Analysis of Matrix Multiplication Parallelization

Kevin Chen
Computer Science Department
Rensselaer Polytechnic Institute
Troy, NY United States
chenk7@rpi.edu
4000 level

Xiaoyu Yang
Computer Science Department
Rensselaer Polytechnic Institute
Troy, NY United States
yangx18@rpi.edu
6000 level

Hongbin Liu
Computer Science Department
Rensselaer Polytechnic Institute
Troy, NY United States
liuh20@rpi.edu
4000 level

Zhe Yan
Computer Science Department
Rensselaer Polytechnic Institute
Troy, NY United States
yanz9@rpi.edu
4000 level

Abstract

This study looks at the parallelization of matrix multiplication which is one of the most widely used matrix operations such as network theory, and other linear algebra operations. With large sets of matrixes, serial runtime becomes large quickly and parallelization can help with this. The goal of this study was to see how much speed-up can be achieved through parallelization. For this study, we specifically choose to re-implement our matrix multiplication algorithm with parallel MPI I/O for input and output to test our high-degree parallel system's interaction. We performed a strong scaling test as well as a weak scaling test in order to analyze and compare the performance of using CPUs via MPI vs. hybrid CPU/GPU with both MPI and CUDA.

Introduction

Since the introduction of matrix multiplication(MM) as an algebra function there have been many strives to make improvements to the run time. In its traditional form matrix multiplication takes two matrices and integrates all their terms to store in a new matrix for its results. This would result in $O(n)^3$ run time. This is extremely bad as we approach dense matrices as this process could take hours to run through sequentially. This is because we would calculate this by having three loops to loop through the three matrices we will be using for calculation. The outer-most loop would be representative the row in the resulting matrix. The two inner loops would represent the two matrices being multiplied together.By parallelizing this process we can improve this in several ways.

In terms of parallelizing matrix multiplication, there are several approaches depending on how the matrices are divided up. The process can be done in several ways. For our approach, we will be doing something called block-striped decomposition. Each subtask will consist of calculating one element of the resulting matrix, which we will call matrix C. Each aggregated subtask will hold a row-wise block and a column-wise block for calculation. Each processor will do one subtask at a time, and repeated use of the processors will be cycled until all elements of the resulting matrix are calculated. Sadly there seems to be some computational error in our parallelization of matrix multiplication in that it returns a matrix thought with some errors included which will be discussed later.

In this paper, we seek to analyze the run-time improvement of this method compared to the traditional sequential method of matrix calculation using the Message Passing Interface(MPI). The algorithm was tested using RPI's AiMOS supercomputer, using up to 8 compute nodes though we may use less.

Related Works

As mentioned previously matrix multiplication is a useful tool for analyzing and representing certain real-world circumstances and improvements have been made to help decrease the run time progressively as time goes on. Here we will explore some of these

implementations done by other researchers and what kind of progress they have made.

In a paper written by C.-C. Chou, Y.-F. Deng, G. Li, Y. Wang, [1] explored the possibility of paralyzing Strassen's method on distributed memory MIMD. They mixed variations of the Broadcast-Multiply-Roll(BMR) method and a Ring method, which is a variation of the BMR method with Strassan's method. In their test, they suggested that the global parallelization of the Strassen method is always more popular than the BMR and Ring methods they tested. Not only does it reduce the number of floating point operations, but it also reduces the number of communication processes. Not everything is good as they did find some shortcomings for this parallelization as it claims the number of processors is "quantized," which destroys the flexibility of using an arbitrary number of processors. The pattern of communication is quite random, causing difficulty in implementation, and the Strassen method has some problems in numerical stability, as recognized by other studies.

