MONASH University

FIT3143 Tutorial Week 9

PARALLEL ALGORITHM DESIGN - PARTITIONING

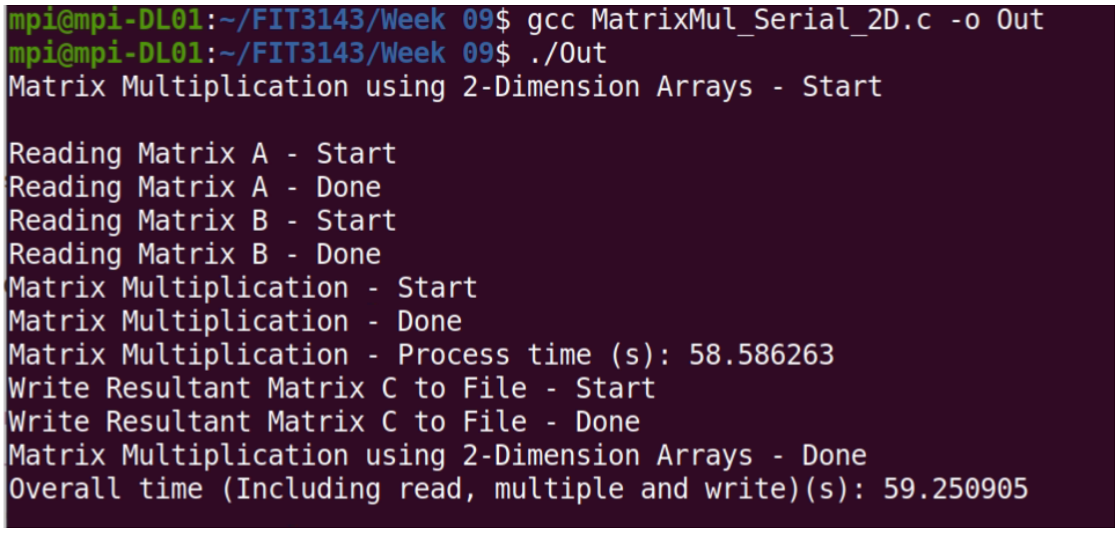
OBJECTIVES

- Optimize algorithm design for improved efficiency
- Design solutions for efficient parallel partitioning problem

Note: Tutorials are not assessed. Nevertheless, please attempt the questions to improve your unit comprehension in preparation for the labs, assignments, and final assessments.

QUESTIONS

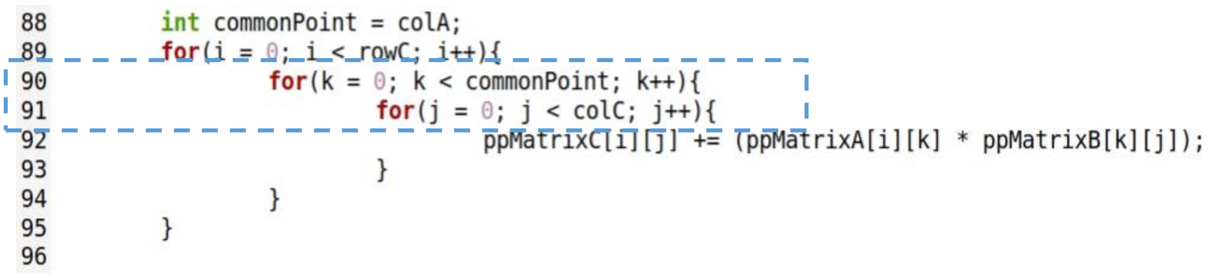
1. The following code file implements a serial matrix multiplication. Two text files (i.e., MA.txt and MB.txt) were preloaded with 2000 × 2000 matrices. Therefore, the multiplication of the matrices in MA.txt and MB.txt would also produce a 2000 × 2000 matrix C. When the code is compiled and executed on a computer with six cores running a native Linux operating system, the following outcome is observed:



```
mpi@mpi-DL01:~/FIT3143/Week 09$ gcc MatrixMul_Serial_2D.c -o Out
mpi@mpi-DL01:~/FIT3143/Week 09$ ./Out
Matrix Multiplication using 2-Dimension Arrays - Start

Reading Matrix A - Start
Reading Matrix A - Done
Reading Matrix B - Start
Reading Matrix B - Done
Matrix Multiplication - Start
Matrix Multiplication - Done
Matrix Multiplication - Process time (s): 58.586263
Write Resultant Matrix C to File - Start
Write Resultant Matrix C to File - Done
Matrix Multiplication using 2-Dimension Arrays - Done
Overall time (Including read, multiple and write)(s): 59.250905
```

