

# Assignment 1

March 8, 2023

## 1 Prerequisite

### 1.1 Correctness Validation

Only ROOT process (master) will check the elements in each matrix (one is the brute-force result and the other is the parallel calculation result) one by one by its position. Notice that due to the precision, the comparison of float point numbers should not use  $!=$  but  $fabs(float_a, float_b) > EPS$  instead.

```

1 #define EPS 1e-6
2 #define fabs(a) (((a) > 0) ? (a) : -(a))
3 int flag = 1;
4 for (int i = 0; i < MAT_SIZE; ++i)
5 {
6     for (int j = 0; j < MAT_SIZE; ++j)
7     {
8         if (fabs(C(i, j) - bfRes[i][j]) > EPS)
9         {
10             flag = 0;
11             printf("%d %d %.20lf %.20lf\n", i, j, C(i, j), bfRes[i][j]);
12             break;
13         }
14     }
15     if (!flag)
16         break;
17 }
18 printf((flag) ? "Correct\n" : "Wrong\n");

```

### 1.2 Row-major and Column-major

If it uses 1D array to store the matrix, there are mainly 2 ways to arrange the elements: row-major and column-major [Figure 1].

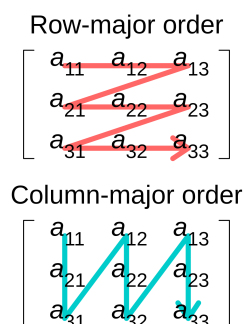


Figure 1: Row and column major order

The difference between row-major and column-major in matrix multiplication is only a transposition, i.e.,

$$C = AB \rightarrow C^T = B^T A^T$$

Which major order is used may be different in versions. How to address for each major is as below.

```

1  #if 1
2  // in row-major order
3  #define A(i, j) a[(i)*MAT_SIZE + (j)]
4  #define B(i, j) b[(i)*MAT_SIZE + (j)]
5  #define C(i, j) c[(i)*MAT_SIZE + (j)]
6  #else
7  // in column-major order
8  #define A(i, j) a[(i) + (j)*MAT_SIZE]
9  #define B(i, j) b[(i) + (j)*MAT_SIZE]
10 #define C(i, j) c[(i) + (j)*MAT_SIZE]
11 #endif

```

Since both the matrices provided in the assignment are symmetric, it will not need more operations during the construction of matrix.

### 1.3 Order of Matrix Multiplication

Usually, the order of iterating matrix elements to matrix multiplication is *ijk*. However, there are another 5 variants, which may have a better performance.

```

1  void brute_force_matmul(double mat1[MAT_SIZE][MAT_SIZE],
2                          double mat2[MAT_SIZE][MAT_SIZE],
3                          double res[MAT_SIZE][MAT_SIZE])
4  {
5      /* this is ijk */
6      for (int i = 0; i < MAT_SIZE; ++i)
7      {
8          for (int j = 0; j < MAT_SIZE; ++j)
9          {
10             res[i][j] = 0;
11             for (int k = 0; k < MAT_SIZE; ++k)
12             {
13                 res[i][j] += mat1[i][k] * mat2[k][j];
14             }
15         }
16     }
17     /* this is ikj */
18     for (int i = 0; i < MAT_SIZE; ++i)
19     {
20         for (int k = 0; k < MAT_SIZE; ++k)
21         {
22             s = mat1[i][k];
23             for (int j = 0; j < MAT_SIZE; ++j)
24             {
25                 res[i][j] += s * mat2[k][j];
26             }
27         }
28     }
29 }

