#define ARRAY\_WIDTH 1000

#define ARRAY\_HEIGHT 50000

//horizontal distance between heights stored in each row of main array

//remains the same for every point and its immediate neighbours in row

//so long as this value is constant, its actual number value is unimportant

#define HORIZONTAL\_POINT\_DIST 50

//number of rows of array to process at each function call to processRows

#define ROWS\_TO\_PROCESS 7

//used to convert return value of asin() from radians to degrees

#define DEGREES\_PER\_RADIAN 57.2958

//used to hold array pointers which will be passed to processRows function

//makes the processRows function parameter list tidier

struct Arrays

{

float\*\* mainArray;

float\*\* distanceArray;

float\*\* angleArray;

};

float\*\* setupMainArray(void); //used for importing and converting data for main array from array.txt and storing it in main array

template <typename type> type\*\* setup2DArrayOnHeap(void); //templated function to setup a 2D array on heap (must allocate arrays on heap due to their large size)

template <typename type> void delete2DArray(type\*\* array); //templated function to release memory used for a given 2D array (must be called for each array)

//calculates distances and angles for elements in the requested rows in the array

//returns the number of rows processed

int processRows(Arrays arrays, int currentRow, int rowsToProcess);

int main()

{

//create a clock object and set it equal to current processor time used by this process (measured in clock ticks)

clock\_t t = clock();

//double-pointers used to point to 2D arrays

//the 2D arrays created have been set up on the heap due to their large size

float\*\* mainArray = setupMainArray();

float\*\* distanceArray = setup2DArrayOnHeap<float>();

float\*\* angleArray = setup2DArrayOnHeap<float>();

//pack array pointers into a struct (for passing in to row processing function)

//this helps reduce the size of the parameter list for processRows()

Arrays arrays;

arrays.mainArray = mainArray;

arrays.distanceArray = distanceArray;

arrays.angleArray = angleArray;

//set current row to look at as row 0

int currentRow = 0;

//loop through mainArray, processing rows until we reach the end of the array

//processRows() will process as many as possible of ROWS\_TO\_PROCESS number of rows until it hits the end of the array

//a call to processRows() where "ROWS\_TO\_PROCESS" > "remaining rows in array" is safe

while (currentRow < ARRAY\_HEIGHT)

//increment currentRow by number of rows processed

currentRow += processRows(arrays, currentRow, ROWS\_TO\_PROCESS);

//release memory used for arrays before finishing program

delete2DArray<float>(mainArray);

delete2DArray<float>(distanceArray);

delete2DArray<float>(angleArray);

//set value of clock object to current processor time used minus processor time used at start of process

t = clock() - t;

//output time taken for program to complete (converting clock ticks to seconds)

cout << "The program took " << ((float)t / (float)CLOCKS\_PER\_SEC) << " seconds from start to finish.\n";

return 0;

}

int processRows(Arrays arrays, int currentRow, int rowsToProcess)

{

//make sure that currentRow is within bounds of array

if (currentRow >= ARRAY\_HEIGHT)

{

cout << "Cannot process row " << currentRow << " as it is beyond the bounds of the array!" << endl;

throw;

}

int rowsProcessed = 0;

//calculate distance results and populate corresponding array

while (currentRow < ARRAY\_HEIGHT && rowsToProcess != 0)

{

for (int j = 0; j < ARRAY\_WIDTH; j++)

{

//set height value to compare with as that of next element in row

int nextColumn = j + 1;

//special case:

//if we are looking at last element in row, "wrap around" and compare it with first element in that row

if (j == ARRAY\_WIDTH - 1)

nextColumn = 0;

//calculate vertical distance between points being compared

float verticalDist = arrays.mainArray[currentRow][nextColumn] - arrays.mainArray[currentRow][j];

//Pythagoras' Theorem to calculate Euclidean distance between the points (hypotenuse of the triangle)

float hypotenuse = sqrt(verticalDist \* verticalDist + HORIZONTAL\_POINT\_DIST \* HORIZONTAL\_POINT\_DIST);

arrays.distanceArray[currentRow][j] = hypotenuse;

//calculate angle of slope from one point to the next using basic trigonometry (theta = sin-1(opposite / hypotenuse))

arrays.angleArray[currentRow][j] = DEGREES\_PER\_RADIAN \* asin(verticalDist / hypotenuse);

}

rowsToProcess--;

rowsProcessed++;

currentRow++;

}

//return number of rows for which distance and angle calculations have been performed

return rowsProcessed;

