Preprint ANL/MCS-P5040-1113

A Parallel Linear Solver for Multilevel Toeplitz Systems with Possibly Several Right-Hand Sides

Jie Chen[a,*], Tom L. H. Li[b], Mihai Anitescu[a]

[a] *Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, USA*
[b] *Department of Mathematics and Computer Science, University of Missouri–St. Louis, St. Louis, MO 63121, USA*

Abstract

A Toeplitz matrix has constant diagonals; a multilevel Toeplitz matrix is defined recursively with respect to the levels by replacing the matrix elements with Toeplitz blocks. Multilevel Toeplitz linear systems appear in a wide range of applications in science and engineering. This paper discusses an MPI implementation for solving such a linear system by using the conjugate gradient algorithm. The implementation techniques can be generalized to other iterative Krylov methods besides conjugate gradient. These techniques include the use of an arbitrary dimensional process grid for handling the multilevel Toeplitz structure, a communication-hiding approach for performing matrix-vector multiplications, the incorporation of multilevel circulant preconditioning for accelerating convergence, an efficient orthogonalization manager for detecting linear dependence in block iterations, and an algorithmic rearrangement to eliminate all-reduce synchronizations. The combined use of these techniques leads to a scalable solver for large multilevel Toeplitz systems, possibly with several right-hand sides. We show experimental results on matrices of size up to the order of one billion with nearly perfect scaling by using up to 1,024 MPI processes. We also demonstrate an application of the solver in parameter estimation for analyzing large-scale climate data.

Keywords: Krylov method, Multiple right-hand sides, Multilevel Toeplitz, FFT, Communication hiding, Allreduce

1. Introduction

The (multilevel) Toeplitz structure is a common matrix pattern arising in many application domains, such as digital signal processing, image processing, optimal control, and stationary time series. A Toeplitz matrix has constant diagonals. It can be represented solely by the first row and the first column, whose entries are called *generating elements*. A multilevel Toeplitz structure is defined recursively on the levels. In the simplest two-level case, each generating element is replaced by a Toeplitz block; this structure is also termed *BTTB*: *block Toeplitz with Toeplitz blocks* [1]. Levels higher than two are common, such as in a three-dimensional spatial or four-dimensional spatiotemporal random field.

The special Toeplitz structure enables the design of fast linear solvers that have a cost asymptotically lower than that of general solver based on triangular factorizations, $O(n^3)$, where n is the number of rows (equivalently, columns) of the matrix. Such methods have been extensively studied; a few representative ones are Levinson-Durbin $O(n^2)$ [2, 3, 4], Bareiss $O(n^2)$ [5, 6], and "superfast" solvers $O(n \log^\alpha n)$ [7, 8, 9, 10]. Compared with the abundance of algorithms for 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. In this paper, we consider the approach of using an iterative Krylov method [1].

Employing a Krylov method offers several advantages, the most important being that the method is independent of the number of Toeplitz levels. Rather, the multilevel structure is exploited in the matrix-vector multiplication and

*Corresponding author.
Email addresses: jiechen@mcs.anl.gov (Jie Chen), ll9n8@mail.umsl.edu (Tom L. H. Li), anitescu@mcs.anl.gov (Mihai Anitescu)

preconditioning. The time cost of a Krylov method on a multilevel Toeplitz matrix is $O(kn \log n)$ and the storage cost is $O(n)$, where k is the number of Krylov iterations. A class of preconditioners based on a corresponding (multilevel) circulant structure has been proposed to ensure that k grows much more slowly than does n [1]. These properties make it possible to solve systems of a very large scale. In fact, for such a scale (e.g., $n = 10^7$ and beyond), a Krylov method appears to be the only viable option, because direct solvers, even those implemented with fine-grained parallelism (e.g., ScaLAPACK [11]), require quadratic storage and easily hit the memory limit on nowadays supercomputers.

The multilevel Toeplitz structure provides a rich design space and requires additional implementation efforts than building the generic Krylov iteration framework, the latter having been implemented in many sophisticated numerical libraries such as PETSc [12] and Trilinos [13]. This paper addresses the Toeplitz-specific issues that are beyond the implementation of a generic Krylov solver.

1. How can the matrix-vector multiplication be carried out efficiently? In principle, the multiplication with a multilevel Toeplitz matrix is performed through multidimensional fast Fourier transforms (FFTs). The implementation is complicated by data partitioning, expansion, and truncation, all of which have significant implications for data movement.
2. How is the data (matrix and vectors) partitioned?
3. How is the preconditioner constructed and applied?
4. For multiple right-hand sides, a block Krylov iteration is often a viable choice, because it converges faster than do single-vector iterations. Block iterations offer flexibility in handling linearly dependent vectors, where usually a rank-revealing factorization is performed and columns are split in the event of linear dependence. The factorization is often expensive because of communications. However, unlike its use in long-term Krylov iterations (such as GMRES), here exact orthogonality is not important. Thus, we implement an orthogonalization procedure that is sufficiently stable for block iterations and also is computationally as efficient as computing one block inner product.
5. We discuss a mathematically equivalent rearrangement of the inner-product and norm calculations for eliminating the all-reduce synchronizations. Removing the synchronizations improves concurrency at a large process count. We demonstrate that the algorithmic rearrangement is numerically stable.

For clarity of presentation, we focus on the real supersymmetric case (meaning that all the Toeplitz levels are symmetric) and the conjugate gradient (CG) algorithm; but the design principles are not restrictive. With slight additional coding effort, the implementation can be generalized to complex Hermitian matrices and nonsymmetric/non-Hermitian matrices. Several techniques, such as matrix-vector multiplication and preconditioning, are independent of the specific choice of the Krylov method and thus are applicable.

An application of the proposed solver is Gaussian process [14, 15, 16, 17] maximum likelihood estimation (MLE), which is a general framework for parameter estimation in statistical data analysis. In a Gaussian process, the covariance matrix is multilevel Toeplitz if the process is observed on a regular grid and if it is stationary. The MLE is an optimization procedure for estimating the parameters in the covariance matrix. Following the MLE method studied in [18, 19], we apply the solver and demonstrate its capability in performing MLE with large-scale data. The example data in this paper comes from climate science. Based on the model computed from MLE, we are able to perform interpolations for existing data and forecasting for future time steps. The solver is a major component of the ScalaGAUSS project.[1]

2. Preliminaries

This section provides background on the mathematics involved in our implementation. Because of the complication in notation, we will use upper case letters (such as A) to denote a matrix, boldface lower case letters (such as $\boldsymbol{y}$) to denote a vector; and sans serif fonts (such as c) to denote a multidimensional data array, which can be one-dimensional (vector), two-dimensional (matrix), and higher dimensional. The first two notations admit mathematical meanings whereas the last one is more suitable for presenting algorithms.

[1] `http://press3.mcs.anl.gov/scala-gauss/`

2.1. Circulant and Toeplitz Matrices

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

$$C = \begin{bmatrix} c_0 & c_{n-1} & c_{n-2} & \cdots & \cdots & c_1 \\ c_1 & c_0 & c_{n-1} & \ddots & & \vdots \\ c_2 & c_1 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & c_{n-1} & c_{n-2} \\ \vdots & & \ddots & c_1 & c_0 & c_{n-1} \\ c_{n-1} & \cdots & \cdots & c_2 & c_1 & c_0 \end{bmatrix} \qquad T = \begin{bmatrix} t_0 & t_{-1} & t_{-2} & \cdots & \cdots & t_{-n+1} \\ t_1 & t_0 & t_{-1} & \ddots & & \vdots \\ t_2 & t_1 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & t_{-1} & t_{-2} \\ \vdots & & \ddots & t_1 & t_0 & t_{-1} \\ t_{n-1} & \cdots & \cdots & t_2 & t_1 & t_0 \end{bmatrix}. \tag{1}$$

We use a subscript n to emphasize the order when necessary. In the real symmetric case, both matrices can be represented solely by their first column. We use one-dimensional arrays $\mathsf{c} = [c_0, \dots, c_{n-1}]$ and $\mathsf{t} = [t_0, \dots, t_{n-1}]$ as data representations of C and T, respectively. (At the end of Section 3 we briefly discuss how to incorporate the entries $t_{-1}, \dots, t_{-n+1}$ in the unsymmetric case.) A Toeplitz matrix T_n can be embedded into a circulant matrix C_{2n} (of twice the size) in the following form:

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

All the elements (except for the diagonal ones) in the two subblocks denoted by $*$ are well defined. In the data representation, we have

$$\mathsf{c}_i = \begin{cases} \mathsf{t}_i & i = 0, \dots, n-1, \\ \text{arbitrary} & i = n, \\ \mathsf{t}_{i-2n} = \mathsf{t}_{2n-i} & i = n+1, \dots, 2n-1. \end{cases} \tag{3}$$

Informally, the embedding means that the first half of c is t, whereas the latter half is a "flipping" of t except for the first entry. By convention, we set this arbitrary entry to be zero.

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

$$UCU^H = \Lambda,$$

where $\Lambda = \mathrm{diag}(\lambda_1, \dots, \lambda_n)$ contains the eigenvalues of C and U is the DFT matrix with entries $U_{jk} = \exp(\mathbf{i}2\pi jk/n)/\sqrt{n}$. Multiplying the vector of all ones on both sides of this equation, one sees that the eigenvalues can be obtained all at once by applying FFT on the first column of C, followed by a scaling $\sqrt{n}$. Then, the matrix-vector product $\boldsymbol{v} = C\boldsymbol{y} = U^H \Lambda U \boldsymbol{y}$ can be calculated by using the precomputed eigenvalues and applying FFT twice (one forward and one backward). The detailed steps are shown in Algorithm 1.

Algorithm 1 Circulant matrix times vector $\boldsymbol{v} = C\boldsymbol{y}$ (data representation v, c, y)

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

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

$$\begin{bmatrix} T_n\boldsymbol{y} \\ * \end{bmatrix} = \begin{bmatrix} T_n & * \\ * & T_n \end{bmatrix} \begin{bmatrix} \boldsymbol{y} \\ \mathbf{0} \end{bmatrix} = C_{2n} \begin{bmatrix} \boldsymbol{y} \\ \mathbf{0} \end{bmatrix},$$

where the $*$ beneath $T_n\boldsymbol{y}$ denotes some unessential subvector. The steps for computing $\boldsymbol{v} = T\boldsymbol{y}$ are shown in Algorithm 2, utilizing the circulant multiplication in Algorithm 1.