Akimova, E. N., & Gareev, R. A [2] paper introduces an automatic compiler transformation that does not require an external code or automatic tuning to attain more than 85% of the performance of an optimized BLAS library. In their paper, they discuss how contemporary compilers and compiler front ends such as GCC and Clang cannot automatically transform a textbook-style implementation of the general matrix-to-matrix multiplication into a code that comes close to the performance of the expertly tuned multi-threaded implementation of matrix-to-matrix multiplication and how specialized libraries do provide high-performance matrix multiplication but requires previously optimized external code and can only be used if such an optimization existed in the first place. Their results concluded that having a better automatic vectorization to produce single-threaded matrix multiplication can help achieve better results.

A.M. Hemeida, S.A. Hassan, Salem Alkhalaf, M.M.M. Mahmoud, M.A. Saber, Ayman M. Bahaa Eldin, Tomonobu Senjyu, Abdullah H. Alayed [3] paper focused on optimizing matrix multiplication using intel's advanced vector extensions (AVX) multicore processor. Their optimization is designed using the AVX instructions sets, OpenMP parallelization, and memory access optimization to overcome bandwidth limitations. Instead of a generalized algorithm to optimize the said procedure, they focused on making a parallel implementation guideline instead where the target architecture's characteristics were taken into account when said algorithms were applied. Their results for their optimization found that multi-threads achieve significantly higher performance than their single-threaded counterparts.

Schatz, Martin., Van De Geijin, Robert., & Poulson, Jack [4] paper spoke about a systematic approach for developing distributed memory parallel matrix multiplication. Scalable implementations using MPI processes continue to be used to help extend these algorithms for 2D to so-called 3D algorithms, which view nodes as a 3D mesh. Ideas of this paper primarily focus on aspects of distributed-memory architectures that utilize a bulk-synchronous communication model for network communication.

The paper written by Tyler M. Smith, Robert van de Geijn, Mikhail Smelyanskiy, Jeff R. Hammond, & Field G. Van Zee [5] examines how they can parallelize the loops for matrix multiplication by using the BLIS approach for single-core implementation. BLIS framework exposes five opportunities for parallelization, and each section of these loops provides advantages in how data is being moved and how many communication calls are used. Factors that were used to determine which loops to parallelize were how much of an impact they made when done, loops that did not yield many results when parallelized were not optimized as the resources used to do the process cost more than what was returned, making it wasted run time and resources. All of their studies and experiments were done on the Blue Gene/Q.

S. Yin, Q. Wang, R. Hao, T. Zhou, S. Mei, and J. Liu's paper [6] examined irregularly shaped matrices for general matrix multiplication usage. They reasoned that for regularly shaped matrices, the libraries could achieve high performance for densely packed matrices but are often found not to work well with irregularly shaped ones. They focused on targeting multi-core DSPs in FT-m7032, a prototype CPU-DSPs heterogeneous processor for HPC, and an efficient implementation-ftIMM - for three types of irregular-shaped GEMMs proposed. FtIMM supports the automatic generation of assembly micro-kernels, two parallelization strategies, and auto-tuning of block sizes and parallelization strategies. The experiments show that ftIMM can get better performance than the traditional GEMM implementations on multi-core DSPs in FT-m7032, yielding up to 7.2x

performance improvement when performing on irregular-shaped matrices.

A presentation by Microsoft [7] examined three different ways matrix multiplication can be done in parallel. The block-striped decomposition has its data distributed such that matrix A would have its rows distributed and matrix B would have its column distributed.each subtask held a row and column to be computed and stored in the resulting matrix C. Another method was called Fox's method, where data was split in a checkerboard format, and each subtask consisted of a block from each of the two matrices being multiplied. Similarly, a third method called Canon's theory also implements the same checkerboard data distribution except in its data communication method. In their results, they found that while all three are great for parallelization, as the matrix size increased, the block-striped algorithm started to struggle to keep pace with the other two methods mentioned.

