Improving Latency for Secure Distributed Matrix Multiplication: Optimizing the Secure Mat-Dot Code Algorithm

CS4991 Capstone Report, 2024

Tiffany Bui
Computer Science
The University of Virginia
School of Engineering and Applied Science
Charlottesville, Virginia USA
tnb6zdz@virginia.edu

ABSTRACT

Large-scale matrix multiplication is complex and costly, so it is commonly run on distributed systems for an efficient run time, but this risks data security within the matrices. Existing algorithms for Secure Distributed Matrix Multiplication, include Secure Mat-Dot Code, Gap Additive Secure Polynomial Code, and Linear Code. The goal of my project was to research and select the best algorithm, implement it with security measures, and make adjustments to improve efficiency. The Secure Mat-Dot Code algorithm was the best, given its flexibility and complexity. My team and I implemented the solution with secret sharing, coding theory, and polynomial codes to reduce latency and keep the matrices private. On a 2048x2048 matrix, average run time was 207 seconds. To further reduce latency and improve efficiency, the best optimization methods were multithreading, pre-calculating the computationally heavy operations, optimizing interpolation, and transposing the matrix. These modifications reduced the run time for a 2048x2048 matrix by 92.3% (207s to 16s). Implementations that would further speed up the program include removing modular arithmetic by implementing a Galois field, replacing the Vandermonde matrix with Lagrange's method, and using OpenMP instead of SHMEM.

1. INTRODUCTION

Matrix multiplication is essential to many fields, including data science and cloud computing. It may be a simple mathematical operation for smaller matrices, but as the matrix grows, it becomes increasingly more difficult to perform matrix multiplication. To get an idea of how complex it becomes, for a naïve matrix multiplication method (with a triple nested for-loop), where the matrix dimensions are $N \times N$, the total time complexity would be $O(N^3)$.

2. RELATED WORKS

D'Oliveria, et. al. (2020) discusses GASP codes as a solution to the problem of inefficient and nonsecure matrix multiplication. They posited that constructing polynomial codes based on arithmetic progressions is a major advantage to this approach, though the complexity of implementing GASP codes is a major drawback. My project did not utilize GASP codes given its complexity and the restraints imposed on me by the internship timeline, but it was important to read about and explore it as a viable option.

Makkonen (2022) discusses different schemes that can be used for secure distributed matrix multiplication. They described schemes such as Secure Mat-Dot code, GASP code, and Linear SDMM. The secure Mat-Dot code is presented as simplistic but effective by

splitting the computation into smaller chunks and using Reed-Solomon codes to securely disperse the chunks to each of the nodes. My project borrowed a portion of Makkonen's recommendation by using the Secure Mat-Dot algorithm in my implementation.

Bryant, et. al. (2012) proposes the use of blocking as a potential solution to optimize code. The major advantages to adopting this approach include the implementation simplicity, since it is just a reorganization of inner for-loops, and its proven effectiveness in increasing temporal locality. My approach to optimizing my code included utilizing blocking, which ended up being very effective in decreasing the run time of my matrix multiplication.

3. PROJECT DESIGN

I decided to use the Secure Mat-Dot algorithm, which divides the matrix multiplication computation into smaller pieces, and uses Reed-Solomon codes to securely distribute the pieces to different nodes.

3.1 Reed-Solomon Codes

A common solution to the problem of inefficient matrix multiplication is parallelism. A naïve implementation would have one node perform the entire matrix multiplication, but parallelism would mean using multiple nodes to work on computing their own subsection of the final matrix. They would perform their computations at the same time and combine their results at the end to get the final product in a timelier manner. The issue with parallelism, however, is the straggler effect–the final answer depends on each node's computations, so the entire matrix multiplication computation is only as fast as its slowest node. The solution to this is using Reed-Solomon codes, also known as polynomial codes, which encode matrices in a way that the result no longer depends on every node finishing its computations, and only requires a subsection of the nodes to finish.

