MONASH University


# Parallel partitioning using Threads and MPI

## OBJECTIVES

- Design and implement a parallel partitioning method using Threads, MPI and a combination of Threads and MPI.

## MARKS

- The lab is worth 6 marks of the unit's final mark.

## INSTRUCTIONS

- Before class: Read Week 11a lecture slides and watch the lecture recording. Attempt all the tasks according to this lab specification before class. Make sure you read the marking rubric.
- During class:
    - You will be given 15 minutes to prepare your code for the demonstration period.
    - After you finish the preparation, you must submit all the files to Moodle.
    - During the demonstration, the teaching staff will ask questions about your code submission(s) and report.
    - Late submission: 5% penalty (per day).
- Marks will not be awarded if you skip the class, do not make any submissions, or make an empty submission to Moodle (unless a special consideration extension has been approved).
- You can use search engines or AI tools to search for information and resources during pre-class preparation. However, search engines and AI tools are not allowed during the code presentation period.
- If you use any AI tools to prepare your answers, **you must declare them during the interview and in your Moodle submission.**
- Always double-check all your codes & answers.

## ASSESSED TASKS

Task 1 (0%, hurdles requirement), Task 2 (60%), and Task 3 (40%)

MONASH University

## Preamble

In matrix multiplication, the product of an m × p matrix A with a p × n matrix B results in an m × n matrix denoted as C is mathematically represented as:

$$c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj} \; ; i = 1,2,3, \dots, n \; and \; j = 1,2,3, \dots, p$$

(Eq. 1)

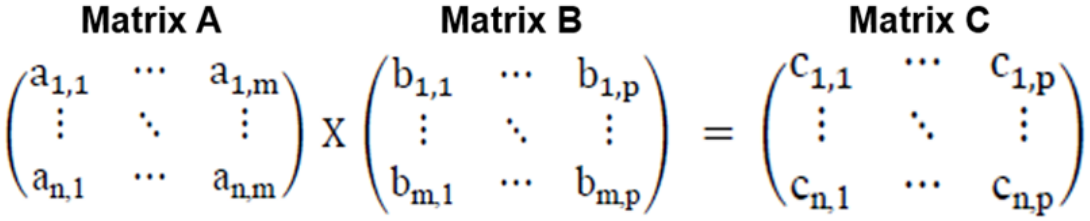



Figure 1 - Illustration of matrix multiplication. Note that Eq. 1 and this figure iterate *i* and *j* from 1. When implementing the equation into the code, you can iterate i and j from 0.

Mathematically, matrix multiplication is a binary operation that yields a resultant matrix based on two input matrices. The general definition of matrix multiplication is illustrated in Figure 1. Basically, given an n×m matrix A and an m×p matrix B, the product of the two matrices would produce an n×p matrix C. Each individual element in the resultant matrix C, denoted as C(i, j), is the dot product of the $i^{th}$ row of matrix A and the $j^{th}$ column of matrix B. This dot product operation is expressed mathematically, as in equation (1). Therefore, to define the product of matrix A and B, the number of columns in matrix A must be equivalent to the number of rows in matrix B.

Code listing 1 describes a simple code to generate matrices. When executing this compiled code, the user will be prompted to enter a file name followed by the size of the matrix. The program will then generate the required matrix and populate its cells with random values before writing the matrix into the **binary** file. To view the content of the binary file, you can load the binary file into https://hexed.it/

Code listing 1 - C code that generates matrices and writes to a binary file. You can also access the code here.

```
////////////////////////////////////////////////////////////////
//////////////////////////
// MatrixGenerator_bin.c
//
----------------------------------------------------------------
--------------------
//
// Generates a N x M  matrix with random cell values and
writes into a binary file
//
//
////////////////////////////////////////////////////////////////
```

```
//////////////////////////
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include <time.h>

#define MATRIX_CELL_MAXVAL 1000

int main()
{
     int row, col;
     int i,j;
     char matrixName[256] = {0};
     unsigned int randNum = 0;

     printf("Welcome to the random matrix file
generator!\n\n");

     printf("Specify the matrix file name (e.g. MA.bin): ");
     scanf("%s", matrixName);

     printf("Specify the matrix row and col values (e.g. 10
10): ");
     scanf("%d%d", &row, &col);

     srand((unsigned int)time(NULL));

     FILE *pFile = fopen(matrixName, "wb");
     fwrite(&row, sizeof(int), 1, pFile);
     fwrite(&col, sizeof(int), 1, pFile);

     for(i = 0; i < row; i++){
          for(j = 0; j < col; j++){
               randNum = 100 + ((unsigned int)rand() %
MATRIX_CELL_MAXVAL);
               fwrite(&randNum, sizeof(unsigned int), 1,
pFile);
          }
     }

     fclose(pFile);
     printf("Done!\n");
     return 0;
}
```

Code listing 2 describes a serial matrix multiplication algorithm. The code reads the content of the input binary files (e.g. MA.bin and MB.bin) into heap memory and computes the matrix multiplication. The resultant matrix is then written into a new binary file (e.g., MC.bin).

