Distributed Matrix-Matrix Multiply with Fox Algorithm *

Gibson Chikafa
chikafa@kth.se

Akhil Yerrapragada
akhily@kth.se

June 8, 2020

1 Introduction

Matrix multiplication is one of the basic matrix computation problems. It is at the heart of many areas of mathematics (e.g linear algebra, numerical analysis) and many important scientific computations in applied mathematics, statistics, physics, economics, and engineering. As a result, there has been an intensive development and study of different algorithms and their complexities for matrix multiplication. Matrix multiplication is highly time-consuming and gives a good opportunity to demonstrate wide range of parallel methods and techniques. Some popular parallel implementations of matrix multiplication are according to the Cannon Method [1], Fox[3] Algorithm. In this exercise, we will develop, analyze the performance, and optimize a distributed-memory version of matrix-matrix multiply Fox Algorithm in C using MPI.

1.1 Matrix Multiplication Problem

Matrix multiplication is a binary operation that produces a matrix from two matrices. Multiplying the matrix A of size $m \times n$ by the matrix B of size $n \times l$ leads to obtaining the matrix C of size $m \times l$ with each matrix C element defined according to the expression:

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik}.b_{kj}, 0 \leq i < m, 0 \leq j < l \tag{1}$$

The number of columns in the first matrix must be equal to the number of rows in the second matrix. As seen in (1) the entry c_{ij} of the product is obtained by multiplying term-by-term the entries of the ith row of A and the jth column of B, and summing these n products. This is further illustrated in the Figure 1.

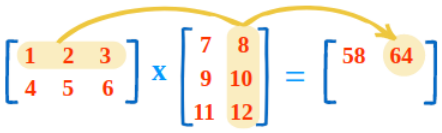

Figure 1: The Element of the Result Matrix C is the Result of the Scalar Multiplication of the Corresponding Matrix A Row of the Matrix A and the Column of the Matrix B. *Image Source* [2]

*Project Report for Methods in High Performnce Computing(DD2356) at KTH Royal Institute of Technology

The pseudo-code for serial implementation of the matrix-vector multiplication may look as follows (hereafter we assume that the matrices participating in multiplication are square, i.e. are of size $Size \times Size$):

```
// Serial algorithm of matrix multiplication
for (i=0; i<Size; i++) {
  for (j=0; j<Size; j++) {
    MatrixC[i][j] = 0;
    for (k=0; k<Size; k++) {
      MatrixC[i][j] = MatrixC[i][j] + MatrixA[i][k]*MatrixB[k][j];
    }
  }
}
```

The serial algorithm is fully implemented and its performance compared with the implementation of the Fox Algorithm.

1.2 Matrix Decomposition: Checkerboard Block Decomposition

A matrix can be decomposed into subsections called blocks or submatrices. Any matrix may be interpreted as a block matrix in one or more ways, with each interpretation defined by how its rows and columns are partitioned. For example, the matrix

$$\mathbf{P} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 4 & 3 & 2 & 1 \\ 1 & 2 & 3 & 4 \end{bmatrix}$$

can be partitioned into four 2×2 blocks

$$P_{11} = \begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix} P_{12} = \begin{bmatrix} 3 & 4 \\ 7 & 8 \end{bmatrix} P_{21} = \begin{bmatrix} 4 & 3 \\ 1 & 2 \end{bmatrix} P_{22} = \begin{bmatrix} 2 & 1 \\ 3 & 4 \end{bmatrix}$$

It is possible to use a block partitioned matrix product that involves only algebra on submatrices of the factors. The partitioning of the matrices is not arbitrary, it requires that the two matrices A and B are partitioned compatibly into blocks. Given an $m \times p$ matrix $\mathbf{A}$ with q row partitions and q column partitions