Figure 1: Serial matrix multiplication performance (2000 × 2000 matrix). Note that performance figures will vary on different hardware platforms.

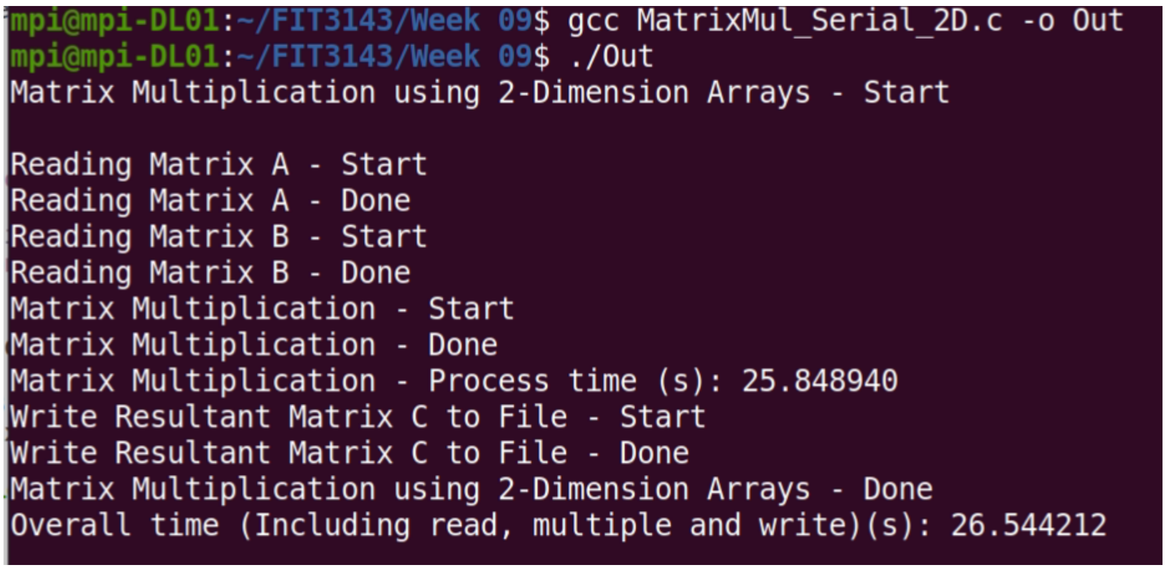
In an attempt to optimise the serial matrix multiplication, lines 90 and 91 in the provided code file were swapped based on the following screenshot:



```
88          int commonPoint = colA;
89          for(i = 0; i < rowC; i++){
90                  for(k = 0; k < commonPoint; k++){
91                          for(j = 0; j < colC; j++){
92                                  ppMatrixC[i][j] += (ppMatrixA[i][k] * ppMatrixB[k][j]);
93                          }
94                  }
95          }
96
```

Figure 2: Lines 90 and 91 were swapped. Prior to swapping the code, the *k-th for* loop represents the third nested loop. After swapping, the *j-th for* loop now represents the third nested loop.