}

## cw1Part3.cpp

### Pseudocode

**Declare global constants storing:**

**Array width**

**Array height**

**Horizontal distance between adjacent points in each row of array**

**Number of threads to create to process rows of the array**

**Declare ThreadData type that will hold:**

**Pointers to each of the three arrays (main array, distance array and angle array)**

**The index number of the current row of the array being processed**

**The number of rows each thread must process**

**The time each thread took to complete**

**main function:**

**Record current CPU time program has used**

**If the number of threads requested is more than the number of rows in the array, print an error message to the user and terminate the program**

**Call setupMainArray function to set up main array (storing numbers read in from text file)**

**Call setup2DArrayOnHeap function to set up the distance and angle result arrays**

**Calculate and output current CPU time used by program**

**Create an array of ThreadData objects of size equal to number of threads requested**

**For each element of ThreadData array**

**Set its array pointers to point to main array, distance array and angle array**

**Set value of current row to 0**

**Set number of rows for each thread to process = number of rows in main array / number of threads requested**

**Calculate number of rows that cannot be split up equally between threads by taking the remainder of the above calculation**

**Set current row of array to 0**

**Declare array of thread identifiers (of size equal to number of threads requested)**

**For each thread requested**

**Set threadData object’s current row to equal actual current row of array**

**If there are still “remainder” rows that have not been allocated to a thread**

**Set threadData object’s rows to process = previously calculated rows to process + 1**

**Decrement remainder row count**

**Increment current row by previously calculated rows to process + 1**

**Else**

**Set threadData object’s rows to process = previously calculated rows to process**

**Increment current row by previously calculated rows to process**

**Create thread (using processRows function as thread function) and pass in threadData object**

**Calculate and output current CPU time used by program**

**For each thread**

**Wait for thread to finish executing**

**Get thread’s return value (the time it took to complete) and output it to screen**

**Calculate and output CPU time taken to wait for all threads to finish**

**Call delete2DArray to deallocate memory used for main array and two results arrays**

**Calculate CPU time used by entire program by subtracting start CPU time from current CPU time**

**Convert CPU time to seconds and output it**

**processRows function:**

**Set up timer for thread**

**Get data passed in as part of the thread parameter (pointers to the arrays, current row of array, and number of rows for thread to process)**

**If current row is greater than number of rows in the array**

**Output an error message and quit the program (because it is impossible to process any more rows)**

**While the current row is less than the number of rows in the array, and the number of rows left to process is not equal to 0**

**For each element in row**

**Set next element to compare with as neighbour directly to the right of current element**

**If current element is last in its row**

**Set next element to compare with as the first element in that row instead**

**Calculate vertical distance between current element and next element by subtracting one’s value from the other’s value**

**Calculate distance between current and next elements using Pythagoras’ Theorem (treating distance as hypotenuse of right-angled triangle). Length of hypotenuse = square root (vertical distance squared + horizontal distance squared)**

**Store the calculated distance in the element of the distance array which corresponds to the current element’s position in the main array**

**Calculate angle of slope between the current and next elements in main array using trigonometry. Angle = Arc sine (vertical distance / hypotenuse length)**

**Store angle in appropriate element of angle array**

**Decrement number of rows to process**

**Increment current row**

**Calculate and return the CPU time it took for the thread to complete**

### Code

#include <iostream>

#include <fstream>

#include <unistd.h>

#include <string>

#include <cmath>

#include <pthread.h>

#include <time.h>

using namespace std;

//height and width of 2D arrays

#define ARRAY\_WIDTH 1000

#define ARRAY\_HEIGHT 50000

//horizontal distance between heights stored in each row of main array

//remains the same between every point and its immediate neighbours in row

//so long as this value is constant, its actual number value is unimportant

#define HORIZONTAL\_POINT\_DIST 50

//number of threads to split array between

#define NUM\_THREADS 4

//used to convert return value of asin() from radians to degrees

#define DEGREES\_PER\_RADIAN 57.2958

//a pointer to an object of this type will be passed to the thread function as a parameter whenever a thread is created

//a different object will be given to each thread - this is the easiest way to avoid a race condition when different threads

//are reading/writing to the currentRow and rowsToProcess member variables

struct ThreadData

{

float\*\* mainArray;

float\*\* distanceArray;

float\*\* angleArray;

int currentRow;

int rowsToProcess;

//used by a thread to return back to main thread the time it took to complete (return value accessed through pthread\_join())

