SPLARF: Series and Parallel Linear Algebra Routines and Functions

Sander Burton[1], Max Clark[1], James Sanford-Luevano[2], Jaxton Winder[1]

[1] *Computer Science, Utah State University*

[2] *Computer Engineering, Utah State University*

Abstract—Linear algebra is used in various application, including large-scale scientific and commercial software. Due to the nature of linear algebra calculations having a high-order polynomial time complexity, speed becomes more crucial as matrix sizes increase. SPLARF is a C++ library of linear algebra routines optimized for speed and efficiency, created to research parallel and serial implementations and provide an interface for commonly used routines. The Message Passing Interface (MPI) is utilized through SPLARF to examine relative overhead for parallel communication, benchmark SPLARF and MPI against common libraries, and explain how gains through parallel computing are possible.

1 Introduction

SPLARF is a C++ library of linear algebra routines optimized for speed and efficiency, created to research parallel and serial implementations and provide an interface for commonly used routines. The need for manipulation of large data sets and increasingly complex calculations in software has led to the frequent use of linear algebra in computing. The use cases for computational linear algebra are exceedingly wide spread, and include academic, commercial, and scientific implementations. Several scientific applications use large matrices and vectors, and consequently require the implementation of increasingly efficient linear algebra routines. This need for ever increasing speed and efficiency has inevitably led to the parallelization of linear algebra routines, splitting the complex and computationally intensive tasks of matrix manipulations among several processors.

The applications for Parallelizing linear algebra routines vary widely from computational fluid dynamics, to the ever expanding field of machine learning. Machine learning often stores gigabytes of data in a single matrix, and uses matrix operations to pre-process the data into a more usable form. Parallelizing linear algebra routines leads to a substantial speedup over the conventional serial approach. Many parallel matrix operation solutions have already been explored, using processor networks such as mesh and hypercube [1]. Our parallel linear algebra project will implement and document sequential and parallel linear algebra algorithms utilizing MPI.

2 Previous Work

The two main motivating factors taken into account when attempting to improve a large set of computations are speed and accuracy; this is similarly the case with our parallel linear algebra calculations. In order to facilitate this increase in speed and accuracy task scheduling should be performed efficiently. In *Scheduling Optimization of Parallel Linear Algebra Algorithms using Supervised Learning* [2], the authors utilize machine learning to research efficient scheduling algorithms. The authors' best strategy utilizes a python script that calculates a decision tree based on block sizes. The authors provide their code freely on GitHub.

Unfortunately the research was performed with HPX, an alternative C++ parallelism library, and after some preliminary attempts to utilize the methods found in the research paper we found the process of translating from the MPI.h to the HPX library would be too time consuming. This does however leave a path to potential future improvements and optimizations for the project.

In *Unifying and Optimizing Parallel Linear Algebra Algorithms* [3], the authors attempt to create an environment for linear algebra calculations that is decomposition-independent. The paper describes a novel environment termed SMD (Subcube Matrix Decomposition) where a recursive hypercube is utilized to perform scheduling of parallel tasks. Other environment configurations and algorithms are discussed and benchmarked against the SMD implementation. One algorithm of interest was LU factorization, which is a common factorization used in many high-level linear algebra functions. With the deadline of the project steadily approaching the decision was made to pursue another LU factorization algorithm that was more easily implemented.

A common operation in linear algebra is matrix inversion. The matrix inversion operation is notoriously complex, especially when considering certain types of matrices. The authors of *A Note on Parallel Matrix Inversion* [4] comment on LINPACK/LAPACK's implementation of matrix inverse. The paper defines a parallel methodology that decreases compute time by up to half compared to the LINPACK-

/LAPACK implementation. After a thorough investigation of whether this algorithm could easily be implemented in MPI, we found that the realization of this algorithm in MPI would be too time consuming with the looming deadline of our project growing closer everyday.

Network scalability is an important topic related to the parallel linear algebra project. MPI has the capability to scale to larger clusters of networked computers, but does it make processing more efficient? *Parallel Linear Algebra on Clusters* [5] discusses scaling of parallel linear algebra algorithms using Scalable LAPACK across 100 MPBS networks. As expected, there are diminishing returns with more computers added. Future Work discusses testing SPLARF over the network.

While researching the topic of parallel algorithms and parallel programming techniques for computational linear algebra, we found several parallelization methods that proved useful to our research. The majority of which we had covered in the first few weeks of the USU CS5500 Parallel Programming course. In an article covering heterogeneous parallel matrix multiplication(HPMax) [6], the article explicitly states the manager/worker configuration is an ideal algorithm for parallelized linear algebra routines. The following quote is an initial description of the HPMax solver framework: "It consists of three main components: [manager], [workers], and GPU(s). The [manager] manages the HPMaX framework." The manager/worker configuration was a crucial configuration for our projects code base. The routine was used in several of our functions. We found the configuration to be an incredibly practical solution to some of our most difficult problems.

An algorithm that initially showed promise was the load balancing algorithm. Although load balancing has found use cases in hierarchical matrices, "... the algorithms for distributing the load of the H-matrix algorithms on p processors are introduced. At this, two different types of load balancing are used: online and offline scheduling.", we found it difficult to discover a practical use case while attempting to use the algorithm for solving matrices in parallel.

3 Design & Development

SPLARF is written in the C++ programming language, for the following reasons:

- Speed
- Memory Efficiency
- Native MPI libraries
- Control over low level functionality

All of the SPLARF routines are programmed with double floating point precision. It is assumed that data validation is performed prior to using this library, as this library is optimised for speed only.