Code listing 2 - Serial C code for matrix multiplication. You can also access the code here.

```
////////////////////////////////////////////////////////////////////
//////////////////////
// MatrixMul_1D_bin.c
//
--------------------------------------------------------------------
--------------------
//
// Multiplies two matrices and writes the resultant
multiplication into a binary file.
//
//
////////////////////////////////////////////////////////////////////
//////////////////////
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include <time.h>

int main()
{
      // Variables
      int i = 0, j = 0, k = 0;

      /* Clock information */
      struct timespec start, end;
      double time_taken;

      clock_gettime(CLOCK_MONOTONIC, &start);

      // 1. Read Matrix A
      int rowA = 0, colA = 0;

      printf("Matrix Multiplication using 1-Dimension Arrays -
Start\n\n");

      printf("Reading Matrix A - Start\n");

      FILE *pFileA = fopen("MA_1000x1000.bin", "rb");
      fread(&rowA, sizeof(int), 1, pFileA);
      fread(&colA, sizeof(int), 1, pFileA);

      int *pMatrixA = (int*)malloc((rowA*colA) * sizeof(int));
      for(i = 0; i < rowA; i++){
            fread(&pMatrixA[i*colA], sizeof(int), colA, pFileA);
      }
      fclose(pFileA);

      printf("Reading Matrix A - Done\n");

      // 2. Read Matrix B
      int rowB = 0, colB = 0;
```

```
      printf("Reading Matrix B - Start\n");

      FILE *pFileB = fopen("MB_1000x1000.bin", "rb");
      fread(&rowB, sizeof(int), 1, pFileB);
      fread(&colB, sizeof(int), 1, pFileB);

      int *pMatrixB = (int*)malloc((rowB*colB) * sizeof(int));
      for(i = 0; i < rowB; i++){
            fread(&pMatrixB[i*colB], sizeof(int), colB, pFileB);
      }
      fclose(pFileB);

      printf("Reading Matrix B - Done\n");

      // 3. Perform matrix multiplication
      printf("Matrix Multiplication - Start\n");

      int rowC = rowA, colC = colB;
      unsigned long long *pMatrixC = (unsigned long
long*)calloc((rowC*colC), sizeof(unsigned long long));

      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]);
                  }
            }
      }

      printf("Matrix Multiplication - Done\n");

      // 4. Write resuls to a new file
      printf("Write Resultant Matrix C to File - Start\n");

      FILE *pFileC = fopen("MC.bin", "wb");
      fwrite(&rowC, sizeof(int), 1, pFileC);
      fwrite(&colC, sizeof(int), 1, pFileC);
      for(i = 0; i < rowC; i++){
            fwrite(&pMatrixC[i*colC], sizeof(unsigned long long),
colC, pFileC); // use fwrite to write one row of columns to the
file
      }
      fclose(pFileC);
      printf("Write Resultant Matrix C to File - Done\n");

      clock_gettime(CLOCK_MONOTONIC, &end);
      time_taken = (end.tv_sec - start.tv_sec) * 1e9;
      time_taken = (time_taken + (end.tv_nsec - start.tv_nsec)) *
1e-9;
      printf("Overall time (Including read, multiplication and
write)(s): %lf\n", time_taken);  // ts

      // Clean up
```

```
      free(pMatrixA);
      free(pMatrixB);
      free(pMatrixC);

      printf("Matrix Multiplication using 1-Dimension Arrays -
Done\n");
      return 0;
}
```

## Task 1 - Matrix Multiplication (Serial and Parallel with POSIX/OMP)

**Marks - 0**

**Hurdles - Yes.**

**Although this task carries no marks, you are required to complete it. If you skip this task, you will not receive any marks in this lab, even if you complete the other tasks.**

In this task, you will analyse the performance of serial matrix multiplication and implement a parallel partitioning algorithm using shared memory parallelism (i.e., POSIX or OMP).

