| | |
|---|---|
| **Name:** | Rohit Toshniwal |
| **Student Number:** | ******* |
| **Programme:** | MCM1 |
| **Module Code:** | CA670 |
| **Assignment Title:** | Concurrent Programming |
| **Submission Date:** | 19 April 2020 |
| **Module Coordinator:** | Dr. David Sinclair |

# Assignment 2

# Efficient Large Matrix Multiplication in Open MP

## 1. Purpose:

We have to develop an **efficient** large matrix multiplication algorithm in Open MP. A prime criterion in the assessment of this assignment will be the efficiency of the implementation and the evidence present to substantiate the claim that the implementations are efficient.

## 2. Description of the design of the program

### A. Preliminaries

For this assignment, I have used Cygwin and GCC complier version 9.3.0 which comes with Open MP libraries.

- Cygwin is a large collection of GNU and Open Source tools which provide functionality similar to a Linux distribution on Windows.
- GCC is GNU Compiler Collection includes front ends for C, C++, Objective-C, Fortran, Ada, Go, and D, as well as libraries for these languages.

### B. System Specifications used for this program

## Device specifications

### IdeaPad Z510

| | |
|---|---|
| Device name | RST |
| Processor | Intel(R) Core(TM) i7-4702MQ CPU @ 2.20GHz 2.20 GHz |
| Installed RAM | 4.00 GB |
| Device ID | 6660EFFD-C232-4800-B8F9-816933C8E8BB |
| Product ID | 00327-60000-00000-AA420 |
| System type | 64-bit operating system, x64-based processor |

## C. Description of the code

***Methods Used in the program***

Matrix multiplication is the root calculation done in many algorithms' implementation like Fibonacci Series, it also plays an important role in quantum mechanics. Various efficient method of matrix multiplication exists. One of the approaches is implanted using the open MP as according to the requirement of the assignment.
For analysing the difference between **serial** and **parallel** matrix multiplication, we have implemented both type of matrix multiplication. Also, this program includes both **static** and **dynamic schedulers** for parallel matrix multiplication. This assignment is performed with various trial on the same machine and compares the different size of the matrix multiplication. In this assignment we have used the same N*N matrix to be multiplied with same N*N matrix.
According to above explanation we can say that two methods are used in the implementation of matrix multiplication:

- Serial matrix multiplication
- Static scheduling open MP
- Dynamic scheduling open MP
- Open MP – Open Multiprocessing

***Overview of the program***

-This code takes size of matrix as the input from user and defined a max size for the matrix which is 5000 to allocate the size of matrix which is enough to be called as a large matrix.

-For capturing the time of the execution and to get the final output, we have used **omp_get_wtime()** function to capture the start and end time of execution.

-We haven't used block method to implement the matrix multiplication instead of that we have used dynamic and static scheduler which gave me an efficient result than the block method.

- The **schedule(static, chunk-size)** clause of the loop construct specifies that for loop has the static scheduling type. Open MP divides the iterations into chunks of size chunk-size and it distributes the chunks to threads in a circular order.

When no chunk-size is specified, Open MP divides iterations into chunks that are approximately equal in size and it distributes at most one chunk to each thread.

So, one of the advantages for using **static scheduling** is that it improves locality in memory access.

- The **schedule(dynamic, chunk-size)** clause of the loop construct specifies that for loop has the dynamic scheduling type. Open MP divides the iterations into chunks of size chunk-size. Each thread executes a chunk of iterations and then requests another chunk until there are no more chunks available.

There is no particular order in which the chunks are distributed to the threads. The order changes each time when we execute for loop.

If we do not specify chunk-size, it defaults to one.

This code is used with open MP for faster execution, there are various parameters used in the code which are explained below.

- **n** - The size of Matrix used for matrix multiplication which we are taking as an input from end user
- **Matrix A, B, C, D -** A and B are the input matrix C is the output matrix for serial matrix multiplication, D is the output matrix for parallel matrix multiplication.

```
void matrix_mult_serial(int n)
{
    int i,j,k;
    double st=omp_get_wtime();
    for(i=0;i<n;i++)
    {
        for(j=0;j<n;j++)
        {
            for(k=0;k<n;k++)
            {
                c[i][j]+=a[i][k]*b[k][j];
            }
        }
    }
    double en=omp_get_wtime();
    printf("Total time taken by Serial multiplication: %lf\n",en-st);
}
```

**Serial implementation of the program**

```
void matrix_mult_parallel1(int n)
{
    //Static Scheduler
    memset(d,0,sizeof d);
    int i,j,k;
    double st=omp_get_wtime();
    #pragma omp parallel for schedule(static,50) collapse(2) private(i,j,k)shared(a,b,c)
    for(i=0;i<n;i++)
        for( j=0;j<n;j++)
            for(k=0;k<n;k++)
                d[i][j]+=a[i][k]*b[k][j];
    double en=omp_get_wtime();
    printf("Total time taken by Parallel-1(Static Scheduler) %lf\n",en-st);
    check(n);
}
```

**Parallel implementation of program using open MP static scheduler**

```
void matrix_mult_parallel2(int n)
{
    //Dynamic Scheduler
    memset(d,0,sizeof d);
    int i,j,k;
    double st=omp_get_wtime();
    #pragma omp parallel for schedule(dynamic,50) collapse(2) private(i,j,k) shared(a,b,c)
    for(i=0;i<n;i++)for( j=0;j<n;j++)for(k=0;k<n;k++)
        d[i][j]+=a[i][k]*b[k][j];
    double en=omp_get_wtime();
    printf("Total time taken by Parallel-2(Dynamic Scheduler) %lf\n",en-st);
    check(n);
}
```

