Multi-threaded Matrix Multiplication

Gracie Bliss
College of Engineering and
Computer Science
The University of Central Florida
Orlando, Florida
Email: graciebliss@knights.ucf.edu

Jose Valencia
College of Engineering and
Computer Science
The University of Central Florida
Orlando, Florida
Email: josevalencia@knights.ucf.edu

***Abstract*—Matrices were first invented in the 1850s by Andrew Cayley, with the term first being introduced by James Sylvester. Since then matrices and matrix multiplication has been used in other forms of mathematics and in many professions. They have been used to make graphs, conduct studies, and calculate statistics, which can aid in almost all jobs. This means that matrices are a very important and integral part of mathematics and matrix multiplication is a necessity. Our project explores the idea that matrix multiplication is a task that can be parallelized, which will result in faster run times. Through two multi-threaded functions, a comparison will take place to investigate the most efficient way of multiplying matrices at all different sizes of matrices. Four functions were created and tested against each other, and the resulting times, in seconds, were compared to establish which would be the overall best and fastest method.**

I. Introduction

Matrix multiplication, which was founded by Andrew Cayley in the 1850s, has been established as the most important matrix operation due to it being commonly used to solve or approximate solutions across a wide range of problems and fields (1). It can be used to create graphs, calculate statistics, and conduct studies and research. Population growth and mortality rate are examples of statistics that can be represented by matrices and calculated using the help of matrix multiplication. This study aims to increase the efficiency of existing matrix multiplication algorithms by adapting them safely to a multi-threaded architecture, where tasks are run parallel to each other instead of in a linear fashion. The usual method of matrix multiplication relies on the dot product operation to multiply various combinations of rows and columns. The calculations involved in this process are well suited for parallelization due to the lack of a need to modify shared resources. In other words, each dot product operation works independently of the other, which allows most of the process to be parallelized, resulting in a significant speed-up. The program will use already established alternative algorithms like the Strassen algorithm and compare them to the method of parallelizing the dot product operations in the usual matrix multiplication method. The results in this paper will demonstrate the dot product algorithm parallelized and benchmark them against other methods and its own linear counterpart. The idea here is that since alternative methods of multiplying matrices are only slightly more efficient, parallelizing the dot product method well enough should result in a faster time than these alternative methods up to a certain large size of matrices.

The goal of the research is to investigate the different algorithms and methods that can be used to multiply matrices and to determine which method is ultimately the fastest-running method available and if it was possible to make the already established methods faster. The goal is also to determine what size matrices being multiplied would lead to the specific methods being the fastest and to analyze why.

II. Problem Statement

The current algorithms to solve matrix multiplication run at O(n^3), with the Strassen algorithm running at O(n^{log7}) at best, making it slightly more efficient than the normal dot product method. Both of these time complexities ultimately, for a single-threaded algorithm, lead to a long run time when working with large matrices, and with the Strassen method producing long run times for smaller matrices as well. Because matrix multiplication is a necessity for so many computations, making a faster algorithm would be beneficial. With this research, we do not intend to create new algorithms that deal with matrix multiplication in better time complexity, but instead, change the implementation of the dot product algorithm from linear to parallel in the most efficient way possible, through two different methods, which would theoretically cause a speed up in the running time and then compare these methods to see if it is faster than using a slightly more efficient method like the Strassen algorithm in a linear fashion. Also, if the dot product method being parallelized results faster than a method like Strassen or divide and conquer, up to what size of matrices would it be faster?

The two algorithms that we will be creating as a result of the additions we are adding to the algorithms that we already know work will be one that we call the multi-threaded algorithm, in which there are multiple threads handling the dot products, and one we call the CPU-threaded algorithm, in which there are multiple threads to reflect the number of cores available.

