CCAMA Optimization: Compiler Language and Parallelization

UTDesign II, Fall 2022

Pointer Solutions

ERIK JONSSON SCHOOL
OF ENGINEERING AND COMPUTER SCIENCE

Faculty Mentor: **Dr. Marco Tacca**
Project Mentor: **Dr. Armin Zare**

Pointer Solutions Team Members:
Ryan Allen (EE) **Juan Cantu** (CE) **Quinn Loach** (EE) **Nick Wagner** (EE)

ABSTRACT

The set of techniques studied by Dr. Zare and his research group, known as data enhanced physics based modeling, is used to model engineering applications as small as a mass spring damper system to as complex as the aerodynamics of an inflight plane. One of these techniques, the covariance completion problem, aims to more accurately model complex systems than would otherwise be capable from first principle, physics based modeling. The algorithm used to solve this problem is known as the customized alternating minimization algorithm (CCAMA) and has been implemented in MATLAB by Dr. Zare.

Due to the computational hindrance that MATLAB possesses for extremely large matrix computations, this project aims to translate CCAMA from MATLAB into our choice of compiler language, C++, to allow for computational speedup. Doing so we used the Armadillo C++ Linear Algebra Library due to its easy to understand syntax designed to be similar to MATLAB, its integration with OpenBLAS allowing for efficient matrix decomposition algorithms, and its integration with openMP to allow for easy and efficient parallelization. Through this project a limited version of the C++ CCAMA code has been implemented thus significantly speeding up the computation of this algorithm. This project has also been well documented such that Dr. Zare's students can further improve the implementation to their liking.

Algorithm 1: Customized Alternating Minimization Algorithm.

input: A, G, $\gamma > 0$, tolerances ϵ_1, ϵ_2, and backtracking constant $\beta \in (0, 1)$.

initialize: $k = 0$, $\rho_{0,0} = 1$, $\Delta_{\text{gap}} = \Delta_{\text{p}} = 2\epsilon_1$, $Y_2^0 = \mathrm{O}_{n\times n}$, and choose Y_1^0 such that $\mathcal{A}_1^\dagger(Y_1^0) = (\gamma/\|Y_1^0\|_2)I_{n\times n}$.

while: $|\Delta_{\text{gap}}| > \epsilon_1$ and $\Delta_{\text{p}} > \epsilon_2$,

$$X^{k+1} = \left(\mathcal{A}^\dagger(Y_1^k, Y_2^k)\right)^{-1}$$

compute ρ_k: Largest feasible step in $\{\beta^j \rho_{k,0}\}_{j=0,1,\ldots}$ such that Y_1^{k+1} and Y_2^{k+1} satisfy (25)

$$Z^{k+1} = \operatorname*{argmin}_Z \mathcal{L}_{\rho_k}(X^{k+1}, Z, Y_1^k, Y_2^k)$$

$$Y_1^{k+1} = Y_1^k + \rho(\mathcal{A}_1(X^{k+1}) + Z^{k+1})$$

$$Y_2^{k+1} = Y_2^k + \rho(\mathcal{A}_2(X^{k+1}) - G)$$

$$\Delta_{\text{p}} = \|\mathcal{A}X^{k+1} + \mathcal{B}Z^{k+1} - \mathcal{C}\|_F$$

$$\Delta_{\text{gap}} = -\log\det X^{k+1} + \gamma\|Z^{k+1}\|_* - J_d(Y_1^{k+1}, Y_2^{k+1})$$

$$k = k + 1$$

choose $\rho_{k,0}$ based on (24)

endwhile

output: ϵ-optimal solutions, X^{k+1} and Z^{k+1}.

TABLE OF CONTENTS

I. INTRODUCTION

Dr. Armin Zare and his research group in the UT Dallas mechanical engineering department focus on using tools from machine learning, optimization, and systems theory combined with physics based modeling techniques to accurately model complex dynamical systems that arise in engineering. This set of techniques is known as data enhanced physics based modeling. Some complex engineering applications including turbulence flow, rough surface analysis, and jet noise modeling, can all accurately be modeled using data enhanced physics based modeling. Moreover, a subset technique of this, known as the covariance completion problem, is heavily studied by Dr. Zare.

The covariance completion problem aims to take previously gathered statistical data of complex systems – gathered from experimentation or simulation – and use it to improve the predictive capability of first principle physics based modeling. This in turn creates a much more accurate model and representation of the system. The algorithm used to solve this problem, known as the customized alternating minimization algorithm (CCAMA), has been implemented into MATLAB by Dr. Zare. With this algorithm, Dr. Zare has been able to accurately model complex systems such as the aerodynamics of plane wings, however, there is a major caveat. Due to the computational bottleneck MATLAB produces when computing eigenvalue decompositions and inversions of dense unstructured matrices, it is an unsuitable implementation for use on high performance computing systems (HPC) to simulate with large input sizes.

Through this project, we aim to translate Dr. Zare's CCAMA MATLAB code into C++. The reasoning for this being that, not only should computation speed naturally increase going from a scripting language to a compiler language, but also that C++ offers a wide variety of highly efficient linear algebra libraries to pick from. The benefit of using one of these libraries is that a majority of these libraries were designed with the intention of parallelizing computationally expensive matrix operations onto multiple cores to improve efficiency. The hope is that with an efficient C++ linear algebra library and the ability to parallelize complex code, CCAMA can run much faster than before, and can be used as an open source algorithm for academics to utilize.

