SIAM J. SCI. COMPUT.
Vol. 42, No. 5, pp. C245–C264

A FLEXIBLE FRAMEWORK FOR MULTIDIMENSIONAL DFTs*

DORU THOM POPOVICI†, MARTIN D. SCHATZ‡, FRANZ FRANCHETTI†, AND TZE MENG LOW†

Abstract. Multidimensional discrete Fourier transforms (DFTs) are typically decomposed into multiple one-dimensional (1D) transforms. Hence, parallel implementations of any multidimentional DFT focus on parallelizing within or across the 1D DFT. Existing DFT packages exploit the inherent parallelism across the 1D DFTs and offer rigid frameworks, that cannot be extended to incorporate both forms of parallelism and various data layouts to enable some of the parallelism. However, in the era of exascale, where systems have thousand of nodes and intricate network topologies, flexibility and parallel efficiency are key aspects all multidimentional DFT frameworks need to have in order to map and scale the computation appropriately. In this work, we show the need for a versatile parallel framework that facilitates the development of a family of parallel multidimentional DFT algorithms by (1) using different data layouts to distribute the data across the compute nodes, (2) exploiting the two different parallelization schemes to different degrees, and (3) unifying the two parallelization schemes within a single framework. We show that the flexibility of selecting different parallel multidimentional DFT algorithms allows for almost linear strong scaling results for problem sizes of 1024^3 on two supercomputers, namely, RIKEN's K-Computer and Oakridge's Summit.

Key words. 3D DFTs, distributed systems, performance, scalability

AMS subject classifications. 68N01, 65T50, 68W15

DOI. 10.1137/19M1288401

1. Introduction.

The multidimentional discrete Fourier transform (DFT) has proven to be an ubiquitous mathematical kernel, that is widely used in a multitude of applications from different scientific fields like molecular dynamics [19, 21, 20], material sciences [15, 14, 16], beam-plasma simulations [34, 35], and quantum mechanics [13, 32, 28, 3]. As we move into the exascale era with massively parallel systems that have thousands of compute nodes, it is vital that parallel multidimentional DFTs be efficient. Multidimensional DFTs are defined in terms of multiple one-dimensional (1D) DFTs. Hence, parallel implementations of the multidimentional DFTs can be classified into two distinct classes. The first class focuses on exploiting parallelism within the 1D DFTs [29, 7], while the second class exploits the inherent parallelism across multiple distinct 1D DFTs. Most state-of-the-art frameworks for computing three-dimensional (3D) DFTs, like FFTW [8], P3DFFT [18], FFTE [31], opt to parallelize across the 1D DFTs.

We illustrate the limitations of the conventional approach of parallelizing multidimentional DFTs in Figure 1, where we report the performance of three different parallelization schemes for computing the 3D DFT of 64^3 and 1024^3 on the K-Computer. Notice that for the data set of size 64^3, parallelizing the 3D DFT on a 1D grid of processors such that each processor computes a two-dimensional (2D) plane yielded the

*Submitted to the journal's Software and High-Performance Computing section September 24, 2019; accepted for publication (in revised form) June 16, 2020; published electronically September 17, 2020.

https://doi.org/10.1137/19M1288401

Funding: This work was supported by DARPA through grants HR0011-13-2-0007 and FA8750-16-2-003, by the NSF through grant ACI 1550486, and by the Exascale Computing Project 17-SC-20-SC.

†ECE, Carnegie Mellon University, Pittsburgh, PA 15213 (dpopovic@alumni.cmu.edu, franzf@cmu.edu, lowt@cmu.edu).

‡Department of Computer Science, University of Texas at Austin, Austin, TX 78712 (martin.schatz@utexas.edu).

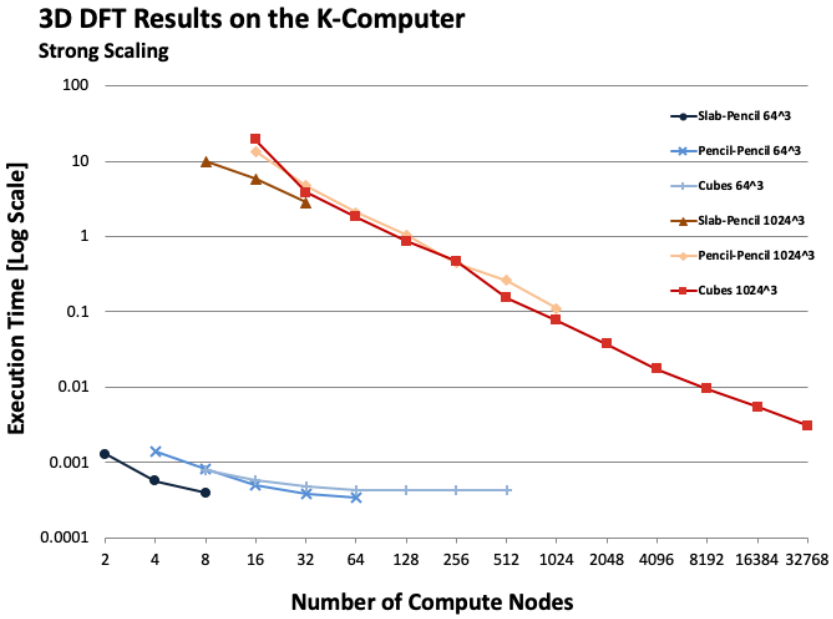

FIG. 1. *Strong scaling results for 3D DFTs of size* 64^3 *and* 1024^3 *using the slab-pencil, pencil-pencil-pencil, and volumetric decompositions on the K-Computer. We report the execution time on the log scale.*

shortest execution time. However, for the 1024^3 input, the same parallelization scheme scales only to 32 processors, while better performance can be attained by switching to a parallelization scheme comprising a 2D grid of processors where each processor computes a small batch of 1D DFTs. Just as with the 1D parallelization scheme, the parallelism on a 2D grid of processors is restricted to at most 1024 processors. By parallelizing within the 1D DFT for the last dimension, we obtain an even shorter execution time using a much greater number of processors. These observations highlight two major limitations of existing approaches: (1) there is a need for one to seamlessly switch between parallel algorithms for computing the multidimentional DFT as different parallel algorithms are superior for different problem sizes, and (2) there is a need to parallelize both across and within the 1D DFTs to scale to larger number of nodes.

There is an increasing interest in scaling the parallel implementations of the multidimentional DFT to higher number of nodes using higher-dimensional grids as shown in Jung et al. [12]. These newer algorithms are a subset of the parallel DFT algorithms highlighted in Johnson and Xu [11], that showed that a significant number of algorithms can arise from the parallelization both within and across multiple 1D DFTs. However, these algorithms are largely unexplored in practice as the existing parallel DFT frameworks make implementing them quite difficult. Due to the rigidity of the existing frameworks, the resulting performance is still unsatisfactory. Therefore, in this work we show that a flexible framework for employing different parallelization schemes to compute the multidimentional DFT on a multidimentional computation grid of processors is needed. By recognizing that the inputs and outputs of a multidimentional DFT are multidimentional arrays of data, we leverage insights from the multilinear algebra (tensor) community to simplify the management of network communication and data layout on a multidimentional grid.

Contributions. This paper makes the following contributions:

- Demonstrates the need for a family of parallel DFT algorithms in order to attain high performance for a given problem size and network configuration.
- Introduces a new data distribution inspired by the multilinear algebra (tensor) community to facilitate the parallelization of the multidimentional DFTs.
- Shows that a flexible framework that unifies both parallelization schemes for multidimentional DFTs on multidimentional grids can be built on a multilinear algebra (tensor) framework.

2. The discrete Fourier transform.

In this section, we briefly present the decomposition of both the 1D and multidimentional DFTs, expressing the algorithms in terms of linear algebra operations and outlining the opportunities for parallelization.

2.1. The 1D discrete Fourier transform.

The 1D DFT is a matrix-vector multiplication, where given the input x, the output y is obtained as

$$y = DFT_n \cdot x. \tag{2.1}$$

The DFT_n is the $n \times n$ DFT dense matrix, defined as

$$DFT_n = \left[\omega_n^{lk}\right]_{0 \le l,k < n} \tag{2.2}$$

with $\omega_n = e^{-k\frac{2\pi}{n}}$ being the complex root of unity. Typically, the computation of the DFT is implemented using the fast Fourier transforms (FFT), where instead of performing $O(n^2)$ complex arithmetic operations by doing the matrix-vector multiplication, a recursive decomposition of the DFT matrix is performed to obtain an $O(n \log(n))$ algorithm. The most widely known of these algorithms is the algorithm in [5]. The Cooley–Tukey algorithm is described as a factorization of the DFT_n matrix, when n is a composite number such as $n = n_0 n_1$.