III. Related Works

The standard method for multiplying two matrices is to multiply each element in a row of the first matrix by the corresponding one in the second matrix. The products are then added to get the final matrix. This process would be repeated for every combination of row and column in the matrices. It would then result in a matrix that is the same size as the original matrices. This algorithm has a time complexity of O(n^3) with n being the size of the matrix. This is because of the required nested loops, making the standard method more inefficient for larger matrices.

The Strassen method was first developed in the 1960s. The algorithm multiplies two matrices of the same size in a more efficient way than the standard algorithm. It divides each matrix into four matrices and computes the products of these recursively. This method reduces the number of multiplications required from seven to eight and brings the time complexity down below O(n^3). This method, however, can be less efficient for smaller matrices. (2)

Parallelization is when a process is divided into smaller tasks so those tasks can be executed simultaneously using multiple cores. Parallelization is most commonly used to speed up complex computations or the process of processing data. The applications parallelization is usually used in would be artificial intelligence, or machine learning, data processing, image processing, video processing, and high-performance computing. Real-world applications could be weather forecasting, video compression, and computer graphics.

Multi-threading is another technique where multiple threads are performing a process at the same time using multiple cores. These techniques can increase the performance of programs in some cases but can also be expensive to implement. Multi-threading is usually used for programs with many I/O operations. Some examples would be graphic user interface applications, web applications, computational applications, and network applications. These are usually used to make sure user interfaces stay responsive to keep up with the operations taking place or so that there can be a thread for each user or user request.

IV. Methodology

As mentioned before, the usual dot product method of matrix multiplication is already well suited to be adapted to a multi-threaded environment. This section will go more in-depth about why this is true and how exactly is this method going to be parallelized and compared to alternative methods.

First note that for matrix multiplication to work with the usual dot product method, the number of columns in the first column must equal the number of rows in the second one. If the two matrices being multiplied satisfy this requirement, we can multiply the matrices by taking the dot product of each horizontal vector in the first matrix with each vertical vector on the second matrix. To find the dot product, a row in the first matrix will be multiplied with the corresponding column of the second matrix, then will be added together, with the result being placed in the corresponding element in the result matrix. For example, if we have a 3x3 for matrix A, and a 3x3 for matrix be, to find the dot product to place in the resulting matrix, C, C[0][0]=A[0][0]*B[0][0]+A[0][1]*B[1][0]+A[0][2]*B[2][0]. The results of these operations are then arranged in a result matrix. If the first matrix has dimensions of $m \times n$ and the second matrix has dimensions of $n \times p$, then the resulting matrix will be of size $m \times p$.

This method can be easily parallelized by calculating each of these dot product operations on separate threads. Although the two initial matrices are a shared resource, the threads will only be reading from them so there is no problem concerning the sharing of the resources, and therefore the parallelization will present so further complications. The resultant matrix is also a shared resource, but when using an array each individual element is safe to be accessed by different threads at the same time, which is exactly the case we run into when each thread records the result of its operation to the result matrix. Therefore, by running each thread parallel to each other, where every thread, is solving a dot product of the elements, the algorithm should be able to complete in a faster time than the average dot product method.

Another method of parallelization that we will be testing is a CPU-threaded method. This method, in contrast to the multi-threaded method, which creates a thread for every computation to fill the result matrix, will create a thread for each core in the CPU and divide the work evenly between those threads, allowing the computer to not become overwhelmed with threads while still running parts of the program in parallel. There are two reasons for creating this method. First, the operating system will not allow the program to create a huge number of threads when working with big matrices and attempting to create one thread per dot product operation. This method resolves this issue because it creates a set number of threads, although they will have more work to do linearly than just one dot product operation. The second reason for this method is that even if the operating system creates all the threads needed, creating and starting threads is an expensive operation, so creating a set amount of threads can keep that overhead controlled, while also gaining the advantage of not running all processes in a linear fashion, preventing an unnecessarily long run time. Once more threads than the amount the CPU has physically available are created, the threads aren't actually running in parallel but instead, the scheduler just alternates them however it sees fit, which could cause the algorithm to run inefficiently. All these factors combined might make CPU-threaded the better multi-threaded matrix multiplication implementation.

```
vector<vector<int>> dot_matrix_mult(vector<vector<int>>& matA, vector<vector<int>>& matB) {
    int rowA = matA.size();
    int colA = matA[0].size();
    int rowB = matB.size();
    int colB = matB[0].size();
    vector<vector<int>> result(rowA, vector<int>(colB, 0));

    for(int i = 0; i < rowA; i++)
    {
        for(int j = 0; j < colB; j++)
        {
            for(int k = 0; k < colA; k++)
            {
                result[i][j] += matA[i][k] * matB[k][j];
            }
        }
    }

    return result;
}
```