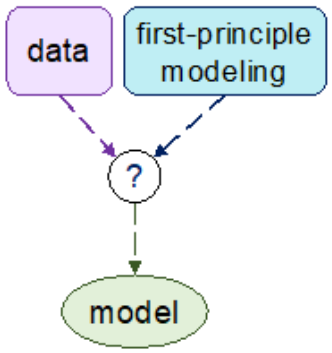


Figure 1: Simplified Diagram of the Covariance Completion Problem

II. PROBLEM ANALYSIS AND OBJECTIVES

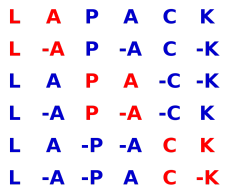
The initial problem facing this project was the lack of experience we as a team had. *Pointer Solutions* is composed of three electrical engineering majors and one computer engineering major. Although most of us did have relevant knowledge of C++, the scope of this task was daunting nonetheless. Our first course of action was to research two main topics: one of course being the C++ linear algebra libraries available and two, understanding what parallelized code entails. Through meetings with Dr. Zare and the team, six main objectives were to be accomplished for the year.

1. Research and evaluate the available C++ linear algebra libraries
2. Understand the difference between shared memory parallelization and distributed memory parallelization and determine which suits the project better
3. Utilize a high performance C++ linear algebra library to implement CCAMA
4. Utilize a library that will allow the C++ CCAMA implementation to be parallelized for computationally expensive tasks
5. Execute the C++ CCAMA implementation on the Lonestar6 HPC
6. Experiment with parallelization
    a. Determine if it improves performance or hinders it
    b. Determine the optimal number of cores

III. C++ LINEAR ALGEBRA LIBRARY RESEARCH

Through researching different C++ linear algebra libraries we found that implementing linear algebra algorithms in C/C++ was a huge field in the world of academia. Many open source libraries were available for use by academics and thus the literature on the topic was extensive. Many of the academics of this field coined the term “eigensolvers” for these linear algebra libraries and much literature was published discussing the use of “eigensolvers” for distributed memory systems vs. shared memory systems. This topic will be explored further in the next section. Through our research the three libraries below consistently appeared within the literature and thus were our top three contenders of choice.

ScaLAPACK and LAPACK



The ScaLAPACK and LAPACK libraries seemed to be the most notorious libraries available as they appeared the most within the literature as being people’s “eigensolver” of choice. LAPACK stands for Linear Algebra PACKage and is one of the oldest and most reliable linear algebra libraries available to academics. It is cited as being extremely efficient in its eigenvalue decompositions as well as simple matrix operations. ScaLAPACK stands for scaled LAPACK. Whereas LAPACK uses shared memory parallelization algorithms, ScaLAPACK used distributed memory. These 2 libraries fundamentally work the same at the

single core level, but they differ when parallelized. Although extremely efficient and cited by many academics, there was one major flaw between these two that simply could not be overlooked. LAPACK and ScaLAPACK both had an extremely difficult syntax to read and understand, especially when not presented with the appropriate background of advanced linear algebra. Figure 2 below showcases the syntax of ScaLAPACK.

```
// % e-values of Xinv
// eigLadYnew = real(eig( A'*Y1new + Y1new*A + C'*(E.*Y2new)*C ));
Hadamard( E, Y2new, T1, desc1 );
Gemmg( "N", "N", 1., T1, C, 0., T3, desc1, desc3, desc3 );
Gemmg( "C", "N", 1., C, T3, 0., T2, desc3, desc3, desc2 );
Gemmg( "C", "N", 1., A, Y1new, 1., T2, desc2, desc2, desc2 );
Gemmg( "N", "N", 1., Y1new, A, 1., T2, desc2, desc2, desc2 );
HermitianEig(T2, eigLadYnew, desc2);
```

Figure 2: ScaLAPACK Syntax Compared to One Line of MATLAB code Commented Above

The issue with this syntax was that without the proper knowledge of exactly what the MATLAB code was doing, there was no instance in which we could've successfully translated the code as their syntax is much too different. This project was meant to be deployed as open source software for academics to use. With that in mind the goal was to build a project for academics strong in MATLAB to understand without having to spend months brushing up on C++. Thus, the wildly different syntax was an immense negative in the LAPACK and ScaLAPACK evaluation.

Eigen

The C++ linear algebra library Eigen is widely considered the industry standard for C++ linear algebra libraries, as the library is used for various projects such as Google's TensorFlow. Eigen claims to be fast by utilizing a variety of methods such as lazy evaluation and explicit vectorization, and the library is also easy to install and use because it has no external dependencies other than the standard C++ library. One major advantage Eigen had over other libraries was that Eigen avoided using static functions for the various linear algebra operations and instead opted to use member functions. Static functions can potentially slow down execution time because they typically generate a lot of overhead on the stack when they have to store the matrix that is inputted into the function, whereas member functions do not need to utilize the stack to store the matrix when the function is called. For these reasons, Eigen was a good choice for implementing the CCAMA algorithm.

Armadillo

The Armadillo C++ linear algebra library was by far the youngest library we found during our research. Not too many scholarly articles were written on Armadillo and those that were, were very recent articles. Armadillo serves as a wrapper library. What that means, is that the linear algebra functionality it provides is through third party dependencies that it wraps around to present an easy to read and understand syntax. The main goal of Armadillo was to provide speed and efficiency of linear algebra functionality through integration of third party dependencies while maintaining a syntax designed to be extremely similar to MATLAB. Figure 3 below showcases the MATLAB to Armadillo table from its official documentation.