When the revised code is compiled and executed on a computer with the same specifications as before, the following outcome is observed:



```
mpi@mpi-DL01:~/FIT3143/Week 09$ gcc MatrixMul_Serial_2D.c -o Out
mpi@mpi-DL01:~/FIT3143/Week 09$ ./Out
Matrix Multiplication using 2-Dimension Arrays - Start

Reading Matrix A - Start
Reading Matrix A - Done
Reading Matrix B - Start
Reading Matrix B - Done
Matrix Multiplication - Start
Matrix Multiplication - Done
Matrix Multiplication - Process time (s): 25.848940
Write Resultant Matrix C to File - Start
Write Resultant Matrix C to File - Done
Matrix Multiplication using 2-Dimension Arrays - Done
Overall time (Including read, multiple and write)(s): 26.544212
```

Figure 3: Serial matrix multiplication performance (2000 × 2000 matrix) based on a revised code of swapping lines 90 & 91 as seen in Figure 2.

a) By comparing Figures 1 and 3, we can observe an approximately 50% improvement in the performance of the serial matrix multiplication algorithm. Explain how the process of swapping lines 90 and 91 in the serial matrix multiplication code file resulted in such a substantial performance improvement.

b) Apart from the optimization as discussed in (a), propose another method which would improve the performance of a serial matrix multiplication algorithm.

Note: It is not compulsory to recompile the provided code here. Focus on analysing the code to identify potential optimisation techniques without having to parallelize the code.

a) In the initially provided serial matrix multiplication code (prior to swapping Lines 90 & 91), the multiplication algorithm can be expanded as follows:

$i = 0, j = 0, k = 0{:}3$

$c_{00} = (a_{00} \times b_{00}) + (a_{01} \times b_{10}) + (a_{02} \times b_{20}) + (a_{03} \times b_{30})$

$i = 0, j = 1, k = 0{:}3$

$c_{01} = (a_{00} \times b_{01}) + (a_{01} \times b_{11}) + (a_{02} \times b_{21}) + (a_{03} \times b_{31})$

$i = 0, j = 2, k = 0{:}3$

$c_{02} = (a_{00} \times b_{02}) + (a_{01} \times b_{12}) + (a_{02} \times b_{22}) + (a_{03} \times b_{32})$

$i = 0, j = 3, k = 0{:}3$

$c_{03} = (a_{00} \times b_{03}) + (a_{01} \times b_{13}) + (a_{02} \times b_{23}) + (a_{03} \times b_{33})$

After swapping Lines 90 & 91, the algorithm can be expanded as follows:

$i = 0, k = 0, j = 0{:}3$

$c_{00} = (a_{00} \times b_{00})$

$c_{01} = (a_{00} \times b_{01})$

$c_{02} = (a_{00} \times b_{02})$

$c_{03} = (a_{00} \times b_{03})$

$i = 0, k = 1, j = 0{:}3$

$c_{00} = c_{00} + (a_{01} \times b_{10})$

$c_{01} = c_{01} + (a_{01} \times b_{11})$

$c_{02} = c_{02} + (a_{01} \times b_{12})$

$c_{03} = c_{03} + (a_{01} \times b_{13})$

$i = 0, k = 2, j = 0{:}3$

$c_{00} = c_{00} + (a_{02} \times b_{20})$

$c_{01} = c_{01} + (a_{02} \times b_{21})$

$c_{02} = c_{02} + (a_{02} \times b_{22})$

$c_{03} = c_{03} + (a_{02} \times b_{23})$

$i = 0, k = 3, j = 0{:}3$

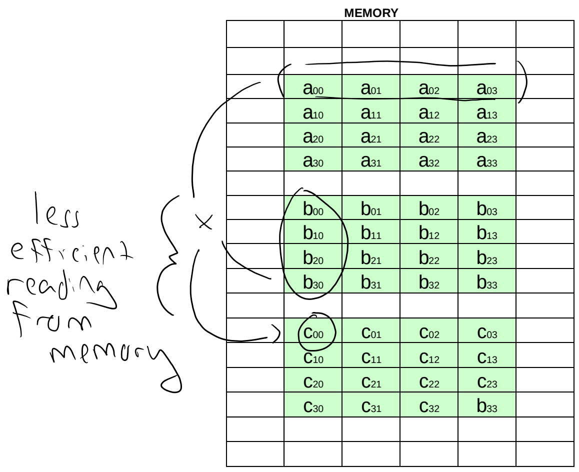
$c_{00} = c_{00} + (a_{03} \times b_{30})$

$c_{01} = c_{01} + (a_{03} \times b_{31})$

$c_{02} = c_{02} + (a_{03} \times b_{32})$

$c_{03} = c_{03} + (a_{03} \times b_{33})$

When comparing both methods, the second method allows for a much more efficient read of matrix b from memory.