To generalize the definition of circulant and Toeplitz to multilevel cases, we use recursion. With the form (1), if each c_i itself is a $(d-1)$-level circulant matrix, then C is a d-level circulant matrix. The data representation of C is a

Algorithm 2 Toeplitz matrix times vector $\boldsymbol{v} = T\boldsymbol{y}$ (data representation v, t, y)

1: Construct the data representation c as the embedding of t according to (3).
2: Embed y into a length-$2n$ vector y' (all remaining entries being zero)
3: Multiply the constructed circulant matrix with y' to obtain the product v' using Algorithm 1
4: Truncate v' (keeping the first n entries) to obtain v

d-dimensional array c of size $n_1 \times \cdots \times n_d$. We let $n_1 \times \cdots \times n_d = n$. For example, when $d = 2$, we have a two-level circulant matrix C, where each $c_i \in \mathbb{R}^{n_1 \times n_1}$ is circulant and there are n_2 such c_i's. The data representation c is an $n_1 \times n_2$ array, where the ith column of c is the first column of c_i. Similar definitions apply to a d-level Toeplitz matrix T with a data representation t. When the context is clear, we will drop the qualifier "d-level" or "multilevel" since the basis case "1-level" is simply a special case.

Now the notational benefit of using data representations is clear: the multiplication algorithms need little modifications. Specifically, for multilevel circulant matrices, in Algorithm 1 "FFT" means "multidimensional FFT." For multilevel Toeplitz matrices, the embedding and the truncation in Algorithm 2 are performed along each dimension of the data array. It suffices to show a pictorial example for the two-level case. The vector $\boldsymbol{y}$ is represented as an $n_1 \times n_2$ array y. Then the embedding is

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

which is of size $2n_1 \times 2n_2$. Similarly, the two-level Toeplitz matrix T has a data form t. Then, its circulant embedding C has a data form

$$\mathsf{c} = \begin{bmatrix} \mathsf{t} & * \\ * & * \end{bmatrix}, \tag{4}$$

where the blocks denoted by $*$ contain the "flipping" of t. Formally, for $i = 0, \ldots, n_1 - 1$,

$$\mathsf{c}_{ij} = \begin{cases} \mathsf{t}_{ij} & j = 0, \ldots, n_2 - 1, \\ \text{arbitrary} & j = n_2, \\ \mathsf{t}_{i,\, 2n_2 - j} & j = n_2 + 1, \ldots, 2n_2 - 1. \end{cases}$$

Then, for $j = 0, \ldots, 2n_2 - 1$,

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

2.2. Conjugate Gradient Algorithm and the Block Version

Given a symmetric matrix $A \in \mathbb{R}^{n \times n}$ and a vector $\boldsymbol{b} \in \mathbb{R}^{n \times 1}$, the standard CG algorithm [20] for solving

$$A\boldsymbol{x} = \boldsymbol{b},$$

is presented on the left panel of Figure 1. It accepts an initial guess $\boldsymbol{x}_0$ and refines the approximate solution $\boldsymbol{x}_{j+1}$ until the residual $\boldsymbol{r}_{j+1}$ falls below a relative tolerance *rtol* or j exceeds the maximum number of iterations *maxit*. In the algorithm, M is the preconditioner, a symmetric positive definite matrix that approximates A.

The algorithm can be generalized to use block iterations for solving a system with s right-hand sides simultaneously [21, 22]:

$$AX = B,$$

where $B \in \mathbb{R}^{n \times s}$. We call such matrices B, where the number of columns s is far smaller than the number of rows n, *block vectors*, and we write $\boldsymbol{b}^{(i)}$ for the ith column of the block vector. The block iteration, as presented in the right panel of Figure 1, resembles the single-vector iteration. In particular, the vectors in CG become block vectors of width s, and the scalar coefficients (σ_j, τ_j, α_j, and β_j) become $s \times s$ coefficient matrices. The orthogonalization

1: $\gamma = \|\boldsymbol{b}\|$
2: $\boldsymbol{r}_0 = \boldsymbol{b} - A\boldsymbol{x}_0$
3: $\rho_0 = \|\boldsymbol{r}_0\|$
4: **if** $\rho_0/\gamma < rtol$ **then return**
5: $z_0 = M^{-1}\boldsymbol{r}_0$
6: $\sigma_0 = z_0^T\boldsymbol{r}_0$
7: $\boldsymbol{p}_0 = z_0$
8: // empty
9: **for** $j = 0, 1, \ldots, maxit$ **do**
10: // empty
11: $\boldsymbol{v}_j = A\boldsymbol{p}_j$
12: $\tau_j = \boldsymbol{p}_j^T\boldsymbol{v}_j$
13: $\alpha_j = \sigma_j/\tau_j$
14: $\boldsymbol{x}_{j+1} = \boldsymbol{x}_j + \alpha_j\boldsymbol{p}_j$
15: $\boldsymbol{r}_{j+1} = \boldsymbol{r}_j - \alpha_j\boldsymbol{v}_j$
16: $\rho_{j+1} = \|\boldsymbol{r}_{j+1}\|$
17: **if** $\rho_{j+1}/\gamma < rtol$ **then return**
18: $z_{j+1} = M^{-1}\boldsymbol{r}_{j+1}$
19: $\sigma_{j+1} = z_{j+1}^T\boldsymbol{r}_{j+1}$
20: $\beta_j = \sigma_{j+1}/\sigma_j$
21: $\boldsymbol{p}_{j+1} = z_{j+1} + \beta_j\boldsymbol{p}_j$
22: **end for**

1: $\gamma^{(i)} = \|\boldsymbol{b}^{(i)}\|$ for all i
2: $R_0 = B - AX_0$
3: $\rho_0^{(i)} = \|\boldsymbol{r}_0^{(i)}\|$ for all i
4: **if** $\rho_0^{(i)}/\gamma^{(i)} < rtol$ for all i **then return**
5: $Z_0 = M^{-1}R_0$
6: $\sigma_0 = Z_0^T R_0$
7: $P_0 = Z_0$
8: Initialize `IDX` = {`idx`} with `idx` = $\{1, 2, \ldots, s\}$
9: **for** $j = 0, 1, \ldots, maxit$ **do**
10: Update `IDX` ← Orthogonalize(P_j, `IDX`)
11: $V_j = AP_j$ ▷ for each `idx` ∈ `IDX`
12: $\tau_j = P_j^T V_j$ ▷ for each `idx` ∈ `IDX`
13: $\alpha_j = \tau_j^{-1}\sigma_j$ ▷ for each `idx` ∈ `IDX`
14: $X_{j+1} = X_j + P_j\alpha_j$ ▷ for each `idx` ∈ `IDX`
15: $R_{j+1} = R_j - V_j\alpha_j$ ▷ for each `idx` ∈ `IDX`
16: $\rho_{j+1}^{(i)} = \|\boldsymbol{r}_{j+1}^{(i)}\|$ for all i
17: **if** $\rho_{j+1}^{(i)}/\gamma^{(i)} < rtol$ for all i **then return**
18: $Z_{j+1} = M^{-1}R_{j+1}$ ▷ for each `idx` ∈ `IDX`
19: $\sigma_{j+1} = Z_{j+1}^T R_{j+1}$ ▷ for each `idx` ∈ `IDX`
20: $\beta_j = \sigma_j^{-1}\sigma_{j+1}$ ▷ for each `idx` ∈ `IDX`
21: $P_{j+1} = Z_{j+1} + P_j\beta_j$ ▷ for each `idx` ∈ `IDX`
22: **end for**

Figure 1: CG (left) and block CG (right). To avoid cluttering, the block CG algorithm omits the indicator `idx` attached to every iterate below line 10. It should be interpreted as the column indicator of the block vectors (upper case letters) and principal submatrix indicator of the coefficient matrices (Greek letters). For example, $V_j = AP_j$ means $V_j(:, \mathtt{idx}) = AP_j(:, \mathtt{idx})$, and $\tau_j = P_j^T V_j$ means $\tau_j(\mathtt{idx}, \mathtt{idx}) = P_j(:, \mathtt{idx})^T V_j(:, \mathtt{idx})$.

step Orthogonalize(P_j, `IDX`) takes a block vector P_j and a set of index sets `IDX` = {`idx`} as input, performs a rank-revealing orthogonalization on each $P_j(:, \mathtt{idx})$ and splits the indices in `idx` if linearly dependent columns are found. The orthogonalization step ensures that each output `idx` gives a set of columns of P_j that are linearly independent. Then, all the calculations starting from line 10 are in essence performed on the subcolumns of the block vectors and on the principal submatrices of the $s \times s$ coefficient matrices. When $s = 1$, block CG is equivalent to CG.

In addition to being able to solve a linear system with multiple right-hand sides, an advantage of block iteration is its fast convergence. Denoting by $\boldsymbol{x}_*^{(i)} = A^{-1}\boldsymbol{b}^{(i)}$ the solution, we have for the convergence rate of (block) CG [21]

$$\|\boldsymbol{x}_j^{(i)} - \boldsymbol{x}_*^{(i)}\|_A \leq \left(\frac{\sqrt{\kappa_s} - 1}{\sqrt{\kappa_s} + 1}\right)^j D^{(i)},$$

where $\|\cdot\|_A$ denotes the A-norm of a vector, $\kappa_s = \lambda_n(M^{-1}A)/\lambda_s(M^{-1}A)$, λ are the eigenvalues of $M^{-1}A$ sorted increasingly, and $D^{(i)}$ is a constant independent of the iteration j. One sees that as s increases, the solution error $\boldsymbol{x}_j^{(i)} - \boldsymbol{x}_*^{(i)}$ decreases faster because κ_s is smaller.

2.3. Circulant Preconditioning

For our case, A is a supersymmetric multilevel Toeplitz matrix. When A is one-level, a class of circulant matrices is proposed as the preconditioner M [1]. The general idea for defining these preconditioners is to impose the circulant structure and to minimize the difference between M and A in some norm. We refer the reader to [1] for a comprehensive exposition of the preconditioners. Here, we will use T. Chan's preconditioner [23], which minimizes the Frobenius norm, that is,

$$M = \underset{C \text{ circulant}}{\operatorname{argmin}} \|A - C\|_F.$$

This preconditioner yields a clustered spectrum of $M^{-1}A$, hence assisting the CG iteration to converge superlinearly. It is straightforward to generalize M to the multilevel case. Although the superlinear convergence property is lost in this case [24], the corresponding multilevel circulant preconditioner yields sufficiently good performance in practice [1]. We point out that, however, a multilevel circulant preconditioner that provably achieves superlinear convergence is not known.

Using the notation in Section 2.1, let the $n_1 \times \cdots \times n_d$ data arrays a and m represent the matrices A and M, respectively. In the one-level case (where $n = n_1$), T. Chan's preconditioner is defined as

$$\mathsf{m}_j = ((n-j)\,\mathsf{a}_j + j\,\mathsf{a}_{n-j})/n, \qquad j = 0, \ldots, n-1.$$

Informally, m is a weighted averaging of a and its flipping, and the weights are defined with respect to the locations of the entries. Then, in the general d-level setting, the averaging is done along each dimension. To avoid notational tediousness, we show in Algorithm 3 the averaging procedure for the three-level case. Generalization to any d is straightforward.

