Modifications to a Toeplitz triangular factorization routine

Steven K. Ufkes

(Dated: September 17, 2017)

The complete impulse response of the interstellar medium (ISM) is a useful quantity in determining the structure of the ISM. In analyzing pulsar scintillation patterns, the magnitude of the ISM impulse response is retained, while its phase is lost. Retrieval of the phase can be accomplished using a Toeplitz triangular factorization technique.[1] Here, we discuss modifications to a routine which performs this factorization. The speed of the routine increased by a factor of 4 by incorporation of BLAS and LAPACK functions and changes to MPI functions.

I. INTRODUCTION

Pulsar radiation is scattered as it passes through the interstellar medium (ISM). The scattered radiation interferes with itself, resulting in scintillation patterns which can be detected on Earth.[1] Using these scintillation patterns, and very-long-baseline interferometry (VLBI), we can deduce structural features of the ISM and pulsars, as well as the distance to scattering sources in the ISM.[1] For these purposes, the complete impulse response (both phase and magnitude) of the ISM is required.[1]

In analyzing the stochastic process of pulsar scintillation, we obtain the power of the electric field, leaving us with the magnitude of the ISM impulse response, but not its phase. The phase of the ISM impulse response can be retrieved using a Toeplitz triangular factorization technique. Here, we discuss an algorithm due to Bereux [2], which is implemented in a Python script and run on the Blue Gene/Q supercomputer (BGQ) at SciNet.

The implementation of the algorithm was created and modified to use parallel processing prior to this project. The factorization routine was found to run many times slower than an optimal speed estimated on the basis of the number of calculations and the processing power of the BGQ. In this paper, we present a number of modifications that have been made to the code, which resulted in a factor of 4 decrease in the execution time.

This paper proceeds as follows: first, we briefly discuss the factorization algorithm and its implementation. Next, we outline the structure of the code and describe the estimation of the execution time of the code. Next, we discuss the methods used to profile the code. Next, we discuss various changes which were made to the code, and the performance gains achieved by each. Finally, we discuss further changes which could yield better performance in the future.

II. TOEPLITZ FACTORIZATION

A. Signal analysis

In constructing the dynamic spectrum from the stochastic electric field emitted by a pulsar, the magnitude of the electric field is retained, while the phase is lost.[1] There are various ways to retrieve the phase of the dynamic spectrum.[1] Here we discuss a method which involves factorizing a Toeplitz matrix which is constructed from the dynamic spectrum after making some assumptions about the physical properties of the system.[1]

As pulsar radiation passes through the ISM, it is scattered and interferes with itself. The signal that reaches the Earth is a frequency and time-dependent scintillation pattern.[1] The scintillation pattern is observed in the so-called *dynamic spectrum* $I(f,t)$ of the pulsar.[1] If the pulsar is assumed to be stationary, the dynamic spectrum is related to the complex electric field of the pulsar as

$$I(f,t) = \langle |E(f,t)|^2 \rangle. \qquad (1)$$

The complex electric field of the pulsar is also the impulse response of the ISM, the desired quantity for pulsar astrometry and characterization of the ISM structure.[1] From this, it is clear that measurement of the dynamic spectrum does not give us the phase of the ISM impulse response directly. Fourier transforming the dynamic spectrum in time and frequency, we obtain the so-called *conjugate spectrum*, $\tilde{I}(\tau, f_D)$.[1] This quantity can be expressed as the auto-convolution of the Fourier transform of the complex electric field of the pulsar

$$\begin{aligned} \tilde{I}(\tau, f_D) &= \mathrm{FT}\{I(f,t)\} && (2)\\ &= \tilde{E}(\tau, f_D) \star \tilde{E}^*(\tau, f_D). && (3)\end{aligned}$$

Here, τ is the differential delay of the signal, f_D is the Doppler frequency, FT denotes Fourier transformation, $\star$ denotes convolution, and $*$ denotes complex conjugation.

