Technical Report 2013-001

3D Discrete Transforms with Cubical Data Decomposition on the IBM Blue Gene/Q

Tomoya Sakai and Stanislav G. Sedukhin

February 27, 2013

Graduate School of Computer Science and Engineering
The University of Aizu
Tsuruga, Ikki-Machi, Aizu-Wakamatsu City
Fukushima, 965-8580 Japan

Technical Report 2013-001

Title:
3D Discrete Transforms with Cubical Data Decomposition on the IBM Blue Gene/Q

Authors:
Tomoya Sakai, Stanislav G. Sedukhin

Key Words and Phrases:
3-dimensional discrete transform, 3-dimensional discrete Fourier transform, computational index space, cubical data decomposition, torus interconnect, extremely scalable algorithm, IBM Blue Gene/Q

Abstract:
This report presents the implementation and performance evaluation of the three dimensional (3D) discrete transforms with cubical data decomposition. We implemented newly proposed GEMM-based algorithm with the 3D data decomposition which can be extremely scaled up to N^3 computer nodes. In this algorithm, only local communications between nearest neighbor nodes are required and overlapping of computation and communication is possible. The algorithm implementation and some optimization techniques for overlapping of computation and communication are presented. As an exmample of the algorithm with 3D data decomposition, we implemented and evaluated the performance of the 3D Discrete Fourier transform in complex-double precision on the IBM BG/Q which has 5-dimensional torus interconnection. Comparison between our results with 3D data decomposition and 3D FFT with 2D data decomposition is included for number of nodes less than N^2.

Report Date:
2/27/2013

Written Language:
English

Any Other Identifying Information of this Report:

Distribution Statement:
First Issue: 10 copies

Supplementary Notes:

Distributed Parallel Processing Laboratory
The University of Aizu
Aizu-Wakamatsu
Fukushima 965-8580
Japan

3D Discrete Transforms with Cubical Data Decomposition on the IBM Blue Gene/Q

Tomoya Sakai and Stanislav G. Sedukhin

Abstract

This report presents the implementation and performance evaluation of the three dimensional (3D) discrete transforms with cubical data decomposition. We implemented newly proposed GEMM-based algorithm with the 3D data decomposition which can be extremely scaled up to N^3 computer nodes. In this algorithm, only local communications between nearest neighbor nodes are required and overlapping of computation and communication is possible. The algorithm implementation and some optimization techniques for overlapping of computation and communication are presented. As an exmample of the algorithm with 3D data decomposition, we implemented and evaluated the performance of the 3D Discrete Fourier transform in complex-double precision on the IBM BG/Q which has 5-dimensional torus interconnection. Comparison between our results with 3D data decomposition and 3D FFT with 2D data decomposition is included for number of nodes less than N^2.

1 Introduction

Three-dimensional (3D) discrete transforms (DT) such as Fourier transform, cosine/sine transform, Hartley transform, Walsh-Hadamard transform, etc., are known to play a fundamental role in many application areas such as spectral analysis, digital filtering, signal and image processing, data compression, medical diagnostics, etc. Increasing demands for high speed in many real-world applications have simulated the development of a number of Fast Transform (FT) algorithms, such as Fast Fourier Transform (FFT), with dramatic reduction of arithmetic complexity [7]. However, further reduction of time complexity is only possible by overlapping these arithmetic operations, i.e. using parallel implementation.

There exists three different approaches for parallel implementation of the 3D discrete transforms. Two of them are especially for Fourier transform.

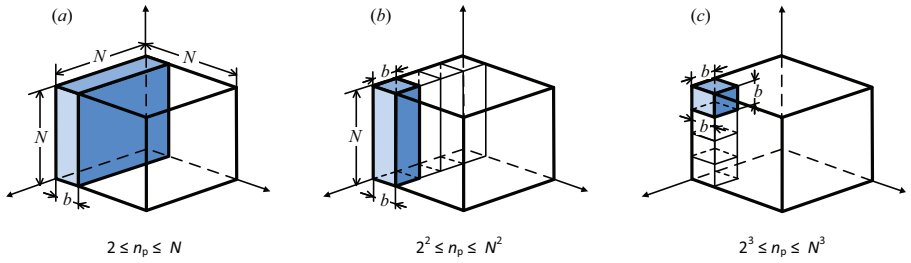


Figure 1: 3D data distribution over n_p computer nodes: (a) 1D or "slab" decomposition, (b) 2D or "pencil" decomposition, and (c) 3D or "cube" decomposition.

The first one is the 1D or "slab" decomposition of a 3D initial data. In this approach, $N \times N \times N$ data is divided into 2D slabs of size $N \times N \times b$, where $b = N/P$ and P is the number of computer nodes, and assigned to each node (see Fig. 1 (a)). This approach allows scaling problem among $n_p = P = N/b$ nodes, where b is the blocking factor. The scalability of the slab-based approach or the maximum number of nodes is limited by the number of data elements along a single dimension of the 3D transform, i.e. $n_p^{max} = N$ when $b = 1$. Different implementation of the 3D FFT with a "slab" decomposition can be found in [10, 11].

The second approach is the 2D or "pencil" decomposition of a 3D $N \times N \times N$ initial data among a 2D array of $n_p = P \times P$ computer nodes ($P = n/b$). Initial $N \times N \times N$ cube is divided into an 1D "pencil" of size $N \times b \times b$ and is assigned to each node. This approach increases the maximum number of nodes from N to N^2. Parallel 3D FFT implementation with a 2D data decomposition is discussed in [9, 17, 22, 2].

In both of these so-called "transposed" approaches, the computational part and inter-node communication part are separated [4]. Moreover, a computational part inside each node is implemented by using either 2D or 1D fast (recursive) algorithm for "slab"- or "pencil"- based decomposition, respectively, without any inter-node communication. However, on completion of each computational part, in order to support contiguity of memory accesses, a transposition of the 3D data array is required to put data in an appropriate dimension(s) into each node. At least one or two transpositions would be needed for the 1D or 2D data decomposition approaches, respectively. Each of this 3D data transposition is implemented by global "all-to-all" inter-node, message-passing communication.

The last approach is the 3D or "cube" decomposition, which is recently proposed in [23]. The 3D or "cubic" decomposition of an $N \times N \times N$ initial data among $n_p = P \times P \times P$ computer nodes where a 3D data "cube" of size $b \times b \times b$ is assigned to each node (see Fig. 1 (c)). The scalability is further improved from

N^2 to N^3. In this approach, blocked matrix multiplication based algorithms are used to compute the basic one-dimensional N-size transform not on a single but on the $P = N/b$ cyclically interconnected (for data reuse) nodes of a 3D torus network. This approach is an integration of a local intra-node computation with a nearest-neighbor inter-node communication at each step of three-dimensional processing.

The proposed [23] algorithm with 3D data decomposition eliminates global communication. In addition, computation and local communication can be overlapped. The 3D transform is represented as three chained sets of the cubical tensor-by-matrix or matrix-by-tensor multiplications which are executed in a 3D torus network of computer nodes by the fastest and extremely scalable orbital algorithms.

Our main objective of this report is to implement and evaluate the 3D discrete transform with 3D data decomposition. For the 3D data decomposition, we propose the method for overlapping of computation and communication. As an example of implementation, the performance of the 3D Discrete Fourier Transform (DFT) with 3D data decomposition is measured on the IBM Blue Gene/Q. We also compare our result of the 3D DFT with 3D data decomposition with the result of 3D FFT with 2D data decomposition which is presented in [12].

The report is organized as follows. In Section 2, the scalar and block notations of the 3D forward and inverse separable transforms are described. In Section 3, we introduce algorithm with the 3D decomposition and psuedcode for its implementation. In Section 4, we show the implementation detiails and some optimization techniques. In Section 5, we discuss the performance result of the 3D DFT in complex-double precision on the Blue Gene/Q. In Section 6, we will conclude the report with conclusions and future works.

2 3D Separable Transform

Let $X = [x(n_1, n_2, n_3)], 0 \leq n_1, n_2, n_3 < N$, be an $N \times N \times N$ cubical grid of input data or three-way data tensor [15]. A separable forward 3D transform of X is another cubical grid of an $N \times N \times N$ data or three-way tensor $\dddot{X} = [\dddot{x}(k_1, k_2, k_3)]$ where for all $0 \leq k_1, k_2, k_3 < N$:

$$\dddot{x}(k_1, k_2, k_3) = \sum_{n_3=0}^{N-1} \sum_{n_2=0}^{N-1} \sum_{n_1=0}^{N-1} x(n_1, n_2, n_3) \cdot c(n_1, k_1) \cdot c(n_2, k_2) \cdot c(n_3, k_3) \quad (1)$$

In turn, a separable inverse or backward 3D transform of three-way tensor

$\dddot{X} = [\dddot{x}(k_1, k_2, k_3)]$ is expressed as:

$$x(n_1, n_2, n_3) = \sum_{k_3=0}^{N-1} \sum_{k_2=0}^{N-1} \sum_{k_1=0}^{N-1} \dddot{x}(k_1, k_2, k_3) \cdot c(n_1, k_1) \cdot c(n_2, k_2) \cdot c(n_3, k_3) \quad (2)$$

where $0 \leq n_1, n_2, n_3 < N$ and $X = [x(n_1, n_2, n_3)]$ is an output $N \times N \times N$ cubical tensor.

