Contents lists available at SciVerse ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Parallel implementation and scalability analysis of 3D Fast Fourier Transform using 2D domain decomposition

Orlando Ayala [a,b,*], Lian-Ping Wang [a]

[a] Department of Mechanical Engineering, 126 Spencer Laboratory, University of Delaware, Newark, DE 19716-3140, USA
[b] Centro de Métodos Numéricos en Ingeniería, Escuela de Ingeniería y Ciencias Aplicadas, Universidad de Oriente, Puerto La Cruz, Venezuela

ARTICLE INFO

Article history:
Received 9 September 2011
Received in revised form 25 July 2012
Accepted 4 December 2012
Available online 20 December 2012

Keywords:
FFT
2D decomposition
Parallel computing

ABSTRACT

3D FFT is computationally intensive and at the same time requires global or collective communication patterns. The efficient implementation of FFT on extreme scale computers is one of the grand challenges in scientific computing. On parallel computers with a distributed memory, different domain decompositions are possible to scale 3D FFT computation. In this paper, we argue that 2D domain decomposition is likely the best approach in terms of using a very large number of processors with reasonable data communication overhead. Specifically, we extend the data communication approach of Dmitruk et al. (2001) [21] previously used for 1D domain decomposition, to 2D domain decomposition. A thorough quantitative analysis of the code performance is undertaken for different problem sizes and numbers of processors, including scalability, load balance, dependence on subdomain configuration (i.e., different numbers of subdomain in the two decomposed directions for a fixed total number of subdomains). We show that our proposed approach is faster than the existing attempts on 2D-decomposition of 3D FFTs by Pekurovsky (2007) [23] (p3dfft), Takahashi (2009) [24], and Li and Laizet (2010) [25] (2decomp.org) especially for the case of large problem size and large number of processors (our strategy is 28% faster than Pekurovski's scheme, its closest competitor). We also show theoretically that our scheme performs better than the approach by Nelson et al. (1993) [22] up to a certain number of processors beyond which latency becomes and issue. We demonstrate that the speedup scales with the number of processors almost linearly before it saturates. The execution time on different processors differ by less than 5%, showing an excellent load balance. We further partitioned the execution time into computation, communication, and data copying related to the transpose operation, to understand how the relative percentage of the communication time increases with the number of processors. Finally, a theoretical complexity analysis is carried out to predict the scalability and its saturation. The complexity analysis indicates that the 2D domain decomposition will make it feasible to run a large 3D FFT on scalable computers with several hundred thousands processors.

1. Introduction

Fast Fourier Transform (FFT) is an analytical and numerical tool widely used in sciences and engineering. Examples include signal processing, applied mathematics, image processing (to identify features in image), spectral analysis to solve differential equations, and control processing. Specifically, in molecular dynamics FFT is used to calculate Coulomb energies

* Corresponding author at: Department of Mechanical Engineering, 126 Spencer Laboratory, University of Delaware, Newark, DE 19716-3140, USA.
E-mail address: omayalah@udel.edu (O. Ayala).

0167-8191/$ - see front matter © 2012 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.parco.2012.12.002

and gradients due to inter-atomic potentials [1,2]. In seismic imaging, structure of portion of the down earth is reconstituted through seismic reflection with the use of FFT to find hydrocarbons [3]. In computational geosciences, geodynamo theory makes use of FFT to calculate the convection and magnetic field generated by celestial bodies such as the Earth [4]. In fluid mechanics, three-dimensional time-dependent turbulent flows are often calculated using Direct Numerical Simulations (DNS) by spectral methods in which FFT is used to relate the flow in physical space to that in the wavevector space [5–7].

In the above applications, FFT is either applied to very large data size in multiple dimensions, or performed for many times (or both). FFT is computationally intensive and at the same time requires global or collective communication patterns. The efficient implementation of FFT on extreme scale computers is often considered one of the grand challenges in scientific computing.

In the last two decades, several strategies have been proposed to parallelize FFT computation and to improve its performance. For example, to speed up the overall performance, Bylaska et al. [1], Iovieno et al. [8], and Cavazzoni and Chiarotti [2] decomposed the domain of the data and performed FFT only on sections where non-zero elements were present, while Poikko [9] did not communicate sections with zero values among parallel processors. In molecular dynamics simulations, the domain can be decomposed over electronic orbital to avoid load imbalance among processors [10]. The nature of mapping of the parallel processors to sub-domains has also been found to affect the latency time on data communication [1]. Haynes and Cote [11] decomposed the domain into even-indexed and odd-indexed sets of data to split the global FFT into local FFTs on each set. Eleftheriou et al. [12] and Fang et al. [13] overlapped communications and calculations of the decomposed domains to speed up the overall FFT calculations. Takahashi [14] divided the three-dimensional (3D) domain into blocks of data that fits the cache memory. Chu and George [15] compiled different strategies that utilized radix-2 and radix-4 schemes for 1D FFT using butterfly operations. Calvin [16] split the 1D FFT procedure into several steps with a multiple-of-two number of data sets and studied the optimal data block size to minimize overhead in communication. Inda and Bisseling [17] divided the butterfly operations in such a way that the data could fit cache memory. Marino and Swartzlander [18] proposed a matrix–vector multiplication strategy of the 1DFFT to avoid communication and allow data independency between processors. However, their approach was designed for a shared-memory computer where all processors had access to the full data field. Furthermore, they indicated that their method involved $N^2/2$ operations which is larger than the $\mathcal{O}(N \log N)$ operation count for the Cooley-Tukey algorithm (N is the number of data points or nodes). Takahashi et al. [19] divided the serial FFT in different columns to better fit the cache memory in order to speed up calculations.

It is difficult to rank the available parallel FFTs since there are many different versions developed for different parallel schemes and different parallel computers, and some are mainly adapted to the specific problems in hand. In general, there are two strategies to parallelize FFT: (1) Using distributed FFT in which the 1D Fourier transform is split into smaller transforms recursively (the Cooley–Tukey algorithm), with each processor handling in parallel each of these smaller transforms, and (2) Transposing data. For a small number of processors, distributed FFT performs better, while for a large number of processors transposing data is better [20]. In this work, we will use the transposing data technique.

The transpose approach relies on domain decomposition. It takes advantage of the fact that a 3D FFT of, for instance, a N^3 data field, is a linear superposition of three subsequent series of N^2 1D FFTs, along each Cartesian coordinate, thus it uses the serial 1D FFT as the building block. The number of transposes after some (or all) the series of N^2 1D FFTs, depend upon the way the whole domain is decomposed or partitioned among all processors. The transpose steps are the only ones that need communications. The single 1D FFT is not parallelized and any available package, such as the well-known FFTW by Frigo and Johnson [28], could be used. These 1D FFT computations are always performed locally making it independent of the transpose communications. The well-known Amdahl's law does not apply for this scheme as the computation alone is embarrassingly parallel becuase the N^2 1D FFTs are divided among all the machines and performed in parallel. However, although the N^2 1D FFTs computation are embarrassingly parallel, the communications for the transposes bring the well-known latency problem for very large number of processors, which is of concern on this paper.

There are three ways to decompose the 3D domain for the transpose approach. The 1D decomposition, also called slab decomposition, divides the domain into equal-size blocks along only one Cartesian coordinate [21]. To perform 1D FFT along the divided Cartesian coordinate, a transpose operation of the data must take place, which involves communication among all parallel processors. The two-dimensional (2D) decomposition, known as the pencil decomposition, partitions the domain along two Cartesian coordinates [22–25]. In this case, two separate communication steps are required. The 3D or volumetric decomposition applies data partitions in all Cartesian coordinates [12,13,26,27]. Here at least three separate data communication steps are needed. It is important to note that, in the 2D and 3D decompositions, the data communication during each communication step does not involve all pairings of processors, but only a subset of processor-processor communications whose details depend on how the data are distributed (see Section 2 for further discussion this aspect in terms of 2D decomposition).

For a given problem size, Takahashi [24] had shown that the 1D-decomposition strategy is better as it utilizes fewer communications but the drawback is that it does not permit the use of a large number of processors because we can only use up to the N number of data points or nodes along the decomposed Cartesian coordinate for the number of processors. On the other hand, the 3D-decomposition suffers from excessive communication. Here, we aim specifically at developing a 2D domain-decomposition implementation of 3D FFTs, so that our spectral flow simulation code can leverage the resources of upcoming petascale computers with at least several hundred thousands processors. Preliminary attempts on 2D-decomposition of 3D FFTs using different communication/implementation strategies for the array-transpose operation have been made by Plimpton at Sandia National Laboratories [22], Pekurovsky [23] (p3dfft), Takahashi [24], and Li and Laizet [25]

(2decomp.org). For example, Plimpton's strategy was to pack and send as much data as possible in a multistage algorithm using MPI_Send and MPI_Irecv commands. Pekurovsky, and Li and Laizet simply used the traditional MPI command MPI_AlltoAllv while Takahashi used MPI_AlltoAll to communicate data. Pekurovsky, Li and Laizet, and Takahashi also used MPI_Cartsub command to establish certain pattern of communication among processors.

Within 1D decomposition, Dmitruk et al. [21] proposed inter-processor data communications using only MPI_Send and MPI_Recv commands and they showed that their proposed approaches performed better than other communication strategies such as the MPI_AlltoAll command. Our main objective here is to extend this idea to 2D decomposition for 3D FFT of real data. We will show that our proposed approach is faster than the existing attempts on 2D-decomposition of 3D FFTs by Pekurovsky [23] (p3dfft), Takahashi [24], and Li and Laizet [25] (2decomp.org). We developed a theoretical complexity analysis that allowed us to predict scalability performance, as well as to estimate the maximum number of processors (P_{max}) up to which the proposed scheme maintains a good scalability. We also show that our scheme performs better than the approach by Nelson et al. [22] when using up to certain number of processors. We further demonstrated theoretically that 3D-decomposition strategy is indeed expensive due to the increased amount of communications and that our proposed scheme can be further improved by overlapping some computations and communications steps.

The paper is organized as follows. In Section 2, we first describe in detail the 2D decomposition strategy that we implemented to perform 3D FFT of real data. A complexity analysis of the implementation is developed in Section 3. Section 4 contains testing and evaluation of the implementation in which we present a detailed analysis of its performance on an IBM cluster (i.e., Bluefire at NCAR) using up to 2048 processors and compare our proposed approach with the different available schemes on 2D-decomposition of 3D FFTs. In addition, we compare the performance of our subroutine on different supercomputers (Bluefire, Hopper, Kraken, and Ranger). Later, after validating the theoretical complexity analysis, we use it to theoretically compare our scheme against other approaches and to explore options to improve it. Finally, conclusions are summarized in Section 5.

