Distributed-Memory Matrix Multiplication

CS-599 Final Project Report

Introduction

Matrix multiplication is an essential component in many mathematical problems, numerical algorithms, etc. There exists several matrix multiplication algorithms, but among those, the simplest, and most widely used, is the so called "Naïve" algorithm. The Naïve algorithm works fine for smaller matrix sizes, but struggle with high run times for larger sizes. One solution to this would be a parallelized version of the Naïve algorithm, that would split the calculation of the resulting matrix between multiple processes. This solution however suffers from very large memory requirements, as each process has to be supplied with copies of the two matrices being multiplied.

In this project, I will look at two so called "distributed-memory" matrix multiplication algorithms: The Cannon algorithm, and the Fox algorithm. These two algorithms work around the high memory requirements of the parallelized Naïve algorithm, by splitting up the two matrices between the processes, and having the processes communicate with each other in order to calculate the resulting matrix. I will implement both the Cannon and Fox algorithms from scratch, as well as a parallelized version of the Naïve algorithm, and then test the three algorithms, and compare them against each other to see how well the two distributed-memory algorithms compare to the Naïve algorithm.

Technical Approach

In this section, I will explain the theoretical implementations of the three algorithms, as well as my own implementation of them. I will also discuss the theoretical performance of the algorithms, and later compare this theoretical performance to the actual results.

The Naïve Algorithm

The Naïve algorithm is quite straight forward. We start with two matrices A[l][m] and B[m][n], and the resulting matrix $A \times B = C[l][n]$. If we assume that the resulting matrix C starts out being filled up with 0's, the Naïve algorithm works as described in the pseudocode below:

```
for (int i=0; i<n; i++) {
  for (int j=0; j<l; j++) {
    for (int k=0; k<m; k++) {
      matrixC[i][j] = matrixC[i][j] + (matrixA[i][k] * matrixB[k][j]);
    }
  }
}
```

Parallelization of this algorithm is also quite straightforward. We provide a copy of matrices A and B to each process, and then we have each process calculate a certain number of rows of matrix C, as illustrated in Figure 1 below, where four processes are assigned one row each on a 4×4 matrix C.

Figure 1

P0	C00	C01	C02	C03
P1	C10	C11	C12	C13
P2	C20	C21	C22	C23
P3	C30	C31	C32	C33

The main advantage about this implementation of the algorithm is its simplicity. As mentioned before however, the main disadvantage is the amount of memory required to store the matrices A, B, and C on all the processes. For large matrix sizes, this will become problematic, and thus we want to use alternative algorithms to try and work around this problem.

Cannon and Fox Algorithms

"Checkerboard" Decomposition

The first step in both the Cannon and Fox algorithms, is to divide the matrices A, B, and C between the processes. This is done in a "checkerboard" pattern, so that each process gets a $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ sized sub-matrix of the A, B, and C matrices, after which we arrange the processes into rows and columns on a "processes-matrix", as seen on Figure 2, where a 4×4 matrix A is split between four processes.

Figure 2

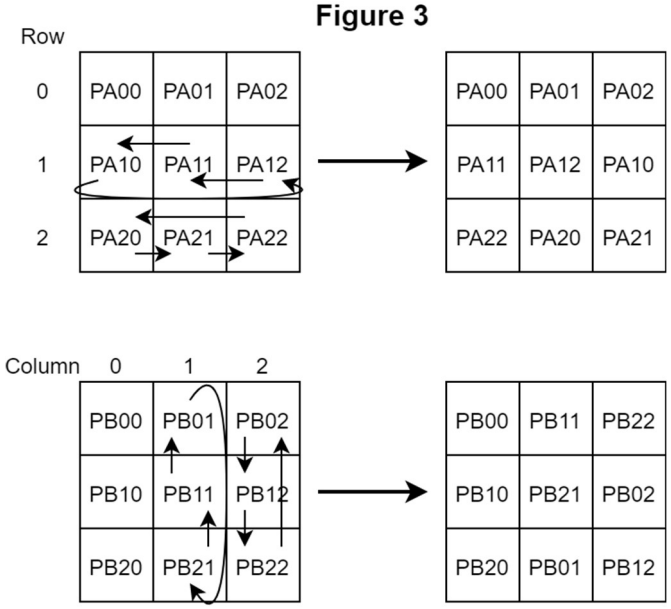
Some conditions have to be met in order for us to be able to split the matrices in this fashion. First of all, we must be able to arrange the processes in a perfect square, that is, for the number of processes p, it must be true that $\sqrt{p}$ is an integer. Second of all, matrices A, B, and C must be $n \times n$ matrices. And third of all, $\sqrt{p}$ must evenly divide n. The last two requirements can be worked around by adding rows and columns of 0's to matrices A, B, and C, such that they become $n \times n$ matrices, and such that $\sqrt{p}$ evenly divides n, but for all my tests, I have simply used $n \times n$ matrices of sizes n that were evenly divisible by $\sqrt{p}$.

