Implementing a Principal Component Analysis in Parallel with Pthreads

Jared Berger
Williams College '21

Jamie Vaccaro
Williams College '21

Abstract

Commonly used in statistical analysis, Principal Component Analysis (PCA) allows us to project data onto a lower-dimensional subspace, while losing minimal information, by finding and projecting onto our data's principal components. In what follows, we give an overview of PCA, explain the steps necessary to do the computation, explain our approach to parallelizing the algorithm, and analyze our results for our different parallel approaches for matrix multiplication and matrix addition, and for our two non-parallel optimizations. Finally, we explain future work that could be done to further optimize and analyze our implementation, and explain our our takeaways regarding the difficulty of performing matrix multiplication, the new insights we have gained by exploring the tradeoff of creating more threads while also introducing more overhead, and the extent to which our implementation's execution times still reflect a certain amount of randomness.

1 What is PCA?

At a high level, Principal Component Analysis (PCA) uses an orthogonal transformation to project m-dimensional data on a smaller subspace. PCA allows us to simplify the complexity of our data while retaining trends and patterns, allowing us to reduce highly-dimensional data and better identify trends and patterns.

Principal components represent linearly uncorrelated variables in our data, which can be calculated by creating a scatter matrix for our data, and finding the dominant eigenvectors of this scatter matrix. Then by multiplying our original data matrix by a new matrix with some number k of these eigenvectors as its columns, we can project our data onto a new k-dimensional subspace.

Figure 1 demonstrates an example of a multivariate Gaussian distribution in two dimensions. The vectors depicted represent the two principal components(ie. the two dominant eigenvectors for this data's scatter matrix). We can imagine projecting our data onto the larger of the two vectors, so that we have a representation of our data in one dimension rather than two. Notice how in doing so, because the vector we chose to project onto represents the maximum variance within our data, we can project our data onto a smaller subspace without losing a lot of information. We can imagine doing this with much larger data sets and with data with hundreds or even thousands of dimensions.

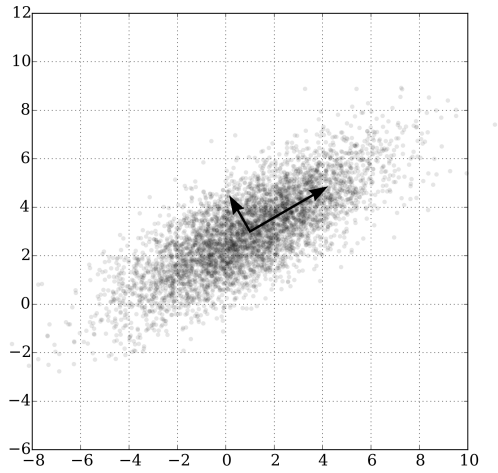

Figure 1: Principal Components of Gaussian Distribution in Two Dimensions [1]

2 How Do We Perform PCA?

To project our data onto a k-dimensional subspace spanned by k of the scatter matrix's principal component vectors, we must perform five steps. We must normalize our data matrix, compute our mean vector, find our scatter matrix, find and choose k eigenvectors, and project our data onto the new subspace.

1. *Normalize Data Matrix*

   We assume our data is presented as an m by n matrix, where m is the number of dimensions of our data and n is the number of elements in our data