//the only way a value can be returned from a thread is using a void pointer

//however, returning a pointer to local storage of a terminated thread will cause an access violation

//therefore, variable is stored locally in main thread instead

float timeTaken;

};

float\*\* setupMainArray(void); //used for importing and converting data for main array from array.txt and storing it in main array

template <typename type> type\*\* setup2DArrayOnHeap(void); //templated function to setup a 2D array on heap (must allocate arrays on heap due to their large size)

template <typename type> void delete2DArray(type\*\* array); //templated function to release memory used for a given 2D array (must be called for each array)

//thread function - takes a ThreadData pointer (which must be passed into the function as a void pointer)

void\* processRows(void\* data);

int main()

{

//create a clock object and set it equal to current processor time used by this process (measured in clock ticks)

clock\_t startTime = clock();

//make sure that number of threads requested is not greater than the number of rows in the array

//if it is, then terminate program (because otherwise useless threads will be created)

if (NUM\_THREADS > ARRAY\_HEIGHT)

{

cout << "Error! Number of threads requested is greater than the number of rows in the array." << endl;

return 1;

}

//double-pointers used to point to 2D arrays

//the 2D arrays created have been set up on the heap due to their large size (and so they can be easily shared between threads)

float\*\* mainArray = setupMainArray();

float\*\* distanceArray = setup2DArrayOnHeap<float>();

float\*\* angleArray = setup2DArrayOnHeap<float>();

//calculate and output elapsed time

clock\_t arrayAllocTime = clock() - startTime;

float time = ((float)arrayAllocTime / (float)CLOCKS\_PER\_SEC);

cout << "Allocation of arrays on the heap takes " << time << " seconds.\n";

//pack array pointers and other data into structs (for passing in to thread function)

//each thread will receive a separate copy of this data - this is the easiest way to avoid

//race conditions when different threads are reading and writing to the struct's currentRow and rowsToProcess members

ThreadData data[NUM\_THREADS];

//initialise members of ThreadData objects

for (int i = 0; i < NUM\_THREADS; i++)

{

data[i].mainArray = mainArray;

data[i].distanceArray = distanceArray;

data[i].angleArray = angleArray;

data[i].currentRow = 0;

}

//split up rows equally between threads and store results in rowsToProcess

int rowsToProcess = ARRAY\_HEIGHT / NUM\_THREADS;

//store number of rows that could not be split up equally between threads

//each thread created will be allocated one of these rows in addition to its normal workload (until no remainder rows are left)

int remainderRows = ARRAY\_HEIGHT % NUM\_THREADS;

//keeps track of current row in array so this data can be passed to threads

int currentRow = 0;

//create required number of thread identifiers

pthread\_t threads[NUM\_THREADS];

//for each thread to be created, set its data parameters and create it, passing in the data

for (int i = 0; i < NUM\_THREADS; i++)

{

data[i].currentRow = currentRow;

//give one of the extra rows to each thread in turn until all rows have been assigned to a thread

if (remainderRows != 0)

{

data[i].rowsToProcess = rowsToProcess + 1;

remainderRows--;

currentRow += rowsToProcess + 1;

}

//once extra rows have been dealt with, use else block to allocate each thread the normal number of rows to process

else

{

data[i].rowsToProcess = rowsToProcess;

currentRow += rowsToProcess;

}

//create thread using the relevant thread identifier and pass it a pointer to the thread data

pthread\_create(&threads[i], NULL, processRows, (void\*)&data[i]);

}

//calculate elapsed time from start to the point straight after the threads have been created

clock\_t threadCreationTime = clock() - startTime;

time = ((float)threadCreationTime / (float)CLOCKS\_PER\_SEC);

cout << "Up to the point where threads are joined, program has taken " << time << " seconds.\n";

//set up void pointer to hold thread return value

void\* threadReturnVal;

cout << "Thread run-time data:\n";

//join each thread back into parent thread, printing out each thread's time to completion

for (int i = 0; i < NUM\_THREADS; i++)

{

pthread\_join(threads[i], &threadReturnVal);

cout << "Thread " << i << " completed in " << ((ThreadData\*)threadReturnVal)->timeTaken << " seconds.\n";

}

//measure time taken to join the threads back together

clock\_t threadJoinTime = clock() - threadCreationTime;

time = ((float)threadJoinTime / (float)CLOCKS\_PER\_SEC);

cout << "Joining of threads takes " << time << " seconds.\n";