Fig. 1. The Dot Product function

With these two new parallel methods of running the dot product algorithm working, it is important to explain how they will be compared to another alternative, slightly more efficient, the Strassen method. This method works by splitting big matrices into smaller ones recursively. The Strassen method also has its own set of formulas for dealing with multiplying the smaller matrices and rejoining them into the result matrix. However, this method only works correctly when the input matrices are of dimensions that are a power of two. Because of this when comparing our parallelized dot product method to this other alternate method, we will work with matrices that fulfill this requirement. Also, as the main idea in this alternate method is to split bigger matrices into small ones, we will only be testing with matrices bigger than 2×2.

The two methods that we have predicted will be faster than the Strassen and the dot product methods will be run alongside them for multiple different sizes of matrices. Each matrix size will be tested multiple times in order to be able to calculate an average, to avoid any bias in the results of the run times. From there, the average run times will be compared to each other and will reveal which method ran the fastest for each matrix size.

The development of the methods that were used for the comparison depended on the algorithms that were already established, available, and tested. The dot product algorithm that is reflected in our dot product function is the basic algorithm that is most commonly used, and most commonly found, when doing any kind of research on how to multiply a set of matrices. It covers the standard practice of taking each dot product individually, one at a time, and placing them into a resulting matrix. We used this algorithm and developed our working dot product algorithm as seen in Figure 1.

The next algorithm we wrote for the comparison was the

```
vector<vector<int>> strassen_matrix_mult(vector<vector<int>> A, vector<vector<int>> B) {
  int n = A.size();
  vector<vector<int>> C(n, vector<int>(n));

  if (n == 1) {
    C[0][0] = A[0][0] * B[0][0];
    return C;
  }

  int m = n / 2;

  vector<vector<int>> A11(m, vector<int>(m)), A12(m, vector<int>(m)), A21(m, vector<int>(m)), A22(m, vector<int>(m));
  vector<vector<int>> B11(m, vector<int>(m)), B12(m, vector<int>(m)), B21(m, vector<int>(m)), B22(m, vector<int>(m));

  for (int i = 0; i < m; i++) {
    for (int j = 0; j < m; j++) {
      A11[i][j] = A[i][j];
      A12[i][j] = A[i][j + m];
      A21[i][j] = A[i + m][j];
      A22[i][j] = A[i + m][j + m];

      B11[i][j] = B[i][j];
      B12[i][j] = B[i][j + m];
      B21[i][j] = B[i + m][j];
      B22[i][j] = B[i + m][j + m];
    }
  }

  vector<vector<int>> M1 = strassen_matrix_mult(A11, matrix_sub(B12, B22));
  vector<vector<int>> M2 = strassen_matrix_mult(matrix_add(A11, A12), B22);
  vector<vector<int>> M3 = strassen_matrix_mult(matrix_add(A21, A22), B11);
  vector<vector<int>> M4 = strassen_matrix_mult(A22, matrix_sub(B21, B11));
  vector<vector<int>> M5 = strassen_matrix_mult(matrix_add(A11, A22), matrix_add(B11, B22));
  vector<vector<int>> M6 = strassen_matrix_mult(matrix_sub(A12, A22), matrix_add(B21, B22));
  vector<vector<int>> M7 = strassen_matrix_mult(matrix_sub(A11, A21), matrix_add(B11, B12));

  vector<vector<int>> C11 = matrix_add(matrix_sub(matrix_add(M5, M4), M2), M6);
  vector<vector<int>> C12 = matrix_add(M1, M2);
  vector<vector<int>> C21 = matrix_add(M3, M4);
  vector<vector<int>> C22 = matrix_sub(matrix_sub(matrix_add(M5, M1), M3), M7);

  for (int i = 0; i < m; i++) {
    for (int j = 0; j < m; j++) {
      C[i][j] = C11[i][j];
      C[i][j + m] = C12[i][j];
      C[i + m][j] = C21[i][j];
      C[i + m][j + m] = C22[i][j];
    }
  }

  return C;
}
```