B. Construction of Toeplitz matrix

To retrieve the phase of the ISM impulse response, we first construct a Toeplitz matrix corresponding to the conjugate spectrum, relying on a few assumptions. We assume that the mean of the dynamic spectrum is constant.[1] We assume that the covariance of the signal is Toeplitz (in matrix form: $\bar{I}(i-j) = \bar{I}(j-i)$).[1] These two assumptions together imply that the random process of pulsar emission is "stationary".[1] We further assume that the matrix describing the dynamic spectrum of the pulsar is is Hermitian: $\bar{I}(i,j) = \bar{I}^*(j,i)$.[1] We assume that the process of scattering by the ISM is linear time-invariant and causal, such that the output signal does not depend on future states of the ISM.[1]

Construction of a Toeplitz matrix is useful, because Toeplitz matrices can be used to express the convolution operation in matrix form.[1] If the conjugate spectrum $\tilde{I}(\tau, f_D)$ is written as a positive definite banded Hermitian Toeplitz matrix, $\bar{I}_{n,m}$, then the convolution operation in Eq. 3 can be expressed as a matrix multiplication.[1] The assumption of causality and stationariness imply that $\bar{I}_{n,m}$ is banded and Hermitian Toeplitz.[1] Thus, Eq. 3 can be written

$$\bar{I}_{n,m} = \bar{E}_{n,m}\bar{E}^{\dagger}_{n,m} \tag{4}$$

Here, $\bar{E}_{n,m}$ is the matrix describing the Fourier transform of the ISM impulse response, and $\dagger$ denotes Hermitian conjugation. The assumption of causality requires that the matrix $\bar{E}_{n,m}$ be lower triangular.[1] If $\bar{I}_{n,m}$ is positive definite, there exists a unique LU factorization known as the Cholesky factorization.[1] In the Schur algorithm, the Cholesky factor of a Toeplitz matrix converges to a Toeplitz matrix.[1]

Because it is not guaranteed to be positive definite, the measured dynamic spectrum $I(f,t)$ is modified to ensure positive definiteness using the following steps. First, write the "measured" ISM impulse response $E'(f,t)$ in terms of the measured dynamic spectrum $I'(f,t)$:

$$E'(f,t) = \sqrt{I'(f,t)} \tag{5}$$

Next, write $E'(f,t)$ as a matrix, and pad it with zeros such that its length along each dimension is doubled. Thus, define a matrix

$$E'_{n,m}(f,t) = \text{zero pad}\{[E'(f,t)]_{n,m}\} \tag{6}$$

Next, Fourier transform the new matrix, and express the conjugate spectrum $\tilde{I}(\tau, f_D)$ in terms of the result

$$\tilde{E}'(\tau, f_D) = \text{FT}\{E'_{n,m}(f,t)\} \tag{7}$$

$$\tilde{I}(\tau, f_D) = \tilde{E}'(\tau, f_D) \star \tilde{E}'^{*}(\tau, f_D) \tag{8}$$

The conjugate spectrum defined in this way is positive definite by construction, because it is the autoconvolution of $\tilde{E}'(\tau, f_D)$.[1]

Using the positive definite form $\tilde{I}(\tau, f_D)$, we construct a blocked Toeplitz matrix

$$\bar{I}_{n,m} = \begin{bmatrix} \bar{I}_0 & \bar{I}_1 & \dots & \bar{I}_{n-1} \\ \bar{I}_1^* & \bar{I}_0 & \dots & \bar{I}_{n-2} \\ \vdots & \vdots & \ddots & \vdots \\ \bar{I}_{n-1}^* & \bar{I}_{n-2}^* & \dots & \bar{I}_0 \end{bmatrix} \tag{9}$$

where each block $\bar{I}_k$ is an $m \times m$ complex Toeplitz matrix, and $\bar{I}_0 = \bar{I}_0^*$. Each block $\bar{I}_k$ corresponds to a specific differential delay $\tau = \tau_k$, and a range of Doppler frequencies f_D. The first row and first column of each block are constructed as (in Python notation)

$$\bar{I}_k[0,:] = \tilde{I}(\tau_k, f_D \in [f_{D0}, f_{D\frac{m}{2}}]) \tag{10}$$

$$\bar{I}_k[:,0] = \tilde{I}(\tau_k, f_D \in [f_{D-\frac{m}{2}}, f_{D0}]). \tag{11}$$