There is a direct correspondence between the equation (1) or (2) and the so-called *multilinear matrix multiplication*, which is used to represents three-way data tensor X or $\dddot{X}$ in different bases and where an $N \times N$ matrix $C = [c(n_s, k_s)] = [c(n, k)], s = 1, 2, 3$ is a (non-singular) change-of-basis matrix (see [15, 18, 14] for more details). It is also interesting to mention that equations (1) and (2) can viewed as the so called Tuker's 3D tensor decomposition which is represented in the form of three-way tensor-by-matrix multiplication [15, 16]. Moreover, the equation (1) or (2) describes the so-called three-way *tensor contraction* which is widely used in $abinitio$ quantum chemistry models [3, 19, 25].

The various separable transforms differ only by the transform coefficient (change-of-basis) matrix $C = [c(n, k)]$ which can be

- *symmetric*, i.e. $C = C^T$, and *unitary*, i.e. $C^{-1} = C^{*T}$, C^* is a complex conjugate of C, like in the Discrete Fourier Transform (DFT), where $c(n, k) = \exp[-\frac{2\pi i}{N}(n \cdot k)] = \cos(\frac{2\pi nk}{N}) - i \sin(\frac{2\pi nk}{N})$ and $i = \sqrt{-1}$, or in the Discrete Hartley Transform (DHT), where $c(n, k) = \cos(\frac{2\pi nk}{N}) - i \sin(\frac{2\pi nk}{N})$;

- unitary and *real*, i.e. *orthogonal*, like in the Discrete Cosine Transform (DCT), where coefficient $c(n, k) = cos[\frac{\pi}{2N}(2n + 1) \cdot k]$ and $C \neq C^T$;

- consists only $\pm$ 1 and be symmetric and orthogonal, like in the Discrete Walsh-Hadamard Transform (DWHT).

We will abbreviate the genetic form of a discrete transform as DXT without taking into account the specific features of a coefficient matrix, i.e. we will use direct algorithm for the 3D transform (1) and (2) with an arithmetic complexity of $O(N^4)$ instead of using the so-called "fast" DXT algorithms, like 3D FFT, with $O(N^3 \log N)$ complexity.

To formulate the problem in the block notation we firstly divide an $N \times N \times N$ input data volume or three-way tensor $X = [x(n_1, n_2, n_3)]$ into $P \times P \times P$ data cubes, where each cube $X(N_1, N_2, N_3), 0 \leq N_1, N_2, N_3 < P$, has the size of $b \times b \times b$, i.e. $b = N/P$ and $1 \leq b \leq N/2$ is the blocking factor. Then the forward 3D DXT (3D FDXT) can be expressed as a block version of the multilinear matrix multiplication:

$$\dddot{X}(K_1, K_2, K_3) = \sum_{N_3=0}^{P-1} \sum_{N_2=0}^{P-1} \sum_{N_1=0}^{P-1} X(N_1, N_2, N_3) \times C(N_1, K_1) \times C(N_2, K_2) \times C(N_3, K_3), \quad (3)$$

where $0 \leq K_1, K_2, K_3 < P$ and $C(N_s, K_s) = C(N, K), s = 1, 2, 3$, is an (N_s, K_s)-th block of the transform matrix C.

It is clear that a 3D block inverse transform (3D IDXT) can be written as:

$$X(N_1, N_2, N_3) = \sum_{K_3=0}^{P-1} \sum_{K_2=0}^{P-1} \sum_{K_1=0}^{P-1} \dddot{X}(K_1, K_2, K_3) \times C(N_1, K_1) \times C(N_2, K_2) \times C(N_3, K_3), \quad (4)$$

where $0 \leq N_1, N_2, N_3 < P$.

Due to separability of the linear transforms, a 3D transform can be splitted into three data dependent sets of 1D transform as it is shown below for the 3D FDXT (3).

At the *first stage*, the $P \times P$ 1D FDXT of $X(N_1, N_2, :)$ are performed for all (N_1, N_2) pairs, $0 \leq N_1, N_2 < P$, as block cubical *tensor-by-matrix multiplication*:

$$\dot{X}(N_1, N_2, K_3) = \sum_{N_3=0}^{P-1} X(N_1, N_2, N_3) \times C(N_3, K_3), \quad (5)$$

where $0 \leq N_1, N_2, K_3 < N$.

At the *second stage*, the $P \times P$ 1D FDXT of $\dot{X}(:, N_2, K_3)$ are implemented for all (N_2, K_3) pairs, $0 \leq N_2, K_3 < P$, as second block tensor-by-matrix multiplication:

$$\ddot{X}(K_1, N_2, K_3) = \sum_{N_1=0}^{P-1} \dot{X}(N_1, N_2, K_3) \times C(N_1, K_1), \quad (6)$$

where $0 \leq K_1, N_2, K_3 < N$.

At the *third stage*, the $P \times P$ 1D FDXT of $\ddot{X}(K_1, :, K_3)$ are implemented for all (K_1, K_3) pairs, $0 \leq K_1, K_3 < P$, as third block tensor-by-matrix multiplication:

$$\dddot{X}(K_1, K_2, K_3) = \sum_{N_2=0}^{P-1} \ddot{X}(K_1, N_2, K_3) \times C(N_2, K_2), \quad (7)$$

where $0 \leq K_1, K_2, K_3 < N$.

By slicing cubical data, i.e. representing three-way tensors as the set of matrices, it is possible to formulate a 3D transform (3) or (4) as conventional block

matrix-by-matrix multiplication. In this case, an initial $P \times P \times P$ data grid or three-way tensor $\{X(N_1, N_2, N_3), 0 \leq N_1, N_2, N_3 < P\}$, is divided into P 1D "slice" or "slabs" or "matrices-of-cubes" along one of axises, for example, along N_2-axis, such that each $b \times b \times b$ data cube $X(N_1, N_2, N_3)$ can be referred as a block element $X(N_1, N_3)_{N_2}$ of the N_2-th $P \times P$ matrix, where $N_2 \in [0, P)$. Then a 3D FDXT (3) can also be computed in three data-dependent stages as chaining sets of the block matrix-by-matrix products.

At the first stage, since there is no dependency in 1D transforming data between all N_2-slabs, the cubical tensor-by-matrix multiplication (5) can be presented as P, independent on N_2, block matrix-by-matrix multiplications:

$$\dot{X}(N_1, K_3)_{N2} = \sum_{N_3=0}^{P-1} X(N_1, N_3)_{N_2} \times C(N_3, K_3), \tag{8}$$

where the same change-of-basis matrix $C = [C(N_3, K_3)]$, $0 \leq N_3, K_3 < P$, is used for all N_2-slices, $N_2 \in [0, P)$.

At the second stage, the original tensor-by-matrix product (6) is computed by the set of P, also independent on N_2, block matrix-by-matrix multiplications in the following form which keeps required row-by-column index agreement:

$$\ddot{X}(K_1, K_3)_{N_2} = \sum_{N_1=0}^{P-1} C(N_1, K_1)^T \times \dot{X}(N_1, K_3)_{N_2}. \tag{9}$$

It can be seen from (9) that the same coefficient matrix $C = [C(N_1, K_1)]$, $0 \leq N_1, K_1 < P$, is used for all N_2 slices, $N_2 \in [0, P)$.

At the third stage, the final tensor-by-matrix product (3) is implemented as the set of P, independent on K_3, block matrix-by-matrix multiplications:

$$\dddot{X}(K_1, K_2)_{K_3} = \sum_{N_2=0}^{P-1} \ddot{X}(K_1, N_2)_{K_3} \times C(N_2, K_2), \tag{10}$$

where the same $P \times P$ matrix $C = [C(N_2, K_2)]$, $0 \leq N_2, K_2 < P$, is used for all K_3-slices, $K_3 \in [0, P)$.

It is clear that an inverse 3D transform (IDXT) (4) is implemented in the reverse order, i.e. as rolling back of a forward 3D DXT. By keeping the slicing of an initial cubical $P \times P \times P$ tensor $\dddot{X}(K_1, K_2, K_3)$ along K_3-axis, a 3D IDXT would require implementation of the following three stages:

Stage I: **for all** pairs (K_1, K_2) **at** slabs $K_3 \in [0, P)$ **do**

$$\ddot{X}(K_1, N_2)_{K_3} = \sum_{K_2=0}^{P-1} \ddot{\ddot{X}}(K_1, K_2)_{K_3} \times C(N_2, K_2)^T, 0 \leq K_1, N_2 < P \qquad (11)$$

After completion, a resulting in this stage cubical tensor $\ddot{X}(K_1, N_2, K_3)$ is used as sliced into 1D slabs along N_2 axis.

Stage II: **for all** pairs (N_1, K_3) **at** slabs $N_2 \in [0, P)$ **do**

$$\dot{X}(N_1, K_2)_{N_2} = \sum_{K_1=0}^{P-1} C(N_1, K_1) \times \ddot{X}(K_1, K_3)_{N_2}, 0 \leq N_1, K_3 < P. \qquad (12)$$

Stage III: **for all** pairs (N_1, N_3) **at** slabs $N_2 \in [0, P)$ **do**

$$X(N_1, N_3)_{N_2} = \sum_{K_3=0}^{P-1} \dot{X}(N_1, K_3)_{N_2} \times C(N_3, K_3)^T, 0 \leq N_1, N_3 < P. \qquad (13)$$

It is easy to verify that the total number of $P \times P$ block matrix-by-matrix multiplications in a forward or inverse 3D DXT is $3P^4$. Each block matrix-by-matrix multiplication is a $b \times b \times b$ tensor-by-matrix product which requires an execution of b^4 scalar fused multiply-add (`fma`) operations, where $b = N/P$ is a blocking factor. The total number of such scalar `fma`-operations for a 3D DXT is, therefore, $3P^4 \cdot b^4 = 3N^4$, i.e. blocking does not change an arithmetic complexity of the transformation. Obviously, this is because our 3D DXT implementation is totally based on a matrix-by-matrix multiplication.