//release memory used for arrays before finishing program

delete2DArray<float>(mainArray);

delete2DArray<float>(distanceArray);

delete2DArray<float>(angleArray);

//calculate and output time taken for entire process to complete

clock\_t endTime = clock() - startTime;

cout << "The program took " << ((float)endTime / (float)CLOCKS\_PER\_SEC) << " seconds from start to finish.\n";

return 0;

}

void\* processRows(void\* data)

{

//get start CPU time of thread

clock\_t t = clock();

//cast pointer back to a pointer to object of type ThreadData

ThreadData\* threadData = (ThreadData\*)data;

//assign local copies of thread parameter data (purely for the sake of readability)

float\*\* mainArray = threadData->mainArray;

float\*\* distanceArray = threadData->distanceArray;

float\*\* angleArray = threadData->angleArray;

int currentRow = threadData->currentRow;

int rowsToProcess = threadData->rowsToProcess;

//make sure that currentRow is within bounds of array

if (currentRow >= ARRAY\_HEIGHT)

{

cout << "Cannot process row " << currentRow << " as it is beyond the bounds of the array!" << endl;

pthread\_exit(NULL);

}

//calculate distance results and populate corresponding array

while (currentRow < ARRAY\_HEIGHT && rowsToProcess != 0)

{

for (int j = 0; j < ARRAY\_WIDTH; j++)

{

//set height value to compare with as that of next element in row

int nextColumn = j + 1;

//special case:

//if we are looking at last element in row, "wrap around" and compare it with first element in that row

if (j == ARRAY\_WIDTH - 1)

nextColumn = 0;

//calculate vertical distance between points being compared

float verticalDist = mainArray[currentRow][nextColumn] - mainArray[currentRow][j];

//Pythagoras' Theorem to calculate Euclidean distance between the points (hypotenuse of the triangle)

float hypotenuse = sqrt(verticalDist \* verticalDist + HORIZONTAL\_POINT\_DIST \* HORIZONTAL\_POINT\_DIST);

distanceArray[currentRow][j] = hypotenuse;

//calculate angle of slope from one point to the next

angleArray[currentRow][j] = DEGREES\_PER\_RADIAN \* asin(verticalDist / hypotenuse);

}

rowsToProcess--;

currentRow++;

}

//calculate elapsed CPU time since thread began, convert it to seconds and assign it to threadData->timeTaken

t = clock() - t;

float seconds = (float)t / (float)CLOCKS\_PER\_SEC;

threadData->timeTaken = seconds;

//finish thread and return time taken to complete (using ThreadData pointer cast to void pointer)

return (void\*)threadData;

}

## Verification of calculations

### Code used for calculations (from cw1Part1)

//horizontal distance between heights stored in each row of main array

//remains the same between every point and its immediate neighbours in row

//so long as this value is constant, its actual number value is unimportant

#define HORIZONTAL\_POINT\_DIST 50

//used to convert return value of asin() from radians to degrees

#define DEGREES\_PER\_RADIAN 57.2958

//calculate vertical distance between points being compared

float verticalDist = mainArray[i][nextColumn] - mainArray[i][j];

//Pythagoras' Theorem to calculate Euclidean distance between the points (hypotenuse of the triangle)

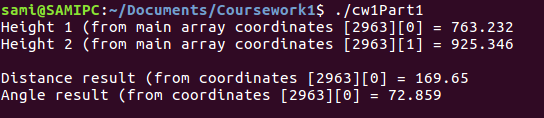
float hypotenuse = sqrt(verticalDist \* verticalDist + HORIZONTAL\_POINT\_DIST \* HORIZONTAL\_POINT\_DIST);

distanceArray[i][j] = hypotenuse;

//calculate angle of slope from one point to the next using basic trigonometry (theta = sin-1(opposite / hypotenuse))

angleArray[i][j] = DEGREES\_PER\_RADIAN \* asin(verticalDist / hypotenuse);

### Test 1 - Double-check calculations for array element [2,963][0]



Height at main array element [2,963][0] = 763.232

Height at next element in row = 925.346

Euclidean distance between heights = (Vertical distance2 + Horizontal distance2)

= ((925.346 – 763.232)2 + 502)

= ((162.114)2 + 502)

= (26,280.949 + 2500)

= (28,780.949)

=169.65

Angle of slope = Degrees per radian \* sin-1(Vertical distance / Euclidean distance or hypotenuse)

= 57.2958 \* sin-1((925.346 – 763.232) / 169.65)

= 57.2958 \* sin-1(162.114 / 169.65)

= 57.2958 \* sin-1(0.95557913)

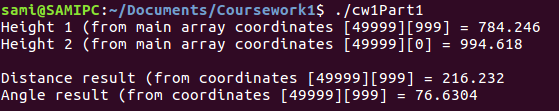
= 57.2958 \* 1.272

= 72.88

\*Arc sine function returns angles in radians, so angle must be multiplied by 57.2958 to convert it to degrees

The distance and angle values calculated between this point and the next by the program are correct, although there is a small difference in the angle value due to rounding errors in the by-hand calculation.