The remaining elements are determined by the condition that the blocks $\bar{I}_k$ be Toeplitz. Finally, to ensure that the Cholesky factor converges to a Toeplitz matrix, the blocks $\bar{I}_k$ are padded with zeros such that their size along each dimension doubles. Further, the matrix $\bar{I}_{n,m}$ is padded with blocks of zeros such that the number of blocks is doubled along each dimension.[1] The result is a banded Hermitian blocked Toeplitz matrix with $2n \times 2n$ Toeplitz blocks of size $2m \times 2m$.

The one-to-one mapping used to construct $\bar{I}_{n,m}$ from $\tilde{I}(\tau, f_D)$ is later used to reconstruct $\tilde{E}(\tau, f_D)$ after the Cholesky factor is obtained from the factorization routine.[1]

The construction of the Toeplitz matrix $\bar{I}_{n,m}$ is performed in a separate routine. After the matrix is constructed, it is saved and used as input for another routine which performs the Toeplitz factorization.

C. Factorization algorithm

Here we employ the Toeplitz factorization routine due to Bereux [2]. The algorithm is based on the generalized Schur algorithm.[1,2] It is suitable for large matrices, as the factorization of a blocked Toeplitz matrix with $n \times n$ blocks of size $m \times m$ requires $O(n^2m^3)$ operations, while the standard Cholesky factorization of positive definite matrices requires $O(n^3m^3)$ operations. The factorization routine was designed for complex symmetric blocked Toeplitz matrices. Here, the algorithm is adapted to factorize complex *Hermitian* blocked Toeplitz matrices.[1]

The algorithm is implemented in a Python script. Mathematical operations are computed using NumPy, and select routines from the BLAS and LAPACK libraries. The algorithm was parallelized using the Python module mpi4py, such that the factorization of a matrix of size $nm \times nm$ (before padding) is performed using $2n$ processes. With this parallelization, the time complexity of the factorization is reduced from $O(n^2m^3)$ to $O(nm^3)$.

In this project, we discuss modifications to the "YTY2" blocked variant of the factorization algorithm, which was found to perform faster than the other variants presented in [2]. In this version of the algorithm, the blocks of size $2m \times 2m$ (after padding) are subdivided into blocks of size $p \times p$, where p is usually chosen to be $2m$, m, $m/2$, or $m/4$. The factorization routine follows the steps given in Algorithms 3, 4, 5, 7, 8, 15, and 16 of [2], with only minor modifications.

III. PROFILING AND MODIFICATION OF FACTORIZATION ROUTINE

A. Outline of expensive operations

Here, we outline the computationally expensive steps in the factorization routine without explaining or justifying the steps themselves. The main step in the factorization routine is the reduction of the so-called generator A to proper form. This step is completed in $2n-1$ iterations of a loop over the parameter k.

At a given time, some processes will be performing computations while others are idling (e.g. processes wait to receive data from other processes before continuing). In attempting to reduce the execution time of a certain section of code, we are primarily concerned with the processes doing actual calculations in that section (i.e. the *rate-limiting processes*).

Because the code is intended to be used for large block sizes ($m \approx 1024$) we primarily focused on increasing the performance of sections in which the rate-limiting process performs $O(m^3)$ operations (the most computationally expensive sections). Each iteration of the loop over k involves 9 nonconcurrent steps in which the rate-limiting process is performing $O(m^3)$ operations. Of these 9 steps, 7 involve the multiplication of matrices with varying dimensions. The remaining 2 steps involve the inversion of triangular matrices.

The code involves numerous MPI broadcast calls, in which one process sends data to all other processes; and many MPI send/receive calls, in which one process sends data to one other process. Given the heavy use of MPI in the factorization routine, we also explored improvements to these methods.