Referring to the illustration above, it can be observed that in the initial approach, to compute the multiplication for c00, this would require reading b00, b10, b20 and b30 from different memory regions. Accessing memory from different regions would require consume more time. By swapping lines 90 & 91, the memory read of matrix b is much more efficient.

b) Flatten the 2D array into a 1D array

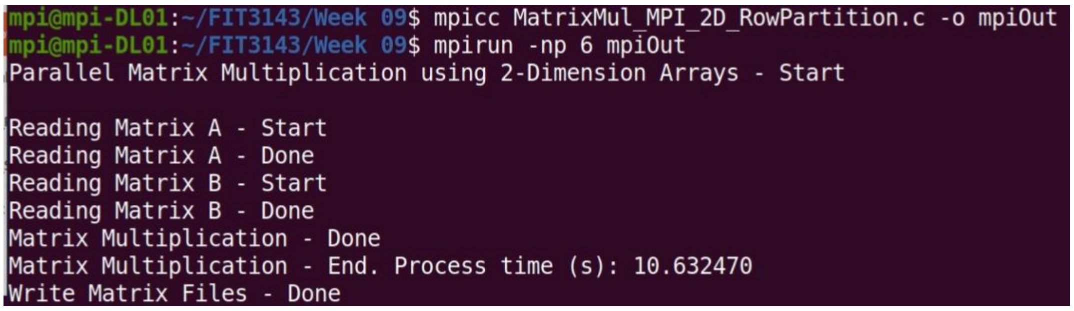
```
    int commonPoint = colA;
    for(i = 0; i < rowC; i++){
          for(j = 0; j < colC; j++){
                for(k = 0; k < commonPoint; k++){
                      pMatrixC[i * colC + j] += (pMatrixA[i * colA + k] *
pMatrixB[k * colB + j]);
                }
          }
    }
```

2. Modify the serial code in Question 1 to parallelize the matrix multiplication algorithm using **OpenMP**. Identify and implement an appropriate partitioning strategy for the matrix multiplication process.

   Note: It is not compulsory to compile and execute the parallel code. Focus your solution on applying a logically correct OpenMP code construct to the `for` loops which are used for the matrix multiplication operation.

```
    int commonPoint = colA;
    #pragma omp parallel for shared(ppMatrixA, ppMatrixB, ppMatrixC,
    commonPoint, rowC, colC) private(i, j, k) schedule(dynamic)
    for(i = 0; i < rowC; i++){
          for(j = 0; j < colC; j++){
                for(k = 0; k < commonPoint; k++){
                      ppMatrixC[i][j] += (ppMatrixA[i][k] * ppMatrixB[k][j]);
                }
          }
    }
```

3. The following code file implements a parallel matrix multiplication algorithm using MPI with row-based partitioning. Two text files (i.e., MA.txt and MB.txt) were preloaded with 2000 × 2000 matrices. Therefore, the multiplication of the matrices in MA.txt and MB.txt would also produce a 2000 × 2000 matrix. When the code is compiled and executed on a computer with six cores running a native Linux operating system, the following outcome is observed:



```
mpi@mpi-DL01:~/FIT3143/Week 09$ mpicc MatrixMul_MPI_2D_RowPartition.c -o mpiOut
mpi@mpi-DL01:~/FIT3143/Week 09$ mpirun -np 6 mpiOut
Parallel Matrix Multiplication using 2-Dimension Arrays - Start

Reading Matrix A - Start
Reading Matrix A - Done
Reading Matrix B - Start
Reading Matrix B - Done
Matrix Multiplication - Done
Matrix Multiplication - End. Process time (s): 10.632470
Write Matrix Files - Done
```