### Test 2 - Double-check calculations for array element [49,999][999] (last element in last row)



Height at main array element [49,999][999] = 784.246

Height at next element in row (wraps around to first element in row) = 994.618

Euclidean distance between heights = (Vertical distance2 + Horizontal distance2)

= ((994.618 – 784.246)2 + 502)

= ((210.372)2 + 502)

= (44,256.378 + 2500)

= (46,756.378)

=216.232

Angle of slope = Degrees per radian \* sin-1(Vertical distance / Euclidean distance or hypotenuse)

= 57.2958 \* sin-1((994.618 – 784.246) / 216.232)

= 57.2958 \* sin-1(210.372 / 216.232)

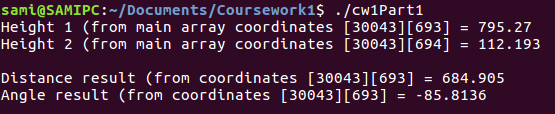
= 57.2958 \* sin-1(0.97289948)

= 57.2958 \* 1.337

= 76.604

Distance and angle results are correct, but again, there is a small rounding error for the angle value in the by-hand version of the calculation.

### Test 3 - Double-check calculations for array element [30,043][693]



Height at main array element [30,043][693] = 795.27

Height at next element in row = 112.193

Euclidean distance between heights = (Vertical distance2 + Horizontal distance2)

= ((112.193 – 795.27)2 + 502)

= ((-683.077)2 + 502)

= (466,594.188 + 2500)

= (469,094.188)

=684.905

Angle of slope = Degrees per radian \* sin-1(Vertical distance / Euclidean distance or hypotenuse)

= 57.2958 \* sin-1((112.193 – 795.27) / 684.905)

= 57.2958 \* sin-1(-683.077 / 684.905)

= 57.2958 \* sin-1(-0.99733102)

= 57.2958 \* -1.498

= -85.813

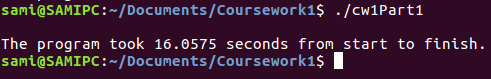
The distance and angle results both match the output of the program.

## Discussion of performance of each implementation

All three implementations deal with an array of 50,000 rows by 1000 columns, which gives a total of 50,000,000 (50 million) elements. Each implementation imports data to populate the main array from a text file. Each implementation also has two more arrays of equal size to the first, to store the distance and angle results for the calculations. Each implementation uses the “float” data type for each of the arrays. Because of the size of the arrays being dealt with, and also because of the fact that they need to be shared between threads, the arrays have been allocated in heap memory. It is important to note that the timing data gathered considers only the CPU time used by the process (or thread) itself. It makes no difference how much intervening CPU time is spent on other processes or threads.

### Machine with four-core CPU

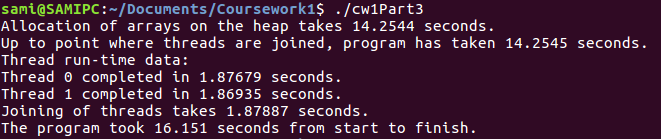
The machine on which the first batch of performance tests have been run has a CPU with four physical cores (each of which has one logical core). Running on this machine, each of the implementations has a total runtime of between 16 and 19 seconds. The first implementation (which processes the array sequentially in a procedural fashion) takes 16.0575 seconds from beginning to end.



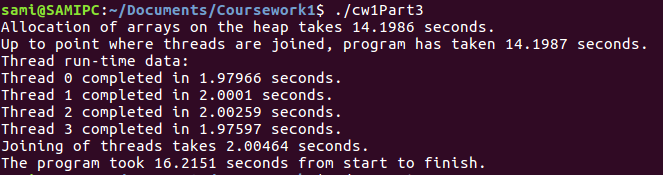
The second implementation (which processes the array through repeated calls to a “processRows” function) takes 16.9985 seconds to complete. It is 941 milliseconds slower than the procedural version, almost a whole second. This is likely to be because of the overhead from repeated function calls.