3 3D Data Decomposition Algorithm

3.1 Algorithm Description

At the beginning, each computer node CN(Q,R,S) in a $P \times P \times P$ torus array holds in a local memory the four $b \times b \times b$ data cubes:

- $X = X(Q, R, S) = X(N_1, N_2, N_3)$,
- $\dot{X} = \dot{X}(Q, R, \tau) = \dot{X}(N_1, N_2, \tau) = \mathbf{0}$,
- $\ddot{X} = \ddot{X}(S, R, \tau) = \ddot{X}(N_3, N_2, \tau) = \mathbf{0}$,
- $\dddot{X} = \dddot{X}(S, Q, \tau) = \dddot{X}(N_3, N_1, \tau) = \mathbf{0}$,

as well as the three $b \times b$ change-of-basis matrices of transform coefficients:

- $C_{\rm I} = C(S, \tau)$, $C_{\rm II} = C(Q, S)$, and $C_{\rm III} = C(R, Q)$,

where $\mathbf{0}$ is a cubical $b \times b \times b$ tensor with all its entries being zero.

Each computer node CN(Q,R,S) has six bi-directional links labeled as $\pm Q$, $\pm R$ and $\pm S$. During processing some blocks of tensor and matrix data are rolled, i.e. cyclically shifted, along (+) or opposite (-) axis (orbit) according to the scheduling vector α (see [24] for more details). As it can be seen from the assignment above, the $P \times P$ matrices $C_{\rm I}$ and $C_{\rm II}$ are replicated among P parallel along R-axis slabs of computer nodes while $P \times P$ matrix $C_{\rm III}$ is duplicated among P parallel along S-axis slabs.

A three-stage orbital implementation of the 3D forward transform in a 3D network of toroidally interconnected nodes $\{\text{CN}(Q, R, S) : 0 \leq Q, R, S < P\}$ under the scheduling function $\tau = (Q + R + S) \bmod P$, i.e. $\alpha = (\alpha_Q, \alpha_R, \alpha_S) = (+1, +1, +1)^T$, is described below.

Stage I. $\dot{X}(Q, R, \tau) = \sum_{0 \leq S < P} X(Q, R, S) \times C(S, \tau)$:

- **for all** P^3 CN(Q,R,S) **do** P times:
  1. **compute**: $\dot{X} \leftarrow X \times C_I + \dot{X}$
  2. **data roll**: $\overset{+S}{\Longleftarrow} \dot{X} \overset{-S}{\Longleftarrow} \; || \; \overset{+Q}{\Longleftarrow} C_I \overset{-Q}{\Longleftarrow}$

Stage II. $\ddot{X}(S, R, \tau) = \sum_{0 \leq Q < P} C(Q, S)^T \times \dot{X}(Q, R, \tau)$:

- **for all** P^3 CN(Q,R,S) **do** P times:
  1. **compute**: $\ddot{X} \leftarrow C_{\rm II}^T \times \dot{X} + \ddot{X}$
  2. **data roll**: $\overset{+Q}{\Longleftarrow} \ddot{X} \overset{-Q}{\Longleftarrow} \; || \; \overset{+S}{\Longleftarrow} \dot{X} \overset{-S}{\Longleftarrow}$

Stage III. $\dddot{X}(S,Q,\tau) = \sum_{0 \le R < P} \ddot{X}(S,R,\tau) \times C(R,Q)$:

- **for all** P^3 CN(Q,R,S) **do** P times:
  1. **compute**: $\dddot{X} \leftarrow \ddot{X} \times C_{\rm III} + \dddot{X}$
  2. **data roll**: $\overset{+R}{\Longleftarrow} \dddot{X} \overset{-R}{\Longleftarrow} \; || \; \overset{+Q}{\Longleftarrow} \ddot{X} \overset{-Q}{\Longleftarrow}$

Due to the orbital (cyclical or rotational) nature of processing, after completion of each stage, i.e. after P "compute-and-roll" steps, all rotated data are returned to the same originated nodes and, therefore, a computed cubical tensor can be immediately used for the next stage of a cyclical 3D processing. Note that initial tensor $X = X(Q,R,S)$ is assigned to the nodes in the canonical (in-order) layout whereas all intermediate and final tensors, $\dot{X}, \ddot{X}$ and $\dddot{X}$, will be distributed in the skewed (out-of-order) layouts.

The first two stages implement the set of P space-independent 2D forward transform on P parallel along R-axis (orbit) slabs, $0 \le R < P$, with the $P \times P$ toroidally interconnected computer nodes $\{CN(Q,*,S)_R : 0 \le Q,S < P\}$ each (see Fig. 2 for the stage I and II). Totally, $2P$ "compute-and-roll" time-steps are needed for each R-th slab-of-nodes to independently implement a 2D forward transform (stage I and stage II) of the R-th slab-of-cubes $X(N_1,N_3)_{R=N_2}$.

This required initial, intermediate and final data distributions are defined and controlled by the modular (orbital) scheduling function τ. By using this scheduling, no data redistribution is needed during processing and the final P independent sets of 1D forward transform in stage III can be immediately started by P, now parallel along S-axis ($0 \le S < P$), slabs of the $P \times P$ toroidally interconnected computer nodes $\{CN(Q,R,*)_S : 0 \le Q,R < P\}$ each (see Stage III in Fig. 2 for $P = 2$). Totally, 3P "compute-and-roll" time-steps are needed for completion of 3D DXT.

An orbital computing of the 3D inverse transform is implemented as rolling back of the described above 3D forward transform where skewed distribution of the final cubical tensor $\dddot{X}$ among computer nodes will correspond to the initial three-way tensor distribution for the 3D inverse transform (see also Fig. 3 for $P = 2$).

After $3P$ "compute-and-roll" time-steps, i.e. after completion of a 3D inverse transform, the resulting cubical tensor X will be distributed in a 3D processor space in the canonical (in-order) layout.

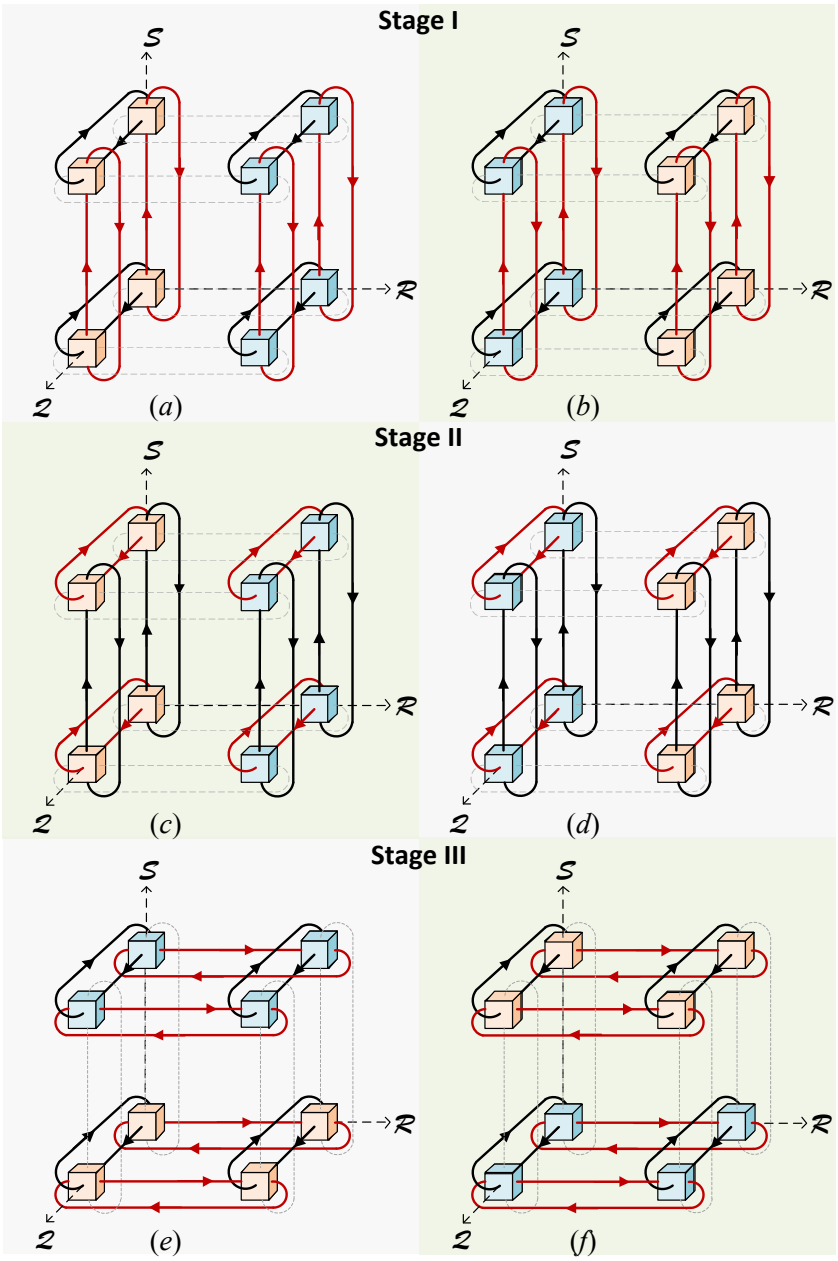


Figure 2: An orbital or circular data communication between nodes for the 3D forward transform.