Fig. 2. The Strassen function

Strassen algorithm, which was completely based off of the Strassen algorithm. This algorithm can easily be found through any research on the current most efficient way of multiplying matrices. This one was once again written in a linear style as that is how the Strassen algorithm is described in the research. Although this algorithm is more efficient than the dot product, it has to use multiple for loops in order to reduce the number of operations by one. This algorithm was for complicated to write than the straightforward dot product algorithm and can be seen in Figure 2.

The next two methods that were written were the actual methods that we developed. The process of developing the multi-threaded algorithm came from our research which proved that although the current algorithms ran with a single thread, we believe that adding multiple threads would make the algorithms faster. In this method, we changed our dot product functions to add a thread for every dot product that needed to be calculated so that all the calculations could be done simultaneously. This function can be seen in Figure 3.

The last function we developed was the CPU-threaded function. In the process of developing the multi-threaded function, there was a realization that as the matrices grew bigger, creating so many threads would be too expensive and therefore eventually make the program run slower rather than faster. With this in mind, we changed the multi-threaded function to create as many threads as we had cores and then added a division of the work to split evenly between the cores. The idea was that although there would be fewer threads, this would ultimately be less expensive to run and would still allow for some parallelization to occur, maintaining a speed-up of some kind. This function can be seen in Figure 4.

```
void mutlithread_matrix_mult_function(vector<vector<int>>& A, vector<vector<int>>& B, vector<vector<int>>& C, int row, int col, int n) {
    int result = 0;
    for (int i = 0; i < n; i++) {
        result += A[row][i] * B[i][col];
    }
    C[row][col] = result;
}

vector<vector<int>> multithread_matrix_mult(vector<vector<int>>& A, vector<vector<int>>& B) {
    int m = A.size();
    int n = A[0].size();
    int p = B[0].size();

    vector<vector<int>> C(m, vector<int>(p));

    vector<thread> threads;

    for (int i = 0; i < m; i++) {
        for (int j = 0; j < p; j++) {
            threads.emplace_back(mutlithread_matrix_mult_function, ref(A), ref(B), ref(C), i, j, n);
        }
    }

    for (auto& t : threads) {
        t.join();
    }

    return C;
}
```