2. Parallel implementation of 3D FFT using 2D domain decomposition

A 3D FFT can be computed by taking a sequence of three 1D FFTs along each direction of the three dimensional block data. On a parallel computer with distributed memory, our strategy is to slice the computational domain along two spatial directions, say, *y*- and *z*- directions (see Fig. 1a). The 1D FFT along the *x*- direction of the data can be easily performed on each processor as each 1D data vector is completely located within a processor. The 1D FFTs along *y*- and *z*- directions, however, cannot be done directly because the data vector is distributed on different processors. These two last 1D FFT computations can be executed only after the data vector is moved into a given processor or along the *x* matrix direction. This exchange of data coordinate directions is the transpose operation when communications among processors take place. FFT computations are performed locally and are independent of the transpose communications.

Fig. 1 shows schematically the 2D decomposition strategy (we only show the Real to Complex 3D FFT as same strategy applies for Complex to Real transforms). Let N_x, N_y, and N_z denote the total number of data points along each Cartesian coordinate, P_y and P_z the number of processors along *y* matrix and *z* matrix directions, with $P = P_y \times P_z$ being the total number of processors used. The data domain is decomposed into pencils (Fig. 1a) with each processor handling vectors in the *x* matrix direction. The processor ID number increases along the *y* matrix direction first, and then the *z* matrix direction. At this first step, the *x*-, *y*-, *z*- directions of the data coincide with the *x* matrix, *y* matrix and *z* matrix directions of the 3D matrix data. In the figure, there are a total of 16 processors, 4 along each of the *y* and *z* matrix directions. Although any number of processors along these two directions could have been used, for better load balance we used 2^n processors because the number of data points is also multiple of 2^n. The 2D decomposition of 3D FFT is performed in five steps:

Step 1, *1D FFT in the x direction.* Each processor performs in parallel a total of $(N_y/P_y) \times (N_z/P_z)$ 1D real-to-complex FFTs in the *x*-direction (Fig. 1b). Two *y*—*z* planes of data are added to yield $(N_x + 2) \times N_y \times N_z$ data matrix to store $(N_x/2 + 1) \times N_y \times N_z$ complex numbers resulting from the real-to-complex FFT in the *x* direction. The two elements of a complex number (its real and the imaginary component) are saved contiguously along the *x* matrix direction.

Step 2, *Transpose data between x and y matrix directions.* The data is transposed between *x* and *y* matrix directions following the *cyclic* communication strategy discussed by Dmitruk et al. [21]. This prepares for the 1D FFT in the *y* direction by placing each *y* data vector into a given processor. It is important to note that this transpose operation only involves data communication within the *x*—*y* slabs (Fig. 1c). The communications for different *x*—*y* slabs can be performed in parallel. The processors at each slab are locally numbered from 0 to $(P_y - 1)$, and then the transpose within each slab is accomplished in two stages. First, blocks of data of dimension $(N_x/P_y) \times (N_y/P_y) \times (N_z/P_z)$ (note that $N_x = N_y = N_z = N$) are exchanged among processors within a slab using temporary arrays. In this paper we will assume $N = N_x = N_y = N_z$ for convenience as we will use our subroutine in a pseudospectral code to model isotropic homogenous turbulence. However, the scheme could be extended to the case of $N_x \neq N_y \neq N_z$. Second, the procedure is completed by locally rearranging the blocks when copying them from the temporary arrays to the main array in a way that the *y*-direction of the data is stored along the *x* matrix direction of the local-processor array.

Fig. 2 illustrates the communication algorithm for the data blocks in a given *x*—*y* slab. At the top left corner of the figure are shown the data blocks used in the transposition. All of them, except the ones along the last column *d* in the

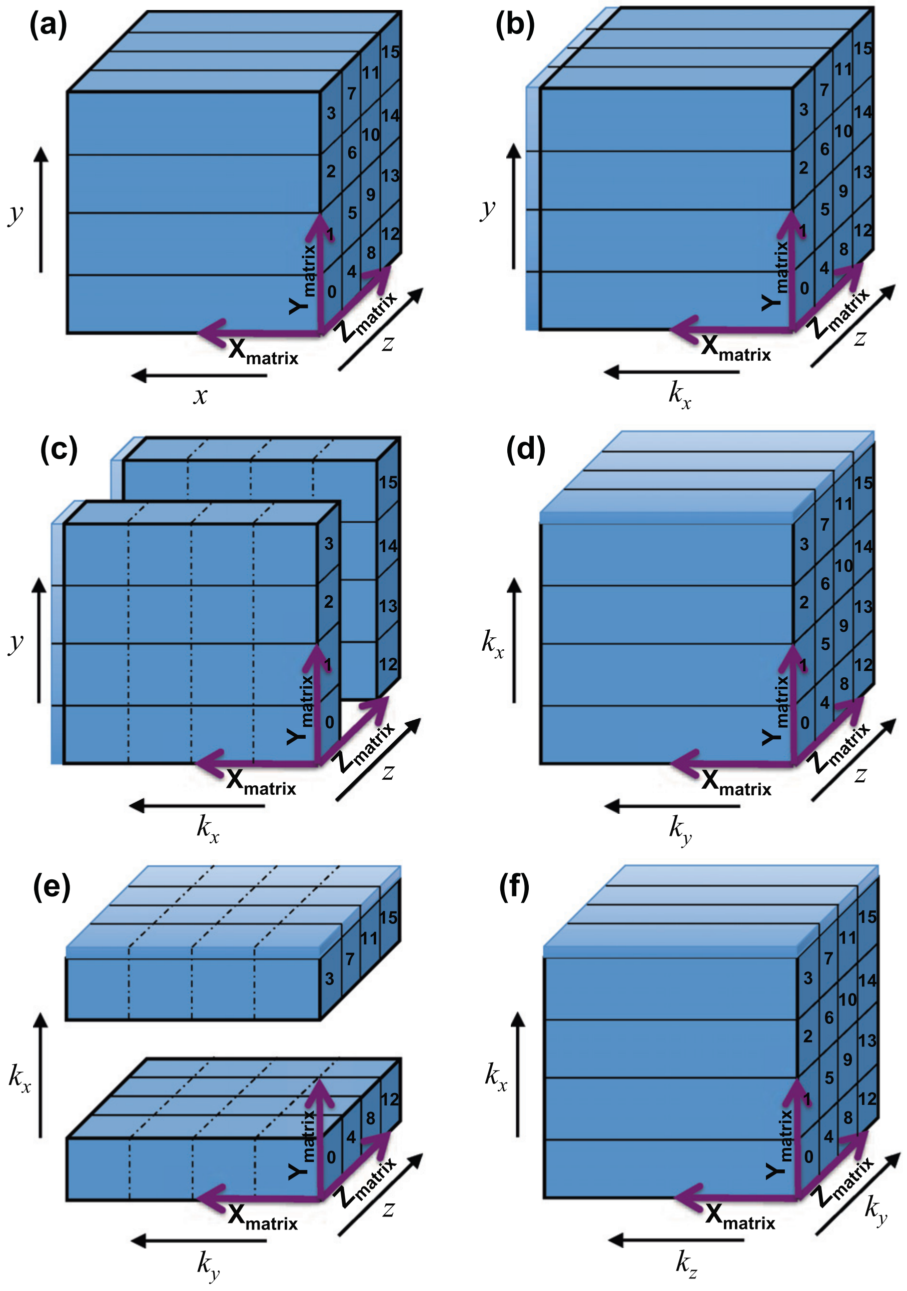

Fig. 1. Illustration of the steps involved in the 3D real to complex FFT using a 2D decomposition strategy: (a) real array, (b) 1D real to complex FFT along x, (c) Transpose between x and y directions, (d) 1D complex to complex FFT along y, (e) Transpose between y and z directions, and (f) 1D complex to complex FFT along z. The computational domain in this example is sliced into four parts along y and z directions, corresponding to sixteen processors in total.

x matrix direction, are of size $(N_x/P_y) \times (N_y/P_y) \times (N_z/P_z)$, while the blocks in the column d are of size $[(N_x/P_y)+2] \times (N_y/P_y) \times (N_z/P_z)$ in this x—y transpose. Since the diagonal data blocks a0, b1, c2, and d3 only need to be transposed within a same processor, there are $(P_y - 1)$ blocks of data being passed from each processor to the others. Therefore, $(P_y - 1)$ communication steps are required, with each step transferring data of size $N^3/(P_y^2 P_z)$. The data communications are ordered through a cyclic permutation of the communicating processors, as sketched per the order shown in Fig. 2a–c. This procedure ensures that all processors will be sending and receiving one block during each communication step. At the first communication step, each processor sends a data block to the nearby right processor and receives another data block from the nearby left processor. The data move from b0 to a1, c1 to b2, d2 to c3, and a3 to d0 (Fig. 2a). For the second communication step, each processor sends a block to the

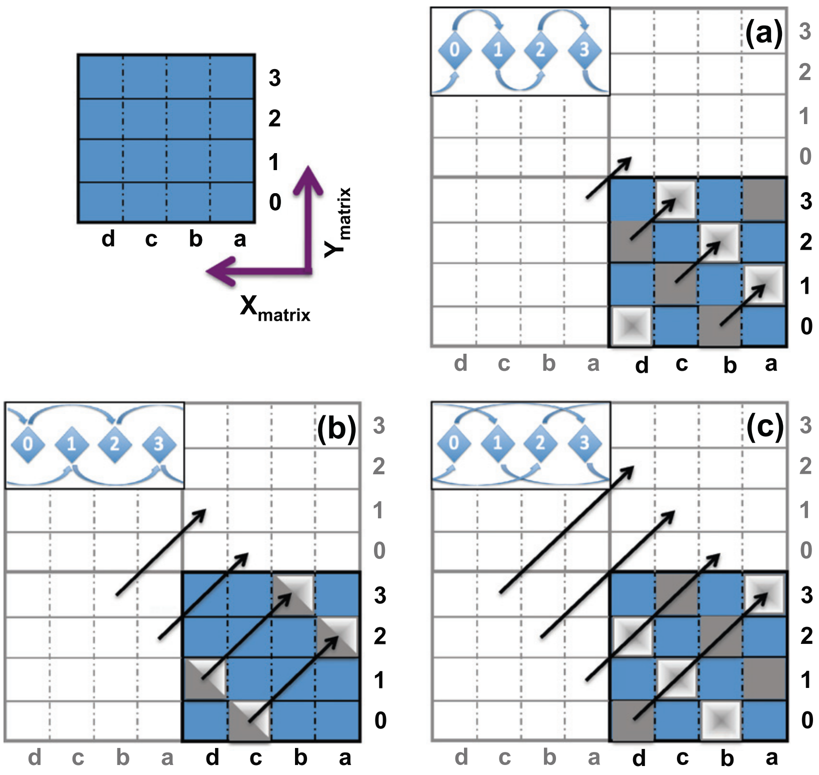

Fig. 2. Illustration of the communication steps in the *x*—*y* transpose algorithm in a given slab shown in Fig. 1. The top left corner shows the data blocks involved in the transposition. (a)–(c) show the data blocks transmitted at each of the three communication steps when there are 4 processors in the *y* direction (the grey lines are mirror images of the data blocks, dark grey blocks are being sent, white blocks are being received). At the top left corner of each figure is shown the processor communication pattern occurring at the same step.

second processor to its right and receives from the second processor to its left. The data move from c0 to a2, d1 to b3, a2 to c0, and b3 to d1 (Fig. 2b). For the last communication step, each processor interacts with the third processor to its right and left. The data move from d0 to a3, a1 to b0, b2 to c1, and c3 to d2 (Fig. 2c). This communication ordering can be extended to any numbers of processors. After all the block data deliveries are completed, the transpose is completed by converting the data inside each processor following $h(k_x, y, z) \rightarrow h'(y, k_x, z)$.