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} & \cdots & \mathbf{A}_{1s} \\ \mathbf{A}_{21} & \mathbf{A}_{22} & \cdots & \mathbf{A}_{2s} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{A}_{q1} & \mathbf{A}_{q2} & \cdots & \mathbf{A}_{qs} \end{bmatrix}$$

and a $p \times n$ matrix $\mathbf{B}$ with s row partitions and r column partitions

$$\mathbf{B} = \begin{bmatrix} \mathbf{B}_{11} & \mathbf{B}_{12} & \cdots & \mathbf{B}_{1r} \\ \mathbf{B}_{21} & \mathbf{B}_{22} & \cdots & \mathbf{B}_{2r} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{B}_{s1} & \mathbf{B}_{s2} & \cdots & \mathbf{B}_{sr} \end{bmatrix}$$

If the partitions are compatible with the partitions of A, the matrix product

$$\mathbf{C} = \mathbf{AB}$$

can be formed blockwise, yielding **C** as an $m \times n$ matrix with q row partitions and r column partitions. The matrices in the resulting matrix **C** are calculated by multiplying:

$$\mathbf{C}_{qr} = \sum_{i=1}^{s} \mathbf{A}_{qi} \mathbf{B}_{ir}.$$

This technique is used for faster matrix multiplication as we we will see in the next subsection.

1.3 Parallel Matrix Multiplication from Decomposition

Decomposition described in the previous subsection allows us to parallelize the matrix multiplication operation. We can define the basic computational subtasks on the basis of the computations performed over the matrix blocks. With regards to this we define the basic subtask as the procedure of computing of the elements of a block of the matrix C.

In order to do all the required computations the basic subtasks should have the corresponding sets of rows of the matrix A and columns of the matrix B. The major issue is the allocation of necessary data in each subtask. If all the necessary data is allocated at once to each subtask it will lead to data duplication as well as 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. As a result, the problem to parallelize matrix operations can be reduced in most cases to matrix data distribution among the processors of the computer system. One of the possible approaches is the Fox algorithm discussed in the next subsection.

1.4 Fox Algorithm

The Fox algorithm is a method of efficiently distributing data according to the Checkerboard Block Decomposition to processes involved in the matrix multiplication. The goal

is to do the matrix multiplication as described in Subsection 1.2 but in parallel. In a parallel matrix multiplication method based on the checkerboard decomposition scheme it is required that the basic subtasks are responsible for computing the separate blocks of the matrix C. It is also required that each subtask should hold only one block of the multiplying matrices at each iteration.

Suppose that matrix A dimension is $m \times k$, and matrix B dimension is $k \times n$. Let p be the number of processes and $q = \sqrt{p}$ be an integer such that it divides m and n. We Create a Cartesian topology with process mesh P_{ij}, and $i = 0..q-1, j = 0..q-1$. Let $\hat{m} = \frac{m}{q}$, $\hat{k} = \frac{k}{q}$ and $\hat{n} = \frac{n}{q}$. We partition A into $\hat{m} \times \hat{k}$ blocks and B into $\hat{n} \times \hat{k}$ blocks. According to [3], algorithm proceeds as follows:

1. Broadcast (in a pipelined fashion) the diagonal subblocks of A in a horizontal direction, so that all processors in the first 'row' have a copy A_{00}, all processors in second row have a copy of A_{11} and so on.
2. Multiply the copied **A** subblocks into the **B** subblocks currently residing in each processor.
3. 'Roll' the **B** subblocks vertically.
4. Do a horizontal broadcast (as in step 1) of the 'diagonal + 1' subblocks of **A**
5. Multiply the copied A subblocks into the currently residing **B** subblocks.

Continue this pattern until **B** has 'rolled' completely around the machine.

2 Methodology

We first present the implementation of the Fox Algorithm using MPI where the multiplication of block the matrices on processes is done using the naive approach. In the next Subsection 2.8 we will discuss the implementation where we improve the implementation block multiplication by utilizing temporal and spatial locality.