Matlab/Octave	Armadillo	Notes
`A(1, 1)`	`A(0, 0)`	indexing in Armadillo starts at 0
`A(k, k)`	`A(k-1, k-1)`	
`size(A,1)`	`A.n_rows`	read only
`size(A,2)`	`A.n_cols`	
`size(Q,3)`	`Q.n_slices`	Q is a cube (3D array)
`numel(A)`	`A.n_elem`	
`A(:, k)`	`A.col(k)`	this is a conceptual example only; exact conversion from Matlab/Octave to Armadillo syntax will require taking into account that indexing starts at 0
`A(k, :)`	`A.row(k)`	
`A(:, p:q)`	`A.cols(p, q)`	
`A(p:q, :)`	`A.rows(p, q)`	
`A(p:q, r:s)`	`A( span(p,q), span(r,s) )`	A(span(first_row, last_row), span(first_col, last_col))
`Q(:, :, k)`	`Q.slice(k)`	Q is a cube (3D array)
`Q(:, :, t:u)`	`Q.slices(t, u)`	
`Q(p:q, r:s, t:u)`	`Q( span(p,q), span(r,s), span(t,u) )`	
`A'`	`A.t()` or `trans(A)`	matrix transpose / Hermitian transpose (for complex matrices, the conjugate of each element is taken)
`A = zeros(size(A))`	`A.zeros()`	
`A = ones(size(A))`	`A.ones()`	[top

Figure 3: MATLAB to Armadillo Syntax Conversion Table

The deliberately easy to understand syntax and the well maintained MATLAB to Armadillo documentation made Armadillo an extremely valuable contender of choice. This not only would allow us to have a much easier time translating, but would also make the CCAMA code much more approachable as an open source software to academics.

As discussed previously, Armadillo serves as a wrapper to other dependencies. Two of the main dependencies we were looking to integrate with were OpenBLAS and Intel MKL.

OpenBLAS

OpenBLAS is an open source library designed to standardize BLAS and LAPACK together into one library. BLAS stands for Basic Linear Algebra Subprograms and was one of the first and most reliable linear algebra C++ packages. However, BLAS only supplies basic functionality and thus LAPACK is used to supply the more complex functions such as eigenvalue decompositions. OpenBLAS also already integrates with openMP which is a C++ multithreading library, thus OpenBLAS is very easily parallelizable. In addition, Armadillo integrating with openBLAS makes this

functionality even easier by simply compiling with an openMP option and openMP will select the optimal number of threads automatically.

Intel MKL
Intel MKL stands for the Intel Math Kernel Library. This library serves nearly the same purpose as OpenBLAS. It contains BLAS, LAPACK, and openMP in its library, but in addition it contains many other mathematical libraries commonly used by academics. The major drawback with Intel MKL is that unlike OpenBLAS it is not open source and is instead a product. In addition to that, although it can run on AMD processors, it is designed explicitly and intentionally to run much more efficiently on Intel processors.

Armadillo was a very favorable choice as it provided easy to use syntax, while providing efficient linear algebra algorithms through MKL or OpenBLAS and automatic parallelization with openMP

IV. DISTRIBUTED vs. SHARED MEMORY SYSTEMS

There are two main types of memory systems that are used when considering parallelized programs, namely shared memory and distributed memory. Research into the topic revealed that distributed memory systems tend to be more efficient at speeding up programs dealing with sparse matrix calculations whereas shared memory systems tend to be more efficient at speeding up programs that require calculations with dense matrices. Each linear algebra library that we found online is designed specifically for use with shared memory systems or distributed memory systems, and so knowing which memory system we would be employing was vital in deciding which library to use. Following is a discussion on what differentiates each type of memory system, which libraries are designed for that type of memory system, and what memory system we ended up using.

Distributed Memory and its Linear Algebra Libraries

In a distributed memory system, multiple processors utilize their own separate memories and are only able to transmit data between the processors using explicit messages to do so. The C++ linear algebra library we found that is optimized for distributed memory systems was ScaLAPACK, which was what Dr. Zare had previously used when attempting to parallelize this program. However, Dr. Zare's set up on campus, from the description he gave us, utilizes a shared memory system, and ScaLAPACK is optimized for distributed memory systems. We believe this is one of the reasons that Dr. Zare did not originally see any improvement from parallelizing with his implementation. We hypothesize that his implementation might see some improvement from implementing on a distributed memory system, but also want to caution that this might not be the best direction in which to explore because our research indicates that shared memory systems would be more useful for dense matrix calculations like those used in the CCAMA algorithm.

Shared Memory and its Linear Algebra Libraries

In a shared memory system, all of the processors share a common main memory; in other words, each processor can see each other processor's reads and writes and each processor has equal latency when accessing the memory. (As a side note, newer computers use more complicated hardware than simply allowing each processor direct access to the same main memory, however the concept of shared memory still works the same from a programming level). We found two libraries that were optimized for shared memory systems: Armadillo and Eigen. Armadillo, a wrapper library, relies on openMP for its parallelization abilities; openMP was one of the most talked about libraries in articles we found discussing memory system choices. Eigen also uses a shared memory system, making both of these libraries ideal for implementing the CCAMA algorithm in C++

C++ Linear Algebra Library Choice - Eigen and Armadillo

Ultimately, we decided that Eigen and Armadillo would be the libraries we utilized to implement CCAMA in C++. For one, their use of shared memory systems makes them applicable to not only Dr. Zare's on campus computer, but any personal computer is likely to utilize shared memory rather than distributed memory, and, on a supercomputer such as LoneStar6, one node can be used with multiple cores to facilitate a shared memory set up. Furthermore, our research suggests that dense matrix calculations will see a better speedup from parallelization utilizing a shared memory system than one using a distributed memory system. Thus, the option that theoretically would yield the most efficient solution was also the one that was most easily available for us to test code on, so it made sense to use shared memory libraries such as Armadillo and Eigen.

V. LONESTAR6 HIGH PERFORMANCE COMPUTING SYSTEM (HPC)

Texas Advanced Computing Center

The Texas Advanced Computing Center, or TACC, is a hub of high performance computing systems (HPC) hosted at the University of Texas at Austin with allowed access to the entire University of Texas system. Thus, as UT Dallas students we are able to remotely access one of their HPCs. TACC is used by many faculty members of the University of Texas system to run their simulations and code for projects they're currently working on. For this project, Dr. Zare was able to give us access to the Lonestar6 HPC to execute our C++ CCAMA implementation and compare the computation speed with MATLAB.