```

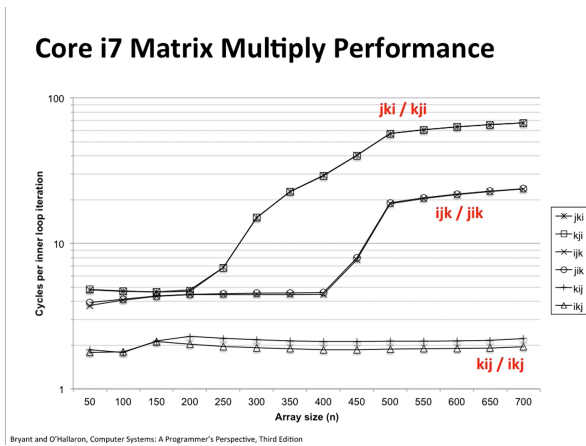


Figure 2: Order of Matrix Multiplication

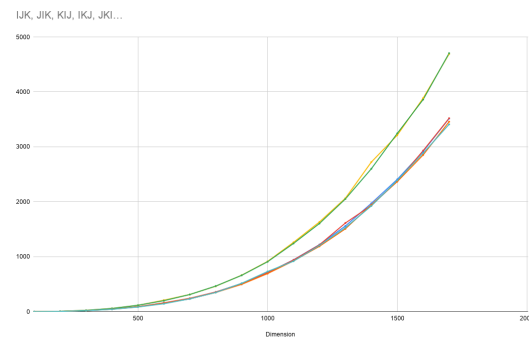


Figure 3: Order of Matrix Multiplication

## 2 Design

### 2.1 Basic

Based on the provided code stub, I have designed 2 basic implements (*mpi\_matrix\_v1.c*, *mpi\_matrix\_v2.c*) and made some basic optimization according to load balancing policy (*mpi\_matrix\_v3.c*) and cache policy (*mpi\_matrix\_cache\_v2.c*, *mpi\_matrix\_cache\_v3.c*).

#### 2.1.1 Version 1 (*mpi\_matrix\_v1.c*)

In this version, the following policies or methods are used.

- *MPI\_Send* and *MPI\_Recv* to communicate master and slaves point-to-point
- 2D array for storing matrices
- Residual rows will all send to the last slave

At the beginning, master will send some parameters (the offset and the number of rows to calculate) and matrices (entire matrix *b* and fragments of matrix *a*) to slaves. As slaves receive the parameters and matrices, they will do brute force multiplication respectively and send back the result to master. After gathering the results, master will check the correctness.

Although the result is correct, master itself is not a worker, i.e., slaves work and master gathers, which does not satisfy the demand. Thus, this version is quickly discarded.

#### 2.1.2 Version 2 (*mpi\_matrix\_v2.c*)

In this version, the following policies or methods are used.

- *MPI\_Bcast*, *MPI\_Scatterv* and *MPI\_Gatherv*
- 2D array for storing matrices
- Residual rows will all send to the last worker

At the beginning, master will broadcast matrix *b* and scatter matrix *a* to all workers (in this version, master itself is a worker). As workers finish brute force multiplication respectively, master will gather the results and check the correctness.

#### 2.1.3 Version 3 (*mpi\_matrix\_v2.c*)

In this version, the following policies or methods are used.

- *MPI\_Bcast*, *MPI\_Scatterv* and *MPI\_Gatherv*

- 2D array for storing matrices
- Rows are distributed as equally as possible

Similar to the process of version 2, but the matrix  $a$  is split as equal as possible.

#### 2.1.4 Version 2 – cache (*mpi\_matrix\_cache\_v2.c*)

In this version, the following policies or methods are used.

- *MPI\_Bcast*, *MPI\_Scatterv* and *MPI\_Gatherv*
- 1D array for storing matrices and row-major order
- Cache and memory access optimization
- Residual rows will all send to the last worker

Similar to the process of version 2, but the matrices are stored in 1D array and the brute force multiplication in each worker is changed from  $ijk$  to  $ikj$ , which increases the rate of cache hit.

#### 2.1.5 Version 3 – cache (*mpi\_matrix\_cache\_v2.c*)

In this version, the following policies or methods are used.

- *MPI\_Bcast*, *MPI\_Scatterv* and *MPI\_Gatherv*
- 1D array for storing matrices and row-major order
- Cache and memory access optimization
- Rows are distributed as equally as possible

Similar to the process of version 3, but the matrices are stored in 1D array and the brute force multiplication in each worker is changed from  $ijk$  to  $ikj$ , which increases the rate of cache hit.

## 2.2 Optimized

There are multiple ways to optimize the parallelisation for improving computation efficiency. One direction is GEMM (General Matrix Multiplication) (*mpi\_matrix\_v4.c*, *mpi\_matrix\_v5.c*), another is using computing or parallel libraries (*mpi\_matrix\_v6.c*).

#### 2.2.1 Version 4 (*mpi\_matrix\_v4.c*)

In this version, the following policies or methods are used.