Grey Ballard, James Demmel, Olga Holtz, Benjamin Lipshitz, and Oded Schwartz [8] paper discussed a critical point of the bottleneck caused n parallelizing Strassen's algorithm is the communication between the processors. In turn, they managed to create an algorithm that is communication efficient and exhibits strong scaling within maximum range, obtaining speedups over the classical and strassen-based algorithms ranging from 24% to 184% for a fixed matrix dimension n=94080, where the number of processors ranges from 49 to 7203. They claim that their approach for parallelization in optimizing communication between the processors can be used to approach other fast matrix multiplication algorithms.

Alfio Lazzaro, Joost VandeVondele, Jürg Hutter, and Ole Schütt [9] explored a possible optimization of parallelization by reducing 2.5D algorithms and one-sided MPI communication in the context of linear scaling electronic structure theory. They compared the original implementation based on Cannon's algorithm and MPI point-to-point communication with an implementation based on MPI one-sided communications (RMA) in both a 2D and a 2.5D approach. The 2.5D approach trades memory and auxiliary operations for reduced communication, which can lead to a speedup if communication is dominant. They garnered a speed-up in performance for the RMA based 2.5D algorithm, up to 1.80x, which is observed to increase with the number of processes involved in the parallelization.

The presentation [10] demonstrates the simple parallel dense matrix and how the computational complexity takes an extremely inefficient route. It introduces Cannon's algorithm in order to improve memory efficiency as well as the DNS algorithm based on partitioning the intermediate data. We will take these designs into consideration for our implementation.

S. Ichimura, T. Katagiri, K. Ozaki, T. Ogita and T. Nagai paper examined what would happen by parallelizing high-precision matrix to matrix multiplication using thread-level parallelism. They experimented with the viewpoint of accuracy and speed. They found that splitting densely packed matrices into sparse matrices by adapting a sparse matrix-vector multiplication allowed a speed up of 1.43 times and a maximum of 38 times speed up when compared to the conventional implementation for densely packed matrix operation.

MPI Parallel I/O Implementation

In our implementation, we used MPI parallel I/O for input and output. We used Parallel MPI I/O in our design because it is a more advanced implementation of the standard MPI I/O, which provides optimizations for parallel file systems and allows data to be distributed across multiple disks. It also supports collective buffering and makes data buffered before writing to the file system. Thus, Parallel MPI I/O is a better choice for large-scale I/O operations compared to other I/O methods.

When implementing the parallel MPI I/O for our matrix multiplication code, we went through five phases: divide the matrix into blocks, distribute the blocks, read the matrixes, compute the matrix multiplication, and write the result.

When dividing the matrix into blocks, we divide the matrixes A, B, and C into blocks of size NxN, where N is the block size. Then we distribute the blocks of matrix A and B among the processes, where each process is responsible for multiplying its assigned blocks. After that, we read the blocks of matrices A and B assigned to each process from the input file using MPI I/O. Besides, each process reads its blocks in parallel, using the file offset and data size to determine its portion of the data. Moreover, we do the matrix multiplication algorithm, where we compute the matrix

multiplication for the assigned blocks of A and B and store the corresponding result in matrix C. Finally, we write the result of blocks of C assigned to each process to the output file using MPI I/O. Furthermore, each process writes its blocks in parallel, using the file offset and data size to determine its portion of the data.

In our implementation, each process reads and writes a portion of the matrix blocks assigned to it using MPI I/O functions, such as *MPI_File_set_view()*, *MPI_File_read_all()*, and *MPI_File_write_all()*. The matrix multiplication is performed on the assigned blocks, and the results are written to the output file using MPI I/O. We randomly generate the input matrices A and B in separate binary files named *inputA.bin* and *inputB.bin.* The output matrix C is also stored in a binary file.

We used several MPI I/O functions for our implementation. MPI_File_set_view() is used to set the file view of each process, allowing each process to read or write a specific portion of the file. MPI_File_read_all() and MPI_File_write_all() is used to perform collective I/O operations that read or write data from all processes at the same time, thereby improving I/O performance. This use of parallel MPI I/O operations makes our implementation achieves high I/O performance on large-scale computing systems.