Lonestar6 Architecture

The Lonestar6 HPC consists of a cluster of 560 compute nodes. Each compute node contains 2 AMD EPYC 7763 64-Core Processors with 256GB of RAM. Thus, each node is capable of running 128 cores in parallel. In addition to the compute nodes, Lonestar6 contains 32 GPU nodes and 3 login nodes to be accessed by users. The figure below summarizes the features contained on the compute nodes.

CPU:	2x AMD EPYC 7763 64-Core Processor ("Milan")
Total cores per node:	128 cores on two sockets (64 cores / socket)
Hardware threads per core:	1 per core
Hardware threads per node:	128 x 1 = 128
Clock rate:	2.45 GHz (Boost up to 3.5 GHz)
RAM:	256 GB (3200 MT/s) DDR4
Cache:	32KB L1 data cache per core 512KB L2 per core 32 MB L3 per core complex (1 core complex contains 8 cores) 256 MB L3 total (8 core complexes) Each socket can cache up to 288 MB (sum of L2 and L3 capacity)
Local storage:	144GB /tmp partition on a 288GB SSD.

Figure 4: Lonestar6 Compute Node Specifications

All Lonestar6 nodes are managed with the Simple Linux Utility Resource Manager, also known as the slurm batch environment. This environment allows the developer a large degree of freedom for running jobs on the HPC. The developer simply creates a slurm file specifying their job specifications such as the queue to be submitted to, how many nodes will be needed, how many cores will be needed, and the expected run time. Once a job is submitted, the developer is able to logout of Lonestar6 and the job will continue to either execute or be queued without worry. Figure 5 below shows this process.

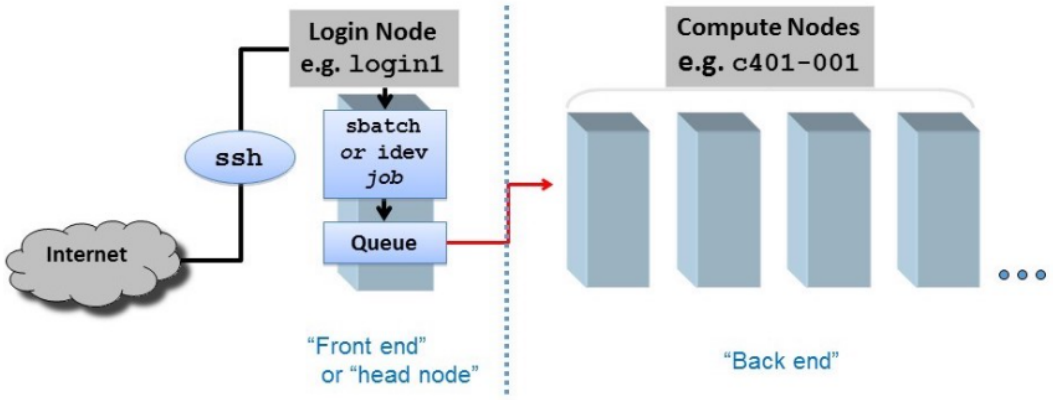


Figure 5: Lonestar6 - Submitting a Job From the Login Node to the Compute Nodes

VI. CCAMA EIGEN IMPLEMENTATION

Windows Environment and Installation

Eigen was very easy to install on Windows because Eigen does not have any external dependencies other than to the standard C++ library. Since Eigen does not require any extra linking, the ease of installation can most likely be extended to other operating systems such as Linux. This contrasts with Armadillo, which requires linking with OpenBLAS and Intel MKL. Although Armadillo comes with pre-compiled versions of OpenBLAS when installing with Windows, that is not available for MacOS or Linux systems, which means OpenBLAS will have to be downloaded separately. Because of this, Eigen is much easier to install than Armadillo.

Implementation Issues

After translating and debugging the Eigen solution, the program was giving the correct output, but, for all sizes of matrices, the run time was significantly slower than both the Armadillo and MATLAB implementations. The issue could not be traced to a single line, which implied that the problem was most likely with the library itself.

Conclusion to Not Use Eigen

Although Eigen is much easier to install and use since there are no extra requirements to install between operating systems and Eigen does not require any external dependencies, the discrepancy in performance when compared to Armadillo and MATLAB was simply unacceptable. For this reason, we decided not to go with Eigen for our final implementation of the CCAMA algorithm.

VII. CCAMA ARMADILLO IMPLEMENTATION

Windows Environment and Installation

As discussed previously Armadillo is a wrapper library around third party dependencies, and the two we were looking into were OpenBLAS and Intel MKL. Although the goal of the project was to execute CCAMA on Lonestar6, we decided best to edit and debug it on Windows using Microsoft Visual Studio before porting it to Lonestar6. Because of this, we did not have to decide on a third party library as the Windows download of Armadillo came with precompiled binaries of the OpenBLAS library.

