AMS 530 PRINCIPLES OF PARALLEL COMPUTING
PROJECT 3 PARALLEL MATRIX MULTIPLICATION

Wenhan Gao
Department of Applied Mathematics and Statistics, Stony Brook University, Stony Brook, NY 11794, USA
wenhan.gao@stonybrook.edu

1 Problem Description

In this project, we wish to write a parallel program to perform matrix multiplication. We implement Cannon's method and Fox's method to multiply two large matrices $A, B \in \mathbb{R}^{N\times N}$, whose elements $A_{ij}, B_{ij} \in (0, 1]$ are random floating point numbers. We wish to test the algorithms with different sizes of matrices, $N = 2^8, 2^{10}, 2^{12}$, on different sizes of processor clusters, $P = 2^2, 2^4, 2^6$.

2 Algorithm Description and Pseudo-code

Both Cannon's method and Fox's method are parallel "naive" matrix multiplication of complexity $O(N^3)$. Every element C_{ij} in the resulting matrix C is the sum of the products of corresponding components of the i-th row of A and the j-th column of B:

$$C_{ij} = \sum_{k=1}^{N} A_{ik} \cdot B_{kj}.$$

This process involves two steps: multiplication and summation. A parallel matrix multiplication algorithm is to efficiently distribute operations onto different processors.

Algorithm 2.1: Sequential Naive Matrix Multiplication

Result: $C \in \mathbb{R}^{N\times N}$
Require: $A \in \mathbb{R}^{N\times N}, B \in \mathbb{R}^{N\times N}$
Initialize $C := \mathbf{0}_{[N,N]}$, a matrix of zeros;
for $i = 0;\ i < N;\ i++$ **do**
  for $j = 0;\ j < N;\ j++$ **do**
    for $k = 0;\ k < N;\ k++$ **do**
      $C_{ij} = C_{ij} + A_{ik}B_{kj}$;
    end
  end
end
return C;

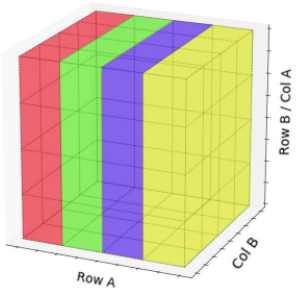
2.1 Cannon's Method

Cannon's method is also called Row-Column Partition method or Ring method. As the name Row-Column Partition suggests, we partition matrix A by rows and matrix B by columns and distribute blocks of rows of matrix A and blocks of columns or matrix B to different processors. In each iteration, we multiply local rows and columns to get some elements of the resulting matrix C. After all processors have done their local computations, we roll up local blocks of rows of A by one processor unit, in other words, we send the block of rows in processor 2 to 1, 1 to 0, etc.. At every iteration, a processor will have a different block of rows of A and a fixed block of columns of B. When every processor have "touched" all the blocks of rows of matrix A and finished calculation with its local columns, the matrix multiplication process is complete. As a result, a processor will calculate a block of columns of matrix C.

This distribution of computations can be illustrated by a 3D super-cube shown in Figure 2.1, where each small cube represent a multiplication and, at the end, an element of C is obtained by summing up all the vertical cubes.

The pseudo-code is in Algorithm 2.2. Note that here we assume that the matrix can be partitioned evenly as specially designed in the project.

Figure 2.1: Cannon's Method; Image From [1]



Algorithm 2.2: Cannon's Algorithm

Result: $C \in \mathbb{R}^{N \times N}$
Require: $A \in \mathbb{R}^{N \times N}$, $B \in \mathbb{R}^{N \times N}$, size (number of processors), rank
1 Partition matrix A into blocks, each has $\frac{N}{P}$ rows;
2 Partition matrix B into blocks, each has $\frac{N}{P}$ columns;
3 Distribute blocks to all processors, in the order of ranks, by MPI_Scatter;
4 Initialize local blocks of C in each processor of $\frac{N}{P}$ columns;
5 **for** *iter = 0; iter < size; iter++* **do**
6 Multiply local rows with local columns and record corresponding C_{ij} in local columns of C ; ▷ 3.3
7 Send local blocks of rows of A to processor whose rank is 1 less than self or the last processor if self is 0 by MPI_Isend and receive from corresponding processors by the same logic ; ▷ 3.4
8 **end**
9 Gather all the elements of C to the root processor;

Note that in above pseudo-code, we omit details of indexing and looping to keep it short, explanatory, and readable. We make comments that refers to the details in lines that are missing such details.