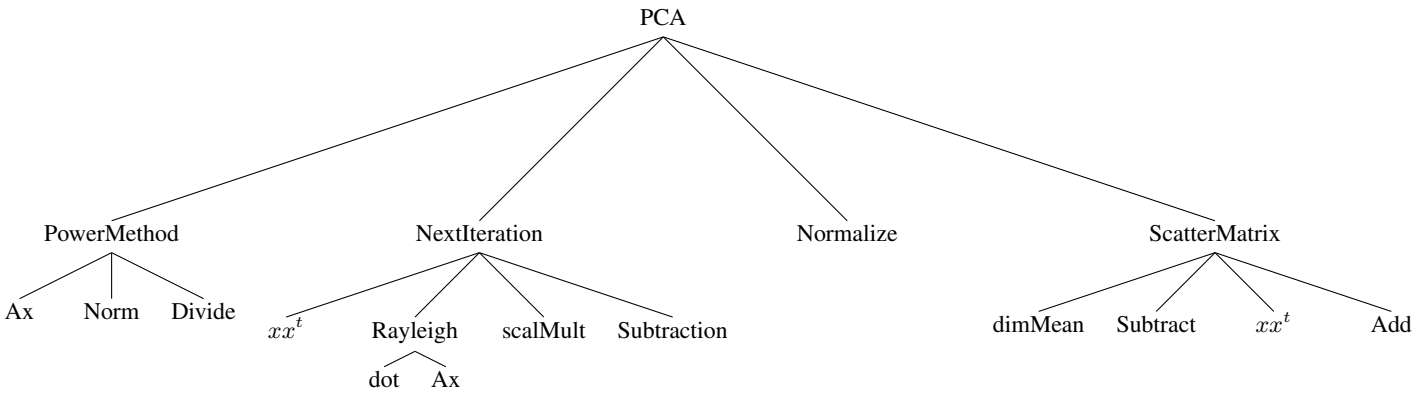

Figure 2: Function Call Tree

set. Therefore, each column in our data matrix can be thought of as a single data point.

To normalize our data, we need to find the mean and the standard deviation. Because the standard deviation requires the mean, we must calculate the mean first. Then we subtract the mean from every entry and divide the whole matrix by the standard deviation.

2. *Compute Mean Vector*

   To compute the mean vector m, we use the following formula:

$$m = \frac{1}{n}\sum_{k=1}^{n} x_k$$

   where again, n is our number of data elements, so x_k represents a column of our data matrix. Essentially, we add each data point together and then divide the resulting vector by the number of columns, giving us the mean for each dimension of our data. [3]

3. *Find Scatter Matrix*

   To compute our data's scatter matrix S, we use the formula:

$$S = \sum_{k=1}^{n}(x_k - m)(x_k - m)^T$$

   Here, for each column vector in our data matrix, we first subtract the mean vector from the column vector and then multiply this vector by its transpose. As the result of each of these multiplications we get a dxd matrix, where d is the number of dimensions of our data, and by summing each of these up we are left with our scatter matrix. [3]

4. *Find and Choose Dominant Eigenvectors*

   To find some number k dominant eigenvectors to project our data onto, we use the power method. In this method, we estimate the dominant eigenvector of our scatter matrix by multiplying a random vector by the matrix a bunch of times, normalizing it in between. By repeating this process after adjusting for the dominant eigenvector we just found, we can find the next-most dominant eigenvector and so on. Doing this repeatedly requires us to find our eigenvectors' corresponding eigenvalues, which we do by finding the Rayleigh quotient.[2]

5. *Project Data onto New Subspace*

   Now, we take the k eigenvalues (where k = the number of dimensions we want to project our data onto) and take these vectors as rows of a new matrix B. We use an orthogonal transformation to project our original data matrix onto our new subspace, by doing B * A, multiplying our new transformation matrix by original data matrix, A.

   By multiplying our k x m transformation matrix by our m x n original data matrix, we are left with a k x n matrix, where k is our new number of dimensions and n is our number of data elements.

3 Parallelization Approach

Figure 2 demonstrates how we have broken up our implementation into a series of smaller functions that call each other to perform all the computation necessary to create our transformation matrix. As the tree in Figure 2 shows, we have a PCA function, the root of the tree, which calls Normalize, ScatterMatrix, PowerMethod, and NextIteration functions, some of which have been broken up further. Because our functions can be broken down into the operations performed at the leaf level of our tree, we have decided to spend our time parallelizing the functions at