In my implementation of the Cannon and Fox algorithms, we assume that the matrices A, B, and C have already been distributed across the processes. Thus, in my implementation of both algorithms, the sub matrices are simply generated on each process before the next steps are performed. Should we want to make the algorithms work with an input algorithm however, or something similar, it should be fairly simple to split the matrices up between the processes.

Cannon Algorithm

There are two more steps to the Cannon algorithm. The first of these is to perform initial alignment of the matrices A and B. This is done by each process shifting the sub-matrices of matrix A to the process i spots to the left on the processes-matrix, where i is the row number on the processes-matrix (starting from 0), and shifting the sub-matrices of matrix B to the process j spots up on the processes matrix, where j is the column number on the processes matrix (starting from 0). In both cases, if a sub-matrix is moves over the "edge" of the processes-matrix, it is instead moved back around on the opposite side. On Figure 3, we see how this is done on a 3×3 processes matrix.

Figure 3

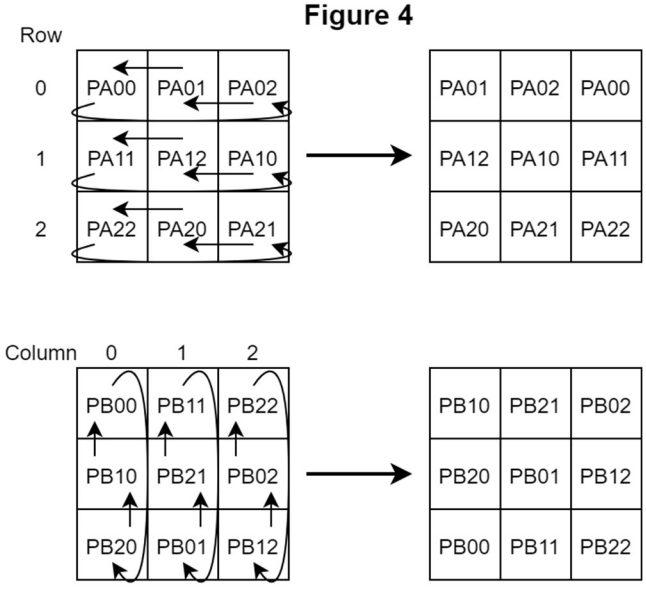


Once the initial alignment is finished, the final step of the Cannon algorithm is the actual matrix multiplication. The matrix multiplication is performed in $\sqrt{p}$ phases. In each phase, the processes update the value of their local matrix C, using their local matrices A and B, as shown in the below pseudocode:

```
int local_matrix_size = n/sqrt(p);
for (int i=0; i<local_matrix_size; i++) {
  for (int j=0; j<local_matrix_size; j++) {
    for (int k=0; k<local_matrix_size; k++) {
      local_matrixC[i][j] = local_matrixC[i][j] + (local_matrixA[i][k] * local_matrixB[k][j]);
    }
  }
}
```

Once this is done, each process shifts their local matrix A one spot to the left on the processes-matrix, and their local matrix B one spot up on the processes matrix, as shown on Figure 4.

Figure 4



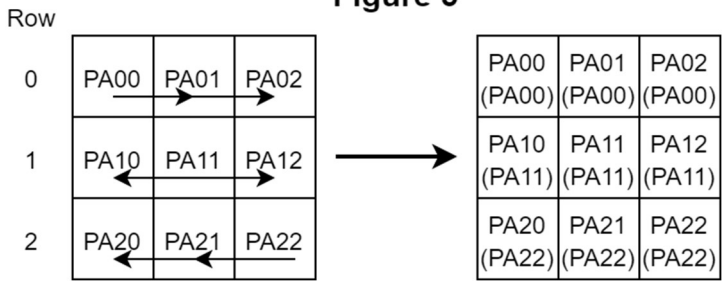
These steps are performed $\sqrt{p}$ times, after which the algorithm is finished.