C:\Users\User\Documents\Concurrent & Parallel Programming\Coursework1\Screenshots\Desktop\cw1part2desktop.png

The third implementation (which uses threads that run concurrently) takes 16.151 seconds to complete when the array processing is split between two threads. It is very similar in speed to the procedural implementation, but 93.5 milliseconds slower. This is surprising, as we would expect at least some benefit from threading. The thread implementation remains faster than the function implementation, however. It can be seen that the vast majority of the execution time (14.2544 seconds) is spent importing array data from a text file and allocating the arrays in heap memory. This is to be expected, due to the size of the arrays and the text file, the overhead involved in loading files from slow secondary memory, and the overhead of allocating memory on the heap. Both threads then take around 1.9 seconds to process the data.



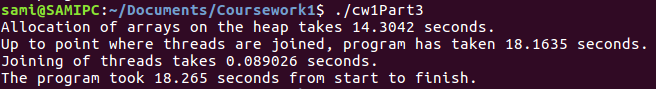
Since the CPU of this machine has four cores, we would normally expect to see improvement in performance if we split the array between four threads, since four threads can take full advantage of the available cores. When four threads are used, the implementation is still significantly faster than the sequential implementation. However, no improvement in performance is seen over the two-thread implementation, and in fact the version with four threads is 64.1 milliseconds slower. When looking at the threads’ timing data averages, it can be seen that the threads in the four-thread implementation take 116.51 milliseconds longer to complete than those in the two-thread implementation. It also takes 125.77 milliseconds longer to wait for threads to complete and join them back to the main thread.



The system seems to be utilising all four cores of the CPU, judging by the hardware information seen when using the “top” command. It seems likely that the reason no performance improvement has been seen from threading is that there is a bottleneck of some kind. The cause of this is difficult to determine. It could be due to ineffective parallelisation of the program, so that it is forced to run in a sequential fashion, or spend long amounts of time waiting for other threads to finish tasks, release locks for reading data etc. However, no locking of any kind has been implemented here. Looking at the code, it seems that the only place where the threads are prevented from running concurrently is during the thread joining loop, and at this point, each of the threads should be very close to completion anyway. Therefore, this seems unlikely to be the cause.

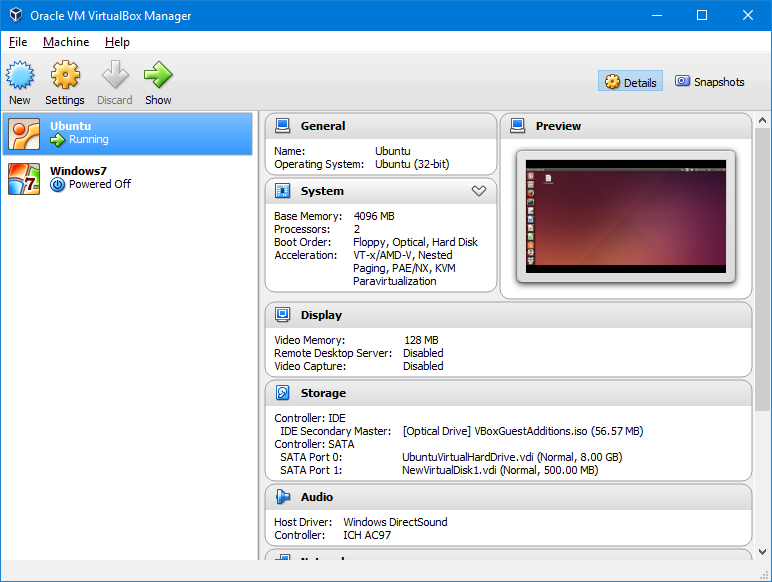
However, the main array and result arrays are each around 200MB in size (a float occupies 4 bytes of memory, and there are 50,000,000 elements in each array of floats). Each thread works on different sections of the array, and there is only limited CPU cache memory (approximately 6MB on this machine). Taking these facts into consideration, and also considering the fact that the use of threading produces no performance benefit of any kind, it seems likely that the process is memory-bound as opposed to CPU-bound, i.e. that the bottleneck is latency in main memory access rather than limited CPU processing speed. According to Hutcheson and Natoli (2011, pp. 4-8), it is common for performance to deteriorate sharply due to memory latency as sizes of arrays being processed become larger than the size of the CPU cache. This could be worsened by ineffective use of the limited CPU cache space, resulting from threads replacing array data being used by other threads (in the cache) whenever a context switch occurs (Robison, 2007).

