Matrix multiplication is utilized in a very diverse range of fields, but it often comes with a big trade-off. For a program to run within a reasonable time frame, the most common solution is to turn to distributed systems and delegate part of the multiplication process to various worker nodes. However, by sending this data to possibly compromised nodes, you are losing the security the matrices previously had by running all processes on the same machine with access to the same physical memory. Through investigating Secure and Distributed Matrix Multiplication this summer, we have implemented a program that allows for both security and efficiency. Our implementation utilizes secret sharing to keep the matrices private, as well as coding theory to reduce latency. We combine both using polynomial codes, or more specifically, the Mat-Dot Algorithm. Overall, we found that our implementation is more efficient than running single-node implementations of matrix multiplications (i.e. Strassen’s, naive), which is the other option to keep the data secure.
1 Background

Matrix multiplication is an essential mathematical process that is used in a variety of diverse fields, from data science to cloud computing. However, even though the math behind matrix multiplication is relatively simple calculations, certain aspects of these multiplications make them difficult and complex to implement on a large scale. One example of this would be the amount of time it takes to perform matrix multiplication. When utilizing naive matrix multiplication, the computer has to traverse over the entirety of both matrices multiple times, making the time for the program to complete increase quickly as the matrix size increases.

In order to combat this dramatic increase in time, many people turn to parallelism. With this strategy, the matrices are split up into sub-matrices and distributed across multiple computing nodes, allowing each computing node to calculate a subsection of the matrix multiplication at the same time, then put all the results together to get the final answer. While this solution does work quite well when implemented in a logical manner, it unfortunately also brings its own set of challenges, such as the straggler effect. The straggler effect is the fact that a program can only calculate the final answer once all nodes have returned their sub-matrix, meaning that the program is only as fast as it’s slowest node or nodes, also known as stragglers. One way to work around the straggler effect is to use polynomial codes. By encoding matrices using polynomial codes, it is no longer needed for every single node to return their answer before the program is able calculate the final solution. Rather, the program only needs a subsection of the nodes to respond, removing the extra time that an exceptionally slow node or nodes might have cost.

Another aspect of matrix multiplication that needs to be considered is the security of the data within said matrices. If the matrices being multiplied contain private data, steps need to be taken in order to hide said data from worker nodes that could possibly be compromised. Even though the inherent nature of distributing subsections of the matrices to different nodes makes it nearly impossible for an individual worker node to infer the original matrices, a group of compromised worker nodes colluding together could calculate the original data. In order to try and prevent collusion as much as possible, randomly generated matrices are used in addition to the original matrices when creating the polynomial code. By doing this, the original data is protected from a specific number of colluding nodes.

2 Algorithm

Our program uses the Secure Mat-Dot algorithm, which—along with many other SDMM algorithms—splits the computation into smaller pieces and uses Reed-Solomon codes to securely distribute those pieces between the nodes. While the original purpose of Reed-Solomon codes is not security but reliability, they have properties which lend themselves well to this application.

2.1 Reed-Solomon Codes

Reed-Solomon codes, also known as polynomial codes, consist of an encoder and a decoder connected by an unreliable communication channel. The Reed-Solomon encoder takes as input a vector \( a = (a_0, a_1, \ldots, a_{n-1}) \) and constructs an \((n-1)\)-degree polynomial \( f \) with the components of \( a \) as coefficients:

\[
f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}
\]

It then evaluates \( f \) at \( m \geq n \) different points \( \alpha_i \), to construct an output vector \( b = (f(\alpha_1), f(\alpha_2), \ldots, f(\alpha_m)) \).

We assume that the \( \alpha_i \) are known to both the encoder and decoder beforehand. Upon receiving \( b \), decoder can recover \( a \) since any \( n \) points define an unique \((n-1)\)-degree polynomial. And depending on the size of \( m \) relative to \( n \), the original vector \( a \) can be recovered even when some elements of \( b \) are lost or corrupted.

Suppose we wish to transmit \( a = (3, 5) \). We first construct our polynomial \( f(x) = 3 + 5x \). For simplicity, we can choose \( \alpha_i = i \) for all \( i \). So if \( m = 2 \), then \( b = (f(1), f(2)) = (8, 13) \). The decoder now knows that \( f(x) \) passes through the points \((1, 8)\) and \((2, 13)\), and since any two points uniquely define a line, concludes that \( f(x) = 3 + 5x \) and therefore \( a = (3, 5) \). But suppose \( b \) is corrupted and the decoder receives \( b' = (9, 13) \) instead. Then it will wrongly conclude that \( f(x) = 5 + 4x \) and therefore \( a = (5, 4) \).

We can prevent this by increasing \( m \). If \( m = 3 \) then \( b = (8, 13, 18) \). If the same error occurs, the decoder will receive \( b' = (9, 13, 18) \). Since no one line passes through all three points \((1, 9), (2, 13), \) and \((3, 18)\), the
decoder will be able to detect the error. It cannot, however, correct the error, since all three pairs of points define a different line and all three lines “miss” exactly one point.

By increasing \( m \) once more, we can enable the decoder to both detect and correct a single error. If \( m = 4 \), then \( b = (8, 13, 18, 23) \), and \( k' = (9, 13, 18, 23) \). Once again, the decoder will detect that an error has occurred since no one line passes through all four points. This time it can not only detect but also correct the error, noticing that \( f(x) = 3 + 5x \) passes through all three of \((2, 13), (3, 18)\), and \((4, 23)\) and misses only the first point \((1, 9)\). But every line passing through \((1, 9)\) misses at least two of the other three points. If only one error occurred, then it must have been the first point that was corrupted. And so, assuming that a single error is more likely to occur than multiple errors, the decoder can now conclude that \( a = (3, 5) \).

As an aside, we can express the encoding process with the formula \( b = Ga \), where \( G \) is an \( m \times n \) generator matrix of the form:

\[
G = \begin{bmatrix}
1 & \alpha_1 & \alpha_1^2 & \cdots & \alpha_1^{n-1} \\
1 & \alpha_2 & \alpha_2^2 & \cdots & \alpha_2^{n-1} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
1 & \alpha_m & \alpha_m^2 & \cdots & \alpha_m^{n-1}
\end{bmatrix}
\]

Thus, if \( m = n \) and \( b \) is uncorrupted, we can decode as \( a = G^{-1}b \).