B. Estimation of execution time

To quantify the performance of the routine, we roughly estimate the optimal execution time based on the number of nonconcurrent floating point operations and the specifications of the BGQ. In estimating the total number of nonconcurrent floating point operations, we consider only the 7 $O(m^3)$ matrix multiplications. We neglect the 2 $O(m^3)$ matrix inversions because the execution time of these steps is small compared the matrix multiplications, and the number of operations involved depends on somewhat complicated details of the inversion routine used. We estimate the execution time for the simplest case, in which $p = 2m$, and assume that the number of floating operations does not change with p.

Consider the operation $A = BC$, where A is an $n \times m$ matrix, B is $n \times p$ and C is $p \times m$. Each of the nm elements of A are computed by the sum

$$a_{ij} = \sum_{\ell=1}^{p} b_{i\ell} c_{\ell j} \tag{12}$$

For complex matrices, as in the factorization routine, computing each of the p terms in the summand requires 4 matrix multiplications and 2 additions. Summing over the real and imaginary parts of the terms requires $(p-1)+(p-1)$ operations. Thus, the total number of floating point operations is

$$nm\big((4+2)p + (p-1) + (p-1)\big) \approx 8nmp.$$

Each of the $2n-1$ iterations of the loop over k involves 5 nonconcurrent matrix multiplications of shape $(2m \times 2m)$ times $(2m \times 2m)$; $2m$ multiplications of shape $(2m \times 2m)$ times $(2m \times 1)$; and $2m$ multiplications of shape $(m \times 1)$ times $(1 \times 2m)$. Thus, the total number of nonconcurrent floating point operations is roughly

$$\begin{aligned}(2n-1)\big(5 \cdot 8 \cdot (2m)^3 + 2m \cdot 8 \cdot (2m)^2 + 2m \cdot 8 \cdot m \cdot 2m\big)\\ = 416(2n-1)m^3\end{aligned}$$

The execution time for each of these sections depends on the processing power allotted to each MPI process. The BGQ system has a processing power of 204.8 GFlops/node. Each node can run between 1 and 64 processes, specified by the parameter *ranks-per-node*. Thus, we take the processing power allotted to each MPI process to be 204.8 GFlops/(ranks-per-node). Under these assumptions, the optimal total execution time is $416(2n-1)m^3/(204.8 \cdot 10^9/\text{ranks-per-node})$.

The target data size is $n = 16384$, $m = 1024$. Thus, for example, if we run the factorization routine using 2048 nodes with 16 ranks-per-node, the optimal execution time is 13.2 days. Prior to any of the modifications in this project, the total execution time for a sample with $n = 512$, $m = 512$ and $p = 1024$ was approximately 10 times greater than the optimal speed calculated as above.

C. Profiling

To identify which sections of the code consumed large parts of the execution time, the code was profiled by various methods. The Python module "cProfile" was used to determine how long a given process spent inside each function. For a given process, cProfile measures the time spent in and the number of calls to each function and subfunction within the code. This method of profiling was somewhat limited, in that certain functions in the code (e.g. wrappers for BLAS and LAPACK functions) apparently are not detected by cProfile.

Profiling with cProfile is also very limited in that individual processes are profiled separately. At a given point in the execution of the code, some processes will be performing a calculation while others wait to receive data from the working processes. The rate-limiting process changes from step to step within the loop over k, and from iteration to iteration of k. As a result, the profiles generated using cProfile show that a given process spends the majority of it's time waiting to receive data from another process, rather than giving information regarding the rate-limiting process. cProfile was used as a preliminary tool to identify functions which were taking exceptionally long, but was not useful in determining what impact a given section of code had on the total execution time.

The output of cProfile suggested that matrix multiplication, matrix inversion, MPI broadcasts, MPI sends, and MPI receives were taking a significant portion of the execution time, and might be be improved.

In order to measure how long the rate-limiting processes spent in each section of the code, we manually timed small sections of code using MPI barriers. A call to the MPI barrier function halts a process until all other processes make a corresponding call to MPI barrier. To time a section of code, we placed MPI barriers at the start and end of the section. A single process records the times immediately following the start and end barriers. The difference between these recorded times is taken to be the time spent by the rate-limiting processes in the measured section.