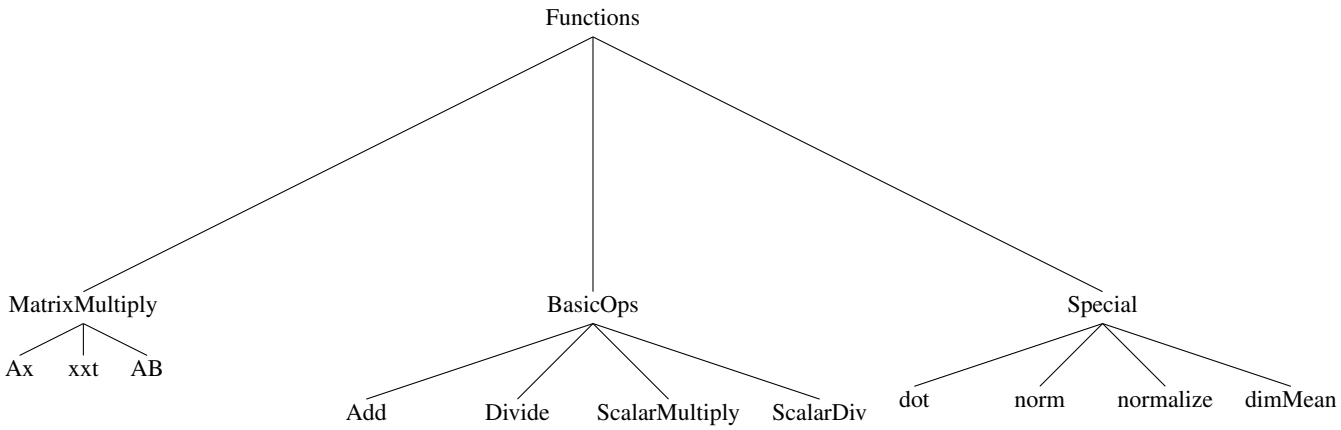

Figure 3: Necessary Operations

this level. For example, if we have efficient parallel Ax, Norm, and Divide functions, then the speed of overall implementation of the Power Method will increase.

Therefore, our approach for parellelization is to parallelize the functions at the leaf level of the tree in Figure 2, creating efficient functions using pthreads that perform all of our necessary computation. We have decided to use pthreads because the five steps enumerated above must be done sequentially, and pthreads allows us a simple means of synchronizing our threads through pthread_join() and sending our threads out to perform different computations with low overhead. Additionally, by using pthreads, we can avoid copying memory excessively (ie. from host to device and back). Instead, we perform much of our computation in place by declaring matrix structs on the heap and passing pointers to these matrices around to our different parallel functions. Pthreads is also much more portable than MPI or GPUs, making this program more useful to have available for most light research work.

Figure 3 demonstrates the operations necessary to perform PCA. Again, we have decided to focus on parallelizing our algorithm by parallelizing the operations at the leaf level of our tree, and rely on calls to these functions to perform all our computation.

1. MatrixMultiply

   (a) Ax

   This operation multiplies a matrix by a vector, necessary for power method iteration to help us find the dominant eigenvectors of our scatter matrix. This is a special case of matrix multiplication, so we thought it might be more efficient to implement it separately.

   (b) xx^t

   This operation multiplies a vector by its transpose, and is necessary when finding our scatter matrix, as explained in the formula on the previous page. This, again, represents a special case of matrix multiplication, so we thought we could develop a separate implementation for it.

   (c) AB

   Full matrix multiplication is not necessary when creating a matrix of dominant eigenvectors, but it is necessary in order to project our original data onto our new subspace. We have five different approaches for parallelizing matrix multiplication. First, we have four approaches in which elements in our result matrix are divided up among some number of threads, and each thread performs all the necessary computation for all their output pixels. For example, if some thread t is responsible for result[1,4] in our result matrix, then t will perform all the intermediate steps necessary to compute the final value that will be written to that location. We have implemented this approach dividing up the output matrix among our threads four different ways: row blocking, column blocking, row interleaving, and column interleaving. We have decided to use these four approaches to test which one is most cache-efficient. Then, we are testing one more approach in which each thread performs some number of intermediate computations for every entry in our result matrix. In this approach, for example, if we had 2 threads, for every output entry in our result matrix, one thread would perform one half of the intermediate computations necessary, our other thread would perform the other half of the intermediate computations necessary, and these intermediate results would be added to-

gether to form our final result. In this implementation, we have decided to avoid having to use locks by having each thread create their own private copy of the result matrix and then add these all together in pthread_join, rather than having threads directly update a shared resource.

```
// rows of output
for(int i=0; i<C->rows; i++)
// cols of output
 for(int j=0; j<C->cols; j++)
 //intermediate computations
  for(int k=0; k<B->rows; k++)
```

