Matrix Multiplication

Course Level:

CS2

PDC Concepts Covered:

PDC Concept	Bloom Level
Concurrency	C
Data Parallel	A
Cache Locality	A

Programming Knowledge Prerequisites:

Basic programming knowledge in C or Java is required for this lab. More specifically, the skills below are needed to complete this lab.

- Variable declaration
- If-else
- For loop, while loop
- Functions/methods

Tools Required:

Java: Java Development Kit (JDK) 8 or later

Prolog and Review:

One of the important concepts in programming is the **if-else** statement. Using the if-else statement, programmer can decide which actions to take in the program. Following is the if-else syntax.

```
if(boolean_expression) {
   /* statement(s) will execute if the boolean expression is true */
} else {
   /* statement(s) will execute if the boolean expression is false */
}
```

If the Boolean expression evaluates to true, then the **if block** is executed, otherwise **else block** of code is executed. Likewise, another important programming concept is looping. Using looping, a programmer can do a set of statements multiple times. A common looping structure in programming languages is the **for-loop**. Consider the following example.

```
for ( initialization; condition; increment ) {
   statement(s);
```

```
}
```

The **initialization** is executed once upon entering the loop. Next, the **condition** is evaluated. If it is true then the body of the loop is executed, otherwise loop exits, and the program continues with the next statement that occurs after the for-loop. After the body executes, control flow jumps to **increment** statement. Typically, the increment statement updates a loop control variable that is used in the condition to determine is the loop should continue. The condition gets evaluated again and the process repeats. Like **for** loop, another common loop structure is **while** loop. In a **while** loop, statements are repeatedly executed as long as a given condition is true. The syntax is given below.

```
while(condition) {
   statement(s);
}
```

For this lab you will also have some idea about cache locality. In computer systems, programs tend to reuse data and instructions near those they have used recently, or exactly same data and instructions. There are two basic types of locality: Temporal and Spatial. In Temporal locality, recently referenced items are likely to be referenced in the near future. In Spatial locality, the data items with nearby addresses tend to be referenced in near future. Strong cache locality gives great performance to a system as it reduces cache misses.

Problem description:

In this lab, you will be writing programs to multiply two matrices. Two matrices can be square or any size. As you will be using personal computer, restrict the dimension of the matrices to below 2000 by 2000 (square matrices). Consider the below diagram, where matrix ***A*** and ***B*** have dimensions (M x K) and (K x N) respectively. Therefore, the resultant matrix ***C*** has dimension (M x N).

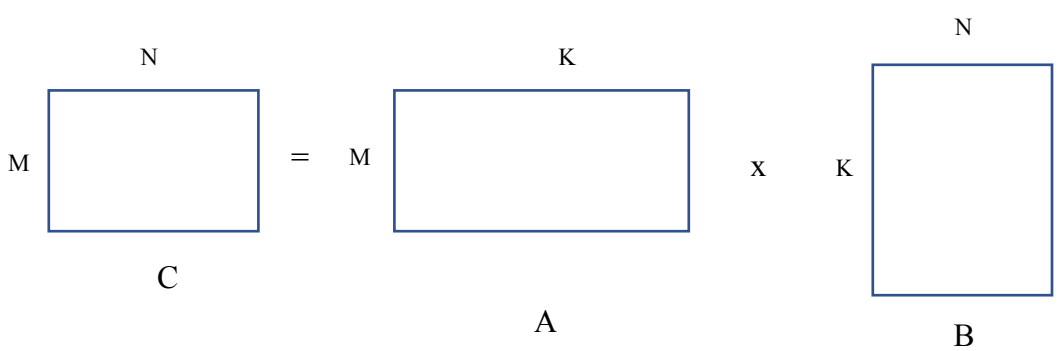

Fig 1: Matrix ***A*** and matrix ***B*** multiply to generate matrix ***C***.

You will write three functions to implement serial, parallel and cache efficient ways of multiplying two matrices.

a) Serial method: Serial method is pretty much the way you did in high school. It's also known as naïve method, where rows of ***A*** matrix multiplies (and adds) column of ***B*** matrix and generates a single cell of first row of ***C*** matrix.
b) Cache efficient method: In this method, cache misses are reduced by multiplying a cell of ***A*** matrix with a row of ***B*** matrix at a time. After each iteration, partial update of a row of ***C*** matrix is generated.
c) Parallel method: In this method, you will divide the number of rows of ***A*** matrix with the number of processors. Each different part of the ***A*** matrix is then assigned to processors. Each processor multiplies their part of ***A*** matrix with the whole ***B*** matrix to generate part of ***C*** matrix. Matrix ***B*** is not partitioned to keep the program as simple as possible.

Methodology:

The serial and the cache efficient multiplications are pretty much straight forward as no partitioning required to the matrices. To implement the parallel version you will partition row-wise to the ***A*** matrix. Each part of the ***A*** matrix will be multiplied with the whole ***B*** matrix. If the number of rows is not evenly divided by the number of processors, the last processor gets some extra rows of ***A*** matrix.

