COL380 Assignment 2 - Matrix Multiplication using MPI

Pratyush Pandey (2017EE10938), Harkirat Dhanoa (2017CS50408)

29th February, 2020

Contents

1 Introduction

Matrix multiplication is a binary operation that produces a matrix from two matrices. For matrix multiplication, the number of columns in the first matrix must be equal to the number of rows in the second matrix. The result matrix, known as the matrix product, has the number of rows of the first and the number of columns of the second matrix.
Historically, matrix multiplication has been introduced for making easier and clarifying computations in linear algebra. This strong relationship between matrix multiplication and linear algebra remains fundamental in all mathematics, as well as in physics, engineering and computer science. Here is the algorithm used for Matrix Multiplication. If A is a n x p matrix, B : p x q and hence C : n x q, then the number of operations required to compute the matrix product is $nq(2p-1)$. For simplicity, usually it is analyzed in terms of square matrices of order n. So that the quantity of basic operations between scalars is $O(2n^3 - n^2) = O(n^3)$.

```
void Matrix_Multiply(float *A, float *B, float *C, int m, int n, int p){
  int i, j, k;
  for (i = 0; i $<$ m; i++i){
        for (j = 0; j $<$ p; j++){
            C[i*p + j] = 0;
            for (k = 0; k $<$ n; k++)
                C[i*p + j] += A[i*n + k] * B[k*p + j];
        }
    }
}
```

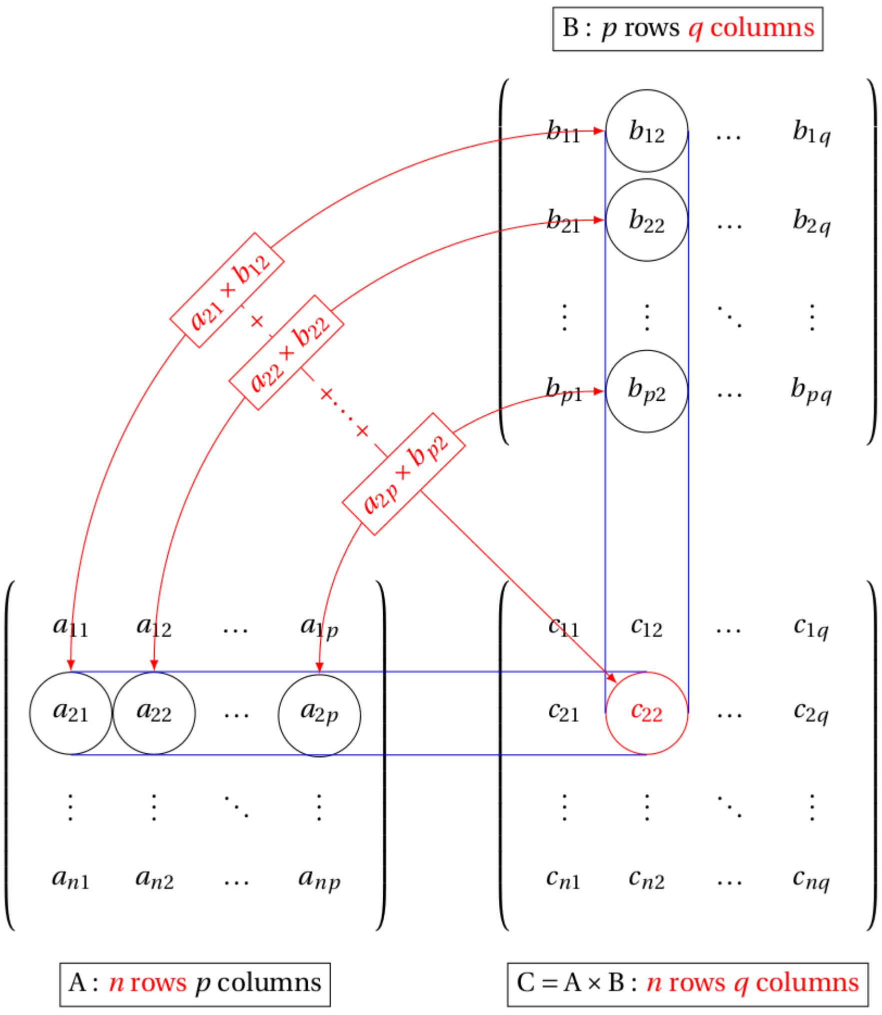

Figure 1: Illustration of matrix multiplication procedure

In this assignment, we have two multiply two rectangular matrices of order m x 32 and 32 x n respectively using MPI. Thus we need to implement data parallelism to distribute tasks effectively between processes, which can communicate with each other to exchange data compute to solve the problem in a faster and more efficient manner