Matrix multiplication of two matrices A and B to get a matrix C is $O(n^3)$, as the triply nested for loop above demonstrates. We implement row blocking and interleaving of the output matrix by parallelizing the outermost loop and column blocking and interleaving of the output matrix by parallelizing the middle loop. Then, by parallelizing the innermost loop, we implement our intermediate computation in which each thread performs some amount of the computation necessary for each entry in our result matrix C, and these results are joined together to get our output matrix.

2. BasicOps

   These four operations are very similar, so we have taken similar approaches for parallelizing them. While matrix addition and division require an input of two matrices and scalar multiplication and scalar division require an input of a matrix and a scalar, all four of these operations require perform some repetitive computation to compute each entry in our result matrix. For this reason, our functions to implement these operations only slightly differ and can be parallelized in the same way. Our approach involves dividing the output matrix elements up among our different threads, and having each thread calculate a subset of our output matrix. For each of these functions, we have implemented row blocking, column blocking, row interleaving, and column interleaving.

3. Special

   The last few operations do not fit neatly into the other two categories. The dot product of two vectors is arguably a basic operation, but we decided to include it in the special category because we are restricting it to be between two vectors. Row blocking and interleaving are the only two plausible ways of approaching this, and we decided to take the former approach because it should be better due to better spatial locality. Finding the norm of a vector means summing the squares of its entries and then taking the square root. We tried this both with a specialized operation and by using the dot product, both in parallel and non-parallel approaches. We will discuss these results later. Normalizing a matrix requires finding the mean and the standard deviation of an entire matrix in order to calculate each entry's zscore. Since many parts of this operation are similar to the basic operations, we applied what we learned in those tests to our approach to this one. Therefore, we use row blocking to divide the task among threads. The last function in this category is calculating the dimensional mean of a matrix. This means finding the mean of each row and place it in the corresponding slot in a vector. Once again, there are similarities to the basic operations. However, dividing this by columns makes less sense than before because that would require locks or some other amount of additional work in order to combine the means at the end.

4 Results

Because much of our program can be broken down into variants of matrix multiplication and variants of other basic matrix operations, we decided to run our tests on our matrix multiplication and matrix addition implementations. For all of our tests, we collected results five times and then calculated the average execution time in milliseconds. We tested our matrix multiplication and matrix addition functions first by performing the computation on two 100x100 matrices, followed by on two 1000x100 matrices. We chose these sizes for our matrices because they would allow us to see how our implementation performs on small and on fairly large matrices, without having to wait several hours for each test to complete in the case of matrix multiplication, since it is a very time-consuming operation.

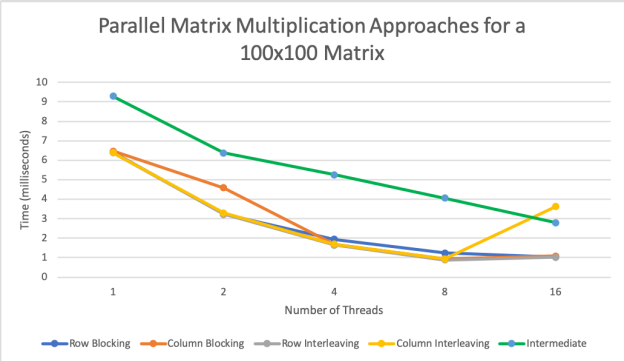

Figure 4: Matrix Multiplication (100x100)

1. Matrix Multiplication

   (a) **Multiplying two 100x100 Matrices**

   As shown in Figure 4, our intermediate computation approach performed significantly worse than our other four approaches. This is likely because when multiplying small matrices, this approach involves a large amount of overhead in order to combine our results from each thread once they have finished.