Step 3, *1D FFT in the y direction.* A total of $[(N_x/2+1)/P_y] \times (N_z/P_z)$ 1D complex-to-complex FFTs are performed in each processor along the *y* direction of the data (Fig. 1d). Care must be taken when the 1D FFTs are executed along the *x* matrix direction of the array because the real and imaginary parts of each complex data element are stored next to one another in the *y* matrix direction (as a result of the transposition).

Step 4, *Transpose data between y and z directions.* The data is then transposed between the *y* (*i.e.*, *x* matrix direction) and *z* directions (Fig. 1e) in a similar fashion as in Step 2, to place the vector data in *z* along the *x* matrix direction.

Step 5, *1D FFT in the z direction.* Finally, each processor performs a total of $[(N_x/2+1)/P_y] \times (N_y/P_z)$ 1D complex-to-complex FFTs along the *z* direction (but stored along the *x* matrix direction). Same caution, as in Step 3, should be exercised because the real and the imaginary parts of each complex data element are stored contiguously but along the *y* matrix direction.

At the end of Step 5, k_x, k_y, and k_z are stored along the y, z, and x matrix directions, respectively, which we shall referred to as the *transposed order*. This mismatch in the transformed space is allowed, as long as the wavevector components are specified in the same manner accordingly.

3. Complexity analysis

To better understand the scalability data and to forecast the parallel performance of our code for any values of problem size (N) and the number of processors (P), we shall conduct a complexity analysis here. Dmitruk et al. [21] showed that the total wall-clock time is the sum of the computational time and the communication time. Following their analysis for 1D decomposition of the 3D FFTs, we develop an approximate quantitative model for the execution time below. By extending the analysis of Dmitruk et al. [21], we find that the computational wall-clock time can be expressed as

Table 1
Estimated elemental times on Bluefire for the complexity analysis.

t_c	0.37 ns/FLOP
t_a	6.37 ns/word
t_w	2.19–11.60 ns/word
t_s	7.00 μs

$$T_{\text{Comp}} = \frac{5}{2}\frac{N^3\log_2(N^3)}{P_yP_z}t_c + \left\{2(P_y-1)\left[\frac{N^3}{P_yP_z}\right]\frac{1}{P_y} + \left[\frac{N^3}{P_yP_z}\right]\right\}t_{a_{XY}} + \left\{2(P_z-1)\left[\frac{N^3}{P_yP_z}\right]\frac{1}{P_z} + \left[\frac{N^3}{P_yP_z}\right]\right\}t_{a_{YZ}} \tag{1}$$

where t_c is the computation time per floating point operation, t_a is the memory-to-memory copy time per word. In the above equation we neglected the effect of the two additional layers used to store the complex number matrix which is a good approximation for large problem sizes N. The first term on the right hand side of Eq. (1) is the actual computation time to perform the three sequences of one-dimensional FFTs. Based upon standard operation counts for the FFT [29], the number of floating point operations (FLOP) is $(5/2)N^3\log_2(N^3)$. The other two terms account for the time spent on copying data to temporary arrays to be communicated: the second term corresponds to the x—y transposition and the third to the y—z transposition. In the subroutine, for each transposition, there are three copying steps for each communication step: first, a copy of data of size $[N^3/(P_yP_z)](1/P_y)$ (or $[N^3/(P_yP_z)](1/P_z)$ for the $y-z$ transpose) from the main array to a small temporary array to be transmitted; second, a copy of data received (same size as transmitted data) to another temporary array. Recall that these two copies occur (P_y-1) times (or (P_z-1) times for the y—z transpose), see steps 2 and 4 in Section 2. And a final copy after all communications are done from the last temporary array of size $[N^3/(P_yP_z)]$ to the main array for the in-processor transposition operation. The time to execute copies might be larger than the other times for the execution of the rest of the subroutine but it is a penalty that has to be paid to perform all the communication steps because the data to be transmitted should be contiguous.

The communication time to perform a 3D FFT can be written as

$$T_{\text{Comm}} = 2(P_y-1)\left[\frac{N^3}{P_yP_z}\right]\frac{1}{P_y}t_{w_{XY}} + 2(P_y-1)t_{s_{XY}} + 2(P_z-1)\left[\frac{N^3}{P_yP_z}\right]\frac{1}{P_z}t_{w_{YZ}} + 2(P_z-1)t_{s_{YZ}} \tag{2}$$

where t_w is the time for transmission of a single word between processors, and t_s is the startup or latency time. The first term on the right hand side corresponds to the communication time during the x—y transposition while the second term corresponds to the y—z transposition. The communication time is composed by the transmission time and the latency time. The total duration of transmission depends on the number of repeated communications (either (P_y-1) or (P_z-1)) and the data size ($[N^3/(P_yP_z)](1/P_y)$ or $[N^3/(P_yP_z)](1/P_z)$) being transmitted. The factor 2 in the transmission term corresponds to the two steps process during data transmission: (1) send to memory buffer, and (2) receive from memory buffer. The total latency time only depends on the number of repeated communications.

The primary motivation of this paper is to develop a 2D decomposition strategy for 3D FFT to be used for PetaScale computers, such as the upcoming Blue Waters. Eqs. (1) and (2) give the total theoretical execution time. In them, the memory-to-memory copy time per word (t_a), the time for transmission of a single word between processors (t_w), and the startup or latency time (t_s) are set to different values for the x—y and y—z transpositions. However, as we will show later, for a large number of processors P, $t_{a_{XY}} \approx t_{a_{YZ}} \approx t_a$, $t_{w_{XY}} \approx t_{w_{YZ}} \approx t_w$, and $t_{s_{XY}} \approx t_{s_{YZ}} \approx t_s$. For such a case, they both can be simplified, thus the total theoretical execution time becomes

$$T_{\text{Total}} = \frac{5}{2}\frac{N^3\log_2\left(N^3\right)}{P}t_c + 6\frac{N^3}{P}t_a + 4\frac{N^3}{P}t_w + 2(P_y+P_z-2)t_s, \tag{3}$$

which, once again, applies for large problem sizes N and large number of processors P. To evaluate the theoretical performance of the FFT algorithm, the elemental times in Eq. (3) (t_c, t_a, t_w, and t_s) have to be determined first. The reader can find in the appendix the discussion on how to obtain these times which we show in Table 1 This elemental times belong to the supercomputer Bluefire at NCAR, an IBM Power 575 cluster.

4. Testing and evaluation

4.1. Detailed analysis of the subroutine

The strategy was implemented on Bluefire at NCAR, an IBM Power 575 cluster (4064 POWER6 processors running at 4.7 GHz, bandwidth of 20 GBps each direction, switch latency of 1.3 μs at peak performance). For comparisons, we also run on the same computer the subroutine developed by Pekurovsky [23] based on 2D decomposition, as well as the 1D decomposition code of Dmitruk et al. [21]. We measured the average wall-clock time for a pair of forward and backward FFTs, by repeating the pair FFTs at least 50 times and dividing the total wall-clock time by the number of repeats. For the in-processor 1D FFT computation, we used the publicly available FFTW by Frigo and Johnson [28]. In Fig. 3a we show the

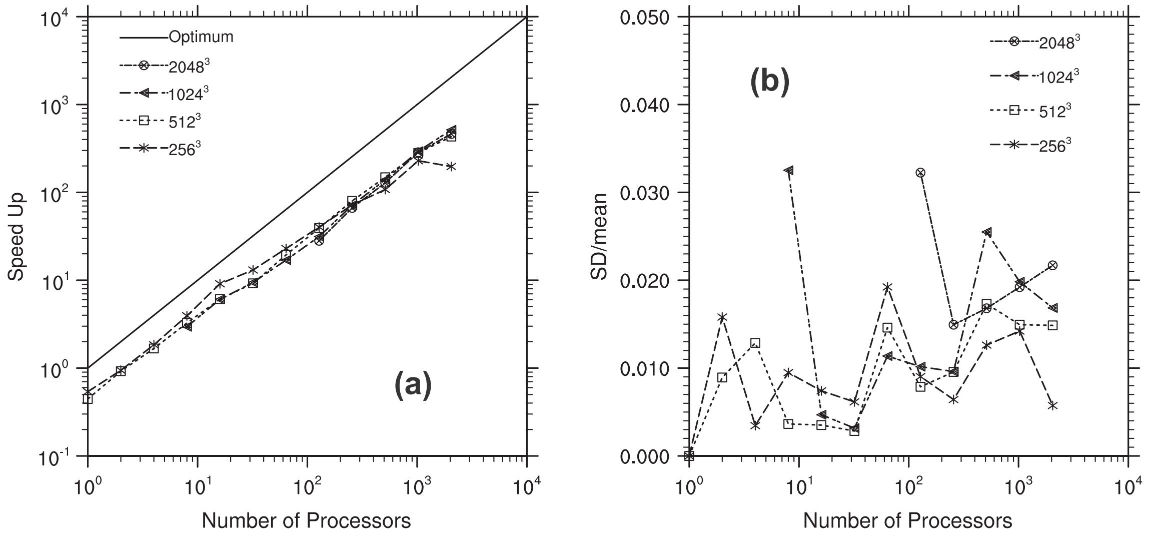

Fig. 3. (a) The speedup factors for different problem sizes as a function of the number of processors. (b) The standard deviation of the processor execution times, normalized by the mean, for different problem sizes.

speedup S ($S = T_1/T_P, T_1$ is the wall-clock time on one processor and T_P is the wall-clock time on P processors) as a function of the number of processors (P), for 4 problem sizes from 256^3 to 2048^3 three-dimensional real data. We used a theoretical estimate for T_1 based on $(5/2)N^3\log_2(N^3)t_c$. Interestingly, all curves nearly collapse to a straight line in the log–log plot. Only for the smallest problem size (256^3) and larger number of processors, the speedup starts to deviate due to the increase in communication times (due to latency) relative to computational times. The latency problem occur because of the startup time a processor takes to engage in the transmission process of the data. It increases with the number of communications (which depends on the number of processors, P and $P_y\ P_z$) as seen in the last term of Eq. (3) while the actual transmission time decreases with P as the data to transmit become smaller (3rd term in Eq. (3)). The latency problem hurts the scalability of parallel codes and will surface when latency time is larger than the actual transmission time (i.e., the 3rd term in Eq. (3) is larger than the 4th term). Approximating $P_y \approx P_z \approx \sqrt{P}$ and for large P, this happens when $P > N^2(t_w/t_s)^{2/3}$. Thus, for $N = 256$, latency is an issue for $P > 302$, or larger than 512 processors for our runs because we use 2^n processors for the tests. For $N = 512$, P should be larger than 2048 processors. This is the reason we do not see deviation from the good scalability for problem sizes larger than $N = 512$.

