DD2356 Project - Distributed Matrix-Matrix Multiply with Fox Algorithm

Nuo Chen

May 31, 2020

1 Introduction

Matrix multiplication is an essential operation in scientific computations. A straightforward implementation works in most cases, but in HPC field, this vanilla implementation is too slow and infeasible. The computation complexity of multiplying a matrix of size NxC with another matrix of size CxM is $O(NCM)$ or $O(N^3)$ if both are square matrices. To speed up the computation, the work can be distributed across multiple CPU threads or processes to compute the sub-product in parallel, and add them up at the end. By utilizing the decomposition and distributed computing, multiplication of large matrices will be feasible in reasonable time.

In this project, I have implemented the Fox algorithm for distributed matrix multiplication using MPI and OpenMP, investigated the performance of the implementation and compared it to the serial implementation.

2 Methodology

The Fox algorithm is an algorithm for distributing multiplication of square matrices (MxM) across a number of processes P. The matrices A and B are divided into P tiles each, which are distributed to the processes. Every process holds a tile of A and B. There are 4 steps in the Fox algorithm [1]:

1. The processes that hold the diagonal tile of A broadcast it to other processes on the same row.
2. Each process multiplies its (received) A tile with the B tile.
3. Each process sends its B tile to the process that is on the row above, and receives a new B tile from the process below.
4. Repeat from step 1 but with "diagonal + i" tile of A, where i = current iteration (starts from 0).

These 4 steps are repeated for $sqrt(P)$ iterations.

2.1 Implementation

The Fox algorithm is implemented in C with MPI and OpenMP. The implementation is described in the pseudo-code below:

Algorithm 1 Fox algorithm - initialization

```
Initialize MPI threads with MPI_THREAD_FUNNELED
// The code below is executed on every process

Get world rank and n_proc
tile size = matrix size / sqrt(n_proc)
row = rank / sqrt(n_proc)    // the row index of this process
col = rank % (matrix size / tile size) // the column index

row_comm = MPI_Comm_split(separator=row, identifier=col)
col_comm = MPI_Comm_split(separator=col, identifier=row)

Allocate space for A_buffer, B_buffer, C_buffer, T_buffer
if (rank == root)
    Allocate space for matrices A and B of size mat_size
    Fill the matrices with random values
    Partition the matrices into tiles(A_tiles, B_tiles)
    for i = 1 to n_proc
        MPI_Send(A_tiles[i], n=tile_size^2, dest=i)
        MPI_Send(B_tiles[i], n=tile_size^2, dest=i)
    copy(A_buffer, A_tiles[0])
    copy(B_buffer, B_tiles[0])
else
    MPI_Recv(A_buffer, n=tile_size^2, from=0)
```

In Algorithm 1, MPI is initialized with the keyword MPI_THREAD_FUNNELED to allow each process to be executed in multiple threads. The row and column index are computed for each process, which are used to split the communicator into separate row and column communicators. The root process initializes the matrices A and B with random numbers. The matrices are divided into tiles and distributed to other processes.

The computation part is described in Algorithm 2. The col_index specifies which processes will broadcast its A tile horizontally, and other processes receive and store the A tile in T_buffer. All processes multiply its A tile with the B tile and update the C_buffer with the result. All processes then send its B tile to the process above in the column communicator, and receive a new B tile.

When finished computing, all processes except the root, will send its C tile to the root process, where the tiles are combined back to the C matrix.