2.2 Fox's Method

Fox's method is also called Broadcast-Multiply-Roll (BMR) method. In Fox's method, we partition both matrices A and B into blocks. A example can be found in Figure 2.2, where the matrix is divided into 4 sub-matrices such as M_{11}, we call a sub-matrix a block. Given the matrices' sizes in this project, let P be the number of processors, each processor stores a sub-matrix of size $\frac{N}{\sqrt{P}} \times \frac{N}{\sqrt{P}}$ of A and a sub-matrix of the same size of B, respectively. We group processors as a matrix form to explain the communication as shown in Figure 2.3.

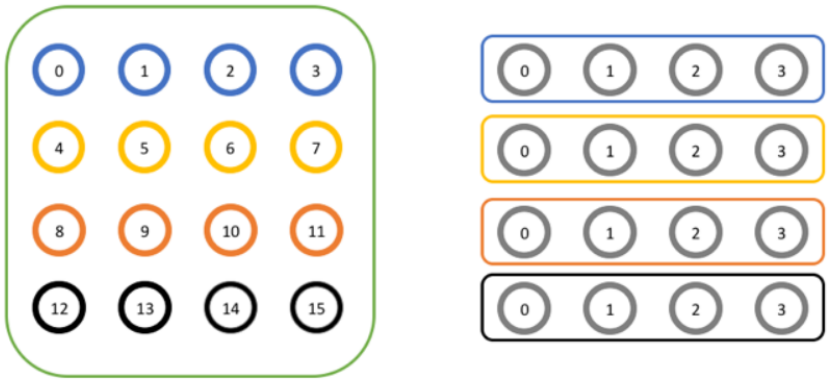
Figure 2.2: Blocks

$$\begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{bmatrix} = \begin{bmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{bmatrix}$$

In fox algorithm, we iterate $\sqrt{P}$ steps. In each iteration, the algorithm broadcasts the "diagonal" blocks of matrix A in processor rows, and then multiply local blocks of A and B to get a local block of sub-matrix of C. In each iteration, processors in the same row will use the same blocks of A to multiply with their local blocks of B. After all processors have done their local computations, we shift the diagonal of the processor group up by one and shift the row of the processor group up by one processor row unit, and this shifting means communication; processors send and receive or

broadcast their local blocks of A and B. After each iteration, we add, element-wise, the local submatrix of C with that from the previous iteration.

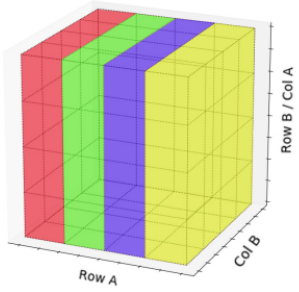
Figure 2.3: Fox Algorithm Processor Grouping and Communication; Image From [2]



This distribution of computations can be illustrated by a 3D super-cube shown in Figure 2.4, where each small cube represent a multiplication and, at the end, an element of C is obtained by summing up all the vertical cubes.

The pseudo-code is in Algorithm 2.3. Note that here we assume that the matrix can be partitioned evenly as specially designed in the project. Also, in the pseudo-code, we omit details of indexing and looping to keep it short, explanatory, and readable. We make comments that refers to the details in lines that are missing such details.