Fig. 3. The multi-threaded function

```
vector<vector<int>> strassen_matrix_mult(vector<vector<int>> A, vector<vector<int>> B) {
  int n = A.size();
  vector<vector<int>> C(n, vector<int>(n));

  if (n == 1) {
    C[0][0] = A[0][0] * B[0][0];
    return C;
  }

  int m = n / 2;

  vector<vector<int>> A11(m, vector<int>(m)), A12(m, vector<int>(m)), A21(m, vector<int>(m)), A22(m, vector<int>(m));
  vector<vector<int>> B11(m, vector<int>(m)), B12(m, vector<int>(m)), B21(m, vector<int>(m)), B22(m, vector<int>(m));

  for (int i = 0; i < m; i++) {
    for (int j = 0; j < m; j++) {
      A11[i][j] = A[i][j];
      A12[i][j] = A[i][j + m];
      A21[i][j] = A[i + m][j];
      A22[i][j] = A[i + m][j + m];

      B11[i][j] = B[i][j];
      B12[i][j] = B[i][j + m];
      B21[i][j] = B[i + m][j];
      B22[i][j] = B[i + m][j + m];
    }
  }

  vector<vector<int>> M1 = strassen_matrix_mult(A11, matrix_sub(B12, B22));
  vector<vector<int>> M2 = strassen_matrix_mult(matrix_add(A11, A12), B22);
  vector<vector<int>> M3 = strassen_matrix_mult(matrix_add(A21, A22), B11);
  vector<vector<int>> M4 = strassen_matrix_mult(A22, matrix_sub(B21, B11));
  vector<vector<int>> M5 = strassen_matrix_mult(matrix_add(A11, A22), matrix_add(B11, B22));
  vector<vector<int>> M6 = strassen_matrix_mult(matrix_sub(A12, A22), matrix_add(B21, B22));
  vector<vector<int>> M7 = strassen_matrix_mult(matrix_sub(A11, A21), matrix_add(B11, B12));

  vector<vector<int>> C11 = matrix_add(matrix_sub(matrix_add(M5, M4), M2), M6);
  vector<vector<int>> C12 = matrix_add(M1, M2);
  vector<vector<int>> C21 = matrix_add(M3, M4);
  vector<vector<int>> C22 = matrix_sub(matrix_sub(matrix_add(M5, M1), M3), M7);

  for (int i = 0; i < m; i++) {
    for (int j = 0; j < m; j++) {
      c[i][j] = C11[i][j];
      c[i][j + m] = C12[i][j];
      c[i + m][j] = C21[i][j];
      c[i + m][j + m] = C22[i][j];
    }
  }

  return C;
}
```

Fig. 4. The CPU-threaded function

V. Evaluation

Our four methods consist of the standard dot product method, the Strassen method, our multi-threaded method, and our CPU-threaded method. All methods were tested over various size matrices, ranging from 8 to 2048 in powers of two, with each method being tested ten times. From those ten trials, the average run times were taken for each matrix size, for each method. For the eight-by-eight matrices, the dot product method averaged $2.1801e^{-}5$, seconds while the CPU threaded method averaged .00096067 seconds. The Strassen method and the multi-threaded method averaged in the .002 second range together, with the multi-threaded method being faster by .006 seconds. For the 16 by 16 matrices, the method becomes faster than the Strassen method with an average of .00565943 seconds while Strassen is an average of .01448043 seconds. The dot product was still the fastest

Type	8	16	32	64	128	256	512	1024	2048
Dot Product	2.18006E-05	8.40793E-05	0.000435	0.002314	0.017076	0.138436	1.051102	8.824688	80.08939
Strassen	0.002969156	0.01448043	0.069885	0.38777	2.714752	19.00486	126.74	883.39	6152.95
Multithreaded	0.002304179	0.005639428	0.015416	0.061	0.244782	x	x	x	x
CPU Threaded	0.000960668	0.000365855	0.000393	0.001077	0.006421	0.049696	0.387749	3.30461	28.47453