When we increase the number of threads dramatically, we would expect to see further deterioration in performance due to overhead from creating threads and having a large number of threads competing for CPU time on a limited number of cores. The more threads there are, the more CPU time is wasted on context switching. With 5000 threads, the process takes 18.265 seconds to complete. This is 2.0499 seconds slower than the four-thread version, a significant difference in terms of CPU time. Using this many threads is detrimental to program performance.



### Virtual machine with two-core CPU

These tests have been conducted on Linux running under a virtual machine (which has been set up to use two CPU cores).



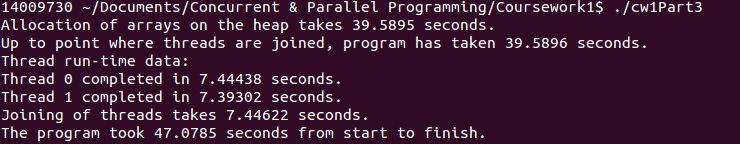
The procedural implementation takes 49.6235 seconds to complete, about three times as long as on the previous machine.

E:\Concurrent & Parallel Programming\Coursework1\Screenshots\Laptop\cw1part1laptop.png

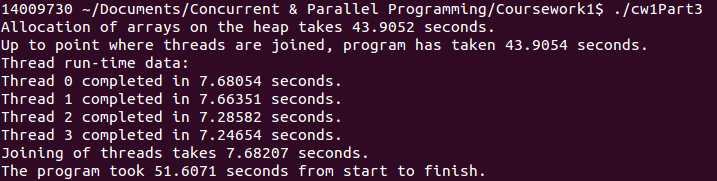
The repeated function call implementation takes 51.9833 seconds to complete. It is 2.3598 seconds slower than the procedural implementation, which is a similar outcome to the results on the previous machine. Again, this is likely due to the overhead from repeated function calls. The difference is larger in absolute terms this time, because the run-time of the process is longer on this slower machine.

E:\Concurrent & Parallel Programming\Coursework1\Screenshots\Laptop\cw1part2laptop.png

With two threads, the program takes 47.0785 seconds to complete. This is faster than the procedural implementation by 2.545 seconds. This is a significant difference in terms of CPU time, and becomes more substantial when ignoring the 39.5895 seconds taken for the program to set up the arrays (during which time the process is unable to run any code concurrently). When adjusting for this by considering only actual calculation time for both implementations, a performance improvement of 25% is seen over the procedural implementation. This is the only case where threading has provided a performance benefit to the application. The likely reason for this is that the process is a little more CPU-bound and less memory-bound on this machine; all else being equal, the CPU is slower on this machine. This being the case, adding a second thread to utilise the second core of the CPU seems to have provided some benefit.



With four threads, the program takes 51.6071 seconds, and is 4.5286 seconds slower than the two-thread version. However, the setup of the arrays (using exactly the same code) has taken 4.3157 seconds longer here, so this can be attributed to normal run-time variation. Accounting for this, the four-thread version is only 0.2129 seconds slower than the two-thread version. Still, having more threads than cores has given no benefit to the program, because in this case only two of the four threads can run at any one time on the CPU, and switching one thread out for another just incurs some overhead for no real gain.



## Conclusion

Threading can provide significant performance gains in some applications. In others, it provides little benefit. Whether threading can be beneficial in terms of performance depends on both the hardware the application is running on, and on what sort of tasks the application is carrying out. In general, threading provides the most benefit if the number of software threads in use is around the same as the number of CPUs (or CPU cores) in the machine. In the ideal case, moving from one thread to a number of threads equal to the number of CPU cores will scale the application’s run time down by that amount. In practice, however, gains of this magnitude are rare, because performance is limited by many factors.

Having more threads than cores can simply add context switching overhead without providing much benefit (since the maximum number of threads that can run at any one time is determined by the number of hardware threads available in the CPU). Likewise, if a machine has only one CPU core, then threading will only be useful in certain circumstances. For example, using threading to parallelise calculations will not help, because only one of these threads will be able to run at any one time. However, threading may still be used to make an application that uses a lot of CPU time on calculations more responsive to user input. This is done by splitting the process into one thread that does the calculations, and another thread that is dedicated to handling user input and can be scheduled to run separately from the main thread. In general, having more threads than cores will not make the application faster, but it can sometimes allow it to make more *useful* progress overall if it has multiple tasks to perform.