2.1 Implementation of the Fox Algorithm using MPI

In our implementation we assume that the number of processes p is a perfect square i.e 4, 9, 16, 15 e.t.c. Let $q = \sqrt{p}$. If A dimension is $m \times k$, and matrix B dimension is $k \times n$, the sizes of the matrices should be such that q divides m, k and n so that we are able to create a compatible partitions and the blocks of the partition are of the same size. By having all the partitions of the same size we have equal load on all the processes in the grid. For simplicity we will assume that both A and B are square matrices of the same size.

We form a square Cartesian grid of $q \times q$. Since the data communications consist of block transmission along rows and columns of the subtask grid, the network topology should be also a square grid[3]. The Cartesian grid corresponds to the structure of the checkerboard block decomposition of the matrix C. We enumerate the processes using the indices of the blocks C_{ij} such that process P_{ij} computes the block C_{ij}. In a ccordance with the Fox algorithm each basic process (i,j) holds four matrix blocks:

1. Block C_{ij} of matrix C, computed by the process;
2. Block A_{ij} of matrix A, placed in the subtask before the beginning of computations;
3. Blocks A'_{ij}, B'_{ij} of matrices A and B, obtained by the subtask in the course of computations.

In the initialization stage each process P_{ij} obtains blocks A_{ij} , B_{ij}. All elements of blocks in **C** on all processes are set to zero. In iterarion l, $0 \leq l < q$, the A_{ij} is transmitted to all processes of the same processor grid row. The index j, which defines the position of the block in the row, is computed according to the following expression:

$$j = (i + l) mod q$$

2.2 The main method

The main function implements the computational method scheme by sequential calling out the necessary subprograms. We declare the necessary variables in the main method such as matrices A, B, and C on the process. We also initialize MPI in the main method.

2.3 Creating the Grid Communicators

The creation of grid communicators is implemented in the function `CreateGridCommunicators`. This function creates a communicator as a two-dimensional square grid, determines the coordinates of each process in the grid and creates communicators for each row and each column separately.

The grid is created by the function `MPI_Cart_create` (the vector Periodic defines the permissibility of data communications among the bordering processes of the grid columns and rows). After the grid has been created, each parallel program process will have it s coordinates in the grid. The coordinates may be obtained by means of the function `MPI_Cart_coords`. Then in addition to the grid topology a set of communicators for each grid column and row separately is created by the function `MPI_ Cart_sub`.

2.4 Process Initialization

Initialization of the processes is performed in the function `Process Initialization`. This function sets the matrix sizes and allocates memory for storing the initial matrices

and their blocks, initial izes all the original problem data. In order to determine the elements of the initial matrices we will use the functions `DummyDataInitialization` and `RandomDataInitialization`.

2.5 Block-Block Matrix Multiplication

The function `ParallelResultCalculation` executes the parallel Fox algorithm of matrix multiplication. The matrix blocks and their sizes must be given to the function as its arguments. From the Fox algorithm, `GridSize` iterations are performed in order to execute matrix multiplication. Each of the iterations consists of the execution of the following operations:

- The broadcast of the matrix A block along the processor grid row (to execute the step we should develop the function `ABlockCommunication`),
- The multiplication of matrix blocks (to carry out the multiplication of matrix blocks we may use the function `SerialResultCalculation`) , which is the serial implementation of the matrix multiply program),
- The cyclic shift of the matrix B blocks along the column of the processor grid (the function `BlockCommunication`).

2.6 Broadcasting Blocks of A

The function `AblockCommunication` broadcasts matrix A blocks to the process grid rows. The leading process `Pivot` that is responsible for sending is chosen in each row of the grid. The number of the process `Pivot` in the row is determined according to the following expression: $Pivot = (i + iter) mod GridSize$ where i is the number of the processor grid row, for which we determine the number of the broadcasting process. We created an additional block for A: the first block `pMatrixAblock` stores the block, which was located on this process before the beginning of the computations, the block `pAblock` stores the matrix block, which participates in multiplication at this algorithm iteration. Before broadcasting the block `pMatrixAblock` is copied to the array `pAblock`, and then the array `pAblock` is broadcast to the row processes. The required communications are executed by means of the function `MPI_Bcast`. It should be noted that the operation is collective, and its localization in separate process grid rows is provided by the communicators `RowComm`, which are created for the set of processes of each row separately.