1.1 Parallel 2D Matrix Multiplication Characteristics

- **Computationally independent:** each element computed in the result matrix C, c_{ij} , is, in principle, independent of all the other elements.
- **Data independence:** the number and type of operations to be carried out are independent of the data. Exception is sparse matrix

- **multiplication:** take advantage of the fact that most of the matrices elements to be multiplied are equal to zero.
- **Regularity of data organization and operations** carried out on data: data are organized in two-dimensional structures (the same matrices), and the operations basically consist of multiplication and addition.
- Parallel matrix multiplication follows **SPMD (Single Program - Multiple Data)** parallel computing model

The sequential algorithm clearly has data parallelism. For one, each iteration of the outer loop is independent of all others, because each updates a different element of the output vector B. The inner loop also has data parallelism, because each iteration multiplies B[j] by a unique matrix element A[i, j] before adding it to C[j]. This is a reduction whose elements are products of the form A[i, j] · B[j]. We could, if we wanted, associate a primitive task to each matrix element A[i, j] which would be responsible for the product A[i, j] · B[j]. This would result in mn primitive tasks. We should assume that m and n are very large numbers, otherwise we would not bother to parallelize the algorithm, so we will probably not have anything near mn processors available. Therefore, we will need to agglomerate to reduce the number of tasks. We will consider three alternative decomposition strategies: two one-dimensional strategies and one two-dimensional strategy. This leads to three ways to decompose the matrix: by rows, by columns, or by rectangular subblocks.

Throughout the discussion of parallel algorithms and runtimes, we make the following assumptions -

- Considered the number of processors available in parallel machines as p. We also assume the runtimes aren't affected between computations with other background processed - i.e. runtimes are calculated in isolation with other load on the machines.
- The matrices to multiply will be A and B. Both will be treated as dense matrices (with few 0's), the result will be stored it in the matrix C. The discussion for sparse matrices has been done separately.
- It is assumed that the processing nodes are homogeneous, due this homogeneity it is possible achieve load balancing.

2 Algorithm

While deciding how to implement data parallelism across multiple processes in this task, it is important to note that the allocation of all the necessary data in each subtask will inevitably lead to data doubling and to a considerable increase of the size of memory used. As a result, the computations must be arranged in such a way that the subtasks should contain only a part of the data necessary for computations at any given moment, and the access to the rest of the data should be provided by means of data transmission.

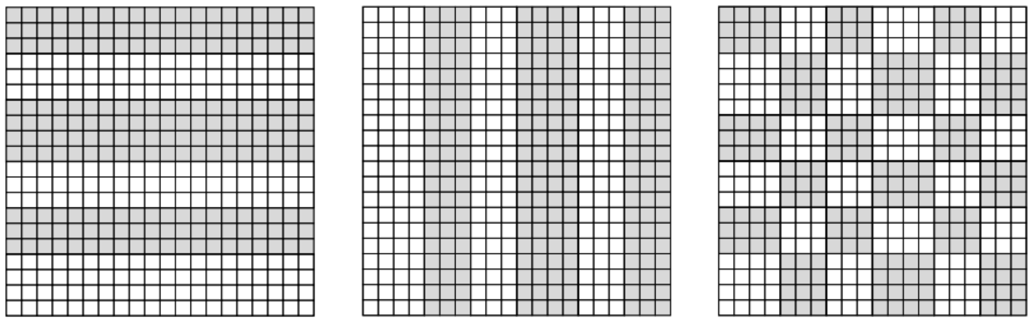
(a) Row-wise block-striped matrix decomposition. (b) Column-wise block-striped matrix decomposition. (c) Checkerboard block decomposition.

Figure 2: Row-wise striped, column-wise striped, and checkerboard block decomposition of a matrix

Here We use 2D Matrix Row Wise Block Striped Data Decomposition (A variant of Fox's Algorithm). Symmetrically, we also tried Column Wise Block Striped Data Decomposition (not described here) to get similar runtimes. The third alternative is to decompose the matrix in two-dimensions, into an r c grid of rectangular subblocks, as shown in Figure above, which is called a checkerboard block decomposition or a 2D-block decomposition - which resulted in improved runtimes.