Performance gains from threading depend strongly on what sort of tasks the application carries out. If the application is CPU-bound, and performs many intensive calculations, then threading can reduce run-times, assuming that these calculations are independent of each other and suitable for parallelisation (and there are enough cores to run each thread). Whenever use of threading requires locking of shared resources, or waiting for other threads to complete, there is a reduction in performance gains from threading. This is because there are times where the application is forced to run in a sequential rather than concurrent manner. Also, in many kinds of applications, it is not the CPU speed that is the bottleneck or limiting factor, but some other aspect such as memory access speed or latency, network speed, user input/output etc. In these cases, little benefit will be seen from threading the application, since the CPU is able to handle what is required of it in suitable time whether or not threading is used.

The application for this project was developed in incremental stages that built on the complexity of the previous versions. First, the sequential version that performed the calculations and stored the results in the result arrays was developed. Then, this version was modified to encapsulate all of the processing into a function, which was called repeatedly to process the whole array in stages. Finally, the program was extended to use POSIX threads, with the function used to process rows becoming the thread function.

There were initial difficulties in appropriately importing data from a text file, converting it to floating point numbers and storing it in the main array. Also, setting up 2D arrays on the heap required some research and experimentation to implement correctly. However, the most challenging area of development was in implementing threading. Setting up threads was not hard, but there were issues with corruption of array data and incomplete processing of the array data.

These issues were due to a pointer to the same thread data object being passed to each thread. This resulted in a race condition, where each thread was reading and writing to the members of the data object used to keep track of the current row of the array being processed, and the number of rows for each thread to process. In order to work correctly, each thread had to perform its reads and writes before the next thread tried to access the data, which in practice, did not happen. This issue was solved by creating separate objects to pass to each thread. This involved the duplication of some data (three pointers and two integers).

In the end, little benefit was seen from threading. It seems likely that this is due to latency in memory access, because of the large size (approx. 200MB each) of the arrays being processed, and counter-productive use of the limited CPU cache space available (although the greatest bottleneck overall is loading the array data from the text file).

Locking is not used in this application because although each thread processes the same arrays, each thread handles a different set of rows. In order to guarantee that all threads have finished writing to the arrays, the main thread waits for all of the other threads to finish before reading from the arrays. However, in more complex situations with threads that have longer-running tasks, this would not be a suitable solution.

If locking were required in this application, then it would be necessary to identify the critical sections of code (anywhere that the shared arrays are read from or written to). These critical sections would need to be managed using semaphores, either a binary semaphore to ensure that only one thread can access the same array elements at any one time, or a counting semaphore to ensure that some other condition has been met before proceeding. The critical sections would be the section of code in the main thread that reads from the arrays, and the section of code in the thread function that writes to them.

If we wanted to ensure that another thread had finished writing to a section of the array before the main thread read it, we could create a binary semaphore or mutex that represents the current completion state of the operation. Before the “writer” thread writes to the array, this semaphore would be told to wait until the lock is open, then signal other threads to wait. Once the writer thread has finished writing, the semaphore would be told to signal completion so that other threads can access the array. Before the “reader” thread tries to read from the array, it would also wait until the lock is open, then signal other threads to wait. After the read is performed, it would open the lock again. This arrangement could be adapted to allow multiple threads to read at the same time, by maintaining a shared variable (appropriately protected with a mutex) that keeps track of the number of readers. The writer thread could be blocked with another mutex that is locked whenever there are readers, and only unlocked when the numbers of readers is 0.

## References

Hutcheson, A. and Natoli, V. (2011) *Memory Bound vs. Compute Bound: A Quantitative Study of Cache and Memory Bandwidth in High Performance Applications.* Available at: http://stoneridgetechnology.com/wp-content/uploads/2014/12/ComputevsMemory.pdf (Accessed: 29 December 2016).

Robison, A. (2007) *Why Too Many Threads Hurts Performance, and What to do About It.* Available at: http://www.codeguru.com/cpp/sample\_chapter/article.php/c13533/Why-Too-Many-Threads-Hurts-Performance-and-What-to-do-About-It.htm (Accessed: 29 December 2016).

Silberschatz, A., Galvin, P. and Gagne, G. (1985). *Operating system concepts.* Reading, Mass.: Addison-Wesley.

1. University Academic Misconduct Regulations [↑](#footnote-ref-1)
2. Information on exclusions to this rule is available from the Advice Centre at each Campus [↑](#footnote-ref-2)