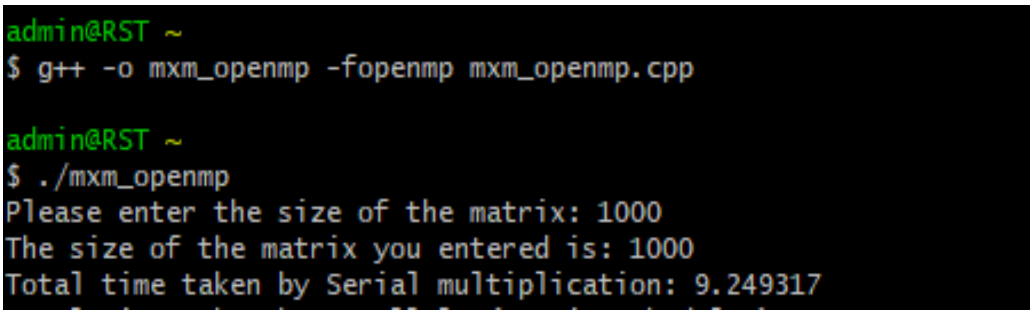
**Parallel implementation of program using open MP dynamic scheduler**

***Open MP variables:***

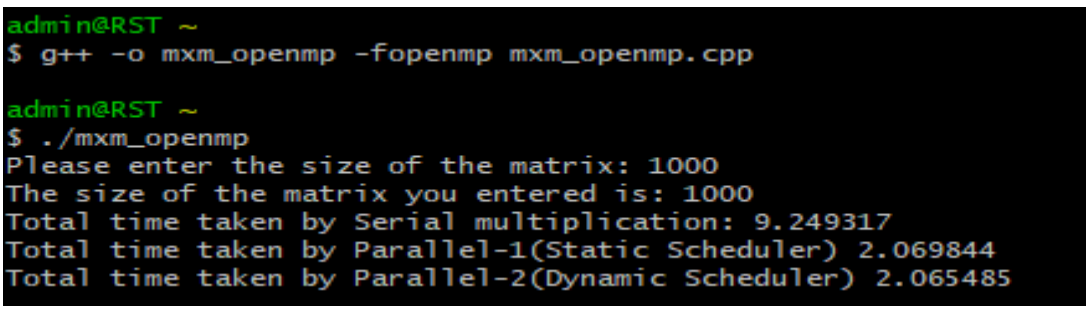
- **Shared** – this variable is used to share the values of variables inside and outside the parallel region, for e.g. here matrix A, B and C are the shared variable. First three are the matrix which are initialized in the code and we need to pass it inside.
- **Static** - The loop iteration is divided into pieces of chunk, and then assigned to threads. If at all the value of chunk is not given, the processor evenly distributes iteration among threads.
- **Dynamic** - There is no particular order in which the chunks are distributed to the threads. The order changes each time when we execute for loop. If we do not specify chunk-size, it defaults to one.
- **Collapse(n)** - Specifies how many loops in a nested loop should be collapsed into one large iteration space and divided according to the schedule clause. The sequential execution of the iterations in all associated loops determines the order of the iterations in the collapsed iteration space.
- **#Pragma** - These directives helps us to specify the functions that are needed to run before program startup (before the control passes to main()) and just before program exit (just before the control returns from main()).

## 3. Efficiency of the program

Efficiency of code is checked by running it multiple times for different matrix sizes.
The size for which the matrix is run is for 250,500,1000,2000.
Below are the snapshots for 1000 size of matrix multiplication output:




**Output of the serial matrix multiplication**




**Output of parallel matrix multiplication**

As we can see from the above results, the time taken by serial matrix multiplication is way more than the parallel. Also, the difference between static parallel and dynamic parallel is negligible but it is very efficient. It took only 2 seconds to calculate the matrix multiplication of size 1000.

| Matrix Size | Serial Matrix Multiplication Execution Time | Parallel Matrix Multiplication Execution Time | |
|---|---|---|---|
| | | Static | Dynamic |
| 250*250 | 0.033777 | 0.017831 | 0.009675 |
| 500*500 | 0.573943 | 0.225916 | 0.1252 |
| 1000*1000 | 9.249317 | 2.065485 | 2.10654 |
| 2000*2000 | 64.743712 | 16.044211 | 15.824889 |

**Different matrix sizes multiplication results**

The experimental results shown above use 2 physical cores with hyper threading, which makes a total of 4 threads, with 2 threads per core. These results give us a good reason to use parallelism as we will get better execution times, increase our productivity, reduce costs and reduce the “time-to-market”.

## 4. References


1. https://opensourceforu.com/2013/01/openmp-schedule-clause-parallel-matrix-multiplication/
2. https://objectcomputing.com/resources/publications/mnb/openmp
3. https://preshing.com/20141108/how-to-install-the-latest-gcc-on-windows/
4. https://www.openmp.org/
5. https://codereview.stackexchange.com/questions/165929/matrix-multiplication-with-openmp-parallel-for-loop
6. http://jakascorner.com/blog/2016/06/omp-for-scheduling.html