a) Compile and execute Code Listing 1. Generate a set of binary files containing matrices. Alternatively, please refer to the sample binary files here.
b) Compile and execute Code Listing 2. Observe the time taken to complete the matrix multiplication for large matrices. You can use Table 1 as a template to populate your results. You may modify the size of the matrices in Table 1 to suit your local computing resources. You are encouraged to use the CAAS platform when testing for large matrix multiplications.
c) **Optional:** Apply Bernstein's condition to objectively verify that each cell has no dependencies for matrix multiplication.
d) Calculate the theoretical speed up. You could use Amdahl's law by assuming that the size of the problem is fixed (i.e., the size of the matrices used for the multiplication). Alternatively, you can use Gustafson's law, assuming the problem size scales with the number of processors. Hint: Refer to the lecture materials in Week 11.
e) Parallelise the serial matrix multiplication C code using data parallelism with either POSIX or OMP threads.
    i) Implement a parallel partitioning algorithm to distribute the workload among the threads equally.
    ii) You could consider a straightforward row partitioning method [or a more complex tile partitioning (or submatrix) approach].
f) Compile and execute the parallel code. Update the parallel time using POSIX/OMP in Table 1 and compute the actual speed up.
g) **Optional:** Compare and analyse the actual speed up versus the theoretical speed up. Include any observations on the differences between the theoretical and actual speed-ups. You can replicate Table 1 to include results for increasing the number of threads and CPU cores. This would provide additional analysis on the performance of matrix multiplication for increasing numbers of threads/cores and matrix sizes.
h) Write your observations above in your report.

Table 1 - Matrix multiplication compute time time for increasing matrix size (Serial & Parallel with POSIX/OMP). **Note: You may reduce the size of the matrices to suit your local computing resources**

| Number of CPU cores or logical processes: ___<br>Number of threads used for POSIX/OMP: ___ | | | | | |
|---|---|---|---|---|---|
| Matrix size | 500x500 | 1000x1000 | 2000x2000 | 3000x3000 | 4000x4000 |
| Serial time, Ts (s) | | | | | |
| Parallel time, Tp - POSIX/OMP (s) | | | | | |
| Speed Up (Ts/Tp) | | | | | |

## Task 2 - Parallel Matrix Multiplication (MPI)

**Maximum marks - 60 marks.**

**Hurdles - No.**

In this task, you will design and implement a parallel partitioning algorithm for matrix multiplication using Message Passing Interface (MPI).

a) Parallelise the serial matrix multiplication C code using data parallelism with MPI. You should implement a tile-based or submatrix approach in the parallel partitioning design. Please refer to Figure 2 for a sample depiction of this method. If you opt for a simpler row-based partitioning approach, you will not obtain the maximum marks for this task. Please refer to Week 11's lecture materials for row and tile-based partitioning methods.
b) Only the root process can access the input matrix binary files (e.g., MA_1000x1000.bin & MB_1000x1000.bin). The root process reads the content of the input binary files into its memory.
c) Implement a parallel partitioning algorithm to distribute the workload among the MPI processes equally.
    i) The root process will distribute the input matrices to the other processes in the MPI communicator. Note that you should apply an efficient approach in distributing the input matrices among the MPI processes, i.e., the root process **should only** send the required rows and columns of the input matrices to the other MPI processes. If the root process broadcasts unnecessary input data (matrices/sub-matrices) to any other MPI processes, the method will be inefficient, and marks will be deducted.
    ii) All processes (including the root process) perform the matrix multiplication based on the partitioning algorithm.
    iii) The root process gathers the computed matrices from other processes and writes the results into a **binary** file (e.g., MC.bin). Only the root process writes the results to a binary file.
d) Compile and execute the parallel code.
    i) Update the parallel time using MPI in Table 2 and compute the actual speed up.
    ii) Measure the time required to send and receive the messages between the root process and other processes and include it in Table 2 (i.e., MPI communication time).

e) Compare and analyse the actual speed up versus the theoretical speed up. Include any observations on the differences between the theoretical and actual speed-ups and the impact of communication time on the speed-up.

**Optional:** You can replicate Table 2 to include results for increasing the number of MPI processes. This would provide additional analysis on the performance of matrix multiplication for increasing numbers of processes and matrix sizes.

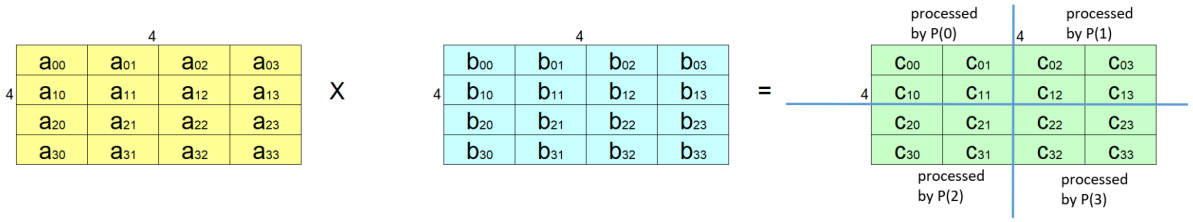
f) Write your observations above in your report.