This barrier method is useful in determining the impact of a section of code on the total execution time without knowing which processes are rate-limiting. However, this method is rather cumbersome because measuring each section of code requires altering the code itself, and analyzing the execution times of sections which are repeated many times. Further, the total execution time is increased by the addition of barriers, complicating measurements.

Using the barrier profiling method, we found that the functions `__seq_update` and `__house_vec` consumed much larger portions of the total execution time than they would be expected to based on the number of floating point operations they involve.

D. Modifications to factorization routine

1. *Matrix multiplication*

In the original version of the code, all matrix multiplications were performed using the function `numpy.dot`, which can be used for matrices of any shape. Initial tests of multiplying two square matrices suggested that `numpy.dot` worked very close to the optimal speed based on the assumptions of $8n^3$ floating point operations to multiply two $n \times n$ matrices and 204.8 GFlops/(ranks-per-node) per process. Further tests revealed that the efficiency of matrix multiplication was strongly affected by the shapes of the matrices being multiplied, with non-square matrices multiplying much slower than optimal in general (Table. I).

The $O(m^3)$ matrix multiplications in the code vary in the shapes and sizes of the matrices multiplied. Many of these steps steps also involve the addition of matrices or multiplication by a constant. The algorithm written by Bereux [2] specifies BLAS routines for each of these steps, which are optimized for the specific operations performed. We attempted to change each of these steps from using NumPy to the specified BLAS routine. Before making each change, we tested the step in isolation to check how the performance differed between NumPy and BLAS.

In general, the BLAS functions performed faster than NumPy, provided that the arrays input to the BLAS functions were column-major contiguous in memory (i.e. adjacent elements in a column of an array have adjacent memory locations).

TABLE I: Actual time compared to optimal time to compute the product of $n \times p$ and $p \times m$ matrices with complex entries. Each product involved the same approximate number of floating point operations, $8nmp$. Execution times were averaged over 10 repetitions. Performance strongly depends on the matrix shapes.

n	p	m	actual/optimal
1	2097152	1	7.357
128	16384	1	3.062
1	16384	128	5.106
128	128	128	1.661
16384	128	1	2.929
1	128	16384	9.261
16384	1	128	11.84
128	1	16384	12.05
2097152	1	1	14.24
1	1	2097152	14.32

Since NumPy arrays are stored in row-major order by default, the lines of code involving BLAS functions were modified such that the transposes of arrays were input to the BLAS routines. This method of transposing proved simpler than forcing NumPy to store arrays in column-major order. NumPy's transpose operation took a negligible amount of time compared to the difference in times of the lines when performed in NumPy versus BLAS.

For each line in which a change was considered, we provide results of tests performed in isolation comparing the performance of NumPy and BLAS. For changes that were implemented, we give the absolute and relative change in the total execution time before and after the change was made.

Manually timing the code revealed that the majority of the execution time was spent on a single line involving the outer product of two vectors with approximately $2m$ elements each. While this line involves $O(m^2)$ operations, it is in a nested loop iterated $(2n-1)(2m)$ times, so that the time complexity of this step is $O(nm^3)$.

We constructed a test of this line of code in isolation to check if a performance increase could be achieved by using the BLAS routine `zgeru`, as suggested by Bereux [2] (Table II). The test showed that a factor of 20 performance increase in this line could be achieved by changing from NumPy to `zgeru`. As a result of making this change in the factorization routine, the total execution time decreased by a factor of 3, or 768 seconds, for a test with $n = 16$, $m = 512$, and $p = 1024$. This change had the largest impact on the execution time of all the changes by a large margin.