### 2.2 Secret Sharing and Arithmetic Properties

Reed-Solomon codes have useful security properties. Consider a scenario: We are no longer coding theorists but cryptographers; we fear not errors but eavesdroppers. Suppose we encode \( a \) to \( b \) using an \( m = n \) Reed-Solomon code, and an adversary manages to intercept \( k < n \) elements of \( b \) as we transmit it. How many elements of \( a \) have been compromised? In one sense, none: referring back to the generator matrix notation, we see that every element of \( a = G^{-1}b \) is a linear combination of every element of \( b \). So without the entire vector \( b \), the adversary cannot reconstruct a single element of \( a \). But in another sense, every element of \( a \) has been compromised: knowing \( k \) elements of \( b \), the adversary can write each element of \( a \) as a linear combination of the remaining \( n - k \) elements of \( b \). In other words, he can restrict the search space of \( a \) from an \( n \)-dimensional space to an \((n - k)\)-dimensional hyperplane. While this is already more information than we want him to have, in some cases he may have prior knowledge about the structure of \( a \) that allows him to reconstruct the entire vector.

We can defend against this adversary by setting only part of \( a \) as useful information and randomly generating the rest. If \( k \) elements of \( a \) are random, then even with \( k \) elements of \( b \) the adversary gains no useful information about \( a \), since the information he wants (the nonrandom elements of \( a \)) was restricted to \((n - k)\) dimensions from the start.

Consider another scenario: After encoding \( a \) into \( b \), suppose that we split \( b \) into its elements (points of \( f \)) and distribute one to each of \( n \) agents for safekeeping. Suppose then that \( k \) of these agents collude, combining their information in hopes of finding \( a \). Mathematically, this scenario is identical to the first. If we randomize \( k \) elements of \( a \), then no \( k \) colluding agents can glean any useful information about \( a \). What we now have is a cryptographic secret sharing scheme. In the special case where \( k = n - 1 \) (i.e. only one element of \( a \) contains useful information), this is known as Shamir’s Secret Sharing.[8]

Reed-Solomon codes have the additional benefit that we can perform arithmetic on the message by performing arithmetic on the codeword. Suppose we have two messages \( a \) and \( a' \), both of length \( n \), encoded into two vectors \( b \) and \( b' \), both of length \( m \). If we add \( b \) and \( b' \), the resulting vector encodes precisely \( a + a' \). If we multiply (element by element) \( b \) and \( b' \), the resulting vector encodes the coefficients of \( f \cdot f' \), where \( f \) and \( f' \) are the polynomials used to encode \( a \) and \( a' \) respectively. This vector has length \( 2n - 1 \) and takes the form \((a_0a_0', a_0a_1' + a_1a_0', a_0a_2' + a_1a_1' + a_2a_0', \ldots, a_{n-1}a_0'_{n-1})\). Thus, by careful construction of \( a \) and \( a' \), we can compute useful values simply by multiplying \( b \) and \( b' \). And if we distribute the elements of \( b \) and \( b' \) to other agents, then as discussed above we can let those agents perform the multiplication for us, without sacrificing security. In this way, we can use Reed-Solomon codes to perform secure distributed matrix multiplication.

### 2.3 Secure Mat-Dot Algorithm

We now introduce the Secure Mat-Dot algorithm, the algorithm used in our implementation. Let \( A \) and \( B \) be two matrices, \( A \) of size \( t \times s \) and \( B \) of size \( s \times r \). We wish to compute the product \( C = AB \), and
we have \( m \) worker nodes at our disposal. However, we need to ensure that up to \( k \) colluding nodes gain no information about either matrix.

The first step is to select \( n \) such that \( 2(n + k) - 1 \leq m \), and split \( A \) and \( B \) into \( n \) submatrices each, as follows:

\[
A = \begin{bmatrix} A_1 & \cdots & A_n \end{bmatrix} \quad \quad B = \begin{bmatrix} B_1 \\ \vdots \\ B_n \end{bmatrix}
\]

Each submatrix \( A_i \) has size \( t \times \frac{n}{s} \), while each submatrix \( B_i \) has size \( \frac{n}{s} \times r \). Assume that \( n \) divides \( s \); if not, we can pad \( A \) with zero columns and \( B \) with zero rows until it does. And observe that \( C = AB = A_1B_1 + \cdots + A_nB_n \). We now encode the \( A_i \) and \( B_i \) using Reed-Solomon codes. We randomly generate matrices \( R_1, \ldots, R_k \) of size \( tx \frac{n}{s} \), and \( S_1, \ldots, S_k \) of size \( \frac{n}{s} \times r \). So the \( A_i \) and \( R_j \) all have the same dimensions, as do the \( B_i \) and \( S_j \). This allows us to define three polynomials:

\[
\begin{align*}
    f(x) &= A_1 + \cdots + A_n x^{n-1} + R_1 x^n + \cdots + R_k x^{n+k-1} \\
    g(x) &= B_n + \cdots + B_1 x^{n-1} + S_1 x^n + \cdots + S_k x^{n+k-1} \\
    h(x) &= f(x) \cdot g(x)
\end{align*}
\]

We can see that \( h \) has degree \( 2(n + k - 1) \) and thus \( \ell = 2(n + k) - 1 \) terms. Furthermore, the \( x^{n-1} \) coefficient of \( h \) is \( A_1B_1 + \cdots + A_nB_n = C \), meaning \( h \) defines a Reed-Solomon encoding of \( C \). So, by computing \( \ell \) points of \( h \), we can interpolate the polynomial and recover \( C \). Since \( \ell = 2(n + k) - 1 \leq m \), we can accomplish this by assigning each of our worker nodes to compute one point of \( h \).

To do so, we generate values \( \alpha_1, \ldots, \alpha_m \) and, for each \( i \in \{1, \ldots, m\} \), compute the matrices \( \tilde{A}_i = f(\alpha_i) \) and \( \tilde{B}_i = g(\alpha_i) \) and send them to node \( i \). After each node receives its matrices, it computes and returns \( \tilde{C}_i = \tilde{A}_i \tilde{B}_i = h(\alpha_i) \). Finally, upon receiving back \( 2(n + k) - 1 \) different points \( (\alpha_i, \tilde{C}_i) \) of \( h \), we interpolate the polynomial \( h(x) = C_1 + C_2 x + \cdots + C_\ell x^{\ell-1} \) and return \( C = C_n \).

### 3 Implementation

In this section, we discuss our specific strategies for implementing the Secure Mat-Dot Algorithm as well as present the pseudo-code for some of the major steps of our program.