Similarities to MATLAB

One of the major reasons for choosing Armadillo as the library to use for our CCAMA implementation was its drastic similarities to MATLAB. This was needed to allow easy portability of the project for open source use. The two figures below show the code snippet of MATLAB and Armadillo showcasing their similarities. Figure 6 is the C++ Armadillo code and figure 7 is the MATLAB code.

```
//Use AMA to solve the gamma parametized problem
int AMAstep = 0;
for ( AMAstep = 1; AMAstep <= MaxIter; AMAstep++) {

    //X minimization step
    Xnew = solve(A.t() * Y1 + Y1 * A + C.t() * (E % Y2) * C, Ibig);
    Xnew = (Xnew + Xnew.t()) / 2;

    vec eigX = real(eig_gen(Xnew));
    double logdetX = sum(log(eigX));

    //Gradient of the dual function
    mat gradD1 = A * Xnew + Xnew * A.t();
    mat gradD2 = (E % (C * Xnew * C.t())) - G;

    mat Rnew2 = gradD2;

    double rho = rho1;
```

Figure 6: CCAMA Code Snippet of C++ Armadillo Implementation

```
% Use AMA to solve the gamma-parameterized problem
for AMAstep = 1 : MaxIter,

    % X-minimization step
    Xnew = (A'*Y1 + Y1*A + C'*(E.*Y2)*C)\Ibig;
    Xnew = (Xnew + Xnew')/2;

    eigX = real(eig(Xnew));
    logdetX = sum(log(eigX));

    % gradient of the dual function
    gradD1 = A*Xnew + Xnew*A';
    gradD2 = E.*(C*Xnew*C') - G;

    Rnew2 = gradD2;

    rho = rho1;
```

Figure 7: CCAMA Code Snippet of MATLAB Implementation

As can be seen comparing the code snippets side by side they are very similar. Although it has some small minor syntax differences, one can easily follow along the MATLAB code and know where that is on the C++ Armadillo code. That is the beauty of Armadillo and was a major reason for getting the code translated properly.

Lonestar6 Environment and Issues

After implementing CCAMA in Armadillo on the Windows environment, it was naturally time to port it to Lonestar6 and execute the program. However, what we failed to anticipate was that only the Windows download of Armadillo came precompiled with a third party dependency – OpenBLAS – thus, we had to install both Armadillo and OpenBLAS and link the two libraries before getting CCAMA to work on Lonestar6.

Intel MKL vs. OpenBLAS

Naturally since we used OpenBLAS on the Windows environment, we wanted to use it on the Lonestar6 environment. However, upon delving more into Lonestar6, we realized that Intel MKL was already provided in Lonestar6 as a module. What that meant was that we simply had to install Armadillo and link it to the already installed Intel MKL. Thus, this was the approach we took.

Unfortunately, a hiccup we came across was that libraries such as BLAS, LAPACK, and OpenMP that were encased within Intel MKL and OpenBLAS had different implementations depending on which of the two libraries one used. Even though our implementation worked on Windows using OpenBLAS, switching to Intel MKL caused a runtime error every time we attempted to submit a job using more than one core. The solution to this was ofcourse to install OpenBLAS and link it to the already installed Armadillo on Lonestar6.

Large Matrices Logic Error

After installing OpenBLAS and linking it with Armadillo in Lonestar6 we were finally able to start collecting actual data. As will be discussed in the results section of Lonestar6 in further detail, 1 core, 2 cores, 4 cores, 8 cores, 16 cores, 32 cores, and 64 cores were all tested. However, one issue did emerge that was unexpected. As discussed previously, depending on if one uses Intel MKL or OpenBLAS, different implementations of BLAS, LAPACK, and OpenMP are contained within those libraries. What we didn't know, and what we inevitably discovered, was that the same was true for different operating systems. The precompiled OpenBLAS Windows version was a different implementation than the Linux version we installed for Lonestar6. Although we were able to get CCAMA to work very well for the most part, any job using an input size of 40 and above would throw a logic error of incompatible matrix dimensions. The error is shown below in figure 8.

```
terminate called after throwing an instance of 'std::logic_error'
  what():  addition: incompatible matrix dimensions: 80x80 and 0x0
/var/spool/slurmd/job611896/slurm_script: line 14: 3857023 Aborted
~
```

Figure 8: Logic Error of Incompatible Matrix Dimensions for input size N = 40

This error would usually be generated after about 100 thousand iterations. Although this issue would not be resolved, an analysis of what we believe the future plans to be will be discussed further in the report.

VIII. INITIAL TEST RESULTS AND ANALYSIS - PERSONAL COMPUTER

Our initial tests with the code were conducted on the personal computer of one of our members. The reason for this was twofold; first, it was convenient to test the code on the computer that it was already written on because it did not require transferring the program to a different computer and learning how to use said computer, and, second, it gave us a baseline of what other users could expect if they were wanting to use this program at home themselves which would allow us to see how much more efficient the supercomputer was. This comparison ended up being very valuable to our development process.

Computer Architecture

The relevant specifications of the computer used are as follows: AMD Ryzen 5 2600 3.4GHz processor, 16Gb RAM 1333MHz.

MATLAB vs. Armadillo with MVSC - Single Core

The first tests we conducted were with the C++ code compiled and ran through Visual Studios. With this, we found that the C++ implementation started being faster than Matlab at N = 19, and the improvement only increased as N got bigger. There were two main limitations to this: Visual Studios does not support openMP—meaning we could not test our code with multithreading—and the computer ran out of memory when trying to solve larger problems; namely, any problem with N = 50 or greater quit before it converged due to lack of memory.

MATLAB vs. Armadillo with g++ - Single and Multicore

In order to solve the first limitation of visual studios, we used g++ to compile the code. This allowed for the use of openMP and thus multithreading. Unfortunately, even with g++ compiling and multithreading, the program was still running out of memory for larger problem sizes. The only fix for this would be to run the program on a computer with more RAM, which is why the program was later tested on the LoneStar6 supercomputer, as discussed later in this report.

Verification of Results

To satisfy that the program was getting the correct answer, we wanted to see a percent error of less than 1% between the C++ output and the Matlab output. To calculate this, we took the difference of the Matlab and C++ output matrices and divided the frobenius norm of this matrix with the frobenius norm of the Matlab output matrix. This process was done for both the X and Z output matrices. In all cases, for both Visual Studios and g++ compiler, the percent error was of a magnitude 1e-4%, so we can confidently say that our implementation of the CCAMA algorithm was generating the correct outputs.