It is important to point out that the wall-clock time t_{ave} averaged over all processors was used for Fig. 3a instead of the traditional actual wall clock time (which corresponds to the maximum wall-clock time among all processors $t_{\max}$). We chose to use this average time and the standard deviation together to monitor the load balance of the subroutine. The more unbalanced a parallel code is, the larger the difference between $t_{\max}$ and t_{ave} and this difference is reflected in the standard deviation of the set of wall-clock times from the P processors which we show in Fig. 3b. We found that it is less than 4% of the mean which indicates that our proposed subroutine balances the computational load appropriately among all processors. In addition, it shows that small differences are found between $t_{\max}$, and t_{ave}, thus either of them can be used as a good approximation to the real wall-clock time.

To better understand how the communication time is compared to the computation time, in Fig. 4, we show the relative percentages of the time used for the communication (Send and Receive), computation, and the copying needed solely for the purpose of pre- and post- processing in the data communication section of the subroutine. Fig. 4 demonstrates that, for all problem sizes considered, the computational time and copying time decrease with P. The communication time increases with P, and eventually dominates the wall-clock time. For 256^3, the communication time reaches 70% for $P = 2048$ and at this point the latency starts to dominate. This further explains the poor speedup and efficiency for this problem size. The transition from computation dominating to communication dominating occurs at larger P as the problem size is increased. Similarly, latency problems could be neglected if it represents less than 1% of the total communication time. This occurs for $P_0(\sqrt{P_0} - 1) = (t_w/t_s)N^3/99$ (P_0 is the number of processor below which latency is negligible). For 1024^3 and 2048^3, P_0 is 256 and 1024 processors respectively (we use 2^n processors). If the latency is negligible, using Eq. (3), the theoretical relative percentages of the time used for the communication, computation, and the copying are:

$$\%_{\text{Comp}} = \frac{1}{1 + \frac{12}{15}\frac{1}{\log_2(N)}\frac{t_a}{t_c} + \frac{8}{15}\frac{1}{\log_2(N)}\frac{t_w}{t_c}}$$

$$\%_{\text{Copy}} = \frac{1}{\frac{15}{12}\log_2(N)\frac{t_c}{t_a} + 1 + \frac{2}{3}\frac{t_w}{t_a}} \tag{4}$$

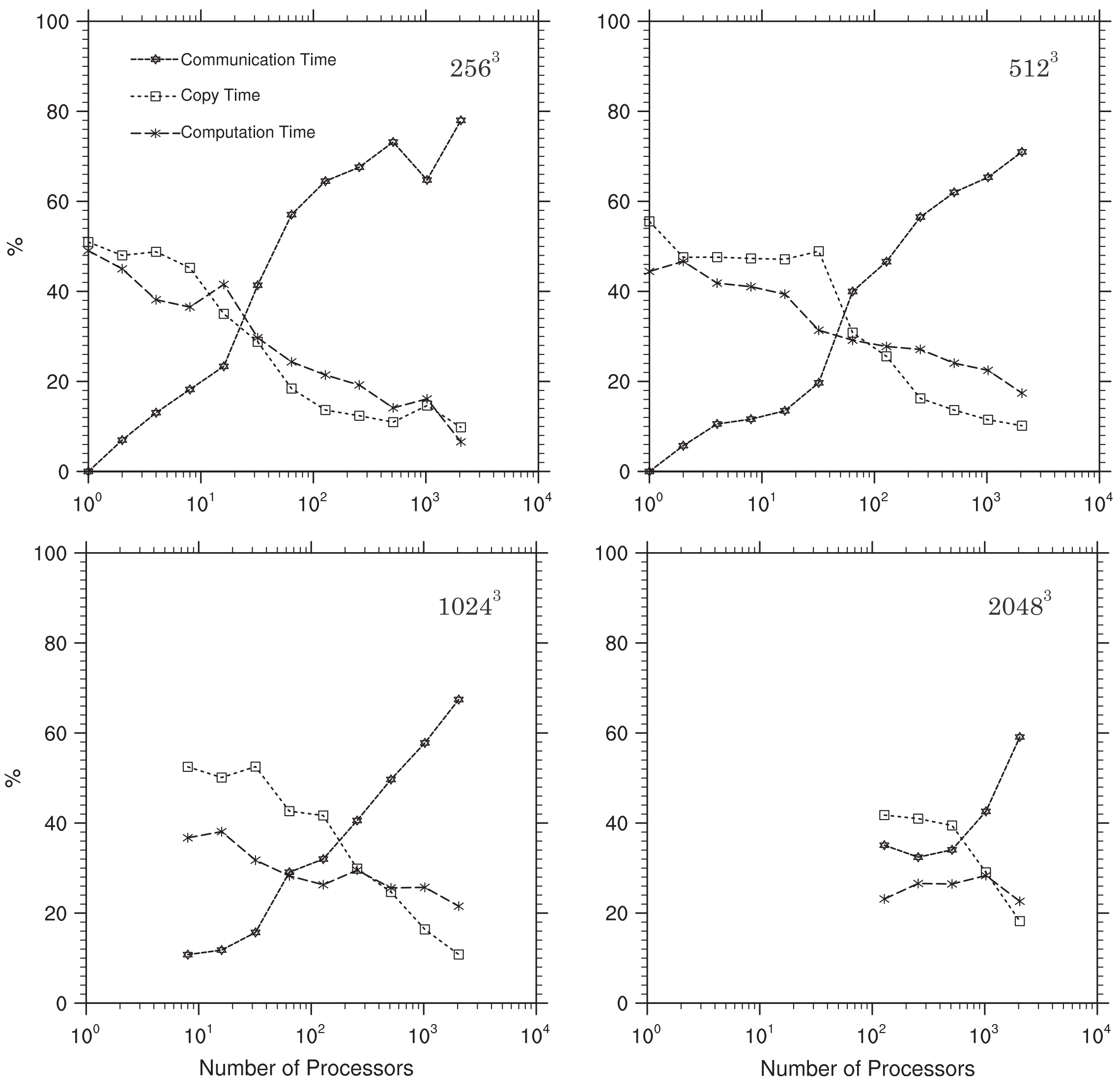

Fig. 4. The relative percentages of the wall-clock time spent for different sections of the code, for different problem sizes.

Table 2
Theoretical relative percentage of communication, computation, and copying with P_0.

P_0	1024^3 256	2048^3 1024
$\%_{\text{Comp}}$	24.69%	26.51%
$\%_{\text{Copy}}$	34.01%	33.19%
$\%_{\text{Comm}}$	41.29%	40.29%

$$\%_{\text{Comm}} = \frac{1}{\frac{15}{8}\log_2(N)\frac{t_c}{t_w} + \frac{3}{2}\frac{t_a}{t_w} + 1}$$

The theoretical prediction for 1024^3 and 2048^3 with P_0 are shown in Table 2 which are reasonablely close to the values shown in Fig. 4. Note that Eqs. (4) show no P dependence. Recall that the complexity analysis was designed for large problem sizes and large number of processors. Thus, for $P < P_0$, Eqs. (4) will predict poorly. On the other hand, for $P > P_0$, latency start to become important. The relative percentage of the execution time due to latency during communication is:

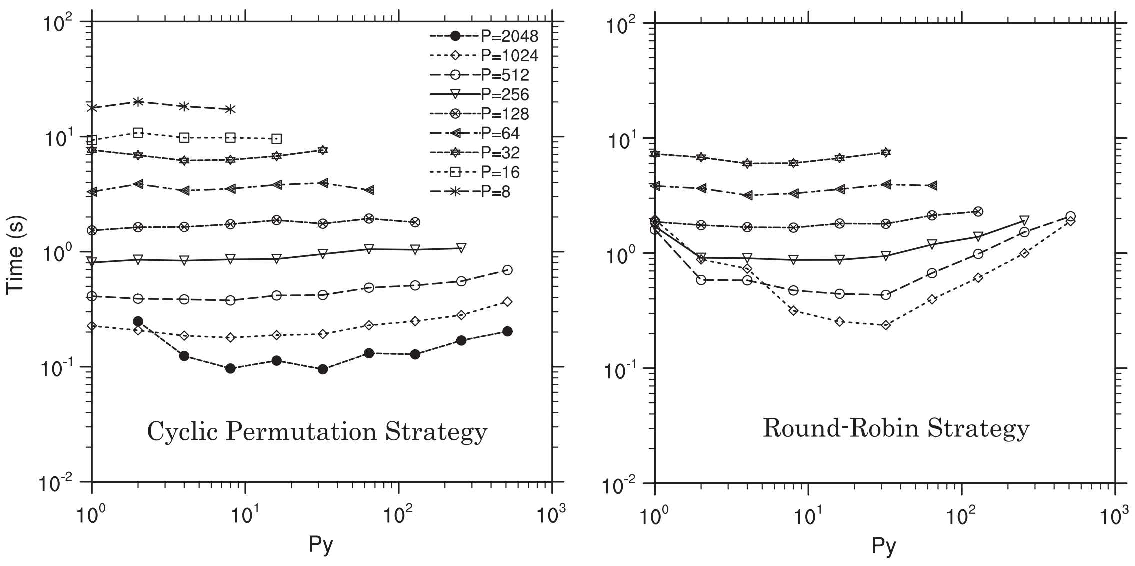

Fig. 5. The measured average execution times of different communication strategies for different processor distribution. The problem size is 1024^3.

$$\%\mathrm{Comm}_{\mathrm{LATENCY}} = \frac{P(\sqrt{P}-1)}{\frac{15}{8}N^3\log_2(N)\frac{t_c}{t_s} + \frac{3}{2}N^3\frac{t_a}{t_s} + N^3\frac{t_w}{t_s} + P\left(\sqrt{P}-1\right)} \tag{5}$$

From this, the increase on the $\%\mathrm{Comm}_{\mathrm{LATENCY}}$ for P larger than P_0 is:

$$\mathrm{Increase}_{\%\mathrm{Comm}_{\mathrm{LATENCY}}} = \frac{K+1}{\frac{KP_0}{P^{3/2}}+1} \tag{6}$$

where

$$K = \left[\frac{15}{8}\log_2(N)\frac{t_c}{t_s} + \frac{3}{2}\frac{t_a}{t_s} + \frac{t_w}{t_s}\right]N^3P_0^{-3/2}$$

which shows how rapidly the influence of latency on execution time grows with number of processors. For instance, just by increasing the number of processors by a factor of 4 from P_0, $\%\mathrm{Comm}_{\mathrm{LATENCY}}$ increases by ~8; and by increasing P by a factor of 8, $\%\mathrm{Comm}_{\mathrm{LATENCY}}$ increases by ~22. Thus, for $P > P_0$ the total execution time is dominated by communication.