Figure 2.4: Fox's Method; Image From [1]



2.3 Main Structure of the Program

The main program contains two C scripts. One for Algorithm 2.2 and the other for Algorithm 2.3. In each program, there are additional features such as recording running time, populating matrices A and B with random floating numbers, printing a vector and a matrix, check if a block is diagonal, etc..

We take the maximum time across all processors as the global elapsed running time. We record the global elapsed running time that each algorithm takes with different matrix sizes and network sizes.

2.3.1 Slurm Configuration

To run the program on Seawulf, we load our C compiler module (mvapich2/gcc12.1/2.3.7), and we compile the C code with mpicc. Upload the C file and slurm file to Seawulf, and then run the program with interactive shell, example commands:

module load slurm

srun -N 1 -n 16 -p short-28core –pty bash (request a 96core processor for the tests with 64 processors)

module load mvapich2/gcc12.1/2.3.7

Algorithm 2.3: Fox's Algorithm

Result: $C \in \mathbb{R}^{N \times N}$
Require: $A \in \mathbb{R}^{N \times N}$, $B \in \mathbb{R}^{N \times N}$, size (number of processors), rank
Partition matrix A into sub-matrices of size $\frac{N}{\sqrt{P}} \times \frac{N}{\sqrt{P}}$;
Partition matrix B into sub-matrices of size $\frac{N}{\sqrt{P}} \times \frac{N}{\sqrt{P}}$;
Distribute submatrices to all processors, in the order of ranks, by MPI_Scatter;
Grouping processors and initiate sub-communicators for each processor row; ▷ 3.5
Initialize local submatrix of C of size $\frac{N}{\sqrt{P}} \times \frac{N}{\sqrt{P}}$ in each processor with all 0s;
for *iter = 0; iter* < $\sqrt{P}$*; iter++* **do**
    Broadcast "current" diagonal submatrix of A in processor rows;
    Multiply local submatrix of A with local submatrix of B to get local submatrix of C, element-wise addition with that from the previous iteration ; ▷ 3.6
    Roll up local submatrices of A up by one processor row unit ; ▷ 3.7
    Processor "diagonal" is moved up by one ; ▷ 3.8
end
Gather all the submatrices of C to the root processor;

mpicc fox.c -o fox -lm

mpirun ./fox

Slurm scripts are also included, but I used interactive shell to run my programs.

3 Results

First of all, we compare the resulting matrix C of these two methods with a small $N = 8$ and $P = 4$ to the resulting matrix produced by the naive method on one processor to verify that the code is running correctly. As shown in Figures 3.9 and 3.10, the distributed matrix multiplication algorithms are running correctly. For large matrices and processor networks as assigned in the project, timing results are shown in Table 1.

		$P = 2^2$	$P = 2^4$	$P = 2^6$
$N = 2^8$	Cannon	0.011798	0.009577	0.006894
	Fox	0.011626	0.008344	0.006421
$N = 2^{10}$	Cannon	0.442445	0.149160	0.073863
	Fox	0.827565	0.215029	0.091342
$N = 2^{12}$	Cannon	20.586222	6.119073	2.000042
	Fox	59.224778	15.112083	4.346956

Table 1: Performance Table

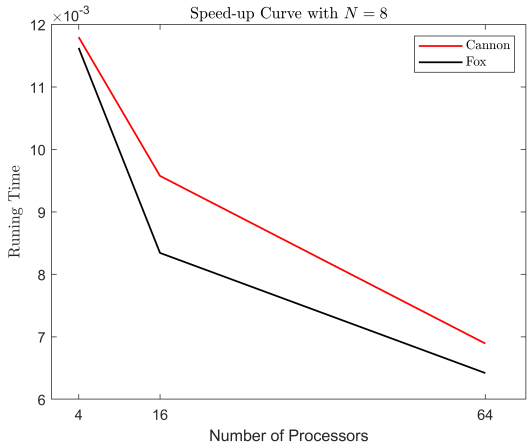
Timing plots are shown in Figure 3.2 with semilogy scale. We also provide the speed-up curve plots with respect to the number of processors individually for each matrix size in Figure 3.1 to show the speed-up.