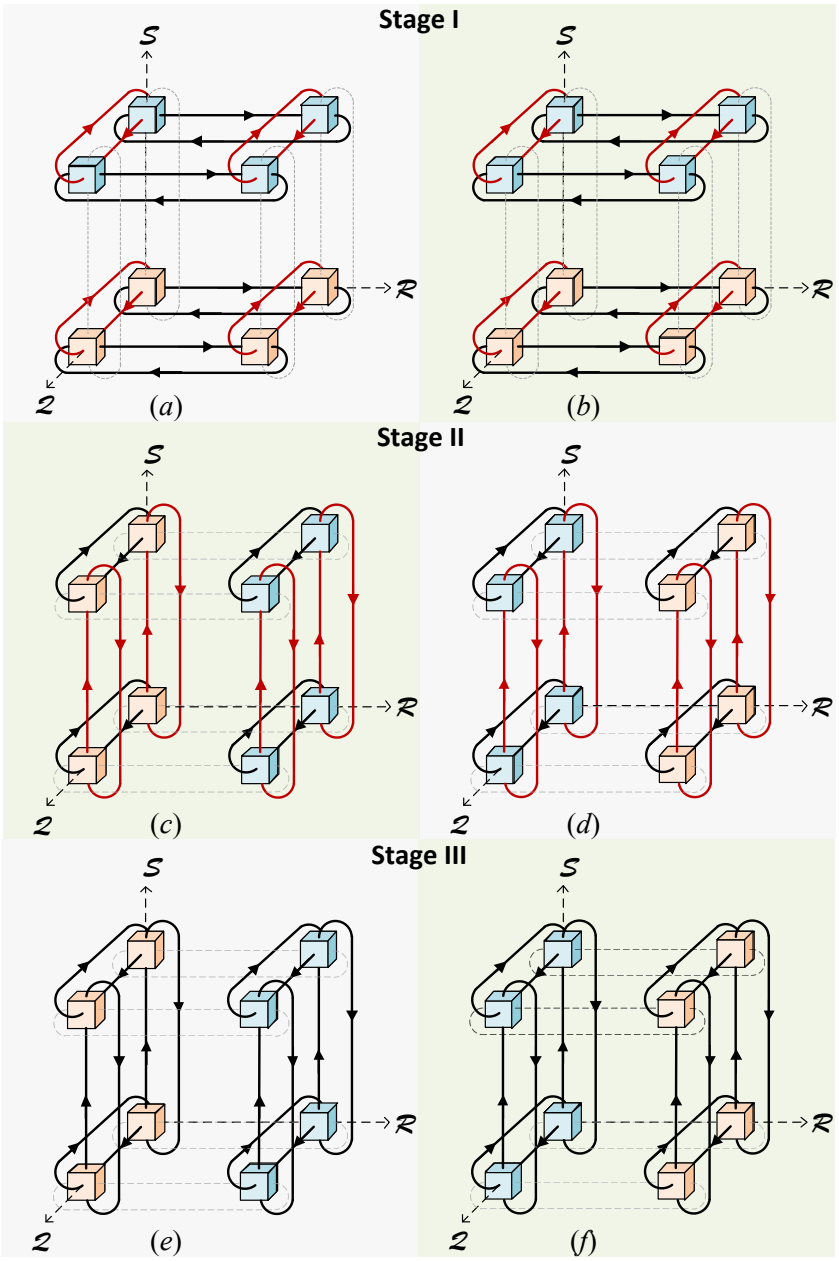


Figure 3: An orbital or circular data communication between nodes for the 3D inverse transform.

Stage I. $\dddot{X}(S,R,\tau) = \sum_{0\leq Q<P} \ddot{X}(S,Q,\tau) \times C(R,Q)^T$:

- **for all** P^3 CN(Q,R,S) **do** P times:
  1. **compute**: $\ddot{X} \leftarrow \dddot{X} \times C_{\mathrm{III}}^T + \ddot{X}$
  2. **data roll**: $\overset{+R}{\Longleftarrow} \dddot{X} \overset{-R}{\Longleftarrow} || \overset{+Q}{\Longleftarrow} \ddot{X} \overset{-Q}{\Longleftarrow}$

Stage II. $\dot{X}(Q,R,\tau) = \sum_{0\leq S<P} C(Q,S) \times \ddot{X}(S,R,\tau)$:

- **for all** P^3 CN(Q,R,S) **do** P times:
  1. **compute**: $\dot{X} \leftarrow C_{\mathrm{II}} \times \ddot{X} + \dot{X}$
  2. **data roll**: $\overset{+Q}{\Longleftarrow} \ddot{X} \overset{-Q}{\Longleftarrow} || \overset{+S}{\Longleftarrow} \dot{X} \overset{-S}{\Longleftarrow}$

Stage III. $X(Q,R,S) = \sum_{0\leq \tau<P} \dot{X}(Q,R,\tau) \times C(S,\tau)^T$:

- **for all** P^3 CN(Q,R,S) **do** P times:
  1. **compute**: $X \leftarrow \dot{X} \times C_I^T + X$
  2. **data roll**: $\overset{+S}{\Longleftarrow} \dot{X} \overset{-S}{\Longleftarrow} || \overset{+Q}{\Longleftarrow} C_I \overset{-Q}{\Longleftarrow}$

3.2 Algorithm's Pseudocodes

Each stage of forward and inverse transforms with 3D data decomposition has common structure, i.e. P steps of "compute-and-roll". The differences of parameters in each stages are:

- the data array is tensor or matrix,
- the data array should be rolled or not. If the data array is rolled, it should be defined which direction the data array is rolled along or opposite,
- the coefficient matrix is transposed or not,
- by slicing cubical tensor into the set of matrices, tensor-matrix or matrix-tensor multiplication can be represented as the set of independent matrix-by-matrix multiplications. Which 2D plane, Q-R, Q-S or S-R, the tensor is sliced along.

Table 1 shows those differences for the each stage (see also algorithm description in Section 3.1). Here, we describe the first operand in tensor-matrix multiplication or matrix-tensor multiplication as A. B is used as the second operand and the output tensor is described as C. "A (B) is tensor/matrix" means the first (second) data array is tensor or matrix. "Rolling direction of $\{A, B, C\}$" defines the direction for the data rolling for the each stage. Here, "-" means the data array is not rolled at the stage. "Transposition of A and B" shows whether input matrix A and B is transposed or not, respectively. "Slicing plane" defines which plane the sliced tensors belong to. By using this information, we simplify the psuedcode of the algorithm with 3D data decomposition.

The psuedocode of the algorithm with 3D data decomposition is shown in Algorithm 1. Each calling for the $Update$ function in line 2-4 corresponds to the stages I, II and III of the forward transform, respectively. X, $\dot{X}$, $\ddot{X}$ and $\dddot{X}$ are $b \times b \times b$ cubical tensors and C is $b \times b$ coefficient matrix. First three arguments are regarded as a data array A, B and C inside $Update$ function. Remaining arguments are used to tell the different parameters of each stage. The 3D block inverse transform has the same structure as the 3D block forward transform (see Algorithm 2).

Table 1: Parameters for each stage of the forward and inverse transforms.

	Forward I	Forward II	Forward III	Inverse I	Inverse II	Inverse III
A is tensor/matrix	Tensor	Matrix	Tensor	Tensor	Matrix	Tensor
B is tensor/matrix	Matrix	Tensor	Matrix	Matrix	Tensor	Matrix
Rolling direction of A	-	-	Q	R	-	S
Rolling direction of B	Q	S	-	-	Q	Q
Rolling direction of C	S	Q	R	Q	S	-
Transposition of A and B	NN	TN	NN	NT	NN	NT
Slicing plane	Q-S	Q-S	Q-R	Q-R	Q-S	Q-S

Algorithm 1 3D block forward transform

function $3DBlockForwardTransform(b, P, X, \dot{X}, \ddot{X}, \dddot{X}, C)$
  $Update(X(Q,R,S), C(S,\tau), \dot{X}(Q,R,\tau),$
    $true, false, NN, false, -, true, Q, true, S, Q-S)$
  $Update(C(Q,S), \dot{X}(Q,R,\tau), \ddot{X}(S,R,\tau),$
    $false, true, TN, false, -, true, S, true, Q, Q-S)$
  $Update(\ddot{X}(S,R,\tau), C(R,Q), \dddot{X}(S,Q,\tau),$
    $true, false, NN, true, Q, false, -, true, R, Q-R)$
end function

Algorithm 2 3D block inverse transform

function $3DBlockInverseTransform(b, p, X, \dot{X}, \ddot{X}, \dddot{X}, C)$
  $Update(\dddot{X}(S,Q,\tau), C(R,Q), \ddot{X}(S,R,\tau),$
    $true, false, NT, true, R, false, -, true, Q, Q-R)$
  $Update(C(Q,S), \ddot{X}(S,R,\tau), \dot{X}(Q,R,\tau),$
    $false, true, NN, false, -, true, Q, true, S, Q-R)$
  $Update(\dot{X}(Q,R,\tau), C(S,\tau), X(Q,R,S),$
    $true, false, NT, true, S, true, Q, false, -, Q-S)$
end function

$Update$ function in Algorithm 3 includes P steps of "compute" and "roll". Computed result is stored in a cubical tensor C. The arguments A and B are input data array of tensor/matrix which depends on the stage of the transform. We can know whether A and B is tensor or matrix from the arguments $IsTensorA$ and $IsTensorB$, respectively. $Compute$ function requires $IsTensorA$, $IsTensorB$, $Transpose$ and $SlicingPlane$ for updating output tensor. $IsSendX$ and $DirX$ is used in $DataRoll$ function. If $IsSendX$ is $true$, the data will send to $DirX$ directions. The "compute" part in line 3 and "roll" part in line 4-12 can be overlapped. The overlapping strategy is shown in Section 4.5.