Another important aspect of 2D domain decomposition is the nature of processor distribution. For a given P, there are different choices of P_y and P_z. Fig. 5 shows how the average wall-clock time changes with P_y when $P = P_y \times P_z$ is fixed, for the 1024^3 problem size and different communication methods. In Section 2, we discussed the cyclic permutation strategy for the communication. We also tested a round-robin strategy, discussed in Dmitruk et al. [21], for our 2D decomposition. The round-robin methodology enforces pairwise communication, where each processor is sending to the same processor from which it is receiving. For a given P, the order of round-robin pairwise communications is specified by a table. This is similar to what is done in round-robin tournaments. A comparison of the cyclic and pairwise communications is portrayed in Fig. 5. Pekurovsky [23] suggested that the performance is generally better when P_y is smaller than P_z and is made equal to or less than the number of processors on a node (32 for Bluefire) for a fixed P. This occurs for the round-robin strategy especially for large number of processors P. The communication time among processors changes on Bluefire depending on whether the communicating processors are located within a same node or not. Thus similar execution times are found regardless of the processor distribution ($P_y \times P_z$) when the total number of processors is close to 32. Furthermore, for the cyclic permutation, this behavior remains for even larger total number of processors. In this strategy the communication is performed in a more orderly fashion starting from nearby processors. In the round-robin scheme, the processors might be interchanging data with processors within a same node or across nodes at different communication steps, leading to a stronger dependence of performance on the processor distribution. We also examined the load balance in Fig. 6 through the standard deviation of the times used by different processors. The cyclic permutation performs better than the other schemes, as the standard deviation is lower than 5% for any processor distribution. Note that the round-robin strategy has larger than 10% deviations for large numbers of processors, thus this scheme is less likely to balance the computational load among processors as the communication strategy is less suitable for the Bluefire networking architecture. By using this scheme, a list of processor pairs determine which processors communicate at each communication step (during a y—z transpose); thus depending on that list, there might always be some pairs with processors located far apart in the network, leading to imbalance in communication times. On the other hand, for the cyclic permutation scheme, the communication is

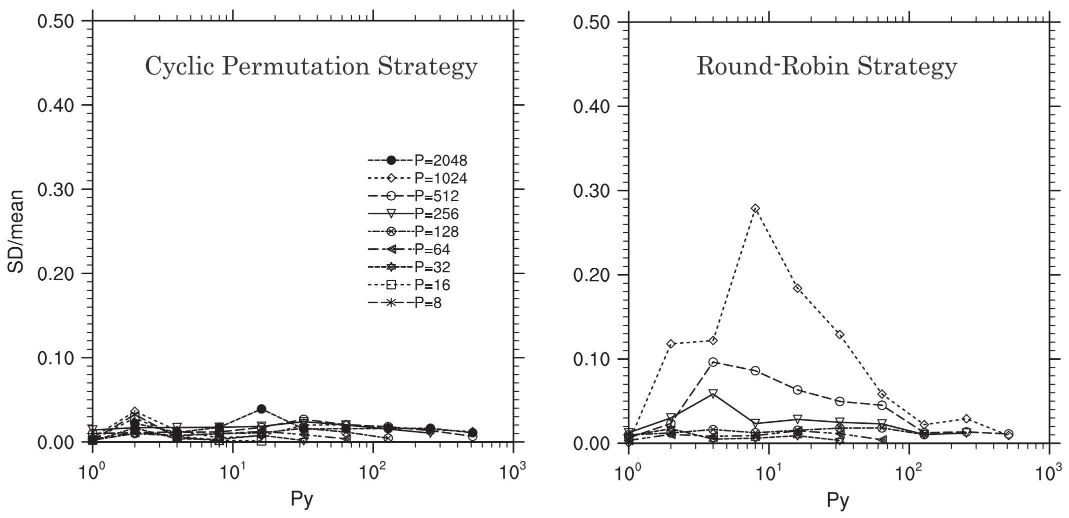

Fig. 6. Standard deviation of the processor execution times for different communication strategies and different processor distribution. The problem size is 1024^3.

typically done, for a given communication step, by processors separated at a fixed distance in the network, yielding a better communication load balance. Thus, cyclic permutation strategy adjust better to the network than round-robin strategy and therefore is chosen in this work.

We performed different experiments for different problem sizes using the cyclic permutation scheme to measure the computation time (first term in Eq. (1)), the copying time (second and third terms in Eq. (1)), and the communication time (Eq. (2)). We show the measured times for the case of 1024^3 problem in Fig. 7, similar trends were found for other problem sizes. In the figure, the times are measured for one forward real-to-complex 3D FFT and one backward complex-to-real 3D FFT. For each FFT, there are three 1D FFTs (one real-to-complex and two complex-to-complex) and two transpose operations. The actual computation times for 1D FFTs are similar regardless of whether it is a transformation from real to complex, complex to real, or complex to complex.

It is expected that the communication time for performing the x—y transposition differs from that of the y—z transposition, as the processors were distributed following the y matrix direction first. Consequently, when executing x—y transposition, many of the communicating processors within a given slab are within a same node allowing faster communication. In contrast, the communicating processors during the y—z transpose are more likely to be located among different nodes, which increases the communication time. In Fig. 7 we also notice that as the number of processors increases, the communication time is similar regardless of whether it is the x—y transposition or y—z transposition. This is because, for large number of processors, the number of processors in the x—y transposition slab are larger than 32 (the number of processors within a single node on Bluefire), thus the communicating processors might be located at different nodes, similar to what happens during the y—z transposition. Therefore, for large number of processors it could be assumed that $t_w \approx t_{w_{XY}} \approx t_{w_{YZ}}$, and $t_s \approx t_{s_{XY}} \approx t_{s_{YZ}}$ as it was done to derive Eq. (3).

Concerning the copying time, the times are different for small numbers of processors. The copying time during the x—y transposition is faster because we are transposing between the first and the second dimensions (x and y matrix directions) of the array. In Fortran, coping data along the first dimension runs faster as the data are adjacent to one another in the memory. For larger numbers of processors, this time is similar no matter what transposition is being done. In this case, the local arrays within each processor are smaller, thus allocating memory for copying purposes is faster as they are close in the memory layout. Thus, for large number of processors it could be assumed that $t_a \approx t_{a_{XY}} \approx t_{a_{YZ}}$ as it was done to derive Eq. (3).

4.2. Comparison with available subroutines and on different supercomputers

In Fig. 8, we plot the average wall-clock time as a function of the number of processors (P), for 4 problem sizes from 256^3 to 2048^3 of three-dimensional real data. We compare our proposed approach with the 2D domain decomposition schemes by Pekurovsky [23] (p3dfft), Takahashi [24], and Li and Laizet [25] (www.2decomp.org). The tests were performed in Bluefire.

Fig. 8 shows that the 2D decomposition strategy developed and implemented here has a better performance than other available implementations but close to Pekurovsky's scheme. The proposed approach was able to run on smaller number of processors than the other implementations. This is an advantage of our strategy for using MPI_ISend plus MPI_IRecv commands instead of MPI_AlltoAll (or MPI_AlltoAllv) as others used. Our subroutine needs roughly 2 times more memory only while the others need at least 3 times more memory for temporal arrays forced by the MPI_AlltoAll command usage. This

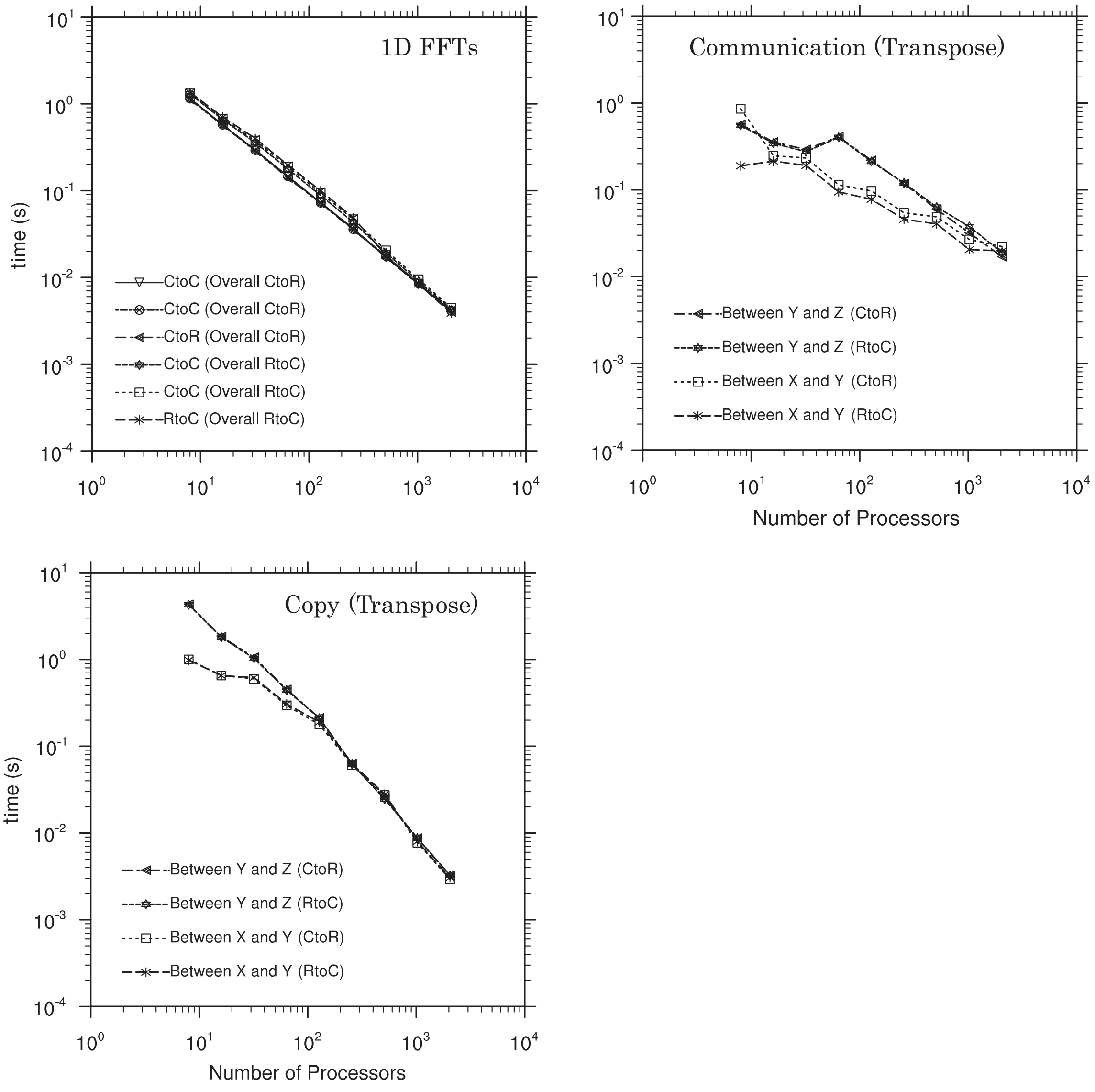

Fig. 7. The average measured execution times for each subroutine section as a function of the number of processors used, for 1024^3 problem size. Similar results were found for other problem sizes.

will also affect time of computation because its execution is slower when the memory is busier. This is an important issue when trying to use the FFT subroutines in other codes such as pseudo spectral codes.