Algorithm 3 Constructing preconditioner M (for $d = 3$)

1: Assign $\mathsf{m} \leftarrow \mathsf{a}$
2: Update $\mathsf{m}_{:,\,:,\,j} \leftarrow ((n_1 - j)\,\mathsf{m}_{:,\,:,\,j} + j\,\mathsf{m}_{:,\,:,\,n_1-j})/n_1$ for $j = 0, \ldots, n_1 - 1$
3: Update $\mathsf{m}_{:,\,j,\,:} \leftarrow ((n_2 - j)\,\mathsf{m}_{:,\,j,\,:} + j\,\mathsf{m}_{:,\,n_2-j,\,:})/n_2$ for $j = 0, \ldots, n_2 - 1$
4: Update $\mathsf{m}_{j,\,:,\,:} \leftarrow ((n_3 - j)\,\mathsf{m}_{j,\,:,\,:} + j\,\mathsf{m}_{n_3-j,\,:,\,:})/n_3$ for $j = 0, \ldots, n_3 - 1$

3. Data Partitioning

In the general d-level case, since the Toeplitz matrix T, the circulant preconditioner M, and the vectors $\boldsymbol{y}$ are all represented as $n_1 \times n_2 \times \cdots \times n_d$ data arrays, a natural approach of parallel processing is to use a logical d'-dimensional process grid of size $p_1 \times p_2 \times \cdots \times p_{d'}$ to partition these data. Let $p_1 \times p_2 \times \cdots \times p_{d'} = p$. Without loss of generality, we require that $p_i > 1$ for $i = 1, \ldots, d'$, and we let the ordered sequence of integers $S = \{1, \ldots, d\}$ be divided into $d' + 1$ consecutive subsequences $S_1, \ldots, S_{d'+1}$, where $S_i \neq \emptyset$ for $i = 1, \ldots, d' + 1$. Then, the processes along the ith dimension of the process grid are responsible for partitioning the S_i dimensions of the data, for $i = 1, \ldots, d'$. The $S_{d'+1}$ dimensions are unpartitioned. For example, when $d = 8$, we can have $d' = 3$ and $S_1 = \{1, 2\}$, $S_2 = \{3, 4, 5\}$, $S_3 = \{6\}$, and $S_4 = \{7, 8\}$. Figure 2 illustrates the data and the process grid with actual sizes.

$$\begin{array}{ll} \text{data size} & \underbrace{76 \times 32}_{} \times \underbrace{12 \times 58 \times 10}_{} \times \underbrace{409}_{} \times \underbrace{48 \times 17}_{} \\ \text{process grid} & \quad 12 \quad\;\; \times \qquad 16 \qquad \times \;\; 13 \;\; \text{unpartitioned} \end{array}$$

Figure 2: Example partitioning of an 8-dimensional data using a 3-dimensional process grid.

We note two popular data partitioning methods for multidimensional FFTs, both of which are special cases of our scheme. The common idea for multidimensional FFTs is to partition some dimensions of the data while leaving the others unpartitioned, so that in-process FFTs can be performed on the latter dimensions first. (More details about FFT are covered in the next section.) The method adopted by FFTW [25] uses a one-dimensional process grid, that is, $p = p_1$, and these processes partition only the first dimension of the data. The other method, adopted by P3DFFT [26] and others [27, 28], handles particularly three-dimensional data (i.e., $d = 3$). It uses a two-dimensional process grid of size $p_1 \times p_2$ to partition the first two dimensions of the data separately, leaving the third dimension unpartitioned. The flexibility of our partitioning scheme lies in not only allowing an arbitrary number of dimensions for the process grid (as long as $d' < d$) but also allowing several dimensions of the data to be collectively partitioned.

One advantage of using a high-dimensional process grid is that it increases the allowable maximum number of processes for solving a problem with a fixed size. This benefit comes with certain sacrifices, depending on the specific computations with the data. For multidimensional FFTs, if one fixes the number of processes but varies the dimensions of the process grid, the higher the dimension, the more overhead that is incurred in performing all-to-all communications. In general, it is difficult to theoretically analyze the best configuration of the process grid under various machine settings, especially because of the variance of interconnection network. Hence, performance tuning (runtime parameter optimization) is a better tool for figuring out the best or near-best configuration under the flexible partitioning scheme we provide.

Note two related issues. The first one is that when performing matrix-vector multiplications with the Toeplitz matrix T, the FFTs are not performed on the data $\mathtt{t}$ that has a size $n_1 \times n_2 \times \cdots \times n_d$ but, rather, on the circulant embedding $\mathtt{c}$ whose size doubles along each dimension: $(2n_1) \times (2n_2) \times \cdots \times (2n_d)$; see (4). Constructing such a $\mathtt{c}$ from $\mathtt{t}$ requires data redistribution, where a naive implementation incurs significant data movement among processes. In the next section, we present an approach that hides the data movement in the FFT calculations, thus eliminating the communication.

The second issue is how to generalize this partitioning scheme for unsymmetric multilevel Toeplitz matrices and preconditioners. If T is unsymmetric, it can no longer be represented by using only the first column; the first row must also appear in the data representation $\mathtt{t}$. Hence, for the one-level case, $\mathtt{t} = [t_0, \ldots, t_{n-1}, *, t_{-n+1}, \ldots, t_{-1}]$, where $*$ is arbitrary. The d-level case is straightforward. Then, one can easily verify, according to (2), that the circulant embedding C has a data representation $\mathtt{c}$ exactly the same as $\mathtt{t}$. In other words, for the nonsymmetric case, the partitioning of T in effect acts on a data of size $(2n_1)\times(2n_2)\times\cdots\times(2n_d)$, whereas the partitioning of the preconditioner and the vectors still acts on data of size $n_1 \times n_2 \times \cdots \times n_d$. The partitionings between these two sizes are switched frequently throughout the CG iterations, using the communication-hiding technique elaborated in the next section.

4. Matrix-Vector Multiplication

The Toeplitz matrix-vector multiplication (A-multiply) is the most computationally intensive component of the CG solver, which requires data embedding/truncation and circulant matrix-vector multiplications, the latter being done through multidimensional FFTs. The preconditioniong step (M^{-1}-multiply) is a straightforward extension because M is circulant, except that the data size is not as large as that in A-multiply.

Both A-multiply and M^{-1}-multiply have a precomputation step where the eigenvalues of the embedding of A, and the eigenvalues of M are computed. Similar to the matrix-vector multiplications, these eigenvalues are computed by using FFTs.

Hence, in this section, we focus on the computation of the product $\boldsymbol{v} = A\boldsymbol{y} = T\boldsymbol{y}$, assuming that the eigenvalues of the embedding of T are known. Recall steps 2–4 in Algorithm 2 for computing $T\boldsymbol{y}$. We propose two communication-hiding techniques to streamline these steps so that the communication overhead is greatly reduced.

4.1. Combining Embedding with FFT

If we naively perform embedding first and then FFT, the embedding incurs communications. The key observation is that FFT requires data transposes; hence the communications for embedding are redundant in the presence of transposes. Let a fiber denote a segment of a multidimensional data where some indices are fixed and the remaining indices take the whole range (thus, we often say *a fiber along some dimension*(*s*), meaning that the dimensions are with respect to the varying indices). For the dimensions along which the fibers are not partitioned, the embeddings along these dimensions do not require interprocess communications. Then, the embeddings along other dimensions are performed only when the fiber along the corresponding dimension becomes unpartitioned during the process of transposes in FFT.

The complete steps are best explained by walking through Figure 3, which uses the example data and process grid in Figure 2. Here, "data" means the vector $\mathtt{y}$, not $\mathtt{t}$. Initially, the eight-dimensional data (whose size is indicated by the numbers above the long rectangular frame) is partitioned by a three-dimensional process grid (whose size is indicated by the numbers inside the frame), wherein the seventh and the eighth dimensions are unpartitioned. The first embeddings are performed along the unpartitioned dimensions, followed by FFTs, resulting in the doubling of sizes along these dimensions. Then, a data transpose is performed between the sixth dimension and the seventh and eighth

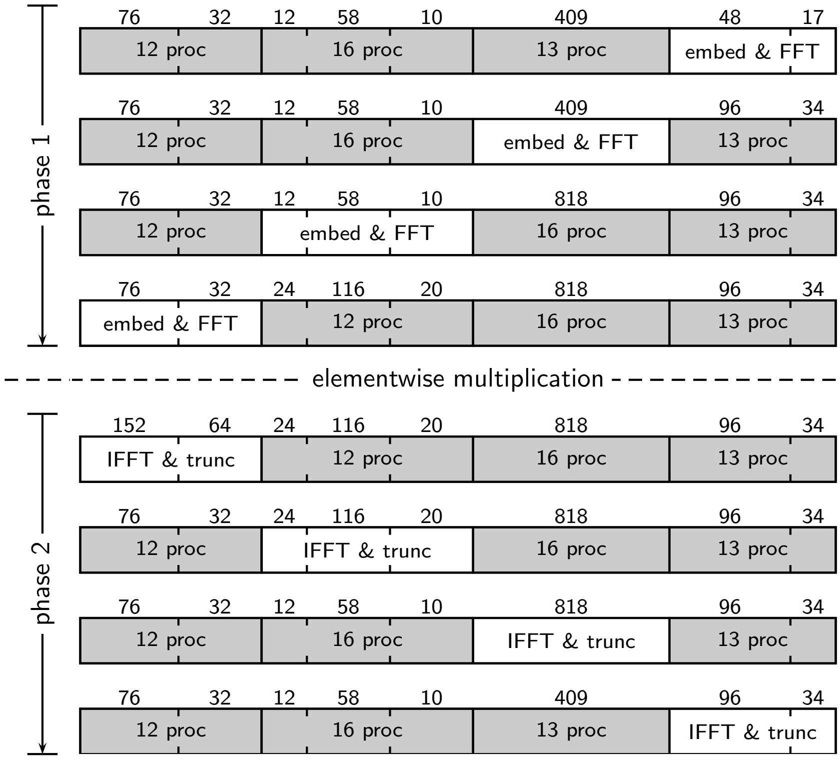


Figure 3: Example in Figure 2 continued: Multiplying an 8-level Toeplitz matrix with vector.

dimensions, so that now the sixth dimension becomes unpartitioned. An embedding followed by FFT is performed along this dimension. This procedure continues until the unpartitioned dimensions are iterated to the front. After the embeddings and FFTs along these dimensions, phase 1 is concluded. Every dimension of the data has doubled the size, and the full multidimensional FFT is completed on the doubled data.

After phase 1, an elementwise multiplication is performed, which in effect multiplies every eigenvalue of the embedding of T with every element of the doubled data. Then comes phase 2, a reverse procedure of phase 1, where the unpartitioned dimensions are iterated from the front to the back, along which inverse FFTs and truncations of the data are performed. The result is data having the same size as the original before entering phase 1. This data is v, the product of $T\boldsymbol{y}$. One sees that, in the entire procedure, the communications of embedding and truncation never occur.