Algorithm 3 Update

```
1:  function Update(C, A, B,
        IsTensorA, IsTensorB, Transpose,
        IsSendA, DirA,
        IsSendB, DirB,
        IsSendC, DirC,
        SlicingPlane)
2:      for step ← 1, P do
3:          Compute(A, B, C, IsTensorA,
        IsTensorB, Transpose, SlicingPlane)
4:          if IsSendA then
5:              DataRoll(A, DirA)
6:          end if
7:          if IsSendB then
8:              DataRoll(B, DirB)
9:          end if
10:         if IsSendC then
11:             DataRoll(C, DirC)
12:         end if
13:     end for
14: end function
```

The $Compute$ function, shown in Algorithm 4, updates cubical tensor by implementing tensor-matrix or matrix-tensor multiplication. In each loop, one slice of matrix from tensor is updated. By using $IsTensorA$ and $IsTensorB$, in line (3-9), whether the calling of this function is for tensor-matrix or matrix-tensor multiplication can be checked. Then, the i-th slice in $SlicingPlane$ is stored to temporary matrix. After checking, an slice of tensor is updated by matrix-matrix multiply-add operation. Depending on the $Transpose$, matrix A' or B' is transposed.

After local computation, the $DataRoll$ function is called for each data array $X \in \{A, B, C\}$ if $IsSendX$ is equal to $true$. According to the selected vector $a = (+1, +1, +1)^T$, the $DataRoll$ function, shown in Algorithm 5, send $Data$ along (+) direction Dir and receive $Data$ from opposite (-) direction, where $Dir \in \{Q, R, S\}$.

Algorithm 4 Compute

function $Compute(A, B, C, IsTensorA, IsTensorB, Transpose, SlicingPlane)$
  for $i \leftarrow 1, b$ **do**
    if $IsTensorA == true$ **then**
      A' is the i-th sliced matrix in $SlicingPlane$ plane of tensor A
      B' is the coefficient matrix B
    else if $IsTensorB == true$ **then**
      A' is the coefficient matrix A
      B' is the i-th sliced matrix in $SlicingPlane$ plane of tensor B
    end if
    C' is the i-th sliced matrix in $SlicingPlane$ plane of tensor C
    if $Transpose == NN$ **then**
      $C' \leftarrow A' \times B' + C'$
    else if $Transpose == TN$ **then**
      $C' \leftarrow A'^T \times B' + C'$
    else if $Transpose == NT$ **then**
      $C' \leftarrow A' \times B'^T + C'$
    end if
  end for
end function

Algorithm 5 Data Rolling

function $DataRoll(Data, Dir)$
  $Data$ are cyclically shifted along Dir axis.
end function

4 Algorithm Implementation

4.1 GEneral Matrix Multiplication

Because tensor-by-matrix or matrix-by-tensor multiplications can be expressed as the set of matrix-by-matrix multiplication, we can use an existing GEMM

[8] library to compute the 3D transform. GEMM operation is defined as $C \leftarrow \alpha op(A) \times op(B) + \beta C$, where both α and β are scalar values, and op(A), op(B) and C are $m \times k$, $k \times n$, $m \times n$ matrices, respectively. The op(X) is non-transposed (N) matrix (X) or transposed (T) matrix (X^T). Several implementations of GEMM from BLAS library such as ATLAS [6] and the Intel Math Kernel Library (MKL) [1] are available.

There are four different GEMM forms:

1. $C \leftarrow \alpha A \times B + \beta C$,
2. $C \leftarrow \alpha A^T \times B + \beta C$,
3. $C \leftarrow \alpha A \times B^T + \beta C$,
4. $C \leftarrow \alpha A^T \times B^T + \beta C$,

which are shortly named as NN, TN, NT and TT forms, respectively. The algorithm with 3D data decomposition requires NN form to compute the stage I and III of the forward transform, and stage II of the inverse transform. TN form is used in the stage II of the forward transform. NT form is used in the stage I and III of the inverse transform.

4.2 Intra-node Data Layout

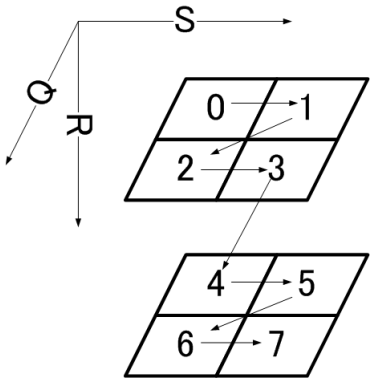


Figure 4: $2 \times 2 \times 2$ tensor data layout

It is assumed that a cubical ($b \times b \times b$) tensor is stored as a single 1D continuous block in the memory of each computer node. We propose that each slice in Q-S plane is stored in row-major order (see Fig. 4 as an example). A scalar element

$A(q,r,s)$ of cubical tensor is stored in m-th place of the memory, where $m = r \times n \times n + q \times n + s$.

The stage I and II of the forward transform require multiplication of each data slice in Q-S plane with coefficient matrix. Each slice of data in Q-S plane is stored sequentially in the node's memory. Thus, the stage I of the forward transform can be implemented directly by calling NN form of GEMM for each slice. The stage II requires transposition of coefficient matrix. However, GEMM allows its input matrix to be transposed. So, the stage II of the forward transform also can be computed by calling TN form of GEMM.

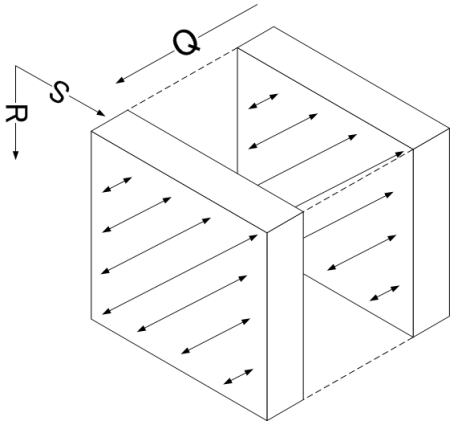


Figure 5: Transposition of each S-R plane

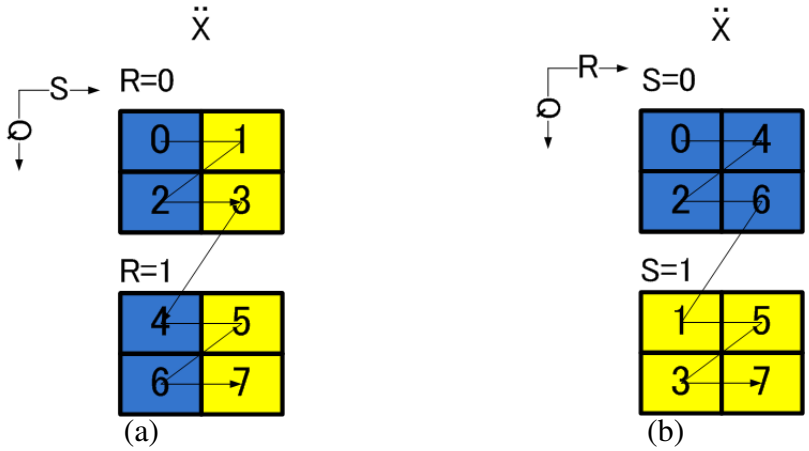


Figure 6: Data layout before data rearrangement (a) and after data rearrangement (b).

The stage III requires multiplication of each data slice in Q-R plane with coefficient matrix, i.e. NN form. However, the elements of intermediate tensor in Q-R plane are not located sequentially with stride one in the node's memory. Thus, we cannot call GEMM library directly. To use GEMM library, the cubical data should be rearranged inside memory of each node. In the local data rearrangement, a scalar element $A(q,r,s)$ will move to $A(q,s,r)$. This required data rearrangement corresponds to the transposition of each S-R plane (see Fig 5). An example of a $2 \times 2 \times 2$ data rearrangement is shown in Fig. 6. After transposition, we can directly call NN form of GEMM.

It is assumed that the inverse transform is computed after executing the forward transform. Thus, the initial data layout for the inverse transform is already properly rearranged. At the stage I of the inverse transform, each $b \times b$ slice in Q-R plane is multiplied with the coefficient matrix. The stage I can be directly computed by using NT form of GEMM without any data rearrangement. The stages II and III for the inverse transform compute each slice in Q-S plane. So, local data rearrangement within each node is required after the first stage. After rearrangement, NN and NT forms of GEMM can be used directly for the stages II and III of the inverse transform, respectively.

4.3 Multi-node Implementation

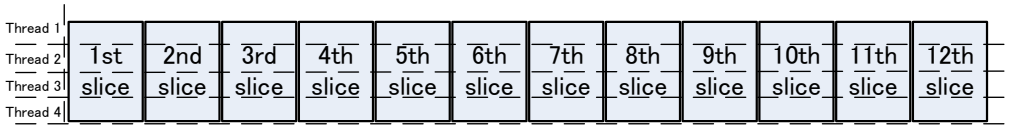
Message Passing Interface (MPI) [20] is used for multi-node implementation. MPI is highly portable and widely used as a standard library for parallel computing. Non-blocking point-to-point communication supported by MPI enables overlapping of computation and communication. `MPI_I_SEND` and `MPI_I_RECV` function supports non-blocking point-to-point communication.