2.7 Distributing Blocks of B

The function `BblockCommunication` performs the cyclic shift of blocks of the matrix B in the process grid columns. Each process transmits its block to the upper neighboring process `NextProc` in the process column and receives the block transmitted from the process `PrevProc`, which stands below it in the grid column. Data transmission is executed by means of the function `MPI_SendRecv_replace`, which provides all the necessary block

transmissions using the same memory buffer `pBblock`. Besides, this function prevents possible deadlocks, which happen when data transmission begins to be performed simultaneously by several processes in the ring network topology.

2.8 Improving the Block matrix multiplication on process

There is an interesting technique called blocking that can improve the temporal locality of inner loops. The general idea of blocking is to organize the data structures in a program into large chunks called blocks. The program is structured so that it loads a chunk into the L1 cache, does all the reads and writes that it needs to on that chunk, then discards the chunk, loads in the next chunk, and so on. In the naive approach when a chunk from B is loaded into the L1 cache, only one value is used and then the whole cache line is discarded. This is therefore not cache friendly implementation. To improve on this we would like to partition `pAMatrix` and `pCMatrix` into $1 \times BlockSize$ *row blocks* and to partition `pBMatrix` into $BlockSize \times BlockSize$ blocks. The $BlockSize$ is chosen with respect to the cache size on the machine so that our blocks can fit directly in the cache. When do blocking and reorder our loops references to `pAMatrix` enjoy good spatial locality because each loaded block in cache is accessed with a stride of 1. There is also good temporal locality because the entire block is referenced `pAMatrix` times in succession. References to `pBMatrix` enjoy good temporal locality because the entire $BlockSize \times BlockSize$ block is accessed $BlockSize$ times in succession. Finally, the references to `pCMatrix` have good spatial locality because each element of the block is written in succession.

We obtain our $BlockSize$ from the following expression: $sizeof L1cache/sizeof(double)$. The `size of L1 cache` is obtained by including the command:
`-DCLS=$(getconf LEVEL1_DCACHE_LINESIZE)` during compilation.

3 Experimental SetUp

3.1 Software and Hardware

The experiment was conducted on Beskow supercomputer. Beskow is a Cray XC series super computer running on Linux operating system. The Cray XC series is a distributed memory system developed as part of Cray's participation in the Defense Advanced Research Projects Agency's (DARPA) High Productivity Computing System (HPCS) program. The Cray-developed Aries interconnect provides high network performance and programming ability. Beskow has two chips. First it contains 2 Haswell processors per node with 22nm technology, 16 cores and 2.30–3.60 GH z clock speed. The caches is range as follows: L1–64KB per core, L2–256KB per core and L3 24MB shared. It contains 4 memory channels with a max bandwidth of 68 GB/s. Second contains 2 Broadwell processors per node with 14nm technology, 18 cores and 2.10–3.30 GHz clock speed. The cache range is as follows : L1–64Kb per core, L2–256Kb per core, L3–27Mb shared. It contains 4 memory channels with max bandwidth of 76.8 Gb/s. The network topology used by beskow

is Dragonfly. It provides scalable global bandwidth while minimizing the number of expensive optical links. Dragonfly is a direct network, meaning it reduces the optical links required for global bandwidth. It is a low diameter network constructed from high radix routers. Aries provides a system-on-a-chip design that combines four high performance network interface controllers (NIC) and a high radix router. Low-cost electrical links are used to connect the NICs in each node to their local router and the routers in a group. The network protocol can transfer 8–64 byte messages with low overhead.

We use the gcc compiler and the MPI version used is 3.0.