   For multiplying two 100x100 matrices, row blocking, column blocking, row interleaving, and column interleaving all demonstrated very similar speedups, likely because all four approaches involve a similar number of accesses to our matrices which are stored on the heap. Column interleaving curiously performed far worse than all of our other approaches, but we attribute this to either particularly bad cache performance or someone else running tests at the same time.

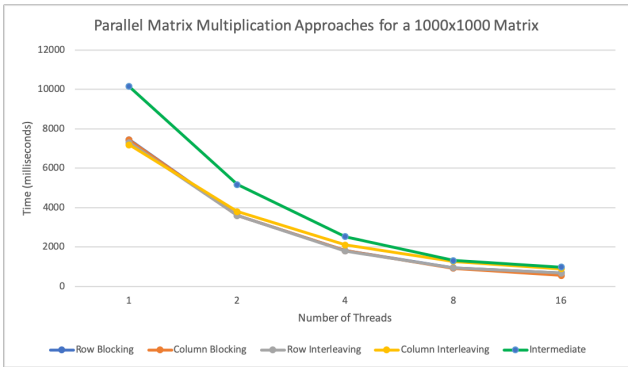

Figure 5: Matrix Multiplication (1000x1000)

   (b) **Multiplying Two 1000x1000 Matrices**

   As shown in Figure 5, while our intermediate computation approach started by performing significantly worse than our other four approaches, the performance of all our approaches converged once the number of threads we used increased. This is likely because, as we have more threads, each thread is able to perform its calculations over the whole result matrix quickly enough so that the overhead of joining together each thread's results becomes less significant. However, our approaches dividing up the result matrix by row and column blocking and interleaving still performed better. This is likely because when a thread calculates an entire entry for our output matrix, it only needs one row from the first input matrix and one column from the second input matrix. Perhaps, then, when doing this computation, the entire row or column is cached and accessed more quickly as the computation, rather than having threads access more random memory locations when calculating intermediate results. Column interleaving sped up the worst our of these four approaches, which might have to do with the fact that our matrices are stored in row-major order. For each output pixel it is responsible for, each thread must always fetch an entire row and entire column, so our results may have depended more on what was already cached or what may have been pre-fetched before program execution, which might explain why column blocking performed the best when using sixteen threads and why there appears to be a certain bit of randomness in our results.

   Because we store our matrices in row-major order and because row blocking, row interleaving, and column blocking all performed very similarly in our tests, we have decided to use row blocking to implement all matrix multiplication-like operations in the final version of our code.

2. Basic Operations

   As we have explained, Matrix Addition, Division, Scalar-Matrix multiplication, and Scalar-Matrix Division are all calculated similarly, so we have decided to demonstrate our implementation of all of these computations by testing our matrix addition function.

   (a) **Adding Two 100x100 Matrices**

   Because matrix addition is a much less intensive operation than matrix multiplication, as shown by Figure 6, adding two 100x100 didn't demonstrate any significant speedup as we used more threads, and actually significantly

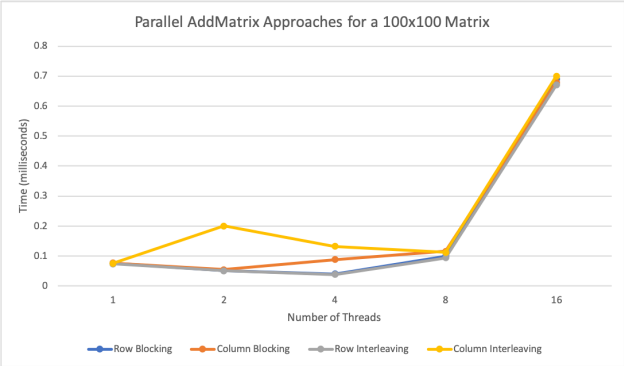

Figure 6: Matrix Addition (100x100)

