# SIT315 – Programming Paradigms

TaskM2.T1P: Parallel Matrix Multiplication

### Greg McIntyre

### 218356779

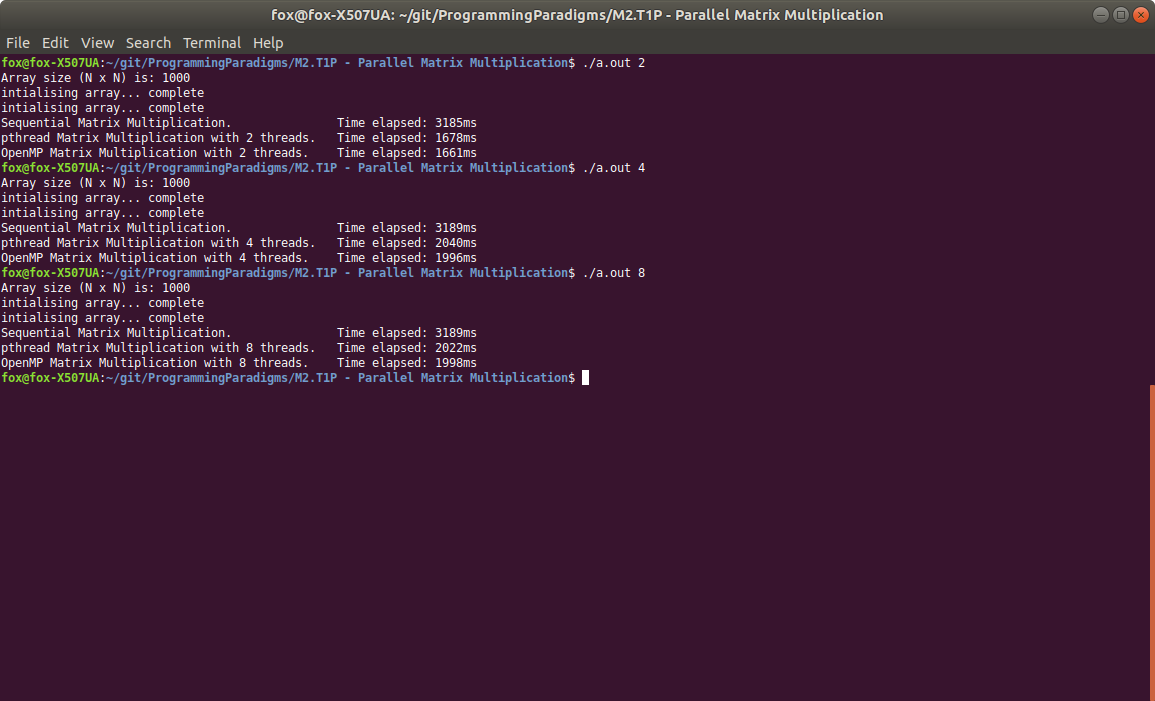
### 1. Implement a sequential matrix multiplication program in C or C++.

Written in C++.

<https://github.com/gregorymcintyre/ProgrammingParadigms/tree/master/M2.T1P%20-%20Parallel%20Matrix%20Multiplication>

or See Appendix B for a static version

### 2. At the end of the program, please print the execution time.



### 3. Once you have completed and tested the program, please review your code and develop a roadmap to parallelise your code.

To parallelise my code, I would like to have the following loop happen in parallel:

void SequentialMatrixMultiplication()

{

int value;

for (int i = 0; i < N; i++)

{

//code

I will implement it to perform the column functions independently, as results are not dependent on each other I should not need to implement mutex with this method, but I will assess as the program develops.

The array values are independent of each other and should be able to be implemented parallel. This would mean that all array values would be calculated concurrently, this should improve the performance of the program significantly.

### 4. Implement your parallel algorithm in C or C++ using pthread library

Same Git

<https://github.com/gregorymcintyre/ProgrammingParadigms/tree/master/M2.T1P%20-%20Parallel%20Matrix%20Multiplication>

### 5. Evaluate the performance of your program

Program performed well, I expected to see a more variable result when adding more threads, but results show that the systems did not improve significantly in runtime when using more than 2 threads.

### 6. Modify your sequential program to use OpenMP to achieve parallelism

Same Git

<https://github.com/gregorymcintyre/ProgrammingParadigms/tree/master/M2.T1P%20-%20Parallel%20Matrix%20Multiplication>

### 7. Evaluate the performance of the OpenMP implementation vs pthread implementation vs the sequential program

The OpenMP Did not yield a decrease in runtime, it was much simpler to implement. But implementing a #pragma omp for yielded the no improvement on the pthread method.

### 8. Submit your task as detailed on the submission details section above to OnTrack

Submitted

## Appendix A: Raw Data

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **500** | **1000** | **1500** | **2000** |
| **SequentialMatrixMultiplication** | 334 | 3378 | 16949 | 36512 |
| **PthreadMatrixMultiplication (1)** | 341 | 3613 | 17104 | 37353 |
| **PthreadMatrixMultiplication (2)** | 172 | 1808 | 8584 | 19338 |
| **PthreadMatrixMultiplication (4)** | 173 | 2106 | 7652 | 20028 |
| **PthreadMatrixMultiplication (8)** | 172 | 2023 | 8087 | 20910 |
| **OpenMPMatrixMultiplication (1)** | 331 | 3544 | 17654 | 36649 |
| **OpenMPMatrixMultiplication (2)** | 193 | 1757 | 8385 | 20226 |
| **OpenMPMatrixMultiplication (4)** | 162 | 2110 | 7893 | 20362 |
| **OpenMPMatrixMultiplication (8)** | 169 | 2027 | 8000 | 21932 |

## Appendix B: Source

/\* SequentialMatrixMultiplication.cpp

\* Greg McIntyre

\* 8/4/19

\*

\* This program creates 2 random arrays of n size and multiplies them together in a sequential, pthread and OpenMP method.