g) Task 2 marks distribution

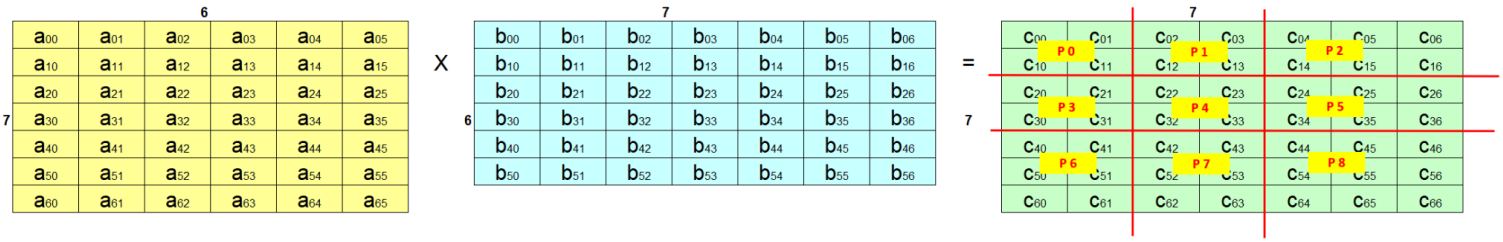
i) Parts (a) to (c): Row-based partitioning method, maximum 20 marks. Tile-based partitioning method, maximum 30 marks.

ii) Parts (d) & (e): 10 marks

iii) Q&A (interview): 20 marks




a) Simpler tile-based (or submatrix) partitioning for parallel matrix multiplication (i.e., one vertical line to split the processes into its submatrices)




b) A more complex tile-based (or submatrix) partitioning for parallel matrix multiplication (i.e., multiple vertical lines to split the processes into its submatrices)

Figure 2 - Sample tile-based (or submatrix) partitioning for parallel matrix multiplication. You may implement either (a) or (b) in Task 2.

Table 2 - Matrix multiplication compute time time for increasing matrix size (Serial & Parallel with MPI). **Note: You may reduce the size of the matrices to suit your local computing resources**

| Number of nodes (Specify 1 if only using your local computer): ___<br>Number of CPU cores or logical processes per node: ___<br>Number of MPI processes: ___ | | | | | |
|---|---|---|---|---|---|
| Matrix size | 500x500 | 1000x1000 | 2000x2000 | 3000x3000 | 4000x4000 |
| Serial time, Ts (s) | | | | | |
| Parallel time, Tp - MPI (s) | | | | | |

| | | | | | |
|---|---|---|---|---|---|
| MPI communication time (s) | | | | | |
| Speed Up (Ts/Tp) | | | | | |

## Task 3 - Parallel Matrix Multiplication (Hybrid MPI - POSIX/OMP)

**Maximum marks - 40 marks.**

**Hurdles - No.**

In this task, you will design and implement a parallel partitioning algorithm for matrix multiplication using a hybrid MPI and POSIX/OMP approach.

a) Modify the parallel code from Task 2 such that:
   i) The root process uses POSIX/OMP threads to read each input matrix **binary file** into memory. You are required to investigate/research parallel file reading with **binary files in C**.
   ii) Each MPI process will use POSIX/OMP threads to compute the matrix multiplication based on the partitioned method. In this context, you will use fewer MPI processes with one MPI process running threads to compute the parallel matrix multiplication. For instance, in task 2 above, if you have 4 CPU cores (or logical processes), you will execute *mpirun* with 4 processes. In task 3, if you have the same 4 CPU cores, you can execute *mpirun* with 2 processes, with each MPI process executing two POSIX/OMP threads to compute the parallel matrix multiplication. You can specify a fixed number of threads per MPI process. However, you should not fix the number of MPI processes in your design. Please refer to Figure 3 for a sample depiction of this method.
   iii) The root process uses POSIX/OMP threads to write the matrix multiplication result into the binary file (i.e., parallel file write).

b) Compile and execute the parallel code.
   i) Update the parallel time using Hybrid MPI-POSIX/OMP in Table 3 and compute the actual speed up.
   ii) Measure the time required to send and receive the messages between the root process and other processes and include it in Table 3 (i.e., MPI communication time).

c) Compare and analyse the actual speed up versus the theoretical speed up. Include any observations on the differences between the theoretical and actual speed-ups and the impact of communication time on the speed-up. Analyse if the communication time in Table 3 is any better than the communication time in Table 2. Provide some reasoning here.