Each team member has taken charge in different areas of design and development. All members implemented at least 1 of the parallel linear algebra routines. Each member contributed in multiple areas of the project, and made larger contributions in areas based on their own expertise. The team's combined expertise includes computational mathematics, project management, software architecture, documentation, algorithms/optimization, unit testing, performance testing, and the C++ language.

The research performed is freely available on Github [7]. The Github repository includes source code for all the functions described in the research, a function testing with make, and the performance results achieved with SPLARF and comparisons against other software.

3.1 MPI Communication Strategies

To minimize overhead, communication strategy is crucial. The strategy that worked the best was a pre-determined cluster working strategy. This strategy used the fact that the work does not change over time. Each processor calculates its slice of work at the beginning of the call and performs it. When the work is complete, each process will share its data with the other processes to stitch together the work performed. This worked in many different areas that did not require knowledge of future work like LU factorization.

A more primitive configuration was a distributed style algorithm. One rank (0) would not perform work, but would distribute and collect work. This was the least efficient on a single computer, but may be a better solution with a networked system with thousands of computers with varying computing power.

3.2 Early Project Difficulties

Early on in the research, a few mistakes were made regarding the path forward. The team was rusty with C and C++ at best; a better approach would have been to put more effort in code structure and potentially even C++ review. Also, team members were all at different levels regarding Jira, Git, and other programming soft skills. Spending more time on this would have prevented a few issues the team ran into during research.

3.3 Non-Core Implementations

3.3.1 MPI Data Casting

A problem found during research was trying to send runsize-timed arrays with different types through MPI. Although MPI has the capability for custom datatypes, research found that custom datatypes were difficult to manage and did not work with arrays with runtime-determined array lengths. To work around this, a datatype with the same size as doubles were used.

The problem with using one datatype is reinterpret casting can be difficult, not intuitive, slow, or dangerous.

A union was programmed to convert between doubles, uint64_ts, int64_ts, and byte arrays. This can be done in a type-safe way, and is easy to read. Listing 1 includes code showing the union in action.

```
union dblUnion
{
    double dbl;
    uint64_t uint64;
    int64_t int64;
    uint8_t uint8[sizeof(double)];
};

// Saving an index as a double,
// extracted as an uint64_t later
double sendBuf[n];
uint64_t index = value;
dblUnion.uint64 = index;
sendBuf[n-1] = dblUnion.dbl;
```

Listing 1: An example of union usage

Another method of reinterpret casing used is pointer casting. An example of this is shown in listing 2. Pointer casting is a bit more dangerous, but fast, as the compiler does not spend time re-assigning or storing the variable.

```
uint64_t value = 1;
double packed = *(double *)&value;
value = *(uint64_t *)&packed;
```

Listing 2: Reinterpret casting by pointer

The end result of this is reducing two MPI calls to one, simplifying communication and reducing communication overhead.

3.4 Basic Arithmetic

3.4.1 Matrix/Vector Scalar Arithmetic

$$\begin{bmatrix} a_{11}(+/\times)s & \cdots & a_{1n}(+/\times)s \\ a_{21}(+/\times)s & \cdots & a_{2n}(+/\times)s \\ \vdots & \ddots & \vdots \\ a_{n1}(+/\times)s & \cdots & a_{nn}(+/\times)s \end{bmatrix} \quad s \in \mathbb{R}, \tag{1}$$

$$\begin{bmatrix} a_{11}(+/\times)s \\ a_{21}(+/\times)s \\ \vdots \\ a_{n1}(+/\times)s \end{bmatrix} \quad s \in \mathbb{R}, \tag{2}$$

Scalar operations are always performed in parallel in a pre-determined order defined by the function. The following functions are implemented:

- Matrix-Scalar Addition
- Matrix-Scalar Subtractions
- Matrix-Scalar Multiplication
- Vector-Scalar Addition
- Vector-Scalar Subtractions
- Vector-Scalar Multiplication

3.4.2 Vector-Vector Addition

$$\begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} = \begin{bmatrix} a_1 + b_1 \\ a_2 + b_2 \\ \vdots \\ a_n + b_n \end{bmatrix} \tag{3}$$

The vector-vector addition performed in SPLARF has two modes: Single threaded, and distributed workload. In single threaded mode, vectors underneath a defined size (currently length 100) will be run on rank 0 and returned. If large arrays are provided, rank 0 serves as the distributor of work, and rank 1 to n-1 are used as worker nodes. When work is finished, rank 0 sends a poison pill to return the summed vectors. Equation 3 shows the mathematical representation of the operation.

3.4.3 Vector-Vector Subtraction

$$\begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} - \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} = \begin{bmatrix} a_1 - b_1 \\ a_2 - b_2 \\ \vdots \\ a_n - b_n \end{bmatrix} \tag{4}$$

The vector-vector addition performed in SPLARF has two modes: Single threaded, and distributed workload. In single threaded mode, vectors underneath a defined size (currently length 100) will be run on rank 0 and returned. If large arrays are provided, rank 0 serves as the distributor of work, and rank 1 to n-1 are used as worker nodes. When work is finished, rank 0 sends a poison pill to return the subtracted vectors. Equation 4 shows the mathematical representation of the operation.