3.2 Running the code and Input Matrices

As already mentioned, their are two implementations: MPI with naive matrix multiply in **fox.c** and MPI with optimized matrix multiply in **fox_optimized.c**. To compile **fox.c** on Beskow using the gcc compiler use the following command: `cc fox.c -o fox.out`. To run the code use the following command: `srun -n <numOfProcess> ./fox.out <matrixSize>`. Remember that the number of processes must be a perfect square and the root must perfectly divide the matrix size provided. To compile the **fox_optimized.c** use the following command: `mpicc -DCLS=$(getconf LEVEL1_DCACHE_LINESIZE) fox_optimized.c -o fox_optimized.out -lm`.

To simplify things, we implemented the function `DummyDataInitialization` that initializes matrices A and B with unit values i.e 1's. The input matrix size is obtained from the command line argument provided when running the code. The `-DCLS=$(getconf LEVEL1_DCACHE_LINESIZE)` allows us to get the L1 cache size.

3.3 Testing

The function `TestBlocks` prints the blocks of the input matrices. This allowed us to confirm that indeed the input matrices were being correctly partitioned. An output of the partitioned matrices in Figure 2.

```
gibson@gibson:~/Documents/GibsonKTH/HighPerformanceComputing/Assignments/FinalProject/Demos$ mpirun -np 4 fox_hybrid.out 8
Initial blocks of matrix A
ProcRank = 0
 1.0000  1.0000  1.0000  1.0000
 1.0000  1.0000  1.0000  1.0000
 1.0000  1.0000  1.0000  1.0000
 1.0000  1.0000  1.0000  1.0000
ProcRank = 1
 1.0000  1.0000  1.0000  1.0000
 1.0000  1.0000  1.0000  1.0000
 1.0000  1.0000  1.0000  1.0000
 1.0000  1.0000  1.0000  1.0000
ProcRank = 2
 1.0000  1.0000  1.0000  1.0000
 1.0000  1.0000  1.0000  1.0000
 1.0000  1.0000  1.0000  1.0000
 1.0000  1.0000  1.0000  1.0000
ProcRank = 3
 1.0000  1.0000  1.0000  1.0000
 1.0000  1.0000  1.0000  1.0000
 1.0000  1.0000  1.0000  1.0000
 1.0000  1.0000  1.0000  1.0000
gibson@gibson:~/Documents/GibsonKTH/HighPerformanceComputing/Assignments/FinalProject/Demos$
```

Figure 2: Initial blocks of the matrix A on each process. A is an 8 by 8 matrix. Since we are using 4 process, the blocks should by 4 by 4 each as shown in the figure.

The function `TestResult` is used to verify the result of multiplication of the input matrices. The function just calls the `SerialResultCalculation` function using the input matrices A and B. It is run only by the root process therefore the result from this procedure does not involve the MPI process. This result is stored in the variable `pSerialResult`. Also it receives the calculated result from the MPI processes in variable `pCMatrix`. We then compare `pCMatrix` with the `pSerialResult` by checking each element in the matrices using a loop.

4 Efficiency Analysis

In this section we will evaluate the effeciency analysis of the Fox algorithm. To formulate the required estimations we will suppose that all the previous assumptions are met, i.e. all matrices are square, their sizes are $n \times n$, the block grid is square and its size is equal to q (i.e. the size of all blocks is $k \times k, k = n/q$), processors form a square grid and their number is $p = q2$.

The complexity of scalar multiplication of the block row of the matrix A by the block column of the matrix B may be estimated as $2(n/q) - 1$. The number of rows and columns in the blocks is equal to n/q. As a result, the time complexity of block multiplication appears to be equal to $(n^2/p)(2n/q - 1)$. The addition of the blocks requires n^2/p operations. With regard to the above given expressions the computational time of the Fox algorithm may be estimated in the following way:

$$T_p(calc) = q[(n^2/p)(2n/q - 1) + (n^2/p)]\tau. \quad (2)$$

(as previously that τ is the execution time of an basic computational operation).