The matrix multiplication algorithm, as described in Algorithm 4, utilizes loop unrolling transformation and OpenMP. By using every A[i][j] and B[i][j]

Algorithm 2 Fox algorithm - multiplication

```
for ite = 0 to sqrt(n_proc)
    // Broadcast A tile and multiply
    col_index = (row+ite) % sqrt(n_proc)
    if (col == col_index)
        MPI_Bcast(A_buffer, n=tile_size^2, root=col, row_comm)
        matmul(A_buffer, B_buffer, C_buffer)
    else
        MPI_Bcast(T_buffer, n=tile_size^2, root=col_index, row_comm)
        matmul(T_buffer, B_buffer, C_buffer)

    // Roll B tile upwards
    dest = (row - 1 < 0)? sqrt(n_proc)-1 : row -1
    MPI_Sendrecv(sendFrom=B_buffer, dest, recvTo=T_buffer, col_comm)
    swap(B_buffer, T_buffer)
```

Algorithm 3 Fox algorithm - collect

```
if (rank == root)
    Allocate space for C_tiles[][][][]
    for i = 1 to n_proc
        MPI_Recv(C_tiles[i], from=i)
    copy(C_tiles[0], C_buffer)
    combine the tiles into the matrix C
else
    MPI_Send(C_buffer, n=tile_size^2, dest=root)
```

Algorithm 4 Matmul

```
void matmul(A,B,C)
    #pragma omp parallel for schedule(auto)
    for (int i = 0; i < size; i+=2)
        for (int j = 0; j < size; j+=2)
            for (int k = 0; k < size; k++)
                C[i][j] += A[i][k] * B[k][j];
                C[i+1][j] += A[i+1][k] * B[k][j];
                C[i][j+1] += A[i][k] * B[k][j+1];
                C[i+1][j+1] += A[i+1][k] * B[k][j+1];
```

twice, the number of loads (from memory) are cut in half, since these are stored in the registers after the first time. OpenMP is used to parallelize the for loops.

2.2 Correctness verification

To verify the correctness of the implementation, two test functions are implemented. Both functions take two C matrices, one that is computed using Fox algorithm and another that is computed directly by multiplying A with B.

The first function compares the differences of sums of each row of both matrices. The output should be 0 for every row.

The second function compares the differences of sums of each tile of both matrices. The output should be 0 for every tile.

3 Experimental setup

Hardware info

- Platform: Super computer - Beskow
- CPU: Intel Xeon E5-2698v3 Haswell 2.3 GHz CPU (16 cores)
- Memory: 64 GB

Software info

- Compiler: GCC compiler
- Compiler flag: -lm -fopenmp
- Packages: MPI, OpenMP

4 Results [B]

Below are the results of running the program with the following parameters:

Number of processes	1	4	16	64	256
Dimension of matrix	256	512	1024	2048	

For a matrix of dimension 256, it will have 256x256 elements of type unsigned long (8 bytes), the total size of this matrix will be 512KB.

4.1 Computation time comparison

Figure 1 compares the time it takes to multiply the matrices. The computation time of multiplying A with B sequentially is proportional to the size of the matrices. The computation time is decreased proportionally to the number of processes, but for small matrices (<512) the overhead of management and communication of processes are larger than the time savings.

Figure 2 compares the total time of allocating matrices, multiplying them and creating the result matrix. It also includes the time of initializing and managing the MPI processes. The results are similar to Figure 1, except that the overhead