In Table 3 there is a closer comparison of our proposed approach against Pekurovski's subroutine for larger problem sizes. We show the ratio the speed-up of our subroutine to the speed-up of Pekurovski's subroutine. We observe that for the cases shown (larger problem sizes and large number of processors), our strategy is 28% faster (on average) than Pekurovski's scheme. Note that for most of those cases, P is larger than P_0 for which the communication is more dominant in the total execution time. Thus, the use of MPI_ISend plus MPI_IRecv commands yields a better performance than the MPI_AlltoAll command.

In Table 4 we confirm this last point. We performed experiments using MPI_ISend plus MPI_IRecv commands and the MPI_AlltoAll command for 32 processors to avoid latency issues and to be able to compare intranode communications and internode communications. For intranode communications we can only use up to 32 processors (Bluefire has 32 processors per node). For internode communications, we used 32 nodes with only one processor per node. In Table 4 we show the speed-up ratio of MPI_ISend+MPI_IRecv communication strategy to MPI_Alltoall communication strategy. Our MPI communication strategy is faster for message sizes larger than 1Kbytes. Therefore, our proposed approach is better when the communication is more dominant and for message sizes lager than 1 Kbytes. The message size being transmitted could be estimated by $\text{MessageSize} = 8N^3/P^{3/2}$ (the factor 8 corresponds to double precision elements), thus the larger number of

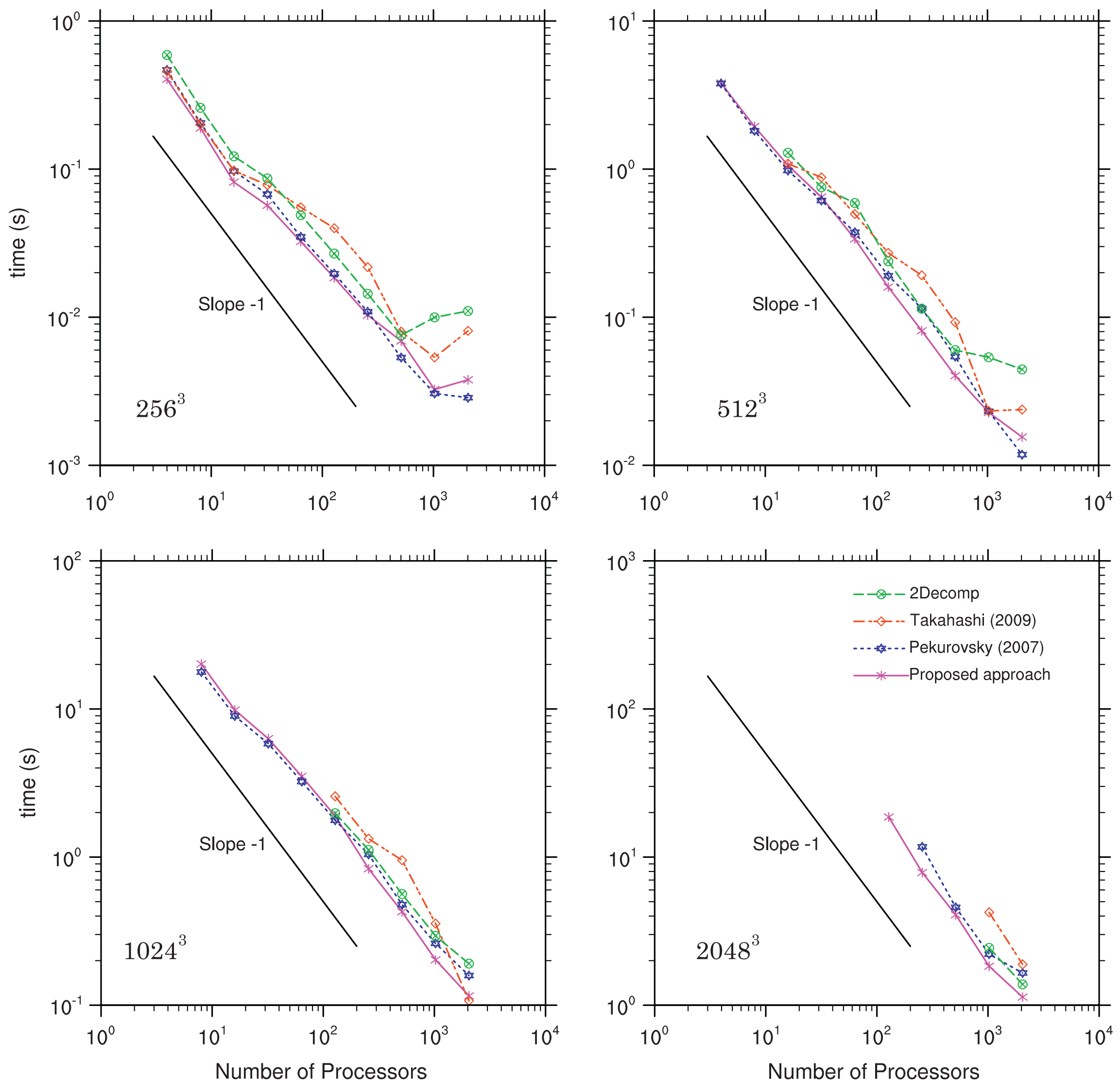

Fig. 8. Execution time as a function of P for different problem sizes and different parallel 3DFFT strategies.

Table 3
Speedup ratio between the proposed subrotine and Pekurovski's subroutine for larger problem sizes which imply large message sizes at time of communication.

Number of processors	1024^3	2048^3
256	1.252	1.491
512	1.116	1.123
1024	1.292	1.207
2048	1.372	1.456

processors up to which MPI_ISend+MPI_IRecv communication strategy performs better is $P_1 = (8/1024)^{2/3}N^2$. For problem sizes of 1024^3 and 2048^3, P_1 is 32768 and 131072 respectively (we use 2^n processors). In general, the proposed 2D domain decomposition strategy for 3DFFT is better than any other available subroutine for $P_0 < P < P_1$.

We tested our proposed subroutine on different supercomputers, including: Hopper (http://www.nersc.gov/systems/hopper-cray-xe6/) from the National Energy Research Scientific Computing Center (NERSC, the UC Oakland Scientific Facility in Oakland, California), Kraken [4] (http://www.nics.tennessee.edu/computing-resources/kraken/) from the National Institute for Computational Sciences (NICS) at the University of Tennessee, and Ranger (http://www.tacc.utexas.edu/resources/hpc/

Table 4
Speedup ratio between an MPI_ISend+MPI_IRecv communication strategy and an MPI_Alltoall communication strategy using 32 processors.

Message size	Internode	Intranode
2 bytes	0.190	0.296
16 bytes	0.243	0.294
128 bytes	0.385	0.496
1 Kbyte	0.769	1.274
8 Kbytes	1.148	1.573
64 Kbytes	1.222	1.435
512 Kbytes	4.478	3.598

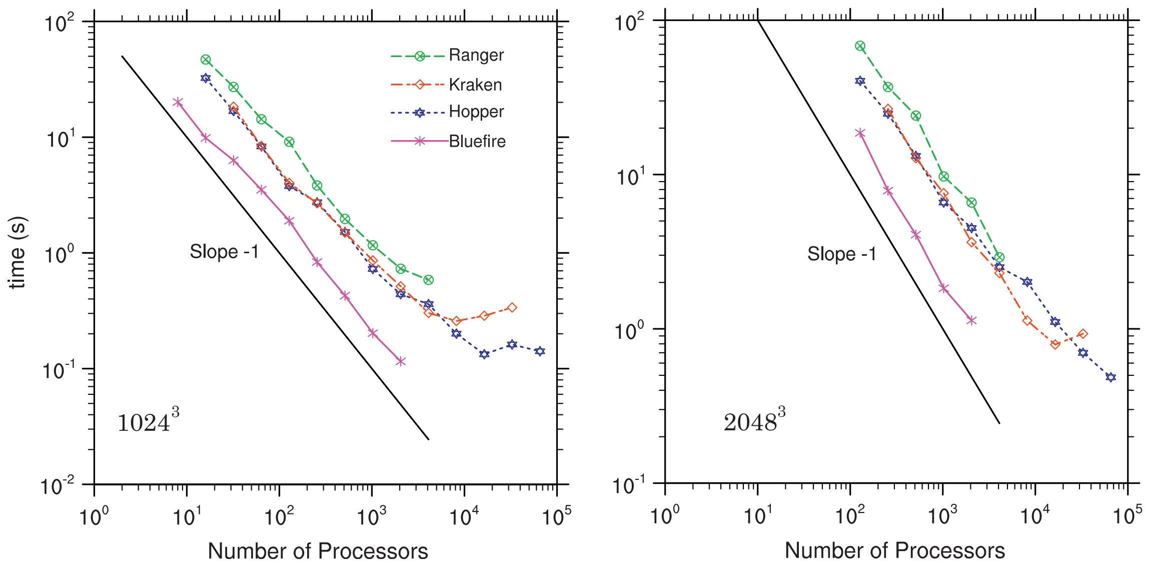

Fig. 9. Execution time as a function of P for different problem sizes in different supercomputers.