3.4.4 Matrix-matrix addition

$$\begin{bmatrix} a_{11} & \cdots & a_{1n} \\ a_{21} & \cdots & a_{2n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{bmatrix} + \begin{bmatrix} b_{11} & \cdots & b_{1n} \\ b_{21} & \cdots & b_{2n} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} + b_{11} & \cdots & a_{1n} + b_{1n} \\ a_{21} + b_{21} & \cdots & a_{2n} + b_{2n} \\ \vdots & \ddots & \vdots \\ a_{n1} + b_{n1} & \cdots & a_{nn} + b_{nn} \end{bmatrix} \tag{5}$$

The matrix-matrix addition is always performed multi-threaded. The communication strategy is rank 0 distributes work. Ranks 1 through n - 1 are worker nodes who communicate with rank 0. When the work is complete, rank 0 sends a poison pill for the remaining workers to return. Equation 5 shows the mathematical representation of the operation.

3.4.5 Matrix-Matrix Subtraction

$$\begin{bmatrix} a_{11} & \cdots & a_{1n} \\ a_{21} & \cdots & a_{2n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{bmatrix} - \begin{bmatrix} b_{11} & \cdots & b_{1n} \\ b_{21} & \cdots & b_{2n} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} - b_{11} & \cdots & a_{1n} - b_{1n} \\ a_{21} - b_{21} & \cdots & a_{2n} - b_{2n} \\ \vdots & \ddots & \vdots \\ a_{n1} - b_{n1} & \cdots & a_{nn} - b_{nn} \end{bmatrix} \tag{6}$$

The matrix-matrix subtraction is always performed multi-threaded. The communication strategy is centrally distributed (rank 0). Ranks 1 through n - 1 are worker nodes who communicate with rank 0. When the work is complete, rank 0 sends a poison pill for the remaining workers to return. Equation 6 shows the mathematical representation of the operation.

3.5 Vector functions

3.5.1 Vector Dot Product

$$\begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} \bullet \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} = \sum_{i=1}^{n} b_i + a_i \tag{7}$$

The vector dot product is always performed multi-threaded. The communication strategy is rank 0 determines work structure for all ranks, including itself. Rank 0 then sends the work information to the remaining workers. When work is complete, data is gathered on all processors. Equation 7 shows the mathematical representation of the operation.

3.5.2 Vector Element-wise Product

$$\begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} \times \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} = \begin{bmatrix} a_1 \times b_1 \\ a_2 \times b_2 \\ \vdots \\ a_n \times b_n \end{bmatrix} \tag{8}$$

The vector element-wise product has two implementations. The first is a series implementation for small vectors. The second is a predetermined workload based on the vector lengths. Rank 0 is The communication strategy is rank 0 determines work structure for all ranks, including itself. Rank 0 then sends the work information to the remaining workers. When work is complete, data is gathered on all processors. Equation 8 shows the mathematical representation of the operation.

3.6 Matrix Functions

$$A = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ a_{21} & \cdots & a_{2n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{bmatrix}, A^T = \begin{bmatrix} a_{11} & \cdots & a_{n1} \\ a_{12} & \cdots & a_{n2} \\ \vdots & \ddots & \vdots \\ a_{1n} & \cdots & a_{nn} \end{bmatrix} \tag{9}$$

The matrix transpose is always performed multi-threaded. The communication strategy is rank 0 distributes work. Ranks 1 through n - 1 are worker nodes who communicate with rank 0. When the work is complete, rank 0 sends a poison pill for the remaining workers to return. Equation 9 shows the mathematical representation of the operation.

3.6.1 Matrix-Matrix Element-wise Multiplication

$$\begin{bmatrix} a_{11} & \cdots & a_{1n} \\ a_{21} & \cdots & a_{2n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{bmatrix} \times \begin{bmatrix} b_{11} & \cdots & b_{1n} \\ b_{21} & \cdots & b_{2n} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} \times b_{11} & \cdots & a_{1n} \times b_{1n} \\ a_{21} \times b_{21} & \cdots & a_{2n} \times b_{2n} \\ \vdots & \ddots & \vdots \\ a_{n1} \times b_{n1} & \cdots & a_{nn} \times b_{nn} \end{bmatrix} \tag{10}$$

The matrix element-wise multiplication is always performed multi-threaded. The communication strategy is rank 0 distributes work. Ranks 1 through n - 1 are worker nodes who communicate with rank 0. When the work is complete, rank 0 sends a poison pill for the remaining workers to return. Equation 10 shows the mathematical representation of the operation.

3.6.2 Matrix-Vector Product

$$\begin{bmatrix} a_{11} & \cdots & a_{1n} \\ a_{21} & \cdots & a_{2n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{bmatrix} \bullet \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} = \begin{bmatrix} (a_{11} \times b_{11}) + & \cdots & + (a_{1n} \times b_{1n}) \\ (a_{21} \times b_{21}) + & \cdots & + (a_{2n} \times b_{2n}) \\ \vdots & \ddots & \vdots \\ (a_{n1} \times b_{n1}) + & \cdots & + (a_{nn} \times b_{nn}) \end{bmatrix} \tag{11}$$

The matrix vector product is always performed multi-threaded. The communication strategy is rank 0 distributes work. Ranks 1 through n - 1 are worker nodes who communicate with rank 0. When the work is complete, rank 0 sends a poison pill for the remaining workers to return. Equation 11 shows the mathematical representation of the operation.

3.6.3 Matrix-Matrix Product

$$\begin{bmatrix} a_{11} & \cdots & a_{1p} \\ a_{21} & \cdots & a_{2p} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{np} \end{bmatrix} \bullet \begin{bmatrix} b_{11} & \cdots & b_{1n} \\ b_{21} & \cdots & b_{2n} \\ \vdots & \ddots & \vdots \\ b_{p1} & \cdots & b_{pn} \end{bmatrix} = \begin{bmatrix} \sum (a_{1i} \times b_{j1}) + & \cdots & + \sum (a_{1i} \times b_{jq}) \\ \sum (a_{2i} \times b_{j2}) + & \cdots & + \sum (a_{2i} \times b_{jq}) \\ \vdots & \ddots & \vdots \\ \sum (a_{ni} \times b_{j2}) + & \cdots & + \sum (a_{ni} \times b_{jq}) \end{bmatrix} \tag{12}$$

Due to the complexity and polynomial nature of matrix compliation, multiple implementations were programmed for matrix * matrix products. The first utilizes predetermined slices of work given to each processor. When all processes are finished, the data is shared and the function returns. The second implementation is a serial matrix * matrix product for small matrices.