#### 3.1 Splitting and Creating Randomized Matrices

The first step in our Secure Distributed Matrix Multiplication program is to either receive or create the original matrices Matrix \( A \) and Matrix \( B \). We save each of these original matrices as a Matrix, which is a uniquely created structure defined with int rows, int cols, and TYPE *data. The next step is to split the original matrices, Matrix \( A \) and Matrix \( B \), into sub-matrices of equal size using block distribution. As specified in the encoding scheme, Matrix \( A \) is to be split along its columns while Matrix \( B \) is to be split along its rows. Therefore, if Matrix \( A \) is size \( t \times s \) and Matrix \( B \) is size \( s \times r \) with \( n \) sub-matrices, then each sub-matrix of Matrix \( A \) would be size \( t \times \frac{n}{s} \) and each sub-matrix of Matrix \( B \) would be size \( \frac{n}{s} \times r \). In order to ensure that all sub-matrices of Matrix \( A \) are the same size and all sub-matrices of Matrix \( B \) are the same size, we initialize their size to the largest possible number of columns and rows, respectively, that could possibly be in a single sub-matrix based on the size of the matrix \( s \) divided by the number of sub-matrices \( n \). In the case that the size of the matrix does not evenly divide by the number of sub-matrices, then the extra row or column in the last few sub-matrices will be filled with zeros.

While we split Matrix \( A \) and Matrix \( B \), we save each separate sub-matrix as a Matrix within an array of type Matrix. For example, once we define the first sub-matrix of Matrix \( A \), then we would save said sub-matrix as a Matrix in \( a_{\text{split}}[1] \), where \( a_{\text{split}} \) is an array of type Matrix and is the same size as the number of sub-matrices. The second sub-matrix would be saved in \( a_{\text{split}}[2] \), and so on. The pseudo-code for all of the above is presented in Algorithms 1 and 2.
Algorithm 1 Split Matrix $A$ by Columns

1: procedure Split By Columns(Matrix $A$, Matrix* $a$ split, numSplits)
2:   $avgCols = get\_Num\_Cols(0)$
3:   for $i$ in numSplits do
4:     $numCols = get\_Num\_Cols(i)$
5:     $offset = get\_Offset(i)$
6:     define\_Matrix(&$a$ split[$i$], $A$.rows, $avgCols$)
7:     for $j$ in $numCols$ do
8:       for $k$ in $A$.rows do
9:         $a$ split[$i$].data[$(k \times avgCols) + j$] = $a$.data[$(k \times A.cols) + j + offset$]

Note: $get\_Num\_Cols(i)$ returns the number for columns in sub-matrix number $i$.
$get\_Offset(i)$ returns the offset from the beginning of the matrix for which sub-matrix number $i$ begins.
$define\_Matrix(Matrix M, t, s)$ allocates enough memory for a $t \times s$ matrix in $M$.data, as well as sets $M$.rows to $t$ and $M$.cols to $s$.

Algorithm 2 Split Matrix $B$ by Rows

1: procedure Split By Rows(Matrix $B$, Matrix* $b$ split, numSplits)
2:   $avgRows = get\_Num\_Rows(0)$
3:   for $i$ in numSplits do
4:     $numData = get\_Num\_Rows(i) \times B$.cols
5:     $offset = get\_Offset(i) \times B$.cols
6:     define\_Matrix(&$b$ split[$i$], $avgRows$, $B$.cols)
7:     for $j$ in $numData$ do
8:       $b$ split[$i$].data[$j$] = $b$.data[$j + offset$]

Note: $get\_Num\_Rows(i)$ returns the number for rows in sub-matrix number $i$.
$get\_Offset(i)$ returns the offset from the beginning of the matrix for which sub-matrix number $i$ begins.
$define\_Matrix(Matrix M, t, s)$ allocates enough space for a $t \times s$ matrix in $M$.data, as well as sets $M$.rows to $t$ and $M$.cols to $s$. 
Once we have split both Matrix $A$ and Matrix $B$ into sub-matrices, we now know the size of all sub-matrices of $A$ and the size of all sub-matrices of $B$. This is important because when using SHMEM, you have to allocate enough space in memory for any value you want to place on different nodes. Since we eventually want to place each of the polynomial evaluations, which will be the same size as the sub-matrices, on their respective node, we need to allocate enough space for said matrix on each node. Therefore, each node needs to know how large each sub-matrix is. In order to distribute this information to each node, we used a `shmembroadcast` to send the number of rows and columns in each sub-matrix of $A$ and the number of rows and columns in each sub-matrix of $B$. Once each node received this information, they `shmемalloc` enough space for the polynomial evaluations $\tilde{A}_i$ and $\tilde{B}_i$ they will eventually be receiving. We are also able to `shmемalloc` enough space on node 0 so that it will be able to accept the answer matrices from all nodes once they finish computing (Matrix* $C$).

The next step is to start encoding our sub-matrices. In order to do this, we first need to create some random matrices to add at the end of the polynomials in order to prevent colluding nodes. The number of random matrices is determined by the number of nodes we want to prevent from colluding. For example, if we added 3 random matrices to each of our polynomials, then 4 nodes would have to collude and share their assigned sub-matrices in order to be able to calculate the original matrices. We start by using `defineMatrix` to create and allocate memory for multiple random matrices. Each random matrix will be the same size as the either the sub-matrices of $A$ or the sub-matrices of $B$. We also seed the random number generator with the time, that way we will get unique values each time we run the program. Then, we go through each of the random matrices and fill them with randomly generated numbers.

### 3.2 Constructing and Evaluating the Polynomials

The next step is to write code that takes in both matrices to be multiplied (Matrix $A$ and Matrix $B$) and to define their terms, coefficients, and exponents within a uniquely created Polynomial structure: containing int terms, Matrix *coeffs, int *exponents. After this, the program will generate an $\alpha_i$ value, evaluate polynomial $f$ at $\alpha_i$ to get $\tilde{A}_i$, send out $\tilde{A}_i$ to node $i$, and then evaluate polynomial $g$ at $\alpha_i$ to get $\tilde{B}_i$, and to send out $\tilde{B}_i$ to the same node $i$. The pseudo-code for the construction and evaluation of the polynomials is presented in Algorithm 3.