3.1 Analysis of Program Performance

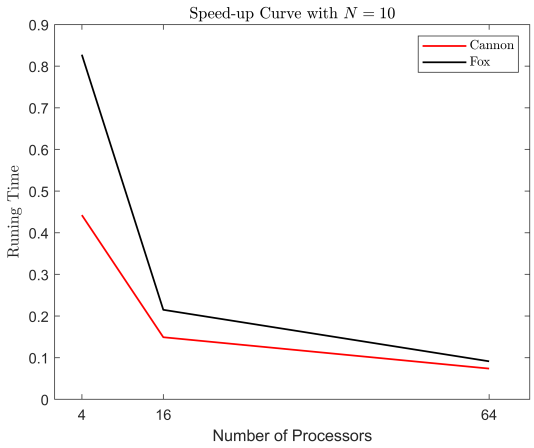
Overall, the program is robust and accurate. We notice that Cannon's method always outperforms Fox's method, and this is a little bit surprising to me. There are two possible reasons. The first possible reason is that, the timing results are obtained from different processors, the one that Cannon's algorithm is on is faster in communication than the one that Fox's algorithm is on. The second possible reason is that the algorithm that I coded to implement is not the most efficient, however, this is very unlikely because most portions of the two programs are the same. From above results, we can conclude that:

- With the same number of processors, the computational cost increases exponentially with the matrix sizes.

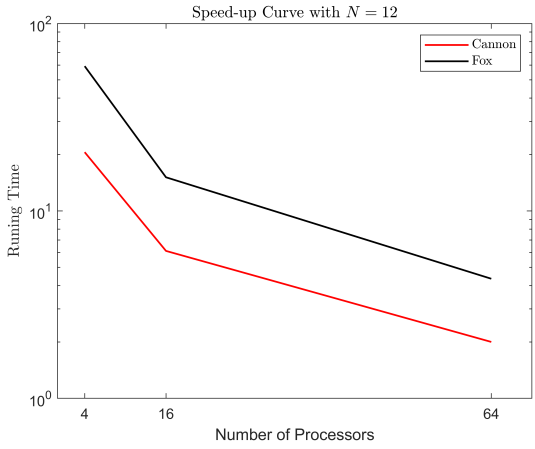
Figure 3.1: Individual Speed-up Curves



(a) $N = 2^8$



(b) $N = 2^{10}$



(c) $N = 2^{12}$

- Under the same matrix size, the running time decreases as the number of processors increases.
- When the scale is small, the increases with respect to the number of processors is not as obvious as larger scale tasks. This is partially because the running time to start up the communication pipeline and also time measurement anomalies.
- We can observe almost a linear speed up with the number of processors.

3.2 Source Code for Submission

Source code will be submitted along with this report. Source code can also be found in this git repo (Clickable Link).

References

[1] WENJING CUN et. al. A unified framework for parallel matrix multiplication. .

[2] Introduction to groups and communicators; MPI Tutorial. https://mpitutorial.com/tutorials/introduction-to-groups-and-communicators/. [Accessed 02-Nov-2022].