The biggest advantage with the Cannon algorithm is that it requires the same amount of memory, no matter how many processes are used, as the matrices are split up evenly between the processes. The downside however is that it does not scale very well, as a large amount of communication between the processes is required to pass the sub-matrices around, especially for larger numbers of processes.

Fox Algorithm

After splitting up the matrices between the processes, the second and last step of the Fox algorithm is the actual matrix multiplication. As was the case for the cannon algorithm, this step is performed in $\sqrt{p}$ phases. In each of these phases, the algorithm does three things: Firstly, process nr $P_{(i,i+k)}$, where i is the process' row number, and k is the current phase (starting from 0), broadcasts its local matrix A to all other processes on the same row on the process matrix. This is shown in Figure 5, where process $P_{(i,i+0)} = P_{(i,i)}$ is broadcasting its local matrix A to all other processes on the row, during phase 0.

Figure 5

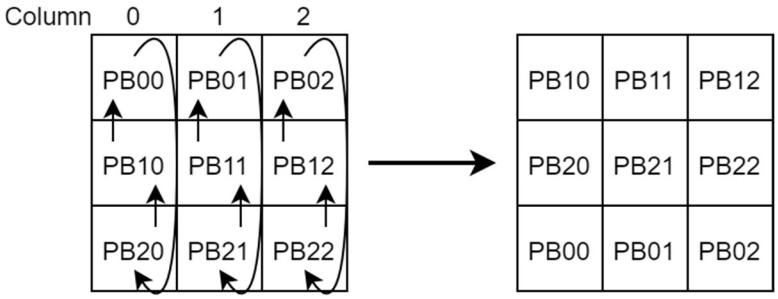


After receiving the broadcasted matrix A, each process updates the value of their local matrix C, using the broadcasted matrix A, and the local matrix B, as shown in the below pseudocode:

```
int local_matrix_size = n/sqrt(p);
temporary_matrixA = p[i][i+k] local_matrixA;
for (int i=0; i<local_matrix_size; i++) {
  for (int j=0; j<local_matrix_size; j++) {
    for (int k=0; k<local_matrix_size; k++) {
      local_matrixC[i][j] = local_matrixC[i][j] + (temporary_matrixA[i][k] * local_matrixB[k][j]);
    }
  }
}
```

After each process has updated the value of their local matrix C, each process then shifts their local matrix B one spot up on the processes-matrix, as was also done in the Cannon algorithm, and as is shown on Figure 6.

Figure 6



The algorithm does these three operations $\sqrt{p}$ times, after which the algorithm is finished.

The advantage of the Fox algorithm is, like the Cannon algorithm, that the memory required is quite low. The Fox algorithm does require more memory than the Cannon algorithm however, as each process needs an extra $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ matrix to store the broadcasted sub-matrix A. The Fox algorithm, like the Cannon algorithm, does not scale very well, owing to the large amount of communication between processes. It does scale better than the Cannon algorithm however, as the local matrix A is simply broadcasted across the entire row, instead of having to be shifted around, as was the case with the Cannon algorithm.

Results

In this section, I will test and analyze my three algorithms, and see how they perform. The test matrices used are always n × n, and are always created inside the programs themselves. Test matrix A contains only 1's, and test matrix B contains only 2's, while the resulting matrix C is set to initially contain only 0's. As mentioned before, the sub-matrices for the Cannon and Fox algorithms aren't created and then divided across the processes, but rather, for the sake of simplicity, they are created directly on the processes.