**Algorithm 3** Pseudo-code for Constructing and Evaluating the Polynomials within Node 0

1: `procedure CONSTRUCTING AND EVALUATING THE POLYNOMIALS`
2:  
3:  
4:  
5:  
6:  
7:  
8:  
9:  
10:  
11:  
12: 

Note: `me` is defined as `shmем_my_pe[]` meaning this procedure is only run on the main node: node 0.

The first important feature of the algorithmic implementation is that while the split Matrix $A$ is evaluated in the polynomial chronologically (i.e.: $A_0$, $A_1$, ..., $A_n$), Matrix $B$ is evaluated in the polynomial in the opposite direction (i.e.: $B_n$, $B_{n-1}$, ..., $B_0$). There are many ways to address this, but we have elected to make two `make_mat_dot_poly()` functions: one that will place the values– $A_0$, $A_1$, ..., $A_n$; into sequential order within the coefficient array of the Polynomial structure, and one that will place the values– $B_0$, $B_1$, ..., $B_n$; into reverse order within the coefficient array of the Polynomial structure. This way, only one `eval_poly()` function is required to evaluate the polynomials since the coefficients are already placed in the correct order.
Regarding \textit{get-points()}, it is a personal decision how one wants to generate their \( \alpha_i \) values. Randomly generating points is an option, but we decided the simplest way to appoint \( \alpha_i \) values was to give them \( i \) as their value. Because of this, we no longer have a need for the \textit{get-points}(n) function or a need to define \( n \) as \( n_p e s - 1 \), and we can directly input \( i \) in \textit{eval-poly()} for the argument \( \alpha_i \).

When defining the exponent array when constructing the polynomial, it is simply an array of exponents. For example, for an equation of degree 5, the exponent array would be: \( \{0, 1, 2, 3, 4, 5\} \). \textit{eval-poly()} used this array to calculate the value that \( \alpha_i \) should be put to the power to. Calculating the power can be done by computing \( \alpha_i \) to the power of index \( i \) of the exponent array. This proved to be time costly by computing unnecessary modular multiplications. In order to make this more time efficient, an exponent array was decidedly unnecessary. A new array of exponents was created within \textit{eval-poly()}, and a new for loop was created before the for loop used to evaluate the polynomial. This new, preceding for loop was used to perform the \( \alpha_i \) powers that could then be used to multiply with the coefficients to yield the terms of the \( \tilde{A}_i \) polynomial. Essentially, it is now performing the exponent calculations beforehand. For example, in the for loop used to compute the \( \tilde{A}_i \) terms, instead of using an array such as \( \{0, 1, 2, 3, 4, 5\} \) to then compute the powers first, an array of \( \{\langle \alpha_i \rangle^0, \langle \alpha_i \rangle^1, \langle \alpha_i \rangle^2, \langle \alpha_i \rangle^3, \langle \alpha_i \rangle^4, \langle \alpha_i \rangle^5\} \) was used to skip a step and reduce the number of total calculations needed to compute the \( \tilde{A}_i \) terms.

Lastly, to optimize the speed in which \( \tilde{A}_i \) and \( \tilde{B}_i \) are sent to node \( i \), a \textit{shmem-put()} function was placed after each \textit{eval-poly()} so that once any polynomial is evaluated, it will immediately be sent out. It is not required that both \( \tilde{A}_i \) and \( \tilde{B}_i \) are sent at the same time to node \( i \), so sending them separately will speed up the process and hide latency. This is an important distinction so that \( \tilde{A}_i \) doesn’t have to wait for \( \tilde{B}_i \) to finish evaluating before it can be sent to node \( i \) and to start the for loop again to start the next evaluation for \( \tilde{A}_{i+1} \).

The above implementations were made to the code in order to optimize the computations and to make the process of constructing and evaluating the polynomials more efficient.

### 3.3 Distributing and Multiplying Sub-matrices

In order to start the multiplication of matrices on each node, each node needs to know when node 0 has actually placed \( \tilde{A}_i \) and \( \tilde{B}_i \) in their memory. In order to know when the data is available, we implemented a flag variable called \textit{flagStart}. After node 0 has placed both matrices on node \( i \), then node 0 will change the \textit{flagStart} value on node \( i \) to 1. Each node uses \textit{shmem-wait-until()} to know when exactly this value is changed.

Once the flag is changed and the node knows that it has both \( \tilde{A}_i \) and \( \tilde{B}_i \), it starts on the matrix multiplication. Overall, we use naive matrix multiplication in this step. However, in order to make the multiplication more efficient, we transpose the \( \tilde{B}_i \) matrix. This way, we are traversing both matrices in row major order, rather than jumping throughout the matrix, which allows us to use our cache more effectively. The pseudo-code for the matrix multiplication is presented in Algorithm 4.

Now that we have \( \tilde{A}_i \) and \( \tilde{B}_i \) multiplied and saved in Matrix \( \tilde{C} \), we have to put \( \tilde{C} \) back onto node 0 so it can decode and compute the final answer. If you recall back to when we broadcast the lengths of the sub-matrices, after we initialized \( \tilde{A} \) and \( \tilde{B} \) on each node, we also initialized an array of matrices \( C[i] \) that was the size of the number of nodes. Therefore, each node will put \( \tilde{C} \) on \( C[i] \) where \( i \) is the node’s id. This is done using \textit{shmem-put()}.

At this point, we come across two problems. The first is we need to have a way to let node 0 know when a node has completed the multiplication and placed their answer in \( C[i] \). This is quite similar to how we previously had to let node \( i \) know when node 0 had placed \( \tilde{A}_i \) and \( \tilde{B}_i \) in memory, and can be solved in a similar way using flags. The second problem is node 0 needs to know the id of the nodes that have returned. This information is required because we need to know \( \alpha \) in order to interpolate the matrix, and since we set \( \alpha_i \) to the node’s id, we need to know the node’s id. We can solve both of these problems using bitflags. Now, every time a node puts their answer in \( C[i] \), they also use \textit{shmem-atomic-or()} to change \textit{PE-done} by \( 1 \ll i \). Basically, this process lets node \( i \) change the 0 in the \( i \)-th spot of \textit{PE-done} to a 1. For example, if node 3 has put its result matrix on node 0, then it would change \textit{PE-done} from ‘...00000’ to ‘...01000’.