Fig. 5. The average run times of all the methods

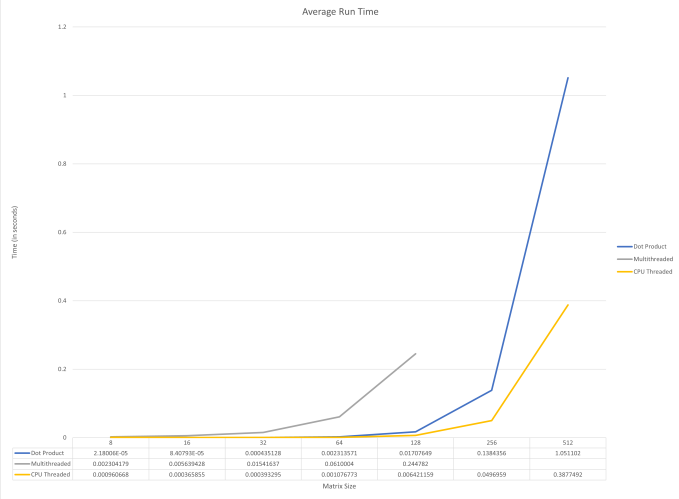

Fig. 6. The average run times of the dot product method, the multi-threaded method, and the CPU threaded method

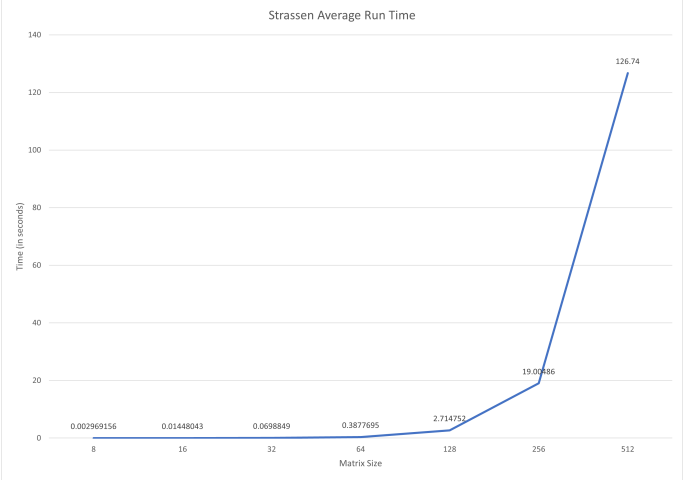

Fig. 7. The average run times of the Strassen method

method at $8.4079e^{-}5$ seconds and the CPU-threaded method was the second fastest with .00036585 seconds. The CPU-threaded method becomes the fastest to multiply the matrices when the matrices become 32 by 32. It averages .000393 seconds with the dot product method averaging .000435 seconds. The Strassen and multi-threaded methods stay significantly slower at .069885 seconds and .015416 seconds, respectively. When the matrices were set to 256 by 256, the multi-threaded method caused the program to abort and had to be removed. The Strassen method run time skyrocketed to an average of 19.00486 seconds and continued to skyrocket through the rest of the trials, eventually ending on an average of 6152.95 seconds for the 2048 by 2048 matrices. The CPU-threaded method stayed the fastest through the rest of the trials and ended with an average of 28.47453 seconds in the 2048 by 2048 matrices. All the averages can be seen in Figure 5.

The averages for the dot product method, the multi-threaded method, and the CPU-threaded method can be seen, in comparison, in Figure 6. The CPU-threaded method stays faster for most of the time. The Strassen method had such high run times the numbers could not be compared to the others in the same graph. The average run times for the Strassen method can be seen in Figure 7.

Overall, most of the methods worked as expected, and all of the methods worked as expected for the smaller matrices. The only two issues that presented themselves through the testing were that the multi-threaded method could not handle bigger matrices and that the Strassen method did not get faster than the other methods at any point.

VI. Discussion

The initial development of the first two processes, the dot product algorithm, and the Strassen algorithm went as expected. Since we used an already-established mathematical algorithm for both, in which we used the research as a form of pseudocode for our code, the development was straightforward and left us confident in our programs. The development of our tested program, however, was more challenging. The initial thought was to simply have one tested algorithm, the multi-threaded algorithm. The hypothesis was that simply adding threads would create a faster run time, which in theory should be accurate as doing multiple calculations at once would make for a faster program. However, upon further thought, we realized that it was very likely that the expense of creating so many threads may not result in the most balanced cost-benefit relationship. It became clear to us that there was a very real chance the cost would be too high to justify creating the threads and ultimately the program could not get any faster and would, in fact, get significantly slower. This led to the development of our CPU-threaded algorithm, which we felt was the best solution for our problem. Not only would it create a more balanced relationship between the cost of creating the threads and the benefit of parallelization, but it would also work for almost all matrix sizes, rather than only working for medium-sized matrices. This addition proved to be a very beneficial addition to our research, as everything that we predicted would happen proved correct.