What are the computation and communication costs of this procedure? Let us break the costs in three categories: (i) cost for in-process FFTs; (ii) cost for data transpose; and (iii) cost for other linear-time operations, such as embedding and truncation of the data. We reuse the notation S_i introduced in Section 3; it denotes a consecutive subsequence of $\{1, \ldots, d\}$ that indicates the data-grid dimensions that are not partitioned during the course of transposes. Define

$$q_i := \sum_{k>i} |S_k| \quad \text{for} \quad i = 1, \ldots, d' + 1.$$

We consider only phase 1 because the cost for phase 2 is the same. In the step where the S_i dimensions are unpartitioned, the number of fibers is $\prod_{j \notin S_i} n_j \cdot 2^{q_i}$, because there are q_i dimensions whose sizes have been doubled. The size of one fiber is $\prod_{j \in S_i}(2n_j)$, on which an in-process FFT is carried out. Then, a data transpose occurs among all p/p_{i-1} independent communication subgroups, if $i \neq 1$. Each subgroup comprises p_{i-1} processes, each holding a data volume of $2^{q_{i-1}} n/p$. Only $1/p_{i-1}$ of the volume stays; the rest is redistributed to the other $p_{i-1} - 1$ processes in the same subgroup. Note that in a multicore environment, the redistributed data is not necessarily transferred through the communication network. The rest of the operations other than in-process FFTs and communications is linear with the current data size, which is $n \cdot 2^{q_i}$. Table 1 summarizes these costs.

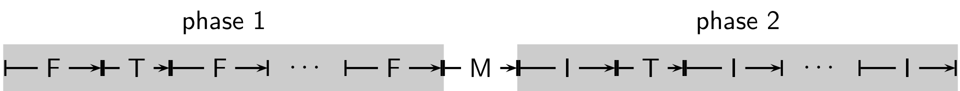
Table 1: Categorical costs of matrix-vector multiplication, phase 1. Assume currently the S_i dimensions of the data grid are unpartitioned.

Number of in-process FFTs	$\prod_{j \notin S_i} n_j \cdot 2^{q_i}$
Data size of each FFT	$\prod_{j \in S_i} (2n_j)$
Number of communication subgroups	p/p_{i-1}
Data size of transpose for each subgroup	$n \cdot 2^{q_{i-1}} \cdot p_{i-1}/p$
Asymptotic cost for all other operations	$O(n \cdot 2^{q_i})$

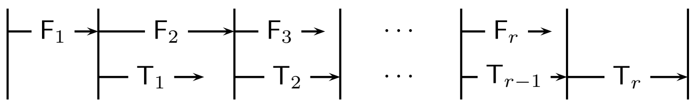
One might be able to carry out a performance modeling based on the numbers listed in Table 1. In this paper, however, we do not pursue such a modeling because of several reasons. First, the FFT library is typically fine tuned and cache oblivious, which leads to a performance curve that is not well aligned with the theoretical complexity. Second, the communication cost for performing transposes varies significantly with the MPI implementation and the machine architecture. Third, to the contrary of the intuition that FFTs are costly, we observe in our implementation that the cost of linear-time operations such as embedding, truncation, and data copying between buffers generally dominates, despite the fact that several coding efforts have been spent on optimizing these operations. Several factors possibly contribute to this phenomenon, including the expensive index calculations for rearranging the layout of high-dimensional data, and frequent cache misses when copying data in strides. Fourth, the pipelining effort proposed in the following subsection increases the difficulty of performance modeling.

4.2. Pipelining FFT with Transpose

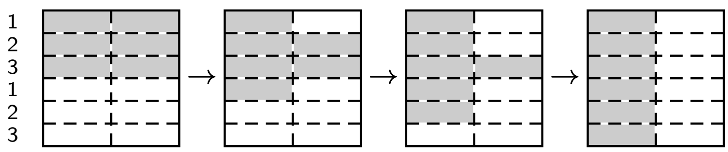
Figure 4(a) shows another view of the flow of Figure 3. In phase 1, embedding/FFT (denoted as "F") and transpose (denoted as "T") are performed in an alternating fashion. The data dependency requires that a second "F" step cannot be started before the previous "T" step is completed. After phase 1 is the elementwise multiplication ("M" step), followed by phase 2. Phase 2 contains similar alternating steps: inverse FFT/truncation (denoted as "I") and transpose.



(a) Another view of Figure 3. "F" means embedding and FFT, "T" means transpose, "M" means elementwise multiplication, "I" means inverse FFT and truncation.



(b) Pipeline one "F" step and one "T" step in (a). Both "F" and "T" are broken into r substeps.



(c) Data transpose in substeps. Gray data reside in one process, and white data reside in the other.

Figure 4: Pipelining the FFTs.

In each "T" step, a large portion of the data is moved between processes. The communication is of all-to-all type, one of the most demanding communications in MPI. The costly demand is caused by the latency for synchronization and the sheer volume of the data needed to be transferred.

One idea of saving the communications is to streamline one "F" step and one "T" step through overlapping [29]. Whereas the overlapping in [29] is performed on the FFTs of different data, we use overlapping on the FFTs of different pieces of one data. Figures 4(b) illustrates our idea. We break an "F" step into substeps and similarly for a "T" step, obeying data dependency. Once the first "F" substep is finished, the procedure forks the first "T" substep, which is performed concurrently with the second "F" substep. The concurrent execution requires a nonblocking all-to-all implementation (see MPI-3 standard, `MPI_Ialltoall`, [30]), which is provided by LibNBC (`NBC_Ialltoall`; see [31]) or MVAPICH2.[2] A synchronization (essentially a wait opertion) is needed before the third "F" substep to ensure that the first nonblocking all-to-all finishes. Then, the third "F" substep and the second "T" substep execute concurrently, and similarly for later substeps.

As an example, let us consider a two-dimensional data with two processes; see Figure 4(c). The data is initially partitioned along the column dimension; one process holds the gray data and the other the white data. A normal "F" step followed by a "T" step would mean jumping from the first box directly to the last one. On the other hand, if we do pipelining, we subdivide the column dimension into three slices for each process. In the first substep, FFT is done on the first slice, followed by a transpose. This results in the second box. Then, in the second substep, FFT and transpose are done on the second slice, resulting in the third box. One more similar substep leads to the last box.

In general, each dimension i of the process grid is associated with a slicing number m_i. In phase 1, when the S_{i+1} dimensions are unpartitioned, the "F" step that performs FFTs along the the S_{i+1} dimensions is pipelined with the following "T" step that transposes data between the S_i dimensions and the S_{i+1} dimensions. Because the S_i dimensions are subdivided into m_i slices, each "F" substep and "T" substep work on one of the slices; see Algorithm 4. The pipelining for phase 2 is similar.

Algorithm 4 Pipelining one "F" and one "T" step

// Assume that the S_{i+1} dimensions are unpartitioned
Embed and FFT on slice 1 of data
Start nonblocking all-to-all on slice 1 of data
for $j = 2$ to m_i **do**
    Embed and FFT on slice j of data
    Start nonblocking all-to-all on slice j of data
    Wait until all-to-all on slice $j - 1$ of data finishes
end for
Wait until all-to-all on slice m_i of data finishes

5. Orthogonalization

The purpose of orthogonalization (see line 10 of the right panel of Figure 1) is to detect linearly dependent columns of $P(:, \texttt{idx})$ for each $\texttt{idx} \in \texttt{IDX}$ and split them out from `idx`. To simplify notation, we temporarily omit the indicator `idx` and write $P \in \mathbb{R}^{n \times s_1}$, where s_1 is the cardinality of `idx`. A common approach of detecting linear dependence of the columns of P is to compute a rank-revealing factorization

$$P\Pi = QR, \qquad Q \in \mathbb{R}^{n \times s_1}, \ \Pi, R \in \mathbb{R}^{s_1 \times s_1}, \ s_1 \ll n, \tag{5}$$

where the columns of Q are orthonormal and Π is a permutation matrix. The term *rank-revealing* refers to the permutation such that if P has a rank $r < s_1$, the first r columns of Q form a basis of the range of P. One simple example of such a factorization is the (thin) QR factorization with pivoting, where R is upper triangular and its $r \times r$ leading principal submatrix is nonsingular.

We now put back the indicator `idx`. In a computer implementation, we let `IDX` be an ordered list, with a `front` pointer pointing to its first element and a `rear` pointer pointing to the last element. Each element `idx` has a `next` pointer that points to the next element in the list. Initially, `IDX` has only one element, $\texttt{idx} = \{1, 2, \ldots, s\}$. An abstraction of the orthogonalization is the following

[2] http://mvapich.cse.ohio-state.edu/overview/mvapich2/

Let `idx` ← `IDX.front`
while `idx` ≠ `IDX.rear.next` **do**
  Find a maximal subset `idx1` ⊂ `idx` such that columns of $P(:, \texttt{idx1})$ are linearly independent.
  Let `idx2` = `idx\idx1`.
  If `idx2` ≠ ∅, replace `idx` by `idx1` and add `idx2` to the end of `IDX`. Update `IDX.rear`.
  Update `idx` ← `idx.next`.
end while

After several block-CG iterations, `IDX` will have some element `idx` containing only one index; afterward, the linear system with respect to this index has converged. Then, `idx` is removed from `IDX`. When `IDX` becomes empty, all systems are solved.

Hence, the key question is how to find the maximal subset `idx1`? Two numerically stable methods are Householder (HOS) and modified Gram–Schmidt (MGS) [32]. These methods are not suitable for parallel implementations, however, because of the high cost in frequent all-reduce synchronizations. The classical Gram–Schmidt (CGS) requires fewer synchronizations, but it causes a severe loss of orthogonality. Furthermore, the number of required synchronizations is still more than one. The TSQR method [33, 34] computes local QR factorizations through HOS and iteratively combines local R factors and updates the local Q factors until Q is globally orthogonal. The total communications in all these combinations and updates are equivalent to one all-reduce operation. The breaking of one all-reduce into substeps of TSQR makes the synchronization more subtle. Effectively, the amount of transmitted data in TSQR is equal to all the initial triangular R factors. TSQR is as stable as HOS.

In our solver, we implement a slightly simpler method, SVQB [35], that requires only one straightforward all-reduce, wherein the amount of transmitted data is approximately the same as that of TSQR. The numerical stability of SVQB is close to that of MGS, sufficient for our use. Different from SVQB, however, in our implementation the matrix factors are not computed.