3.6.4 Diagonal And Triangle $Ax = b$ Solvers

Common linear algebra operations involve finding a solution to the system of equations represented by $Ax = b$, where A is a square matrix of size $N \times N$, b is a known vector of size $N \times 1$, and x is the vector you aim to find of size $N \times 1$. In SPLARF, we have solvers for special cases of these systems where matrix A possesses certain properties.

The following solvers exist in the SPLARF library.

- Diagonal Solver
  - Solves the system $Ax = b$ with a given matrix A that is **diagonal** utilizing forward substitution in $O(n)$ time
  - A diagonal matrix A contains non-zero elements along it's diagonal, and 0's everywhere else.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & -2 & 0 \\ 0 & 0 & 3 \end{bmatrix}$$

- Lower Triangular
  - Solves the system $Ax = b$ with a given matrix A that is **lower triangular** utilizing forward substitution in $O(n^2)$ time
  - A lower triangular matrix A contains non-zero elements along it's diagonal and below the diagonal, and 0's everywhere above the diagonal.

$$\begin{bmatrix} 1 & 0 & 0 \\ 2 & -2 & 0 \\ -3 & -4 & 3 \end{bmatrix}$$

- Upper Triangular
  - Solves the system $Ax = b$ with a given matrix A that is **upper triangular** utilizing backward substitution in $O(n^2)$ time
  - An upper triangular matrix A contains non-zero elements along it's diagonal and above the diagonal, and 0's everywhere below the diagonal.

$$\begin{bmatrix} 1 & 4 & -3 \\ 0 & -2 & 2 \\ 0 & 0 & 3 \end{bmatrix}$$

Each of these solvers will produce solutions to $Ax = b$, if such a solution exists. The current implementation is only done in serial. These were created with the intention of creating parallel implementations that would be utilized by more advanced features of the system that are planned.

The source code for these routines is provided in the `src/solvers` directory of the SPLARF Github repository [7].

3.6.5 LU Factorization

For a given square matrix A, there may exist a pair of two square matrices that are Lower-triangular and Upper-triangular in form–denoted L and U respectively–that when multiplied together, produce $LU = A$. Finding a decomposition of a given matrix A into a lower/upper triangular pair LU enables one to "chain together" the lower-triangular and upper-triangular $Ax = b$ equation solvers in a special way, such that a solution to $Ax = b$ can be found for a square matrix A that has an LU factorization.

For matrices that have such an LU decomposition (without the additional use of a pivot matrix P), SPLARF provides a serial algorithm that produces this LU factorization in $O(n^3)$ time, where n represents the dimension of the given square matrix A. As an $A = LU$ factorization is *not* unique (meaning there may exist more than one LU factorization for a given matrix A)–the produced LU factorization is restricted to producing a lower triangular matrix L that is *lowerunitriangular*: having only 1's on the main diagonal, and possesses all other properties of a lower triangular matrix.

The source code for this routines is provided in the `src/lu` directory of the SPLARF Github repository [7].

3.7 Helper Functions

SPLARF includes functions for generating pseudo-random data. These are useful for testing performance and learning to use the library. Functions exist for each of the following:

- Random vector
- Random square matrix
- Random diagonal matrix
- Random upper triangular matrix
- Random lower triangular matrix
- Random unit lower triangular matrix

Each generation function is parameterized to give the caller more control over the generated vector/matrix. The size parameter is used to specify the size of a vector, or the size of an nxn matrix. The maxMagnitude parameter is used to specify the maximum and minimum number for any entry in the matrix, i.e. the generated numbers lie in the range $[-magnitude, magnitude]$. The generators use the Mersenne Twister algorithm and a random device to ensure as much randomness as possible.

Other helper functions include utilities for printing matrices and vectors in various formats, such as vertically or horizontally for vectors. This allows smaller data to be visualized more easily, and is useful for testing and debugging. PrintDblUnion() is useful for visualizing the double union explained previously.

3.8 Unit Testing

To quickly test changes to the codebase, several unit tests were designed. These unit tests utilize a makefile to build, run, and compare test files against result text files. Test files are located in the ./test/ directory, and follow the src directory from there. Each test file has a main entry function with the file type of .cpp. For each test file, a file with the same path and filename except *.o.result appended is also created. The *.o.result file is a text file that contains the expected STDOUT string that the test produces.

Running the tests is simple to do. In the root of the project, use the command `make runtests` to compile, run, and check all test results against their respective files. The results of each test are stored in the ./testresults/ directory. If a test is failing, check the corresponding folder in ./testresults.

4 Documentation

Since the SPLARF library is intended for use by third parties, documentation was critical to teach developers how to use SPLARF routines. Writing nice looking, thorough, and easy to understand documentation is time-consuming. We decided to use a developer tool called Doxygen to assist us in streamlining and automating our documentation process. Doxygen is the de facto standard tool for generating C++ documentation from formatted comments in the source code.