Our prediction at the beginning, before we tested, was that for the very small matrices, the cost of creating threads would be too high, and therefore the basic dot product method would be the fastest for the first few matrix sizes. After that, we predicted that the Strassen method would become the fastest or that the multi-threaded would be the fastest. The idea was that both were made for larger matrices and would cause them to take over as the fastest method as the benefit of parallelization would start to outweigh the cost of creating threads, although we did consider that this could only apply to the medium-sized matrices as too many threads would be a problem for the CPU to handle. We did think that the Strassen method would become faster before the CPU-threaded method, although we acknowledged that it could have gone either way. Lastly, We predicted that the CPU-threaded would become the overall fastest in the end because the creation of so many threads could be not justified by the pure amount of calculations that could be done at that same time. If all threads could be done simultaneously or scheduled through a scheduler, the most efficient version of the program could be the result of the CPU-threaded method making this method also the fastest.

Our initial hypothesis that the standard dot product method of multiplying matrices could be made faster through parallelization turned out to be correct. Although the multi-threaded method

never became faster than the others, it is possible that it eventually would have; however, that could require a computer that could create an infinite amount of threads without crashing, which was not possible in our testing. From the numbers we gathered, while we were able to test the multi-threaded algorithm, it could have possibly been in the competition to be a contender for the fastest algorithms as the matrices got bigger. Unfortunately, the multi-threaded, although looks like it's slowing down slower, was still slower in the 128 x 128 matrix multiplication, which was not expected, however, is probably a result of us underestimating the overhead that would come from the creation of all the threads. The expected that it would be faster at some point but we also did not expect the overhead to be as high as it was, if we had correctly predicted the overhead, we would have predicted that the multi-threaded algorithm would stay slower for longer. The number seemed to get faster at a good rate which could lead to the prediction that if we had no limitations, it would possibly have been the fastest, beating out the CPU-threaded. This makes sense as the creation of the CPU-threaded algorithm was in case of an issue like the multi-threaded algorithm not being able to handle the number of threads necessary. The CPU-threaded method ended up being the fastest method for bigger matrices. As predicted, the dot product method was faster for the smaller matrices, the 8 by 8 matrices, and the 16 by 16 matrices. It can also be assumed that it would be faster for smaller matrices. This is expected as creating threads can be expensive and unnecessary if there is a small number of computations to compute. The Strassen method, theoretically, would have been one of the fastest as the matrices got bigger, as it is a more efficient model of matrix multiplication; however, Strassen never became the fastest and, in fact, was the least efficient of all the methods we tested, at every size of the matrices that we tested. Although it was surprising that it was always less efficient than the methods with multiple threads, as we expected the creation of threads to make both the multi-threaded and the CPU-threaded methods slower, it is understandable that Strassen would be slower as it has multiple nested for loops. We knew the for loops could take up time to run through and therefore it was not completely unexpected for the Strassen method to not be the fastest method. Between the multi-threaded method and the CPU-threaded method, we expected that the CPU-threaded method would become the fastest in the end; however, we thought that the multi-threaded would be the fastest for the medium-sized matrices, as not too many threads would be created that the CPU couldn't handle it but enough would be made that each operation could have its own thread. This never became the fastest, but it possibly could have been if it was possible to create an unlimited number of threads.