Matrix Multiplication Implementation

Our Matrix Multiplication implementation involves writing custom kernel functions that perform the multiplication in parallel on the GPU.

Our CUDA kernel function *MatrixMultiplication* takes in 6 different inputs, the pointer point to the input matrix A, the pointer point to input matrix B, the pointer points to the result matrix, the int number representation of the column number of matrix A, the int number representation the row number of result matrix, and int number representation the column number of the result matrix. Within the function, each thread calculates a single element of the result matrix using the two input matrices, matrix A and matrix B. The calculation is performed using the standard formula for matrix multiplication: c(i,j) = sum_k (a(i,k) * b(k,j)), where the i and j in the formula represent the indices of the current element of the result matrix. Also, k represents the index of the column of a and the row of b being multiplied. Each thread has been assigned a unique (line, column) index using the blockIdx, blockDim, and threadIdx variables. The if statement at the beginning of the function checks whether the current thread is outside the bounds of the result matrix. If so, the thread returns without performing any computation. Otherwise, the thread proceeds to calculate the appropriate element of the result matrix using the input matrix A and matrix B.

The function *cudaInit* initializes a CUDA environment for our matrix multiplication. It is implemented to initialize the necessary memory and data structure for our matrix multiplication calculation on GPU using CUDA. It sets up the necessary memory on the GPU and copies the input matrices from the host to the device. The function takes in the rank of the current device in a multi-device setup, pointers to the input matrix in CPU memory, the column numbers of the first matrix, the total number of elements in the resulting matrix after calculation, the number of rows in the first matrix, and the size of the second matrix in bytes. The function first get the number of available CUDA devices and check for errors. Then allocate memory on the GPU for the input matrix and output matrix. Then it copies the input matrix from the host memory to the device memory. At last, it returns the matrix object which contains the device pointer to the input matrix and output matrix.

The function cudaReduce uses the same format as assignment 4 from the Rensselaer Polytechnic Institute CSCI-4320 course. The use of it is to call the kernel function. The function takes 8 inputs with the pointer point to the result matrix, the pointer point to the struct Matrix, int number represents the total number of elements in the result Matrix, int number represents the column number of row of the input matrix A, int number represents the row number of row of the input matrix A, int number represent the column number of row of the input matrix B, int number represents the row number of row of the input matrix B. The function first calculates the dimensions of the CUDA grid and block using the dim3 data type. The *blocks* variable is a 3D block dimension, where the x and y dimensions are calculated based on the number of columns and rows of MatrixA and the *NTHREADS_X* and *NTHREADS_Y* constants. The *threads* variable is a 3D thread dimension, where the x and y dimensions are set to the *NTHREADS_X* and *NTHREADS_Y* constants. The function calls the MatrixMultiplication kernel function using the <<<...>>> syntax, passing in the input matrices MatrixA and MatrixB that are stored in the struct Matrix and also the output matrix result. The

kernel function is launched with the *blocks* and *threads* dimensions. The parameters RowsNo2, RowsNo/size, and ColsNo correspond to the number of rows of MatrixB, the number of rows of MatrixA divided by the number of processors size, and the number of columns of MatrixA, respectively. Then the function copies the result matrix result from the GPU device to the host using cudaMemcpy, and synchronizes the device using *cudaDeviceSynchronize*().

For the comparison between the code running in the CPU mode using MPI I/O and hybrid(both CPU and GPU) mode. We also create a serial matrix multiplication file using the same MPI I/O method and code. Moreover, the performance result will be the comparison of the CPU mode matrix multiplication and the hybrid mode matrix multiplication.

Performance Result

As we mentioned before, our matrix multiplication has some errors in the result matrix. However, since the column and row numbers are correct, we believe the multiplication implementation of the parallel mode is correct, which we believe a possible problem occurred when setting up the algorithm for parallelization where there might have been synchronization errors. In this case, we believe that the run time will still be useful since it did the correct number of calculations, which we can use in our report.