Analysis

The execution times for Matlab, Visual Studios, and g++ compiler for 1-8 threads are shown below:

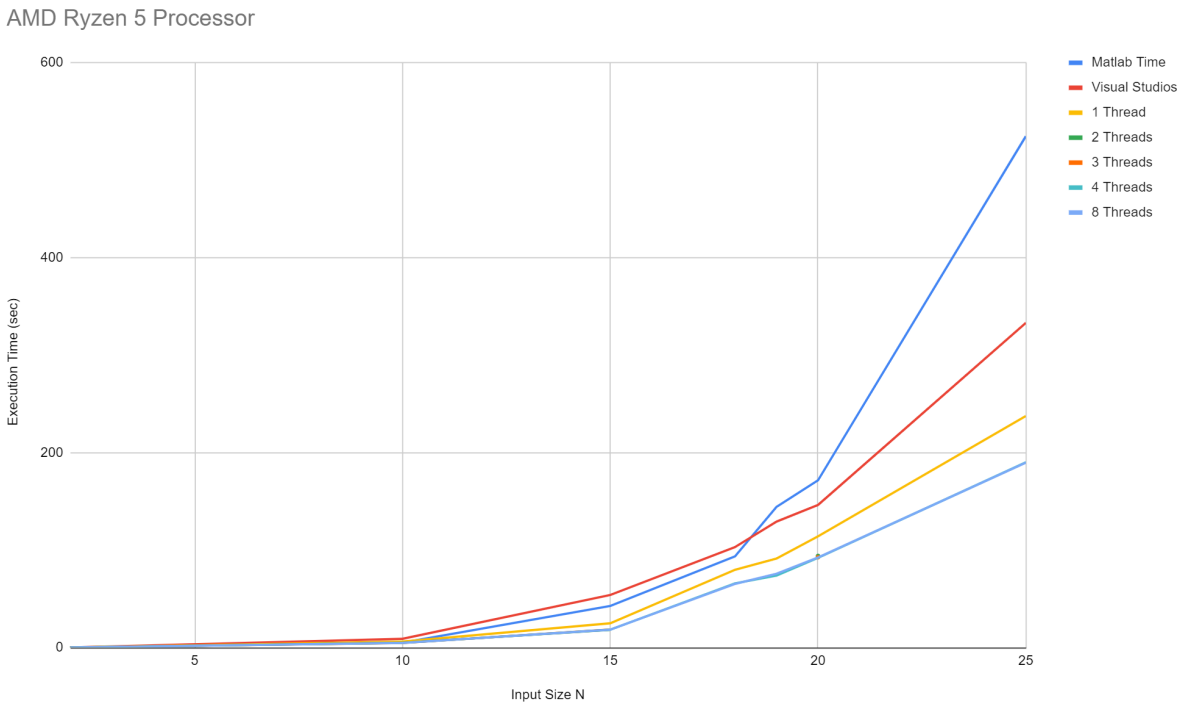


Figure 9: Armadillo vs. MATLAB on Personal Computer

There are a few main observations to point out in this chart. First and most obvious: all C++ implementations are faster than Matlab for any problem with N > 18, and the g++ compiler yielded faster results than both Matlab and Visual Studios for all problem sizes for all numbers of threads. Second, parallelizing the program (using more than one thread) did yield a noticeable speedup. However, using more than two threads did not speed the program up more. This result can most likely be explained by Armadillo's method of parallelization. Namely, Armadillo does not give the user control over how many threads are used, but rather the user can specify a maximum number of threads to allocate the program, and the library automatically determines what the ideal number of threads would be. The way Armadillo decides this is based off of Amdhal's law, but it is sufficient to say that for problems with a small N on a personal computer, Armadillo determined that 2 threads was optimal and did not use more than that. We expect that if we were able to run larger problems (that is, if we did not have the memory constraint on this computer) that we would see use or more threads and thus get a different execution times for different number of maximum threads.

IX. MATLAB RESULTS AND ANALYSIS - LONESTAR6

MATLAB on Personal Computer vs. MATLAB on Lonestar6

N	Matlab Time	Matlab Steps
10	4.94	5149
20	171.63	25301
30	1118	60469
40	7474.74	150203

Figure 10: Personal Computer Results

N	Matlab Time	Matlab Steps
10	22.876354	5160
20	372.099119	24938
30	3094.739728	59135
40	15772.4	150081

Figure 11: Lonestar 6 Results

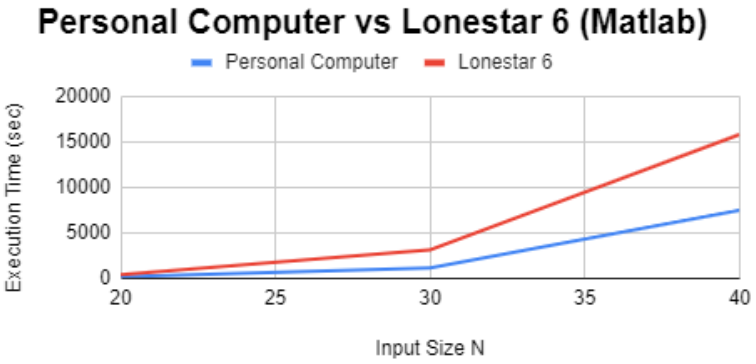


Figure 12: MATLAB Personal Computer vs. MATLAB Lonestar6

Analysis

Multiple tests conducted with MATLAB and CCAMA between both the personal computer and Lonestar 6 show that using a personal computer provided faster run times than when using Lonestar 6. In comparison, the personal computer converged the CCAMA MATLAB code in half the time that it took for Lonestar 6 to compute CCAMA and in some cases 3x the amount of time in nearly the same amount of steps. Questions were raised if possibly something was wrong with the process and procedures for submitting a matlab slurm script and batch job. However, when testing a MATLAB program to compute for 2 + 2, the personal computer computed this in .005 seconds while using MATLAB with Lonestar 6 took .01 seconds. Further research online showed that others have experienced the same discrepancies in runtimes for their programs when using an HPC due to the lack of parallelization and inherent latency. Using a HPC, such as Lonestar 6, will be vital when computing larger values of N and taking advantage of the multiple cores available.