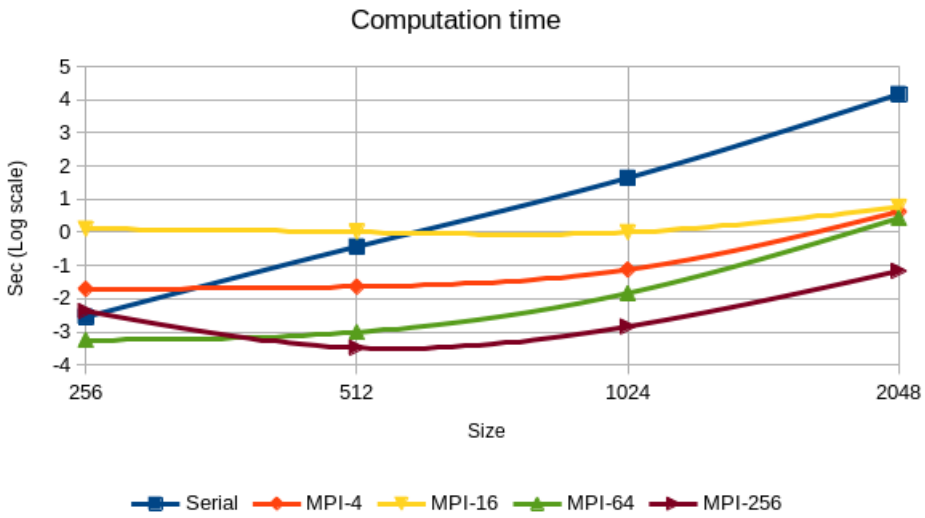

Figure 1: Comparing the matrix multiplication time only

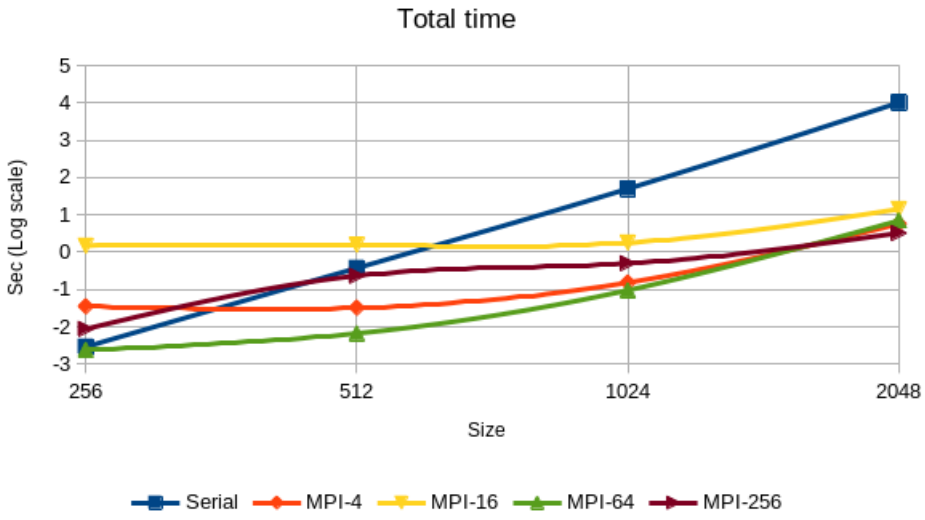

Figure 2: Comparing the total time of initializing, computing and collecting

of creating and managing 256 MPI processes is now larger than the time saving for matrices of dimension up to 1024.

Figure 3, 4 and 5 show the speed up of Fox algorithm with MPI processes. The X axis is the number of processes used and the series are dimension of the matrices. From Figure 3 and 4, it is clear that the speed ups are not linearly