An alternative view of the mathematical description of the Cooley–Tukey algorithm defined in [5] is as follows:

$$\tilde{y} = (Twid_{n_0 \times n_1} \odot (\tilde{x} \cdot DFT_{n_1}))^T \cdot DFT_{n_0}, \tag{2.3}$$

where $\tilde{x}$ and $\tilde{y}$ are two matrices of size $n_0 \times n_1$ and $n_1 \times n_0$, that represent the input x and output y, respectively. The columns of matrix $\tilde{x}$ ($\tilde{y}$) are n_1 (n_0) contiguous subvectors (i.e., x_i, $0 \le i < n_0$ or y_i, $0 \le i < n_1$) of the input (output) vector x (y), and are obtained by splitting x (y) into subvectors of size n_0 (n_1) as follows:

$$\tilde{x} = \left[x_0 \quad | \quad x_1 \quad | \quad \dots \quad | \quad x_{n_1-1}\right].$$

The matrix $\tilde{y}$ is similarly constructed.

The reason for the formulation of the Cooley–Tukey algorithm as described by (2.3) is that the four stages of the algorithm (as depicted in Figure 2) are made explicit. The first step applies the DFT_{n_1} on the rows of the matrix $\tilde{x}$ (I), assuming data are stored in column major order. The elements within each column of either matrix $\tilde{x}$ or $\tilde{y}$ are in consecutive locations in memory. This implies that in order to compute the first stage of the DFT, elements are accessed at a stride of n_0. The result of the first stage is then pointwise multiplied with the so-called twiddle matrix (II), which is defined as

$$Twid_{n_0 \times n_1} = \left[\omega_n^{kl}\right]_{0 \le l < n_0 \text{ and } 0 \le k < n_1}, \tag{2.4}$$

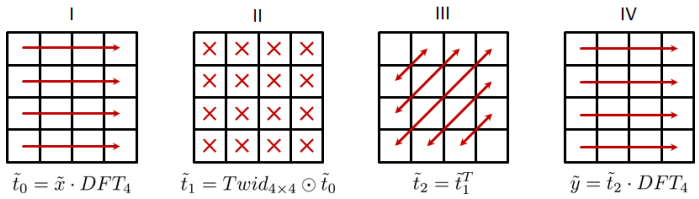

FIG. 2. *The decomposition of the* 1*D DFT using the Cooley–Tukey algorithm for a given problem size of* $16 = 4 \cdot 4$. *The algorithm requires four steps, namely, two batched DFTs* (I) *and* (IV) *of size* 4, *a pointwise operation* (II), *and a transposition* (III). *Data are stored in column major order.*

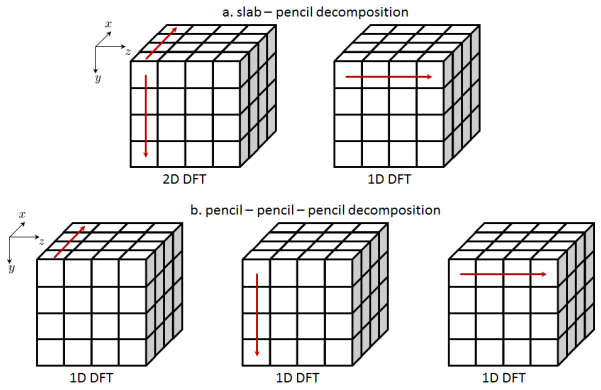

FIG. 3. *Two algorithms for computing the* 3*D DFT. The slab-pencil algorithm* (a) *decomposes the* 3*D DFT into a* 2*D DFT and a* 1*D DFT. When applied, the* 2*D DFT is decomposed into the corresponding* 1*D transforms. The pencil-pencil-pencil algorithm* (b) *decomposes the* 3*D DFT into three* 1*D DFTs applied in the corresponding dimensions.*

where ω_n represents the complex roots of unity. The resultant matrix is then transposed (III) and finally the second DFT of size n_0 is applied in the rows (IV). Again, the elements required for the fourth stage of the computation are read at a stride. Due to the transposition, the second DFT cannot start computation until all previous stages have been completed.

2.2. *n*-Dimensional DFTs. Multi-dimensional DFTs can be defined in terms of multiple 1D DFTs and multidimentional DFTs of lower dimensions. For example, Figure 3 shows two variants for computing the 3D DFT. The first algorithm represents the so-called slab-pencil decomposition [29, 6, 30], where the 3D DFT is decomposed into a batch of 2D DFTs followed by a batch of multiple 1D DFTs. The second algorithm represents the pencil-pencil-pencil decomposition [29, 6, 30], where the 3D DFT is decomposed into three batches of 1D DFTs, where each 1D DFT is applied in the three dimensions.

The slab-pencil algorithm views the input (output) column vectors x (y) as 2D matrices $\tilde{x}$ ($\tilde{y}$) of size $(n_0 n_1) \times n_2$. Mathematically the decomposition is expressed as

$$\tilde{y} = (DFT_{n_0 \times n_1} \cdot \tilde{x}) \cdot DFT_{n_2}. \tag{2.5}$$

Data are stored in column major, hence the 2D DFT is applied on the columns, while the 1D DFT is applied on the rows.

The pencil-pencil-pencil algorithm reshapes the input (output) vectors into 3D cubes $\hat{x}$ ($\hat{y}$) of size $n_0 \times n_1 \times n_2$. The input (output) column vector x (y) is decomposed

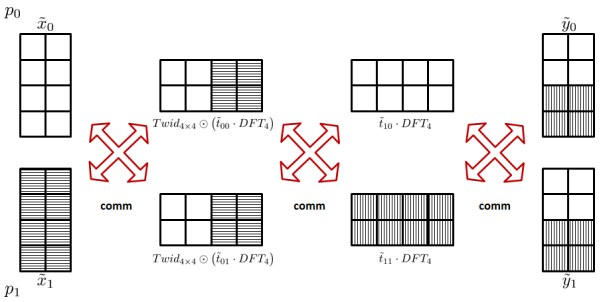

FIG. 4. *The parallel implementation of the Cooley–Tukey algorithm for the* DFT_{16} *on* $p = 2$ *processors. Data are distributed between the processors, each processor does its local computation, and then exchanges the data. This parallel implementation requires three communication stages.*

into n_2 groups of n_1 subgroups of size n_0. Hence, the 3D cube $\hat{x}$ can be viewed as a matrix of matrices such as

$$\hat{x} = \left[\tilde{x}_0 | \quad \tilde{x}_1 | \quad \dots \quad | \quad \tilde{x}_{n_2}\right], \tag{2.6}$$

where each $\tilde{x}_i$ is the 2D matrix of size $n_0 \times n_1$ for all values $0 \leq i < n_2$. Mathematically, the pencil-pencil-pencil algorithm is expressed as

$$\hat{y} = \left[(DFT_{n_0} \cdot \tilde{x}_0) \cdot DFT_{n_1} | \quad \dots \quad | \quad (DFT_{n_0} \cdot \tilde{x}_{n_2}) \cdot DFT_{n_1}\right] \cdot DFT_{n_2}, \tag{2.7}$$

where the DFT_{n_0} is applied in the depth dimension, the DFT_{n_1} is applied in the column dimension, and finally the DFT_{n_2} is applied in the row dimension. Data are stored in column major and therefore the dimension corresponding to n_0 is laid out in the fastest dimension in memory, while the dimension corresponding to n_2 is laid out in the slowest dimension in memory.

2.3. Parallelizing the 1D DFT. Parallelizing the 1D DFT on p processors requires the parallelization of all four compute stages. Traditionally, the input matrix $\tilde{x}$ (stored in column major order) is split such that each processor receives n_1/p columns of size n_0. Distributing the data using this data layout implies that computation cannot start since each processor does not have the necessary data points. As such, an all-to-all communication step is needed to redistribute the data such that each processor receives n_0/p rows of size n_1. The first and second stage of the Cooley–Tukey algorithm can then be applied locally. A second all-to-all communication is performed to transpose (stage III) the data across the processors. After this communication step, each processor owns n_1/p rows of size n_0 and thus can locally compute the last stage of the algorithm. Storing the data in the correct order requires a third communication step. The described computation gives the so-called six step algorithm [33] defined as

$$\tilde{y} = \left(DFT_{n_0} \cdot \left(Twid_{n_0 \times n_1} \odot \left(DFT_{n_1} \cdot \tilde{x}^T\right)\right)^T\right)^T. \tag{2.8}$$

The parallel implementation of the six step algorithm is depicted in Figure 4, and it requires a total of three communication steps. The astute reader will recognize that *some of the communication steps can be avoided by storing the initial data in a different layout. This has been observed in literature, but seldom exploited in practice.*