X. ARMADILLO RESULTS AND ANALYSIS - LONESTAR6

Armadillo on Personal Computer vs. Armadillo on Lonestar6

N	Execution Time
2	0.0523
10	8.93
15	53.89
18	103.04
19	129.15
20	146.21
25	333.18

Figure 13: Personal Computer Results

N	Execution Time
10	2.239
20	33.1416
30	204.6

Figure 14: Lonestar 6 Results

The above results display the single core results of CCAMA Armadillo implementation on the personal computer, figure X, and the single core results on Lonestar6, figure X. As can be seen from the figures above, simply switching to the Lonestar6 architecture drastically improved the execution time of the CCAMA Armadillo implementation.

Armadillo vs. MATLAB - Lonestar6

N	C++ Execution Time	Matlab Execution Time
10	2.239	22.876354
20	33.1416	372.099119
30	204.6	3094.739728

Figure 15: Armadillo vs. MATLAB Lonestar6 Single Core Results (sec)

The above table showcases the speedup we receive on Lonestar6 when CCAMA is implemented using the Armadillo C++ library rather than MATLAB on a single core. Without parallelization, the CCAMA Armadillo implementation is already getting a speedup of nearly 10%.

As far as speedup from utilizing multiple cores to parallelize the code, no significant speedup was achieved. On the tests we ran, size N =10 to N = 30, we tested 2, 4, 8, 16, 32, and 64 cores in parallel. As can be seen in the figure below, aside from 2 cores which caused a decrease in performance, performance stayed relatively the same as we continued to add more cores.

N	1 thread	2 threads	4 threads	8 threads	16 threads	32 threads	64 threads
10	2.239	2.249	2.256	2.231	2.249	2.251	2.227
20	33.1416	61.6	39.333	33.086	36.103	33.179	33.137
30	204.6	363.96	206.97	206.485	207.258	205.125	205.959

Figure 16: CCAMA Armadillo Execution Times for Multiple Threads (sec)

Analysis

Based on the results from the figures above several conclusions can be drawn. Simply translating CCAMA from MATLAB to C++ has given a significant speedup in performance, and at the level of input sizes we tested, there was no significant speedup for parallelizing the code. As mentioned previously, we were only able to test limited input sizes due to the logic error we were having. However, according to the literature on Armadillo, we believe if this error is to be resolved and test higher input sizes there would be a significant speedup through parallelization. The figure below demonstrates this prediction.

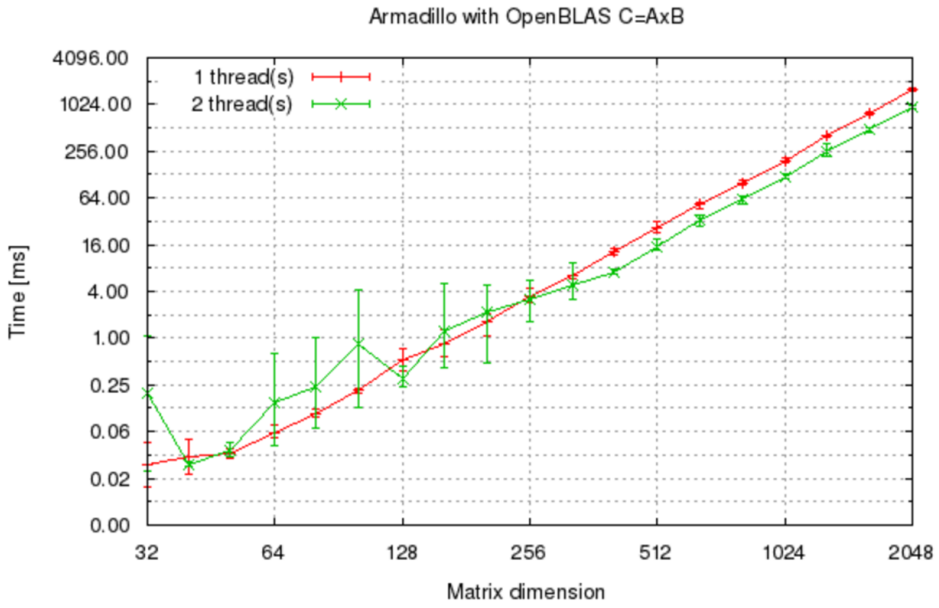


Figure 17: Armadillo Study Showcasing that Parallelization has Speedup Impact at Matrices Larger Than 500x500 (Log Scale)

According to the study above, our current results of the project match what is to be expected, no speedup from parallelization at small input sizes. However, this study does show that speedup with parallelization as the input size increases drastically, does improve dramatically by up to two times. Although we were not able to observe parallelization speedup first hand, we were able to observe the 10% speedup from simply switching from MATLAB to C++ Armadillo library as summarized by the figure below.