We first performed a strong scaling study for our matrix multiplication implementation, where the problem size of our study remains the same with an increased CPU and GPU node count.

The problem size of this performance test is the size of the matrix–the matrix multiplication of two 768X768 matrices. The node count increases from 1 node to 2 nodes to 4 nodes to 8 nodes over time.

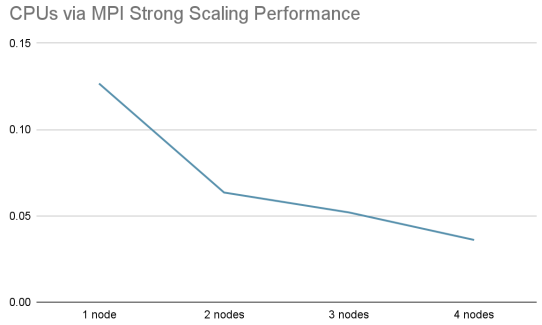
The purpose of our strong scaling test is to test the overall computational time of the job scales with the number of processing elements and to see how well our implementation can speed up the processing time for a fixed-size problem by utilizing more computational resources.

We got the following performance data from our strong scaling performance test. For CPUs via MPI method, 1 node takes 0.126656 seconds, 2 node takes 0.063504 seconds, 4 nodes take 0.051943 seconds, and 8 nodes take 0.036027 seconds. For CPU/GPU hybrid method, 1 node takes 0.001155 seconds, 2 nodes take 0.000792 seconds, 4 nodes take 0.000757 seconds, and 8 nodes take 0.000723 seconds.

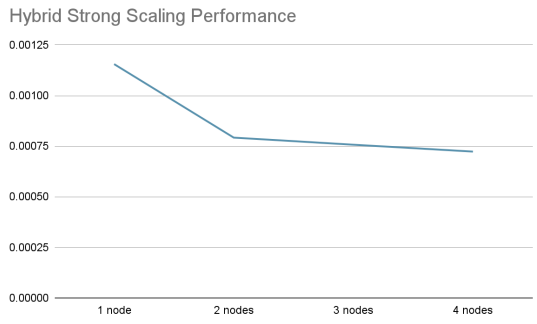
Below is a table of the performance result we got from running the strong scaling test.

type/node num	1 node	2 nodes	4 nodes	8 nodes
CPU	0.126656 s	0.063504 s	0.051943 s	0.036027 s
Hybrid	0.001155 s	0.000792 s	0.000757 s	0.000723 s

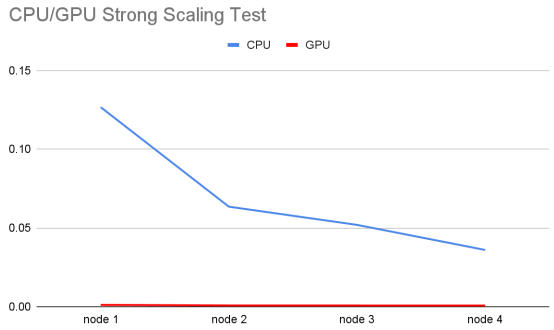
The graph of the performance curve by testing CPUs via MPI method with a strong scaling performance test is shown below.



The graph of the performance curve by testing hybrid GPU/CPU with both MPi and CUDA methods with a strong scaling performance test is shown below.



By showing the CPUs via MPI with hybrid CPU/GPU performance result in the same graph, we got the following graph. Note that because the GPU performance is so fast when compared to the CPU performance, it almost overlaps with the x-axis.



For our weak scaling test, the problem size, size of the matrix, increases with the increase of node counts. The goal of our weak scaling test is to test how well the system can maintain constant processing time per unit of workload as the total workload increases along with the number of processing units.

In our weak scaling test, the workload per processing element remains the same, but the overall problem size grows with the number of processing elements. If our design has perfect weak scaling, adding more processing units would lead to a linear increase in the total workload without increasing the processing time per unit of workload.