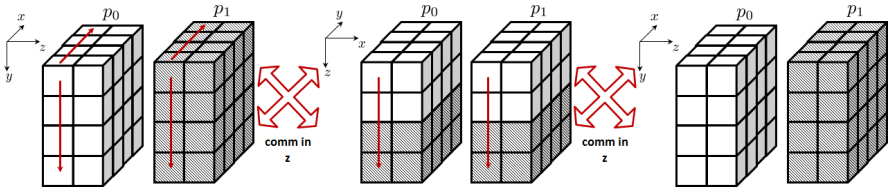

FIG. 5. *The parallel implementation of the slab-pencil algorithm for computing the* 3*D DFT on a* 1*D mesh of two processors. Each processor applies batches of* 2*D DFTs followed by batches of* 1*D DFTs. The implementation requires two communications stages.*

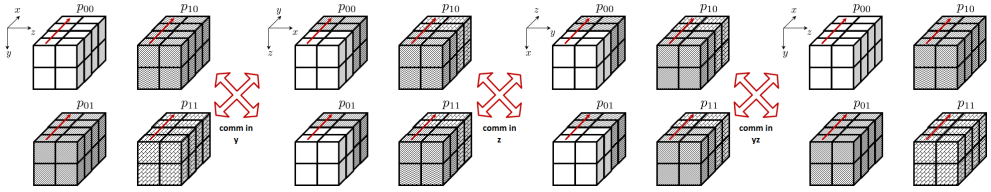

FIG. 6. *The parallel implementation of the pencil-pencil-pencil algorithm for computing the* 3*D DFTs on a* 2*D mesh of size* 2×2 *compute nodes. Each processor computes batches of* 1*D DFTs in each dimension. The implementation requires three communications stages.*

2.4. Parallelizing the n-dimensional DFTs. Since the multidimentional DFT can be decomposed into multiple lower-dimensional DFTs, most prevalent DFT frameworks exploit this feature for parallelization. We illustrate this parallelization scheme on the two previously presented algorithms for computing the 3D DFT. Parallelizing the slab-pencil algorithm is performed by viewing the processors as a 1D grid and distributing the data cube such that each processor receives n_2/p 2D slabs of data of size $n_0 \times n_1$. Each processor then locally computes multiple 2D DFTs on the given slabs. Data are then exchanged across the processors so that each processor can compute $(n_0 \times n_1)/p$ 1D DFTs of size n_2. Finally, another communication step is required to store the data in the correct format, as shown in Figure 5.

The pencil-pencil-pencil algorithm is parallelized in the two dimensions corresponding to n_1 and n_2, across a mesh of processors of size $p_y \times p_z$. Each processor receives a batch of 1D pencils as seen in Figure 6. Each processor can apply 1D DFTs locally. In order to apply the other two 1D DFTs data are first exchanged between the processors in the p_y dimension of the mesh, followed by the communication between the processors on the p_z dimension. Each communication stage rotates the data cube from xyz to yzx and zxy as seen in Figure 6. Each communication step requires data to be packed and unpacked before and after the all-to-all communication. Finally, in order to store the data in the correct format a global communication between all processors is required. In practice, the overall execution time is reduced by dropping the last communication stage. However, this comes at the cost of having to compute the subsequent computations on shuffled data points.

2.5. Limitations of existing parallelization schemes. Notice that in both cases, the amount of parallelism is limited by the size of a particular dimension of the data. For the slab-pencil algorithm, the maximum number of processors that can participate in the computation is $\max(n_0, n_1, n_2)$ as each processor is assigned a 2D slab of data. Similarly for the pencil-pencil-pencil case, the number of participating processors is upper bound by $\max(n_0 n_1, n_1 n_2, n_0 n_2)$ since each processor is assigned

at least one 1D DFT to compute locally. However, using the maximum number of processors in each dimension requires extra communication steps, unless data are left permuted. The level of parallelism within the communication is also limited by the algorithm. The slab-pencil distribution spreads the data on a 1D grid. Hence, data exchange for computing the 1D DFT requires the communication between all processors. The pencil-pencil-pencil case parallelizes the communication in two dimensions, which is done between groups of processors. All frameworks that implement the parallelization across the 1D DFT depend on efficient all-to-all communications [2, 25]. However, as we will briefly show in the results section, the all-to-all communication cannot efficiently scale as the number of processors increases.

3. Parallelize n-dimensional DFTs in all n dimensions. In this section, we present the approach for combining both forms of parallelism for computing n-dimensional DFTs, focusing on the 3D case. First, we discuss the data layouts and how they influence the communication of the parallel 1D DFT. Then, we use the elemental cyclic distribution described in [24, 26] and outline the steps required to compute a parallel 3D DFT. We emphasize the connection with the multilinear algebra domain, since any multidimentional DFT can be decomposed as operations on the dimensions of high dimension tensors, where the inputs and outputs are viewed as multidimentional arrays.

3.1. Data layouts. Data need to be distributed across the processing nodes. Depending on how the data are distributed across the network, the 1D DFT computation may require one or more communication stages. In the following paragraphs, we present three data layouts, block cyclic, blocked, and elemental cyclic, and discuss how they influence the number of communication stages for a parallel 1D DFT.

The block cyclic distribution is de facto data distribution for linear algebra frameworks such as ScaLAPACK [4]. Figure 7(B) shows the block cyclic layout for a 16 element vector across $p = 2$ processing nodes using a block size $b = 2$ elements. The 1D vector is split into eight chunks of two elements, and each chunk is round robin distributed across the two processing nodes. Given this data layout, the 1D DFT applied to the input vector requires only one communication step. Viewing the 1D vector as a 2D matrix as shown in Figure 2, the block cyclic distribution spreads out the data such that each of the two processing nodes receives b sequential rows. Each processor can therefore start its local computation. After the inherent transposition and the final local computation (stages III and IV of the 1D DFT algorithm), the block cyclic distribution is preserved from input to output.

The number of communication steps for the 1D DFT is one and the block cyclic distribution is preserved, if the following relationship,

$$p^2 \leq \frac{n}{b^2}, \tag{3.1}$$

holds for a given problem size n, a blocking size b, and p processing nodes. The problem size n must be divisible by b^2p^2. As such, the 1D DFT can at most be distributed across $\sqrt{n/b^2}$ processing nodes. If the number of processing nodes is increased beyond that value, the number of communication steps is increased as shown in Inda and Bisseling [10], and possibly the block cyclic distribution cannot be maintained from input to output. For this work, we focus on the cases, where the constraints are satisfied, and we leave the other cases as future work.

The blocked distribution is the preferred data layout for most DFT-based frameworks [31, 9, 18]. The blocked data layout is a special case of the block cyclic distribu-

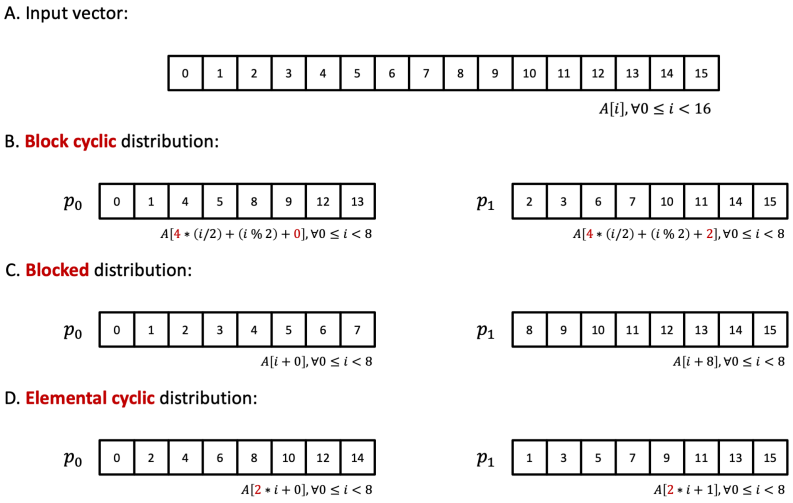

FIG. 7. *Distributions of a* 16 *element* 1*D vector across two nodes,* p_0 *and* p_1*. The block cyclic layout distributes blocks of size two elements in a round robin fashion. The blocked layout distributes eight sequential elements to each processor. The elemental cyclic layout scatters the even location points to the first processor and the odd location points to the second compute node.*

tion, when $b = n/p$. Figure 7(C) shows the blocked layout for the 16 element vector on $p = 2$ processing nodes. The 1D vector is split evenly between the two nodes, each receiving a continuous block of eight elements. As outlined in the previous section, the parallel 1D DFT of size 16 requires three communication stages. Viewing the blocked distributed 1D array as a 2D matrix, each processor receives multiple columns of the original input vector. Since computation is done in the rows, a first communication step is required. A final communication step is needed to preserve the data distribution. The blocked distribution imposes that the problem size n be divisible by p^2 if the processor grid or the data layout is to remain invariant for the duration of the computation. This suggests that a 1D DFT of size n can at most be parallelized on $\sqrt{n}$ compute nodes.