2.1 Work Distribution between Different Subprocesses

In row/column-wise block-striped decomposition, each process will have a group of either $\lceil n/p \rceil$ or $\lfloor n/p \rfloor$ consecutive rows, each a primitive task.
In a checkerboard block decomposition, each matrix element is a primitive task, and the number p determines the possible decomposition of the matrix. It works best when p is a square number so that the grid can have an equal number of rows and columns. If p is a prime number, the matrix cannot be decomposed using the checkerboard block method, because it degenerates into either the row-wise striped or column-wise striped method, as either the number of rows or the number of columns in the grid will be 1. If p is not square and not prime, there are two factors r and c such that p = rc, and they should be chosen to be as close to each other as possible. Given that the grid is r x c, then each process will have a rectangular block of adjacent matrix elements with either $\lceil m/r \rceil$ or $\lfloor m/r \rfloor$ rows and either $\lceil n/c \rceil$ or $\lfloor n/c \rfloor$ columns.
No matter which decomposition we choose, each process will store in its local memory just the part of the matrix for which it is responsible. The alternative is to replicate the entire matrix within each process's memory. There are two reasons for which this is not a good idea. First is that the solution would not scale well because as the problem size increased, we would not be able to store the entire matrix in each process's local memory; under the assumption that these would be large matrices, we would run out of memory. Secondly, replicating the entire matrix is very inefficient in terms of time and space. In terms of space, we would be storing p copies of an m x n matrix instead of a total of one copy distributed among the p processes. If each entry used b bytes, this would be a waste of $bmn(p1)$ bytes of total memory. In terms of time, it means that the algorithm would have taken time to distribute those matrix elements in the beginning, even though the processes had no need for most of the matrix that they received.
In our algorithm, each process will need the entire B matrix. Even if the process will not need the entire matrix, it is not particularly wasteful of storage specially when products of it's dimensions np is smaller than the matrix A of size mn, and the complexity of storing the vector will not increase the space or time complexity of the algorithm, and will ultimately reduce communication overhead. Nonetheless, it is worth investigating solutions with both replicated vectors and vectors decomposed and distributed among the processes,

2.2 Analysis of Information Dependencies

Two conditions take place

- Basic subtasks is responsible for computation of separate blocks of the matrix C
- Each subtask contains only one block of the matrix A and the entire matrix B at each iteration, based on rank of the process

The indices of the blocks of the matrix C contained in the subtasks are used for enumeration of the subtasks. Thus, subtask i computes i^{th} block of the matrix C. So the set of subtasks forms a square grid, which corresponds to the structure of the block presentation of the matrix C. Here, number of blocks are divided as :

$$\text{Number of Horizontal blocks of Matrix A} = \frac{\text{Number of rows in Matrix A}}{\text{Number of tasks}}$$

In accordance with the Fox algorithm each basic subtasks i contains three matrix blocks:

- The horizontal block C_i of the matrix C, computed by the subtask
- Blocks $A_i$of the matrix A and matrix B, obtained by the subtask in the course of computations

Other than this, partitioning variables are also transferred to each process so they know the range of indices in which they have work with Matrix A and how much of the total matrix C to compute. Lets call these variables block-specifiers.

2.3 Blocking and Non Blocking P2P Communication Algorithm Stages

1. **The initialization stage**: The matrices A and B are initialised by the master task with randomly generated float values in the range [0, 1]. All elements of blocks C_i in all subtasks are set to zero. This stage isn't counted in computation times.
2. **The computation stage**: At this stage the following operations are carried out at each iteration t, $0 \leq$ t $<$ Number of tasks: q.
   - For each process t, $0 \leq$ t $<$ m, the block A_l of subtask i is transmitted to all the subtasks of the same grid row.
   - The block specifiers, which defines the position of the subtask in the row, is computed according to the following expression:

     $$\text{row\_partition} = m/\text{number of processes } |\text{extra\_rows} = m\%\text{number of processes}$$

     And this is how these break the matrices to be send to each processes:

     ```
     total_elems = 0;

     /* send info from master to slave */
     for(i=1; i<size; i++)
     {
         send_rows = row_partition;
         if(i<=extra_rows)
             send_rows += 1;

         MPI_Send(&total_elems, 1, MPI_INT, i, tag, MPI_COMM_WORLD);
         MPI_Send(&send_rows, 1, MPI_INT, i, tag, MPI_COMM_WORLD);
         MPI_Send(&A[total_elems*n], send_rows*n, MPI_FLOAT, i, tag, MPI_COMM_WORLD);
         MPI_Send(&B, n*p, MPI_FLOAT, i, tag, MPI_COMM_WORLD);
         total_elems += send_rows;
     }
     ```

   - Blocks A_i and B obtained by each subtask i as a result of block transmission are multiplied and added to block C_i.

2.4 Collective Communication Algorithm Stages

1. **The initialization stage**: The matrices A and B are initialised by the master task with randomly generated float values in the range [0, 1]. All elements of blocks C_i in all subtasks are set to zero. This stage isn't counted in computation times.

2. **The computation stage**: At this stage the following operations are carried out at each iteration t, $0 \leq$ t $<$ Number of tasks: q.

