

Parallelizing the Conjugate Gradient Algorithm for Multilevel Toeplitz Systems

Jie Chen

*Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60517
Email: jiechen@mcs.anl.gov*

Lihua Li

*Department of Mathematics and Computer Science
University of Missouri–St. Louis
St. Louis, MO 63121
Email: ll9n8@mail.umsl.edu*

Abstract—Multilevel Toeplitz linear systems appear in a wide range of scientific and engineering applications. While several fast direct solvers exist for the basic 1-level Toeplitz matrices, in the multilevel case an iterative solver provides the most general and practical solution. Furthermore, iterative methods are asymptotically faster than many stable direct methods even for the 1-level case. This paper proposes several parallelization techniques that enable an efficient implementation of the conjugate gradient algorithm for solving multilevel Toeplitz systems on distributed-memory machines. The two major differences between this implementation and that for a general sparse linear solver are (1) a communication-efficient approach to handle data expansion and truncation and data transpose simultaneously; (2) the interleaving of matrix-vector multiplications and vector inner product calculations to reduce synchronization cost and latency. Similar ideas can be applied to the implementation of other iterative methods for Toeplitz systems that are not necessarily symmetric positive definite. Scaling results are shown to demonstrate the usefulness of the proposed techniques.

Keywords-Toeplitz; conjugate gradient; FFT; all-reduction

I. INTRODUCTION

Many scientific and engineering applications give rise to (multilevel) Toeplitz linear systems. Examples include solving partial differential equations, integral equations, digital signal processing, image processing, optimal control, and stationary time series. A Toeplitz matrix has constant diagonals. This special structure enables the development of algorithms that run faster than $O(n^3)$, which is the cost of solving a general, unstructured dense linear system using a direct method. An extensive literature has been devoted to the solution of Toeplitz systems (with parallel implementation), including fast algorithms (such as Levinson-Durbin [1]–[3] and Bareiss [4], [5]), “superfast” algorithms using divide-and-conquer strategies (see, e.g., [6]–[12]), iterative algorithms based on Newton iterations [13], and algorithms for banded Toeplitz systems (see, e.g., [14], [15]). Some of the algorithms can be generalized for block-Toeplitz or Toeplitz-block systems [16].

A multilevel Toeplitz matrix is defined recursively with respect to the number of levels (sometimes the term “multilevel” is dropped for convenience if the distinction with 1-level is unimportant). In the simplest case, a 2-level Toeplitz matrix is a block-Toeplitz matrix where each block

itself is Toeplitz. A d -level Toeplitz matrix usually comes from a problem with a d -dimensional regular grid structure. Compared with the abundance of algorithms for 1-level Toeplitz systems, algorithms for multilevel Toeplitz systems are rare. One of the reasons is that extending the above algorithms to fully utilize the recursive Toeplitz structure is not straightforward.

Iterative methods (Krylov subspace methods) provide a flexible set of algorithms for solving general linear systems. For symmetric positive definite Toeplitz systems, the conjugate gradient (CG) algorithm with circulant, banded, or similar preconditioners was primarily studied (see, e.g., [17]–[21]). Other Krylov algorithms are also applicable; for example, MINRES and GMRES can be used to solve indefinite and unsymmetric Toeplitz systems, respectively (see [22] for a comprehensive treatment of iterative methods). Two crucial computational concerns in applying an iterative method are the efficient multiplication of the matrix with a vector and the efficient construction and application of a preconditioner. For multilevel Toeplitz matrices, the matrix-vector multiplication can be carried out by using fast Fourier transforms (FFTs), as can the application of the preconditioner if it is circulant. This technique lays down the foundation for using iterative methods for solving systems with an arbitrary number of Toeplitz levels.

This paper discusses the parallelization of CG for multilevel Toeplitz systems on distributed-memory machines. The matrix-vector multiplication in this case differs significantly from that in the sparse case. A sparse matrix often arises from discretizations (for example, of differential operators). By domain decomposition, the communication in the multiplication is local. On the other hand, the FFT for multiplying circulant or Toeplitz matrices requires global communications (all-to-all type) because data transpose is needed. What complicates this multiplication is that the Toeplitz matrix and the vector must be expanded in size (called embedding) before the multiplication, and the results must be truncated to yield a correct-sized vector after the multiplication. If not properly implemented, the embedding and truncation incur extra communications. We propose interleaving the embedding and truncation with each substep of the FFT calculation, so that the former can take advantage

of the communications in the latter in order to save the extra cost.

The second improvement of the proposed implementation is the handling of vector inner product or norm calculations (to facilitate presentation, throughout this paper we use “inner product” to mean either the inner product or the norm). Inner product calculations are a common bottleneck in parallel iterative solvers because they require global synchronizations. Especially for sparse solvers, synchronizations can constitute over 50% of the overall run time when the CPU cores grow to over thousands. A focus of modern sparse solver design is to modify the numerical algorithm in order to reduce the number of inner product calculations (see, e.g., [23]–[27]). In our case, it is also beneficial to consider how to reduce the synchronizations. One will see that with simple mathematical derivations the inner products need not be computed *after* the matrix-vector products, even though in the original CG algorithm it appears so. The inner products can be equivalently expressed by using intermediate results in the matrix-vector multiplication. Therefore, we can utilize the all-to-all communications required in the multiplications to simultaneously compute the inner products, thus eliminating the synchronizations and reducing latency.

II. PRELIMINARIES

A. Circulant Matrices, Toeplitz Matrices and, Matrix-Vector Multiplications

A circulant matrix C and a Toeplitz matrix T , of order n , are defined in the following forms, respectively:

$$C = \begin{bmatrix} c_0 & c_{n-1} & c_{n-2} & \cdots & \cdots & c_1 \\ c_1 & c_0 & c_{n-1} & \ddots & & \vdots \\ c_2 & c_1 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & c_{n-1} & c_{n-2} \\ \vdots & & \ddots & c_1 & c_0 & c_{n-1} \\ c_{n-1} & \cdots & \cdots & c_2 & c_1 & c_0 \end{bmatrix}, \quad (1)$$

$$T = \begin{bmatrix} t_0 & t_{-1} & t_{-2} & \cdots & \cdots & t_{-n+1} \\ t_1 & t_0 & t_{-1} & \ddots & & \vdots \\ t_2 & t_1 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & t_{-1} & t_{-2} \\ \vdots & & \ddots & t_1 & t_0 & t_{-1} \\ t_{n-1} & \cdots & \cdots & t_2 & t_1 & t_0 \end{bmatrix}.$$

We will use a subscript n to emphasize the order when necessary. Since this paper addresses real symmetric matrices, C and T can be represented solely by their first columns, $\mathbf{c} = [c_0, \dots, c_{n-1}]$ and $\mathbf{t} = [t_0, \dots, t_{n-1}]$, respectively. Note that the Toeplitz matrix T_n can be embedded into a circulant

matrix C_{2n} (of twice the size):

$$C_{2n} = \begin{bmatrix} T_n & * \\ * & T_n \end{bmatrix}.$$

All the elements (except for the diagonal ones) in the two subblocks denoted by $*$ are well defined. In the vector representation, this is

$$\mathbf{c}_i = \begin{cases} \mathbf{t}_i & i = 0, \dots, n-1, \\ \text{arbitrary} & i = n, \\ \mathbf{t}_{2n-i} & i = n+1, \dots, 2n-1. \end{cases}$$

Informally, the first half of \mathbf{c} is \mathbf{t} , whereas the latter half is a “flipping” of \mathbf{t} except for the first entry.

The multiplication of C with a vector \mathbf{y} utilizes the fact that C can be diagonalized by a discrete Fourier transform (DFT). Specifically, the diagonalization is

$$UCU^H = \Lambda,$$

where $\Lambda = \text{diag}(\lambda_1, \dots, \lambda_n)$ contains the eigenvalues of C and U is the DFT matrix with entries $U_{jk} = \exp(i2\pi jk/n)/\sqrt{n}$. Multiplying the vector of all ones on both sides of the above equation, we obtain

$$\sqrt{n}U\mathbf{c} = \lambda,$$

where λ denotes the vector of all eigenvalues. This means that the eigenvalues of C are obtained by a DFT of the first column and a scaling \sqrt{n} . Therefore, the matrix-vector product $\mathbf{v} = C\mathbf{y}$ can be computed by using FFTs; see Algorithm 1.