Figure 4: Parallel matrix multiplication performance using MPI with row partitioning (2000 × 2000 matrix). Note that performance figures will vary on different hardware platforms

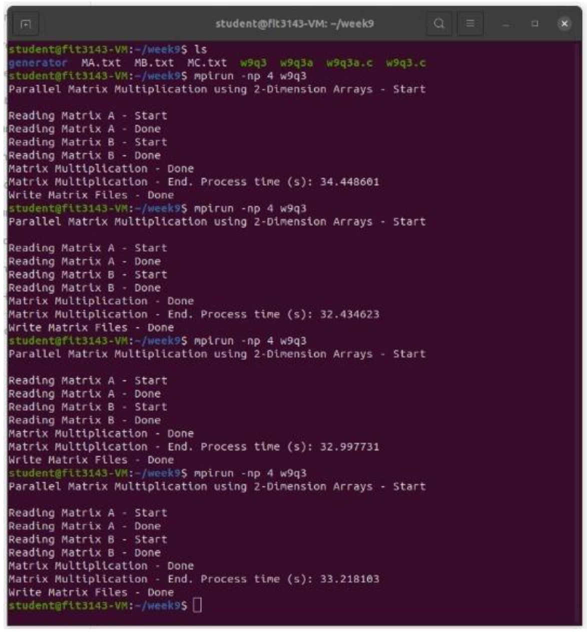
In this code, blocking send, receive and broadcast MPI functions were used to distribute matrices A and B between the MPI.

Modify the blocking communication functions in this code to that of non-blocking communication. Discuss possible outcomes in terms of its performance.

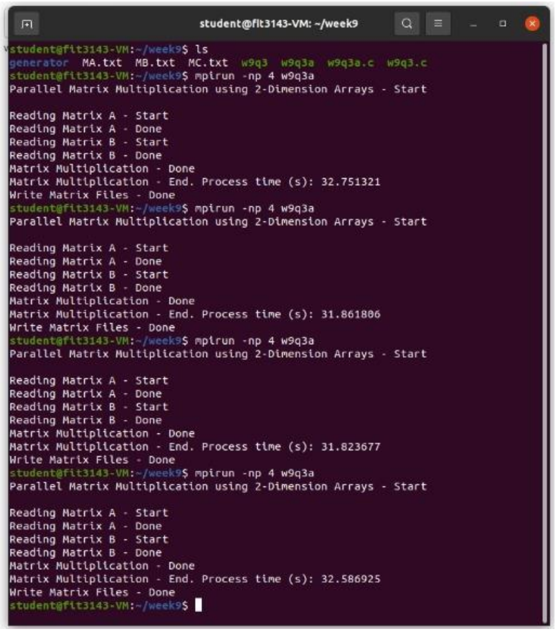
```
// 3. Root process distribute matrices A and B to other processes
      if(my_rank == 0){
            row_offset = rows_per_procs;
            for(i = 1; i < processors; i++){
                  // reset the tagSendCounter
                  tagSendCounter = 0;

                  if(i == processors - 1){
                        loopCondition = (i * rows_per_procs) + 
rows_per_procs + rows_remain;
                  }else{
                        loopCondition = (i * rows_per_procs) + 
rows_per_procs;
                  }

                  for(j=row_offset; j< loopCondition; j++){
                        // MPI_Send(ppMatrixA[j], ma_col, MPI_INT, i, 0, 
MPI_COMM_WORLD);
                        MPI_Isend(ppMatrixA[j], ma_col, MPI_INT, i, 
tagSendCounter++, MPI_COMM_WORLD, &messageReq[messageCounter++]);
                  }
                  row_offset += rows_per_procs;

            }

      }else{
            rowsToSendOrReceive = endRow - startRow;
            for(i=0; i<rowsToSendOrReceive; i++){
                  //MPI_Recv(ppMatrixA[i], ma_col, MPI_INT, 0, 
tagRecvCounter++, MPI_COMM_WORLD, &stat);
                  MPI_Irecv(ppMatrixA[i], ma_col, MPI_INT, 0, 
tagRecvCounter++, MPI_COMM_WORLD, &messageReq[messageCounter++]);
            }
```

```
        }

        // Wait all messages to be buffered
        MPI_Waitall(messageCounter,messageReq, messageStat);
```