The problem size of this performance test is the size of the matrix–the matrix multiplication of two 768X768 matrices for test 1, multiplication of two 1536X1536 matrices for test 2, and two 3072X3072 matrices for test 3. The node count increases from 1 node to 2 nodes to 4 nodes.

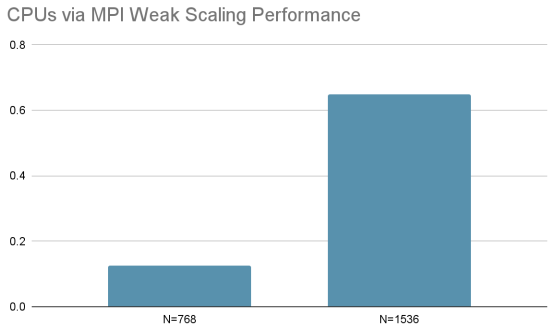
We got the following performance data from our weak scaling performance test. For CPUs via MPI method, the first test takes 0.126656 seconds, and the second takes 0.648317 seconds. For CPU/GPU hybrid method, the first test takes 0.001155 seconds, the second takes 0.000792 seconds, and the third takes 0.003647 seconds.

Below is a table of the performance results from running the weak scaling test.

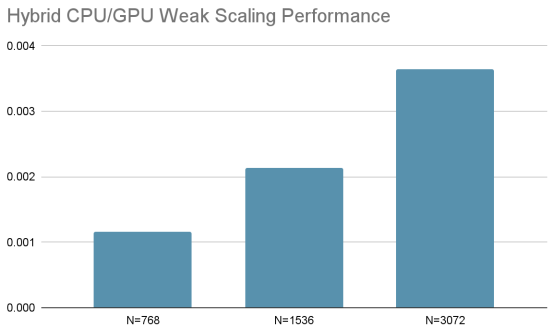
	CPU	Hybrid
N = 768	0.126656 s	0.001155 s
N = 1536	0.648317 s	0.00213 s
N = 3072	N/A	0.003647 s

Due to the limitations of the resources, we cannot provide the data of matrix rows and columns of 3072 and 6144 since 3072 requests 8 compute nodes on AiMOS and 6144 requires 16 nodes on AiMOS. However, calculating the 3072x3072 matrix may result in an out-of-memory error either due to not enough resources, such as sharing a node with another user, or the allocated resources were not enough to parse the data causing the memory error. Therefore, we can only provide the data as plotted above due to the limitation of time and resources.

The graph of the performance curve by testing CPUs via MPI method with a weak scaling performance test is shown below.

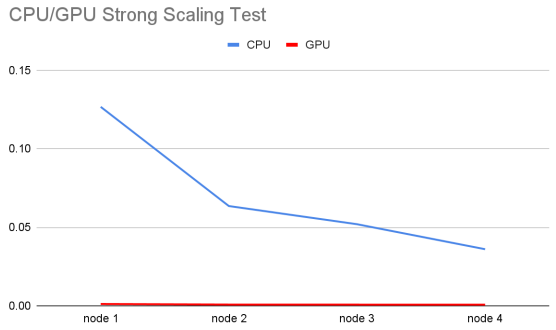


The graph of the performance curve by testing hybrid GPU/CPU with both MPi and CUDA methods with a weak scaling performance test is shown below.



Analysis

We compared and analyzed both the CPUs via MPI and hybrid CPU/GPU performance for our strong scaling test.



As shown in the above graphs, hybrid GPU/CPU clearly performs better than CPUs via MPI. For 1 node, the hybrid implementation is around 109 times faster. For 2 nodes, it is around 80 times faster. For 4 nodes, it is around 69 times faster. For 8 nodes, it is around 50 times faster. This showed that our parallel MPI I/O implementation is successful and has better performance compared to serial implementation. It also shows that parallel MPI I/O is a better choice for large-scale I/O operations.