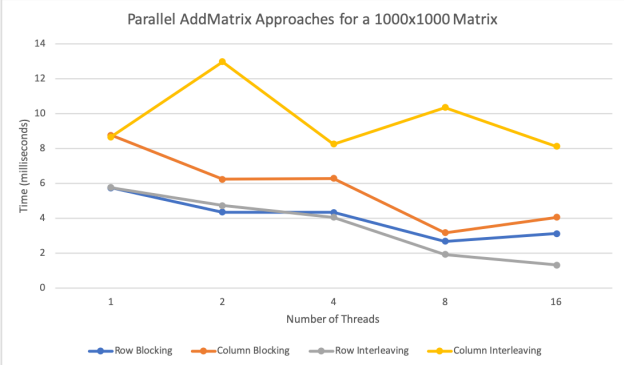

Figure 7: Matrix Addition (1000 x 1000)

slowed down once we got to eight threads. This is likely because for small matrices, matrix addition using only one thread is already pretty fast, only one operation needs to be perfored for each entry in the output matrix. By performing this operation with more and more threads, we only add more overhead, creating multiple threads, having each thread calculate which computation to perform, and waiting for all threads to finish, without significantly speeding up our computation.

(b) **Adding Two 1000x1000 Matrices**

As shown by Figure 7, Matrix Addition performed better using our row blocking and interleaving methods than using our column blocking and interleaving methods. This, is likely due to the fact that our matrices are stored in row-major order, meaning that elements of a row are stored in consecutive memory locations, while elements of a column are spread out. This makes accessing elements within the same row quicker because of spatial locality, as the entire row might be cached as soon as one of its elements is needed by our program. While column blocking did result in a modest speedup, column interleaving did not, with the execution time oscillating from the 9-13 millisecond range, as we tested it with different numbers of processes. These changes in execution time when running using column interleaving might be indicative of poor cache performance, and the execution time might depend more on whether we already have a hot cache or not.

Because row blocking and row interleaving performed the best, and we expect row blocking to be most cache efficient under normal circumstances, we have chosen to use row blocking to implement matrix addition, matrix division, matrix-scalar multiplication, and matric-scalar division in the final version of our code.

5 Non-Parallel Optimization Results

1. Norm

   The norm is the length of a vector. We took two different approaches to calculating it. In one approach, we wrote a specialized function whose only job is to calculate the norm of a vector. In the other approach, we took the square root of the dot product of the vector and itself, which does the same operations using code that already exists elsewhere. We implemented both of these approaches sequentially and in parallel for a total of four functions. Finding the norm of a vector is one of the less expensive operations in PCA. In both the dot product and specialized cases, the sequential versions were superior to the multi-threaded at our benchmark workflow of 1000 dimensions. With the noise and overhead of handling multiple threads involved, the two different versions behaved comparably. However, the sequential dot product sequential version was the best out of all of the available options. Therefore, we decided to use the sequential dot product version of the norm function in both the sequential and multi-threaded versions of our final PCA function.

2. powerMethod Optimization

   The Power Method is an approximation algorithm for guessing the dominant eigenvalue of a matrix. By multiplying a vector by the matrix and scaling it back to a unit vector between each iteration, it typically quickly converges on the eigenvector. Unfortunately, this does not work if the initial eigenvector is orthogonal to the dominant eigenvector. The only way to figure out whether or not we are