The elemental cyclic distribution is used in linear algebra frameworks [24, 26] since it offers better load balancing across the network. The elemental cyclic distribution is a special case of the block cyclic distribution for the case where $b = 1$. Figure 7(D) shows the elemental cyclic layout for the 16 element vector on $p = 2$ processing nodes. Note that all even location elements are sent to the first processor, while all odd location elements are sent to the second processor. Similarly to the block cyclic case, a 1D DFT on elemental cyclic distributed data points requires one communication step between the p nodes. The elemental cyclic distribution is preserved if

$$p^2 \leq n. \tag{3.2}$$

Since $b = 1$, the number of nodes p on which a 1D DFT of size n can be parallelized is $\sqrt{n}$ nodes. Note that the 1D DFT on elemental cyclic distributed data points requires a single all-to-all communication, while being able to be scaled to the same number of processors as the blocked distribution case.

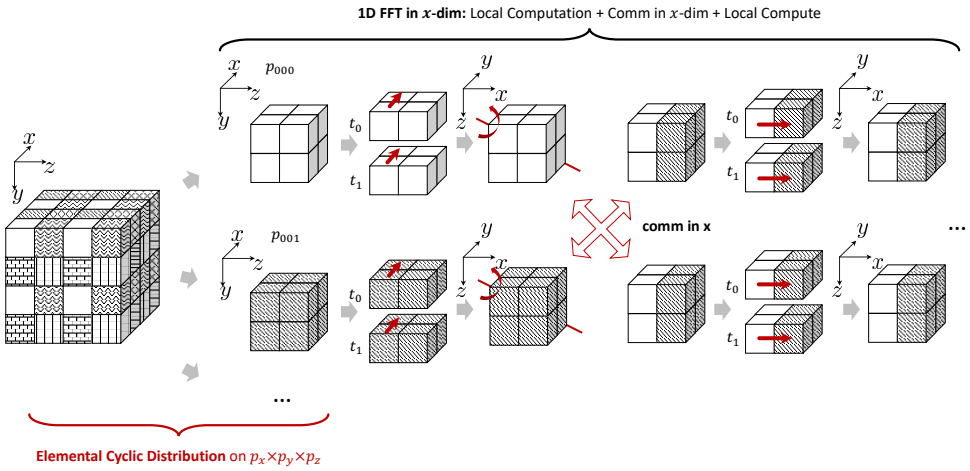

FIG. 8. *The 3D input $\hat{x}$ is elemental cyclic distributed on the 3D processing mesh. Each processor applies its local computation, rotates the data, and packs it for the the communication stage. Since the local data do not contain the entire data for a full 1D DFT 2, the local computation is partial. After the communication, data are unpacked, and each processor applies the remaining 1D DFT computation. The steps are repeated for the remaining 1D DFTs in the y and z dimensions.*

3.2. Elemental cyclic parallel 3D DFT. In this section, we describe the 3D DFT algorithm for a data cube $\hat{x}$ of size $n_0 \times n_1 \times n_2$. The cube is distributed elemental cyclically across the corresponding dimensions of a 3D mesh of $p_x \times p_y \times p_z$ compute nodes. For the 3D DFT computation, we impose that the input and the output have the same orientation and distribution, and that the processing grid remains invariant. As discussed in the previous subsection, these requirements are satisfied if the conditions $p_x^2 | n_0$, $p_y^2 | n_1$, and $p_z^2 | n_2$ hold.

Figure 8 shows how the initial cube $\hat{x}$ is distributed across the processor grid. Note that each dimension of the input cube is elemental cyclic distributed on the corresponding dimension of the grid. Let $\hat{x}^l$ be the local copy of the initial distribution. As outlined in section 2, $\hat{x}^l$ is represented as a matrix of matrices such that

$$\hat{x}^l = \left[\tilde{x}_0^l | \quad \tilde{x}_1^l | \quad \dots \quad | \quad \tilde{x}_{n_2/p_z}^l \right], \tag{3.3}$$

where each $\tilde{x}_i^l$ is a 2D matrix of size $(n_0/p_x) \times (n_1/p_y)$ for all values $0 \le i < (n_2/p_z)$. Given this distribution, the local computation cannot fully apply a 1D DFT. However, each processor can apply *partial* computations.

The 3D DFT decomposes into three 1D DFTs applied in the corresponding dimensions of the data cube. However, given (2.3) where the 1D DFT is decomposed into four stages that are applied on a 2D matrix, the 3D DFT can be viewed as an operation on a *six-dimensional* (6D) *tensor*. Each dimension of the data cube is interpreted as a 2D matrix. Since each dimension of the data cube is distributed elemental cyclic, each processor owns rows of the 2D matrix interpretation, as shown in section 3.1. As shown in Figure 8, each processor can apply the first DFT stage in the x dimension and the corresponding twiddle computation, if the twiddle factors are themselves distributed elemental cyclic in the same dimension. In addition to the two compute stages, there is a subsequent data rotation, where the current dimension on which the computation is applied is moved from the fastest dimension in memory

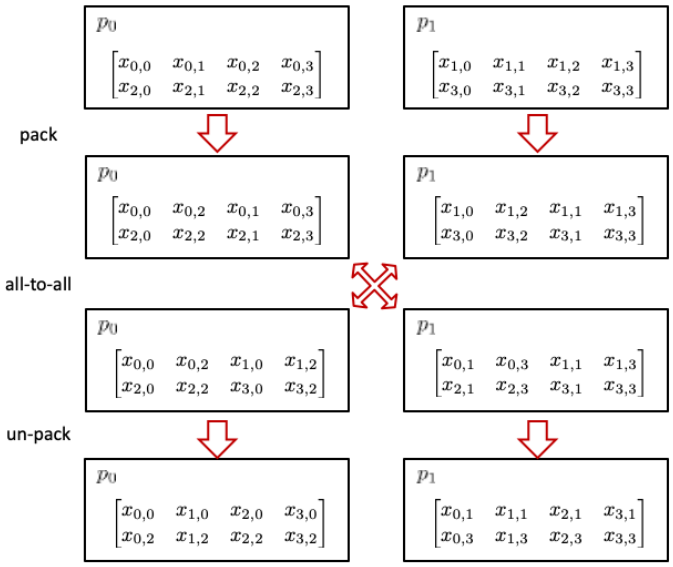

FIG. 9. *The transposition operation required by the* 1*D DFT computation in* (2.3) *is split into two local and one global transpositions. The local transpositions are implemented as packing/unpacking routines, while the global transposition is implemented as an all-to-all collective. Overall, the three steps form the communication step required by the* 1*D DFT.*

to the slowest dimension in memory. This data movement is required to improve bandwidth utilization for the subsequent communication stage.

As outlined in (2.3), the 1D DFT algorithm requires a transposition. The transposition must be applied in each dimension. Typically, the transposition is decomposed into multiple local transpositions and a global transposition, as shown in Figure 9. The local transpositions are implemented as packing routines that reshape the data into the send and receive buffers, respectively, while the global transposition is implemented as an all-to-all communication collective. To avoid packing routines with inefficient memory access patterns, where data are accessed elementwise at large strides, the previous compute stages rotate the local data set. For example, in 8 it can be seen that the x and y dimensions of the local data cube are moved into the slowest and fastest dimensions in memory, respectively. The subsequent packing and unpacking of the data cube in the x dimension is done at a larger granularity, that is more favorable to the memory bandwidth.

The remaining DFT computation in the x dimension can be applied. Note that the computation has to be applied in the slowest dimension in the memory, since the data have been rotated in the previous stage. The same steps are applied for the subsequent y and z dimensions. However, compute stages can be grouped together, to reduce roundtrips to local main memory and to improve data locality. Note that the last DFT stage of the 1D DFT in the x dimension can be merged with the first stages of the 1D DFT in the y dimension, as shown in Figure 10. The same grouping can be applied to the y and z computations. In addition, the cube rotation across the main diagonal can also be fused with the computation. The rotation preserves the data orientation, since this operation is applied three times. Given that the constraints are satisfied, the output also has the same elemental cyclic distribution as the input.