We also calculate the speed-up for both implementations, and the result of the speedup is shown below.

	Speedup 1	Speedup 2	Speedup 3
CPU	1.99	2.44	3.52
Hybrid	1.45	1.53	1.60

We used the following equation for speed-up calculation: S = t_serial/ t_parallel, where S stands for speedup, t_serial stands for the processing time when using a single processing unit, and t_parallel stands for the processing time when using multiple processing units. From the speedup calculation, we can see that CPUs via MPI yield better speed-up compared to hybrid CPU/GPU. One explanation is that reading, writing, and assigning the workload to each process takes time. When the node number increases, the system has to do more calculations to assign the workload to each process, which can affect the overall performance of the design. Thus when the node count increases, the performance speedup for our parallel MPI I/O does not increase as much as serial implementation.

Based on our weak scaling test result, we can quickly get the weak scaling efficiency for both the implementation. We calculate the weak scaling efficiency using the equation E = t_1/t_n, where E stands for the weak scaling efficiency, t_1 stands for the processing time for a single processing unit with a smaller problem size, and t_n stands for the processing time of multiple processing units with a proportionally larger problem size. For the CPUs via MPI implementation, the weak scaling efficiency is around 0.2. For the Hybrid implementation, the weak scaling efficiency is around 0.54.

A weak scaling efficiency close to 1 indicates good weak scaling performance since the performance is consistent. A lower value scaling efficiency suggests that the design might not be maintaining constant processing time per processing unit as the workload increases.

Although both designs could not reach perfect weak scaling efficiency, we can still see that our hybrid implementation achieves higher weak scaling performance compared to the other design.

Summary

In this study, we implemented a variety of matrix multiplication using the block striped method, which used the rows of one matrix and the column of another to calculate the resulting matrix. As expected, the parallelization of the algorithm managed to perform better than the sequential one. For our strong scaling test, as shown, we see that the hybridized version performed better in run times as we gave it more cores to run the test with. However, as shown, we see that the improvement reaches up to a certain limit in that there is little significant improvement in any more cores we add for it to enter. In this case, Providing additional ones would just be a waste of resources and be too pricey for an increase in the effort.

Future Works

Given the limited time frame to complete this study. We only explored the block-striped decomposition by splitting the task of one matrix by columns and another by rows. Also, for testing purposes, all the values are fixed to 1. Another variation we found using this same method was using both matrices rows as subsets of data. The steps are similar, except it required a few more aggregated calculations which depended on previous iterations of said method. We also used N by N size matrices, the same dimensions for testing and practice of use. We could try to see if various shapes of matrices would play a role in such calculations and how it

could affect run time in the long run for densely packed matrices.

We also mentioned there were two other methods called Fox's and Cannon's method which distributed the data in a checkerboard format where they divided the two matrices into smaller ones for each CPU to compute. We looked at the papers comparing the striped decomposition and Cannon's/Fox's method. It showed that the striped decomposition performed better than others to a certain degree. In the future, we can try to implement all the mentioned methods and try to compare them in various ways, such as matrix size and shape, to see if they could be compared. MPI implementations could be explored to see how much of a speed-up can be contributed compared to just using multiple threads. After correcting the multiplication method, we leave space for the column-row matrix computation, not just N by N size matrices. Plus, during our testing, we found that MPI_Scatter and MPI_Bcast can also contribute to improving performance. In this circumstance, we can do more complex computations with these methods.

Strassen's method was frequently mentioned in the following papers with modifications to suit the tested architecture. Perhaps we can compare the various methods and try to optimize them to MPI and use the AiMos supercomputer and see how that would stand up to its variations.

Contributions

The workload of this project is divided among our four members: Kevin Chen, Hongbin Liu, Xiaoyu Yang, and Zhe Yan. Kevin Chen works on background research, project report paper, and necessary documentation. Hongbin Liu works on the MPI Parallel I/O part of the project and project report relating to MPI Parallel I/O as well as matrix multiplication. Xiaoyu Yang and Zhe Yan worked on the matrix multiplication algorithm and project report relating to the matrix multiplication implementation. All team members work on the performance testing, analysis, and report summary.