   **Optional:** You can replicate Table 3 to include results for increasing the number of MPI processes and threads. This would provide additional analysis of the performance of matrix multiplication for increasing numbers of processes, threads and matrix sizes. You may need to use the CAAS platform when running with large numbers of MPI processes and/or threads.

d) Write your observations in (a) to (c) in your report.
e) Task 3 marks distribution

i) Part (a): 10 marks
ii) Parts (b) and (c): 10 marks
iii) Q&A (interview): 20 marks

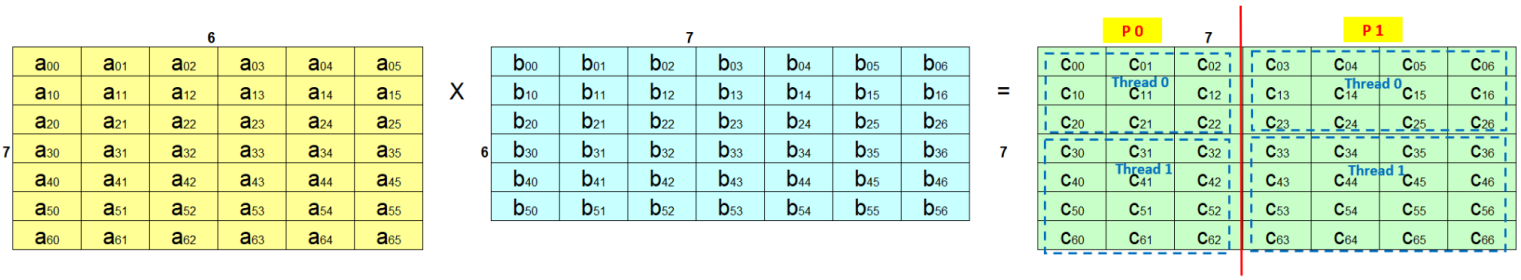



Figure 3 - Sample Hybrid MPI-Threads partitioning method. In this illustration, the workload is partitioned into two MPI processes. Each MPI process creates two threads to sub-partition the workload. To reiterate, you can specify a fixed number of threads per MPI process. However, you should not fix the number of MPI processes in your design.

Table 3 - Matrix multiplication compute time time for increasing matrix size (Serial & Hybrid MPI-POSIX/OMP). **Note: You may reduce the size of the matrices to suit your local computing resources**

| Number of nodes (Specify 1 if only using your local computer): ___<br>Number of CPU cores or logical processes per node: ___<br>Number of MPI processes: ___<br>Number of threads per MPI process: ___ | | | | | |
|---|---|---|---|---|---|
| Matrix size | 500x500 | 1000x1000 | 2000x2000 | 3000x3000 | 4000x4000 |
| Serial time, Ts (s) | | | | | |
| Parallel time, Tp - Hybrid (s) | | | | | |
| MPI communication time (s) | | | | | |
| Speed Up (Ts/Tp) | | | | | |

## What to submit

a) Task 1 (0%, Hurdles requirement)
   i) Parallel POSIX/OMP matrix multiplication code (task1.c).
   ii) A report containing Table 1.
b) Task 2 (60%, No hurdles)
   i) Parallel MPI matrix multiplication code (task2.c).
   ii) A report containing Table 2 and an analysis of the results.
   iii) A zipped file containing one sample of the input and output binary files (task2.zip).
c) Task 3 (40%, No hurdles)
   i) Hybrid MPI-POSIX/OMP matrix multiplication code (task3.c).
   ii) A report containing Table 3 and an analysis of the results.

iii) A zipped file containing one sample of the input and output binary files (task3.zip).

Note: You can have one report containing the tabulated results and analysis for Tasks 1 to 3 (Report.docx or Report.pdf).

## Important points

a) You can modify the serial C code to optimise it before parallelising it. This includes transposing matrix B before multiplying it with matrix A.
b) You can modify Tables 1 to 3 above to include additional columns or rows to aid your analysis and explanation. As mentioned above, you can also reduce the size of the input matrices to suit your local computing resources. You can compile the **`MatrixGenerator_bin.c`** file and execute it to generate smaller input matrices. Please ensure that the input matrices' correct dimensions (i.e., rows and columns) are used when performing the matrix multiplication.
c) Usage of the CAAS platform is not mandatory for this lab. Nevertheless, running the experiments on the CAAS platform would be beneficial when testing with large matrices.
d) You should use the same computer settings for serial and parallel versions of your experiments. Please refrain from running the serial code on one platform and the parallel code on another platform. It could lead to inconsistent results.
e) If you refer to generative AI to aid in this lab, please ensure that you declare it in your report.