Appendix

Figure 3.2: Performance Result

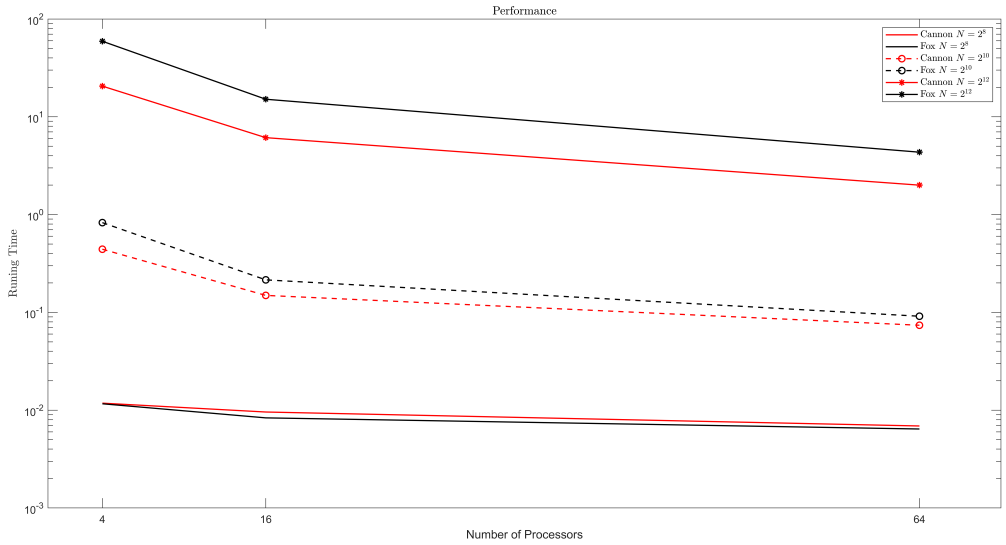


Figure 3.3: Cannon's Algorithm Detail 1

```
for (iter = 0; iter < size; iter++){ // loop through all blocks of rows
    for (r = 0; r < localRCs; r++){  //loop through local rows and columns to calc sum
        for (c = 0; c < localRCs;c++){
            for (e = 0; e < N;e++){ // loop through all elements in a row/column
                sum += localA[e + r*N]*localB[e + c*N];
            }
            localC[k *localRCs+c*N + r] = sum; //put the sum at the correct index
            sum = 0;
        }
    }
```

Figure 3.4: Cannon's Algorithm Detail 2

```
// send local rows of matrix A to rank -1, i.e., roll up all rows of A by one processor unit
MPI_Isend(&localA, localSize, MPI_FLOAT, (rank == 0 ? size-1 : rank -1), 0, MPI_COMM_WORLD, & request);
MPI_Irecv(&localA, localSize, MPI_FLOAT, (rank == size-1 ? 0 : rank +1), 0, MPI_COMM_WORLD, & request);
// make sure all processors has received the information, then go to the next iteration
MPI_Wait(&request, &status);
```

Figure 3.5: Fox's Algorithm Detail 1

```
MPI_Comm row_comm;
MPI_Comm_split(MPI_COMM_WORLD, color, rank, &row_comm);
int row_rank, row_size;
MPI_Comm_rank(row_comm, &row_rank);
MPI_Comm_size(row_comm, &row_size);
```

Figure 3.6: Fox's Algorithm Detail 2

```
// calc C
for (r =0; r < xy; r++){
        for (c = 0; c < xy; c++){
            for (e = 0; e < xy;e++){
                sum+=localA[e +r*xy]* localB[e*xy+c];
            }
            localC[r*xy+c] += sum;
            sum = 0;
        }
}
```

Figure 3.7: Fox's Algorithm Detail 3

```
// shift B up by one processor row unit
MPI_Isend(&localB, localSize, MPI_FLOAT, (rank < XY ? rank + size-XY : rank -XY), 0, MPI_COMM_WORLD, & request);
MPI_Irecv(&localB, localSize, MPI_FLOAT, (rank >= size-XY ? rank - size +XY : rank +XY), 0, MPI_COMM_WORLD, & reques
```

Figure 3.8: Fox's Algorithm Detail 4

```
int iter_diagonal(int rank, int XY, int iter){
    int diag = rank/XY*XY + rank/XY;
    if (diag + iter < (rank/XY+1)*XY){
        return diag + iter;
    }
    else{
        return (diag + iter)-XY;
    }
}
```