There are numerous lines of code involving matrix multiplications of the form $A \leftarrow A+BC$ or $A \leftarrow BC^\dagger$, where A, B and C are complex matrices of shape $2m\times 2m$. The

TABLE II: Time to compute the outer product of complex vectors with sizes n and m using NumPy and the BLAS routine zgeru. Executions times are the minima of 100 repetitions.

n	m	NumPy (s)	zgeru (s)	Numpy/zgeru
512	512	0.01496	0.00078	19.15
1024	1024	0.06001	0.00256	23.45
2048	2048	0.23898	0.01222	19.55

TABLE III: Time to compute $A \leftarrow A + BC$, where A is $n \times m$, B is $n \times p$, and C is $p \times m$; all complex entries. Performance of NumPy is compared to the BLAS routine zgemm. Execution times are the minima of 10 repetitions.

n	p	m	NumPy (s)	zgemm (s)	NumPy/zgemm
512	512	512	0.104	0.101	1.0287
1024	1024	1024	0.780	0.769	1.0149
2048	2048	2048	6.033	6.007	1.0043

algorithm by Bereux [2] suggests that the BLAS routine `zgemm` be used for these calculations.

Tests of these two types of calculations were performed in isolation to compare the performance of NumPy and `zgemm` (Tables III and IV). These tests suggested that modest speed boosts of 0.4–3% could be achieved by switching from NumPy to `zgemm`. These changes were made in the factorization routine. In all, 6 such lines were changed, 3 of which had time complexity $O(nm^3)$. As a result of these changes, the total execution time of the factorization decreased by roughly 1%, or 4 seconds, for a test with $n = 16$, $m = 512$, and $p = 1024$. This tiny difference indicated that NumPy performs as well or better than a corresponding BLAS routine in some cases.

Two of the 9 $O(nm^3)$ operations are of the form $A \leftarrow A + BB^\dagger$, where A and B are $2m \times 2m$ matrices with complex entries. In both cases, only the upper triangular part of the result is used in future calculations. The algorithm by Bereux [2] suggests that the BLAS function `zsyrk` be used for this calculation. Here, the matrices B are Hermitian, so the appropriate BLAS function is `zherk`.

TABLE IV: Time to compute $A \leftarrow BC^\dagger$, where A is $n \times m$, B is $n \times p$, and C is $p \times m$; all complex entries. Performance of NumPy is compared to the BLAS routine zgemm. Execution times are the minima of 10 repetitions.

n	p	m	NumPy (s)	zgemm (s)	NumPy/zgemm
512	512	512	0.111	0.108	1.0263
1024	1024	1024	0.843	0.836	1.0086
2048	2048	2048	6.751	6.720	1.0046

TABLE V: Time to compute $A \leftarrow A + BB^\dagger$, where A and B are $n \times m$ matrices with complex entries. Performance of NumPy is compared to the BLAS routine zherk. Execution times are the minima of 10 repetitions.

n	m	NumPy (s)	zherk (s)	NumPy/zherk
512	512	0.1402	0.0644	2.18
1024	1024	0.9555	0.4513	2.12
2048	2048	6.4498	3.4649	1.86

We constructed a test of this line of code in isolation to compare the performance of NumPy to `zherk` (Table V). The test shows that the line of code could be sped up by a factor of 2 by switching to `zherk`. After making these changes, the total execution time of the code decreased by 2.7%, or 12 seconds, for a test with $n = 16$, $m = 512$, and $p = 1024$.

Thus far, we have discussed 6 of the 7 nonconcurrent $O(nm^3)$ matrix multiplication operations. The last of these 7 lines is a calculation of the form $v = Aw$, where v and w are complex vectors of length $2m$, and A is a complex matrix with shape $2m \times 2m$. The paper by Bereux [2] suggests that the BLAS function `zgemv` be used for this line.

We constructed a test of this line of code in isolation to compare the performance of NumPy to `zgemv`. The test suggested that for column-major contiguous input arrays, `zgemv` performed 25% faster than NumPy, but for noncontiguous input arrays, `zgemv` performed significantly slower than NumPy.

In this line of code, one of the input arrays is constructed out of other smaller arrays, and contains data which is inherently noncontiguous as result. Because of the way the input arrays at this step are constructed, the input method cannot be modified using transposition to force the input arrays to be column-major contiguous. Thus, the code was not modified to use `zgemv` at this step.

Thus far, we have discussed all of the 7 $O(nm^3)$ matrix multiplication operations. We also considered changes to some $O(nm^2)$ matrix multiplications. One line in the code involves a calculation of the form $a \leftarrow v^\dagger v$, where a is a real scalar and v is a complex vector of length $2m$. While one of these calculations is $O(m)$, it appears in a nested loop which is iterated $(2n-1)(2m)$ times, so the time complexity for this step is $O(nm^2)$.