We will now estimate now the overhead on data co mmunications among the processors. One of the processors of the processor grid row transmits its matrix A block to the rest of the grid row processors at each iteration. The execution of this operation may be provided in $\log_2 q$ steps, if the topology of the network is a hypercube or a complete graph. As a result, the time complexity of data communications in accordance with the Hockney model may be estimated as follows:

$$T_p^1(comm) = \log_2 q(\alpha + \omega(n^2/p)/\beta) \quad (3)$$

where α is the latency, β is the network bandwidth, ω is the size of a matrix elements in bytes.

After multiplying the matrix blocks the processors send their matrix B blocks to the processors, wh ich are upper neighbors in the processor grid columns (the first row processors send their data to the last row processors). These operations may be carried out

by the processors in parallel and, thus the ti me complexity of these communication operations is the following:

$$T_p^2(comm) = \alpha + \omega(n^2 p)/\beta. \tag{4}$$

The total execution time for the Fox algorithm may be defined by means of the following relations:

$$T_p = T_p(calc) + T_p^1(comm) + T_p^2(comm). \tag{5}$$

Clock speed is 2.6GHZ on Beskow (Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz). We assume that the processor can do 2.6G operations per second. Therefore $\tau = 1/clockrate = 1/(2.6 * 109) = 3.85 * 10^{(-10)} s$. β(bandwidth) is $\approx$ 4.36 Gb/s and α(latency) $\approx$ 6.73 ms from assignment 3.

5 Results

The serial code, and both implementations of the Fox algorithm were run against with N = 100, 200, 400, 800, 1600, 3200, 4000 and 4800 matrices and varying number of processors.

		Parallel Fox					
		16 Processes		25 Processes		100 Processes	
MatrixSize	Serial Code	Model	Observed	Model	Observed	Model	Observed
100	0.004662	6.8978E-06	0.0019454	6.5823E-05	0.0101152	4.25689E-05	0.0142732
200	0.031737	0.00013796	0.0063958	0.000131646	0.0176524	8.51378E-05	0.0179986
400	0.268908	0.000275912	0.030236	0.000263292	0.027468	0.0001702756	0.031973
800	2.565009	0.000551824	0.1738786	0.000526584	0.2082438	0.0003405512	0.0807858
1600	33.641634	0.001103648	1.2217628	0.001053168	1.7457652	0.0006811024	0.4604908
3200	377.627652	0.002207296	13.0115506	0.002106336	13.8041432	0.0013622048	3.8240854
4000	931.759783	0.00275912	35.3018382	0.00263292	25.6659656	0.001702756	7.1149452
4800	1700.641634	0.003310944	82.6207928	0.003159504	51.7815708	0.0020433072	14.2924442

Figure 3: Table showing results for Serial Matrix Multiplication, Modelled time for fox algorithm and the actual observed time of the Fox algorithm (non-optimized)