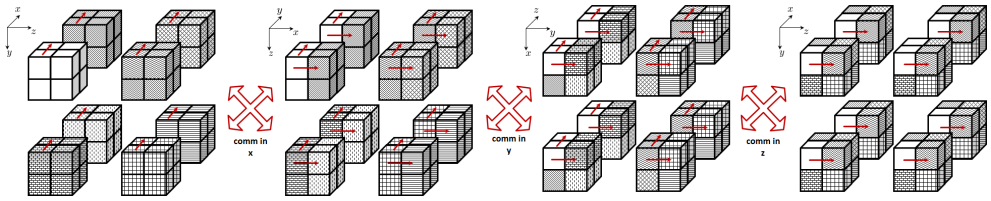

FIG. 10. *The parallel implementation of the 3D DFT using the volumetric decomposition. The data are distributed elemental cyclic in all three dimensions across a 3D mesh of* $2 \times 2 \times 2$ *compute nodes. The implementation requires three communication steps and the elemental cyclic distribution is preserved between the compute stages.*

4. Experimental results. In this section, we describe the experimental setup and the results obtained on both the RIKEN AICS K-Computer and on the Oak Ridge Summit Supercomputer, outlining the importance of having a flexible framework to choose the appropriate algorithm for a given problem size and number of nodes. We begin by presenting the characteristics of the two systems and then briefly outline some of the implementation details for parallelizing the computation using MPI and OpenMP. Last, we analyze strong scaling results of the 3D DFT for 64^3, 256^3, and 1024^3 using the three decompositions (slab-pencil, pencil-pencil-pencil, and volumetric). All implementations use the elemental cyclic distribution in the dimensions specified by the algorithm. We impose that the distribution and orientation of the input and output be the same. We give a breakdown of the performance and emphasize that there is not one decomposition that gives the best solution.

4.1. System configuration. The RIKEN AICS K-Computer consists of approximately $80,000$ SPARC64 VIIIfx nodes. Each compute node has a single CPU with 16 GB of main memory and a total bandwidth of 64 GB/s (main memory bandwidth). Each SPARC64 VIIIfx has eight cores with one thread per core running at 2.0 Ghz and 6 MB of L2 cache. Each core can do 128-bit single instruction multiple data (SIMD) instructions, giving a peak performance for double precision fused multiply-add instructions of 128 GFlops. Each compute node is connected by a 6D mesh/torus network (TOFU interconnect). The TOFU interconnect provides a logical 3D torus network for each job. The nodes have 10 links with a bandwidth of 10 GB/s full-duplex and the network allows for multiple pathways to communicate data. The infrastructure allows programs to be adapted to 1D, 2D, and 3D torus networks. The maximum number of nodes in one dimension is 54, while the maximum torus size is $48 \times 54 \times 32$ [1].

The Oak Ridge Summit Supercomputer consists of approximately 4608 physical nodes, each node having two IBM Power 9 processors and six NVIDIA Volta V100 GPUs. We provide information about the CPUs and skip the details of the GPUs since the current implementation only targets the CPU side of the system. Each compute node has two sockets, each socket with a local main memory of 256 GB with a bandwidth of 135 GB/s (main memory bandwidth). The two sockets are connected via an X-Bus that provides almost 64 GB/s between the two CPUs. Each socket has 21 cores, four threads per core, and a last level cache of approximately 120 MB that is distributed between pairs of cores. The cores can execute 128-bit SIMD instructions, similarly to the K-Computer. All the compute nodes are connected via a fat-tree topology, that provides approximately 25 GB/s full-duplex per link.

The current implementation of the DFT framework is built on top of the tensor framework, namely, the redistribution operations and tensor expressions (ROTE) framework [26]. ROTE provides a formal notation for describing the data layout of tensors that are distributed on multidimentional meshes. By describing the data layouts before and after a data redistribution operation, ROTE provides the infrastructure to map the necessary data movement into a sequence of one or more collective communication subroutines to achieve the desired data movement. The ideas are not tied to the ROTE framework and can easily be implemented in any tensor framework. While the data distribution and redistribution is handled by ROTE, we use FFTW for the local DFT computation. In the following section, we describe how we parallelize the computation within and across the compute nodes.

4.2. OpenMP + MPI parallelism. The framework uses OpenMP [17] parallelism for the local computation and MPI [27] for the distributed computation. The MPI is hidden away by the ROTE infrastructure. ROTE allows users to specify the grid, the tensors, the distribution, and the communication and, hence, writing the communication is straightforward. The construction of the compute node grid requires the `MPI_COMM_WORLD` global communicator and the size and shape of the grid. Based on the shape of the grid, the global communicator is split accordingly. The construction of the data tensors requires information about the shape, dimensions, and the grid on which the tensors are going to be distributed upon. Each tensor object provides the necessary functionalities to specify the communication, such as the all-to-all communication which is required by the 1D DFTs as described in Figure 9.

Parallelizing the computation using OpenMP is made explicit. We use `#pragma omp parallel region` to specify and create a pool of threads. Based on the thread id (`omp_get_thread_num`), each thread will compute on its own chunk of data as shown in Figure 8 and discussed in the previous section. We use the clause `#pragma omp barrier` to synchronize the threads and the clause `#pragma omp master` to specify that only the master thread can perform the all-to-all communication. The ROTE framework has been updated so that a pool of threads is created up front, without having to deal with nested thread creation. The benefit is that a single pool of threads is created up front at the beginning of the code, and that pool is used for both the computation and the packing operations. The K-Computer provides eight threads in total, and in the current implementation we use all of them. As for Summit, we use only one socket with 16 threads, each thread on its own core. We do not use the hyperthreads for the results presented in this paper.

4.3. Results and discussion. In the section, we focus on the 3D DFT results for cubic data sets of sizes 64^3, 256^3, and 1024^3, using all three algorithms, slab-pencil, pencil-pencil, and volumetric. For all of the configurations, we report strong scaling results, where we keep the problem size fixed and increase the number of compute nodes in each dimension as shown in Table 1. We focus on the power of two-sized grids of nodes, and we try to use all the allowable compute nodes. On the K-Computer we use 32K nodes, while on Summit we can scale up to 4K nodes. We run one MPI rank per physical node and use eight or 16 threads per node, depending on the system. For measurements, we use the `MPI_Wtime` function and measure the 3D DFT computation in steady state. We run the 3D DFT in a loop for approximately 10 minutes on the K-Computer and 5 minutes on Summit. We then take the average execution time for each experiment. We try to reduce network noise by forcing each MPI process to sleep for 30 to 60 seconds, and then to execute 10 to 20 dry runs that are not timed.

TABLE 1

Table showing the different configurations for the 1D, 2D, and 3D grids. For the RIKEN K-Computer [1] we choose all configurations, since the system has $48 \times 54 \times 32$ compute nodes on the 3D torus. On Summit, we execute up to 4k compute nodes, the maximum number of physical nodes available on the system. For the current experiments, we only allocate one MPI rank per physical compute node; we leave as future work the analysis for multiple ranks per node.

# of nodes	1D Grid	2D Grid	3D Grid
2	(2)	-	-
4	(4)	(2,2)	-
8	(8)	(4,2)	(2,2,2)
16	(16)	(4,4)	(4,2,2)
32	(32)	(8,2)	(4,4,2)
64	-	(8,8)	(4,4,4)
128	-	(16,8)	(8,4,4)
256	-	(16,16)	(8,8,4)
512	-	(32,16)	(8,8,8)
1k	-	(32,32)	(16,8,8)
2k	-	-	(16,16,8)
4k	-	-	(16,16,16)
8k	-	-	(32,16,16)
16k	-	-	(32,32,16)
32k	-	-	(32,32,32)

Figure 11 shows the strong scaling results for data cubes of sizes 64^3 and 1024^3. The results for the 256^3 data cube are depicted in the top two plots in Figure 13. The main theme is that there is not one algorithm that gives the shortest execution time. Choosing the slab-pencil algorithm over pencil-pencil and volumetric algorithms, depends on the problem size and the number of compute nodes. For example, for a problem size of 64^3, the slab-pencil algorithm seems to provide the best performance, compared to the other two algorithms. The disadvantage is that the execution time cannot be scaled to more than eight nodes, given the current implementation. Recall that we impose the output to have the same distribution and orientation as the input. Note that on Summit, the scaling of the 64^3 on more nodes can be done by changing the DFT algorithms, however, as we will discuss in the next sections, some issues related to computation and packing need further attention.

As the problem size increases, it is clearer that having multiple algorithms can provide advantages when scaling to larger number of nodes. For example, scaling the 256^3 3D DFT to 4k nodes requires all three algorithms. While, the slab pencil works for a small number of processors (2 to 16 nodes), and the pencil-pencil algorithm gives better results for a medium number of processors (32 to 128 nodes), the volumetric algorithm is needed to further scale the problem to 4k nodes. Note the same scaling behavior of the 3D DFT on the 1024^3 data cube on the K-Computer. On Summit, the same implementation gives different results. In Figure 11, the bottom-left plot shows that the volumetric algorithm provides the shortest execution time, irrespective of the number of nodes. As we will show in the next subsections, the network communication clearly shows different regimes where one algorithm is better than the other, however, we argue that computation and packing routines need to be further optimized, and we leave the optimization of the local computation as future work.