Figure 3.9: Fox's Algorithm Result

```
[wenhgao@sn001 ~]$ mpirun ./fox
Total global elapsed running time for PMM is 0.000669 seconds
Matrix A is:
[-0.026192 0.185182 -0.979547 0.991897 0.203131 0.053560 0.528873 0.777945
-0.568991 0.577396 -0.134913 -0.318136 -0.967951 -0.272606 -0.928154 -0.636893
0.880769 0.475074 -0.422073 0.965087 0.577660 0.959805 -0.350369 0.434331
0.313116 0.213931 0.118487 0.197881 -0.581465 -0.330617 -0.929911 0.536036
-0.419220 0.852080 -0.521920 -0.060237 -0.164186 0.162274 -0.362681 -0.433916
0.608332 -0.871363 0.675809 0.877355 0.251792 -0.415275 0.579229 0.272738
0.131871 0.318714 -0.058015 0.665692 0.158046 0.263243 0.424572 0.745799
0.609341 0.759723 0.200954 0.379495 -0.816183 0.774627 -0.915019 0.923054 ]
Matrix B is:
[0.735955 -0.570580 0.029637 -0.936135 -0.889310 -0.821252 0.630984 -0.672203
0.575365 -0.866708 -0.893240 0.019678 -0.477260 0.518847 -0.855186 0.045654
0.548289 0.901222 0.466971 0.781057 0.018647 0.106534 0.737750 0.080639
0.011727 -0.821797 0.895795 0.150536 0.925274 -0.509619 0.032490 0.950858
0.011110 0.158706 0.817167 -0.944259 -0.100432 0.485444 -0.403394 0.950434
0.780015 -0.273181 0.326518 -0.905656 -0.453261 0.321881 -0.879239 -0.839992
0.124818 0.609951 -0.741522 0.777799 -0.172034 0.795365 0.829327 0.032904
0.874436 0.285151 0.637079 -0.547254 0.964220 0.763046 0.047366 0.216852 ]
Matrix C is:
[0.352139 -1.281457 0.551848 -0.842293 1.448872 0.637851 -0.518903 1.224396
-1.060404 -0.862809 -1.478093 1.178258 -0.300093 -0.867212 -1.132430 -0.764181
1.792588 -2.348168 1.591120 -2.924474 -0.138903 -0.371526 -1.477476 0.138715
0.509126 -0.836222 0.498719 -0.334728 0.689831 -0.953658 -0.112097 -0.192286
-0.405089 -1.335506 -1.159902 -0.044064 -0.512375 0.114448 -1.777991 -0.177048
0.316849 0.880640 1.712285 0.512897 1.025540 -0.669446 2.412951 1.106375
1.168663 -0.526020 0.663859 -0.527833 0.856365 0.779860 -0.118298 0.658995
2.288291 -1.772954 0.625770 -1.489095 0.228709 -0.448547 -1.171604 -1.254216 ]
Matrix C calculated using one processor is:
[0.352139 -1.281457 0.551848 -0.842293 1.448872 0.637851 -0.518903 1.224396
-1.060404 -0.862809 -1.478093 1.178258 -0.300093 -0.867212 -1.132430 -0.764181
1.792588 -2.348168 1.591120 -2.924474 -0.138903 -0.371527 -1.477476 0.138715
0.509126 -0.836222 0.498719 -0.334728 0.689831 -0.953658 -0.112097 -0.192286
-0.405089 -1.335506 -1.159902 -0.044064 -0.512375 0.114448 -1.777991 -0.177048
0.316849 0.880640 1.712285 0.512897 1.025540 -0.669446 2.412951 1.106375
1.168663 -0.526020 0.663859 -0.527833 0.856365 0.779859 -0.118299 0.658995
2.288291 -1.772954 0.625770 -1.489095 0.228709 -0.448547 -1.171604 -1.254216 ]
Total elapsed running time for single processor MM is 0.000422 seconds
[wenhgao@sn001 ~]$
```