Although our experiment turned out mostly how we expected, there are a few improvements we can recommend for research in this same area in the future. First, for future research, we would not recommend testing against the Strassen model. Although this is a popular model and is commonly used, it was nowhere near the other methods in terms of run time and also felt like an unnecessary comparison in the long run. Although it is possible that the Strassen model was slower due to our implementation, overall our implementation follows the basic Strassen model. The dot product model is the most widely used and therefore is the most important model to use as a comparison anyways. Not using the Strassen model also opens up more possibilities for the sizes of the matrices. Since the Strassen model was the method that required the size to be a power of two we were limited in what sizes we could choose. Having more options would have meant more tests that could have been conducted that would have included the multi-threaded method. This would have been beneficial to the research, however, for our purposes, we felt having the Strassen model in the comparisons was more beneficial to our study as it gave a better idea of the efficiency of each method. Next, we would recommend creating different methods to test. If a new method could be tested there is a greater possibility for a better understanding of what could make the multiplication of matrices more efficient. Discovering new ideas for algorithms or testing and changing already established algorithms could lead to more research, which could inspire others to conduct other research. Next, we would recommend trying to test the multi-threaded method with a better computer than we did. If there had been fewer limitations, based on the graph, the multi-threaded got slower at a slower rate than the rest of the algorithm which leads us to believe if a tester with a powerful computer, the multi-threaded method could have done really well in the end. Lastly, we would recommend trying our CPU-threaded algorithm with a computer that has a different number of cores.

Ultimately, the research and experiment both went well and were beneficial. We got a lot of results that we had expected to get and were not disappointed with the overall results. Although not all results were as we expected, upon further reflection it is very easy to understand why we got the results that we did. With more research, a very efficient method of multiplying matrices could be created, or even a program that could complete all the operations that can be done with matrices. The research that could be done, which would be more in-depth, that could be a continuation of the experiment that we conducted could lead to the next most efficient solution for solving matrices, however, it would require better resources and no time constraints, which was not possible in our testing environment. Regardless, we feel the conclusions that were made in our experiment and the conclusions that could be made with a more in-depth experiment are incredibly interesting, and further research could lead to benefits for almost every profession. In addition, we feel as though we achieved what we had hoped to with this research by finding a faster way of multiplying matrices, which was our CPU-threaded method.

VII. Conclusion

Parallelizing the matrix multiplication algorithm did decrease the run time of the algorithm, taking less time to find the result matrix. This is true for most matrix sizes, specifically the larger matrices. Although making the algorithm use multiple threads only made the program have a faster run time than all the already-established methods for one of our versions, the CPU-threaded method, the other version, the multi-threaded method, although slower than the standard dot product method, still proved to be faster than the Strassen method of multiplying matrices, meaning we did achieve the goal of the research that we were conducting.

There are many possible factors that can affect the run time for the algorithms that solve the multiplication of two matrices. The size of the matrices is a big factor, arguably the overall most important factor, as the bigger the matrices that are being multiplied, the longer it will take to find the resulting matrix, no matter the method that is used. With that being stated, the size of the matrices also determines which method will be the fastest to return the resulting matrix. The other factor that can affect the speed at which the program will finish is the number of cores a computer has, which also affects which method will be the fastest. Since one of our methods could not handle the number of threads that were created during testing, for a relatively small size of matrices, this is also an important factor to consider.

Overall, creating methods of matrix multiplication with multiple threads did prove to be more efficient and faster than the most commonly used methods, under specific circumstances. To efficiently solve the multiplication of two matrices, there would need to be a consideration for the environment and conditions under which the matrices were being multiplied. However, taking

into consideration that for the bulk of the time, the CPU-threaded method was the fastest, the statement that parallelizing the algorithms for multiplying matrices has the potential to create a faster running program that would benefit people in all professions proved to be true.

REFERENCES

[1] Britannica, "Matrix," https://www.britannica.com/science/matrix-mathematics.

[2] S. Shukla, "Strassen's algorithm for matrix multiplication," https://www.topcoder.com/thrive/articles/strassenss-algorithm-for-matrix-multiplication.