If the code is correctly implemented, compiled and executed, you should see in improvement in the communication when distributing the rows of matrix A to the other processes, which in turn improves the overall parallel matrix multiplication time. The following figures illustrate the performance difference using blocking communication (left) and non-blocking communication (right)



```
student@fit3143-VM:~/week9$ ls
generator  MA.txt  MB.txt  MC.txt  w9q3  w9q3a  w9q3a.c  w9q3.c
student@fit3143-VM:~/week9$ mpirun -np 4 w9q3
Parallel Matrix Multiplication using 2-Dimension Arrays - Start

Reading Matrix A - Start
Reading Matrix A - Done
Reading Matrix B - Start
Reading Matrix B - Done
Matrix Multiplication - Done
Matrix Multiplication - End. Process time (s): 34.448601
Write Matrix Files - Done
student@fit3143-VM:~/week9$ mpirun -np 4 w9q3
Parallel Matrix Multiplication using 2-Dimension Arrays - Start

Reading Matrix A - Start
Reading Matrix A - Done
Reading Matrix B - Start
Reading Matrix B - Done
Matrix Multiplication - Done
Matrix Multiplication - End. Process time (s): 32.434623
Write Matrix Files - Done
student@fit3143-VM:~/week9$ mpirun -np 4 w9q3
Parallel Matrix Multiplication using 2-Dimension Arrays - Start

Reading Matrix A - Start
Reading Matrix A - Done
Reading Matrix B - Start
Reading Matrix B - Done
Matrix Multiplication - Done
Matrix Multiplication - End. Process time (s): 32.997731
Write Matrix Files - Done
student@fit3143-VM:~/week9$ mpirun -np 4 w9q3
Parallel Matrix Multiplication using 2-Dimension Arrays - Start

Reading Matrix A - Start
Reading Matrix A - Done
Reading Matrix B - Start
Reading Matrix B - Done
Matrix Multiplication - Done
Matrix Multiplication - End. Process time (s): 33.218103
Write Matrix Files - Done
student@fit3143-VM:~/week9$ 
```



```
student@fit3143-VM:~/week9$ ls
generator  MA.txt  MB.txt  MC.txt  w9q3  w9q3a  w9q3a.c  w9q3.c
student@fit3143-VM:~/week9$ mpirun -np 4 w9q3a
Parallel Matrix Multiplication using 2-Dimension Arrays - Start

Reading Matrix A - Start
Reading Matrix A - Done
Reading Matrix B - Start
Reading Matrix B - Done
Matrix Multiplication - Done
Matrix Multiplication - End. Process time (s): 32.751321
Write Matrix Files - Done
student@fit3143-VM:~/week9$ mpirun -np 4 w9q3a
Parallel Matrix Multiplication using 2-Dimension Arrays - Start

Reading Matrix A - Start
Reading Matrix A - Done
Reading Matrix B - Start
Reading Matrix B - Done
Matrix Multiplication - Done
Matrix Multiplication - End. Process time (s): 31.861806
Write Matrix Files - Done
student@fit3143-VM:~/week9$ mpirun -np 4 w9q3a
Parallel Matrix Multiplication using 2-Dimension Arrays - Start

Reading Matrix A - Start
Reading Matrix A - Done
Reading Matrix B - Start
Reading Matrix B - Done
Matrix Multiplication - Done
Matrix Multiplication - End. Process time (s): 31.823677
Write Matrix Files - Done
student@fit3143-VM:~/week9$ mpirun -np 4 w9q3a
Parallel Matrix Multiplication using 2-Dimension Arrays - Start

Reading Matrix A - Start
Reading Matrix A - Done
Reading Matrix B - Start
Reading Matrix B - Done
Matrix Multiplication - Done
Matrix Multiplication - End. Process time (s): 32.586925
Write Matrix Files - Done
student@fit3143-VM:~/week9$ 
```

4. The parallel matrix multiplication algorithm code using POSIX with submatrices applies a single vertical divider. For instance, for a square matrix multiplication (6 × 6) as illustrated in Figure 5, the odd numbered threads are positioned on right column of the submatrix and even numbered are positioned on the left column of the submatrix.