For notational consistency, let the matrices $\tilde{W}$, W, D, U, Λ in what follows all have size $s \times s$. First, a block inner product is computed:

$$\tilde{W}(\texttt{idx}, \texttt{idx}) = P(:, \texttt{idx})^T P(:, \texttt{idx}) \quad \text{for all } \texttt{idx} \in \texttt{IDX}. \tag{6}$$

This is where the only communication of the whole orthogonalization procedure appears; the block inner product can be computed with one all-reduce. Next, iterate through the list `IDX`, which might be dynamically adjusting. For each element `idx`, a diagonal normalization is performed:

$$W(\texttt{idx}, \texttt{idx}) = D(\texttt{idx}, \texttt{idx})^{-1/2} \tilde{W}(\texttt{idx}, \texttt{idx}) D(\texttt{idx}, \texttt{idx})^{-1/2}, \tag{7}$$

where D is the diagonal part of $\tilde{W}$. This results in a W whose diagonal is a constant 1. We want to find a maximal subset `idx1` ⊂ `idx` such that $W(\texttt{idx1}, \texttt{idx1})$ is numerically full rank. This calculation can be done by using a pivoted Cholesky factorization [36]. A more straightforward approach that directly uses LAPACK [37] routines is to perform a spectral decomposition:

$$W(\texttt{idx}, \texttt{idx}) U(\texttt{idx}, \texttt{idx}) = U(\texttt{idx}, \texttt{idx}) \Lambda(\texttt{idx}, \texttt{idx}),$$

where U contains the eigenvectors and Λ contains the eigenvalues. We then find a maximal subset of eigenvalues that are numerically nonzero. That is, using a threshold `eps`, let

$$\texttt{idx3} = \{k \in \texttt{idx} \mid |\Lambda(k,k)| > \texttt{eps} \cdot \lambda_{\max}\}, \quad \text{where} \quad \lambda_{\max} = \max_{k \in \texttt{idx}} \{|\Lambda(k,k)|\}.$$

In a usual numerical-rank computational procedure `eps` is chosen to be the machine precision `2.2204e-16`; we found through extensive validations, however, that setting `eps` to be `2.2204e-14` is a safer choice in the presence of diagonal normalization (7). A too-small `eps` cannot distinguish duplicate columns in an adversarial situation. Then, `idx1` is extracted through a pivoted-QR factorization of $U(\texttt{idx}, \texttt{idx3})$, where $U(\texttt{idx1}, \texttt{idx3})$ is square and has full rank.

Note that contrary to what the name of the procedure suggests, orthogonality is not essential. In fact, we never explicitly compute the orthogonalization (5). More important is to ensure that each resulting `idx` ∈ `IDX` contains linearly independent columns, so that in line 13 of Figure 1 the submatrix $\tau_j(\texttt{idx}, \texttt{idx})$ to be inverted is nonsingular. We further safeguard the iterations by replacing inverse with pseudoinverse in lines 13 and 20. In particular, the pseudoinverse is performed by deflating the small singular components of the matrix with respect to the singular values that are less than `eps` times the largest singular value. The appeal of this orthogonalization method is that the only communication used is for computing the block inner product (6). This type of communication facilitates the algorithmic rearrangement of CG, as discussed in the following section.

6. Eliminating All-Reduce

All-reduce is required for computing norms and inner products. The all-reduce synchronization causes processes first entering the synchronization point to wait for the later ones, hence wasting CPU cycles. A nonblocking version of all-reduce is not helpful, because the all-reduce result is immediately used in the following calculations, meaning that the availability of the result acts as a virtual synchronization point.

Let us examine closely why processes might enter the all-reduce synchronization point in succession rather than simultaneously. We focus on the matrix-vector multiplication of CG (line 11, left panel of Figure 1) as an example, because immediately following it is the inner product calculation. Recall the flowchart, Figure 4(a), of the matrix-vector multiplication. Before the final "I" step, an "I" step and a "T" step are pipelined by using the method illustrated in (b). Then, processes in the same communication subgroup enter the final "I" step concurrently, but processes in different subgroups are not synchronized. Thus, a time lag may exist between processes in different subgroups starting the final "I" step, and hence, finishing the step.

To improve concurrency, we may eliminate the synchronization point required by all-reduce. The approach we propose here rearranges the inner-product calculation. We still use line 11 and the line that follows as an example. Procedurally, these two consecutive steps are

$$\boldsymbol{v}_j = T\boldsymbol{p}_j \quad \rightarrow \quad \tau_j = \boldsymbol{p}_j^T \boldsymbol{v}_j. \tag{8}$$

Let the embeddings of $\boldsymbol{v}_j$, T, and $\boldsymbol{p}_j$ be denoted by $\boldsymbol{v}'_j$, C, and $\boldsymbol{p}'_j$, respectively, and let the multilevel circulant C admit a diagonalization $UCU^H = \Lambda$. Then, we have

$$\boldsymbol{v}'_j = C\boldsymbol{p}'_j = U^H \Lambda U \boldsymbol{p}'_j \quad \text{and} \quad \tau_j = \boldsymbol{p}'^T_j \boldsymbol{v}'_j = (U\boldsymbol{p}'_j)^H (U\boldsymbol{v}'_j).$$

The first relation is nothing but the method to compute Toeplitz matrix-vector multiplication through embedding and truncation. The second relation holds because U, the DFT matrix, is unitary. Therefore, the procedural steps (8) can be modified to

$$\underbrace{\boldsymbol{p}''_j = U\boldsymbol{p}'_j}_{\text{phase 1}} \quad \rightarrow \quad \underbrace{\boldsymbol{p}'''_j = \Lambda \boldsymbol{p}''_j}_{\text{“M” step}} \quad \rightarrow \quad \underbrace{\begin{cases} \boldsymbol{v}'_j = U^H \boldsymbol{p}'''_j \\ \tau_j = (\boldsymbol{p}''_j)^H (\boldsymbol{p}'''_j) \end{cases}}_{\text{phase 2}}. \tag{9}$$

One sees that τ_j is computed concurrently with phase 2 of the matrix-vector multiplication. The way to calculate this inner product is not to directly call all-reduce but, rather, to use the all-to-all communications in phase 2 to accumulate the local sums.

This idea is used in several places of CG and block CG, eliminating *all* the all-reduce synchronizations. The essence is to rewrite the inner products by using the DFT matrix and to utilize the inverse FFT communications to accumulate the rewritten, local sums. For norms, they even need not be rewritten, and the local sums can be accumulated through either the inverse FFT communications in phase 2 or the FFT communications in phase 1. In the following we list the places (following Figure 1) where this idea applies, and we summarize their implementations.

1. CG, lines 3 to 6. The preconditioning follows the same procedure: phase 1 → "M" step → phase 2. Hence, the norm ρ_0 is computed in phase 1; the inner product σ_0 is computed in phase 2. The convergence test is inserted between phase 1 and the "M" step. If all the systems have converged, the solver returns immediately, abandoning "M" step and phase 2.
2. CG, lines 11 to 12. Already explained.
3. CG, lines 16 to 19. Similar to item 1.
4. Block CG, lines 3 to 6. Similar to item 1.

5. Block CG, lines 10 to 12. Recall that the communication in the orthogonalization procedure occurs in computing a block inner product (see Section 5, in particular, (6)). Hence, this block inner product is computed in phase 1. Then, `IDX` is updated between phase 1 and the "M" step. The block inner product τ_j is computed in phase 2. Note that τ_j must be computed after `IDX` has been updated.
6. Block CG, lines 16 to 19. Similar to item 1.

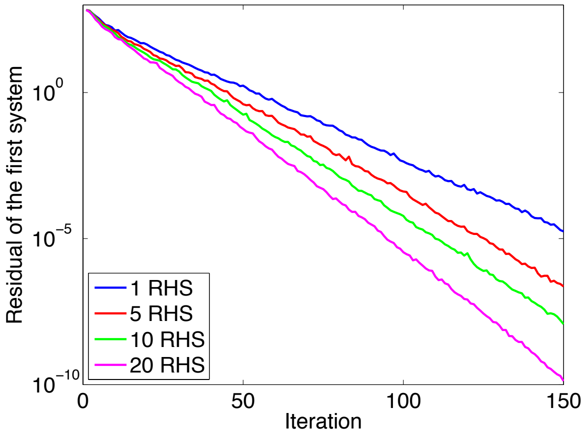
Note that the rearrangement affects the inner products and norms but not the matrix-vector products. Since the rearrangement applies unitary transformations, it is numerically stable. Hence, the numerical properties of CG and block CG are not changed.

7. Experiments

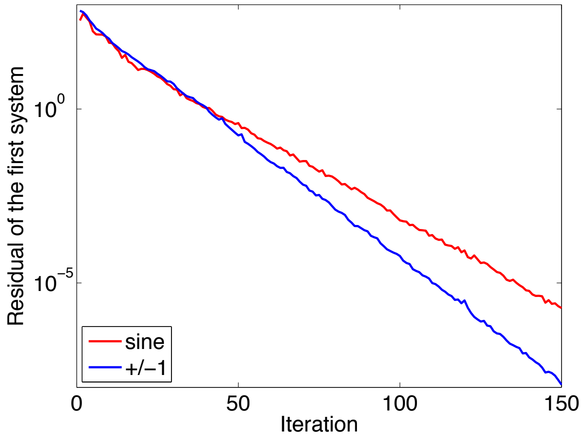
In this section we show experiments to demonstrate the correctness of the solver and the effectiveness of the techniques proposed in the preceding sections. The experiments were conducted on a cluster of 310 compute nodes, each of which has 16 cores (Intel Sandy Bridge) and 64 GB of memory. The machine uses QLogic QDR InfiniBand for interconnect, and thus we used the library LibNBC on InfiniBand [38] for performing nonblocking all-to-all communications. Because the best performance of LibNBC collectives is achieved by enabling asynchronous progress and by using a spare core for communication, we launched eight MPI processes on each compute node (recent hardware may offer an offloading feature at the network interface so that spare cores are not needed [29]). The MPI implementation is MVAPICH2. The in-process serial FFTs were computed by using the FFTW library without threading [25]. The multilevel Toeplitz matrix was generated from the Matérn kernel [15, 14, 16] on a regular grid. It is positive definite for any number of dimensions. The Matérn order was set to be 1.0; the physical range of the data was 100.0 along each dimension, whereas the kernel scale was set to be $7.0, 10.0, 13.0, 5.0, 12.0$ to make the kernel anisotropic. The experiments reported in this section cover the configurations of three-, four-, and five-dimensional data grids and two- and three-dimensional process grids.

Figure 5(a) shows the residual histories for solving the linear system with s right-hand sides, where $s = 1$, 5, 10, and 20, respectively. The right-hand sides are filled with independent and random values taking ± 1 with equal probability. We use a four-dimensional data grid partitioned by a two-dimensional process grid for illustration (details are presented in the caption of Figure 5). One clearly sees that the more right-hand sides, the faster the residuals decrease. For the same s, different residual curves are lumped together; thus we show only the history for the first system. All the systems were converged almost simultaneously, and there was no splitting of columns in orthogonalization. In plot (b), we fix $s = 10$ and show the residual histories for two types of right-hand sides. One type is the random ± 1's, whereas the other type generates the ith entry in the jth vector by following the rule $\sin((i + j - 2)\pi/50)$. For the latter type, one can show that every three consecutive vectors are linearly dependent. Thus, column splitting must happen in the orthogonalization procedure when we use zero as the initial guess. In fact, we examine the log and find that in the first iteration the ten right-hand sides are split in five groups of two. Because the number of right-hand sides becomes smaller in the group, the decrease in residuals is slower than the ± 1 case. We conclude that the behaviors shown in both plots well agree with theory.