) from the Texas Advanced Computing Center (TACC) at The University of Texas at Austin. They are ranked by www.top500.org at 16th, 21st, and 40th respectively in the top 500 fastest supercomputers list. The comparison of the performance of our proposed subroutine in those machines is shown in Fig. 9. We found faster execution time in Bluefire because the IBM Power6 processors run at 4.7 GHz while in the other machines run at up to 2.6 GHz. New parallel machines are designed with slower processors but with a large amount of them. The supercomputers Hopper and Kraken are very similar in design (both are Cray machines with a 3D torus network), this is the reason the performance of the subroutine is very similar. We notice some saturation for the case of 1024^3 for number of processors larger than ~10,000. At these point, latency issues become important. Aside from this observation, there is a good scalability of our 2D domain decomposition 3DFFT subroutine for all the machines.

4.3. Complexity analysis study

With the estimates of the elemental times obtained in the Appendix and shown in Table 1, we can compare the results from theoretical equation to the actually measured timing results (Fig. 10). In general, the agreement between the two is satisfactory. For large problem sizes, two theoretical curves are plotted, with t_w set to 2.19 ns/word (solid line) and 11.60 ns/word (dash line), respectively. The dash line provides a better prediction for 2048^3 problem size.

Plimpton at Sandia National Laboratories presents another 2D decomposition strategy for complex-to-complex 3D FFTs (for details see Nelson et al. [22]). His strategy was to pack and send as much data as possible in a multistage algorithm using MPI_Send and MPI_Irecv commands. Here we present a complexity analysis for his strategy. The difference in this approach is that the data of size $(1/2)[N^3/P]$ are sent $\log_2 P_y$ times for x–y transpose and $\log_2 P_z$ times for y–z transpose. Thus assuming a large number of processors, the theoretical execution time is:

$$T_{\text{Total}} = \frac{5}{2}\frac{N^3\log_2\left(N^3\right)}{P_yP_z}t_c + \left\{2\log_2(P)\left[\frac{1}{2}\frac{N^3}{P}\right] + 2\left[\frac{N^3}{P}\right]\right\}t_a + 2\log_2(P)\left[\frac{1}{2}\frac{N^3}{P}\right]t_w + 2\log_2(P)t_s. \quad (7)$$

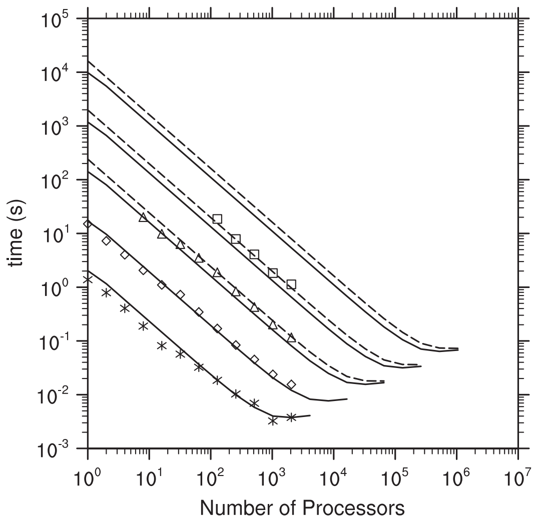

Fig. 10. Comparison between actually measured execution time and theoretical times from the complexity analysis for a backward and forward 3D FFT cycle. The symbols represent the measured timing data: ($*$) for 256^3, ($\diamond$) for 512^3, ($\triangle$) for 1024^3, and ($\square$) for 2048^3. The lines correspond to the theoretical prediction. For the dash lines, the value of t_a was estimated using 2048^3 problem size. For the solid lines, t_a was estimated using smaller problem sizes. The predictions for 4096^3 problem size are also shown.

Another strategy to parallelize the 3DFFT is to decompose the domain along all three Cartesian coordinates. For this 3D domain decomposition setup, the 1D Fourier transform could be split into smaller transforms recursively with each processor handling in parallel each of these smaller transforms (distributed FFT scheme); or, we could still use serial 1D Fourier transform (transposing data scheme). Pelz [20] claimed that for a small number of processors, distributed FFT performs better, while for a large number of processors transposing data is better. Since we are interested in using a large number of processors in Petascale machines, we will still keep the transposing data scheme for the 3D domain decomposition approach. In this case, a first step of data movement among certain processors is needed to convert the 3D domain decomposition layout into a modified 2D domain decomposition layout in order to allocate the data in such a way that serial 1D Fourier transforms can be completed locally along one of the Cartesian coordinates. After that, two data transposes are needed similar to what was explained in Section 2 for the 2D domain decomposition layout. In addition, there is a 4th communication step to have the Fourier transformed data converted from the modified 2D domain decomposition layout back to the 3D domain decomposition layout (in some cases, this last step could be avoided). We also developed a complexity analysis for this scheme to be able to theoretically compare it against our proposed subroutine:

$$T_{\text{Total}} = \frac{5}{2}\frac{N^3\log_2\left(N^3\right)}{P}t_c + 12\frac{N^3}{P}t_a + 8\frac{N^3}{P}t_w + 2\left(2P_x + P_y(P_x + P_z) - 4\right)t_s. \tag{8}$$

For a large problem size, 4096^3, we compare theoretically Plimpton's strategy, our approach presented here, Dmitruk et al.'s strategy and a 3D domain decomposition scheme by using the estimated elemental times (t_c, t_a, t_w, and t_s) for Bluefire (see Fig. 11). The 1D-decomposition implementation of Dmitruk et al. [21] is better than all other strategies because it only handles one transposition (thus, one communication step). It is known that the 1D-decomposition strategy is better than the 2D-decomposition strategy, Takahashi [24]; its drawback is that we can only use N number of processors which limits its use on Petascale machines. As far as the 3D-decomposition strategy, this is a more expensive one because of the 4 additional communication steps. Making it to saturate faster than any of the other strategies. In addition, our approach performs better than Plimpton's scheme for reasonable (for today) number of processors. For extremely larger number of processors (more than a few hundred thousand), the number of communication steps in our strategy is $(P_y - 1)$ (or $(P_z - 1)$) while for Plimpton's it is $\log_2 P_y$ (or $\log_2 P_z$), so ours is considerably larger for larger number of processors. This triggers a problem with the latency time in our methodology, which is not important in Plimpton's approach.

We also note that right before our strategy starts to saturate, a maximum speed up is obtained. Thus, by determining this maximum we could have a theoretical optimal number of processors below which our approach is the best. Setting the derivative of the theoretical speed up based on Eq. (3) to zero, we obtain:

$$P_{\text{Max}} = \left[\left(\frac{5}{4}\log_2\left(N^3\right)t_c + 3t_a + 2t_w\right)\frac{N^3}{t_s}\right]^{2/3} \tag{9}$$

Here we have assumed that $P_y = P_z \approx \sqrt{P}$. Using the theoretical estimates for the elemental times discussed above, we find that $P_{max} = 694,532$ for this 4096^3 problem size which is comparable to what is shown in Fig. 11. For this maximum number of processors P_{max} our proposed approach is still better than Plimpton's strategy.

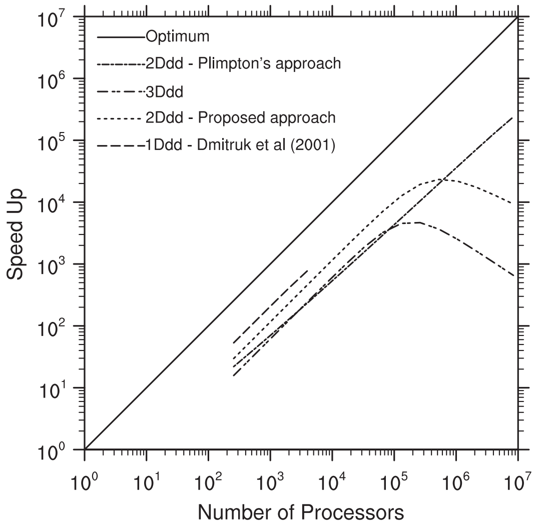

Fig. 11. Theoretical speed up factors of the different domain decomposition strategies using the transpose approach as discussed in this paper and Plimpton's scheme. The problem size is 4096^3

An additional advantage of our proposed approach is that it uses [4] MPI_Isend and MPI_Irecv which are non-blocking MPI commands while MPI_Alltoall is a blocking command. Therefore our subroutine could be modified to perform computations and communications at the same time in certain steps. Following the explanation of the five steps procedure in Section 2, steps 2 and 3 and steps 4 and 5 could be overlapped. After the first step is done, instead of transposing the whole $x-y$ slab of size N^3/P_y, we could transpose only a portion of $N^3/P_y/M$ (M is a number of subdivisions on the slabs to be transposed). After the first slab portion is transposed, we could overlap the second 1D FFT computation on this first slab portion with the transpose of a second slab portion. Then, overlap computations of the second slab portion with transpose of the third slab portion, and so on. For large number of processors, the dominant time is the communication time, with this overlapping scheme we could save an important portion of the computational time. The drawback of the scheme is that the latency increases with the number of subdivisions M as more communication substeps are taken. We developed a complexity analysis of this scheme to theoretically explore additional improvement that could be obtained by overlapping:

$$T_{\text{Total}} = \frac{5}{2}\frac{N^3\log_2\left(N^{(1+\frac{2}{M})}\right)}{P}t_c + 6\frac{N^3}{P}t_a + 4\frac{N^3}{P}t_w + 2(P_y + P_z - 2)Mt_s. \tag{10}$$

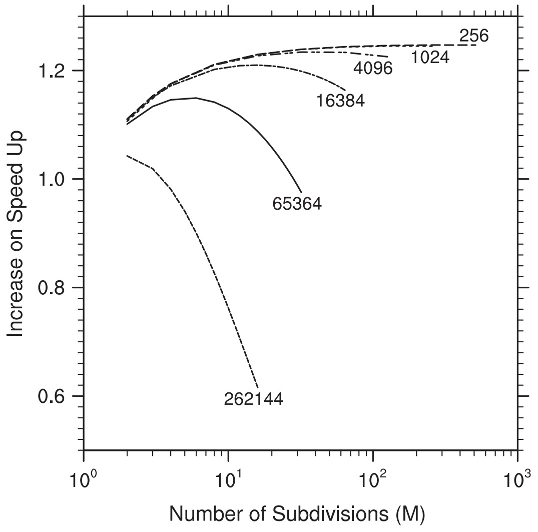

Fig. 12. Theoretical study on the increase on the speed up of the proposed approach by overlapping FFT computations with transpose communications for different number of processors (256, 1024, 4096, 16384, 65364, and 262144). The problem size is 8192^3.