We constructed a test of this line in isolation to compare the performance of the BLAS function `dznrm2` to NumPy (Table VI). The test suggests that the line of code could be sped up by a factor of 2.3–3.3 by switching from NumPy to `dznrm2`. This change was made in the code, and the total execution time decreased by 0.5%, or 2.2 seconds, for a test with $n = 16$, $m = 512$, and $p = 1024$. While this performance gain was almost negligible, the test suggests that for larger values of m, the relative performance gain should increase.

TABLE VI: Time to compute $a \leftarrow v^\dagger v$, where a is a real number and v is a vector of length n with complex entries. Performance of NumPy is compared to the BLAS routine dznrm2. Execution times are the minima of 1000 repetitions.

n	NumPy (ms)	dznrm2 (ms)	NumPy/dznrm2
512	0.089	0.038	2.35
1024	0.116	0.041	2.84
2048	0.150	0.046	3.28

TABLE VII: Time to compute the inverse of an $n \times n$ upper triangular matrix with complex entries. Performance of the SciPy function inv is compared to the BLAS function ztrtri. Execution times are the minima of 10 repetitions.

n	inv (s)	ztrtri (s)	inv/ztrtri
512	0.204	0.043	4.80
1024	1.278	0.267	4.79
2048	9.371	1.558	6.02

There are numerous lines involving $O(nm^2)$ and $O(nm)$ operations of the form $y \leftarrow y + cx$, where y and x are complex vectors of length $2m$, and c is a complex scalar. Tests of these lines of code in isolation suggested that no appreciable performance increase could be achieved by switching from NumPy to the appropriate BLAS function `zaxpy`, so no changes were made to the code.

2. *Matrix inversion*

So far, we have discussed 7 of the 9 nonconcurrent $O(nm^3)$ operations. The remaining 2 operations involve calculations of the inverse of an upper triangular matrix with complex entries and shape $p \times p$. Since $p \sim m$, and these inversions are each calculated $2n-1$ times, the times complexity of these two steps is $O(nm^3)$.

In the original code, the inverses were computed using the NumPy routine `inv`, which can be used to invert general square matrices. While the paper by Bereux [2] does not suggest a specific routine to use for this inversion, we considered use of the LAPACK routine `ztrtri`, which is optimized for triangular matrices.

We constructed a test in isolation to compare the performance of `inv` to `ztrtri` (Table VII) in inverting triangular matrices. The test suggests that `ztrtri` can compute the inverse 2.5–6 times faster than `inv`. We changed these lines in the factorization to make use of `ztrtri`. As a result, the total execution time decreased by 6%, or 67 seconds, for a test with $n = 16$, $m = 512$, and $p = 1024$. Based on the test in isolation, the improvement of these lines should be more significant for larger values of p.

3. *MPI functions*

The factorization routine uses the Python module mpi4py to pass data between processes. The module mpi4py comes with two versions of many of its functions applicable to different types of data. The first version of each function can be used for generic Python objects, such as lists. These functions have names beginning with a lowercase letter. The second version is for buffer-like objects, such, as NumPy arrays. These functions have names beginning with an uppercase letter. The mpi4py documentation claims that the uppercase versions for buffer-like objects are faster than the lowercase versions for generic Python objects.

The factorization routine involves numerous calls of MPI broadcast. The original code used the lowercase version `bcast` for these steps. We tried changing all of these broadcast calls from the lowercase version to the uppercase version, `Bcast`. This change required some modification of the data being transferred in the broadcast call. For example, a `bcast` call sending the Boolean value `True` or `False` was converted to a `Bcast` call sending a NumPy array containing 1 or 0. Changing all of the `bcast` calls to `Bcast` calls reduced the total execution time of the factorization routine by 9%, or 94 seconds, for a test with $n = 32$, $m = 512$, and $p = 128$.