Next, we investigate the effectiveness of pipelining the local FFT calculations with data transposes. We perform two sets of tests, one varying the data grid and fixing the process grid, whereas the other is the opposite. In both tests we vary the number m_i of slices. We show in Table 2 the average times of 30 repeating runs, each of which includes 60 matrix-vector multiplications, whereas in Table 3, the average times of 30 repeating runs of 30 CG iterations. The numbers in parentheses are standard deviations. In each table we highlight with boldface numbers the confidence intervals in which the shortest average run time lies. One sees that in most of the cases performing pipelining improves the running time. Particularly in the last column of the top part of the tables and in the first column of the bottom part, the improvement is substantial. In addition, the trends of the numbers in the two tables agree well. Clearly, the optimal running times are achieved at different m_i's for different data and process grids. A pattern of these optimal and near-optimal cases suggests that $n_1/(p_1m_1)$, $n_2/(p_2m_2)$, $n_3/(p_2m_3)$ be mostly 8 or 16. This implies that possibly the number of fibers in each pipelining substep should be fixed in order to achieve best performance. Further investigation of the optimal performance regarding the relationship between n_i's, p_i's, and m_i's can be done by carrying out performance tuning [39], but it is beyond the scope of this paper.



(a) Different numbers of right-hand sides.



(b) Different types of right-hand sides.

Figure 5: Residual history. Data grid $15 \times 24 \times 33 \times 16$. Process grid 4×4. $S_1 = \{1, 2\}$, $S_2 = \{3\}$, $S_3 = \{4\}$. All $m_i = 1$.

Table 2: Running times (in seconds) of 60 matrix-vector multiplications when the m_i's change.

Fixed process grid 8×8

Data grid→	$128 \times 128 \times 128$	$256 \times 256 \times 128$	$512 \times 256 \times 256$	$512 \times 512 \times 512$
m_i : 1,1,1	**4.52**(±**0.31**)	**18.54**(±**0.92**)	80.55(±2.03)	331.90(±4.62)
m_i : 2,2,2	**4.61**(±**0.21**)	**18.52**(±**1.10**)	**72.10**(±**2.26**)	276.82(±6.46)
m_i : 4,4,4	5.62(±0.57)	**18.80**(±**0.71**)	**70.77**(±**3.11**)	**264.83**(±**4.71**)
m_i : 8,8,8	7.94(±1.28)	**19.28**(±**0.84**)	**70.54**(±**3.00**)	**265.52**(±**4.77**)
m_i : 16,16,16	13.16(±0.87)	30.84(±4.93)	**72.13**(±**2.91**)	**262.03**(±**4.47**)

Fixed data grid $256 \times 256 \times 256$

Process grid→	4×4	8×8	16×16	32×32
m_i : 1,1,1	138.93(±3.81)	39.88(±1.16)	11.78(±0.68)	**3.70**(±**0.28**)
m_i : 2,2,2	127.04(±4.31)	**36.65**(±**1.40**)	**10.58**(±**0.25**)	5.08(±0.25)
m_i : 4,4,4	**122.19**(±**3.83**)	**36.82**(±**1.50**)	12.56(±0.67)	6.84(±0.08)
m_i : 8,8,8	**122.69**(±**6.46**)	38.54(±2.39)	26.14(±2.15)	12.16(±0.15)

We also evaluate the usefulness of eliminating all-reduce synchronizations in inner product calculations. First, we have verified the residual history of the modified algorithm that eliminates all-reduce against that of the standard algorithm shown in Figure 1. The histories are almost the same, indicating that the modified algorithm is numerically stable. In Table 4 we list the time ratios between the modified algorithm and the standard algorithm by varying the data grid and the process grid. These results were obtained by activating pipelining with all m_i's set to 2 for simplicity. One sees that all the ratios in the table are approximately 80%. Thus, eliminating all-reduce is always useful in improving concurrency and shortening the overall running time.

Another set of tests demonstrates the scaling of the solver. These tests include (a) a three-dimensional data grid partitioned by a two-dimensional process grid, and (b) a five-dimensional data grid partitioned by a three-dimensional process grid. In both cases, each dimension of the process grid partitions the corresponding dimension of the data grid, leaving the last dimension(s) of the data grid unpartitioned. Fixing a total data size and the grid dimension, the size along each dimension is set such that the the products $\prod_{j \in S_i} n_j$ are as even as possible, and that the n_j's in each S_i are as even as possible as well. We vary the process count from 16 to 1024. For case (a), we vary the data size from 64 million to 1 billion. Since the embedding doubles the size along each dimension, to be consistent, the data size for case (b) is downsized by four times compared with case (a). Moreover, we must use reasonable slicing numbers because pipelining has been shown to significantly improve the running time. According to the previous observation, it is fair to

Table 3: Running times (in seconds) of 30 CG iterations when the m_i's change.

Fixed process grid 8×8

Data grid→	$128 \times 128 \times 128$	$256 \times 256 \times 128$	$512 \times 256 \times 256$	$512 \times 512 \times 512$
m_i : 1,1,1	**2.69(±0.11)**	10.78(±0.45)	46.78(±1.07)	208.88(±3.14)
m_i : 2,2,2	**2.67(±0.08)**	**9.88(±0.27)**	**41.15(±1.59)**	178.43(±3.79)
m_i : 4,4,4	3.12(±0.15)	10.15(±0.23)	**41.22(±1.60)**	**170.22(±2.89)**
m_i : 8,8,8	5.18(±0.20)	12.11(±0.31)	**41.21(±1.10)**	**169.62(±4.66)**
m_i : 16,16,16	11.15(±0.72)	19.32(±1.07)	49.12(±1.63)	**168.87(±2.93)**

Fixed data grid $256 \times 256 \times 256$

Process grid→	4×4	8×8	16×16	32×32
m_i : 1,1,1	86.99(±6.10)	22.11(±0.63)	**5.82(±0.18)**	**2.22(±0.19)**
m_i : 2,2,2	79.42(±5.45)	**19.99(±0.61)**	**5.88(±0.15)**	3.10(±0.20)
m_i : 4,4,4	**75.14(±4.45)**	**20.15(±0.67)**	7.07(±0.21)	5.03(±0.09)
m_i : 8,8,8	**73.22(±3.00)**	21.98(±0.82)	11.69(±0.36)	11.14(±0.20)

Table 4: Time ratio between not using all-reduce and using all-reduce for inner product calculations. Both cases use pipelining.

Data grid→	$256 \times 256 \times 256$	$256 \times 256 \times 512$	$256 \times 512 \times 512$	$512 \times 512 \times 512$	$512 \times 512 \times 1024$
$p_i : 4 \times 4$	80.98%	80.34%	80.00%	78.66%	78.99%
$p_i : 4 \times 8$	82.30%	81.08%	81.95%	80.74%	80.87%
$p_i : 8 \times 8$	82.51%	81.02%	81.14%	80.88%	81.81%
$p_i : 8 \times 16$	82.82%	81.40%	80.85%	79.48%	80.31%
$p_i : 16 \times 16$	83.65%	82.49%	80.81%	79.46%	79.37%

set $n_1/(p_1m_1) = n_2/(p_2m_2) = n_3/(p_2m_3) = 8$ for case (a) and $n_1/(p_1m_1) = n_2/(p_2m_2) = n_3/(p_3m_3) = n_4n_5/(p_3m_4) = 8$ for case (b). Then, in Figure 6 we plot the running times. The times are measured as an average of 30 repeating runs, each of which contains 30 CG iterations. We see that the times for both cases are rather consistent. The solid curves indicate strong scaling, and the dashed ones indicate weak scaling. The percentage numbers are parallel efficiency for strong scaling by using the time at the smallest process count as the reference. Clearly, for case (a), strong scaling is close to perfect, and weak scaling has a slight increasing trend when the number of processes increases. This observation is repeated in case (b), except that the parallel efficiency is slightly lower. In both cases, the appealing scalings confirm the success of the solver in large-scale calculations.

In the last set of experiments, we compare the proposed solver with the standard direct solver implemented in ScaLAPACK [11], which is one of the few parallel solvers that can possibly handle multilevel Toeplitz systems in a "large scale." Nevertheless, the quadratic memory cost and cubic time cost fundamentally hinder the scalability of a direct solver and they pose severe difficulty for even supercomputers at $n = O(10^7)$ and beyond. Even when n is small/moderate, Table 5 shows that the proposed solver is advantageous over ScaLAPACK in both running time and memory consumption. The memory footprint was measured by using the `Valgrind` profiler `Massif`. We note that for the proposed solver, the profiled numbers are much larger than the `malloc`'ed memory size. This phenomenon might be caused by the memory registration needed in the InfiniBand software stack and by the LibNBC library for performing communications.

8. Application

We demonstrate an application of the linear solver in analyzing large-scale climate data based on Gaussian process modeling. Gaussian processes are a popular statistical model for characterizing data of a stochastic nature. Many components of climate data exhibit a Gaussian behavior. Here, we consider the downwelling solar flux at the Earth's surface and describe it by using a Gaussian process with a covariance kernel of certain smoothness. The modeling encloses the computational problem of parameter fitting and the applications of interpolation and forecasting. Our focus is the computational capability of Gaussian processes supported by the linear solver implemented in this paper.

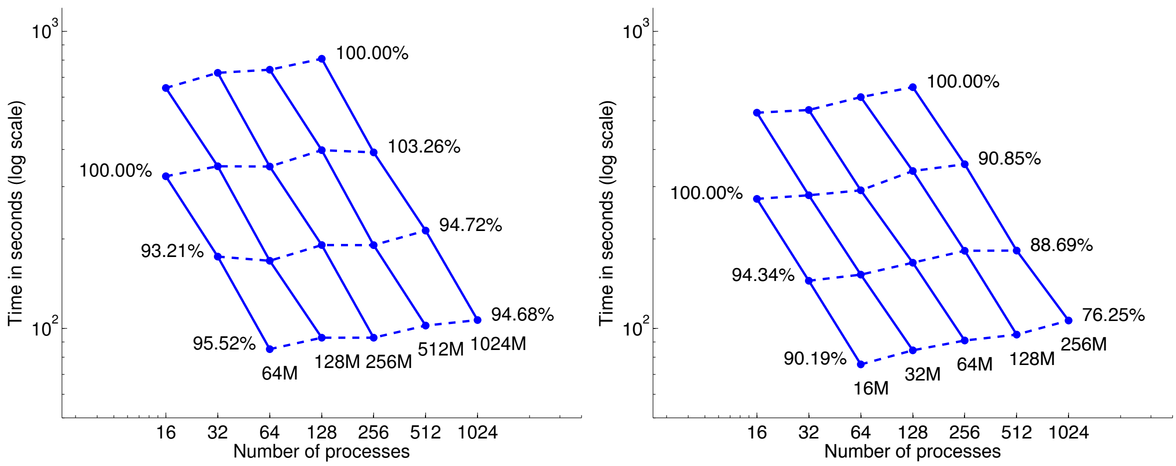


Figure 6: Strong scaling (solid) and weak scaling (dashed). Percentages are parallel efficiency. Numbers after each solid curve indicate data size. Left: Three-dimensional data grid partitioned by two-dimensional process grid. Right: Five-dimensional data grid partitioned by three-dimensional process grid.