Another issue with parallelism is security. If the original matrices contain sensitive information, then it is possible for compromised worker nodes to extract that information. Security vulnerabilities can be addressed by Reed-Solomon codes, which are designed for reliability and security. Randomly generated matrices also add an extra layer of security when used in combination with the original matrices when creating the polynomial codes.

3.2 Secure Mat-Dot Algorithm

The Secure Mat-Dot Algorithm splits the matrix multiplication computation into chunks and uses Reed-Solomon codes to distribute the pieces securely and reliably between the nodes.

Let A and B be two matrices, A is of size $t \times s$ and B is of size $s \times r$. The goal is to compute $C = AB$, with m worker nodes available, while also ensuring that up to k colluding nodes gain no information about either matrix.

We begin by selecting n such that $2(n+k)-1 \leq m$, and splitting both matrices into n submatrices:

$$A = [A_1 \dots A_n]$$

$$B = \begin{bmatrix} B_1 \\ \dots \\ B_n \end{bmatrix}$$

Now, $C = AB$ is equivalent to $A_1B_1 + \dots + A_nB_n$, since we have A_n submatrices and B_n submatrices. Each submatrix of A, which will be referred to as A_i, has a size of $t \times \frac{s}{n}$, and each submatrix of B, which will be referred to as B_i, has a size of $\frac{s}{n} \times r$. If n does not divide s, then zero columns and zero rows are added

to A and B respectfully, until n divides s. Each submatrix of A and B is then encoded using Reed-Solomon codes, and the following matrices are randomly generated: matrices $R_1, \ldots, R_k$ of size $t \times \frac{s}{n}$ (so that each submatrix has the same dimensions as A_i), as well as matrices $S_1, \ldots, S_k$ of size $\frac{s}{n} \times r$ (so that each submatrix has the same dimensions as B_i). Thus, we can define three polynomials:

$$f(x) = A_1 + \ldots + A_n x^{n-1} + R_1 x^n + \ldots + R_k x^{n+k-1}$$
$$g(x) = B_n + \ldots + B_1 x^{n-1} + S_1 x^n + \ldots + S_k x^{n+k-1}$$
$$h(x) = f(x) \cdot g(x)$$

h has a degree of $2(n+k-1)$, and when simplified, h has $l = 2(n+k)-1$ terms. h defines a Reed-Solomon encoding of C since the x^{n-1} term in h is $A_1B_1 + \ldots + A_nB_n$. Thus, by computing l points of h, we can interpolate the polynomial and find C. Since $l = 2(n+k)-1 \leq m$, we can recover C by having each worker node compute one point of h.

We accomplish this by generating values $\alpha_1, \ldots, \alpha_m$. For each $i \in \{1, \ldots, m\}$, we compute the matrices $\widetilde{A}_i = f(\alpha_i)$ and $\widetilde{B}_i = g(\alpha_i)$ and send them to node i. Each node then computes $\widetilde{C}_i = \widetilde{A}_i\widetilde{B}_i = h(\alpha_i)$ and returns $\widetilde{C}_i$. Once $2(n+k)-1$ different points $(\alpha_i, \widetilde{C}_i)$ of h are returned, we interpolate the polynomial $h(x) = C_1 + C_2x + \ldots + C_lx^{l-1}$ and return $C = C_n$, the product of AB.

4 RESULTS

The base case implementation of the Secure Mat-Dot Algorithm had a total run time of 207 seconds, where the size of the matrix was 2048x2048, the number of splits (of A and B into the submatrices) were 15, and the number of worker nodes used were 55. While keeping these values constant, we tested different optimization methods. The most effective optimization methods were to compile with optimization level -O3, pre-calculating the α_i powers, improving how we gather shares of elements, removing unnecessary computations and data structures, and transposing the matrix to improve memory access patterns. Our final run time with the optimizations was 16 seconds.

We also wanted to see if blocking the matrix to improve cache use would be better than transposing. Transposing was faster given our constants, but we wanted to alter the values of splits and matrix dimensions to see which had a better total runtime. The transpose solution was consistently faster for varying matrix dimensions and number of splits. Thus, the transpose solution is the optimal solution.