Algorithm 1 Circulant matrix times vector

- 1: Pre-compute λ as the FFT of \mathbf{c} multiplied by \sqrt{n}
 - 2: Compute \mathbf{z} as the FFT of \mathbf{y}
 - 3: Compute \mathbf{w} as the elementwise product of λ and \mathbf{z}
 - 4: Obtain the result \mathbf{v} as the inverse FFT of \mathbf{w}
-

The multiplication of T with a vector \mathbf{y} exploits the circulant embedding of T :

$$\begin{bmatrix} T\mathbf{y} \\ * \end{bmatrix} = \begin{bmatrix} T & * \\ * & T \end{bmatrix} \begin{bmatrix} \mathbf{y} \\ \mathbf{0} \end{bmatrix}.$$

Algorithm 2 shows the steps of computing the matrix-vector product $\mathbf{v} = T\mathbf{y}$.

Algorithm 2 Toeplitz matrix times vector

- 1: Embed the matrix T_n into a circulant matrix C_{2n}
 - 2: Embed \mathbf{y} into a length- $2n$ vector \mathbf{y}' (all remaining entries being zero)
 - 3: Multiply C_{2n} by \mathbf{y}' using Algorithm 1, obtaining \mathbf{v}'
 - 4: Truncate \mathbf{v}' (keeping the first n entries) to obtain \mathbf{v}
-

Multilevel circulant matrices are defined recursively with respect to the number of levels. Thus it suffices to define

the 2-level case. With the form (1), if each c_i itself is a circulant block, then the resulting matrix is 2-level circulant. We use $C_{n_0 n_1}$ to denote such a matrix, where each c_i has size $n_0 \times n_0$, and there are n_1 such c_i 's. We say that $C_{n_0 n_1}$ is of order (n_0, n_1) . A 2-level circulant matrix can also be represented solely by its first column. The first column of $C_{n_0 n_1}$ consists of n_1 subvectors of length n_0 , where each subvector corresponds to the first column of some c_i . We pack these subvectors column by column, obtaining an $n_0 \times n_1$ data tensor \mathbf{c} . We formally call \mathbf{c} a (data) representation of $C_{n_0 n_1}$.

If we generalize the number of levels, a d -level circulant matrix $C_{n_0 \dots n_{d-1}}$ is of order (n_0, \dots, n_{d-1}) . It can be represented solely by its first column, which can then be reshaped as an $n_0 \times \dots \times n_{d-1}$ tensor. We always use \mathbf{c} to denote this tensor. Similar definitions and discussions apply to a d -level Toeplitz matrix $T_{n_0 \dots n_{d-1}}$, which is represented by \mathbf{t} , an $n_0 \times \dots \times n_{d-1}$ tensor. To simplify notation, we always let $n = n_0 \dots n_{d-1}$.

Multiplying a d -level circulant matrix C with a vector \mathbf{y} can still use Algorithm 1, with minor changes:

- The notation \mathbf{y} means an $n_0 \times \dots \times n_{d-1}$ tensor, a reshaped result of \mathbf{y} , and similarly for the output.
- The (inverse) FFTs are changed to d -dimensional (inverse) FFTs.

The steps of multiplying a d -level Toeplitz matrix T with a vector \mathbf{y} are similarly modified from Algorithm 2, except that the embeddings of T and \mathbf{y} need more descriptions. It suffices to show an example in the 2-level case. Recall that the notation \mathbf{y} means an $n_0 \times n_1$ tensor. Then the embedding is

$$\mathbf{y}' = \begin{bmatrix} \mathbf{y} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix},$$

which is of size $2n_0 \times 2n_1$. That is, the embedding should expand along each direction of the tensor \mathbf{y} . Similarly, the embedding of \mathbf{t} has the form

$$\mathbf{c} = \begin{bmatrix} \mathbf{t} & * \\ * & * \end{bmatrix},$$

where the blocks denoted by $*$ contain the ‘‘flipping’’ of \mathbf{t} . Formally, for $i = 0, \dots, n_0 - 1$,

$$\mathbf{c}_{ij} = \begin{cases} \mathbf{t}_{ij} & j = 0, \dots, n_1 - 1, \\ \text{arbitrary} & j = n_1, \\ \mathbf{t}_{i, 2n_1 - j} & j = n_1 + 1, \dots, 2n_1 - 1. \end{cases}$$

Then, for $j = 0, \dots, 2n_1 - 1$,

$$\mathbf{c}_{ij} = \begin{cases} \mathbf{t}_{ij} & i = 0, \dots, n_0 - 1, \\ \text{arbitrary} & i = n_0, \\ \mathbf{t}_{2n_0 - i, j} & i = n_0 + 1, \dots, 2n_0 - 1. \end{cases}$$

B. Conjugate Gradient with Toeplitz Systems

The standard CG algorithm [22] for solving a linear system $A\mathbf{x} = \mathbf{b}$ using a preconditioner $M \approx A$ is shown in Algorithm 3. It requires an initial guess \mathbf{x}_0 . The CG iteration (the for-loop) is run until the residual \mathbf{r}_{j+1} is sufficiently small. At such a step j , the algorithm returns \mathbf{x}_{j+1} as the approximate solution.

Algorithm 3 CG

```

1:  $\mathbf{r}_0 = \mathbf{b} - A\mathbf{x}_0$ ,  $\mathbf{z}_0 = M^{-1}\mathbf{r}_0$ ,  $\mathbf{p}_0 = \mathbf{z}_0$ 
2: for  $j = 0, 1, \dots, \text{maxit}$  do
3:    $\alpha_j = (\mathbf{r}_j, \mathbf{z}_j) / (A\mathbf{p}_j, \mathbf{p}_j)$ 
4:    $\mathbf{x}_{j+1} = \mathbf{x}_j + \alpha_j \mathbf{p}_j$ 
5:    $\mathbf{r}_{j+1} = \mathbf{r}_j - \alpha_j A\mathbf{p}_j$ 
6:   if  $\|\mathbf{r}_{j+1}\| / \|\mathbf{b}\| < \text{rtol}$  then return
7:    $\mathbf{z}_{j+1} = M^{-1}\mathbf{r}_{j+1}$ 
8:    $\beta_j = (\mathbf{r}_{j+1}, \mathbf{z}_{j+1}) / (\mathbf{r}_j, \mathbf{z}_j)$ 
9:    $\mathbf{p}_{j+1} = \mathbf{z}_{j+1} + \beta_j \mathbf{p}_j$ 
10: end for

```

When A is 1-level Toeplitz, several choices of a circulant preconditioner exist. These preconditioners all help the CG algorithm converge superlinearly [17]. Although the superlinear convergence property is lost in the multilevel case [28], the corresponding multilevel circulant preconditioners yield sufficiently good performance in practice [17]. In this paper we consider T. Chan’s preconditioner [19].

Using the notation in the preceding subsection, let the $n_0 \times \dots \times n_{d-1}$ tensors \mathbf{a} and \mathbf{m} represent the matrices A and M , respectively. In the 1-level case ($d = 1$), the preconditioner is defined as

$$\mathbf{m}_j = ((n - j)\mathbf{a}_j + j\mathbf{a}_{n-j})/n, \quad j = 0, \dots, n - 1.$$

Informally, \mathbf{m} is a weighted averaging of \mathbf{a} and its flipping, and the weights are defined with respect to the locations of the entries. Then, in the general d -level case, the averaging is done along each dimension. Algorithm 4 summarizes this averaging procedure (in the subscript notation such as ‘‘ \dots, j, \dots ’’, j is the i th index).

Algorithm 4 Constructing preconditioner M

```

1: Assign  $\mathbf{m} \leftarrow \mathbf{a}$ 
2: for  $i = 0, \dots, d - 1$  do
3:   for  $j = 0, \dots, \lfloor n_i/2 \rfloor - 1$  do
4:      $\mathbf{m}_{\dots, j, \dots} \leftarrow ((n_i - j)\mathbf{m}_{\dots, j, \dots} + j\mathbf{m}_{\dots, n_i - j, \dots})/n_i$ 
5:      $\mathbf{m}_{\dots, n_i - j, \dots} \leftarrow \mathbf{m}_{\dots, j, \dots}$ 
6:   end for
7: end for

```

C. Parallel Multidimensional FFT

Multidimensional FFTs are needed to apply a multilevel circulant preconditioner M to vectors. More important, multidimensional FFTs are required to multiply the multilevel

Toeplitz matrix A with vectors. We use \mathbf{z} to mean a general $n_0 \times \cdots \times n_{d-1}$ data tensor in the complex field. A d -dimensional FFT on \mathbf{z} is equivalent to d consecutive 1-dimensional FFTs along each dimension of \mathbf{z} .