Table 5: Comparison between the proposed solver and ScaLAPACK for solving multilevel Toeplitz systems. Relative residual tolerance for the proposed solver was `1.0e-16`. Block size for ScaLAPACK was 32×32. Neither solver was threaded.

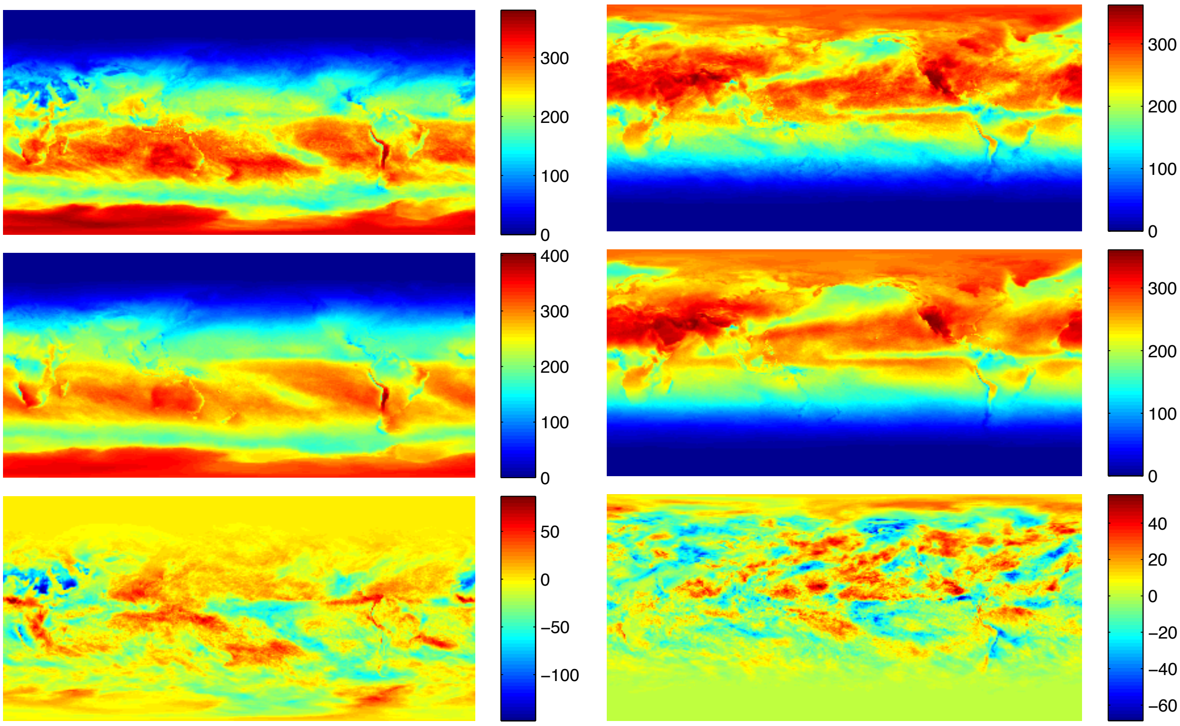
	Data grid (Matrix size)	128×128 ($16k \times 16k$)	256×256 ($64k \times 64k$)	$16 \times 16 \times 16$ ($4k \times 4k$)	$32 \times 32 \times 32$ ($32k \times 32k$)
Proposed solver	Process grid	4	16	2×2	4×4
	# iterations	703	2367	92	461
	Time (sec)	2.5075	11.624	0.4400	3.8538
	Memory (MB)	104.48	2448.7	29.699	478.88
ScaLAPACK	Process grid	2×2	4×4	2×2	4×4
	Time (sec)	241.47	2213.6	10.168	336.09
	Memory (MB)	2221.1	34265.	192.86	9104.4

The climate data[3] is a numerical simulation of the Community Atmosphere Model[4] that includes a system of partial differential equations whose initial conditions are given by physical measurements and that is solved by using spectral element methods. The simulation provides a much higher resolution output than that obtained through observations as the initial input. The output contains monthly average data (of 63 months) with a spatial resolution of 1 degree in both latitudes and longitudes. Thus, the total number of data points is over 4 million. Figure 7 shows two snapshots of the global distribution of the solar flux, one in November (top of (a)) and one in May (top of (b)). After removing the seasonal effects by subtracting the mean of the data in every 12 months, the resulting data as shown in the bottom row of Figure 7 is Gaussian-like, with low smoothness in correlation.

Denote by $\phi(\boldsymbol{x}, \boldsymbol{y})$ the covariance between two spatiotemporal locations $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^d$, and let Φ be the covariance matrix with $\Phi_{ij} = \phi(\boldsymbol{x}_i, \boldsymbol{x}_j)$, $i, j = 1, \ldots, n$. The problem of model fitting is to parameterize the covariance kernel ϕ and to obtain the parameters that best explain the data. Denote by $\boldsymbol{z}$ the demeaned data as shown in Figure 7, and let the set of parameters be $\boldsymbol{\theta}$. For Gaussians, the maximum likelihood fitting amounts to maximizing the log-likelihood

$$\mathcal{L}(\boldsymbol{\theta}) = -\frac{1}{2} \boldsymbol{z}^T \Phi^{-1} \boldsymbol{z} - \frac{1}{2} \log \det \Phi - \frac{n}{2} \log 2\pi,$$

[3]`http://trac.mcs.anl.gov/projects/parvis`
[4]`http://www.cesm.ucar.edu/models/cesm1.2/cam/`



(a) Month 1

(b) Month 7

Figure 7: Top: Solar flux on the earth surface. Middle: Periodic mean of the data. Bottom: Demeaned data (random field).

or equivalently, solving the first order optimality condition (as known as *score equations*)

$$\frac{1}{2}z^T\Phi^{-1}\frac{\partial\Phi}{\partial\theta_i}\Phi^{-1}z-\frac{1}{2}\operatorname{tr}\left(\Phi^{-1}\frac{\partial\Phi}{\partial\theta_i}\right)=0,\qquad\forall\, i. \tag{10}$$

To overcome the difficulty in evaluating the log-det term in the log-likelihood function and the trace term in the score equations when Φ is large, Anitescu et al. [18] proposed to use the approximate (but unbiased) score equations

$$\frac{1}{2}z^T\Phi^{-1}\frac{\partial\Phi}{\partial\theta_i}\Phi^{-1}z-\frac{1}{2N}\sum_{j=1}^{N}\boldsymbol{u}_j^T\Phi^{-1}\frac{\partial\Phi}{\partial\theta_i}\boldsymbol{u}_j=0 \tag{11}$$

for estimating the parameters, where, in the simplest form, the $\boldsymbol{u}_j$'s are i.i.d. symmetric Bernoulli vectors and there are N of them. Stein et al. [19] showed that the ratio between the variance of the estimates by solving (11) and that of the estimates by solving (10) is bounded and is independent of n when the condition number of Φ is bounded. Stein et al. [19] further showed several numerical examples where the ratio of variances grows slowly with n even when the condition number is unbounded. A computational advantage of this approach is that the evaluation of the approximate equations requires only matrix-vector multiplications and matrix-solves but not trace or determinant calculations. Here, since the data is observed on a regular grid, Φ is three-level Toeplitz. Then, the matrix-vector multiplications and the matrix-solves can be carried out efficiently by using the algorithms presented in this paper.

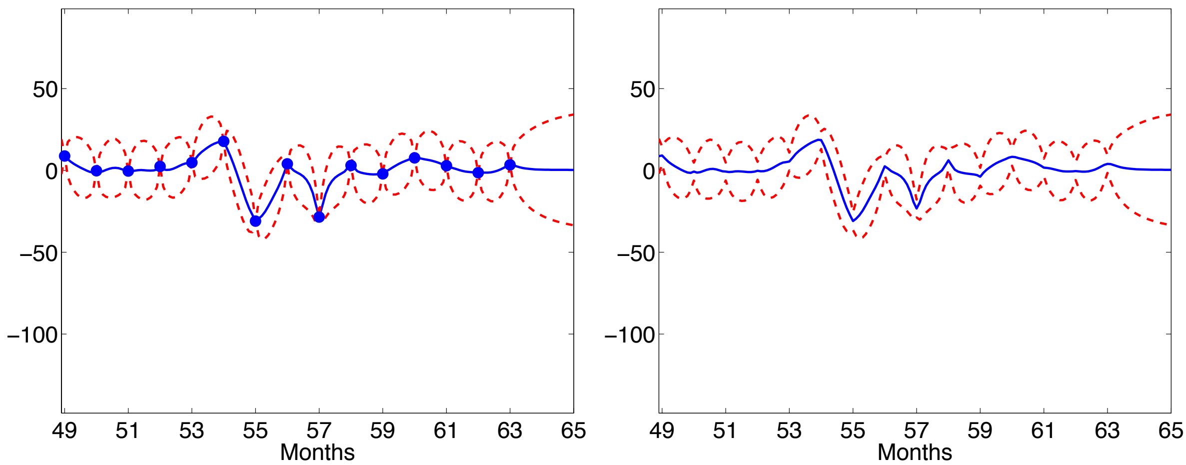
We hypothesized a Matérn covariance structure

$$\phi(\boldsymbol{x},\boldsymbol{y})=\theta_0\frac{(\sqrt{2\nu}r)^\nu\mathsf{K}_\nu(\sqrt{2\nu}r)}{2^{\nu-1}\Gamma(\nu)},\qquad r=\sqrt{\sum_{i=1}^{d}\left(\frac{x_i-y_i}{\theta_i}\right)^2}, \tag{12}$$

where θ_0, θ_1, θ_2, θ_3 are scale parameters to be fitted (K_ν is the modified Bessel function of the second kind of order ν). Because of the roughness exhibited in the data, we set $\nu = 0.5$. To impose physical meanings, we assumed that the data grid was in a cubic region of 180 degrees in latitude $\times$ 360 degrees in longitude $\times$ 63 months. The unit of the solar flux was W/m^2. A trust-region Newton method in the PETSc package [12] was used to solve (11). We started from an initial guess $\theta_0 = 50$ W/m^2, $\theta_1 = 1.8$ degrees in latitude, $\theta_1 = 3.6$ degrees in longitude, and $\theta_3 = 0.63$ months and reached a solution $\hat{\theta}_0 = 345.0(\pm 30.0)$ W/m^2, $\hat{\theta}_1 = 20.57(\pm 1.80)$ degrees in latitude, $\hat{\theta}_1 = 35.57(\pm 3.10)$ degrees in longitude, and $\hat{\theta}_3 = 1.724(\pm 0.154)$ months. The numbers in parentheses are two times the standard deviation (95% confidence interval). The solution was obtained with several trial-and-error adjustments in the numerical calculations. At this solution, the *rtol* for CG is `1e-8`, block size $s = 9$, and the number of random vectors $N = 8$. Note that a small N already yields sufficiently accurate estimates; the standard deviations of the results of (11) are only approximately 1.3 times those of (10). Because the calculation process was adjusted several times, the number of accumulated Newton iterations was postestimated to be several dozen. The number of block CG iterations for each Newton step ranged from several hundreds to one thousand. The cumulative CPU walltime was approximately 20 hours by using 256 MPI processes, including the original unsuccessful attempts in the guessing of Gaussian process starting parameters.