- *MPI\_Bcast*, *MPI\_Scatterv* and *MPI\_Gatherv*
- 1D array for storing matrices and column-major order
- Cache and memory access optimization
- Rows are distributed as equally as possible and despite the last worker, number of distributed rows are the multiple of 4.
- Unroll for  $1 \times 4$

The parallel calculation roadmap is similar to version 3 – cache. However, the way of storing matrix is in column-major order. The iteration is unrolled manually for  $1 \times 4$  and the number of distributed rows is modified correspondingly.

### 2.2.2 Version 5 (*mpi\_matrix\_v5.c*)

In this version, the following policies or methods are used.

- *MPI\_Bcast*, *MPI\_Scatterv* and *MPI\_Gatherv*
- 1D array for storing matrices and column-major order
- Cache and memory access optimization
- Rows are distributed as equally as possible and despite the last worker, number of distributed rows are the multiple of 4.
- Unroll for  $4 \times 4$

Similar to the process of version 4, but unroll for  $4 \times 4$ .

### 2.2.3 Version 6 (*mpi\_matrix\_v6.c*)

In this version, the following policies or methods are used.

- *MPI\_Bcast*, *MPI\_Scatterv* and *MPI\_Gatherv*
- 1D array for storing matrices and column-major order
- Cache and memory access optimization
- Rows are distributed as equally as possible and despite the last worker, number of distributed rows are the multiple of 4.
- Unroll for  $4 \times 4$
- Use OpenMP

Similar to the process of version 5, but unroll for  $4 \times 4$ .

## 3 Running Result

All programs were run in single-VM and double-VM for 1,2,4,8,16,32 processes. Instructions for running programs are listed below for instance.

Instruction on single-VM

```
1 mpicc mpi_matrix_v1.c -o mat_v1
2 mpirun --oversubscribe -np 4 ./mat_v1
```

Instruction on double-VM

```
1 mpicc mpi_matrix_v1.c -o mat_v1
2 mpirun --oversubscribe -np 4 -N 2 --host 192.168.56.101,192.168.56.102 -mca btl_tcp_if_include
   192.168.56.0/24 ./mat_v1
```

The running logs and statistics [Figure 4] are packed in the zip. All results are correct comparing with the brute-force multiplication. The x-axis and the y-axis of the following figures [Figure 5, 6, 7, 8] are MPI processes and computation time (seconds), respectively.

	mpi_matrix_v1.c	mpi_matrix_v2.c	mpi_matrix_cache_v2.c	mpi_matrix_v3.c	mpi_matrix_cache_v3.c	mpi_matrix_v4.c	mpi_matrix_v5.c	mpi_matrix_v6.c
SINGLE-VM time (sec)	basic					optimized		
	last		equal		unroll		block	
	non-cache		cache		non-cache		cache	
	send-recv	scatterv-getherv	scatterv-getherv	scatterv-getherv	scatterv-getherv	scatterv-getherv	scatterv-getherv	scatterv-getherv
1	NULL	0.47721	0.299547	0.429566	0.302921	0.096637	0.076865	0.073342
2	0.372626	0.218272	0.148688	0.204736	0.161382	0.082969	0.040513	0.03886
4	0.201945	0.145762	0.081822	0.163776	0.08798	0.041347	0.044641	0.055514
8	0.207485	0.244012	0.139796	0.196819	0.110111	0.054966	0.047725	0.068059
16	0.288931	0.301524	0.145832	0.227806	0.132444	0.099234	0.05637	0.05161
32	0.250144	0.357883	0.312076	0.303504	0.164735	0.132245	0.075052	0.064341
DOUBLE-VM time (sec)	basic					optimized		
	last		equal		unroll		block	
	non-cache		cache		non-cache		cache	
	send-recv	scatterv-getherv	scatterv-getherv	scatterv-getherv	scatterv-getherv	scatterv-getherv	scatterv-getherv	scatterv-getherv
1	NULL	0.452304	0.357318	0.461458	0.364195	0.124399	0.09642	0.078811
2	0.554697	0.310668	0.251699	0.3469	0.249141	0.134429	0.124926	0.125154
4	0.327214	0.238296	0.199858	0.241153	0.209976	0.141827	0.106509	0.107024
8	4.830148	0.533553	0.610343	0.518451	0.473597	0.396306	0.371116	0.403691
16	21.064986	0.953823	0.648883	0.737046	14.65135	9.633081	9.789566	0.694965
32	15.906701	2.135606	4.133428	1.733726	31.744913	0.965467	1.243169	26.477465

Figure 4: Statistics

### 3.1 Single-VM

It is obvious that no matter which version is, there is a tendency that the computation time is from high to low then high again as MPI process grows. The highest performance happens in either 2-process or 4-process.