Doxygen works by reading doxygen-formatted source code comments which precede functions, classes, structs, and files. After parsing the formatted comments, Doxygen uses its internal abstract model of the comments (representing our API), and generates html, latex, and other types of output documents with the descriptions, parameters, function names, and return values of the routines in our library.

4.1 Configuration

Configuring Doxygen to work with our project was the first problem we faced. Doxygen is centered around programs that are object-oriented, and by default it displays classes and structs. For our library to play nicely with MPI, we opted to simplify things by creating a series of functions, not contained within objects. We ended up configuring doxygen to display functions by the file they are declared in, and also in a main list of functions. This meant that we also needed a doc comment at the beginning of each of our header files. We wanted to be able to use the markdown files from our github repository as part of the generated documentation, to avoid re-writing things twice whenever we made changes. This integration required some additional configuration settings in the Doxyfile. It is not perfect yet, but all of our .md files can be read in the generated html documentation. We used Doxygens GUI interface to set up all of these options, then saved those configurations to a Doxyfile.

4.2 Syntax

A doxygen comment is formatted as follows:

```
/**
 * Short Dexcription of function
 *
 * Longer description of function
 * which can include more detail
 *
 * @param name parameter description
 * @param name parameter description
 ...
 * @return description of return value
 */
```

4.3 Generating the Documentation

With a doxyfile configured, generating the documentation is as simple as running

```
doxygen Doxyfile
```

from the project root. Note: you must have doxygen installed and added to your $PATH envrionment variable for the doxygen command to be recognized.

4.4 Using the Documentation

To use the generated html documentation, navigate into the doc/ folder and open the index.html file. This will take you to the main page of the documentation. From there click on the links in the description to browse all functions, or browse functions by file. From these views the API and descriptions of each of our functions is presented in a much easier to understand format. Doxygen also generates latex documentation, a condensed version of which is used in the previous section to outline the functionality of our library.

5 Results

Each set of results includes data not shown in this paper for brevity. The programs and resulting data can be found in the repository on Github [7] in the ./performance-testing folder.

TABLE 1
Benchmarking hardware specifications

Computer 1	Computer 2
Dell G15 5510 laptop	Dell XPS 17 9700
i7-10870H CPU	i7-10875H CPU
2.20 GHz base clock	2.30 GHz base clock
5 GHz boost clock	5.1 GHz boost clock
Ubuntu 22.04 (beta)	Windows 10
16GB RAM	32GB RAM

5.1 Test Equipment

Note 1: Including the -g3 and -O3 flags when compiling with g++ made a world of difference in terms of performance, in fact, using these flags improved run time by several orders of magnitude.

Note 2: Computer 1 and Computer 2 processors have a drastically higher single-threaded frequency than multi-threaded frequency. It is possible and likely that during single threaded testing, performance results are better than per-core multi-threaded performance, ceteris paribus.

5.2 Matrix-Matrix Product, Computer 1

The matrix-matrix product is the most difficult of all the basic matrix operations. The difficulty in this operation is due to the nature of the algorithm. As n grows, the algorithm outer loop grows by $n \times p$ while the inner loop grows by n. This makes the matrix-matrix product the defacto benchmark for our library.

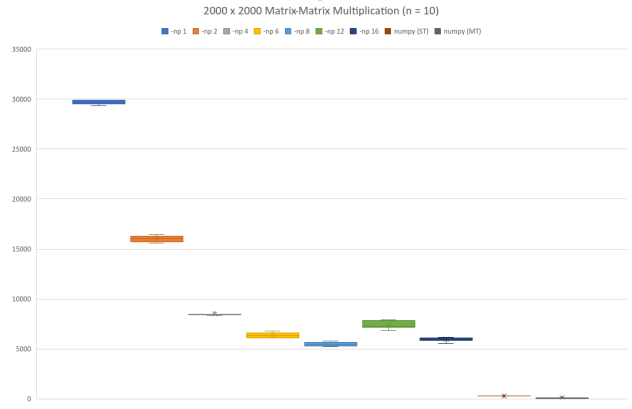

Fig. 1. Computer 1 benchmark of SPLARF matrix-matrix product function and numpy in a large format for readability

Files tested: *matrixMultiplyNumpy.py*, *matrixMultiplyNumpySingleThread.py*, *matrixMultiplySplarf.cpp*, *matrixMultiplySplarfSingleThread.cpp*. The results for a 2000x2000 matrix-matrix product are shown in figure 1. The SPLARF results are logarithmic until approximately 8 processes, where diminishing returns occur.

The Numpy results are shown with the best results from SPLARF in figure 2. This figure shows just how efficient Numpy is with this operation.

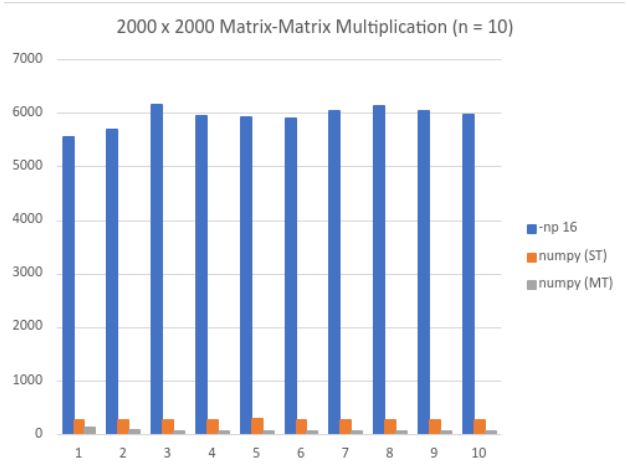

Fig. 2. Computer 1 benchmark SPLARF -np 16, Numpy single and multi-threaded

Numpy was able to beat SPLARF test by more than 10x. During testing, generating a random number to multiply for each cell took substantially longer than the matrix multiplication itself. Numpy is open source, so the source was researched. Underneath the linear algebra package in the Numpy source code is a reference to LAPACK.