The next process to look at is how node 0 accepts and keeps track of the responses from all of the other nodes. The first point to understand is that node 0 doesn’t need responses from all nodes in order to actually calculate the final result matrix. Since we are using the Secure Mat-Dot Algorithm, we only
need a sub-section of nodes to respond. By choosing to use polynomial codes, the number of nodes which must respond is equal to the number of terms in the final polynomial. The number of terms in the final polynomial can be found by calculating $2 \times (\text{number of terms in } f \text{ and } g) - 1$, where the number of terms in each separate polynomial ($f$ and $g$) is the number of times we split the original matrices ($A$ and $B$) plus the number of random matrices we added to the polynomial. If the number of nodes we have working is greater then the number of terms in the final polynomial, then we only need a sub-section of nodes to respond before continuing on the final calculations, practically eliminating the straggler effect.

Now that we know how many responses we need to wait for, we can go about dealing with the actual responses. Like we used on each node previously, we also utilize `shmem_wait_until()` to know when `PE_done` has been changed by a node and is no longer 0. Once this happens, then we traverse the entire bitflag checking each bit to see if it has been changed to 1. If the $i$th bit has been changed to 1, then we know that node $i$ has finished calculations and put it’s answer in $C[i]$. We also know that said answer matrix has a corresponding $\alpha$ that equals $i$. This is an essential fact and allows us to preform specific calculations that will be explained in the next section. Finally, we subtract $1 \ll i$ from `PE_done` so that we can again `shmem_wait_until()` `PE_done` is not 0. This entire process is repeated until we get the minimum number of responses we need to solve the final matrix, which as we explained above is not always the same as the number of working nodes. The pseudo-code for how node 0 keeps track of node responses, as described above, is presented in Algorithm 5.

Algorithm 5 Wait For and Keep Track of Responses on Node 0

1: **procedure** Track Responses()
2:   if me == 0 then
3:     \textit{FinalTerms} = $2 \times \text{numTerms} - 1$
4:     VdmInverter \^{}inv = vdm\_inv\_init(\textit{FinalTerms}, (\text{npes} - 1))
5:     while inv \rightarrow \text{prog} < inv \rightarrow \text{size} do
6:       \text{shmem\_int64\_wait\_until(} & \text{PE\_done, SHMEM\_CMP\_NE, 0})
7:       for i = 1, i in \text{npes} do
8:         if \text{PE\_done} & (1 \ll i) then
9:           vdm\_inv\_update(inv, i)
10:          \text{PE\_done} -= 1 \ll i

Note: me is defined as `shmem\_my\_pe()`, in other words, me == 0 refers to node 0.

`vdm\_inv\_init(i, j)` initialized the Vandermonde inverse matrix with $i$ being the number of terms in the final polynomial and $j$ being the number of working nodes.

`vdm\_inv\_update(VdmInverter inv, i)` updates the Vandermonde inverse matrix (inv) based on the $\alpha$ ($i$).
3.4 Interpolating the Product Polynomial

The final step in our algorithm is to interpolate the product polynomial \( h \). After evaluating the polynomial at a number of points equal to the number of terms, we interpolate it by solving a system of \( \ell \) linear equations, each of the form

\[
C_1 \alpha_1 + C_2 \alpha_2 + \cdots + C_\ell \alpha_\ell^{\ell-1} = h(\alpha_i),
\]

where \( \ell \), as defined previously, is the number of terms in \( h \). In this case \( C_1, C_2, C_3, \ldots \) are the variables we wish to solve for and \( 1, \alpha_1, \alpha_1^2, \ldots \) are their coefficients. And solving a system of linear equations is as simple as inverting a matrix.

But here we run into a problem. Ideally, we would maximize parallelism by computing this inverse while the worker nodes are busy multiplying their \( A_i \) and \( B_i \). Unfortunately, this matrix depends on the selection of points \( (\alpha_i, h(\alpha_i)) \) we use to interpolate \( h \), which (due to potential stragglers) we can only know once the first \( \ell \) nodes have returned. Thus, if we rely on standard Gaussian elimination, we cannot begin to interpolate this matrix until \( \ell \) nodes have completed.

While we have no way to know a priori the exact composition of our matrix, we do know its general form and can use that to our advantage. Whereas previously we used \( \alpha_i \) to denote the value at which \( f \) and \( g \) were evaluated for node \( i \), we will now use it to denote the value for the \( i \)-th node to return (if all the nodes return in order, these definitions are equivalent). Using this notation, our matrix has the following form, known as a Vandermonde matrix:

\[
V_{\ell} = \begin{bmatrix}
1 & \alpha_1 & \alpha_1^2 & \cdots & \alpha_1^{\ell-1} \\
1 & \alpha_2 & \alpha_2^2 & \cdots & \alpha_2^{\ell-1} \\
& \vdots & \vdots & \ddots & \vdots \\
1 & \alpha_\ell & \alpha_\ell^2 & \cdots & \alpha_\ell^{\ell-1}
\end{bmatrix}
\]

This matrix is invertable if and only if all the \( \alpha_i \) is unique, which in our application is guaranteed to be the case. For all \( j = 1, 2, \ldots, \ell \), let us define the function \( q_{\ell,j} \) as follows:

\[
q_{\ell,j}(x) = \prod_{k=1, k \neq j}^{\ell} (\alpha_k - x)
\]

Note that \( q_{\ell,j} \) is a polynomial of degree \( \ell - 1 \), so for some constants \( p_{n,i,j} \) we have:

\[
q_{\ell,j}(x) = \sum_{i=1}^{\ell} p_{\ell,i,j} x^{i-1}
\]

Now we define the inverse Vandermonde matrix as follows:

\[
V^{-1}_{\ell} = \begin{bmatrix}
p_{\ell,1,1} & p_{\ell,1,2} & \cdots & p_{\ell,1,\ell} \\
p_{\ell,2,1} & p_{\ell,2,2} & \cdots & p_{\ell,2,\ell} \\
& \vdots & \ddots & \vdots \\
p_{\ell,\ell,1} & p_{\ell,\ell,2} & \cdots & p_{\ell,\ell,\ell}
\end{bmatrix}_{\ell \times \ell}
\]

Indeed, every element \( w_{i,j} \) of \( V_{\ell} V^{-1}_{\ell} \) comes out to \( w_{i,j} = \sum_{k=1}^{\ell} \alpha_i^{k-1} \frac{p_{\ell,1,k}}{q_{\ell,j}(\alpha_i)} = \frac{q_{\ell,j}(\alpha_i)}{q_{\ell,j}(\alpha_i)} \). If \( i = j \), then \( w_{i,j} = 1 \); if \( i \neq j \), then \( q_{\ell,j}(\alpha_i) \) must contain a factor \((\alpha_i - \alpha_j)\) and thus \( w_{i,j} = 0 \). This confirms that \( V^{-1}_{\ell} \) as defined is the inverse of \( V_{\ell} \).