Figure 3.10: Cannon's Algorithm Result

```
[wenhgao@sn002 ~]$ mpirun ./ring
Total global elapsed running time for PMM is 0.000197 seconds
Matrix A is:
[-0.294479 0.811652 0.908530 0.735964 0.844181 0.100135 0.364026 0.803509
0.832700 -0.988076 0.069034 0.141580 -0.437979 0.229258 -0.666344 0.698552
0.448587 0.126952 -0.993254 0.034533 -0.830284 -0.610808 -0.098229 -0.179654
0.346726 -0.762993 -0.394114 -0.752449 0.924296 0.163386 0.346216 0.297129
-0.627859 -0.848081 0.286997 -0.770185 0.321119 -0.009546 -0.188293 -0.042639
-0.356929 -0.425807 -0.427044 -0.583137 -0.701448 -0.244796 -0.557814 -0.560429
-0.523587 0.193215 0.777941 0.220122 -0.957964 0.321951 -0.978623 0.012176
0.473681 0.913403 -0.591887 -0.456111 0.910515 -0.648566 0.351425 0.167029 ]
Matrix B is:
[0.579179 -0.912225 -0.503458 0.677416 0.306800 -0.461736 -0.232897 0.286604
0.154025 0.502537 0.773615 0.831791 -0.454291 0.995479 0.869407 0.039177
0.510204 0.357116 -0.137084 0.850927 -0.865332 0.533742 -0.807300 -0.265529
-0.167730 -0.584240 0.378587 0.167907 0.476809 -0.742048 -0.138062 -0.205198
-0.575919 -0.621113 0.186452 0.456712 -0.380993 -0.777109 0.810800 -0.841567
0.194369 0.248956 -0.178256 -0.502748 0.580249 0.644768 -0.947668 -0.185672
-0.408510 0.763410 0.423030 0.099061 -0.789424 -0.146257 -0.720687 -0.335553
0.586369 -0.953363 -0.669734 -0.293334 0.299093 -0.647299 -0.208995 0.827838 ]
Matrix C is:
[0.150278 -0.416549 0.685652 1.507876 -1.204910 -0.282056 0.098464 -0.630863
1.320186 -2.159775 -2.011739 -0.761480 1.746968 -1.302848 -1.366393 1.280466
0.061050 -0.260374 0.054439 -0.459062 0.941636 -0.254468 0.816972 1.086579
-0.459343 -0.953241 -0.904991 -0.574343 -0.006526 -1.427541 -0.039077 -0.349862
-0.353535 0.394098 -0.660435 -0.870541 -0.414922 -0.030206 -0.302439 -0.371925
-0.136731 0.782990 -0.259722 -1.145422 0.573360 0.777323 0.320577 0.473070
1.107701 0.640394 -0.268375 -0.194929 0.511346 1.773254 -0.747581 0.690646
-0.506510 -0.535934 0.698781 1.228129 -0.925621 -0.572161 2.289298 -0.203188 ]
Matrix C calculated using one processor is:
[0.150278 -0.416549 0.685652 1.507876 -1.204910 -0.282056 0.098464 -0.630863
1.320186 -2.159775 -2.011739 -0.761480 1.746968 -1.302848 -1.366393 1.280466
0.061050 -0.260374 0.054439 -0.459062 0.941636 -0.254468 0.816972 1.086579
-0.459343 -0.953241 -0.904991 -0.574343 -0.006526 -1.427541 -0.039077 -0.349862
-0.353535 0.394098 -0.660435 -0.870541 -0.414922 -0.030206 -0.302439 -0.371925
-0.136731 0.782990 -0.259722 -1.145422 0.573360 0.777323 0.320577 0.473070
1.107701 0.640394 -0.268375 -0.194929 0.511346 1.773254 -0.747581 0.690646
-0.506510 -0.535934 0.698781 1.228129 -0.925621 -0.572161 2.289298 -0.203188 ]
Total elapsed running time for single processor MM is 0.000489 seconds
[wenhgao@sn002 ~]$
```