   - Process 0 scatters matrix A into t blocks A_t one for each remaining process.
   - Process 0 broadcasts matrix B to all the other processes.
   - The block specifiers, which defines the position of the subtask in the row, is computed according to the following expression:

     $$\text{row\_partition} = m/\text{number of processes } |\text{extra\_rows} = m\%\text{number of processes}$$

     And this is how these break the matrices to be send to each processes:

     ```
     /* send info from master to slave */
     MPI_Scatter(A, THIRTY_TWO * row_partition, MPI_INT,locA, THIRTY_TWO *
         row_partition, MPI_INT, 0, MPI_COMM_WORLD);
     MPI_Bcast(B, THIRTY_TWO*n , MPI_INT ,1, MPI_COMM_WORLD );
     MPI_Gather(locC, n * row_partition , MPI_INT, C, n * row_partition , MPI_INT ,
         0, MPI_COMM_WORLD ) ;
     ```

   - Blocks A_i and B obtained by each subtask i as a result of block transmission are multiplied and added to block C_i.

3 Analysis of Algorithms

3.1 Row/column wise striped matrix data decomposition

For an m x n matrix, the sequential algorithm has time complexity $\Theta(mn)$. To simplify the analysis, however, we will assume that the matrix is square, i.e., m = n, so that the sequential algorithm has time complexity $\Theta(n^2)$. We do not include the time spent performing I/O, whether it is a sequential or a parallel algorithm; in particular we do not include the overhead of distributing the matrix to each processor. Similarly, we exclude the communication overhead, which is $\Theta(n \log p)$. Therefore, we start by determining the time complexity of the computational portion of the algorithm. Each task has at most $\lceil n/p \rceil$ rows of the matrix and iteratively computes their inner products with a vector of size n; therefore each task spends time proportional to n $\lceil n/p \rceil$ in computation. Hence, the time complexity of the computational part of the parallel algorithm is

$$\Theta(n^2/p)$$

Including the communication overhead, the time complexity becomes $\Theta(\log p + n)$. The total time complexity would be

$$\Theta((n^2/p) + \log p + n)$$

If we assume that p is very small in comparison to n, then n^2/p is much larger than n and this would be the term that dominates the time complexity.

3.1.1 Scalability and efficiency

As noted above, the time complexity of the sequential algorithm is $\Theta(n^2)$, so $T(n, 1) = n^2$.
Speedup $S_p = \frac{n^2}{n^2/p} = p$
Efficiency $E_p = \frac{n^2}{p(n^2/p)} = 1$
Hence Developed method of parallel computations allows to achieve ideal speed-up and efficiency characteristics.

3.2 2D "Checkboard" Block Decomposition

To simplify the analysis, we will assume again that the matrix is square, i.e., it is n n, so that the sequential algorithm has time complexity $\Theta(n^2)$. We will also assume for simplicity that the number of processes, p, is a square number, so that the grid is $\sqrt{p}x\sqrt{p}$ and that each subblock is of size at most $\lceil n/\sqrt{p} \rceil$ x $\lceil n/\sqrt{p} \rceil$. Although we will not count in our analysis the communication time associated with reading and distributing the matrix and the vector to each processor, we will determine what that time is.
The algorithm to read and distribute the matrix takes each of n rows of the matrix and sends it by subblocks to the processes that own those blocks. As there are $\sqrt{p}$ subblocks in each row, and there are n rows, there are a total of $\lceil n/\sqrt{p} \rceil$ messages sent, each of size $\lceil n/\sqrt{p} \rceil$. The communication time is therefore $\lceil n\sqrt{p} \rceil(\lambda + \lceil n/\sqrt{p} \rceil \beta)$ or approximately $n\sqrt{p}\lambda + n^2/\beta$, which is $\Theta(n^2)$. This is as expected, as the matrix contains n^2 elements that must be delivered to all processes.

The time complexity of the computational portion of the algorithm is the time spent computing the partial inner products. Each task computes the matrix product of an approximately n/p n/p matrix with a vector of size $n\sqrt{p}$ and therefore takes $\Theta(n^2/p)$ steps to perform the computation. The time complexity of the communication overhead not associated with input of the matrix or the vector is the time that all processes take when participating in the simultaneous row reductions of their partial sums of the result vector. There are p processes in each row, and the message size is $n/\sqrt{p}$. This is exactly like the time to broadcast the input vector along a column, so each sum reduction has a complexity of $\Theta((n \log p)/\sqrt{p})$. The total time to perform the matrix-vector multiplication using the two-dimensional block decomposition is therefore

$$\Theta(n^2/p + (n \log p)/\sqrt{p})$$

3.2.1 Scalability and efficiency

As noted above, the time complexity of the sequential algorithm is $\Theta(n^2)$, so $T(n,1) = n^2$.
Speedup $S_p = \frac{n^2}{n^2/p} = p$
Efficiency $E_p = \frac{n^2}{p(n^2/p)} = 1$
Hence, Developed method of parallel computations allows to achieve ideal speed-up and efficiency characteristics.