For parallel implementation, two data partitioning methods are the most popular. The first method is to cut \mathbf{z} along the first dimension. An example of this approach is implemented in the FFTW library [29]. Fixing the first index, all the data are local, residing within one processor. Therefore, a $(d-1)$ -dimensional FFT is first performed along the 2nd to the d th dimension. Then, a transpose is carried out so that data along the first dimension becomes local. A remaining 1-dimensional FFT is performed to complete the whole transform.

This method has two drawbacks. First, the maximum number of processors is restricted by the size of the first dimension. Second, the transpose operation requires global communication using *all* processors on the *whole* data.

In order to overcome these drawbacks, the second method is to use a d' -dimensional grid of processors ($d' < d$) and to partition the data \mathbf{z} along d' dimensions. A simplified example of this approach is implemented in the P3DFFT library [30], which handles 3-dimensional data using a 2-dimensional grid of processors. In the most general setting, without loss of generality we assume that \mathbf{z} is partitioned along the 1st to the d' th dimension. The transform is carried out by first performing a $(d-d')$ -dimensional FFT along the $(d'+1)$ -th to the d th dimension. Then, to facilitate presentation, we combine the dimensionality of these dimensions and regard them as one dimension only; we call it the “last dimension.” Next, the procedure continues with a loop that iterates through the first d' dimensions. For the loop index i from d' down to 1, a transpose is performed between the i th and the $(i+1)$ -th dimension (so that the data along the i th dimension become local), followed by a 1-dimensional FFT along the i th dimension. Note that if $i = d'$, the $(i+1)$ -th dimension is the “last dimension” instead of the original $(d'+1)$ -th dimension. When the loop finishes, the whole transform is complete.

An obvious advantage of this method is that the number of processors can maximally scale to the product of the size of the first d' dimensions. More important, within the loop each transpose requires the participation of only subgroups of processors. For example, at the i th iteration, the subgroup of processors along the i th dimension participate in the transpose, but different subgroups are independent. This reduces the data size in the all-to-all type of communications and is more desirable when \mathbf{z} has many dimensions.

Note that the first method can be considered a special case of the second method (by letting $d' = 1$). In this paper we compare the performance of 1- and 2-dimensional partitioning on 3-dimensional data in experiments.

III. SKELETON OF THE ALGORITHM

Based on the preceding section, the mathematically oriented Algorithm 3 is now rewritten as Algorithm 5 that is tailored for Toeplitz systems and parallel implementations. The major changes are twofold:

- The eigenvalue calculation of the embedding of A and that of M as a preprocessing step is brought to the front. This step can be separated from the main CG iteration for the case of multiple right-hand sides.
- Several intermediate variables (τ_j and σ_j) are introduced to facilitate computer implementation and further improvements (see Section V).

Algorithm 5 CG for multilevel Toeplitz systems

```

// The following can be separated out for multiple  $\mathbf{b}$ 's
1: Call TOEP-EMBED-EIGVAL( $A, \Lambda_1$ ) to obtain eigenvalues  $\Lambda_1$  of the multilevel circulant embedding of  $A$ 
2: Construct multilevel circulant preconditioner  $M$ 
3: Compute eigenvalues  $\Lambda_2$  of  $M$ 
4: Call TOEP-MULT( $\Lambda_1, \mathbf{x}_0, \mathbf{y}_0$ ) to obtain  $\mathbf{y}_0 = A\mathbf{x}_0$ 

// Work for each  $\mathbf{b}$ 
5:  $\gamma = \|\mathbf{b}\|$ 
6:  $\mathbf{r}_0 = \mathbf{b} - \mathbf{y}_0$ 
7:  $\rho_0 = \|\mathbf{r}_0\|$ 
8: if  $\rho_0/\gamma < rtol$  then return
9: Compute  $\mathbf{z}_0 = M^{-1}\mathbf{r}_0$  using  $\Lambda_2^{-1}$ 
10:  $\sigma_0 = (\mathbf{r}_0, \mathbf{z}_0)$ 
11:  $\mathbf{p}_0 = \mathbf{z}_0$ 
12: for  $j = 0, 1, \dots, maxit$  do
13:   Call TOEP-MULT( $\Lambda_1, \mathbf{p}_j, \mathbf{v}_j$ ) to obtain  $\mathbf{v}_j = A\mathbf{p}_j$ 
14:    $\tau_j = (\mathbf{v}_j, \mathbf{p}_j)$ 
15:    $\alpha_j = \sigma_j/\tau_j$ 
16:    $\mathbf{x}_{j+1} = \mathbf{x}_j + \alpha_j\mathbf{p}_j$ 
17:    $\mathbf{r}_{j+1} = \mathbf{r}_j - \alpha_j\mathbf{v}_j$ 
18:    $\rho_{j+1} = \|\mathbf{r}_{j+1}\|$ 
19:   if  $\rho_{j+1}/\gamma < rtol$  then return
20:   Compute  $\mathbf{z}_{j+1} = M^{-1}\mathbf{r}_{j+1}$  using  $\Lambda_2^{-1}$ 
21:    $\sigma_{j+1} = (\mathbf{r}_{j+1}, \mathbf{z}_{j+1})$ 
22:    $\beta_j = \sigma_{j+1}/\sigma_j$ 
23:    $\mathbf{p}_{j+1} = \mathbf{z}_{j+1} + \beta_j\mathbf{p}_j$ 
24: end for

```

Let us examine the algorithm line by line. First, the subroutines TOEP-EMBED-EIGVAL and TOEP-MULT compute eigenvalues of the embedding of a Toeplitz matrix and the Toeplitz matrix-vector product, respectively. They require special treatments, and they will be discussed in detail in the next section. We skip the lines with these subroutines here.

Line 2 constructs the preconditioner, following Algorithm 4. Since \mathbf{a} is partitioned along one or more of the dimensions, when looping over these dimensions transposes

have to be performed so that data along these dimensions become local. This procedure follows the same idea of parallel FFT, where there is a loop containing transposes. At the end, the data partitioning of m is different from what it started with. Then, m needs to be transposed back so that the partitioning agrees with that of a .

Line 3 computes the eigenvalues of a multilevel circulant matrix, which simply requires a multidimensional FFT with a scaling (line 1 of Algorithm 1). In fact, the scaling can be removed because it does not affect the convergence behavior of CG.

Lines 9 and 20 multiply a circulant matrix with a vector, following Algorithm 1 (from line 2 to the end), with modifications for multilevels as discussed in Section II-A.

The remaining lines involve scalar or vector operations, and the implementation is straightforward.

IV. PARALLEL TOEPLITZ MATRIX-VECTOR MULTIPLICATION

One difficulty of naively using Algorithm 2 (even with the modifications for multilevels) for multiplying a Toeplitz matrix with a vector is that the embedding is expensive. The reason is that the embedding along certain dimensions that are partitioned by processors requires interprocessor communications. As an example, consider a 2-level Toeplitz matrix T . Its representation is a 2-dimensional data tensor t . Then the embedding is in the form

$$c = \begin{bmatrix} t & * \\ * & * \end{bmatrix}.$$

Suppose t is partitioned horizontally by using two processors. In order that c is similarly partitioned, one needs to fill the lower part of c . Furthermore, one needs to redistribute the data (for example, the upper part of c is now held by only one processor). Communication is inevitable if one wants to construct the embedding c explicitly.

To avoid this communication, one can combine the steps of the embedding and the FFT, since the embedding can take advantage of the transposes in the FFT for data redistribution. The details are best explained by walking through Figure 1(a), which showcases the situation that the data is partitioned along only one dimension. The flowchart mainly demonstrates the steps of the multiplication; however, the first phase of the flow can also be used to compute the eigenvalues of the embedding of the Toeplitz matrix. The correctness of the embedding and the multiplication will be shown later.

To compute $v = Ty$, we need to embed y into a larger data tensor by padding zeros. The first step is to embed along the 2nd to the d th dimensions, shown in the figure as “row-plane embedding.” After the embedding, we immediately perform a $(d - 1)$ -dimensional FFT on these dimensions. Next, a transpose between the 1st dimension and the 2nd to d th dimensions is carried out, so that data along the

1st dimension become local. Then, we perform a further embedding (zero-padding) along the 1st dimension, followed immediately by a 1-dimensional FFT along this dimension. This in fact completes the d -dimensional FFT on the full embedding of y . The result is denoted by z .