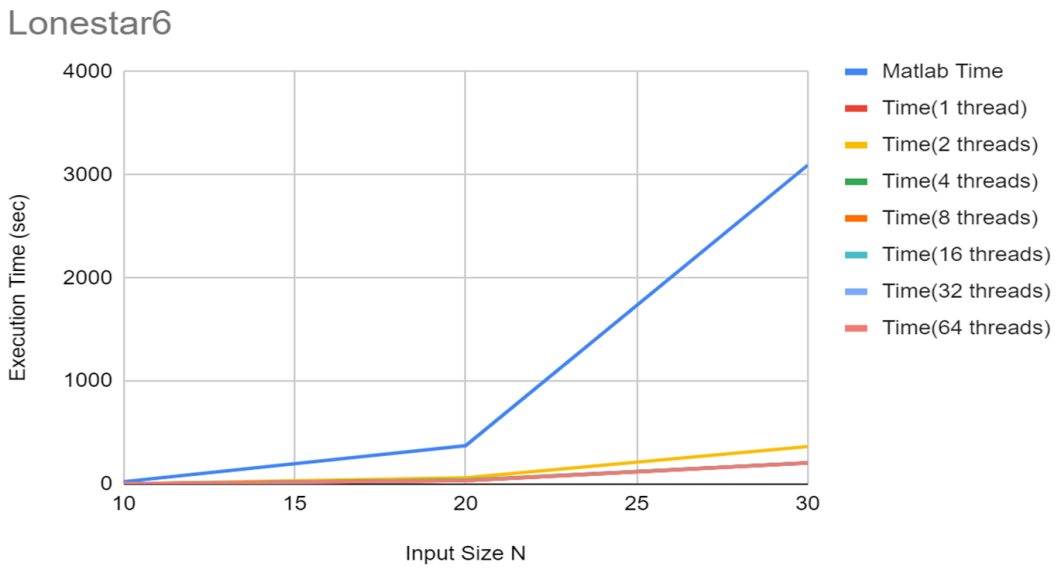


Figure 18: Graphical Summary of Armadillo vs. MATLAB on Lonestar6

XI. PROJECT TRANSFER AND NEXT STEPS

Project Management via GitHub

Throughout the development of this project, we used GitHub to collaborate on the code. Since we were only translating the code from Matlab and not designing a new algorithm, we did not go through multiple versions of the code but rather had a translating phase and a debugging phase. As such, we did not have much use for using GitHub for version control, so the GitHub serves more as a repository of the code as well as documentation on the project.

Tasks Completed

Initial research into C++ linear algebra libraries was completed by everyone; however, once some preliminary research was done and a tentative first library—Armadillo—was chosen, tasks began to be divided among members. Ryan continued researching different libraries and compiling documentation on which ones seemed they would be useful to us while Quinn, Juan, and Nick began the initial translation for Armadillo.

Once we had finished the initial Armadillo translation, Quinn spearheaded the debugging process for the program, Juan did some further research into memory systems, and Nick and Ryan began translating the program using the Eigen library. Eventually, Ryan took over the Eigen translation and Nick and Juan focused on learning how to use LoneStar6 (LS6).

Once Armadillo was debugged, Quinn began testing it on his personal computer, yielding the results discussed in section **VIII** of this paper. Ryan finished translating Eigen and moved into the debugging phase and Nick and Juan continued with LS6.

The LS6 work consisted of uploading files to the supercomputer, installing external libraries and dependencies (Armadillo, OpenBLAS), linking and compiling, writing slurm files, and submitting jobs. Once this was done, Juan focused on testing the Armadillo code on LS6 and Nick focused on testing Matlab on LS6.

Finally, each team member finalized documentation to be uploaded to the GitHub repository.

Armadillo Portability

A major factor in deciding the C++ linear algebra library to use was ease of project portability. This was not only in case changes needed to be made following our time with the project, but also because the project was being designed for use as an open source software. Thus, is the reason we went with Armadillo as our library choice of implementation. Because of its similarity to MATLAB, Dr. Zare and his research group should have a fairly easy time understanding the code to be able to resolve the logic error appearing at larger matrix sizes. In addition, once completed, academics across the nation should have little trouble transitioning from MATLAB to C++ in an effort to understand this open source software.

Armadillo Installation and Lonestar6 Environment Documentation

Because this project will have to be continued on by Dr. Zare and his research group, documentation detailing step by step how to install Armadillo on Windows, Armadillo and OpenBlas on Lonestar6, structure the file directories in Lonestar6, and to configure Armadillo, and how to submit jobs using slurm files to the compute nodes will all be provided to Dr. Zare.

XII. CONCLUSION

This project was presented to us by Dr. Zare as his implementation of CCAMA on MATLAB was running far too slow for practical application use. His vision is to deploy his implementation of CCAMA as an open source software for other academics to use to model the complex dynamical systems they are studying. Our goal with this project was to improve computational performance of CCAMA by implementing the code into C++ utilizing the Armadillo library. In addition, we hoped to develop further speedup by parallelizing the C++ code through OpenMP. Although we were not able to fully test the power of parallelization, we were able to fully implement CCAMA onto C++ using the Armadillo linear algebra library. Doing so, we were able to see significant speedup from MATLAB into C++ for CCAMA. As for the future of the project, Dr. Zare and his research group now have a functioning implementation of CCAMA in C++ that if they can resolve the logic issue we were having, they would be able to experience the full speedup effects of parallelization.

REFERENCES

https://arma.sourceforge.net/

https://arma.sourceforge.net/armadillo_joss_2016.pdf

https://arma.sourceforge.net/armadillo_lncs_2018.pdf

http://fulir.irb.hr/6585/1/6743%20-%20An%20overview%20of%20dense%20eigenvalue%20solvers%20for%20distributed%20memory%20systems%20-%20proofread.pdf

https://portal.tacc.utexas.edu/user-guides/lonestar6

https://stackoverflow.com/questions/22389117/parallelisation-in-armadillo

https://stackoverflow.com/questions/36642382/main-difference-between-shared-memory-and-distributed-memory#:~:text=Shared%20memory%20allows%20multiple%20processing,one%20processing%20element%20to%20another.

https://www.openblas.net/