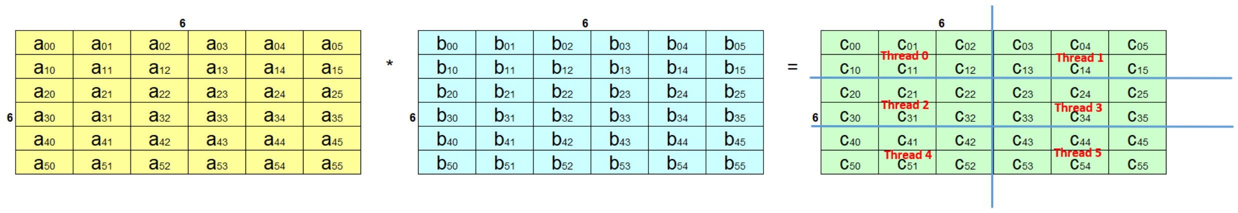


Figure 5: Parallel square matrix multiplication (6 × 6) using submatrix based with a single vertical divider.

Modify the algorithm in the thread function of the given code such that the size of the submatrix (or tiles) is based on taking the $\sqrt{p}$, where p represents the number of threads (i.e., NUM_THREADS). You can make the following assumptions:

a) The input matrices are square based matrices of $n \times n$. Refer to Figure 6 for a sample illustration of the revised method.

b) The value of NUM_THREADS is a perfect square.

c) $n \, \% \, \sqrt{NUM\_THREADS} = 0$

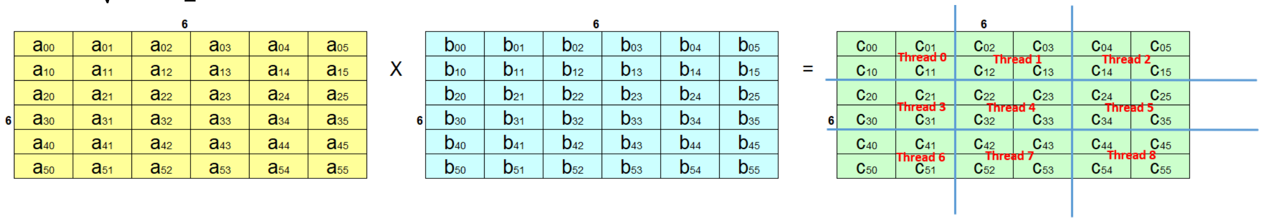


Figure 6: The size of the submatrix (or tiles) is based on $\sqrt{p}$, where p represents the number of threads. In this illustration, p = 9. Hence, $\sqrt{9}$ equals 3 and the tiles are arranged in a 3 × 3 structure. Modify the algorithm in the given code to implement this method.

```
void* MatrixMulFunc(void* pArg)
{
      int i = 0, j = 0, k = 0;
      //int row_start_point, row_end_point;
      //int col_start_point, col_end_point;
      int threadId = *((int*)pArg);

      int tileRows = (int)sqrt(NUM_THREADS);
      int tileCols = (int)sqrt(NUM_THREADS);
      int tileRowPos = threadId / tileCols;
      int tileColPos = threadId % tileCols;

      int row_start_point = tileRowPos * (rowC / tileRows);
      int row_end_point = row_start_point + (rowC / tileRows);

      int col_start_point = tileColPos * (colC / tileCols);
      int col_end_point = col_start_point + (colC / tileCols);

      int commonPoint = colA;
      for(i = row_start_point; i < row_end_point; i++)      {
            for(j = col_start_point; j < col_end_point; j++) {
                  for(k = 0; k < commonPoint; k++) {
                        ppMatrixC[i][j] += ppMatrixA[i][k] *
ppMatrixB[k][j];
                  }
            }
      }
      return 0;
}
```