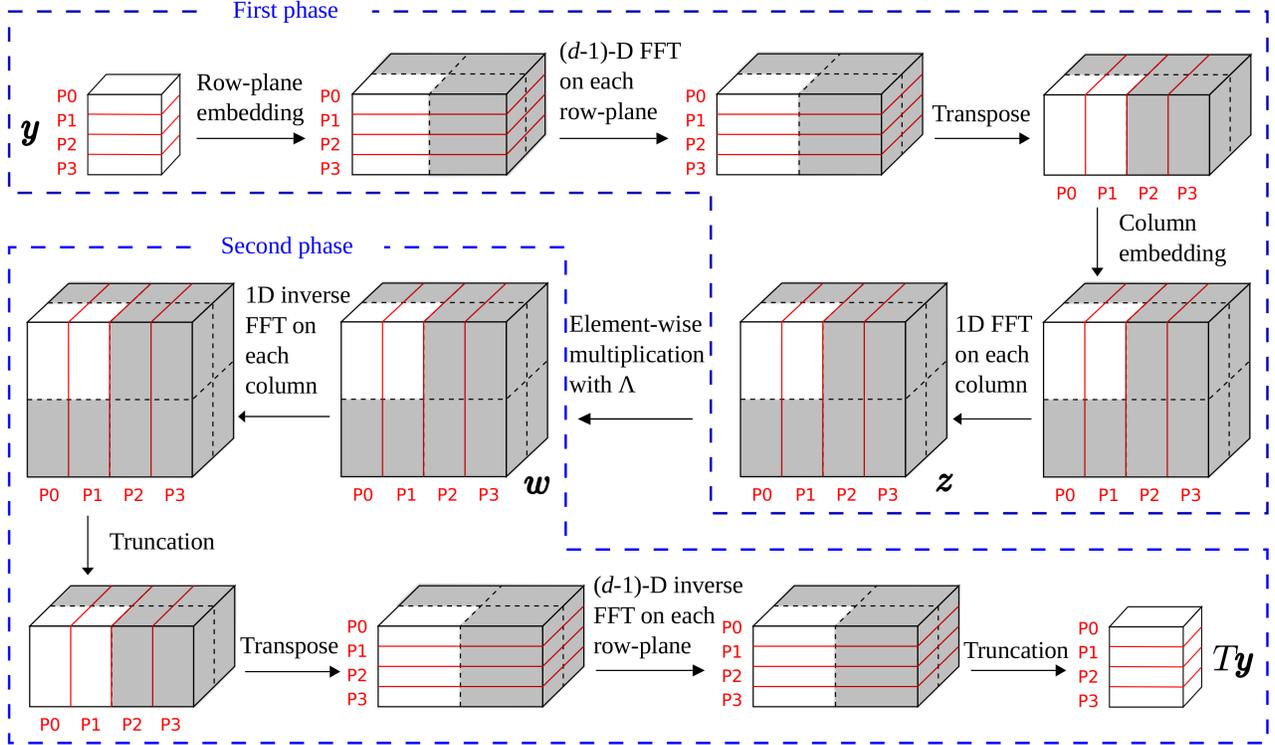
The procedure for obtaining the eigenvalues λ of the embedding of t is similar, except that the “zero-padding” is changed to “flipping.” Corresponding to the figure, the first flipping is to embed each row-plane to a larger data tensor such that it represents a $(d - 1)$ -level circulant matrix, and the second flipping is to embed each column to a longer column such that it represents a 1-level circulant matrix.

With z and λ ready, an elementwise multiplication is performed, resulting in w . The final step is to perform a d -dimensional inverse FFT and the truncation simultaneously, in order to obtain the final result v . This, in fact, is the reverse procedure of obtaining z , and it corresponds to the second phase of Figure 1(a). We perform a 1-dimensional inverse FFT along the 1st dimension, followed by a truncation that rids the latter half of the data along this dimension. Next, a transpose is performed so that the data is cut along the 1st dimension. Then, a $(d - 1)$ -dimensional inverse FFT along the 2nd to d th dimensions is carried out, and the data is truncated to half along each of these dimensions. The result is the data v of dimension $n_0 \times \dots \times n_{d-1}$, the desired matrix-vector product.

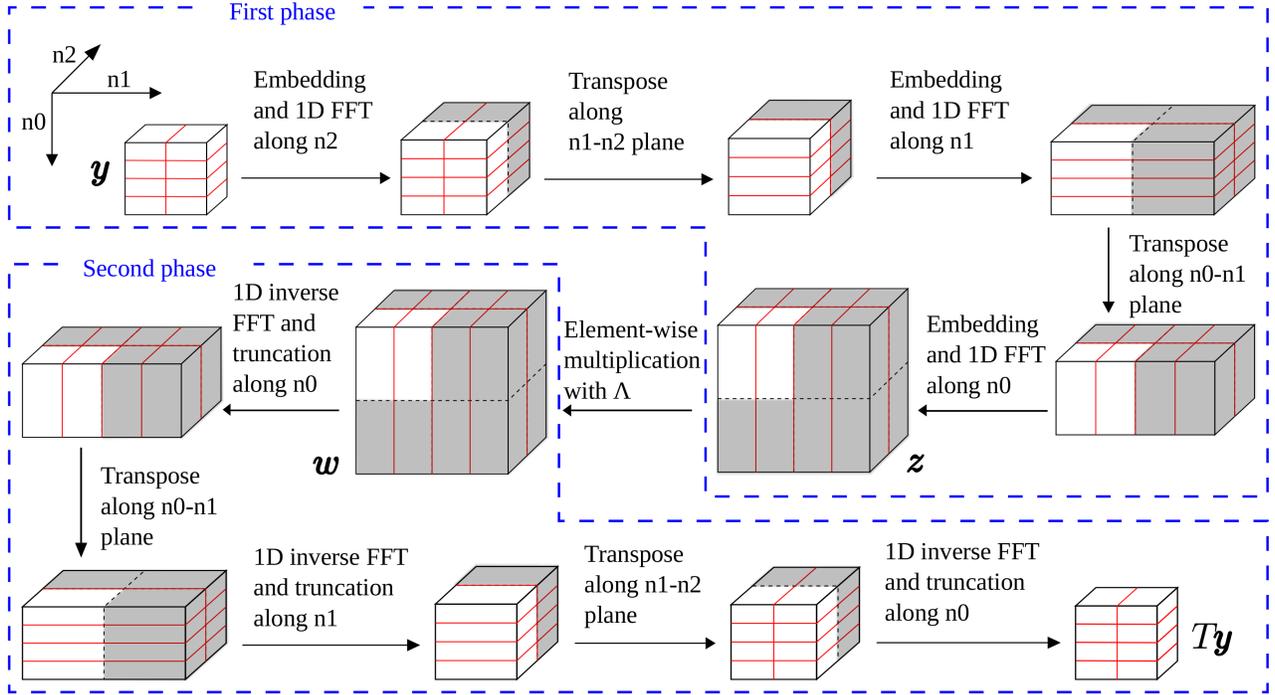
It is important to show that the combination of embedding and FFT (also the combination of inverse FFT and truncation) yields correct results, because the former is not fully done before the latter happens. The proof is simple. For zero-padding, the FFT of zeros is zeros, so the embedding along the 1st dimension can be delayed until the data along this dimension becomes local. Similarly, flipping the data entries after FFT is equivalent to first performing FFT, then flipping the FFT result. Furthermore, the truncation of data along any dimension can be done at any time as long as FFT along this dimension has been done. Therefore, the truncations along all dimensions need not be done all at once at the end. This completes the proof of the correctness of Figure 1(a).

Similar ideas apply to the situation where the data is partitioned along more than one dimension. Figure 1(b) shows an example of 3-dimensional data partitioned along the first two dimensions. The procedure begins with embedding and FFT along the unpartitioned dimension(s), then iteratively performing transposes so that each time the data along a new dimension becomes local; thus, embedding and FFT are done along this dimension. After the iterative loop, both z and λ are ready, so an elementwise multiplication is performed, resulting in w . Then, w goes through the reverse procedure (inverse FFT and truncation), and the matrix-vector product v is obtained. The correctness of the procedure follows the arguments for the case of Figure 1(a).

The discussions are summarized in Algorithm 6. This algorithm is a parallel improvement of Algorithm 2. The key



(a) 1-dimensional partitioning



(b) 2-dimensional partitioning

Figure 1. Flowcharts of computing Toeplitz matrix-vector product $\mathbf{v} = T\mathbf{y}$. This procedure corresponds to the subroutine TOEP-MULT($\Lambda, \mathbf{y}, \mathbf{v}$) in Algorithm 5, where Λ contains the eigenvalues of the embedding of T . The first phase of each chart also shows the steps of performing the subroutine TOEP-EMBED-EIGVAL(T, Λ).

of the improvement lies in the interleaving of embedding and FFT (also inverse FFT and truncation) along each dimension.

