Texas A&M University Kingsville
Department of EECS
CSEN 5322 Operating Systems
Assignment 1 Matrix Multiplication

Mengxiang Jiang

October 19, 2022

Contents

Chapter 1

Introduction

Matrix multiplication has a huge number of applications in not just computer science but almost every technical field, ranging from computer graphics and neural networks to Markov chains used in both quantum mechanics and economics. As such, a huge amount of research has been done to efficiently do it. One might think the time complexity $\Theta(n^3)$ bounds the number of scalar multiplications needed, but Strassen's runs in $\Theta(n^{\lg 7})$ or about $O(n^{2.81})$ time.[2] This is by no means the best algorithm, and it is still an open problem what the optimal time complexity is. In fact, Deep Mind recently published a paper about using reinforcement learning to reduce the number of scalar multiplications for this purpose.[3] However, in this assignment, I am not going to use one of the more advanced algorithms but instead parallelize the naive algorithm, thereby taking advantage of hardware. The programming language I will use is C and the main library to parallelize the computation is MPI.

Chapter 2

Design

Originally, I intended to send a single row and a single column from the input matrices to each process to compute each entry of the output matrix independently. However, after doing this and testing the performance, it was significantly worse than not parallelizing the code at all, so instead, I decided to use the implementation given by the Lawrence Livermore National Laboratory MPI tutorial.[1] The way this implementation works is by having a master process divide the number of rows in input matrix by the number workers and sending each worker that many rows of A. If the two numbers do not evenly divide, then the extra rows are sent to each worker until no extra are left. Next, the entire input matrix B is sent to all the workers. Then the workers uses the naive matrix multiplication algorithm to calculate an intermediate output matrix and sends it to the master, who puts it together in the output matrix C. Below is a picture of how this works for a 5x3 matrix times a 3x2 matrix:

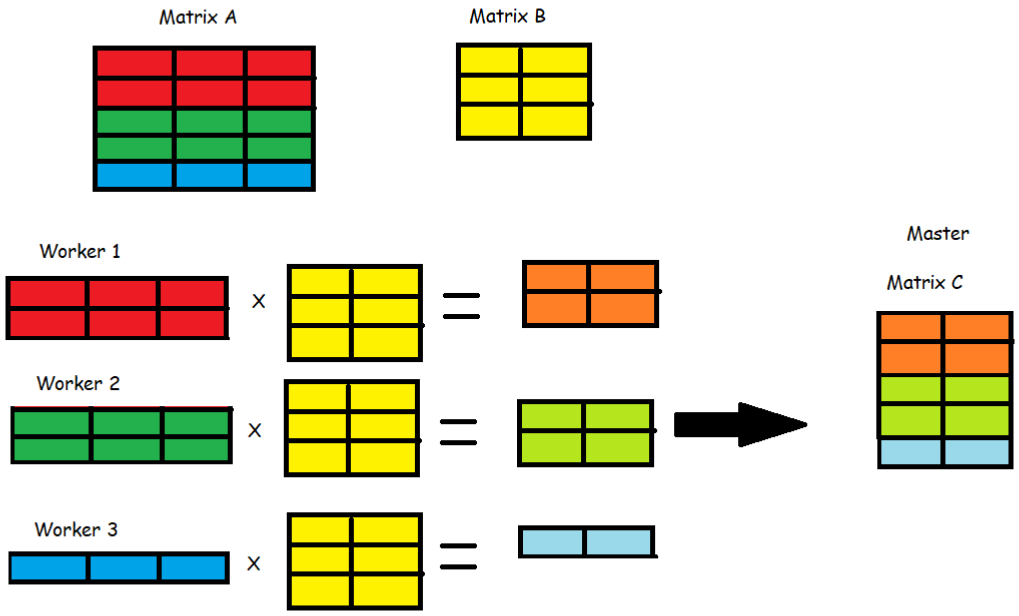

Figure 2.1: distributive algorithm

Chapter 3

Results

Running the read from file write to file matrix multiplication code (matrixmul.c) with 4 processes (1 master 3 workers) gives the following results:

```
$ mpicc matrixmul.c -o matrixmul
$ mpirun -np 4 ./matrixmul input0.txt
mpi_mm has started with 4 tasks.
Initializing matrices...
Number of Rows of A: 3
Number of Columns of A (also Rows of B): 4
Number of Columns of B: 5

Matrix A:

    3.45    -5.12     2.06   -19.10
    1.23     0.90     6.20    10.00
    9.57     9.01    -2.06     2.39
*****************************************************

Matrix B:

    5.43   -12.50    65.02    -1.10     6.20
    3.21    -9.02     4.56   -11.22     4.01
    5.82     1.08    -2.06   -23.45   -13.72
    83.47    56.10     4.28    42.20    -2.34
*****************************************************
Sending 1 rows to task 1 offset=0
Sending 1 rows to task 2 offset=1
Sending 1 rows to task 3 offset=2
Received results from task 1
Received results from task 2
Received results from task 3
*****************************************************
Result Matrix:

-1580.01   -1066.23   114.94   -800.58    17.34
880.35   544.20   114.16   265.16   -97.23
268.41   -69.04   677.79    37.45   118.08
*****************************************************
Done.
```

```
wall clock time = 0.003205
```

Listing 3.1: terminal1.txt

Note that the total time (0.003205 seconds) is longer than a sequential multiplication on random matrices of size 100x100 (0.001178 seconds). This is likely due to reading and writing to file, which is incredibly slow in comparison.