Using the fitted model (12), we performed kriging, a synonym of *interpolation* for existing data and of *forecasting* for the future. Figure 8 shows two kriging results. Plot (a) is a time series at the location 41.4385N,87.5W (the grid point closest to Chicago, USA), where the solid dots are the data; the curve interpolating them is the computational result; and the envelopes indicate a 95% confidence interval. The range of the vertical axis was set to be the same as that of the whole data so that the width of the envelopes is interpretable. The computational results at the grid locations must coincide with the known data, with a zero standard deviation. The curve after the last solid dot serves as a forecasting of the near future. It is not surprising that the confidence deteriorates quickly after the last known data point, indicating that forecasting cannot be reliable for a distant future.

Plot (b) shows another time series at the location 41.8384N,87.7W (somewhere in the Chicago metropolitan area). This location is not on the data grid. One sees that the kriging results are similar to those shown in plot (a). Note that the confidence interval is narrow for locations close to the grid.



(a) Location 41.4385N,87.5W (on grid).

(b) Location 41.8384N,87.7W (not on grid).

Figure 8: Kriged time series.

9. Conclusions

We have presented an implementation of the CG solver for solving multilevel Toeplitz linear systems that arise in various scientific and engineering applications, an example being statistical analysis of large-scale climate data as shown in the preceding section. The implementation uses a series of parallelization techniques in order to achieve a favorable scaling in a distributive computing environment using MPI. The largest experiment reported in this paper was carried out with a matrix size of more than one billion and a process count of more than one thousand.

Different from the implementation of iterative linear solvers for sparse matrices, as in popular software packages including PETSc and Trilinos, the techniques proposed in this paper exploit the multilevel Toeplitz structure and also reformulate the mathematical algorithm. The implementation treats single and multiple right-hand sides in a unified manner so that the interface can be simplified. The data-partitioning scheme and the pipelining idea ensure a communication-efficient program.

An avenue of future work is to better understand the contribution of different program parameters in terms of performance. This requires performance modeling and performance tuning. One of the important parameters is the number m_i of slices in pipelining. In the experiments we have derived a simple "rule of thumb" to determine a good m_i but this rule is hardly the final conclusion. Another performance factor is the configuration of the process grid given the data grid. When the data grid is in high dimension, the choice of the process grid is too flexible. This flexibility would turn to a burden for end users if no general guideline is provided.

Acknowledgment

This work was supported by the U.S. Department of Energy under Contract DE-AC02-06CH11357. We gratefully acknowledge use of the Fusion cluster and Blues cluster in the Laboratory Computing Resource Center at Argonne National Laboratory. We thank Robert Jacob of Argonne National Laboratory for sharing with us the climate data. We are indebted to Michael Stein of University of Chicago for the discussion of modeling and preprocessing of the data. We are also thankful to the three anonymous referees whose comments have substantially improved the paper.

References

[1] R. H. Chan, X.-Q. Jin, An Introduction to Iterative Toeplitz Solvers, SIAM, 2007.

[2] N. Levinson, The Wiener RMS error criterion in filter design and prediction, J. Math. Phys. 25 (1947) 261–278.

[3] J. Durbin, The Fitting of Time-Series Models, Review of the International Statistical Institute 28 (3) (1960) 233–244.

[4] I. Gohberg, I. Koltracht, A. Averbuch, B. Shoham, Timing Analysis of a Parallel Algorithm for Toeplitz Matrices on a MIMD Parallel Machine, in: Proceedings of International Conference on Parallel Processing (ICPP), 1991.

[5] E. H. Bareiss, Numerical solution of linear equations with Toeplitz and vector Toeplitz matrices, Numer. Math. 13 (1969) 404–424.

[6] R. P. Brent, Parallel algorithms for Toeplitz systems, in: Numerical Linear Algebra, Digital Signal Processing and Parallel Algorithms (edited by G. H. Golub and P. Van Dooren), Springer-Verlag, 1991.

[7] G. S. Ammar, W. B. Gragg, Superfast solution of real positive definite Toeplitz systems, SIAM J. Matrix Anal. Appl. 9 (1) (1988) 61–76.

[8] M. Stewart, A Superfast Toeplitz Solver with Improved Numerical Stability, SIAM J. Matrix Anal. Appl. 25 (3) (2003) 669–693.

[9] S. Chandrasekaran, M. Gu, X. Sun, J. Xia, J. Zhu, A Superfast Algorithm for Toeplitz Systems of Linear Equations, SIAM J. Matrix Anal. Appl. 29 (4) (2007) 1247–1266.

[10] V. Y. Pan, Concurrent Iterative Algorithm for Toeplitz-like Linear Systems, IEEE Transactions on Parallel and Distributed Systems 4 (5) (1993) 592–600.

[11] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, R. C. Whaley, ScaLAPACK Users' Guide, Society for Industrial and Applied Mathematics, Philadelphia, PA, ISBN 0-89871-397-8 (paperback), 1997.

[12] S. Balay, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, H. Zhang, PETSc Users Manual, Tech. Rep. ANL-95/11 - Revision 3.4, Argonne National Laboratory, 2013.

[13] M. Heroux, R. Bartlett, V. H. R. Hoekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, A. Salinger, H. Thornquist, R. Tuminaro, J. Willenbring, A. Williams, An Overview of Trilinos, Tech. Rep. SAND2003-2927, Sandia National Laboratories, 2003.

[14] M. Stein, Interpolation of Spatial Data: Some theory for Kriging, Springer-Verlag, 1999.

[15] J.-P. Chilès, P. Delfiner, Geostatistics: Modeling Spatial Uncertainty, Wiley-Interscience, 1999.

[16] H. Wendland, Scattered Data Approximation, Cambridge University Press, 2005.

[17] C. Rasmussen, C. Williams, Gaussian Processes for Machine Learning, MIT Press, 2006.

[18] M. Anitescu, J. Chen, L. Wang, A matrix-free approach for solving the parametric Gaussian process maximum likelihood problem, SIAM J. Sci. Comput. 34 (1) (2012) A240–A262.

[19] M. L. Stein, J. Chen, M. Anitescu, Stochastic Approximation of Score Functions for Gaussian Processes, Annals of Applied Statistics 7 (2) (2013) 1162–1191.

[20] Y. Saad, Iterative Methods for Sparse Linear Systems, SIAM, 2nd edn., 2003.

[21] D. P. O'Leary, The block conjugate gradient algorithm and related methods, Linear Algebra Appl. 29 (1980) 293–322.

[22] A. A. Nikishin, A. Y. Yeremin, Variable Block CG Algorithms for Solving Large Sparse Symmetric Positive Definite Linear Systems on Parallel Computers, I: General Iterative Scheme, SIAM J. Matrix Anal. Appl. 16 (4) (1995) 1135–1153.

[23] T. F. Chan, An Optimal Circulant Preconditioner for Toeplitz Systems, SIAM J. Sci. Stat. Comput. 9 (4) (1988) 766–771.

[24] S. S. Capizzano, Matrix algebra preconditioners for multilevel Toeplitz matrices are not superlinear, Linear Algebra Appl. 343–344 (1) (2002) 303–319.
[25] M. Frigo, S. G. Johnson, The Design and Implementation of FFTW3, Proceedings of the IEEE 93 (2) (2005) 216–231.
[26] D. Pekurovsky, P3DFFT: a framework for parallel computations of Fourier transforms in three dimensions, SIAM J. Sci. Comput. 34 (4) (2012) C192–C209.
[27] A. Canning, Scalable Parallel 3D FFTs for Electronic Structure Codes, in: High Performance Computing for Computational Science - VECPAR 2008, Springer, 280–286, 2008.
[28] A. Canning, J. Shalf, N. Wright, S. Anderson, M. Gajbe, A Hybrid MPI/OpenMP 3d FFT for Plane Wave First-principles Materials Science Codes, in: Proceedings of CSC12 Conference, 2012.
[29] K. Kandalla, H. Subramoni, K. Tomko, D. Pekurovsky, S. Sur, D. K. Panda, High-performance and scalable non-blocking all-to-all with collective offload on InfiniBand clusters: a study with parallel 3D FFT, Computer Science - Research and Development 26 (3-4) (2011) 237–246.
[30] Message Passing Interface Forum, MPI: A Message-Passing Interface Standard, Version 3.0, `http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf`, 2012.
[31] T. Hoefler, A. Lumsdaine, W. Rehm, Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI, in: Proceedings of the 2007 International Conference on High Performance Computing, Networking, Storage and Analysis, 2007.
[32] G. H. Golub, C. F. Van Loan, Matrix Computations, Johns Hopkins University Press, 1996.
[33] J. W. Demmel, L. Grigori, M. F. Hoemmen, J. Langou, Communication-optimal parallel and sequential QR and LU factorizations, Tech. Rep. UCB/EECS-2008-89, University of California Berkeley, also appears as LAPACK Working Note #204, 2008.
[34] M. Hoemmen, A Communication-Avoiding, Hybrid-Parallel, Rank-Revealing Orthogonalization Method, in: Proceedings of 2011 IEEE International Parallel & Distributed Processing Symposium, 2011.
[35] A. Stathopoulos, K. Wu, A Block Orthogonalization Procedure with Constant Synchronization Requirements, SIAM J. Sci. Comput. 23 (6) (2002) 2165–2182.
[36] C. Lucas, LAPACK-Style Codes for Level 2 and 3 Pivoted Cholesky Factorizations, Tech. Rep. 442, University of Manchester, also appears as LAPACK Working Note #161, 2004.
[37] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, D. Sorensen, LAPACK Users' Guide, Society for Industrial and Applied Mathematics, 3rd edn., 1999.
[38] T. Hoefler, A. Lumsdaine, Optimizing non-blocking Collective Operations for InfiniBand, in: Proceedings of the 22nd IEEE International Parallel & Distributed Processing Symposium, 2008.
[39] D. H. Bailey, R. F. Lucas, S. Williams (Eds.), Performance Tuning of Scientific Applications, Chapman & Hall/CRC Computational Science, CRC Press, 2010.