**Declaration on Plagiarism**

This form must be filled in and completed by the student(s) submitting an assignment

|  |  |
| --- | --- |
| Name: | Rebekah Jennifer Manimaran |
| Student Number: | 19210518 |
| Programme: | Msc Computing – Data Analytics |
| Module Code: | CA670 |
| Assignment title: | Concurrent Programming |
| Submission Date: | 19/04/2020 |
| Module Coordinator | Dr. David Sinclair |

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. I have read and understood the Assignment Regulations. I have identified and included the source of all facts, ideas, opinions, and viewpoints of others in the assignment references. Direct quotations from books, journal articles, internet sources, module text, or any other source whatsoever are acknowledged and the source cited are identified in the assignment references. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.

I have read and understood the referencing guidelines found at <http://www.dcu.ie/info/regulations/plagiarism.shtml>, https://www4.dcu.ie/students/az/plagiarism and /or recommended in the assignment guidelines

Name: Rebekah Jennifer Date: 19-04-2020

**Efficient Large Matrix Multiplication in OpenMP**

In this OpenMP assignment, an efficient large matrix multiplication algorithm is developed using naïve implementation and parallelism.

In naïve implementation, serial computing is followed where a problem is broken into streams of instructions that are executed sequentially one after the other on a single processor. But, in a parallel computing program a problem is broken into parts that can be solved concurrently. Each part is further broken into streams of instructions where they execute on different processors.

In this assignment, shared memory architecture like OpenMP is used where a memory unit that contains global address space is shared with multiple processors that perform different tasks.

OpenMP (Open Multi-Processing) is a technology where a sequential program can be converted into a shared memory multithreaded program. I have used the GCC compiler in order to run my OpenMP program. I have used the possible constructs and functions to execute them in a parallel manner.

Matrix multiplication is one of the most common and most computationally intensive operations. The size of the problem increases as the dimensions of the matrix increase. Meaning, **a 50x50 matrix is computationally much more intensive compared to a 10x10 matrix.**

One method to reduce the computationally intensive task is by implementing a parallelism concept into the operations. In my program, I have parallelized the most computationally intensive loop using OpenMP. It was seen that out of the two for loops in the program, the for loop that performs the matrix multiplication was highly intensive.

The sequential code and parallelised code are computed and attached to the assignment files.

**Description and design of the Program**

First, in my program I have declared the necessary header files and used the function declaration for Sequential\_Multiplication and Parallel\_Multiplication. The main function gets the input of the dimension size of the matrix (DIM) from the user and performs dynamic allocation of memory space (**malloc**) for the input and output matrices.

Since the size of double is 8 bytes the statement **A = (double \*\*)malloc(DIM\*sizeof(double \*))** will allocated 8000 bytes of memory (if DIM = 1000). If space is insufficient the allocation in the memory fails and returns a null pointer. Later, the function definition for sequential program and parallel program is called in the main function with respective arguments of the matrices. **free()** is used to reduce the wastage of memory by deallocating it.

**Naïve Matrix Multiplication:**

Naïve matrix multiplication is nothing but normal serial flow of multiplying two matrices. In my program, I have used dimension size DIM value = **1000** (1000\*1000 matrix), where execution time was **9.624076 seconds**. For a large numerically intensive problem like matrix multiplication which executes the instructions one at a time in a single processor. This impacts the memory management and processor speed for complex inter-related real-world problems. Thus the real world runs on parallel simulation which saves time, because of multiple threads and memory associated with it.

**Parallel Program for Matrix Multiplication:**

Parallel multiplication is where the number of threads to execute the program is assigned with the OpenMP libraries as **omp\_set\_num\_threads (5)**.

**omp\_get\_wtime ()** – This is a function used to optimize the execution time which returns the elapsed wall clock time in seconds.

**#pragma omp parallel private(i,j,k,t\_id, tmp) shared(A,B,C,t\_num)** - Compiler directive is used whenever I want to execute a block in parallel with multiple threads. Example: if there are 4 threads, there are 4 codes of multiplication executing in a parallel manner called a fork-join model. A private variable is local to each thread and shared variables have a single copy among all threads.

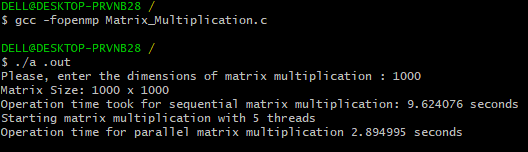
**#pragma omp for schedule (static, chunk)** – Schedule clause is used with the parallel region of matrix multiplication that shows how the iterations are divided among the thread in the loop. In my program, I have used static to split the computation task based on chunk size. Chunk size is calculated by matrix size and the number of threads. This divides the iterations statically at run time.

The program has been implemented without any race conditions where the correctness of the program is checked based on the order in which the threads are scheduled.

Also, to reduce excessive overheads and have better performance print statements aren’t used within the group.

**Execution time:**

The execution time of the parallel process is **2.894995 seconds** with 5 threads and 1000 dimension.

****

The efficiency of the program is improved by 70% using OpenMP for a matrix size of 1000 with the number of threads as 5.

The code was run on a laptop with 8GB RAM and 1TB Memory with Over clockable 2.20GHz resulted in a decrease of execution time as the number of threads increase.

The below chart shows the **efficiency of execution time as a function of the number of threads**.

As the size of the thread increases the computation time to calculate the matrix multiplication task in decreased.

**Speed up and efficiency of large values of Dimensions**

This chart shows the speed up on time and efficiency by comparing it with naïve matrix multiplication using OpenMP multithreaded code.

As the dimensions, the matrix increase the efficiency moves up in a gradual manner and time allocated to execute program increases fastly.

We should note that parallelizing the code does not decrease the execution time. There might be chances that the execution time will increase when there is a communication overhead due to threading. The threads will have to communicate with each other and this will increase the execution time.

Also, by changing the number of threads, the results change since we don’t know which thread is running at which point in time. Also, the execution time changes when the number of threads is increased or decreased. This is because the work is split between the threads and they are all working on problem sizes of N/p where N is the size of the problem.

References:

[1] “Matrix multiplication” [Online], https://en.wikipedia.org/wiki/Matrix\_multiplication, Accessed on : April 5, 2020

[2] “Dynamic Memory Allocation in C using malloc(), calloc(), free() and realloc()”[Online] ,https://www.geeksforgeeks.org/dynamic-memory-allocat ion-in-c-using-malloc-calloc-free-and-realloc/,Accessed on : April 8, 2020

# [3]“MXM\_OPENMP Matrix Multiplication with OpenMP “<https://people.sc.fsu.edu/~jburkardt/c_src/mxm_openmp/mxm_openmp.html>

# [4] Parallel Matrix Multiplication [C][Parallel Processing] [https://medium.com/tech vision/parallel-matrix-multiplication-c-parallel-processing-5e3aadb36f27](https://medium.com/tech%20vision/parallel-matrix-multiplication-c-parallel-processing-5e3aadb36f27)