Since parallel calculation is more efficient than sequence calculation, the time cost is decreasing when MPI process grows in small range, especially for 1-process to 2-process. However, since the number of cores on the virtual machine is limited, plenty of processes have to wait for switched to running. On the contrary, it slows down the calculation.

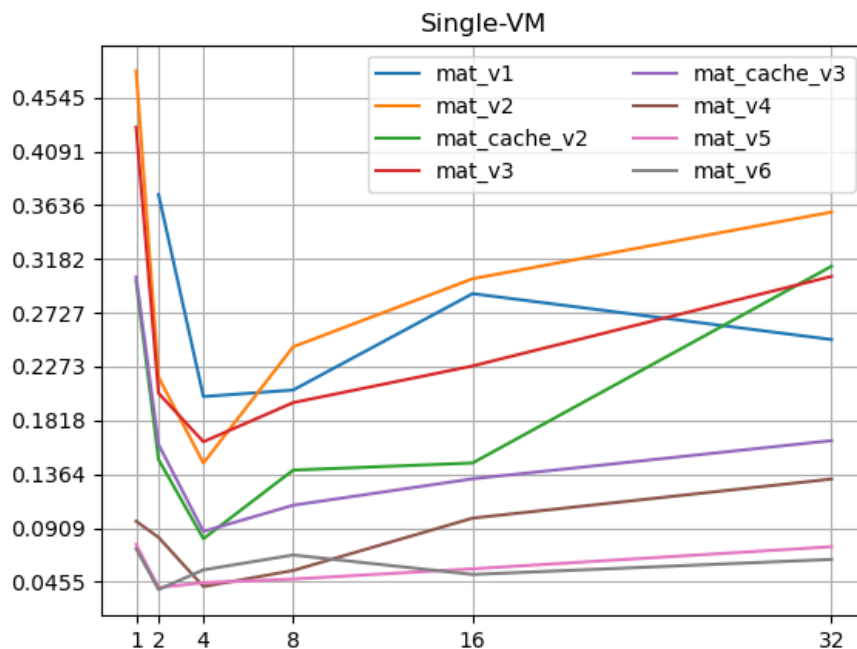


Figure 5: Single-VM: MPI processes vs. computation time

### 3.2 Double-VM

The performance is poor as more processes, especially for version 1, 3-cache, 4, 5, 6. Concentrating on process 1, 2, 4, the time cost trend is also from high to low, but version 4, 5, 6 are extraordinary.

I have no idea why extraordinary situations happened.