The execution of the parallel 3D DFT can be split into three parts as shown in Figure 8. Nodes communicate via the all-to-all collective to exchange data points. Each processor packs and unpacks the data before and after the all-to-all communication. Finally, each processor applies local 1D DFTs, pointwise scaling operations

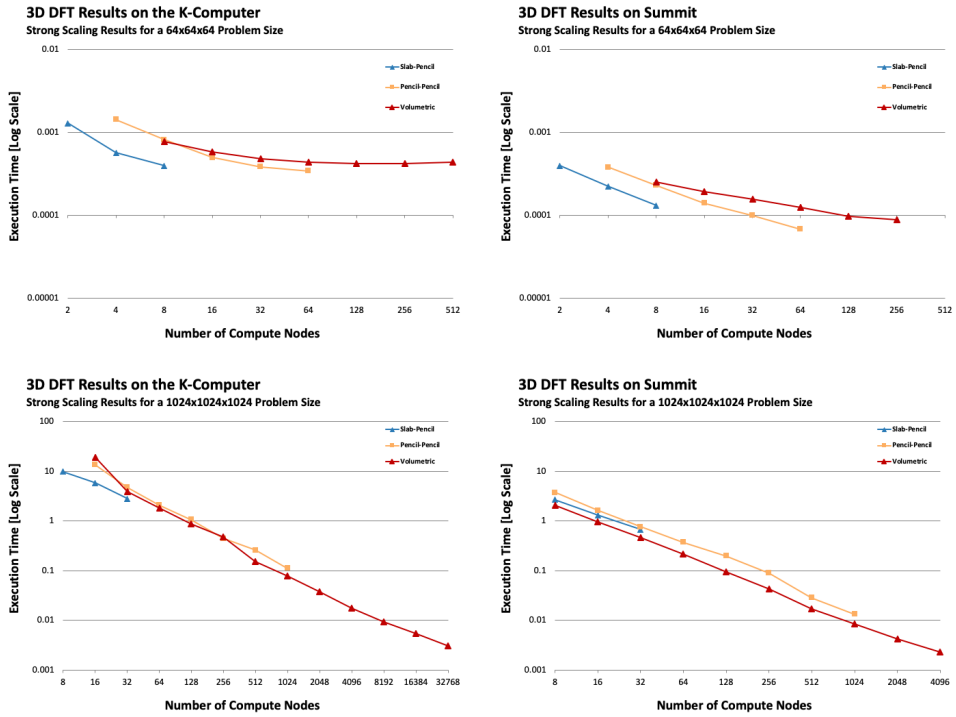

FIG. 11. *Strong scaling results for data cubes of sizes* 64^3 *(top plots) and* 1024^3 *(bottom plots) obtained on the K-Computer and on Summit. The plots show the strong scaling results for all three algorithms, assuming the data are distributed elemental cyclic across the corresponding* $1D$*,* $2D$*, and* $3D$ *grids of nodes, respectively.*

and rotations on their local data. In the next sections, we dive in analyzing each of these components in order to understand the execution times. First, we focus on the communication layer, outlining the crossing points between the all-to-alls, which will give a rough estimate of where to swap DFT algorithms. If computation and packing are efficiently done, then the 3D DFT should have similar behavior to the all-to-all communication. Second, we present a breakdown of the execution for the 256^3 problem size, analyzing the computation and packing. We emphasize that some components need further work.

4.3.1. MPI all-to-all. The 3D DFT algorithm requires one or more all-to-all communication steps. The slab-pencil algorithm requires one all-to-all, since one dimension of the data cube is distributed on a 1D mesh using the elemental cyclic layout. The pencil-pencil algorithm requires two all-to-all communication steps, because two dimensions of the data cube are elemental cyclic distributed on 2D mesh. Finally, the volumetric implementation requires three all-to-all communications, since all dimensions of the data cube are distributed elemental cyclic on a 3D mesh. The scaling behaviors of the three DFT algorithms are highly correlated with the scaling behaviors of the multiple all-to-alls.

In Figure 12, we show the execution times of only the network component for the DFT algorithms. We remove the local computation and packing and measure the one, two, and three all-to-alls. Each experiment assumes a distributed data cube on a 1D, 2D, and 3D mesh for each of the three problem sizes, 64^3, 256^3, and 1024^3. We

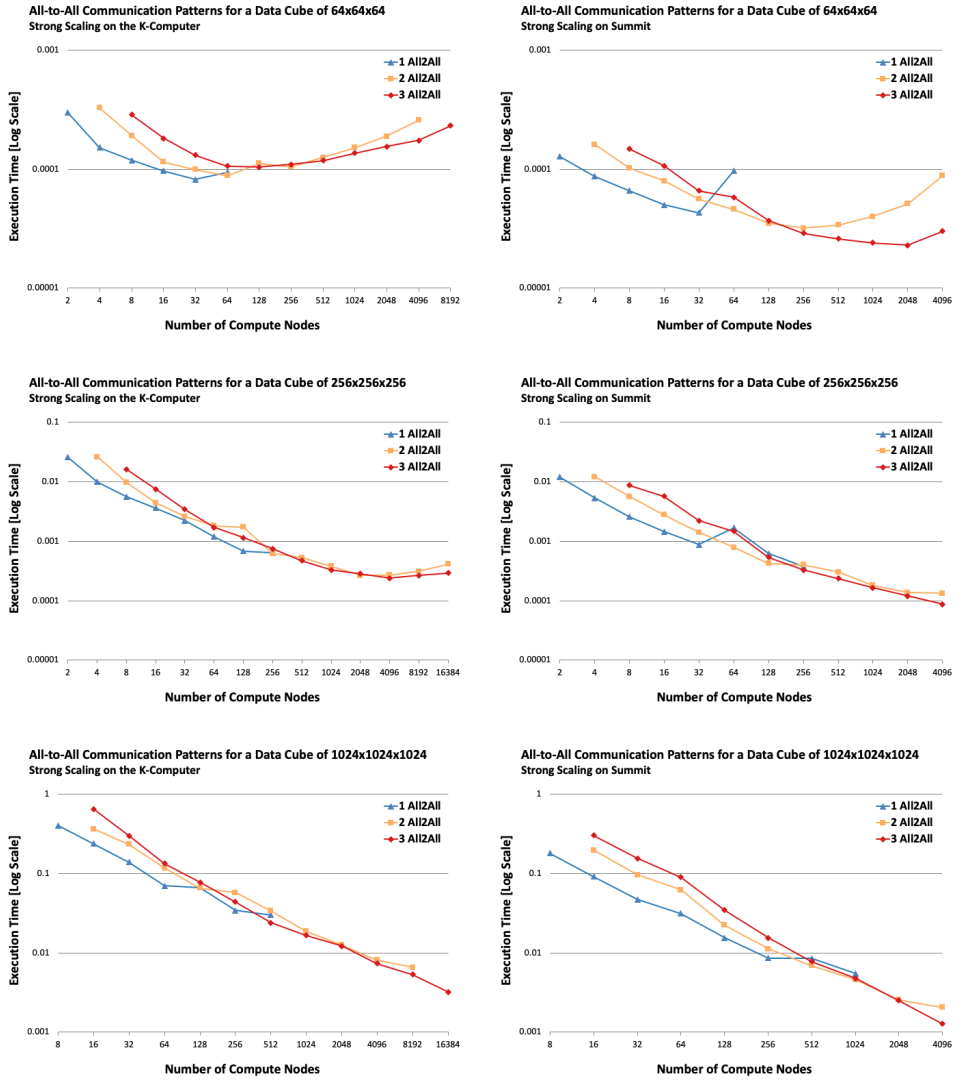

FIG. 12. *The execution of one, two, and three all-to-alls applied on 1D, 2D, and 3D grids of nodes, respectively. The number of all-to-alls depends on the 3D DFT algorithm, slab-pencil, pencil-pencil, and volumetric. The plots report the execution time when the number of nodes is increased for three problem sizes* 64^3, 256^3, *and* 1024^3.

measure the code in steady state, forcing a warm-up phase similar to measurements of the 3D DFT. For the all-to-all experiments we increase the number of nodes to be equal to the size of the cube's dimensions, in contrast to the DFT experiments where we keep the number nodes in each dimension bounded by the elemental cyclic constraint. For example, for a data cube of size 64^3 distributed on a 1D mesh, the number of nodes is increased to 64 nodes, rather than eight as in the DFT case.