Algorithm 6 Toeplitz matrix times vector, parallel version

- 1: **function** TOEP-EMBED-EIGVAL(T, Λ)
 - 2: Embed t and compute d -dimensional FFT of the embedding simultaneously (corresponding to the first phase of Figure 1(a)/1(b); embedding means “flipping”).
 - 3: Obtain λ by scaling the above result by \sqrt{n} .
 - 4: **end function**

 - 5: **function** TOEP-MULT($\Lambda, \mathbf{y}, \mathbf{v}$)
 - 6: Embed \mathbf{y} and compute d -dimensional FFT of the embedding simultaneously (corresponding to the first phase of Figure 1(a)/1(b); embedding means “zero-padding”). Let the result be \mathbf{z} .
 - 7: Compute \mathbf{w} as the elementwise product of λ and \mathbf{z} .
 - 8: Compute d -dimensional inverse FFT of \mathbf{w} and truncate the FFT result simultaneously (corresponding to the second phase of Figure 1(a)/1(b)). The end result is \mathbf{v} .
 - 9: **end function**
-

It is interesting to count the number of transposes and the involving data size in order to complete one matrix-vector multiplication (excluding the precomputation of the eigenvalues λ). We need only to count the most general case: d -dimensional data with d' -dimensional partitioning. Let the grid of processors have a size $p_0 \times p_1 \times \dots \times p_{d'-1}$, and let p be the total number of processors. After the first embedding, the data size becomes $(n_0 \times \dots \times n_{d'-1}) \times (2n_{d'} \times \dots \times 2n_{d-1}) = 2^{d-d'} n$. Then a transpose between the d' th dimension and the “last dimension” (see Section II-C) is performed. Therefore, there are $p/p_{d'-1}$ all-to-alls occurring simultaneously, and each all-to-all involves a group of $p_{d'-1}$ processors with data size $2^{d-d'} n / (p/p_{d'-1})$. After the transpose, data embedding is done along the d' th dimension, so the total data size doubles. Repeatedly doing this analysis, one easily concludes that there are d' transposes in the first phase. For each transpose, the data size of each all-to-all and the number of all-to-alls are listed in the following table:

i th transp.	size each all-to-all	# concur. all-to-alls
1	$2^{d-d'} p_{d'-1} \cdot n/p$	$p/p_{d'-1}$
2	$2^{d-d'+1} p_{d'-2} \cdot n/p$	$p/p_{d'-2}$
\vdots	\vdots	\vdots
d'	$2^{d-1} p_0 \cdot n/p$	p/p_0

The analysis of the second phase reverses the numbers listed in the table.

V. ELIMINATING ALL-REDUCTION

A common bottleneck in parallel iterative solvers is the inner product calculations, because each calculation requires

an all-reduction operation to sum the local inner products held in each processor in order to obtain the global result. The all-reduction incurs a global synchronization, and there are several such synchronizations within each iteration.

The idea of removing these repeating synchronizations is to hide the latency in other global communications (in this algorithm, the all-to-alls for transposing data). If the data is partitioned along only one dimension, all processors participate in the all-to-all. Thus, *if the data for all-reduction is available at the time of transpose*, then the all-reduction can be equivalently carried out by first performing a local (partial) sum, then transmitting the partial sum by all-to-all, followed by a summation of the gathered partial sums. The implementation is straightforward.

On the other hand, if the data is partitioned along more than one dimension, the participants of each all-to-all consist of only a subgroup of processors, and several all-to-alls are needed to complete one multidimensional FFT. Therefore, the calculation of the sum will be completed only after the all-to-alls are done along every dimension of the processor grid. It suffices to show an example of a 2-dimensional grid of processors. We use (i, j) to indicate a grid location. To sum the number a_{ij} held in each processor, the first batch of all-to-alls exchanges a_{ij} along the rows of the processor grid so that each processor in the i th row has a partial sum $a_i = \sum_j a_{ij}$. Then the second batch of all-to-alls exchanges a_i along the columns of the processor grid. Thus each processor has the final sum $a = \sum_i a_i$.

It is, however, not obvious in Algorithm 5 why the data for all-reduction is ready when transpose is being carried out. For example, in the algorithm one seems to need to obtain $\mathbf{v}_j = A\mathbf{p}_j$ before the inner product $\tau_j = (\mathbf{v}_j, \mathbf{p}_j)$ can be computed. To compute the two terms simultaneously, we derive a mathematically equivalent inner product for τ_j .

To express terms in general, we consider the Toeplitz multiplication $\mathbf{v} = T\mathbf{y}$ and the inner product $\sigma = (\mathbf{y}, \mathbf{v})$. Let \mathbf{y}' be the embedding of \mathbf{y} , and let $C = U^H \Lambda U$ be the diagonalization of the embedding C of T . Then the essence of computing \mathbf{v} consists of the following steps:

- 1) $\mathbf{z} = U\mathbf{y}'$ (first phase of Figure 1(a)/1(b));
- 2) $\mathbf{w} = \Lambda\mathbf{z}$;
- 3) $\mathbf{v}' = U^H\mathbf{w}$, and \mathbf{v} is the truncation of \mathbf{v}' (second phase of Figure 1(a)/1(b)).

Then, the inner product can be equivalently expressed in the following way:

$$(\mathbf{y}, \mathbf{v}) = (\mathbf{y}', \mathbf{v}') = (\mathbf{w}, \mathbf{z}).$$

The first equality results from the fact that \mathbf{y} is embedded into \mathbf{y}' with zeros, and the second equality is because U is unitary. Therefore, σ can be computed as the inner product of \mathbf{w} and \mathbf{z} , which are readily available before step 3.

Incorporating this idea into the CG iteration, we introduce two new subroutines, TOEP-MULT-AND-DOT-

PROD($\Lambda, \mathbf{y}, \mathbf{v}, \sigma$) and CIRC-MULT-AND-DOT-PROD-AND-CONVG-TEST($\Lambda, \mathbf{y}, \mathbf{v}, \sigma, tol$). The former subroutine computes the Toeplitz matrix-vector product $\mathbf{v} = T\mathbf{y}$ and the inner product $\sigma = (\mathbf{y}, \mathbf{v})$ simultaneously. The latter subroutine first computes $\rho = \|\mathbf{y}\|$. If $\rho < tol$, it returns with the indication of convergence of the CG iteration. Otherwise, it computes the circulant matrix-vector product $\mathbf{v} = C\mathbf{y}$ and the inner product $\sigma = (\mathbf{y}, \mathbf{v})$ simultaneously.

The implementation of TOEP-MULT-AND-DOT-PROD has been clear from the preceding discussions. The same idea is used in implementing CIRC-MULT-AND-DOT-PROD-AND-CONVG-TEST($\Lambda, \mathbf{y}, \mathbf{v}, \sigma, tol$). To be specific, the essence of computing $\mathbf{v} = C\mathbf{y}$ consists of the following steps (where C is diagonalized as $U^H\Lambda U$):

- 1) $\mathbf{z} = U\mathbf{y}$;
- 2) $\mathbf{w} = \Lambda\mathbf{z}$;
- 3) $\mathbf{v} = U^H\mathbf{w}$.

Therefore,

$$\sigma = (\mathbf{y}, \mathbf{v}) = (\mathbf{w}, \mathbf{z}),$$

and thus its computation is inserted into step 3. Furthermore, the computation of $\rho = \|\mathbf{y}\|$ is inserted into step 1. If $\rho < tol$, CIRC-MULT-AND-DOT-PROD-AND-CONVG-TEST returns indication of convergence, and steps 2 and 3 are skipped.

These two new subroutines are summarized in Algorithm 7. Using these subroutines, Algorithm 8 is the final algorithm of this paper.