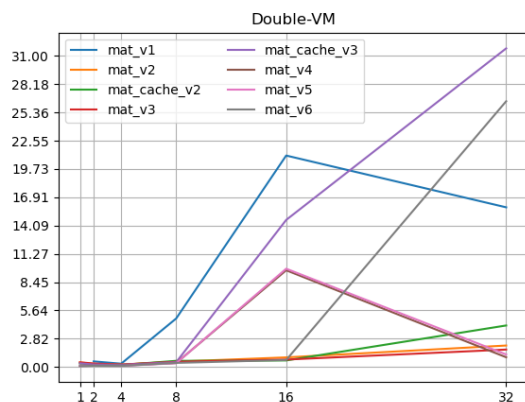


Figure 6: Double-VM: MPI processes vs. computation time

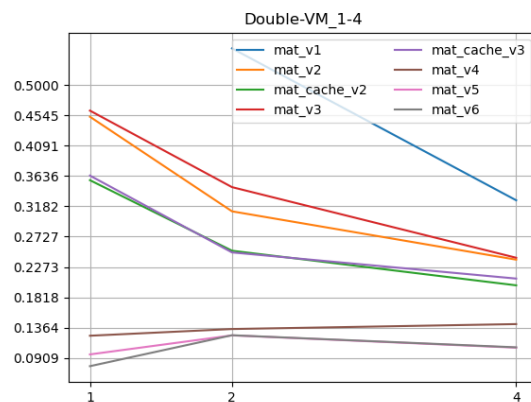


Figure 7: Double-VM for process 1, 2, 4: MPI processes vs. computation time

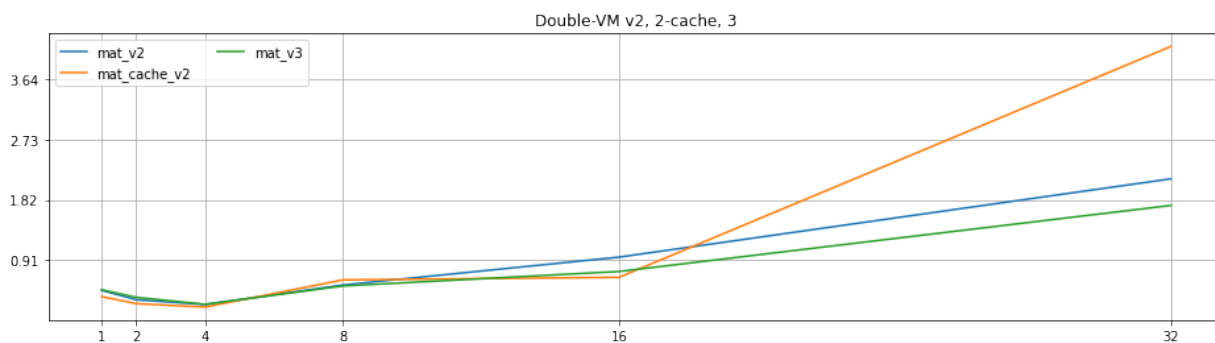


Figure 8: Dingle-VM for version 2, 2-cache, 3: MPI processes vs. computation time

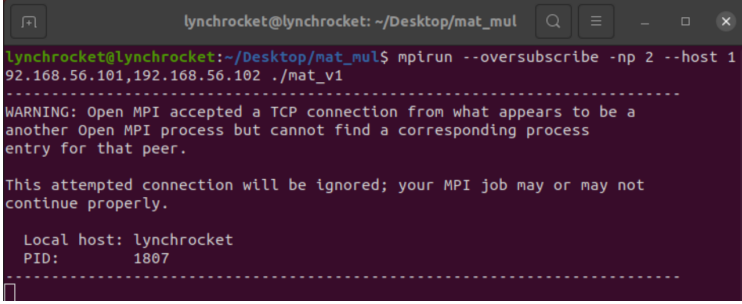
### 3.3 Compare Single with Double VM

Except the extraordinary versions, the tendency of computation efficiency as MPI process grows on single-VM and double-VM are similar. However, compared with single-VM, the performance of double-VM is poorer overall. It is probably because the cost for communication between virtual machine is much more expensive than the improvement of computation efficiency.

## 4 Problems

### 4.1 Connection refused

There was a confused problem happened in running programs on double-VM.

A terminal window titled 'lynchrocket@lynchrocket: ~/Desktop/mat\_mul' showing the execution of an MPI program. The command 'mpirun --oversubscribe -np 2 --host 192.168.56.101,192.168.56.102 ./mat\_v1' is entered. The output shows a warning message: 'WARNING: Open MPI accepted a TCP connection from what appears to be a another Open MPI process but cannot find a corresponding process entry for that peer.' followed by 'This attempted connection will be ignored; your MPI job may or may not continue properly.' and 'Local host: lynchrocket PID: 1807'.

```
lynchrocket@lynchrocket: ~/Desktop/mat_mul
lynchrocket@lynchrocket:~/Desktop/mat_mul$ mpirun --oversubscribe -np 2 --host 1
92.168.56.101,192.168.56.102 ./mat_v1
-----
WARNING: Open MPI accepted a TCP connection from what appears to be a
another Open MPI process but cannot find a corresponding process
entry for that peer.

This attempted connection will be ignored; your MPI job may or may not
continue properly.

Local host: lynchrocket
PID:      1807
-----
```

Figure 9: Problem 1

After searching on the Internet, I found the reason. If no concrete network adapter or network segment were specified for communication between hosts, then OpenMPI will attempt to communicate with all network adapters, causing an error. So I set parameters (`-mca btl_tcp_if_include 192.168.56.0/24`) to specify the network segment to run on.