LAPACK [8] is a linear algebra library written in FORTRAN originating from the 90s. LAPACK is still in active development, with the latest update occurring in 2022. The matrix-matrix product algorithms in LAPACK are hyper optimized, and are the reason why the Numpy calculations are much faster than SPLARF.

To control for MPI overhead, the algorithm used in SPLARF was converted to a single-threaded application compiled with g++ -g3 -O3 that excluded all MPI calls. The single-threaded application was benchmarked against the SPLARF library using -np 1. The results from this test show the overhead requirements for MPI. The results of the single-threaded algorithm benchmarked against the multi-threaded algorithm with -np 1 is shown in figure 3.

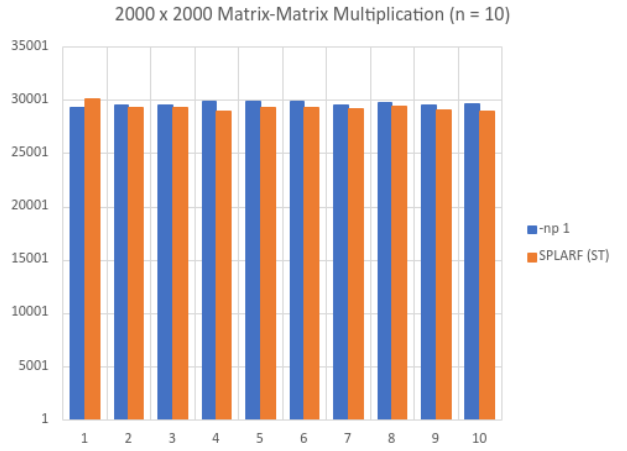

Fig. 3. Computer 1 benchmark SPLARF -np 1, SPLARF single-threaded

The result of the test shown in figure 3 confirms two things. The overhead of the MPI library is low, as the single-threaded test application was only 1.1% faster than the MPI application. This is well within the standard deviation of performance Computer 1 varies with over time. The test also confirms that the SPLARF algorithm, not MPI, is causing the performance decrease relative to Numpy. In Future Work, using LAPACK with MPI as a message interface is discussed.

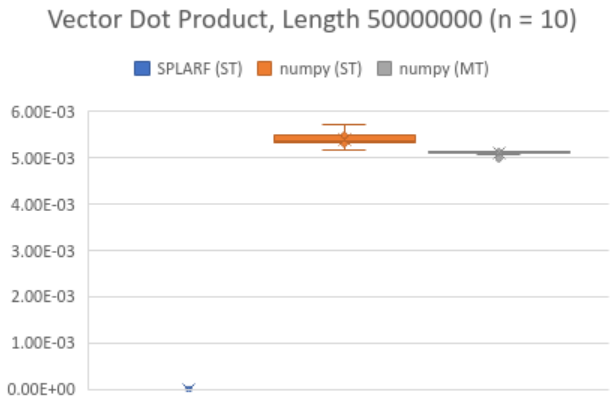

Fig. 5. Computer 1 vector dot product algorithm performance

5.3 Vector Dot Product, Computer 1

A benchmark was designed for the vector dot product algorithm designed for SPLARF. There are three main configurations: MPI, SPLARF single-threaded, and Numpy. The SPLARF single-threaded algorithm is identical to the MPI algorithm, except for the MPI communication wrapper. The purpose of this test is to measure MPI's overhead, validate the algorithm's efficiency, and compare results against Numpy. The files used for this benchmark are *dotProdPerformance.cpp*, *dotProdSerialPerformance.cpp*, *dotProdPerformance.py*, *dotProdPerformanceSingleThreaded.py*.

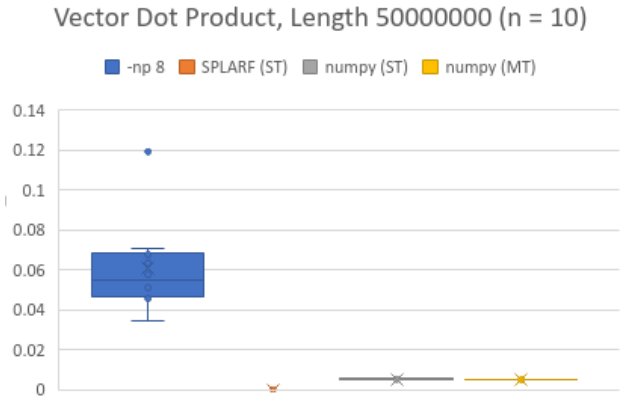

Fig. 4. Computer 1 vector dot product best results

The results from figure 4 show two separate items crucial to understanding the underlying performance for both MPI and Numpy. The first item is the comparison of SPLARF MPI and Numpy scores. These results were not as drastic as the matrix-Matrix product results shown in 2, but Numpy still out-paces the MPI performance by $\approx$ 40ms. However, the performance of the SPLARF algorithm single-threaded allows us to precisely calculate the overhead of MPI. In this particular circumstance, it is $\approx$ 50ms and does decrease with multiple processors.

To examine the second item of speed, only the SPLARF single-threaded and numpy results are shown in figure 5. The single threaded operation is orders of magnitude faster than either of the Numpy solutions. The data shows that the overhead of Numpy/Python is $\approx$ 5ms.