The factorization routine also involves numerous MPI send/receive calls. We changed all of these from the lowercase versions `send`/`recv` to the uppercase versions `Send`/`Recv`. Again, this change required modifying the form of the data being sent (e.g. changing a Python float64 to a NumPy array with one element). As a result of these changes, the total execution time of the factorization routine decreased by 10%, or 95 seconds, for a test with $n = 32$, $m = 512$, and $p = 128$.

4. *Miscellaneous changes*

In one section of the code which is iterated $(2n-1)(2m)$ times, each process checks its own rank using a for loop. This for loop was replaced with a single if statement. Due to this minor change, the total execution time was reduced by a factor of 2.2%, or 10 seconds, for a test with $n = 16$, $m = 512$, and $p = 1024$.

5. *Optimal operating parameters*

The factorization routine divides the $2m \times 2m$ matrices into smaller $p \times p$ matrices, and performs calculations on these. This functionality was added before this project in order to improve performance of the code. After making the above changes, we tested the performance of the code with different values of p to find the value at which the factorization is performed the fastest (Table VIII). For the test with $m = 512$, the factorization routine works

TABLE VIII: Time to perform the Toeplitz factorization for different values of p, when $n = 16$, $m = 512$. The factorization was performed using 16 ranks-per-node and 4 OpenMP threads per MPI process.

p	$2m/p$	time (s)
1024	1	384.696
512	2	350.134
256	4	331.572
128	8	324.000
64	16	333.411

fastest with $p = m/4$. The factorization performed 16% faster when $p = m/4$ compared to $p = 2m$.

If the value of m is increased beyond 512, the user may wish to find the value of p which optimizes the factorization. This can be done by timing a small number of iterations of the loop over k for different values of p. The factorization routine has not been tested for $m > 512$.

IV. DISCUSSION

A. Summary of changes

As a result of changes made to the code, the total execution time has been reduced by a factor of 4 for the largest data set tested ($n = 16$, $m = 512$). In its current state, for a test with $n = 16$, $m = 512$, and $p = 1024$, the decomposition routine takes 95% longer than the optimal speed, accounting only for matrix multiplication.

Significant improvements were achieved by switching from code written in standard Python/NumPy notation to optimized BLAS and LAPACK routines. The increase in performance for individual lines converted to BLAS/LAPACK routines varies from almost no change to factor of 20 speed increases.

The increased performance of BLAS and LAPACK routines appears to require that the input arrays be column-major contiguous. For noncontiguous input arrays, the same command written in NumPy may significantly outperform a BLAS or LAPACK routine. Thus, if further conversions are made, the lines of code to be changed should be tested in isolation to verify that the BLAS or LAPACK routine actually improves performance.

B. Future improvements

There are numerous ways in which the performance of the factorization routine might be improved further. Currently, the code uses double precision numbers. The performance of the code might be improved by switching to single precision numbers. Preliminary tests suggested that the code would run 5-10% faster by making such a change, but the switch caused the routine to crash after a few iterations of the loop over k for an unknown reason.

While we have attempted to improve all of the computationally expensive parts of the code, further improvements might be achieved by modifying computations in the code which are repeated many times. For example, the functions `__seq_update` and `__house_vec` are called $(2n - 1)(2m)$ times. Even minor improvements to the efficiency of these functions could significantly improve performance.

The Python environment on the BGQ system used for this project currently does not have the most recent versions of the Python modules mpi4py, numpy, or scipy. The performance of the code might be improved somewhat by updating these modules.

The performance of the code could likely be improved significantly if the entire routine was rewritten in Fortran or C, which are generally considered to be faster than Python. Such rewriting would also simplify the use of BLAS and LAPACK functions, which currently require wrappers in a SciPy module which does not contain all BLAS and LAPACK functions. However, given that the code currently runs reasonably close to an idealized optimal speed, the amount of work required to rewrite the code in Fortran or C would likely not be worth the improvement.

[1] N. Afsariardchi, Final report for AST1500, Canadian Institute for Theoretical Astrophysics, University of Toronto (2011).

[2] N. Bereux, Linear Algebra and it's Applications **404**, 193 (2005).