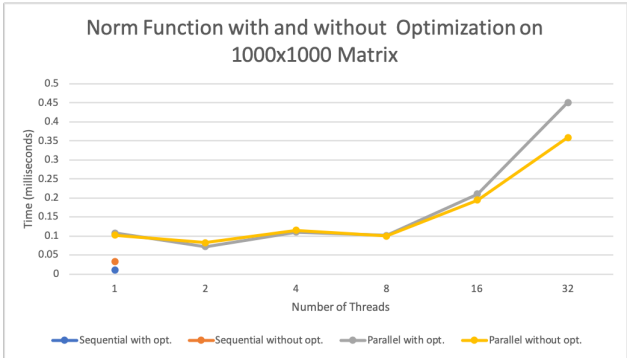

Figure 8: Non-parallel Opt. #2

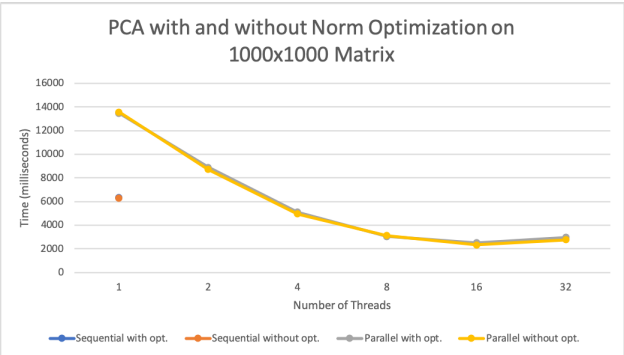

Figure 9: Non-parallel Opt. #2

orthogonal to the dominant eigenvector is to calculate the eigenvalue it is associated with. If the corresponding eigenvalue is zero, it is orthogonal and we need to start over with a new random vector. This element of randomness is unavoidable but is most likely to affect very small matrices with few dimensions where the overhead of starting over is less of a concern. In order to ensure that we calculate the correct eigenvector, we need to calculate its eigenvalue using the Rayleigh approximation. We also need to calculate this eigenvalue when resetting the scatter matrix to find the next most dominant eigenvector. Originally, we just calculated it twice, but upon discovering the redundant calculation we removed it by storing the eigenvalue the first time we calculate it. This is important because we need to multiply a matrix by a vector in order to do the Rayleigh approximation, which is an expensive operation. However, in the context of the entire algorithm the difference was lost in the noise of the data. The benefit is clearest in the sequential version where there there is less randomness in the total compute time, but we decided to apply this optimization in all of our final versions of PCA.

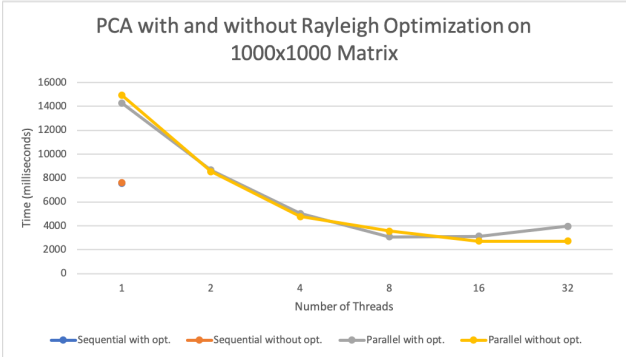

Figure 10: Non-parallel Opt. #1

6 Future Work

While we decided to run all of our tests on and optimize our code for square matrices, our work could be extending by optimizing our code based on the data matrix we are given. For example, while our intermediate computation performed worse than our other four approaches when multiplying two square matrices, we believe it might perform better when multiplying non square matrices. To better explain this, consider the case of multiplying a 5x1000 matrix by a 1000x5 matrix. Our output matrix will be 5 by 5, but 1000 intermediate multiplications are necessary to get each value in the output matrix. In this case, if we have access to a high number of threads, it is much better to parallelize the intermediate computation rather than splitting up the output matrix and having individual threads responsible for performing all 1000 multiplications just to find one output entry. Alternatively, consider the case of multiplying a 1000x5 matrix by a 5x1000 matrix. In this case, our output matrix is quite large, 1000x1000 to be exact, and the number of elements, 1,000,000, is far greater than the number of intermediate computations that must be done to find each output element. Therefore, in this case, it is better to parallelize by giving each thread some portion of the output matrix to compute entirely, rather than having each thread perform some number of intermediate computations but still have to loop over the entire large output matrix and then spend a lot of time combining these results.

Essentially, when multiplying an mxn matrix by an nxp matrix, if m*p is large, we should use one of our four row and column blocking and interleaving approaches for the output matrix, while if $n * n$ is large, we should parallelize the intermediate computations. By having our program use one of these approaches based on the size of the matrices it is given, our code code be further optimized and run more efficiently on real-world data it is given. While square matrices allowed us to limit

the scope of our tests and collect consistent results, we would need to move beyond this to ensure our implementation is efficient for all possible input data matrices.