4 Optimizations

4.1 Algorithmic Optimisations

Rather than the master task giving out the primitive multiplication subtasks to the workers and then waiting to receive and combine the results, we made the master do one part of the primitive multiplication tasks as well - thus making sure all the resources are used adequately.
Then we go forward in resource utilization and update our P2P parts as shown in the graphs below.

METHOD 1: Process 0 (P_0) divides data into p-1 parts and gives it to process P_1,..,P_p which compute product and return back to P_0

METHOD 2: Process 0 (P_0) divides data into p parts and gives it to process P_0,..,P_p and all (including P_0) compute the product and return back to P_0

METHOD 3: Process 0 (P_0) divides data into p parts and gives it to process P_0,..,P_p and all (including P_0) compute the product and send it to P_p

4.2 Running Time Analysis

METHOD 1: Here P_1,...,P_p will be bottleneck because these are the ones which do the matrix multiplication (and not P_0)

METHOD 2: Process 0 (P_0) is the bottleneck since it not only sends and receives communicated data but also does matrix multiplication like other processes.

METHOD 3: Bottleneck will be one of P_0 and P_p because P_0 has to send data to all others and also compute matrix product meanwhile P_p has to receive resulting values from all other processes and also compute matrix product itself.

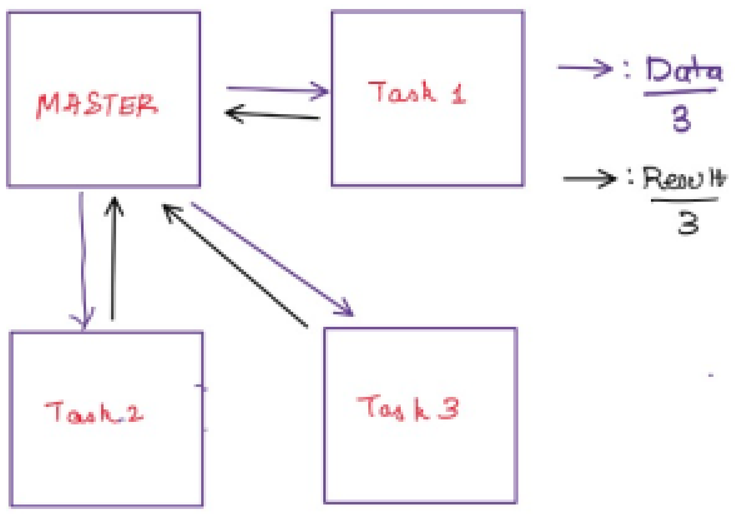

Fig 1: Naive approach (compute and return).