We also wanted to compare our implementations to a single-node naïve implementation and a single-node Strassen's algorithm implementation. They are single node to maintain security. We expected our Secure Mat-Dot algorithm implementation to run slower since it had the added steps to improve security and distributions. However, when compared at varying numbers of nodes, splits, and matrix dimensions, our algorithm consistently performed better than the naïve and Strassen's implementations.

5 CONCLUSION

The project demonstrated that the implementation of secure distributed matrix multiplication using polynomial codes and a shared memory system is feasible and more efficient than other single-node options such as the naïve approach and Strassen's algorithm. This opens the door to further exploration since the incorporation of polynomial codes in secure distributed matrix multiplication is proven to be effective and efficient.

6 FUTURE WORK

In the future, I would want to improve and optimize the code. One example would be improving the way I am performing matrix

multiplication, which is through transposing the matrix, by trying multi-threading or algorithms such as Strassen's algorithm. Also, I could remove the modular arithmetic by expanding the field to all real numbers or by implementing a Galois field. I could test Lagrange's method in place of the Vandermonde matrix. I could also try different parallelism systems; I am currently using SHMEM, but I could experiment with OpenMP. I could also test a completely different algorithmic scheme instead of the Secure Mat-Dot algorithm, such as GASP code.

7 ACKNOWLEDGMENTS

This project was completed while I was an Advanced Computing Systems Research Intern for the National Security Agency. I was part of a three-person scrum team, including Bailey Canham from California State University, Los Angeles, and Nathan Daly from Johns Hopkins University. We were mentored by Fiona Knoll, who is an Assistant Professor of Computer Science at the United States Naval Academy.

REFERENCES

Bj¨orck, ˚Ake, and Victor Pereyra. "Solution of Vandermonde Systems of Equations." Mathematics of Computation, vol. 24, no. 112, 1970, pp. 893–903. JSTOR, https://doi.org/10.2307/2004623. Accessed 21 Jul. 2022.

Bryant, Randal E, and David R O'Hallaron. CS:APP2e Web Aside MEM:BLOCKING: Using Blocking to Increase Temporal Locality*. 5 June 2012, https://csapp.cs.cmu.edu/public/waside/waside-blocking.pdf.

D'Oliveira, Rafael G., et al. "GASP Codes for Secure Distributed Matrix Multiplication." IEEE Transactions on Information Theory, vol. 66, no. 7, 11 Feb. 2020, pp. 4038–4050., https://doi.org/10.1109/tit.2020.2975021.

Li, Jie, and Camilla Hollanti. "Private and Secure Distributed Matrix Multiplication Schemes for Replicated or MDS-Coded Servers." IEEE Transactions on Information Forensics and Security, vol. 17, 30 Jan. 2022, pp. 659–669., https://doi.org/10.1109/tifs.2022.3147638.

Makkonen, Okko. "New Schemes for Secure Distributed Matrix Multiplication: Cooperative and Analog SDMM." Aaltodoc, 22 Mar. 2022, p. 78+4., http://urn.fi/URN:NBN:fi:aalto-202203272571. Accessed 20 July 2022.

Martin D. Schatz, Robert A. van de Geijn, and Jack Poulson. 2016. Parallel Matrix Multiplication: A Systematic Journey. SIAM J. Sci. Comput. 38, 6 (2016), C748–C781. https://doi.org/10.1137/140993478.

"Selecting Optimization Options." Documentation – Arm Developer, Arm Limited, https://developer.arm.com/documentation/100748/0612/using-common-compiler-options/selecting-optimization-options.

"Shamir's Secret Sharing." Wikipedia, Wikimedia Foundation, 5 July 2022. https://en.wikipedia.org/wiki/Shamir

"Strassen Algorithm." Wikipedia, Wikimedia Foundation, 13 June 2022.

"The Importance of Scalability in Software Design." Award-Winning App Development Company, Concepta Technologies, https://www.conceptatech.com/blog/importance-of-scalability-in-software-design.