Further, while we currently use a set number of threads for all of our specialized functions, we could optimize the number of threads we are using both based on our data matrix size and based on performance results for each of our individual functions. Rather than running our code with say 32 threads, we could specify to our program that we have 32 cores available, but then have our program decide how many threads to use for each function in order to minimize overhead for simple operations that are already pretty fast. While clearly for matrix multiplication with large matrices, we always want to use as many threads as we can get our hands on, this is not the case for more simple, less-intensive operations.

We have optimized our program for being run on one machine with up to 32 cores, but ideally, we would also like our code to be even further scalable, so it can easily be run on multiple machines at the same time. For this, we might explore converting our code to MPI, and running and testing it on multiple machines rather than just on limia, and seeing whether this overhead is worth it for sufficiently large matrices.

7 Conclusion

One of our biggest takeaways from implementing PCA is that matrix multiplication is a really slow operation. We originally attempted to perform tests by multiplying two 5000x5000 matrices, yet doing this operation 5 times using only 1 thread to get a baseline result did not finish executing within 24 hours. As an $O(n^3)$ operation, even approaches that greatly speed up matrix multiplication, still noticeably take a few seconds to complete. While we believe we have optimized our code for being run on one machine with pthreads, we also have a new understanding of the limitations of CPU processors when performing such memory and computation intensive operations. With matrix multiplication being such an important operation, we know better understand why special accelerator units are developed just to perform this operation, as we learned about in the article we read about tensor processing units. By using a GPU or tensor processing unit to perform matrix multiplication, we both have access to more threads, and memory optimization approaches such as tiling can be used so that each thread has access to the memory it needs to perform its assigned operations. Even when parallelizing matrix multiplication so that it can be done on dozens of cores at the same time, our implementation of matrix multiplication by the performance capabilities of our CPU, both having to do with how many cores are available and how quickly we can access our matrices which are all stored on the heap.

Additionally, we have learned a lot more about the trade-offs of parallelization by investigating the results of parallelizing versus the effects of the overhead involved in creating different threads. It was interesting to see how for small numbers of threads, our parallel code to do all of PCA was much slower than our naive sequential code. However, once we were using greater than four threads, our parallel version did, in fact, run quicker than our sequential version. This demonstrates the tradeoff of how using more threads results in more overhead, and how each thread needs to do a sufficient amount of computation in order to justify its creation. Another example of this parallel/sequential tradeoff occurred in testing our implementations of dot product. To our surprise, our sequential implementation of dot product performed better than any of our parallel approaches. Because dot product is a simple operation, even when done using large matrices, it turns out it is more effecient to just do the computation sequentially than to divide it up among different threads.

Finally, despite our careful testing of several approaches to parallelizing PCA, it appears there is still a certain amount of randomness in our algorithm's running time. For our test of matrix multiplication on 1000x1000, we got very different results for column interleaving than for any of other approaches. We attribute this variation in results to cache efficiency, due to column interleaving likely being the least cache-efficient of all of our implementations. Because of this, the performance of our implementation when run with this method of parallelization may have depended more on other factors such as what was already in machine's cache, rather than on the efficiency of the code itself. Additionally, in our results for testing our whole PCA algorithm with and without our Rayleigh quotient optimization, with only one thread, our code with the optimization performed better, but as the number of threads increased, sometimes the code with the optimization performed better, and sometimes the code without the optimization performed better. One reason for this variability might be because as part of power method iteration, to find the dominant eigenvectors of our scatter matrix, we start with a randomly generated matrix. If our randomly generated matrix happens to be orthogonal to the dominant eigenvector, then our approximation will never converge, and we need to restart this iteration of power method with a different random matrix. If this happens, we end up doing extra computation, which can cause our program to as a whole run slightly slower.

References

[1] Wikimedia Commons. *Gaussian Scatter PCA*. https://commons.wikimedia.org/wiki/File:GaussianScatterPCA.svg.

[2] Tai-Ran Hsu. *Applied Engineering Analysis, Ch. 10*. http://ergodic.ugr.es/cphys/LECCIONES/FORTRAN/power_method.pdf.

[3] Sebastian Raschka. *Implementing a Principal Component Analysis (PCA) - in Python, step by step*. https://sebastianraschka.com/Articles/2014_pca_step_by_step.html.