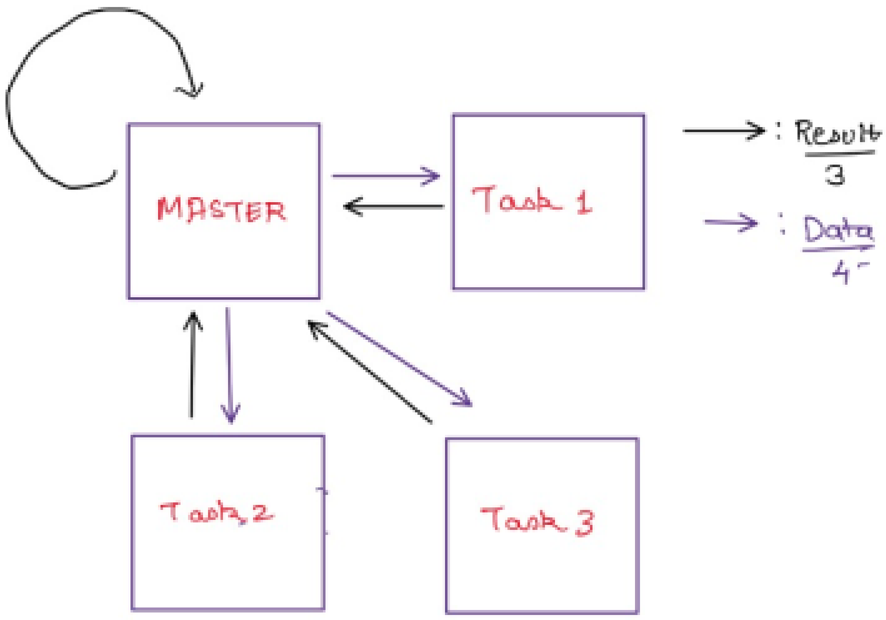

Fig 2: Optimisation 1: MASTER computes as well

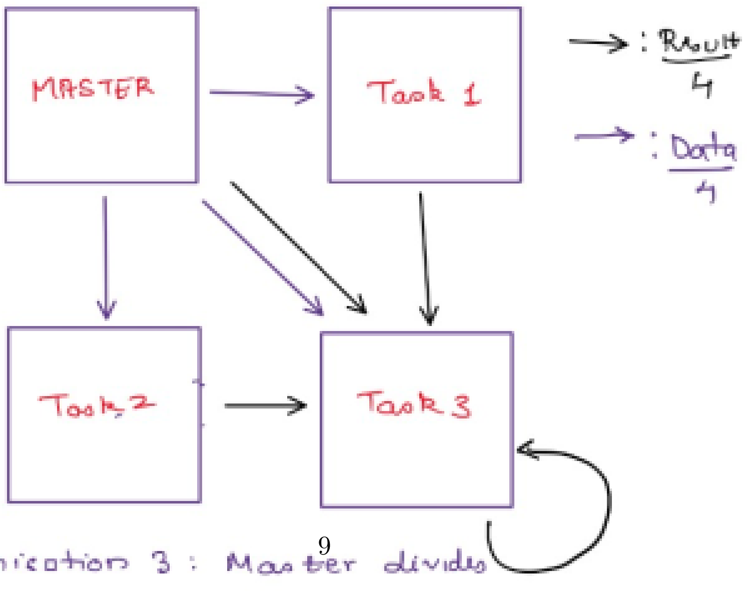

Fig 3: optimisation 3: Master divides works and computes itself; Task 3 combines result.

Further, the values of N (Size of matrix) can be anything - both P2P and collective algorithms used do not rely on the discretisation of values of N based on the number of processes p. N needn't be a multiple of p or a perfect square - and the algorithm can multiple two rectangular matrices with different outer dimensions, none of them a perfect square or a multiple of P.
The extra rows in this scenario is divided such that the first N mod p rows get one extra row to multiply.

4.3 Data Structure

We used $float*$ to represent our matrices to be multiplied and used the row major format to store elements in 1D fashion. This is better than column major since C/$C++$ stores arrays in a contiguous row major fashion in the memory to avoid cache misses.
For dense matrices the 1D approach is likely to be faster since it offers better memory locality and less allocation and de-allocation overhead. Dynamic-1D consumes less memory than the 2D approach. The latter also requires more allocations.
Size: The 2D array will require a tiny bit more memory than the 1D array because of the pointers needed in the 2D array to point to the set of allocated 1D arrays.
Speed: The 1D array may be faster than the 2D array because the memory for the 2D array would not be contiguous, so cache misses would become a problem. We used pointers to pass matrices to avoid duplicacy of function arguments.

4.4 Sparse vs Dense Inputs

Memory: For a full matrix, you store 8 bytes (one double) per entry. For a sparse matrix, you store 12 bytes per entry (one double for the value, and one integer for the column index of the entry). In other words, if your sparsity is below 67% i.e., for nearly any matrix any reasonable person would call sparse, the sparse matrix format will not only yield better memory use but also better compute time.
Speed: While all the times are within an order of magnitude of each other, multiplying a dense and a sparse matrix takes about twice as long as multiplying two sparse matrices together, and multiplying a sparse and dense matrix takes about three times as long. (The sparse-dense multiply is actually one area where Armadillo 8.500 makes massive gains over previous versions. This operation used to take much longer due to using an un-optimised multiplication algorithm.)

4.5 Memory

Declaring an array of suppose size $n = 4000$ will lead to $4000 * 4000 * 8$ *bytes* of memory being allocated, since datatype *double* takes 8 *bytes*. This is 128 MB per such matrix, and in the code we have 5 matrices of size $n * n$. So for $n = 4000$, 640 MB of RAM is being used. Which makes it extremely important to free up memory by deleting this allocated memory once the requirement for it finishes.

4.6 Performance

The distribution of data and computing division across multiple processors offers many advantages:

- With MPI it is required less effort in terms of the timing required for data handling, since each process has its own portion.
- MPI offers flexibility for data exchange.

4.7 Compiler Optimisations

Used compiler flags -
-flto -march=native -mtune=knl -O3 -Ofast
What they do -

- flto - Link Time Optimization (LTO) gives GCC the capability of dumping its internal representation (GIMPLE) to disk, so that all the different compilation units that make up a single executable can be optimized as a single module. This expands the scope of inter-procedural optimizations to encompass the whole program (or, rather, everything that is visible at link time).
- march=native - adds all optimization flags applicable to the hardware I'm compiling on. To check which flags are actually being activated do gcc -march=native -Q –help=target ...
- O3 - use -Ofast instead. It enables all -O3 optimizations. It also enables optimizations that are not valid for all standard compliant programs.

O3 flag isn't used in compile times due to artificial speedups and loss of parabolic $O(n^2)$ runtime graphs.

4.8 Correctness

Correctness of the algorithm was verified by calculating the norm of $\|SM - PM\|$, where SM - Matrix multiplication computed sequentially, and PM - Matrix multiplication result computed parallely which was of the order 10^{-5} for matrix size = 8000 and smaller (10^{-6}) for smaller sizes of n.