5.4 LU Factorization Single-Threaded, Computer 1

Although time ran out before a parallel version of LU factorization could be tested, a single threaded version was developed. This version, compiled with g++ -g3 -O3, was compared against Scipy (a Numpy alternative). Scipy uses LAPACK to solve Linear Algebra under the hood like Numpy. Figure 6 shows the results of the test. The results are similar to Matrix-Matrix Product testing, and confirms that there are significant optimizations to matrix algorithms LAPACK is performing.

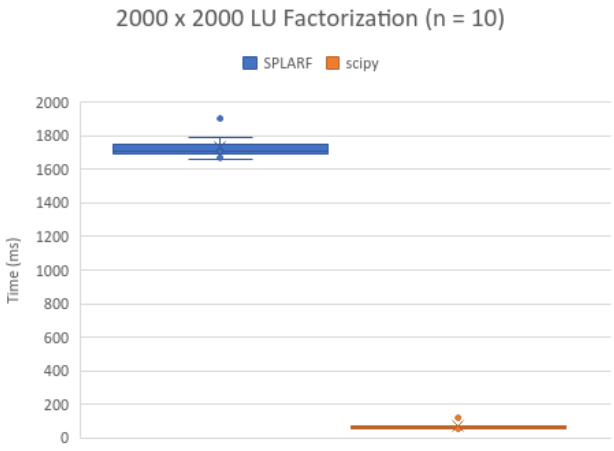

Fig. 6. Computer 1 LU factorization algorithm performance

5.5 Arithmetic Functions, Computer 2

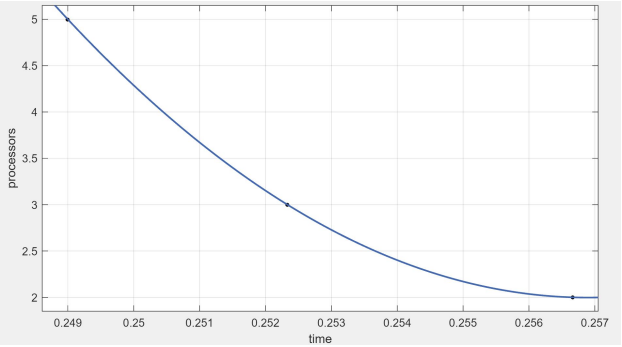

Fig. 7. Amount of Processors vs. Time

The image above is a matlab produced visual representation of the performance of our addition and subtraction methods. On the x-axis it shows the amount of time needed to perform the operations, and on the y-axis the amount of processors used. For the sake of accuracy, each single point on the graph is representative of the average of 30 time samples taken at the associated amount of processors. By observation we see that as the number of processors we use increases the time it takes to calculate the operations steadily decline. This trend is more evident as more processors are added, and can be observed in our next image.

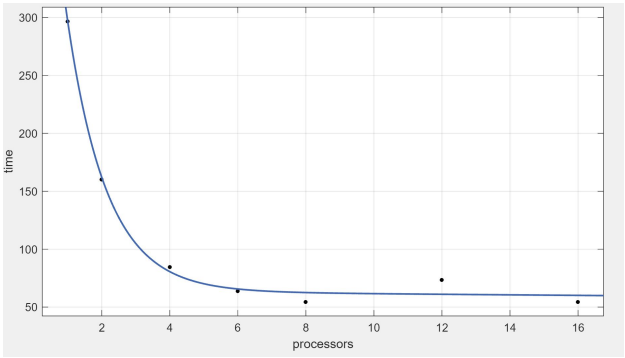

Fig. 8. Time vs. Amount of Processors

The image above is a visual representation of the performance of the project's parallel matrix multiplication operation. The figure shows on the x-axis the number of processors, and on the y-axis the amount of time it took for operation to be performed ten times on a set of large matrices. By observation we see a drastic improvement in performance as processors are added. The trend most closely resembles an exponential curve, because we see as the number of processors increase an exponential reduction in time asymptotically approaching fifty seconds. We do not see a significant reduction in time passed eight processors.

5.6 Results Overview

The results from the research show that the underlying algorithm is crucial to maximizing speed. In the case of matrix-matrix products, Numpy utilizes the LAPACK matrix algorithms, which substantially speed up the calculations. In the vector dot product benchmarks, Numpy had $\approx$ 5ms of overhead, while MPI had $\approx$ 50ms of overhead.

The vector dot product research shows that the fastest MPI implementation will have $\approx$ 50ms of overhead. If an operation takes longer than that, the gains from parallel processing using MPI will overtake the losses from overhead. This speed factor will change from computer to computer and will also change when networked. A potential network benchmark used to optimize systems is discussed in Future Work.

Although not a direct focus of this research, the speed of Python was a factor for our testing. A stereotypical view of Python is that it is a slow language relative to other languages. In many circumstances this is true, but in this case it was proven false. Although Python created $\approx$ 5ms of overhead, Python's overhead plus LAPACK's efficient matrix multiplication algorithm vastly outpaced the SPLARF algorithm. Algorithm selection is a crucial part of speed, and in this case, more research could have been performed specifically on the matrix multiplication algorithm. This is discussed in Future Work.

6 Future Work

6.1 LAPACK Optimization

As shown by the results in figure 1, Numpy was able to beat SPLARF by a significant margin. The Numpy source code was reviewed; the linear algebra calculations performed in Numpy are actually LAPACK [8] functions. These functions are written in Fortran and are extremely optimized.

Future work for this project may be incorporating LAPACK functionality into the library and wrapping MPI around it. The research shows that the overhead MPI costs in performance is not statistically significant utilizing the SPLARF matrix-matrix product algorithm. Further research could be performed on the performance benefits of wrapping LAPACK functions. The researched performed shows that it is likely that wrapping LAPACK with C++ would provide excellent performance benefits and may exceed the performance Numpy has shown.