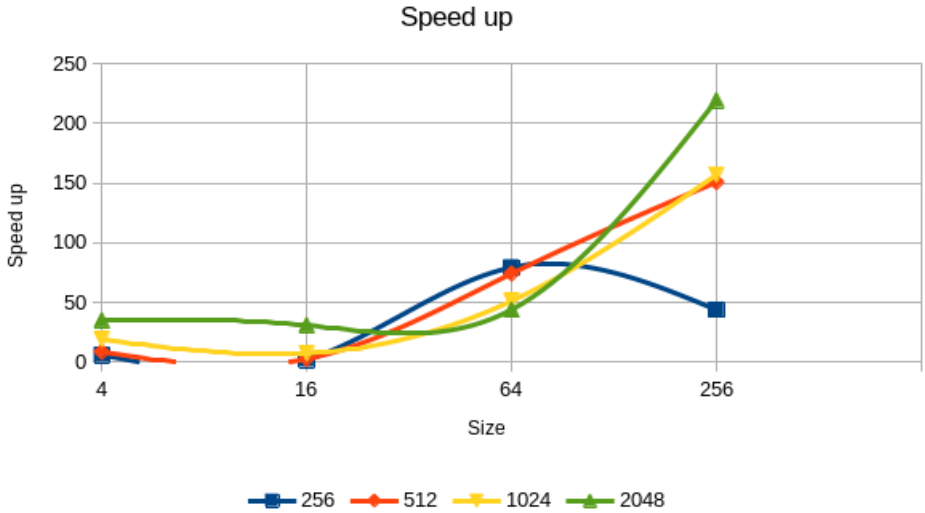

Figure 3: Speed up of using MPI processes compared to the serial solution

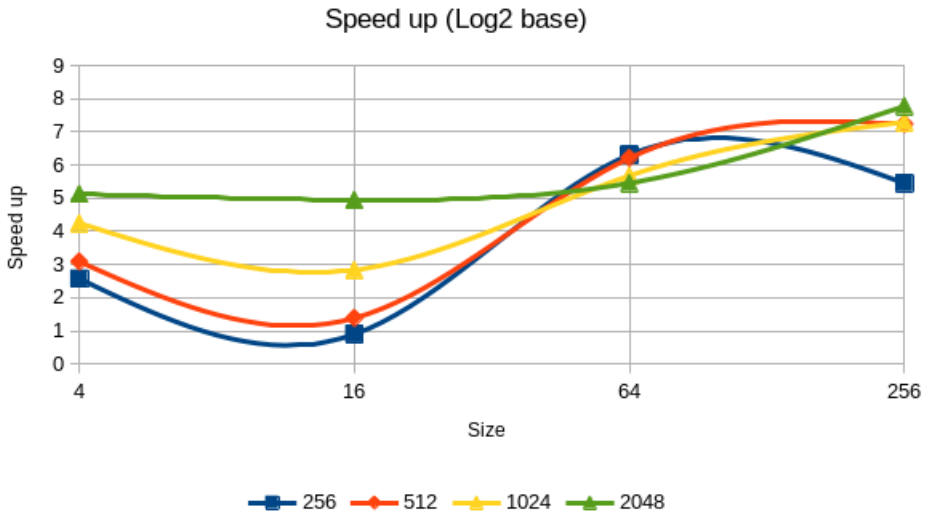

Figure 4: Log2 scale

proportional to the number of processes. For larger matrices ($\geq$ 512), the performance scales well with the increased number of processes. Figure 5 shows the gained speed up per incresed process is around 1 after 16 processes.

Figure 6 compares the matrix multiplication with and without OpenMP parallelization with matrix dimension 2048. The benefit of parallelizing for loops