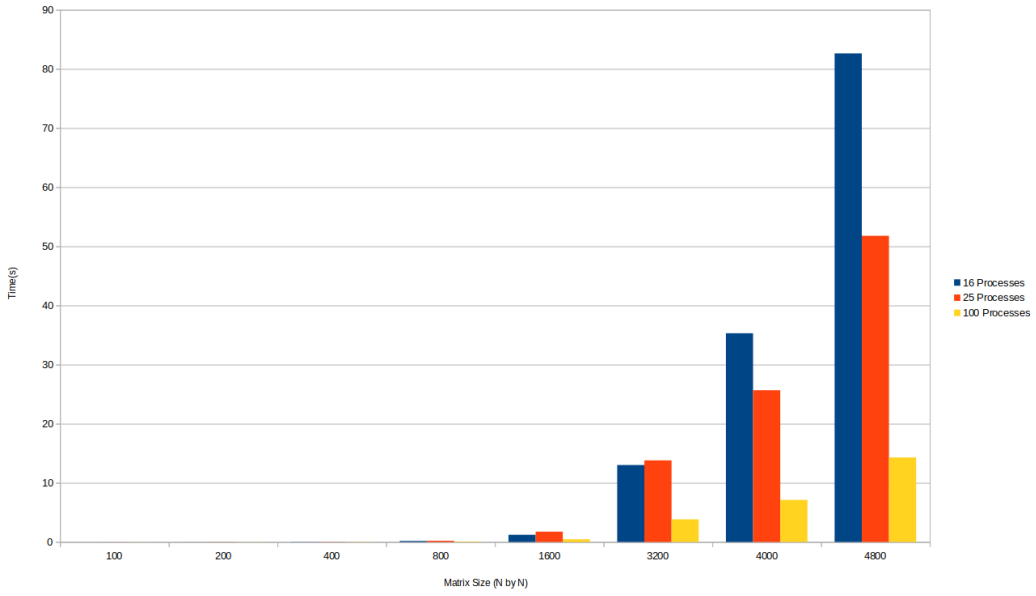

Figure 4: The execution time of N by N matrices run with 16, 25 and 100 processes

Strong scaling was performed for 4800×4800 input matrices varying the number of processes and the results are shown in the table below. The optimized Fox implementation was used.

Number of Processes	Time(s)
1	325.236805
64	15.2191886
100	9.1051652
256	4.673465
400	2.7603246

Figure 5: Execution time for 4800×4800 matrix using 1, 64, 100 and 400 processes

Table below shows results for naive and optimized implementations for different sizes of matrices.

Matrix Size	16 Proc (Naive)	16 Proc (Optimized)	25 Proc (Naive)	25 Proc (Optimized)	100 Proc (Naive)	100 Proc (Optimized)
100	0.0019454	0.0020486	0.0101152	0.0020832	0.0142732	0.0190248
200	0.0063958	0.0059918	0.0176524	0.0051182	0.0179986	0.0183676
400	0.030236	0.0355908	0.027468	0.0260232	0.031973	0.0380416
800	0.1738786	0.2062132	0.2082438	0.1730142	0.0807858	0.0918822
1600	1.2217628	1.0797232	1.7457652	1.2791708	0.4604908	0.4553226
3200	13.0115506	7.957964	13.8041432	6.5138718	3.8240854	2.8780112
4000	35.3018382	15.3327772	25.6659656	11.0189938	7.1149452	5.3386276
4800	82.6207928	26.2904714	51.7815708	18.4082448	14.2924442	9.1051652

Figure 6: Naive and optimized implementations for different sizes of matrices. The results for naive are the same as in Figure 3 and are shown here for comparison