\*/

#include "pch.h"

#include <stdio.h>

#include <iostream>

#include <sys/time.h>

#include <time.h>

#include <pthread.h>

#include <omp.h>

using namespace std;

#define N 2000

//#define NUM\_THREADS 2

//int N;

int NUM\_THREADS;

pthread\_mutex\_t mutx;

int inputArray1[N][N];

int inputArray2[N][N];

int outputArray[N][N];

void intialiseArray(int array[N][N]) {

cout<<"intialising array... ";

for (int i = 0; i < N; i++)

{

for (int j = 0; j < N; j++)

{

array[i][j] = rand() % ((100 - 1) + 1) + 1;

}

}

cout<<"complete"<<endl;

} //intialises array with random values, uses the N global variable

void printArrays(int array[N][N]){

cout <<"[";

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

cout << "[";

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

cout << array[i][j];

std::cout << " ";

}

std::cout << "]\n";

}

std::cout << "]\n\n";

} //prints array to console

void SequentialMatrixMultiplication()

{

int value;

for (int i = 0; i < N; i++)

{

for (int j = 0; j < N; j++)

{

value = 0;

for (int k = 0; k < N; k++)

{

value += inputArray1[i][k] \* inputArray2[k][j];

}

outputArray[i][j] = value;

}

}

} //performs a sequential matrix mutiplication

void \*pthreadMatrixMultiplication(void \*threadid)

{

long tid = (long)threadid;

long value;

int range = N/NUM\_THREADS;

int start = tid \* range;

int end = start + range;

//pthread\_mutex\_lock(&mutx);

//cout<<tid<<":"<<start<<"-"<<end<<endl;

//pthread\_mutex\_unlock(&mutx);

for (int i = start ; i < end ; i++)

{

for (int j = 0; j < N; j++)

{

value = 0;

for (int k = 0; k < N; k++)

{

value += inputArray1[i][k] \* inputArray2[k][j];

}

//pthread\_mutex\_lock(&mutx);

outputArray[i][j] = value;

//pthread\_mutex\_unlock(&mutx);

}

}

//cout<<"Done"<<endl;

pthread\_exit(NULL);

} //performs a threaded matrix multiplication using the global NUM\_THREADS value

void OpenmpMatrixMultiplication()

{

#pragma omp parallel

{

//cout<<omp\_get\_thread\_num()<<endl;

int value;

#pragma omp for

for (int i = 0; i < N; i++)

{

for (int j = 0; j < N; j++)

{

value = 0;

for (int k = 0; k < N; k++)

{

value += inputArray1[i][k] \* inputArray2[k][j];

}

outputArray[i][j] = value;

}

}

}

} //performs a threaded matrix multiplication using OpenMP

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

NUM\_THREADS = atoi(argv[1]); //pull argv value for threads

//N = atoi(argv[2]);

struct timeval timecheck;

pthread\_t threads[NUM\_THREADS];

pthread\_mutex\_init(&mutx, NULL);

omp\_set\_num\_threads(NUM\_THREADS);

cout<<"Array size (N x N) is: "<<N<<endl;

intialiseArray(inputArray1);

intialiseArray(inputArray2);

//cout << "Input Array"<<endl;

//printArrays(inputArray1);

//cout << "Input Array"<<endl;

//printArrays(inputArray2);

//cout << "Output Array"<<endl;

//printArrays(outputArray);

cout<<"Sequential Matrix Multiplication.\t\tTime elapsed: ";

gettimeofday(&timecheck, NULL);

long timeofday\_start = (long)timecheck.tv\_sec \* 1000 + (long)timecheck.tv\_usec /1000;

SequentialMatrixMultiplication();

gettimeofday(&timecheck, NULL);

long timeofday\_end = (long)timecheck.tv\_sec \* 1000 + (long)timecheck.tv\_usec /1000;

double time\_elapsed = timeofday\_end - timeofday\_start;

cout<<time\_elapsed<<"ms"<<endl;

//cout << "Output Array"<<endl;

//printArrays(outputArray);

cout<<"pthread Matrix Multiplication with " << NUM\_THREADS << " threads.\tTime elapsed: ";

gettimeofday(&timecheck, NULL);

timeofday\_start = (long)timecheck.tv\_sec \* 1000 + (long)timecheck.tv\_usec /1000;

for (long tid = 0 ; tid < NUM\_THREADS;tid++){

pthread\_create(&threads[tid], NULL, pthreadMatrixMultiplication, (void \*)tid);

}

for (long tid = 0 ; tid < NUM\_THREADS;tid++){

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

}

gettimeofday(&timecheck, NULL);

timeofday\_end = (long)timecheck.tv\_sec \* 1000 + (long)timecheck.tv\_usec /1000;

time\_elapsed = timeofday\_end - timeofday\_start;

cout<<time\_elapsed<<"ms"<<endl;

//cout << "Output Array"<<endl;

//printArrays(outputArray);

cout<<"OpenMP Matrix Multiplication with " << NUM\_THREADS << " threads.\tTime elapsed: ";

gettimeofday(&timecheck, NULL);

timeofday\_start = (long)timecheck.tv\_sec \* 1000 + (long)timecheck.tv\_usec /1000;

OpenmpMatrixMultiplication();

gettimeofday(&timecheck, NULL);

timeofday\_end = (long)timecheck.tv\_sec \* 1000 + (long)timecheck.tv\_usec /1000;

time\_elapsed = timeofday\_end - timeofday\_start;

cout<<time\_elapsed<<"ms"<<endl;

//cout << "Output Array"<<endl;

//printArrays(outputArray);

return 0;

}