5 RunTime Analysis

Size	Number of processes	Sequential	P2P Blocking	P2P Non Blocking	Collective
500	2	0.0318	0.0222	0.0264	0.0132
500	4	0.0318	0.0085	0.0123	0.0078
1000	2	0.1111	0.0895	0.0118	0.051
1000	4	0.1111	0.0322	0.0466	0.0273
2000	2	0.3821	0.3434	0.4085	0.1977
2000	4	0.3821	0.1273	0.1848	0.1087
4000	2	1.5089	1.3714	1.6247	0.8121
4000	4	1.5089	0.5004	0.7333	0.4349
6000	2	3.3192	3.0815	3.6055	1.8445
6000	4	3.3192	1.1089	1.638	0.97
8000	2	5.8305	5.5132	6.4653	3.3699
8000	4	5.8305	2.0031	2.9364	1.7413
10000	2	8.9502	8.5246	9.9834	5.105
10000	4	8.9502	3.1762	4.5641	2.7314
12000	2	13.1551	12.2265	14.8842	7.3472
12000	4	13.1551	4.472	6.5466	3.9122
1000	6	0.0318	0.021	0.0347	0.0216
2000	6	0.1111	0.081	0.1369	0.0829
4000	6	0.3821	0.3335	0.05565	0.302
6000	6	1.5089	0.7469	1.236	1.9375
8000	6	3.3192	1.3331	2.257	1.2382
10000	6	5.8305	2.0267	3.4812	1.8909
12000	6	8.9502	2.9466	5.0504	2.7104

Table 1: Runtime aggregation for different algorithms against input size and number of processes. All times are in seconds.