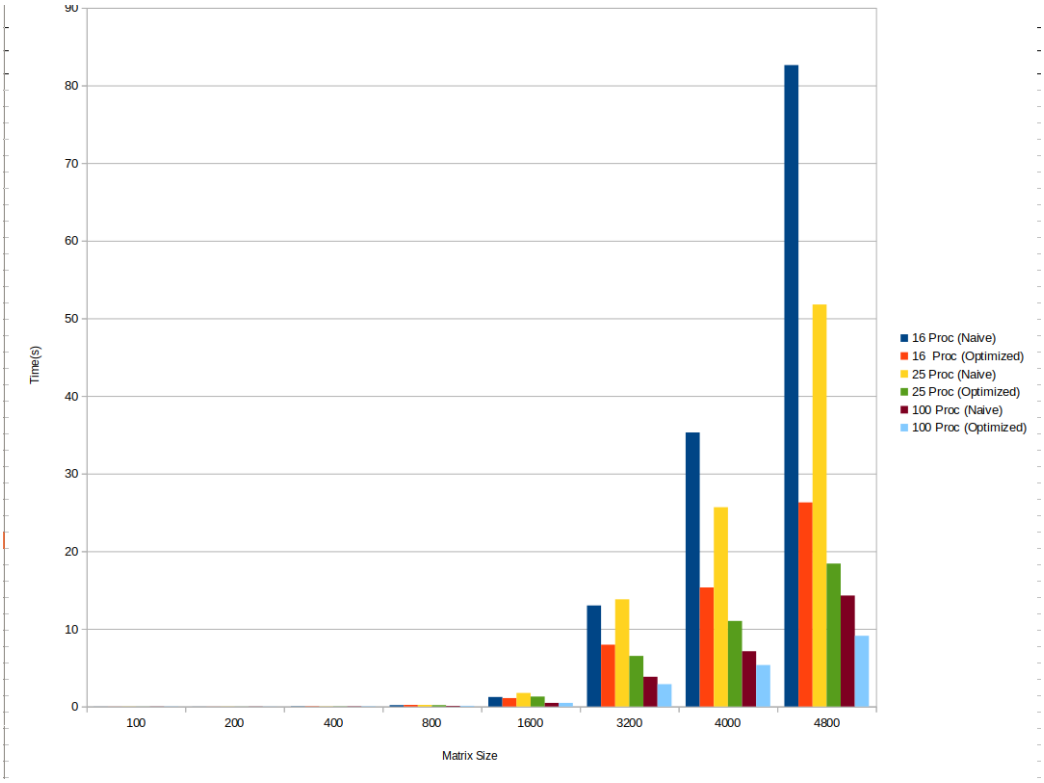

Figure 7: Graph showing Naive and optimized implementations performance for different sizes of matrices.

6 Discussion

6.1 Modelled vs Observed Results

As seen from Figure 3 the modelled time is much less than the observed time. This is due to several factors. The major reason is that the theoretical time does not include data access aspects such as data movement and cache misses. Also the time estimated for floating point operations is not realistic: this was observed in assignment 1 that the calculated time from CPU clock does not directly map to time to execute a single floating point operation. Furthermore, the bandwidth estimation used may be higher than the actual one.

6.2 Serial Code vs Parallel Code (non optimized)

As seen from Figure 3, there is a great improvement when we parallelize matrix multiplication using MPI. This is more evident when the size of the matrices becomes larger. For example, 4800 by 4800 matrix for serial code takes 1700 seconds while for parallel implementation it takes only 14s.

6.3 Increasing the number of processes

As seen from all parallel implementations, increasing the number of processes improves the performance of the Fox algorithm. This is more evident when the size if the matrices become larger as seen from the strong scaling results in Figure 5 where the matrix size is kept constant.

6.4 Naive vs Optimized

From the observed results, it can be seen that one of the major factors affecting performance is the memory access operations i.g cache misses. In optimized implementation we tried to leverage spatial and temporal locality to reduce cache misses and improve the performance, and the results agrees with the theory. The more the matrices become larger the more the Optimized outperform the naive implementation. For example, using 100 processes for 4800 by 4800 processes, the optimized implementation is 50 percent faster than the naive approach. For smaller number of processes e.g 16, the performance improves by over 100 percent.

6.5 More improvements

The code can be further optimized in different ways. For example, one can use a hybrid implementation of OpenMP and MPI. OpenMPI can be used to parallize the loops in the function that does the multiplication of the blocks. Furthermore, one can try to transponse the B matrix to reduce the cache misses of B access. However, We cannot tell which implementation will deliver better results as some of these methods depend on the machine architecture.

7 Conclusion

In this assignment, we have seen that the Fox algorithm is a powerful algorithm for improving the performance of matrix multiplication. The observed time for parallel implementation using MPI is much less than that for serial approach. It has also been observed that data reads and writes is big factor that affects performance. For the case of matrix multiplication, optimizing the cache access is a one nice approach to improve the performance.

References

[1] *Cannon's algorithm*, 2020 (accessed June 4, 2020). https://en.wikipedia.org/wiki/Cannon's_algorithm.

[2] *How to Multiply Matrices*, 2020 (accessed June 4, 2020). https://www.mathsisfun.com/algebra/matrix-multiplying.html.

[3] Fox G.C and Otto S.W. Matrix algorithms on a hypercube I:Matrix multiplication. *Parallel Computing*, 4:17–31, 1987.