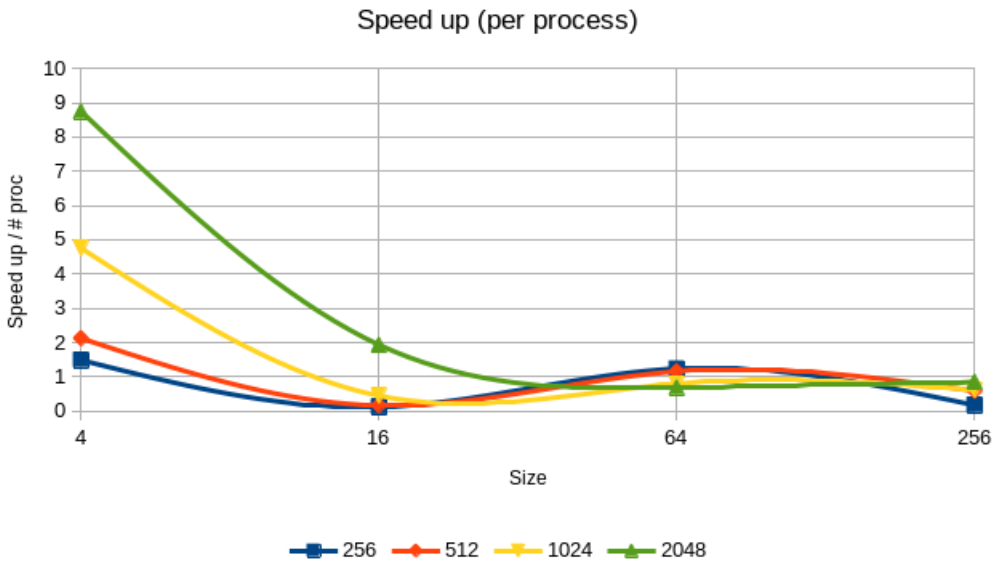

Figure 5: Speed up per process

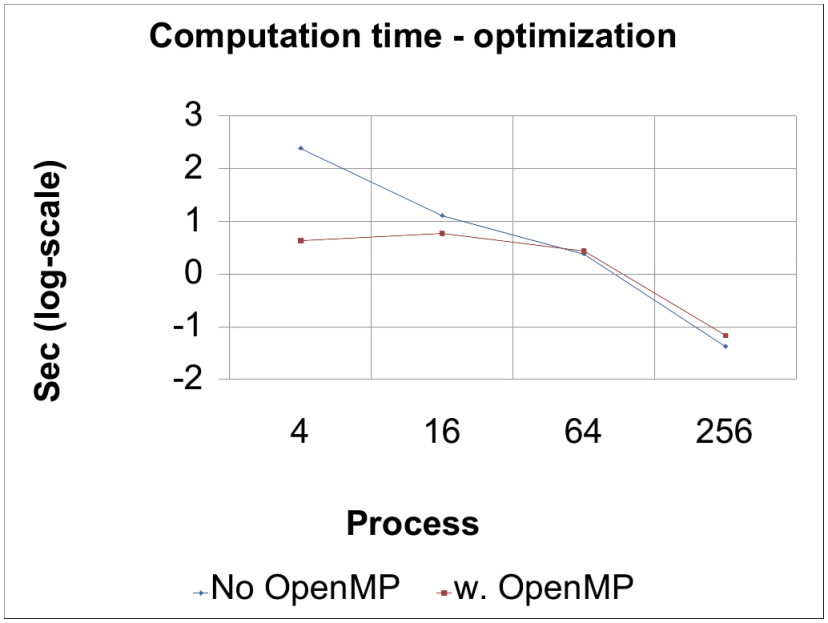

Figure 6: Optimization results

is most significant when running with fewer processes, and the benefit starts

to diminish as the number of processes increases. When using 256 processes, performing the matrix multiplication sequentially is actually faster than with parallelized for loops.

5 Discussion [B]

From the results we can clearly state that it is beneficial to use fox algorithm for large matrices. The overhead of creation, communication and management of processes are fixed cost and negligible, when the program need to perform matrix multiplication multiple times.

A noteworthy detail is that the experiments with 64 and lower processes are performed on one single computing node on the Beskow machine. The node is equipped with 2 Intel Xeon E5-2698v3 which have 16 physical cores each. Running with 64 processes is possible on a single node with hyper-threading, but the performance is inherently worse than running on separate nodes. However, for these experiments, it might not be any notable difference due to the small matrices and the overhead of communication between nodes.

The experiment with 256 processes is performed on 4 nodes, with 64 processes on each node. The performance is decreased for matrices of dimension 256, because the time is mostly spent on sending and receiving tiles between processes. When multiplying matrices of dimension 2048, the speed up (220x) is close to the ideal speed up (256x). We might get better speed up if we use even larger matrices, such as 8192 or 16384, but then one matrix will take 512 MB and 2 GB of space.

The benefit of using OpenMP to optimize the for loops inside each process is decreasing as the number of processes increases, as shown in Figure 6. One possible explanation is that, on one computing node, all available threads are already occupied by the processes, and for each process more threads are created to parallelize the work. In the end, all these threads are not executing in parallel but concurrently. Moreover, for large number of processes, the tile sizes are comparatively small and not worth the overhead of using OpenMP.

6 Conclusion

In the project, we have implemented the Fox algorithm in C with MPI and OpenMP, and experimented with different matrix sizes and number of processes. The results indicate that if multiplying small matrices (256x256), there is no benefit of parallelizing the multiplication. For larger matrices ($>$ 512x512), the performance scales proportionally to the number of processes.

References

[1] G.C Fox, S.W Otto, and A.J.G Hey. *Matrix algorithms on a hypercube I: Matrix multiplication.* 1987.