Each algorithm is tested on 1, 4, 9, 16, 25, and 36 processes. For 25 and 36 processes, the processes are split between two nodes. For each process count, each algorithm is tested for matrix sizes n × n of n = 720, 1440, 2880, and 5760. The algorithms are all tested on the NAU Monsoon cluster, always on the Skylake Xeon nodes, and always with the --exclusive command. All of the jobscripts used to run the programs have been provided in Appendix A. For each run, I have noted the total calculation time (thus, I'm ignoring the time it takes to create the test matrices). For each of the runs where n = 5760, I have also calculated the speedup and parallel efficiency, as well as measured the MaxRSS using the jobstats -j <jobid> command on the monsoon terminal, shown in the example screenshot below. The results are shown in tables 1 through 6, and figures 7 through 9.

```
[fvg6@cn68 ~/ondemand/HPC_Final_Project ]$ jobstats -j 38586355
JobID            JobName          ReqMem    MaxRSS    ReqCPUS   UserCPU     Timelimit   Elapsed    State       JobEff
=============================================================================================================================
38586355         CS599FP          97.6G     391M      28        09:44.585   00:20:00    00:09:57   COMPLETED   17.88
=============================================================================================================================
```

Table 1: The Naïve Algorithm

Processes	Time for n=720 (s)	Time for n=1440 (s)	Time for n=2880 (s)	Time for n=5760 (s)	Jobscript Name (.sh)
1	0.3726	8.177	137.234	440.507	script_naive_p1
4	0.2115	2.6762	35.74	359.7734	script_naive_p4
9	0.228	2.2738	16.879	165.9345	script_naive_p9
16	1.016	2.164	11.78	106.594	script_naive_p16
25	0.8115	2.739	14.013	94.2787	script_naive_p25
36	0.704	3.163	11.7159	87.4148	script_naive_p36

Table 2: Parallel Efficiency and Memory Usage for the Naïve Algorithm (n=5760)

Processes	Speedup	Parallel Efficiency	Memory Usage
1	1	1	391 MB
4	1.2244	0.306	57.7 GB
9	2.655	0.295	50.8 GB
16	4.1326	0.2583	47.7 GB
25	4.6724	0.1869	91.1 GB
36	5.039	0.14	126 GB

Table 3: Cannon Algorithm

Processes	Time for n=720 (s)	Time for n=1440 (s)	Time for n=2880 (s)	Time for n=5760 (s)	Jobscript Name (.sh)
1	0.449	8.85	136.806	553.3486	script_cannon_p1
4	0.5797	2.8477	25.043	307.9341	script_cannon_p4
9	0.4586	2.307	12.6231	134.73	script_cannon_p9
16	0.38	1.813	8.123	68.126	script_cannon_p16
25	0.8844	3.538	14.706	71.844	script_cannon_p25
36	0.881	3.4606	14.067	63.4965	script_cannon_p36

Table 4: Parallel Efficiency and Memory Usage for the Cannon Algorithm (n=5760)

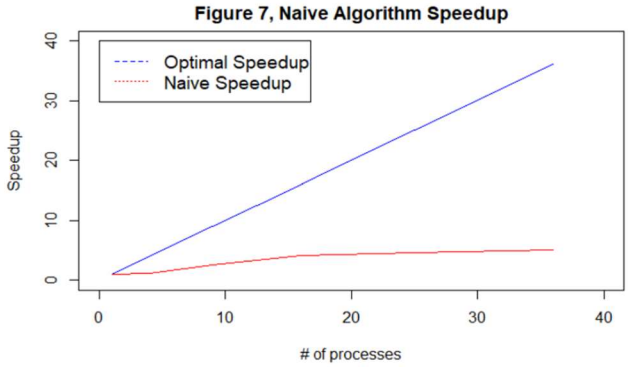
Processes	Speedup	Parallel Efficiency	Memory Usage
1	1	1	650 MB
4	1.797	0.449	175 MB
9	4.107	0.456	87.6 MB
16	8.122	0.5076	56.5 MB
25	7.702	0.308	116 MB
36	8.7146	0.242	109 MB