To simplify a 3D implementation, a virtual 3D Cartesian inter-node process topology provided by MPI is used. In a 3D Cartesian topology, each MPI process is connected to its neighbors and we can set cyclic boundaries for the 3D torus interconnection. To create a Cartesian grid, `MPI_CART_CREATE` function is used. The information about nearest neighbor node is obtained by using `MPI_CART_SHIFT` function. This function only provide the information and MPI doesn't support cyclic shift. Thus, we should use the combination of the non-blocking point-to-point communication function, `MPI_I_SEND` and `MPI_I_RECV`, and `MPI_CART_SHIFT`.

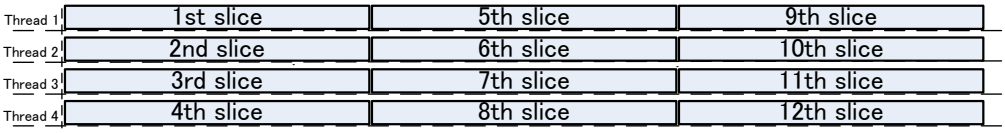
4.4 Multi-thread Optimization

Some systems support multi-threading on each node. There exists two possible ways to optimize tensor-matrix multiplication on such systems. The first one is to use multi-threaded GEMM library. For example, ATLAS [6] supports

multi-thread GEMM implementation. In the first approach, each slice of the tensor is computed by multiple threads (see Fig. 7 (a) for 12 slices and four threads). The other one is to use OpenMP. Each slice in tensor is computed by different threads at the same time (see Fig. 7 (b)). To use OpenMP, we just add one line, `#pragma omp parallel for`, before loop of matrix-matrix multiplication (see Algorithm 3). We cannot say that which one is better because it depends on the system. To obtain the best performance, it is better to evaluate both implementations on mutli-threaded systems.



(a) In multi-threaded GEMM, each slice is computed by four threads.



(b) In OpenMP implementation, each slice is computed in parallel.

Figure 7: Examples of task distribution, each slice of matrix-matrix multiplication, among four threads.

4.5 Multi-node Optimization

Each stage of the algorithm with 3D data decomposition consists of P steps of "compute-and-roll". The local computation and inter-node communication at each step can be overlapped. In each rolling part, two of three data arrays, (input cubical tensor A, coefficient matrix B and output cubical tensor C), are rolled. Depends on which data arrays are rolled, we should consider three different strategies for optimization.

4.5.1 Overlapping Data

Let us consider the situation that rolled data are the input cubical tensor A and coefficient matrix B. The output tensor C is fixed inside a computer a node. Thus, we can send A and B while independently updating the tensor C. In this case, a perfect (100%) overlapping can be possible. This optimization technique is used in the stage III of the inverse transform.

4.5.2 Slicing Data

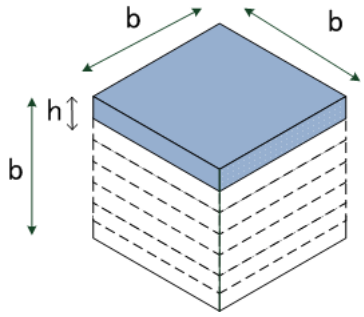


Figure 8: Sliced cubical tensor.

Next case when coefficient matrix B and computed cubical tensor C are rolled. Transferring of the coefficient matrix B can be totally overlapped with computation. However, output tensor should be updated before transmitting. To solve this problem, we use slicing of cubical output tensor. Let us define h as the slice height for the $b \times b \times b$ output tensor, where b mod $h = 0$. An output tensor is divided into $b \times b \times b/h$ slabs (see Fig. 8). Each data slab is transferred immediately after updating and is overlapped with computing of subsequent slab. In this case, only data transferring of the last slice cannot be overlapped with the computation. Simplified scheduling for updating slabs when number of slabs is 4, i.e, $b/h = 4$, is shown in Fig. 9. The stage I of the forward transform uses this overlapping technique.

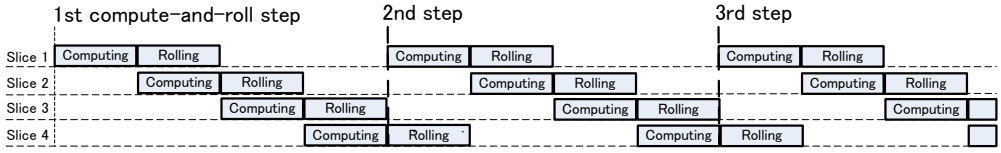
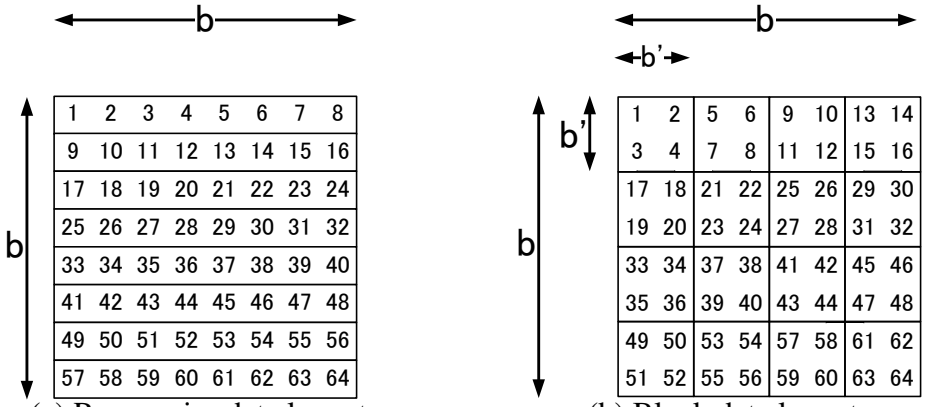


Figure 9: Overlapping of computing and data rolling for 4 divided slabs.

4.5.3 Overlapping and Slicing

When the input cubical tensor A and output cubical tensor C should be rolled, both of the overlapping and slicing data techniques are employed for input tensor A and output tensor C, respectively. This technique is required for remaining stages.

4.6 Blocked Matrix Multiplication



(a) Row-major data layout (b) Block data layout

Figure 10: (a) Row-major data layout and (b) block data layout; the number in each data layout indicates the order of memory assignment.

As we discussed previously, each slice of the tensor is stored as row-major order in Q-S plane or Q-R plane. It is well known that blocked algorithms increase the performance reusing data. We apply two levels of blocking for the ZGEMM, which is GEMM for complex-double precision, on the Blue Gene/Q.

At the first level of blocking, each $b \times b$ slice of tensor is divided into $b' \times b'$ square blocks. The data layout is changed to store each $b' \times b'$ block sequentially in the memory. This kind of data access pattern is called as block data layout (BDL) [21] (see Fig. 10 (b)). It can shown that $b' = 16$ on the BG/Q is the optimal parameter. If the size of problem for each node, b, is less than 16, we solve the problem by using traditional row-major data layout (see Fig. 10 (a)).

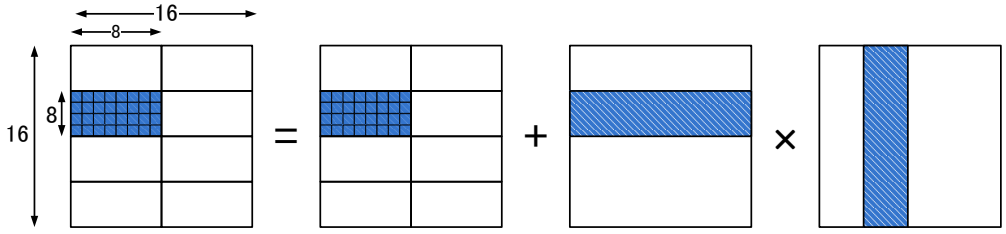


Figure 11: The second level of blocking

As the second level of blocking, to utilize SIMD instructions on the BG/Q, the

16×16 resulting matrix is divided into 4×8 rectangular matrices (see Fig. 11). To update these 4×8 matrices efficiently, we implemented NN form of ZGEMM kernel by using intrinsic functions for vectorized fused multiply-add (`fma`) and fused multiply-subtract (`fms`) operations.

Currently, we only implemented NN form of ZGEMM. The 3D forward and inverse transforms with cubical data decomposition also requires TN and NT form. To deal with it, additional transposition for the coefficient matrix is required. However, time for transposition of the 2D $b \times b$ matrix can be negligible in total time of computing.

5 Performance Evaluation on the IBM Blue Gene/Q

In this Section, performance evaluation of the 3D DFT with 3D data decomposition in complex-double precision on the IBM Blue Gene/Q is presented. This system supports 5D torus interconnected networks. Following formula is used to estimate performance:

$$\text{Performance (Gflop/s)} = \frac{\text{The number of flops}}{\text{Time (sec)}}. \tag{14}$$

The number of floating point operations for 3D DFT can be defined as $3 \times 8 \times N^4$.

The BG/Q compute chip (BQC) is a System-on-Chip (SoC) design combining CPUs, caches, network and message unit on a single chip [13]. A single BG/Q rack contains 1024 BG/Q nodes. Each node contains the BQC and 16 GB of memory. Each BQC has 16 PowerPC A2 cores. The A2 core runs at 1.6 GHz and allows 4 `fmas` per cycle, translating to a peak performance per core of 12.8 Gflop/s or 204.8 Gflop/s for the BQC. The BG/Q has a 5-D torus interconnected topology. Each compute node has 10 communication links with its neighbors. Each torus link can simultaneously send at 2GB/s and receive at 2GB/s [5].