Furthermore, we can define recursive formulae for \( p_{\ell,i,j} \) and \( q_{\ell,j}(\alpha_j) \):

\[
q_{\ell,j}(x) = \begin{cases}
1 & \ell = 1 \\
q_{\ell-1,j}(x) & \ell > 1, \ell = j \\
(\alpha_{\ell} - x)q_{\ell-1,j}(x) & \ell > 1, \ell \neq j
\end{cases}
\]

\[
p_{\ell,i,j} = \begin{cases}
1 & \ell = 1, i = 1 \\
0 & \ell = 1, i \neq 1 \\
p_{\ell-1,i,j} & \ell > 1, \ell = j \\
\alpha_{\ell} p_{\ell-1,i,j} - p_{\ell-1,i-1,j} & \ell > 1, \ell \neq j
\end{cases}
\]
The proof of the first flows directly from the definition of $q_{ℓ,j}$. Likewise, the first three cases of the second formula flow directly from the definition of $p_{ℓ,i,j}$. We will use these to prove the final case. First, note that $p_{ℓ,i,j} ≠ 0$ only if $p_{ℓ-1,i,j} ≠ 0$ or $p_{ℓ-1,i-1,j} ≠ 0$, which in turn requires that $1 ≤ i ≤ ℓ$. In particular, we use the fact that $p_{ℓ,0,j} = p_{ℓ,ℓ+1,j} = 0$. Now we expand $q_{ℓ,j}(x)$ for $ℓ > 1, ℓ ≠ j$:

$$
q_{ℓ,j}(x) = (αℓ - x)q_{ℓ-1,i,j}
= (αℓ - x) \sum_{i=1}^{ℓ-1} p_{ℓ-1,i,j}x^{i-1}
= \sum_{i=1}^{ℓ-1} αℓp_{ℓ-1,i,j}x^{i-1} - \sum_{i=1}^{ℓ-1} p_{ℓ-1,i,j}x^{i}
= \sum_{i=1}^{ℓ} αℓp_{ℓ-1,i,j}x^{i-1} - \sum_{i=1}^{ℓ} p_{ℓ-1,i-1,j}x^{i-1}
= \sum_{i=1}^{ℓ} (αℓp_{ℓ-1,i,j} - p_{ℓ-1,i-1,j})x^{i-1}
$$

Therefore $p_{ℓ,i,j} = αℓp_{ℓ-1,i,j} - p_{ℓ-1,i-1,j}$, completing the proof.

These recursive formulae allow us to compute $V_{ℓ}^{-1}$ progressively by initializing a table of $p_{ℓ,i,j}$ and $q_{ℓ,j}$ then updating them according to the formulae whenever a new $α_i$ is received. This is exactly what the functions vdm_inv_init() and vdm_inv_update() do in Algorithm 5. In this way, by the time the $ℓ$-th node returns a result, we have already completed most of the work to invert the Vandermonde matrix.

Now only one step remains: to interpolate $h$ using $V_{ℓ}^{-1}$. Applying to $h$ the matrix form of Reed-Solomon codes, where $\tilde{C}_i = h(α_i)$ represents the $i$-th matrix returned by a worker node, we see that

$$
\begin{bmatrix}
\tilde{C}_1 \\
\vdots \\
\tilde{C}_ℓ
\end{bmatrix}
= V_{ℓ}^{-1}
\begin{bmatrix}
C_1 \\
\vdots \\
C_ℓ
\end{bmatrix}
$$

and therefore

$$
\begin{bmatrix}
\tilde{C}_1 \\
\vdots \\
\tilde{C}_ℓ
\end{bmatrix}
= V_{ℓ}^{-1}
\begin{bmatrix}
C_1 \\
\vdots \\
C_ℓ
\end{bmatrix}
$$

Since our true product matrix $C$ is encoded as the $n$-th coefficient of $h$ (where $n$ is the number of pieces $A$ and $B$ are each split into), it is given as:

$$
C = C_n = \sum_{j=1}^{ℓ} \frac{p_{ℓ,n,j}}{q_{ℓ,j}(α_j)} \tilde{C}_j
$$

We apply this formula element by element to compute $C$.

### 3.5 Computation Over a Prime Field

For both computational and cryptographic reasons, we perform our arithmetic over a finite field rather than the field of real numbers.

- Computationally, the secure mat-dot algorithm implemented using integers or floating point types will quickly meet overflow or precision errors. Consider that we have to evaluate polynomials $f$ and $g$ of degree $n + k - 1$ at a different point for each worker node. If we use $m = 64$ nodes total (setting $α_i = i$), split our matrices into $n = 8$ pieces, and wish to protect against $k = 4$ colluding nodes, then the highest-order terms in $f(α_{63})$ and $g(α_{63})$ will have magnitude $α_{63}^{n+k-1} = 63^{11} ≈ 6.2 \times 10^{19}$ even without its coefficient. By contrast, the maximum value of a 64-bit integer is $9.2 \times 10^{18}$. Thus integer overflow is practically unavoidable. We could avoid overflow by using floating point numbers; here, however, we run the risk of accumulating error from limited precision.
• Cryptographically, finite field arithmetic is necessary for truly secure randomness. Even with floating point numbers, we must bound the elements of $A$ and $B$ to prevent overflow in $C$. While this is the case even for standard matrix multiplication algorithms, our algorithm has the additional constraint that we must bound the elements of our random matrices to prevent overflow in any of the intermediate computations. This bounding costs us some of our security, opening up the possibility that $k$ or fewer colluding nodes could learn about the structure of $A$ or $B$. Finite fields avoid this pitfall by allowing the random matrices to span the whole field, guaranteeing that $k$ or fewer colluding nodes will indeed learn nothing about $A$ or $B$.

While we perform all our computations over a finite field, that does not mean they are only useful for modular arithmetic. With properly bounded matrix values, the matrix product over a finite field will be indistinguishable from the matrix product over the integers themselves. With a large enough field, this bounding will be hardly more restrictive than that required to avoid overflow.