Algorithm 7 Subroutines eliminating all-reduction

- 1: **function** CIRC-MULT-AND-DOT-PROD-AND-CONVG-TEST($\Lambda, \mathbf{y}, \mathbf{v}, \sigma, tol$)
 - 2: Compute d -dimensional FFT of \mathbf{y} and $\rho = \|\mathbf{y}\|$ simultaneously. Let the FFT result be \mathbf{z} .
 - 3: Return with indication of convergence if $\rho < tol$.
 - 4: Compute \mathbf{w} as the elementwise product of λ and \mathbf{z} .
 - 5: Compute d -dimensional inverse FFT of \mathbf{w} , the result being \mathbf{v} . Meanwhile, compute $\sigma = (\mathbf{w}, \mathbf{z})$, which is equivalent to the inner product of \mathbf{y} and \mathbf{v} .
 - 6: **end function**
 - 7: **function** TOEP-MULT-AND-DOT-PROD($\Lambda, \mathbf{y}, \mathbf{v}, \sigma$)
 - 8: Embed \mathbf{y} and compute d -dimensional FFT of the embedding simultaneously (corresponding to the first phase of Figure 1(a)/1(b); embedding means “zero-padding”). Let the result be \mathbf{z} .
 - 9: Compute \mathbf{w} as the elementwise product of λ and \mathbf{z} .
 - 10: Compute d -dimensional inverse FFT of \mathbf{w} and truncate the FFT result simultaneously (corresponding to the second phase of Figure 1(a)/1(b)). The end result is \mathbf{v} . Meanwhile, compute $\sigma = (\mathbf{w}, \mathbf{z})$, which is equivalent to the inner product of \mathbf{y} and \mathbf{v} .
 - 11: **end function**
-

Algorithm 8 Improved implementation of Algorithm 5

- // The following can be separated out for multiple \mathbf{b} 's
- 1: Call TOEP-EMBED-EIGVAL(A, Λ_1) to obtain eigenvalues Λ_1 of the multilevel circulant embedding of A
 - 2: Construct multilevel circulant preconditioner M
 - 3: Compute eigenvalues Λ_2 of M
 - 4: Call TOEP-MULT($\Lambda_1, \mathbf{x}_0, \mathbf{y}_0$) to obtain $\mathbf{y}_0 = A\mathbf{x}_0$
 - // Work for each \mathbf{b}
 - 5: $\gamma = \|\mathbf{b}\|$
 - 6: $tol = \gamma \cdot rtol$
 - 7: $\mathbf{r}_0 = \mathbf{b} - \mathbf{y}_0$
 - 8: Call CIRC-MULT-AND-DOT-PROD-AND-CONVG-TEST($\Lambda_2^{-1}, \mathbf{r}_0, \mathbf{z}_0, \sigma_0, tol$) to obtain $\mathbf{z}_0 = M^{-1}\mathbf{r}_0$ and $\sigma_0 = (\mathbf{r}_0, \mathbf{z}_0)$ simultaneously. Return if converged.
 - 9: $\mathbf{p}_0 = \mathbf{z}_0$
 - 10: **for** $j = 0, 1, \dots, maxit$ **do**
 - 11: Call TOEP-MULT-AND-DOT-PROD($\Lambda_1, \mathbf{p}_j, \mathbf{v}_j, \tau_j$) to obtain $\mathbf{v}_j = A\mathbf{p}_j$ and $\tau_j = (\mathbf{v}_j, \mathbf{p}_j)$ simultaneously
 - 12: $\alpha_j = \sigma_j / \tau_j$
 - 13: $\mathbf{x}_{j+1} = \mathbf{x}_j + \alpha_j \mathbf{p}_j$
 - 14: $\mathbf{r}_{j+1} = \mathbf{r}_j - \alpha_j \mathbf{v}_j$
 - 15: Call CIRC-MULT-AND-DOT-PROD-AND-CONVG-TEST($\Lambda_2^{-1}, \mathbf{r}_{j+1}, \mathbf{z}_{j+1}, \sigma_{j+1}, tol$) to obtain $\mathbf{z}_{j+1} = M^{-1}\mathbf{r}_{j+1}$ and $\sigma_{j+1} = (\mathbf{r}_{j+1}, \mathbf{z}_{j+1})$ simultaneously. Return if converged.
 - 16: $\beta_j = \sigma_{j+1} / \sigma_j$
 - 17: $\mathbf{p}_{j+1} = \mathbf{z}_{j+1} + \beta_j \mathbf{p}_j$
 - 18: **end for**
-

VI. EXPERIMENTAL RESULTS

In this section, we show several experiments to demonstrate the effectiveness of the parallelization strategies discussed in the preceding sections. The programs were implemented in C, compiled with MVAPICH2 and GCC. The in-processor serial FFTs were called from the FFTW library [29]. The experiments were performed on a cluster with 2,560 computing cores and an InfiniBand QDR network. We note that the combinatorial choice of FFT libraries, MPI implementations, C compilers, and machine architectures might have a significant impact on the performance of the overall program, because it has been shown that the scaling of multidimensional FFT is sensitive to both software and hardware. However, obtaining the optimal performance of multidimensional FFT is not the ultimate goal; rather, we show the usefulness of the ideas presented in this paper given a decent FFT environment. The all-to-all were implemented by directly using MPI_Alltoall(v).

The linear systems in the experiments were all 3-level Toeplitz thus the data was 3-dimensional. Therefore, partitioning of the data could be performed along one dimension or two dimensions. We call these 1D and 2D partitionings,

respectively. The performances of the solver were significantly different under the two partitioning schemes, as will be shown.

The Toeplitz matrix was generated from the Matérn kernel that is positive definite for all dimensions [31]–[33]:

$$\phi(r) = \frac{(\sqrt{2\nu}r)^\nu K_\nu(\sqrt{2\nu}r)}{2^{\nu-1}\Gamma(\nu)}, \quad (2)$$

where K_ν is the modified Bessel function of the second kind of order ν , Γ is the Gamma function, and r is elliptical distance

$$r = \sqrt{\sum_{i=1}^d \left(\frac{x_i - y_i}{\ell_i}\right)^2} \quad (3)$$

between two d -dimensional points \mathbf{x} and \mathbf{y} and for a set of scale parameters ℓ_1, \dots, ℓ_d . The Matérn kernel is widely used in spatial statistics, and the solution of the respective linear system is required in numerous statistical analysis tasks, such as regression and maximum likelihood estimation [31]–[34]. In our case, given a 3-dimensional grid in the physical domain $[0, L_1] \times [0, L_2] \times [0, L_3]$, the matrix entry with respect to a pair of grid locations \mathbf{x} and \mathbf{y} is defined by using (2) and (3). Therefore, the resulting matrix is 3-level Toeplitz. In fixed-domain asymptotics, the matrix is increasingly ill-conditioned as the grid becomes denser and denser [35]. Parameters used in the experiments of this paper were $\nu = 0.5$, $\ell_1 = 7$, $\ell_2 = 10$, $\ell_3 = 13$, and $L_1 = L_2 = L_3 = 100$.

Figure 2 shows the strong and weak scalings for one CG iteration. The dashed lines indicate perfect scaling. For each scaling trend in plot (a), the leftmost marker indicates the use of the minimum number of cores such that all the solver data can be fit into the main memory. Because of hardware limitations, the maximum number of cores used in the experiments was 1,024. We observe the following. First, 2D partitioning offers clear advantages over 1D partitioning. For the former, not only is the running time shorter, but also it scales better. (Moreover, 2D partitioning can utilize more processors to further reduce the running time.) Second, plot (a) shows a satisfactory behavior of the strong scaling in the 2D partitioning case. For the first three variations in the core numbers, the drop of running time is close to the perfect scaling decrease, despite the fact that the scaling starts to deteriorate when more cores are used. In plot (b), the overall trend for 2D partitioning is favorable.

The fact that 2D partitioning is superior over 1D partitioning is further demonstrated in Figure 3, which shows the proportions of computation time and communication time. The left plot corresponds to the scenario of a fixed grid size, whereas the right plot corresponds to a fixed grid size per core. One sees that as the number of cores increases, in both partitioning schemes the proportion of communication time increases. However, the increase for 2D partitioning is far slower than that for 1D partitioning.

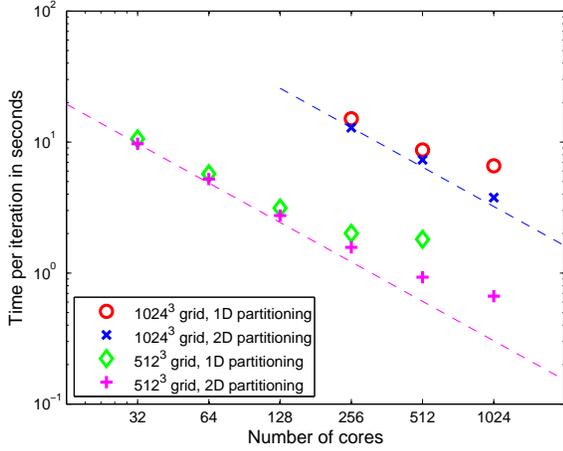
In fact, even though the proportion of communication time in 1D partitioning is smaller at the beginning, it catches up quickly and constitutes a majority part of the overall run time.

To demonstrate the effectiveness of the elimination of all-reductions in the solver, we show in Figure 4 the ratio of the run time per iteration when the solver is implemented by interleaving the inner product calculations with matrix-vector multiplications (cf. Algorithm 8), over the running time when the inner products are calculated by using all-reduction. Clearly, a ratio less than 1 shows the advantage of the proposed algorithm. Only the results of 2D partitioning are shown because we have demonstrated that it is superior over 1D partitioning. Both plots in the figure show that as the number of cores increases, the ratio decreases. For example, in plot (a), the savings by eliminating all-reductions are more than 15% with 1,024 cores. This is expected because the synchronization cost and the variance of time for processors entering the synchronization point is high. We project that the savings will be more significant as the number of cores increases.