Two aspects can be extracted from the results shown in Figure 12. First, all experiments show that for a fixed problem size, the execution of the all-to-alls decreases as the number of nodes is increased. However, beyond a certain number of nodes, the execution experiences slowdowns. For example, for the 64^3 data cube distributed

on a 1D mesh, the single all-to-all exhibits a slowdown when the number of nodes goes beyond 32 nodes on both the K-Computer and Summit. This may be due to the change in the all-to-all algorithms. Bruck et al. [2] have shown that there are two algorithms for the all-to-all, each having the following asymptotic bounds:

$$\lceil \frac{p-1}{k} \rceil \alpha + \lceil \frac{p-1}{k} \rceil \frac{n}{p^2} \beta, \tag{4.1}$$

$$\log_{k+1}(p)\alpha + \log_{k+1}(p) \frac{n}{(k+1)p} \beta, \tag{4.2}$$

where p represents the number of nodes, n represents the total amount of data across all p, and k represents the number of ports a compute node can send and receive. α represents the start-up cost and β represents the inverse of the bandwidth on each of the links. The first algorithm 4.1 minimizes data movement and is useful when the number of nodes is small and the amount of data is large. The second algorithm 4.2 minimizes latency and is useful when the number of nodes is large but the amount of data is small. For a given size n, one would use the first algorithm as long as

$$p^2 \geq \frac{n\beta}{\alpha}, \tag{4.3}$$

and after he/she would choose the second algorithm.

Second, there are crossing points, where the execution time of a single all-to-all is slower than two all-to-alls or where the execution time of two all-to-alls is slower than three all-to-alls, for the same problem size n and the same number of nodes p. This emphasizes that given a problem size n, it is better to do multiple smaller all-to-alls on fewer compute nodes than one single all-to-all between a large number of nodes. In addition, note that these crossing points appear when the all-to-all algorithms are presumed to be swapped, given the above description of the asymptotic bounds. A model can be derived to decide when to choose the algorithms and the number of all-to-alls. However, we leave that as future work, since the current framework uses the all-to-all collectives from the OpenMPI library, where each routine is treated as a black box.

4.3.2. Execution breakdown. The network communication is just one part of the DFT computation. The timing of the full DFT algorithm also requires the timing of the local computation and the timing of the packing routines, which may be a significant percentage of the overall execution. The time spent in the local operations depends on the local problem size, the implementation of the local computation, and the hardware characteristics of the compute nodes, such as number of threads, cache hierarchy, and bandwidth to main memory. We isolate the local computation, the packing routines, and the network communication by surrounding each section with `MPI_Wtime` function calls. We run the DFT algorithms in steady state and gather the averaged timings from each MPI rank. We report the slowest execution time for each section. In Figure 13, we provide results to show how much of the execution time is spent in computation, packing, or data movement over the network, for a data cube of size 256^3 using all three algorithms. Two important insights can be drawn from these results. First, as the number of nodes gets larger, the time spent in the local computation becomes insignificant. Second, as the number of nodes increases, a higher fraction of time is spent in the packing routines and network communication.

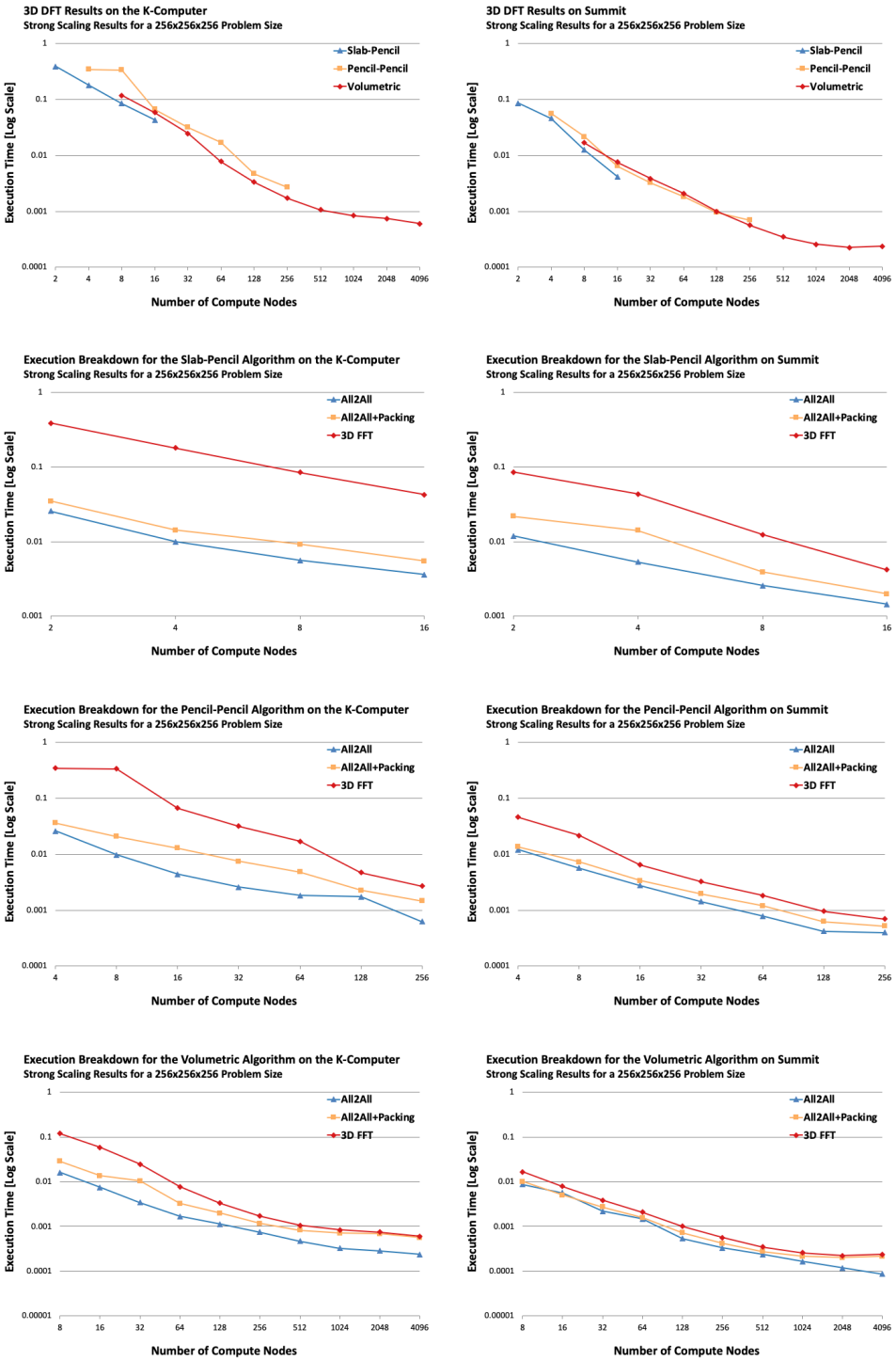

FIG. 13. *Strong scaling results for the 3D DFT of size* 256^3 *using the three algorithms and a breakdown of their execution. The bottom six plots show the time spent on the network (blue line), the time spent in packing (yellow line), and the time spent in compute (red line).*

As shown in Popovici, Franchetti, and Low [22], computation can be improved even for the small number of nodes. First, local DFT operations are implemented using FFTW's sequential Guru interface. While the interface provides the flexibility of applying small DFTs in different dimensions and rotating the data as the results are written back into memory, similarly to the rotations depicted in Figure 8, the performance is somewhat lacking. The FFTW Guru interface does not efficiently utilize the cache hierarchy, and it does not offer the capabilities to merge pointwise operations with the DFT calls. Lack of these capabilities imposes an additional pass through the data in order to perform the pointwise multiplication. Second, the DFT operations are well known to be memory latency bound problems [23], in the sense that computation stalls waiting for data from memory. However, using double buffering techniques, where some threads do computation and some threads prefetch data, can improve local computation by almost two times.

The packing routines can be implemented to make better use of the memory hierarchy. Data packing becomes more significant as indicated by the increased gab between the blue and orange lines, as outlined in the bottom two plots in Figure 13. One reason is that the packing routines are not cache friendly. Due to how the parallelization of the packing routines is done, threads will start conflicting on the cache-line length. The number of threads does not change as the number of compute nodes is increased and the amount of data is decreased. We leave the compute and the packing optimizations as future work. The optimizations are meant to bring the DFT execution time closer to that of the communication, which clearly emphasizes good scaling and the need for multiple algorithms for the same problem sizes.

5. Conclusion. In this paper, we presented a flexible framework for implementing parallel multidimentional DFTs on multidimentional processing grids. Specifically, we show that for different input problem sizes and different computational resource availability, different parallel multidimentional DFT algorithms are necessary for attaining efficient performance. In addition, we show that it is necessary to parallelize within the 1D DFT in order to scale the computation of the multidimentional DFT towards higher number of processing units. Despite incurring more rounds of communication, we show that for large enough data sizes, parallelizing with the 1D DFTs can improve the overall execution time over the conventional approach of simply parallelizing across multiple 1D DFTs.