Our implementation uses a Mersenne prime field, the integers mod $p = 2^{31} - 1$, only a factor of two smaller than the range of 32-bit integers. Because of its proximity to $2^{31}$, this prime has an efficient modulus implementation based on bitwise operations. Modulus and division by a large prime are quite difficult to compute; by a large power of two, however, they cost only a single bit shift or mask. Fortunately, we can define the former operations in light of the latter.

For some integer $x$, define $q$ and $r$ as the quotient and remainder of $x$ divided by $p$, and likewise $q'$ and $r'$ as the quotient and remainder of $x$ divided by $p + 1$. That is, $x = pq + r = (p + 1)q' + r'$.

• First, $x + q = (p + 1)q + r$, meaning $r$ is also the remainder of $x + q$ divided by $p + 1$.

• Second, $q' + r' = (p + 1)q' + r' - pq' = x - pq' - p = p(q - q') + r$, meaning $q - q'$ is the quotient when $q' + r'$ is divided by $p$. So if $q' + r' < p$, then $q = q'$; if $p \leq q' + r' < 2p$, then $q = q' + 1$; and so on. Of course, this still requires a division by $p$, exactly what we intended to avoid. But since $r' < p + 1$, we are guaranteed that $q' + r' < 2p$ as long as $q' \leq p$, and this is true for all $x \leq p^2$. And given two operands $y, z < p$, both $y + z$ and $yz$ will be less than $p^2$. Thus we can safely precondition on $x \leq p^2$, and define $q$ as follows:

$$q = \begin{cases} q' & q' + r' < p \\ q' + 1 & q' + r' \geq p \end{cases}$$

Using these identities, we implement our modulus function as follows:

**Algorithm 6 Compute Modulus of $x$ by a Mersenne Prime $p = 2^y - 1$**

1: procedure COMPUTE MODULUS
2:   $q' = x \gg y$
3:   $r' = x \& p$
4:   $q = q' + (q' + r' >= p)$
5:   $r = (x + q) \gg \mu$

Note: we take advantage of the fact that C’s comparison operators evaluate to 0 for false or 1 for true.

4 Results

In this section we will be discussing the various effective and ineffective optimization methods we tested, how we landed on our final program, graphs that demonstrate the scalability of our solution, and how our secure distributed matrix multiplication implementation compares to naive methods (i.e. Strassen’s, naive).

4.1 Optimization Methods

After combining each section of code together and having it run smoothly, we found some limitations in our code: we could not use more than 64 processes, and the prime field had to be greater than the npes.
With this in mind, we began collecting data and testing optimization methods to see if they would speed up, slow down, or have no effect on the computation times. We collected data on the time to run `eval_poly()`, time to run `mm_multi()`, time to interpolate, and total run time. We did this so that we could see which functions were slowest to pin point what we wanted to work on optimizing. We kept a few things constant throughout the trials: the size of the matrix – 2048x2048, the number of splits – 15, the number of prevents – 10, and the number of nodes – 55.

**Base case (in seconds):** Total run time: 207

**Optimization #1:** Paralleling `eval_poly()` with `shm_put()`

We tested to see if running two processes – first being the polynomial evaluation and sending, and second being the polynomial gathering and solving; in parallel would improve the time in which the program was able to run calculations. We used pthreads to achieve this, and we tested the code and varied the number of threads. Unfortunately, with the modifications, the time to run the entire program was over 100 seconds slower, and proved to be an ineffective optimization method. This may be because the constant memory access with parallelizing may have slowed it down. We reverted the code back to its original state and tried the next optimization idea.

Total run time: 350

**Optimization #2:** Interweaving the sending and receiving into one loop

We made an assumption that if we were to combine the send and receive loops (which, at the time, were separate loops), the program would not constantly need to check memory since the receiving would occur directly after the sending, and the computation times would improve. Unfortunately, with the modifications, the time to run the entire program was around 10 seconds slower. The slower runtime may be because we had to incorporate a `shm_wait_until()` for the implementation to work.

Total run time: 217

**Optimization #3:** Optimization level -O3

We reverted the code back to its original state and tried the next optimization idea, which was to compile with a higher optimization level. When it came to deciding which optimization level to try, we found that -O3 was best for our purposes since it would lead to faster performance. Fortunately, with the modifications, the time to run the entire program was significantly faster, and ran around 140 seconds faster.

Total run time (with -O3): 71

**Optimization #4:** Pre-calculating the $\alpha_i$ powers

As discussed in Section 3.2, but to reiterate, a new array of exponents was created within `eval_poly()`, and a new for loop was created before the for loop used to evaluate the polynomial. The new, preceding for loop performed the $\alpha_i$ powers that could then be used to multiply with the coefficients to yield the terms of the $\tilde{A}_i$ polynomial. Essentially, `eval_poly()` is now performing the exponent calculations beforehand. Fortunately, the runtime was 4 times faster. This proved to be an incredibly effective optimization method. We will keep this code and try the next optimization idea.

Total run time (with -O3): 21 || Total run time (without -O3): 54

**Optimization #5:** Optimizing interpolation

Within `interpolate_matrix.c`, we improved how we gather shares of elements and removed unnecessary computations when it came to computing elements. We also removed an array and had a file copy directly from the source so that there was less copying of data. We also improved `vandermonde_inverter.c` by combining numerators and denominators when returning so that there would not be any repeated powers and fewer multiplications. All of these small edits to optimize the interpolation resulted in a run time comparable to how it was previously with only minor improvements. We will keep this code since it cut a second off our total run time without -O3, and try the next optimization idea.

Total run time (with -O3): 21 || Total run time (without -O3): 53

**Optimization #6.1:** Block matrices

We implemented blocking in place of our naive matrix multiplication function because blocking is a way
to improve the cache use, and we saw improvements in our run time. We also tested different blocking sizes and found that the block size of 16 yielded the best results.

Table 1: Optimization Methods

<table>
<thead>
<tr>
<th>Optimization Methods</th>
<th>Total Run Time</th>
<th>Total Run Time (w/ -O3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>#0: Base case</td>
<td>207s</td>
<td>—</td>
</tr>
<tr>
<td>#1: Paralleling eval_poly() with shmem_put()</td>
<td>350s</td>
<td>—</td>
</tr>
<tr>
<td>#2: Interweaving the sending and receiving into one loop</td>
<td>217s</td>
<td>—</td>
</tr>
<tr>
<td>#3: Optimization level -O3</td>
<td>—</td>
<td>71s</td>
</tr>
<tr>
<td>#4: Pre-calculting the α_i powers</td>
<td>54s</td>
<td>21s</td>
</tr>
<tr>
<td>#5: Optimizing interpolation</td>
<td>53s</td>
<td>21s</td>
</tr>
<tr>
<td>#6.1: Block matrices</td>
<td>52s</td>
<td>18s</td>
</tr>
<tr>
<td>#6.2: Block matrices (extra sum step)</td>
<td>50s</td>
<td>17s</td>
</tr>
<tr>
<td>#7: Transposing</td>
<td>46s</td>
<td>16s</td>
</tr>
</tbody>
</table>

Table 1 presents a table that summarizes the findings from Section 4.1.