The IBM BG/Q system at the High Energy Accelerator Research Organization (KEK), Japan, is used to evaluate performance of the 3D DFT with 3D data decomposition in double precision. We measured the performance of 3D DFT in complex-double precision with different number of nodes. We used our optimized ZGEMM. The performance is measured on 32 ($2 \times 2 \times 2 \times 2 \times 2$), 128 ($2 \times 2 \times 4 \times 4 \times 2$) and 512 ($4 \times 4 \times 4 \times 4 \times 2$) nodes. The details are described in Table 2. The first column of the table shows the number of nodes and the second column shows the total number of MPI processes. The third column shows the number of MPI processes per node. BG/Q allows to assign multiple cores per MPI processes. The forth column shows the maximum number of the available threads per MPI process.

Table 2: The available number of nodes and number of MPI processes on the BG/Q.

# Nodes	# MPI Processes	# MPI Processes per node	# threads per MPI processes
32	64 (4^3)	2	32
32	512 (8^3)	16	4
128	512 (8^3)	4	16
512	512 (8^3)	1	64

5.1 Performance Evaluation

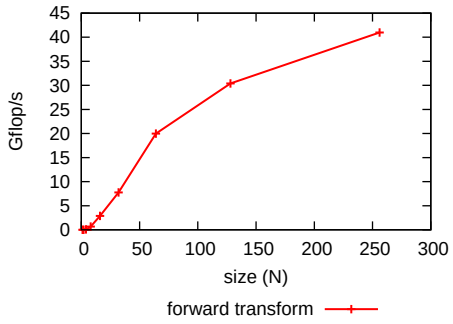
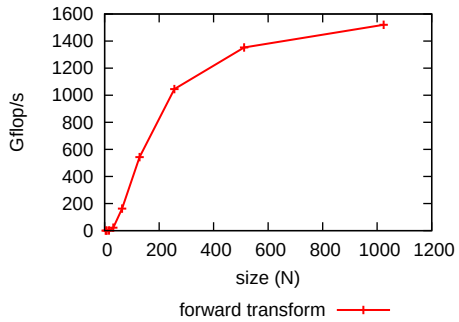


Figure 12: Performance results with 1 node on the BG/Q.

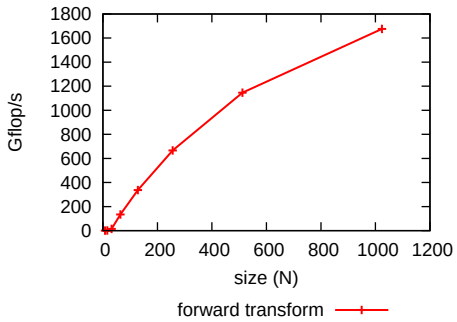
Fig. 12 shows the one node results on the BG/Q. The available performance efor the forward and inverse transforms are almost the same. It achieves about 41 Gflop/s when $N = 256$ for the forward and inverse transforms. This is 20% of the hardware peak performance in one node.

The performance results on the BG/Q is shown in Fig. 13. For all four different configuration of nodes, the performance of the forward and inverse transforms are almost the same. On 32 nodes, the performance for the 3D DFT with 64 MPI process is 1519.3 Gflop/s (23.2% of the hardware peak performance) and with 512 MPI process 1675.5 Gflop/s (25.6% of peak). The obtained performance on 128 nodes with 512 MPI processes is 6563.05 Gflop/s (25.0% of the peak). The efficiency is almost the same when the number of nodes are 32. The obtained performance of 512 nodes with 512 MPI processes is 25579.08 Gflop/s (24.4% of the peak).

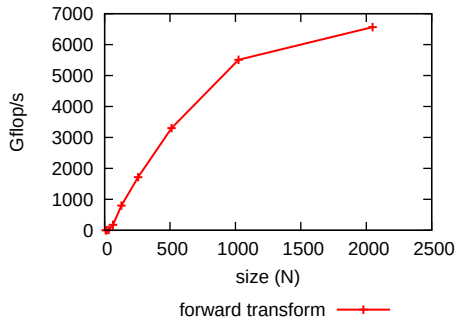
To better understand how the computation time is compared to the communi-



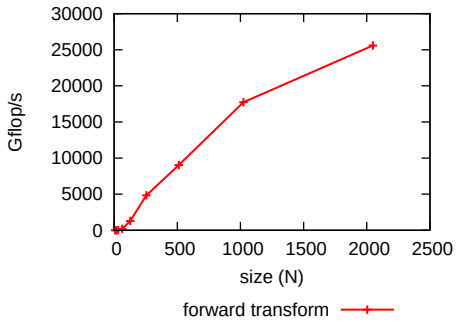
(a) 64 MPI processes (32 nodes $\times$ 2 MPI process per nodes).



(b) 512 MPI processes (32 nodes $\times$ 16 MPI process per nodes).



(c) 512 MPI processes (128 nodes $\times$ 4 MPI processes per nodes).



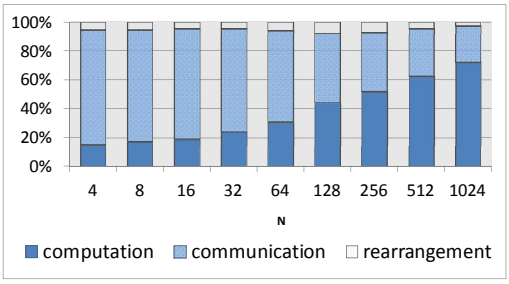
(d) 512 MPI processes (512 nodes $\times$ 1 MPI process per nodes).

Figure 13: Performance results on the BG/Q.

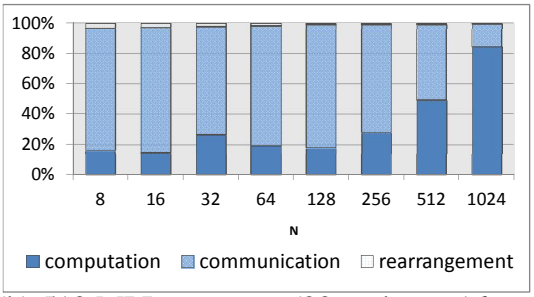
cation time, we show the relative percentages of the time spent for "computation", "communication" and other "rearrangement". Here, "computation" includes both updating of output tensor and overlapped communication time. From Fig. 14, we can see that the ratio of computation increases smoothly and reaches more than 70% of the total execution time. Because of the scalability of the torus cluster we can expect that the tendency of the relative percentage of computation time is the same for larger number of nodes and we can say that good weak scaling can be possible on this system.

5.2 Strong Scalability

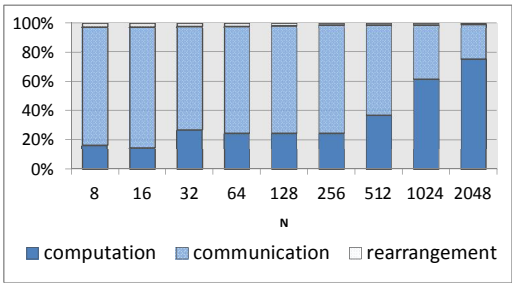
The execution time is shown in Fig. 15. The x-axis shows the number of MPI processes on a logarithmic scale and the y-axis shows the execution time on a logarithmic scale. For $N = 1024$, the speedup T_{32}/T_{128} is 3.72 and T_{128}/T_{512} is 3.24. For $N = 2048$, the speedup T_{128}/T_{512} is 3.85 which is close to the maximum speedup (512 nodes / 128 nodes = 4). For the large size, we can see good strong scalability.



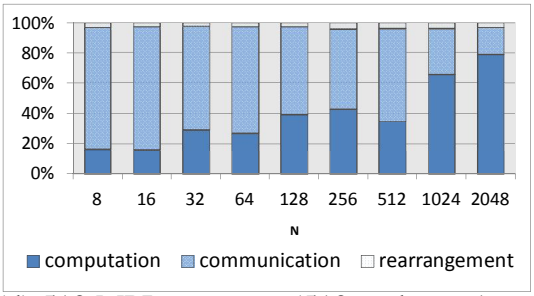
(a) 64 MPI processes (32 nodes $\times$ 2 MPI process per nodes).



(b) 512 MPI processes (32 nodes $\times$ 16 MPI process per nodes).



(c) 512 MPI processes (128 nodes $\times$ 4 MPI processes per nodes).



(d) 512 MPI processes (512 nodes $\times$ 1 MPI process per nodes).

Figure 14: The relative percentage of computation, communication and all other overheads for the inverse 3D DFT on the BG/Q with different number of MPI processes.

5.3 Comparison with 3D FFT

We compared our result of 3D DFT with 3D data decomposition with 3D FFT with 2D data decomposition [12] when the problem size is 1024^3. The number of flops of the 3D DFT and FFT is approximately $24 \times N^4$ and $15 \times N^3 \log_2 N$, respectively. Thus, the 3D DFT is approximately $24N^4/15N^3 \log_2 N = N/log_2 N$ times slower than 3D FFT, i.e, 163.84 times slower when $N = 1024$. We can see that 3D DFT is only 6-8 times slower than 3D FFT (see Table 3), i.e. about 19-27 times faster than theoretical estimation.