While we showed improved performance as we scale to a larger number of nodes, further improvements to performance can be attained. As shown in Figure 13, a drawback of the current ROTE framework is that a local packing step is required to repack the data back into elemental-cyclic form after the collective communication. The time for the repacking is significant as it could be as much as 50% of the overall execution time. One possibility for reducing this packing time is to expose the packing performed by ROTE or allow ROTE to accept data in packed form. This is something we are currently exploring. We also believe that the ROTE notation for describing data layout and the processing grid can be extended to other architectures such as GPUs and multi-GPU systems. This can be achieved by replacing the MPI routines with the appropriate data movement primitives for GPUs and multi-GPUs. As the hierarchy of threads, thread blocks, streaming processors, and GPUs can be described as a multidimentional array, this suggests that ROTE can be used as the notation for porting the existing code to GPUs and multi-GPU systems.

Acknowledgments. The authors would like to thank Dr. Imamura Toshiyuki and the Advanced Institute for Computational Science at RIKEN for granting access

to the K-computer. The content, views and conclusions presented in this document are those of the author(s) and do not necessarily reflect the position or the policy of the sponsoring agencies.

REFERENCES

[1] *Riken AICS*. https://www.r-ccs.riken.jp/en/.

[2] J. Bruck, C.-T. Ho, S. Kipnis, E. Upfal, and D. Weathersby, *Efficient algorithms for all-to-all communications in multiport message-passing systems*, IEEE Trans. Parallel Distrib. Systems, 8 (1997), pp. 1143–1156.

[3] A. Canning, L. Wang, A. Williamson, and A. Zunger, *Parallel empirical pseudopotential electronic structure calculations for million atom systems*, J. Comput. Phys., 160 (2000), pp. 29–41.

[4] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker, *ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers*, in Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation, IEEE Computer Society, Los Alamitos, CA, 1992, pp. 120–127.

[5] J. W. Cooley and J. W. Tukey, *An algorithm for the machine calculation of complex fourier series*, Math. Comp., 19 (1965), pp. 297–301.

[6] I. T. Foster and P. H. Worley, *Parallel algorithms for the spectral transform method*, SIAM J. Sci. Comput., 18 (1997), pp. 806–837.

[7] F. Franchetti, Y. Voronenko, and M. Püschel, *FFT program generation for shared memory: SMP and multicore*, in Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, ACM, New York, 2006, pp. 115–es.

[8] M. Frigo and S. G. Johnson, *The design and implementation of FFTW*3, Proc. IEEE, 93 (2005), pp. 216–231.

[9] M. Frigo and S. G. Johnson, *FFTW: Fastest fourier transform in the west*, Astrophysics Source Code Library (2012).

[10] M. A. Inda and R. H. Bisseling, *A simple and efficient parallel FFT algorithm using the BSP model*, Parallel Comput., 27 (2001), pp. 1847–1878.

[11] J. Johnson and X. Xu, *A recursive implementation of the dimensionless FFT*, in Acoustics, Speech, and Signal Processing, 2003 Proceedings, ICASSP'03, Vol. 2, IEEE, Piscataway, NJ, 2003, pp. 649–652.

[12] J. Jung, C. Kobayashi, T. Imamura, and Y. Sugita, *Parallel implementation of* 3*d FFT with volumetric decomposition schemes for efficient molecular dynamics simulations*, Comput. Phys. Commun., 200 (2016), pp. 57–65.

[13] R. A. Kendall, E. Aprà, D. E. Bernholdt, E. J. Bylaska, M. Dupuis, G. I. Fann, R. J. Harrison, J. Ju, J. A. Nichols, J. Nieplocha, T. P. Straatsma, T. L. Windus, and A. T. Wong, *High performance computational chemistry: An overview of NWChem a distributed parallel application*, Comput. Phys. Commun., 128 (2000), pp. 260–283.

[14] R. A. Lebensohn, *N-site modeling of a 3d viscoplastic polycrystal using fast Fourier transform*, Acta Mater., 49 (2001), pp. 2723–2737.

[15] R. A. Lebensohn, A. K. Kanjarla, and P. Eisenlohr, *An elasto-viscoplastic formulation based on fast Fourier transforms for the prediction of micromechanical fields in polycrystalline materials*, Int. J. Plast., 32 (2012), pp. 59–69.

[16] S.-B. Lee, R. Lebensohn, and A. D. Rollett, *Modeling the viscoplastic micromechanical response of two-phase materials using fast Fourier transforms*, Int. J. Plast., 27 (2011), pp. 707–727.

[17] OpenMP Architecture Review Board, *OpenMP Application Program Interface Version* 4.0, May 2018, https://www.openmp.org/.

[18] D. Pekurovsky, *P*3*DFFT: A Framework for Parallel Computations of Fourier Transforms in Three Dimensions*, SIAM J. Sci. Comput., 34 (2012), pp. C192–C209, https://doi.org/10.1137/11082748X.

[19] S. Plimpton, R. Pollock, and M. Stevens, *Particle-mesh Ewald and RRESPA for parallel molecular dynamics simulations*, in Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing, SIAM, Philadelphia, 1997.

[20] S. J. Plimpton, *FFTs for (mostly) particle codes within the DOE exascale computing project*, Technical report, SAND2017-127D1PE, Sandia National Laboratory, Albuquerque, NM, 2017.

[21] S. J. Plimpton and A. P. Thompson, *Computational aspects of many-body potentials*, MRS Bull., 37 (2012), pp. 513–521.

[22] D. T. Popovici, F. Franchetti, and T. M. Low, *Mixed data layout kernels for vectorized complex arithmetic*, in 2017 IEEE High Performance Extreme Computing Conference, HPEC 2017, Waltham, MA, 2017, IEEE, Piscataway, NJ, 2017, pp. 1–7.

[23] D. T. Popovici, T.-M. Low, and F. Franchetti, *Large bandwidth-efficient FFTs on multicore and multi-socket systems*, in IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, Piscataway, NJ, 2018.

[24] J. Poulson, B. Marker, R. A. Van de Geijn, J. R. Hammond, and N. A. Romero, *Elemental: A new framework for distributed memory dense matrix computations*, ACM Trans. Math. Software, 39 (2013), 13.

[25] B. Prisacari, G. Rodriguez, C. Minkenberg, and T. Hoefler, *Bandwidth-optimal all-to-all exchanges in fat tree networks*, in Proceedings of the 27th International ACM Conference on Supercomputing, ACM, New York, 2013, pp. 139–148.

[26] M. D. Schatz, *Distributed Tensor Computations: Formalizing Distributions, Redistributions, and Algorithm Derivation*, PhD thesis, University of Texas at Austin, Austin, TX, 2015.

[27] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra, *MPI: The Complete Reference*, The MIT Press, Cambridge, MA, 1996.

[28] T. Straatsma, E. Bylaska, H. van Dam, N. Govind, W. de Jong, K. Kowalski, and M. Valiev, *Advances in scalable computational chemistry: NWChem*, in Annual Reports in Computational Chemistry, Vol. 7, Elsevier, Amsterdam, 2011, pp. 151–177.

[29] P. N. Swarztrauber, *Multiprocessor FFTs*, Parallel Comput., 5 (1987), pp. 197–210.

[30] R. A. Sweet, W. L. Briggs, S. Oliveira, J. L. Porsche, and T. Turnbull, *FFTs and three-dimensional Poisson solvers for hypercubes*, Parallel Comput., 17 (1991), pp. 121–131.

[31] D. Takahashi, *An implementation of parallel 3-D FFT with 2-D decomposition on a massively parallel cluster of multi-core processors*, in Parallel Processing and Applied Mathematics, 8th International Conference, PPAM 2009, Wroclaw, Poland, Revised Selected Papers, Part I, Czestoschowa University of Technology, Czestoschowa, Poland, 2009, pp. 606–614.

[32] M. Valiev, E. J. Bylaska, N. Govind, K. Kowalski, T. P. Straatsma, H. J. Van Dam, D. Wang, J. Nieplocha, E. Apra, T. L. Windus, and W. A. de Jong, *NWChem: a comprehensive and scalable open-source solution for large scale molecular simulations*, Comput. Phys. Commun., 181 (2010), pp. 1477–1489.

[33] C. Van Loan, *Computational Frameworks for the Fast Fourier Transform*, SIAM, Philadelphia, 1992.

[34] J.-L. Vay, A. Almgren, J. Bell, L. Ge, D. Grote, M. Hogan, O. Kononenko, R. Lehe, A. Myers, C. Ng, J. Park, R. Ryne, O. Shapoval, M. Thévenet, and W. Zhang, *Warp-X: A new exascale computing platform for beam–plasma simulations*, Nucl. Instrum. Methods Phys. Res. Sect. A, 909 (2018), pp. 476–479.

[35] J.-L. Vay, I. Haber, and B. B. Godfrey, *A domain decomposition method for pseudo-spectral electromagnetic simulations of plasmas*, J. Comput. Phys., 243 (2013), pp. 260–268.