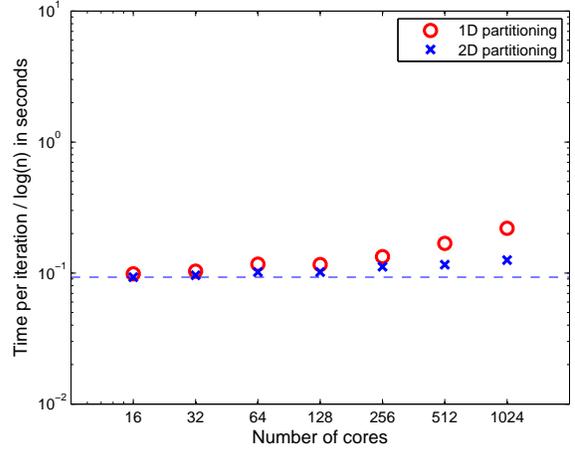
All the above figures were drawn with respect to one iteration. On closing this section, we show the scalings of the overall run time of the solver; see Figure 5. Again, only the results of 2D partitioning are shown. Plot (a) shows strong scaling, which looks similar to Figure 2(a), as expected. Plot (b) shows weak scaling. Since the increase of the grid size causes the respective linear system to be increasingly ill-conditioned (even with the use of a multilevel circulant preconditioner), the variation in the iteration counts becomes an undesired factor in the weak scaling test. To improve the solver, we used a filtering technique proposed in [35] to further reduce the condition number. In fact, when the Matérn order ν is 0.5, a Laplacian filter suffices to make the condition number upper bounded by a constant independent of the grid size. Therefore, with this technique, the number of iterations varies only slightly. In plot (b), the numbers of iterations for the solver to converge to a relative tolerance of 10^{-6} are 15, 12, 13, 15, 13, 14, 15, respectively. Note that when the grid sizes along each dimension are the same (e.g., $256 \times 256 \times 256$ at 16 cores), the number of iterations is slightly larger than that when the grid sizes are not the same along each dimension (e.g., $512 \times 256 \times 256$ at 32 cores). One sees an interesting pattern that for every three problem sizes the numbers of iterations vary roughly in a periodic manner. Then in plot (b) one observes a similar pattern for every three consecutive markers. In general, we conclude that the iterative solver scales well in the weak scaling test.

VII. CONCLUSION

Solving a large-scale linear system with respect to a multilevel Toeplitz matrix is required in various science and engineering applications. An iterative Krylov solver provides a principled framework for the solution of such a linear

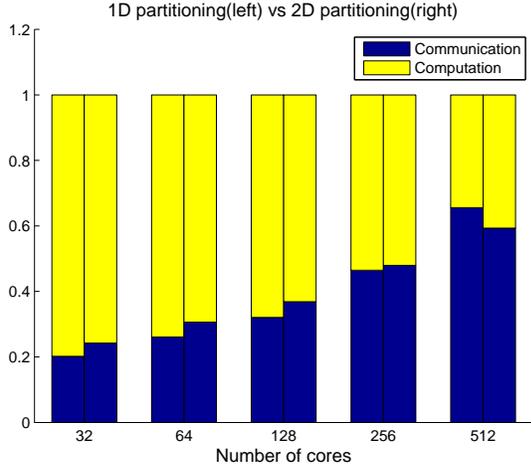


(a) Strong scaling

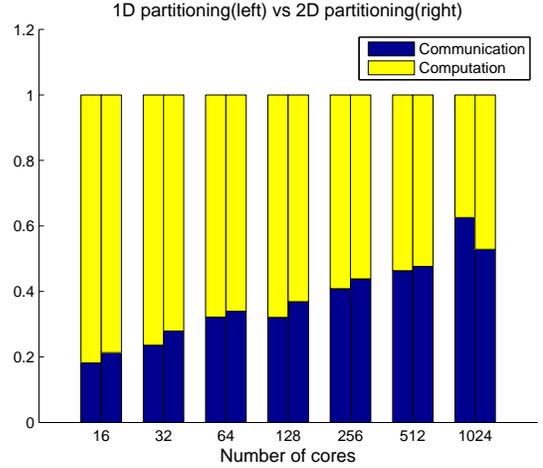


(b) Weak scaling (2^{20} grid points per core)

Figure 2. Scalings. For strong scaling, vertical axis is the average time T per iteration. For weak scaling, vertical axis is $T/\log n$.



(a) Fixed grid size $512 \times 512 \times 512$



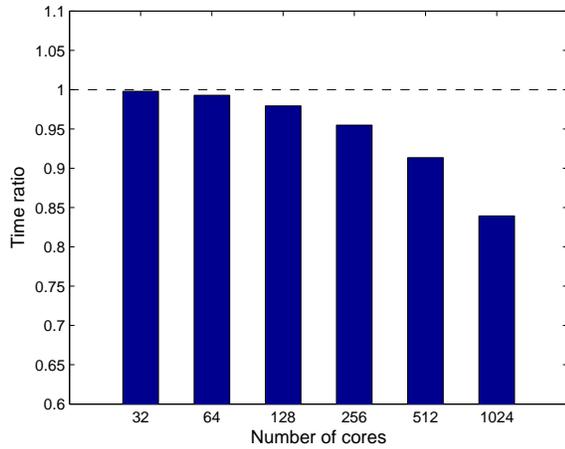
(b) Fixed 2^{20} grid points per core

Figure 3. Computation and communication time split up.

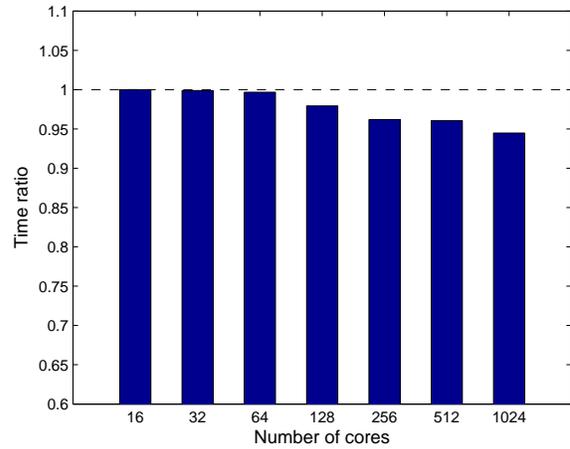
system. We have proposed a parallel implementation of the CG algorithm and shown its effectiveness in an example Toeplitz kernel that is popularly used in spatial statistics. The implementation addresses the reducing of communication cost and latency, by designing the Toeplitz matrix-vector multiplication such that data embedding and truncation are injected into each substep of a multidimensional FFT, and by interleaving the matrix-vector multiplications with inner product calculations to eliminate all-reduction synchronizations. The general idea of the parallel strategies can be used in implementing Krylov solvers other than CG for solving Toeplitz linear systems that are not necessarily symmetric positive definite.

Because the FFT is a major computational component of the solver, and because all currently known implemen-

tations of the multidimensional FFT heavily rely on all-to-all communications in distributed memory computers, the solver will eventually be communication intensive as the number of processors increases. Thus, the scalability of all-to-all communications is critical. What partially alleviates the burden is the flexibility to use a higher dimensional grid of processors so that only a subgroup of them participate in one all-to-all. We have seen in the experiments that for 3D data, the 2D partitioning yields far better performance than does 1D partitioning. We project that for d -dimensional data (for example, spatiotemporal data has three dimensions in space and one dimension in time), a $(d - 1)$ -dimensional partitioning may yield the optimal performance.