For the random nxn square matrix multiplication, I increased the maximum n from 50 to 500 because 50x50 was too small to notice. I also decreased the number of executions from 100 iterations to 10 iterations since the average times didn't seem to change much. For parallel execution, I did multiple runs with number of workers ranging from 2 to 8. Below is the plot of the results:

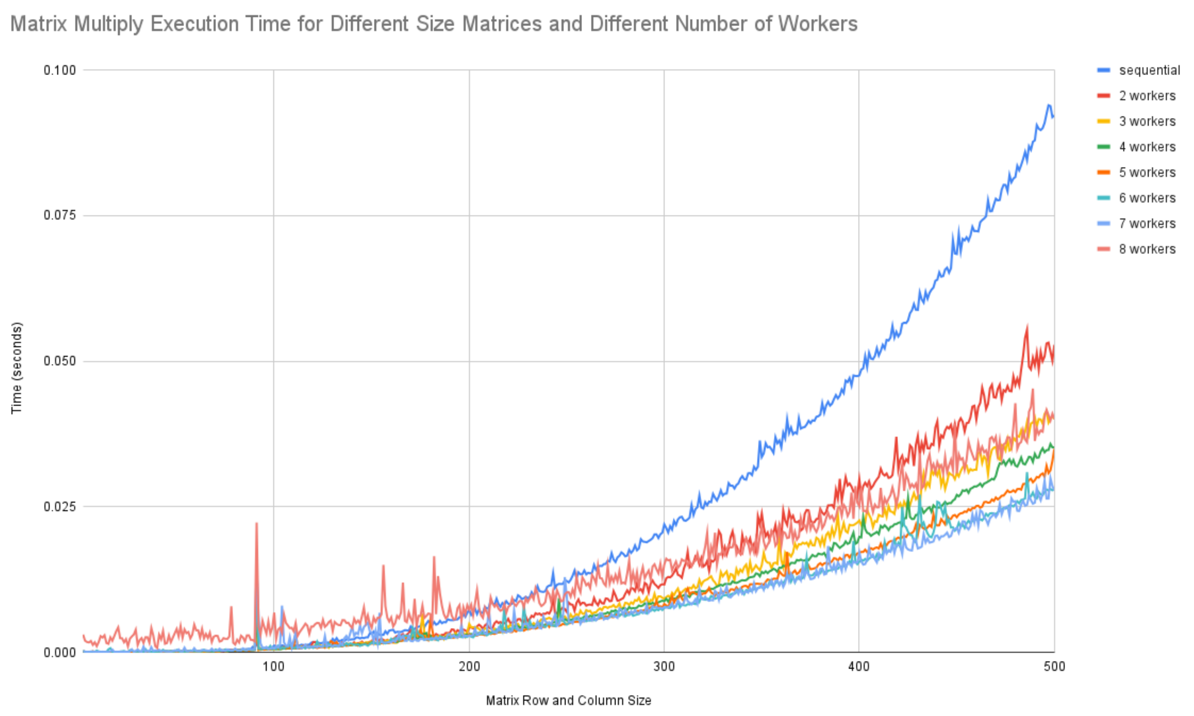

Figure 3.1: execution times

The biggest performance gain is from sequential to 2 workers. Every additional worker saw less gain, and when at 8 workers, there was a performance loss, very likely due to my computer having only 4 cores and 8 logical processors. Below is a copy paste of my dxdiag report:

```
------------------
System Information
------------------
      Time of this report: 10/19/2022, 02:44:00
             Machine name: DESKTOP-25GTGVH
               Machine Id: {9D8D4EA9-61DB-4C7A-BB3A-78BEF6E203AB}
         Operating System: Windows 10 Pro 64-bit (10.0, Build 19043) (19041.vb_relea
                 Language: English (Regional Setting: English)
      System Manufacturer: ASUS
             System Model: All Series
                     BIOS: 2205 (type: UEFI)
                Processor: Intel(R) Core(TM) i7-4790K CPU @ 4.00GHz (8 CPUs), ~4.0GH
                   Memory: 16384MB RAM
      Available OS Memory: 16260MB RAM
                Page File: 23200MB used, 6371MB available
              Windows Dir: C:\WINDOWS
          DirectX Version: DirectX 12
      DX Setup Parameters: Not found
         User DPI Setting: 96 DPI (100 percent)
       System DPI Setting: 96 DPI (100 percent)
          DWM DPI Scaling: Disabled
                 Miracast: Available, no HDCP
Microsoft Graphics Hybrid: Not Supported
 DirectX Database Version: 1.0.8
           DxDiag Version: 10.00.19041.2075 64bit Unicode
```

Note I am using Windows Subsytem for Linux using Ubuntu 14.04.6 LTS trusty with mpich version 3.0.4 when running these programs.

Chapter 4

Bibliography

[1] Blaise Barney. Message passing interface (mpi). `https://hpc-tutorials.llnl.gov/mpi/`, February 2022.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

[3] Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Francisco J. R. Ruiz, Julian Schrittwieser, Grzegorz Swirszcz, David Silver, Demis Hassabis, and Pushmeet Kohli. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610(7930):47–53, Oct 2022.