References

1. C.-C. Chou, Y.-F. Deng, G. Li, Y. Wang, Parallelizing Strassen's method for matrix multiplication on distributed-memory MIMD architectures, Computers & Mathematics with Applications, Volume 30, Issue 2,1995, Pages 49-69, ISSN 0898-1221, https://doi.org/10.1016/0898-1221(95)00077-C.

2. Akimova, E. N., & Gareev, R. A. (n.d.). *Algorithm of automatic parallelization of generalized matrix ...* Retrieved April 20, 2023, from https://ceur-ws.org/Vol-2005/paper-01.pdf
3. A.M. Hemeida, S.A. Hassan, Salem Alkhalaf, M.M.M. Mahmoud, M.A. Saber, Ayman M. Bahaa Eldin, Tomonobu Senjyu, Abdullah H. Alayed, Optimizing matrix-matrix multiplication on intel's advanced vector extensions multicore processor, Ain Shams Engineering Journal, Volume 11, Issue 4,2020, Pages 1179-1190, ISSN2090-4479, https://doi.org/10.1016/j.asej.2020.01.003.
4. Schatz, Martin., Van De Geijin, Robert., & Poulson, Jack. . *CI OMPUT C Vol. 38, no. 6, pp. C748{C781 - University of Texas at Austin*. Retrieved April 20, 2023, from https://www.cs.utexas.edu/users/flame/pubs/2D3DFinal.pdf
5. Tyler M. Smith, Robert van de Geijn, Mikhail Smelyanskiy, Jeff R. Hammond, & Field G. Van Zee. (n.d.). *Anatomy of high-performance many-threaded matrix multiplication*. Retrieved April 20, 2023, from https://www.cs.utexas.edu/users/flame/pubs/blis3_ipdps14.pdf
6. S. Yin, Q. Wang, R. Hao, T. Zhou, S. Mei and J. Liu, "Optimizing Irregular-Shaped Matrix-Matrix Multiplication on Multi-Core DSPs," 2022 IEEE International Conference on Cluster Computing (CLUSTER), Heidelberg, Germany, 2022, pp. 451-461, doi: 10.1109/CLUSTER51413.2022.00055.
7. *Parallel methods for matrix multiplication - INPE*. (n.d.). Retrieved April 21, 2023, from http://www.lac.inpe.br/~stephan/CAP-372/matrixmult_microsoft.pdf
8. Grey Ballard, James Demmel, Olga Holtz, Benjamin Lipshitz, and Oded Schwartz. 2012. Communication-optimal parallel algorithm for strassen's matrix multiplication. In Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures (SPAA '12). Association for Computing Machinery, New York, NY, USA, 193–204. https://doi.org/10.1145/2312005.2312044
9. Alfio Lazzaro, Joost VandeVondele, Jürg Hutter, and Ole Schütt. 2017. Increasing the Efficiency of Sparse Matrix-Matrix Multiplication with a 2.5D Algorithm and One-Sided MPI. In Proceedings of the Platform for Advanced Scientific Computing Conference (PASC '17). Association for Computing Machinery, New York, NY, USA, Article 3, 1–9. https://doi.org/10.1145/3093172.3093228
10. *University of Notre Dame*. (n.d.). Retrieved April 21, 2023, from https://www3.nd.edu/~zxu2/acms60212-40212/Lec-07-3.pdf
11. S. Ichimura, T. Katagiri, K. Ozaki, T. Ogita and T. Nagai, "Threaded Accurate Matrix-Matrix Multiplications with Sparse Matrix-Vector Multiplications," 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Vancouver, BC, Canada, 2018, pp. 1093-1102, doi: 10.1109/IPDPSW.2018.00168.