4.2 Graphs

After optimizations #6.2 and #7, we landed on two possible final solutions. We first compared these solutions to see which was faster by making plots comparing their total run time data. The plots are in Figure 1 and Figure 2. In Figure 1, we kept nodes, splits, and prevents constant, and as we expected, the transpose solution performed better with a consistently faster run time for each matrix dimension. We also graphed time vs. dimension at different numbers of splits in Figure 2, and our transpose solution had a faster run time for each level of splits. The next goal was to test the scalability\(^1\) of our transpose solution at various levels of security. Thus, we are testing how the code will perform at varying sizes of matrices, while also considering how the number of splits, prevents, and npes affect the outcome. For the graphs, we are exclusively comparing the total run time data points with -O3.

In Figure 3 we have a Time vs. Nodes graph where the number of splits and prevents is constant, and each line represents a varying matrix dimension. The lines are constant, demonstrating that the number of nodes used has a trivial effect on the total run time, which makes sense since we do not have to wait for all of the nodes to respond before proceeding with our computations.

\(^1\)A system is considered scalable when it doesn’t need to be redesigned to maintain effective performance during or after a steep increase in workload. See reference [10] for more information.
Figure 1: Time vs. Dimension (N = 16, S = 2, P = 2)

Figure 2: Time vs. Dimension (N = 64, P = 8)
In Figure 4 we have a Time vs. Prevents graph where the matrix dimension and number of nodes are constant, and each line represents a different number of splits. The lines are linear, and demonstrate that as the number of prevents increase for each number of splits, the total run time also increases.

In Figure 5 we have a Time vs. Dimensions graph where the number of nodes and prevents are constant, and each line represents a different number of splits. The lines are increasing exponentially, demonstrating how as the matrix dimensions get larger, so does the total run time. This makes sense since each succeeding point represents a total run time for a matrix four times larger than the previous, and the total run time is also four times slower.

In Figure 6 we have a Time vs. Splits graph where the matrix dimension and number of nodes are constant, and each line represents a different number of prevents. The lines are logarithmic, demonstrating that as the number of splits increase, the total run time decreases.

4.3 Comparison to a Single-Node Method

As a benchmark, we will compare our code to a single-node naive and single-node Strassen’s algorithm implementation. These implementations are single-node in order to maintain security. We expect our MatDot code to run slower than the Strassen’s implementation since our code has the extra steps to make it secure and distributed, and we would like to see the extent that the distributed aspect affects total run time. However, surprisingly, but thankfully, when comparing our transpose and blocking solutions with the single-node Strassen’s and single-node naive implementations, while comparing data at varying number of nodes, splits, prevents, and matrix sizes, our transpose and blocking solutions consistently performed significantly better than the Strassen’s and naive implementations (with the transpose solution performing the best overall). An instance of this is presented in Figure 7 where we have a Time vs. Dimensions graph where the number of nodes is 64, number of splits is 8, and number of prevents is 2. The only time our Mat Dot Algorithm implementations did not perform as well as the single-node implementations was when the number of splits was 2, but it is important to keep in mind that this is also the worst case scenario.

5 Conclusion and Future Work

There are many possible ways to improve and optimize our existing code. One option is we can optimize our matrix multiplication, the \texttt{mm\_multi()} function. Currently, we are implementing a transpose
Figure 4: Time vs. Prevents (sdmm transpose, D = 2048, N = 64)

Figure 5: Time vs. Dimensions (sdmm transpose, N = 64, P = 8)
Figure 6: Time vs. Splits (sdmm_transpose, D = 2048, N = 64)

Figure 7: Time vs. Dimensions (N = 64, S = 8, P = 2)
matrix multiplication, and while this is quite efficient, there are other options that could possibly improve performance even more. For example, utilizing multi-threading or other algorithmic optimizations such as Strassen’s algorithm\(^2\). Another option to improve our code would be using pthreads in the `eval_poly()` function. While implementing pthreads hasn’t helped in previous optimizations, by running more tests we may be able to find a strategic way of using the pthreads, such as in the `eval_poly()` function, that would speed up the program. An additional way we could decrease our programs run-time would be to remove the modular arithmetic. Since we are using fields in our algorithm, we end up doing a lot of of modular operations, which are extremely slow. If we could decrease or remove these operations, that would have a good chance of speeding up our program. Some possible ways to do this would be to expand the field to all real numbers or implement a Galois field.

There are also a lot of other options to explore in regards to the algorithmic choices we made in our implementation. First, we could test other methods in place of the Vandermonde matrix, such as Lagrange’s method. We could also look into other types of systems for parallelism in place of SHMEM, such as OpenMP. However, a major avenue of exploration would be testing different algorithmic schemes for secure and distributed matrix multiplication. In this implementation, we used the Secure Mat-Dot algorithm, but there are many other algorithms out there. For example, the GASP code \([3]\), which requires a much lower response rate from all the worker nodes in order to complete calculations. References to other new methods to explore include: \([4]\), \([5]\). Overall, while our implementation works very well in it’s intended purpose, there are still multiple improvements that we can consider, as well as whole new avenues of exploration in which we can use this program as a stepping stone.

Overall, through this project we have shown that a practical implementation of secure distributed matrix multiplication using polynomial codes and a shared memory system is not only feasible, but more efficient than comparative single-node options. We have also conducted multiple rounds of optimizations and performance analysis on our implementation, leading us to a solution with minimal run-time. These results open the door to further exploration into using polynomial codes within secure distributed matrix multiplication since we know they are both effective and efficient.

6 Acknowledgements

we ❤ fiona + fesi

References


\(^2\)A matrix multiplication algorithm that is faster and has better asymptotic complexity than the standard matrix multiplication algorithm for large matrices. Although there are better algorithms out there for extremely large matrices, Strassen’s is one of the fastest for matrices of practical sizes. See reference \([9]\) for more information.