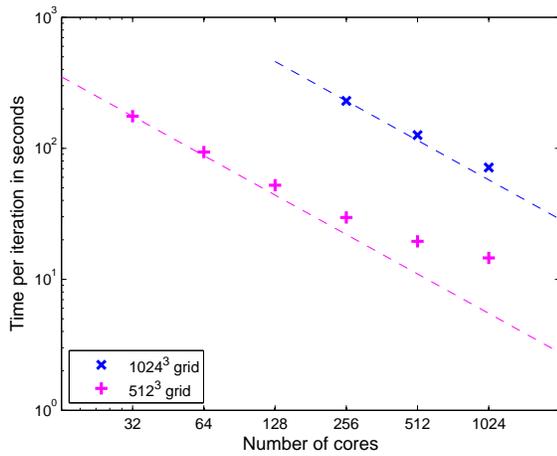


(a) Fixed grid size $512 \times 512 \times 512$

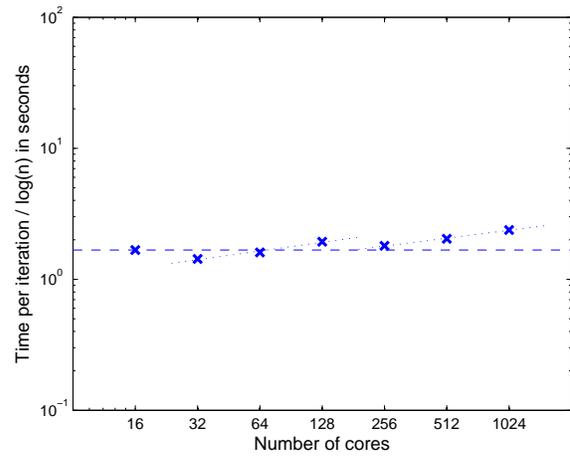


(b) Fixed 2^{20} grid points per core

Figure 4. Time improvement by interleaving inner product calculations with matrix-vector multiplications. Only results of 2D partitioning are shown.



(a) Strong scaling



(b) Weak scaling (2^{20} grid points per core)

Figure 5. Scalings of the running time for solving the linear system. Only results of 2D partitioning are shown.

ACKNOWLEDGMENT

We gratefully acknowledge use of the Fusion cluster in the Laboratory Computing Resource Center at Argonne National Laboratory. This work was supported by the U.S. Department of Energy under Contract DE-AC02-06CH11357.

REFERENCES

[1] N. Levinson, “The Wiener RMS error criterion in filter design and prediction,” *J. Math. Phys.*, vol. 25, pp. 261–278, 1947.

[2] J. Durbin, “The fitting of time-series models,” *Review of the International Statistical Institute*, vol. 28, no. 3, pp. 233–244, 1960.

[3] I. Gohberg, I. Koltracht, A. Averbuch, and B. Shoham, “Timing analysis of a parallel algorithm for Toeplitz matrices on

a MIMD parallel machine,” in *Proceedings of International Conference on Parallel Processing*, 1991.

[4] E. H. Bareiss, “Numerical solution of linear equations with Toeplitz and vector Toeplitz matrices,” *Numer. Math.*, vol. 13, pp. 404–424, 1969.

[5] R. P. Brent, “Parallel algorithms for Toeplitz systems,” in *Numerical Linear Algebra, Digital Signal Processing and Parallel Algorithms*, G. H. Golub and P. V. Dooren, Eds. Springer-Verlag, 1991.

[6] R. R. Bitmead and B. D. Anderson, “Asymptotically fast solution of Toeplitz and related systems of linear equations,” *Linear Algebra Appl.*, vol. 43, pp. 103–116, 1980.

[7] R. P. Brent, F. G. Gustavson, and D. Y. Y. Yun, “Fast solution of Toeplitz systems of equations and computation of Padé

- approximants,” *Journal of Algorithms*, vol. 1, no. 3, pp. 259–295, 1980.
- [8] M. Morf, “Doubling algorithms for Toeplitz and related equations,” in *Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing*, 1980.
- [9] F. de Hoog, “A new algorithm for solving Toeplitz systems of equations,” *Linear Algebra Appl.*, vol. 88–89, pp. 123–138, 1987.
- [10] G. S. Ammar and W. B. Gragg, “Superfast solution of real positive definite Toeplitz systems,” *SIAM J. Matrix Anal. Appl.*, vol. 9, no. 1, pp. 61–76, 1988.
- [11] M. Stewart, “A superfast Toeplitz solver with improved numerical stability,” *SIAM J. Matrix Anal. Appl.*, vol. 25, no. 3, pp. 669–693, 2003.
- [12] S. Chandrasekaran, M. Gu, X. Sun, J. Xia, and J. Zhu, “A superfast algorithm for Toeplitz systems of linear equations,” *SIAM J. Matrix Anal. Appl.*, vol. 29, no. 4, pp. 1247–1266, 2007.
- [13] V. Y. Pan, “Concurrent iterative algorithm for Toeplitz-like linear systems,” *IEEE Trans. on Parallel and Distributed Systems*, vol. 4, no. 5, pp. 592–600, 1993.
- [14] J. Grcar and A. Sameh, “On certain parallel Toeplitz linear system solvers,” *SIAM J. Sci. Stat. Comput.*, vol. 2, no. 2, pp. 238–256, 1981.
- [15] X.-H. Sun, “A scalable parallel algorithm for periodic symmetric Toeplitz tridiagonal systems,” *International Journal of Computer Research*, vol. 10, no. 1, pp. 89–98, 2001.
- [16] P. Alonso, J. M. Badá, and A. M. Vidal, “An efficient parallel algorithm to solve block-Toeplitz systems,” *The Journal of Supercomputing*, vol. 32, no. 3, pp. 251–278, 2005.
- [17] R. H.-F. Chan and X.-Q. Jin, *An Introduction to Iterative Toeplitz Solvers*. SIAM, 2007.
- [18] K.-W. Mak and R. H. Chan, “Parallel implementation of 2-dimensional Toeplitz solver on MasPar with applications to image restoration,” in *Proceedings of High Performance Computing on the Information Superhighway*, 1997.
- [19] T. Chan, “An optimal circulant preconditioner for Toeplitz systems,” *SIAM J. Sci. Stat. Comput.*, vol. 9, no. 4, pp. 766–771, 1988.
- [20] R. H. Chan and P. T. P. Tang, “Fast band-Toeplitz preconditioners for Hermitian Toeplitz systems,” *SIAM J. Sci. Comput.*, vol. 15, no. 1, pp. 164–171, 1994.
- [21] D. Bini and F. Benedetto, “A new preconditioner for the parallel solution of positive definite Toeplitz systems,” in *Proceedings of the second annual ACM Symposium on Parallel Algorithms and Architectures*, 1990.
- [22] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. SIAM, 2003.
- [23] M. Mohiyuddin, M. Hoemmen, J. Demmel, and K. Yelick, “Minimizing communication in sparse matrix solvers,” in *Proceedings of the Conference on High Performance Computing, Networking, Storage and Analysis*, 2009.
- [24] J. Rosendale, “Minimizing inner product data dependencies in conjugate gradient iteration,” in *Proceedings of the International Conference on Parallel Processing*, 1983.
- [25] A. T. Chronopoulos and C. W. Gear, “s-step iterative methods for symmetric linear systems,” *J. Comput. Appl. Math.*, vol. 25, no. 2, pp. 153–168, 1989.
- [26] E. de Sturler and H. van der Vorst, “Reducing the effect of global communication in GMRES(m) and CG on parallel distributed memory computers,” *Appl. Numer. Math.*, vol. 18, no. 4, pp. 441–459, 1995.
- [27] L. T. Yang and R. P. Brent, “The improved BiCGStab method for large and sparse unsymmetric linear systems on parallel distributed memory architectures,” in *Proceedings of International Conference on Algorithms and Architectures for Parallel Processing*, 2002.
- [28] S. S. Capizzano, “Matrix algebra preconditioners for multi-level Toeplitz matrices are not superlinear,” *Linear Algebra Appl.*, vol. 343–344, no. 1, pp. 303–319, 2002.
- [29] M. Frigo and S. G. Johnson, “The design and implementation of FFTW3,” *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005.
- [30] [Online]. Available: [\url{http://code.google.com/p/p3dfft/}](http://code.google.com/p/p3dfft/)
- [31] J.-P. Chilès and P. Delfiner, *Geostatistics: Modeling Spatial Uncertainty*. Wiley-Interscience, 1999.
- [32] M. Stein, *Interpolation of Spatial Data: Some Theory for Kriging*. Springer-Verlag, 1999.
- [33] H. Wendland, *Scattered Data Approximation*. Cambridge University Press, 2005.
- [34] M. Anitescu, J. Chen, and L. Wang, “A matrix-free approach for solving the parametric gaussian process maximum likelihood problem,” *SIAM J. Sci. Comput.*, vol. 34, no. 1, pp. A240–A262, 2012.
- [35] M. L. Stein, J. Chen, and M. Anitescu, “Difference filter preconditioning for large covariance matrices,” *SIAM J. Matrix Anal. Appl.*, vol. 33, no. 1, pp. 52–72, 2012.

The submitted manuscript has been created by the University of Chicago as Operator of Argonne National Laboratory (“Argonne”) under Contract No. DE-AC02-06CH11357 with the U.S. Department of Energy. The U.S. Government retains for itself, and others acting on its behalf, a paid-up, nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.