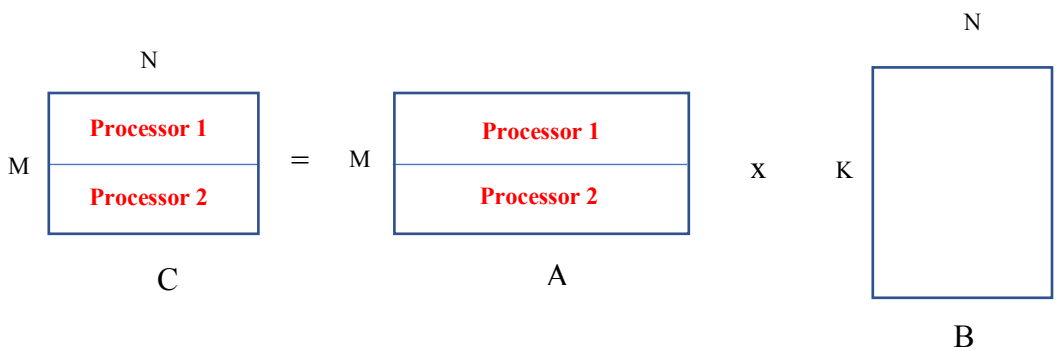

Fig 2: Data Partitioning of A matrix among the two processors.

Figure 2 shows how matrix ***A*** is partitioned into two equal halves and assigned to two different processors. Now both processors concurrently perform multiplication of ***A*** (half) and ***B*** matrices.

***Note to the instructor:** Instructor can illustrate **Data parallel** by pointing to that different processors are doing the same computation but on different pieces of data.

Implementation:

For this lab, you will be implementing three versions of matrix multiplications: serial, cache efficient and parallel. To create matrices two dimensional arrays can be used. Fortunately, you can use one

dimensional array to make your life easy for this lab. Consider three matrices with their dimension below.

A: M x K (M and K are number of rows and columns of ***A*** matrix)
B: K x N (K and N are number of rows and columns of ***B*** matrix)
C: M x N (M and N are number of rows and columns of ***C*** matrix)
You can declare the three matrices like below.

```
double[] A = new double[M*K];
double[] B = new double[K*N];
double[] C = new double[M*N];
```

Now, any 2D cell location say (i, j) can be located by the below expression.

```
j + (i * num_of_columns)
```

Matrices are initialized with random floating point numbers. Consider below code of random initialization to ***A*** and ***B*** matrices.

```
void initialize_matrix(double* matrix, int row, int col){
    for (int i = 0; i < row; i++ ){
        for(int j = 0; j < col; j++){
            matrix[j+ i*col] = i+j;//random number
        }
    }
}
```

Now, serial implementation can be performed using three for loops. Consider the following pseudo code.

Serial_matrix_multiplication()
Inputs:
A - Array of matrix A
B - Array of matrix B
C - Array of matrix C
M - Rows of matrix A
K - Columns of matrix A
N – Column of matrix B
Outputs:
C – Computed cells of C matrix
Begin
    Loop index from i=0 to M
        Loop index from j=0 to N
            Set my_sum to zero

```
                Loop index from k=0 to K
                    Multiply   A[k + i*K] with  B[j + k*N] and update my_sum
                Store my_sum to C[j + i*N]
End
```

In case of cache efficient version, second and third loops will need interchange. This makes processor to read row of ***B*** matrix instead of column of ***B*** matrix. It reduces cache misses as the C/C++ compiler reads array elements in row major order.

```
Cache_efficient_matrix_multiplication()
Inputs:
A - Array of matrix A
B - Array of matrix B
C - Array of matrix C
M - Rows of matrix A
K - Columns of matrix A
N – Column of matrix B
Outputs:
C – Computed cells of C matrix
Begin
        Loop index from i=0 to M
            Loop index from k=0 to K
                Set temp_var to A[k + i*K]
                Loop index from j=0 to N
                    Multiply   temp_var with  B[j + k*N] and update C[j + i*N]
End
```

In the above pseudo code, variable temp_var is set to A[k + i*K], which stays in the cache until the third loop ends. This is an example of Temporal Cache locality we discussed earlier. When the address of B[j + k*N] is referenced in third loop, the cache block reads consecutive elements in row major order from the main memory into the cache. This reduces memory read time by the processor while running in the third loop as the processor finds the data available in the cache. This is an example of Spatial cache locality.

At this point, you got idea about the serial and cache efficient implementation of matrix multiplication. Now, we will talk about the parallel implementation. As always you will use OpenMP threads for this lab. To spawn threads the below directive is used before the code block that is run by the threads.

```
//#omp parallel num_threads(thread_count) shared( A, B, C, M, N, K,
thread_count)
```

As we discussed earlier, you will partition the ***A*** matrix and assign to the threads. The matrix ***B*** will not be partitioned as each thread needs the entire ***B*** matrix. You can do the partitioning using the below code snippet.

```
int my_rank= Pyjama.omp_get_thread_num();
int num_parts = M/thread_count;
int my_first_i = num_parts * my_rank;
int my_last_i;
if (my_rank == thread_count-1) my_last_i = M;
else my_last_i = my_first_i + num_parts;
```

In the above code, M is the number of rows of ***A*** matrix. Every thread calculates its starting and ending row position of ***A*** matrix. Thus, every thread can work with a smaller block of ***A*** matrix (num_parts x K) to multiply the ***B*** matrix simultaneously.