There is the possibility that MPI's overhead will affect the speed LAPACK may provide considering its speed. Future work for this may include optimizing parameters for parallel processing. For example, a 100x100 matrix may work best when there is no overhead (i.e., single-threaded), but the overhead of MPI with a 20000x20000 matrix may be worth the process splitting.

At the time of writing, ScaLAPACK was found, which is an open-source library that wraps MPI around LAPACK functions. Although ScaLAPACK exists, there still exists a

$$\text{ST}\begin{cases}\text{Vec.} < \text{a} \\ \text{Mat.} < \text{b} \\ \dots\end{cases}, \text{MPI}\begin{cases}\text{Vec.} < \text{e} \\ \text{Mat.} < \text{f} \\ \dots\end{cases}, \text{MPI (net)}$$

Fig. 9. Selection algorithm for work sizes

need to find optimizations for more customized Linear Algebra functionality and implement newer research ScaLAPACK may be too old for.

6.2 Network Performance Testing

One of the benefits that MPI has over other multi-threading interfaces is that it has the ability to utilize non-shared memory. In other terms, MPI can be used in a networked environment. Due to time constraints and complexity, no benchmarking was performed across networks. Future work may be testing and benchmarking cross-computer MPI performance, and optimizing large problems to reduce the amount of bandwidth and latency for network transfers.

6.3 Matrix Rotation Algorithms

A common usage for matrix arithmetic is utilizing matrices for dimensional position. Rotation and translation matrices are helpful in many fields, such as 3D graphics and robotics. Built-in support for matrix rotations could be added to help with development of these applications.

6.4 Linear Algebra Optimization Algorithm

As our research shows, depending on the scale of work, different processing strategies should be utilized to maximize efficiency and processing speed. Each system has differing communication, processing speeds, processors, and networked connections, so it is impossible to derive a magic number that maximizes efficiency. An algorithm could be derived that uses a similar strategy used in the research's performance testing to find the breakpoints for each step.

6.5 $Ax = b$ Solver With LU Factorization

With the creation and verification of serial routines to solve $Ax = b$ for lower triangular and upper triangular matrices and a serial routine to generate an LU factorization for a matrix A, SPLARF is in a prime position to implement a solver to the system $Ax = b$ for a given matrix A that has an LU factorization by transforming $Ax = b$ to $LUx = b$. By finding the LU factorization of A and solving a sequence of systems of equations with the lower-triangular and upper-triangular solvers, we can produce a solution to $Ax = b$ for an even larger class of matrices A than we currently have.

6.6 LU Factorization In Parallel With SAXPY Operations

The creation of serial routines to produce an LU Factorization in our research proved to be successful. We also created our source code such that it performs SAXPY row operations internally, a design choice that prepares us for easy parallelism while computing intermediary row operations. This choice enables us to theoretically create parallel LU factorization algorithm in an $O(\frac{n^3}{p})$ time complexity, where p is the number of processors.

With a parallel LU factorization algorithm, and an $Ax = LUx = b$ solver in serial, SPLARF would be in a position to implement a parallel routine to solve $Ax = b$ for any matrix A that has an LU factorization. Even if the lower and upper triangular solvers are not themselves parallel, the resulting solver will also theoretically have a parallel $O(\frac{n^3}{p})$ time complexity.

REFERENCES

[1] *Parallel Algorithm - Matrix Multiplication*. [Online]. Available: https : / / www . tutorialspoint . com / parallel_algorithm / matrix_multiplication.htm (visited on 03/14/2022).

[2] G. Laberge, S. Shirzad, P. Diehl, H. Kaiser, S. Prudhomme, and A. S. Lemoine, "Scheduling optimization of parallel linear algebra algorithms using Supervised Learning," English, 2019, pp. 1–13.

[3] M. Angelaccio and M. Colajanni, "Unifying and optimizing parallel linear algebra algorithms," *IEEE Xplore*, Dec. 1993. DOI: 10 . 1109 / 71 . 250119. [Online]. Available: https : / / www . researchgate . net / publication / 3299736_Unifying_and_optimizing_parallel_linear_algebra_algorithms.

[4] E. S. Quintana, G. Quintana, X. Sun, and R. van de Geijn, "A Note On Parallel Matrix Inversion," en, *SIAM Journal on Scientific Computing*, vol. 22, no. 5, pp. 1762–1771, Jan. 2001, ISSN: 1064-8275, 1095-7197. DOI: 10 . 1137 / S1064827598345679. [Online]. Available: http: / / epubs . siam . org / doi / 10 . 1137 / S1064827598345679 (visited on 03/17/2022).

[5] F. G. Tinetti, "Parallel Linear Algebra on Clusters," Jan. 2005. [Online]. Available: https: / / www.researchgate.net / publication / 252929924_Parallel_Linear_Algebra_on_Clusters.

[6] H. Kang, H. Kwon, and D. Kim, "HPMaX: Heterogeneous parallel matrix multiplication using CPUs and GPUs," Oct. 2020. [Online]. Available: https: / / dlnext.acm.org/doi/abs/10.1007/s00607-020-00846-1.

[7] Winder, Burton, Clark, and Sanford-Luevano, *SPLARF: Series and Parallel Linear Algebra routines and functions*, version 1.0, 2022. [Online]. Available: https: / / github.com/jaxtonw/sp22-cs5500-project.

[8] *LAPACK*, version 1.0. [Online]. Available: https : / / github.com/Reference-LAPACK/lapack.