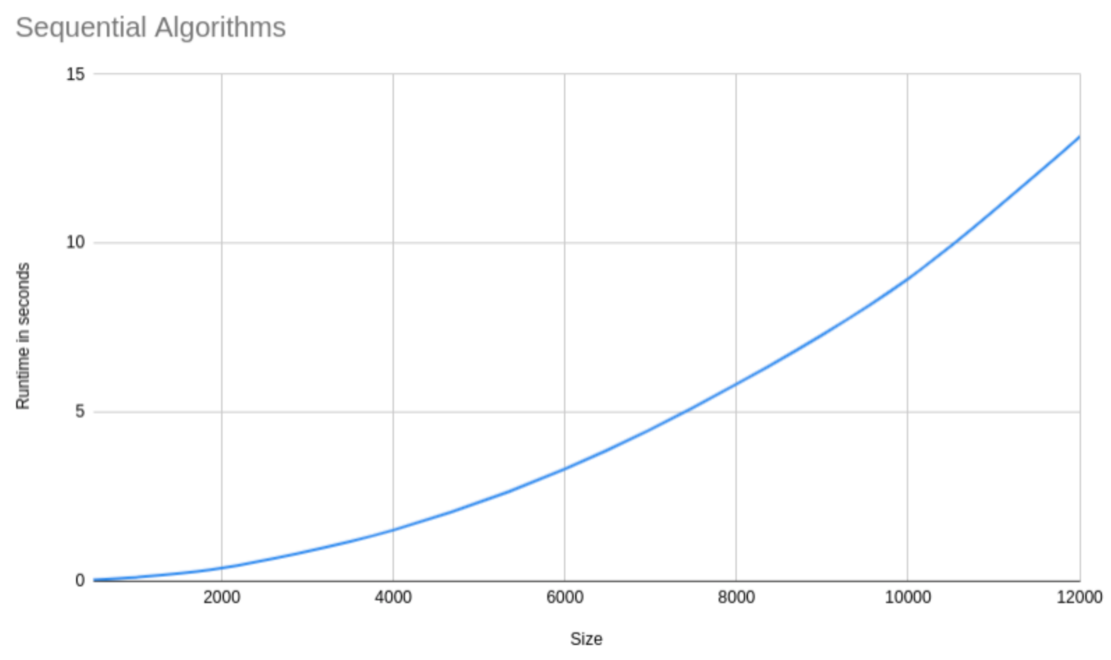

Figure 4: Sequential Runtimes analysis

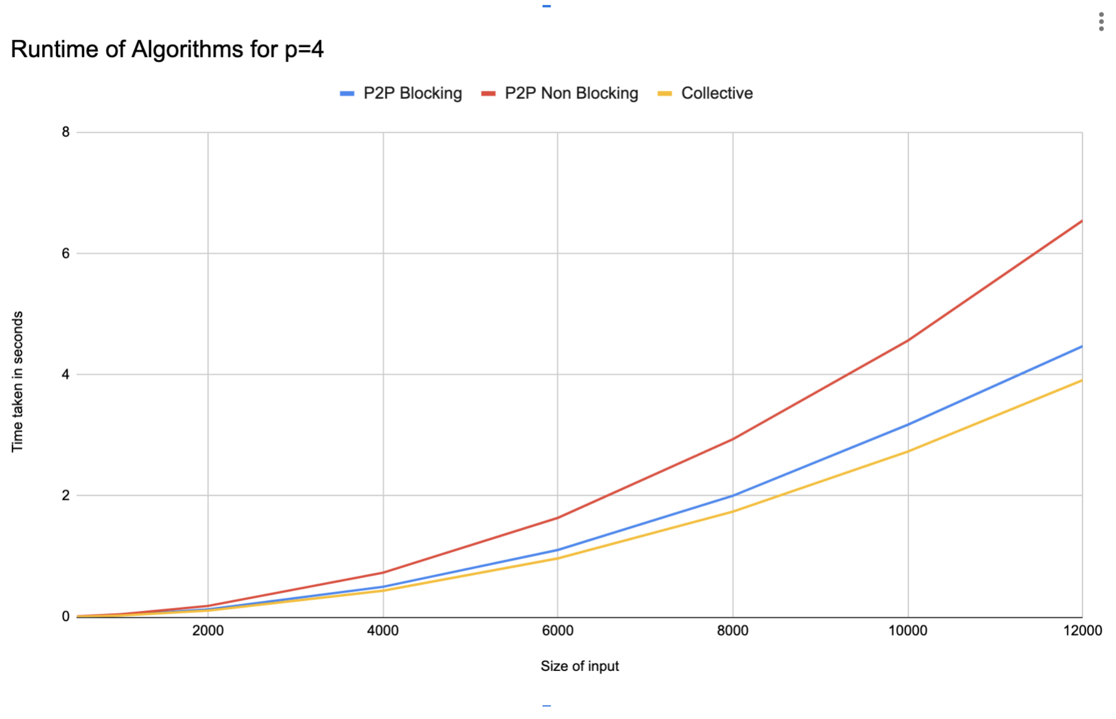

Figure 5: Algorithm Runtime at p=4

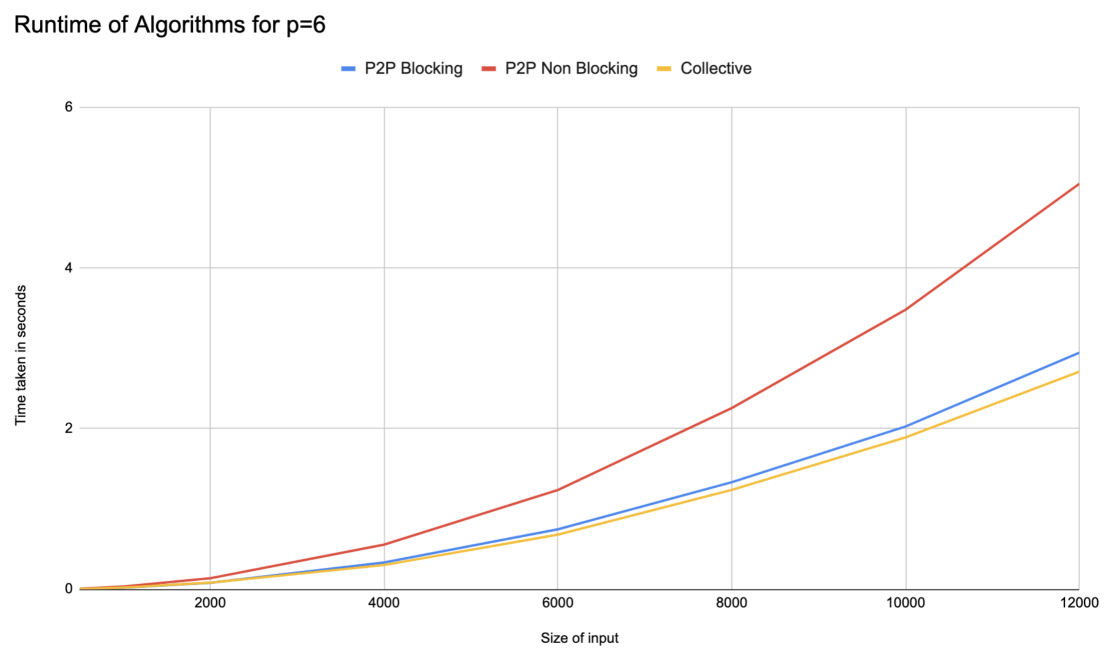

Figure 6: Algorithm Runtime at p=6

Appendix

The following is the machine information on which the execution run times were calculated
Machine 1:
model name : Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz
Ram : 16GB
Number of cores : 4
Machine 2:
model name : MacBook Pro i7th 9th Gen CPU @ 3GHz
Ram : 16GB
Number of cores : 6