Currently, our one node implementation for tensor-matrix multiplication routine is only 20% of the peak performance of one node, and it affect the performance on multinode. We expect that it is still possible to optimize performance in one node, and thus further improvement of performance can be possible. Moreover, currently it is practically impossible to use the number of nodes which is equal to the problem size, i.e. 1024^3, to demonstrate performance of the algorithm with 3D data decomposition. We can expect that there is a possibility that 3D DFT is faster than 3D FFT especially the number of nodes is equal to size of

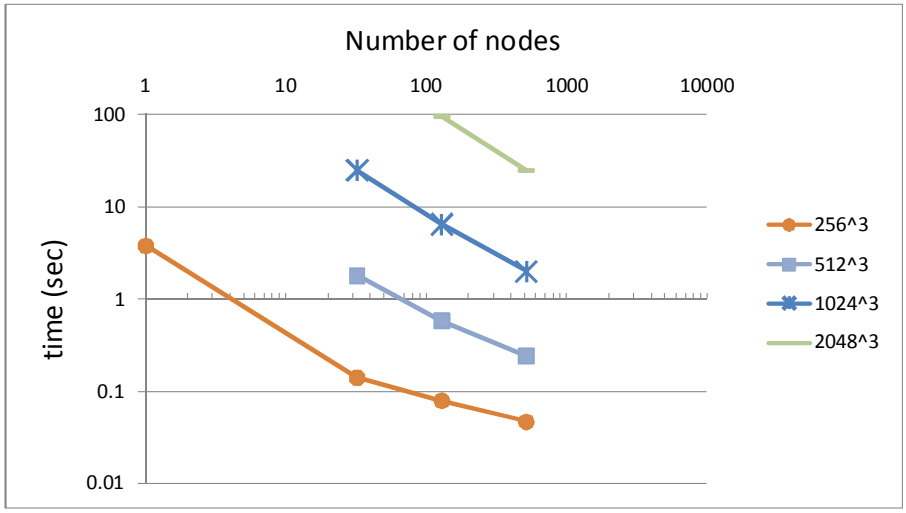


Figure 15: Speedup

problem.

Table 3: Comparison of execution time between 3D DFT with 3D data decomposition and 3D FFT with 2D decomposition [12] when the problem size is 1024^3.

# Nodes	3D FFT [12] (sec)	3D DFT (sec) (forward)	DFT/FFT (forward)	3D DFT (sec) (inverse)	DFT/FFT (inverse)
32	2.731	15.749	5.77	16.251	5.95
128	0.713	4.790	6.72	4.886	6.85
512	0.179	1.488	8.31	1.475	8.24

6 Conclusions

We have shown the implementation and performance evaluation of the algorithm with 3D data decomposition. To the best of our knowledge, this work is the first implementation of the 3D data decomposition of the discrete forward/inverse transform.

We proposed pseudocode, how to implement the algorithm and some optimization techniques for the 3D data decomposition. Our proposed implementa-

tion requires GEMM routine from BLAS and MPI, which are usually available on the supercomputers. Thus, by using those libraries, the implemented code can be easily ported to any environment. There are two possible ways for multit-thread optimization, using multi-threaded GEMM or OpenMP. Which is better is depends on the system. Each of the forward and inverse transforms of 3D discrete transform has three different stages. We define three different optimization techniques based on which data arrays are rolled.

We evaluated 3D DFT on the BG/Q with various number of nodes as an example of implementation for the algorithm with 3D data decomposition. In this environment, we can compare our results with the 3D FFT on the BG/Q. The practical execution time for 3D DFT is about 19-27 times faster than the theoretical estimation. Therefore, we can predict that the 3D DFT with 3D data decomposition can be faster than 3D FFT in the future supercomputers where the number of nodes will be equal to the size of data, i.e., N^3.

6.1 Future Works

Future work includes following things. Our implementation of the algorithm with 3D data decomposition is required to compute the small size of matrix-matrix multiplication efficiently. However, currently, GEMM can achieve the high performance when the problem size is relatively large. To improve GEMM performance in small size is considered. To make the analytical model is also important task to estimate the performance with different problem sizes and number of processors.

Acknowledgment

The performance evaluation on the IBM Blue Gene/Q is supported by the Large Scale Simulation Program No. 12/13-12 (FY2012) of High Energy Accelerator Research Organization (KEK). We thank N. Nakasato, from The University of Aizu, T. Minami and F. Yuasa, from KEK, for their support to make it possible to access the IBM Blue Gene/Q.

References

[1] "Intel Math Kernel Library (Intel MKL)," http://software.intel.com/en-us/intel-mkl/.

[2] O. Ayala and L. Wang, "Parallel implementation and scalability analysis of 3D fast Fourier transform using 2D domain decomposition," *Parallel Computing*, 2012.

[3] G. Baumgartner, A. Auer, D. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. Harrison, S. Hirata, S. Krishnamoorthy, S. Krishnan, C. Lam, Q. Lu, M. Nooijen, R. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov, "Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 276 –292, feb. 2005.

[4] E. Brachos, "Parallel FFT Libraries," Master's thesis, The University of Edinburgh, 2011. [Online]. Available: http://www.brachos.gr/files/E-Brachos_FFT.pdf

[5] D. Chen, N. Eisley, P. Heidelberger, R. M. Senger, Y. Sugawara, S. Kumar, V. Salapura, D. L. Satterfield, B. D. Steinmacher-Burow, and J. J. Parker, "The IBM Blue Gene/Q Interconnection Fabric," *IEEE Micro*, vol. 32, no. 1, pp. 32–43, 2012.

[6] R. Clint Whaley, A. Petitet, and J. Dongarra, "Automated empirical optimizations of software and the ATLAS project," *Parallel Computing*, vol. 27, no. 1, pp. 3–35, 2001.

[7] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Mathematics of Computation*, vol. 19, pp. 297–301, 1965.

[8] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff, "A set of level 3 basic linear algebra subprograms," *ACM Trans. Math. Softw.*, vol. 16, pp. 1–17, March 1990. [Online]. Available: http://doi.acm.org/10.1145/77626.79170

[9] M. Eleftheriou, J. E. Moreira, B. G. Fitch, and R. S. Germain, "A Volumetric FFT for BlueGene/L," in *HiPC*, ser. Lecture Notes in Computer Science, T. M. Pinkston and V. K. Prasanna, Eds., vol. 2913. Springer, 2003, pp. 194–203.

[10] S. Filippone, "The IBM parallel engineering and scientific subroutine library," in *PARA*, ser. Lecture Notes in Computer Science, J. Dongarra, K. Madsen, and J. Wasniewski, Eds., vol. 1041. Springer, 1995, pp. 199–206.

[11] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005, special issue on "Program Generation, Optimization, and Platform Adaptation".

[12] S. Habib, V. A. Morozov, H. Finkel, A. Pope, K. Heitmann, K. Kumaran, T. Peterka, J. A. Insley, D. Daniel, P. K. Fasel, N. Frontiere, and Z. Lukic, "The universe at extreme scale: multi-petaflop sky simulation on the BG/Q," in *SC*, 2012, p. 4.

[13] R. A. Haring, M. Ohmacht, T. Fox, M. Gschwind, D. L. Satterfield, K. Sugavanam, P. Coteus, P. Heidelberger, M. A. Blumrich, R. Wisniewski, A. Gara, G. L.-T. Chiu, P. A. Boyle, N. Chist, and C. Kim, "The IBM Blue Gene/Q Compute Chip," *IEEE Micro*, vol. 32, no. 2, pp. 48–60, 2012.

[14] C. Hillar and L.-H. Lim, "Most tensor problems are NP hard," 2009. [Online]. Available: http://www.citebase.org/abstract?id=oai:arXiv.org:0911.1393

[15] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM Review*, vol. 51, no. 3, pp. 455–500, September 2009.

[16] L. D. Lathauwer, B. D. Moor, and J. Vandewalle, "A multilinear singular value decomposition," *SIAM J. Matrix Anal. Appl*, vol. 21, pp. 1253–1278, 2000.

[17] N. Li and S. Laizet, "2DECOMP&FFT - A Highly Scalable 2D Decomposition: Library and FFT Interface," in *Cray User Group 2010 conference*, 2010, pp. 1–13. [Online]. Available: http://www.2decomp.org/pdf/17B-CUG2010-paper-Ning_Li.pdf

[18] L.-H. Lim, "Numerical Multilinear Algebra: a new beginning?" in *Matrix Computations and Scientific Computing Seminar*, 2006, pp. 1–51. [Online]. Available: http://galton.uchicago.edu/~lekheng/work/mcsc1.pdf

[19] Q. Lu, X. Gao, S. Krishnamoorthy, G. Baumgartner, J. Ramanujam, and P. Sadayappan, "Empirical performance model-driven data layout optimization and library call selection for tensor contraction expressions," *J. Parallel Distrib. Comput.*, vol. 72, no. 3, pp. 338–352, Mar. 2012. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2011.09.006

[20] Message Passing Interface Forum, "MPI: A Message-Passing Interface standard, (version 3.0)," 2012, http://www.mpi-forum.org/.

[21] N. Park, B. Hong, and V. K. Prasanna, "Tiling, Block Data Layout, and Memory Hierarchy Performance," *IEEE Trans. Parallel Distrib. Syst.*, vol. 14, no. 7, pp. 640–654, 2003.

[22] D. Pekurovsky, "P3DFFT," *Website*, October, 2012. [Online]. Available: http://www.sdsc.edu/us/resources/p3dfft/

[23] S. G. Sedukhin, "Co-design of Extremely Scalable Algorithms/Architecture for 3-Dimensional Linear Transforms," The University of Aizu, Tech. Rep. TR2012-001, July 2012. [Online]. Available: ftp://ftp.u-aizu.ac.jp/u-aizu/doc/Tech-Report/2012/2012-001.pdf

[24] S. G. Sedukhin, A. S. Zekri, and T. Myiazaki, "Orbital algorithms and unified array processor for computing 2D separable transforms," *Parallel Processing Workshops, International Conference on*, vol. 0, pp. 127–134, 2010. [Online]. Available: http://dx.doi.org/10.1109/ICPPW.2010.29

[25] E. Solomonik, J. Hammond, and J. Demmel, "A preliminary analysis of cyclops tensor framework," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2012-29, Mar 2012. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-29.html