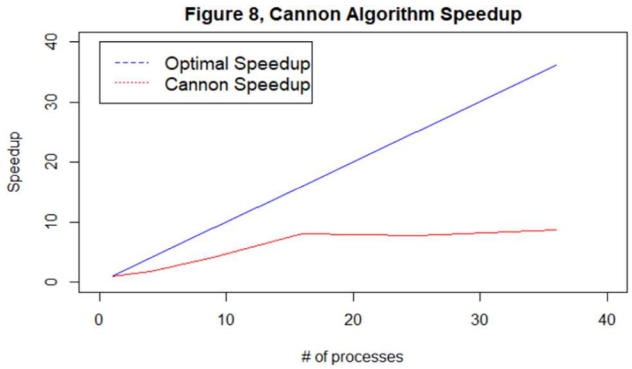
Table 5: Fox Algorithm

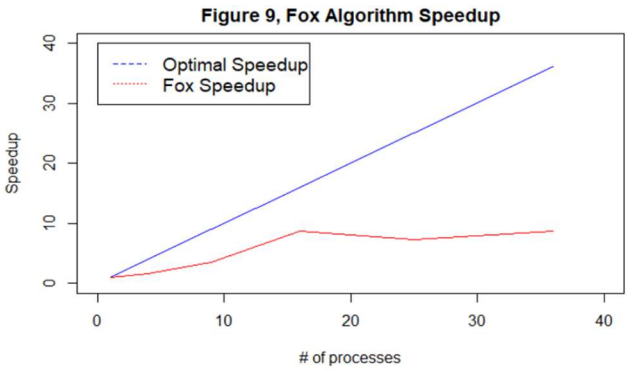
Processes	Time for n=720 (s)	Time for n=1440 (s)	Time for n=2880 (s)	Time for n=5760 (s)	Jobscript Name (.sh)
1	0.452	6.3152	79.7152	507.97	script_fox_p1
4	0.4174	2.379	19.314	315.446	script_fox_p4
9	0.5237	2.281	11.871	145.465	script_fox_p9
16	0.4383	1.855	8.507	58.791	script_fox_p16
25	0.87	3.418	14.174	69.345	script_fox_p25
36	0.8385	3.1867	12.8771	58.793	script_fox_p36

Table 6: Parallel Efficiency and Memory Usage for the Fox Algorithm (n=5760)

Processes	Speedup	Parallel Efficiency	Memory Usage
1	1	1	650 MB
4	1.61	0.4025	175 MB
9	3.49	0.388	87.2 MB
16	8.64	0.54	56.6 MB
25	7.325	0.293	117 MB
36	8.64	0.24	112 MB







Noteworthy Observations

From table 2 and figure 7, we see that my implementation of the Naïve algorithm has quite poor scalability. The algorithm has a minimal amount of communication between processes, so poor scalability like this could indicate that the algorithm is memory bound. Indeed, on table 2 we see that the MaxRSS is very high when this algorithm is run on multiple processes, in fact, I had to modify the script_naive_p36.sh jobscript, because I had not allotted the algorithm enough memory for this number of processes.

From tables 4 and 6, and figures 8 and 9, we see that the scalabilities of my implementations of the Cannon and Fox algorithms are not great either, but still better than the Naïve algorithm. This is likely because, despite the large amount of communication between ranks, these two algorithms are not memory bound. This is confirmed in tables 4 and six, as we can see that the MaxRSS is much lower, compared to the Naïve algorithm, and in fact, it becomes smaller as more processes are used.

From tables 4 through 6, and figures 8 and 9, we see that between $p = 16$ and $p = 25$, the speedup of the Cannon and Fox algorithms actually decreases, and the time to calculate increases, despite the increase in number of processes. This is likely a testament to the amount of communication between the processes that these two algorithms need, as for $p = 16$ only a single node is used, while for $p = 25$, two nodes are used. This means that communication time between two processes increases significantly, if they are located on different nodes, and as a result, the speedup decreases, and total calculation time increases.

Finally, we see that the scalability and memory requirements for the Cannon and Fox algorithms are nearly identical (although the Cannon algorithm has a quicker total calculation time), despite the fact that, theoretically, the Cannon algorithm should have worse scalability, but require less memory than the Fox algorithm. This is likely a result of the way I have implemented the Cannon algorithm, as I have added an extra $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ matrix to each node, that assists in shifting the matrices around in each phase of the matrix calculation.

Conclusion

Compared to my implementation of the Naïve algorithm, the Cannon and Fox algorithms that I implemented consistently had better performance when multiple ranks were used. The Cannon and Fox algorithms had better scalability, due to not being memory bound, and required way less memory than the Naïve algorithm. Thus, when multiplying large matrices, the Cannon and Fox algorithms seem to be good choices for matrix multiplication algorithms, provided that multiple processes are available. An interesting discovery I made was that adding more processes could potentially slow the program down, instead of speed it up, when using the Cannon or Fox algorithms. This was because of the large amount of communication required between processes that these algorithms require.