Here we assume that the number of subdivisions M is the same for the x–y and y–z transposes. M should be less than N/P_y and N/P_z. Note that if $M = 1$, we recover the original complexity analysis Eq. (3). For a problem size of 8192^3, we studied the improvement of our subroutine with overlapping for different number of processors and a permitted range of subdivisions M (see Fig. 12). The scheme can improve the subroutine by at least 10% and up to ~25%. Similar trends were found for other problem sizes. However, we can see that for very large number of processors the overlapping scheme actually hurts the performance. The maximum number of processors could be estimated by determining the minimum execution time with respect to M for only 2 subdivisions. In doing so:

$$P_{\max_{\mathrm{IMPROVE}}} = \left(\frac{5}{16}N^3 \log_2(N)\frac{t_c}{t_s}\right)^{2/3}. \tag{11}$$

For 1024^3, $P_{\max_{\mathrm{IMPROVE}}}$ is 3156 (2048 using 2^n processors); and for 2048^3, $P_{\max_{\mathrm{IMPROVE}}}$ is 13,455 (8192 using 2^n processors). Hence, the overlapping scheme to speed up our subroutine is only effective for moderate number of processors.

5. Summary and conclusions

In this paper, we have described in details a parallel MPI implementation for 3D Fast Fourier Transform using 2D domain decomposition. Our approach differs from other available parallel 3D FFT with the same domain distribution strategy in the communication scheme used. We utilized and studied a cyclic permutation scheme with MPI_ISend and MPI_IRecv commands. We also developed a theoretical complexity analysis in order to predict the execution time for large problem size and large number of processors. The resulting analysis was found to be in good agreement with the measured timing data for all the problem sizes and numbers of processors tested. Our paper extends the results of Dmitruk et al. [21] for 1D decomposition to 2D decomposition. A linear scalability was obtained for large problem sizes and large number of processors.

We examined another communication method, a round-robin approach, to compare against our proposed approach. The cyclic permutation yielded a good and better load balance, with less than 4% fluctuations in wall-clock times among processors even for different data distributions among processors. Overall, the cyclic permutation scheme is less sensitive to how the data is divided and distributed among processors.

We also analyzed the percentages of the time used for computation, data transmission, and copying, and found that communication eventually dominates the wall-clock time but this happens later, i.e., with larger numbers of processors, for larger problem size. Latency is negligible for number of processors (P) less than P_0 ($P_0(\sqrt{P_0} - 1) = (t_w/t_s)N^3/99$) and it grows rapidly with P.

The wall-clock times in our proposed approach were better than other published parallel strategies. Our proposed approach is 28% faster (on average) than the strategy by Perkurovski [23], its closest competitor, for large number of processors and large problem sizes. Our scheme is better for the range of number of processors: $P_0 < P < P_1$, where P_1 $(= (8/1024)^{2/3}N^2)$ is the maximum number of processors for the message sizes to be larger than 1 Kbytes. We also derived a complexity analysis for the 2D decomposition strategy proposed by Plimpton at Sandia National Laboratories (for details see Nelson et al. [22]). We found that our strategy scales better than Plimpton's approach, until the latency becomes an issue. The maximum number of processors to sustain approximately the linear scalability has been estimated, which shows that our 2D decomposition of 3D FFTs is likely to perform well on Petascale supercomputers for large problem sizes.

It should be noted that some improvements to our parallel strategy could be further implemented. For large number of processors, the communication section takes more than half the total execution time, thus a natural next step is to overlap the calculations while communicating data. This approach could only be implemented in our proposed approach as we used non-blocking MPI commands. This is in contrast to the other available 2D domain decomposition 3DFFT subroutines that used blocking MPI command (MPI_Alltoall). Another possible improvement is to extend the cyclic permutation communication strategy to 3D domain decomposition implementation. We explored theoretically both improvements and found that by overlapping computations and communications we could improve the performance by at least 10% but not more than 25% for moderate number of processors ($P_{\max_{\mathrm{IMPROVE}}}$, see Eq. 11). As for the 3D domain decomposition implementation, we showed it to be more expensive because of the 4 communication steps which makes it to saturate faster than the other strategies.

The work presented here illustrates the feasibility of simulating turbulent flows on PetaScale computers using the pseudo-spectral method. For isotropic homogeneous turbulence, nine 3D FFTs are required per time step and this typically takes over 80% of the total CPU time. Having a highly-scalable implementation of 3D FFTs is equivalent to a highly scalable, high-resolution direct numerical simulations (DNS) of turbulence on PetaScale computers. We have recently integrated the approach reported here into a pseudo-spectral DNS code, the performance of the DNS code will be presented in a separate paper.

Acknowledgements

This work was made possible by support from the National Science Foundation through grants OCI-0904534, ATM-0730766, and CRI-0958512 and by the National Center for Atmospheric Research (NCAR). NCAR is sponsored by the National Science Foundation. Computing resources were provided by: 1) National Center for Atmospheric Research through CISL-35751010, CISL-35751014 and CISL-35751015, 2) Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number OCI-1053575, and 3) The National Energy Research Scientific

Computing Center (NERSC), which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

Appendix A. Elemental times in the complexity analysis

The elemental times discussed in the complexity analysis can be obtained empirically by curve fitting execution times such as the ones shown in Fig. 7 or through some elemental test runs on the machine (Bluefire – IBM cluster, in this case). The test runs were performed using $P_y \sim P_z$.

A.1. t_c

This could be estimated by doing repeatedly some floating point operations. We carried out simple array multiplications, the results are shown in Table A.1. The average time for t_c was approximately 1.5 ns/FLOP. This value may vary depending on what operation is performed (sum, exponential, sine, cosine, etc.). To overcome this, t_c can also be obtained by curve fitting data points of the computational time with the operation count $(5/2)(N^3\log_2(N^3)/P)$, see Fig. A.1. As the experiments were performed for backward and forward FFT pairs, an additional constant value of two in the horizontal axis should appear. The slope of the line gives $t_c \approx 0.37$ ns/FLOP, which is only 4 times faster than the 1.5 ns/FLOP quoted above. For the in-processor 1D FFT computation, we used the publicly available FFTW by Frigo and Johnson [28]. Recall that the FFTW being used here is highly optimized and it adapts to the machine architecture even taking advantage of the cache memory to speed up calculations.

Table A.1
Computation time per floating point operation (t_c) for different array sizes. The average was obtained after performing numerous multiplications.

Array size	2^3	4^3	8^3	16^3	32^3	64^3	128^3
t_c (ns/FLOP)	1.549	1.490	1.527	1.514	1.529	1.542	1.520

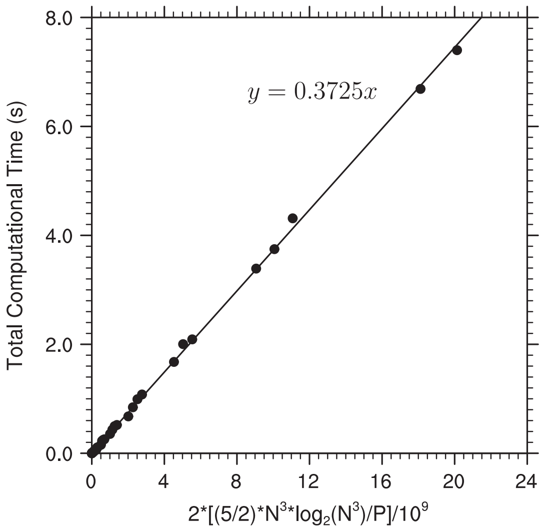

Fig. A.1. The measured computation time as a function of the computational operation counts. The slope provides an estimate for the computation time per floating point operation (t_c).

Table A.2
Memory-to-memory copy time per word (t_a, ns/word) for different array sizes. The average was obtained after performing numerous copies of an array.

Array size	2^3	4^3	8^3	16^3	32^3	64^3	128^3	256^3
t_a	6.33	2.84	2.91	3.96	5.78	60.81	557.81	944.34
$t_{a_{FORTRAN}}$	7.54	2.61	1.34	1.00	1.99	4.89	9.88	9.91

In addition, this value is related not only to arithmetic operations but also to memory-to-memory copy of data needed to execute the one-dimensional FFT algorithm. $1/t_c = 2.7$ GFLOPS is similar to those reported in the benchmark section of the FFTW web page. The IBM Power6 processors on Bluefire run at 4.7 GHz, 2.7 GFLOPS is 57% of the peak 4.7 GFLOPS (assuming one floating point operations per clock cycle).

A.2. t_a

The elemental copy time t_a could be determined by running standard loops of memory-to-memory copy on a single processor and the results are shown in Table A.2. Note that if the copy is performed following the Fortran order (*i.e.*, copying along the first array direction, then the second, and lastly the third so the data is accessed contiguously in memory), t_a (the third row) is lower due to the better use of the adjacency of memory locations. The data for the second row were obtained by purposely copying the data in the reversed order (*i.e.*, the third array direction first, then the second, and lastly the first). The memory-to-memory copy time varies with the array size. Some typical array sizes on the range of optimal performance of our subroutine are between $\sim 6^3$ and $\sim 128^3$. An average value from Table A.2 on that range is 2.31 ns/word following "Fortran order" and 18.39 ns/word not following it.

Since, for large number of processors, the copying time is similar for the x–y and y–z transpositions; t_a can be obtained by curve fitting data points of total copying time vs $6(N^3/P)$, see Fig. A.2 where the slope is approximately 6.37 ns/word. This value falls in the range of values shown in Table A.2. It is larger than the average value of t_a following the "Fortran order" and smaller than t_a without using the Fortran order. This is expected as the total copying time covers the copying during the x–y transpositions (better for Fortran order) and the y–z transpositions (worse for Fortran order).

A.3. t_s and t_w

The communication time is composed of the data transmission time and the latency time. They can be obtained by timing several simple communications (send and receive commands) using different array sizes between two processors only. The results of such experiments on Bluefire are shown in Fig. A.3. The slope of the curves for large data sizes gives t_w and the timing value in the limit of very small data size provides the latency time t_s. As the message size decreases, a plateau indicates the latency time. An average value is taken to give $t_s \approx 7$ μs. This value should be viewed as an approximation. The slope of each curve for large data size provides an estimate for t_w. However, this slope changes with how busy the network of the machine is. In the complexity analysis we assumed t_w to be independent of such uncontrolled factors.

We obtain t_w by curve fitting data points of data transmission times (i.e., total communication time minus the latency time) as a function of $4(N^3/P)$, see Fig. A.4. The average slope of the curve fitting will give t_w.

As noted before, the data transmission time $t_{w,YZ}$ for the y–z transpose is larger than that for the x–y transpose, this is clearly shown in Fig. A.4. The data for 2048^3 problem size are plotted separately, as they are much higher than other data for smaller problem sizes. For 1024^3 and smaller problem sizes, the transmission times are similar and become closer for large numbers of processors. From Fig. A.4, we obtain an average $t_w = (t_{w,XY} + t_{w,YZ})/2 \approx 2.19$ ns/word for 1024^3 and smaller

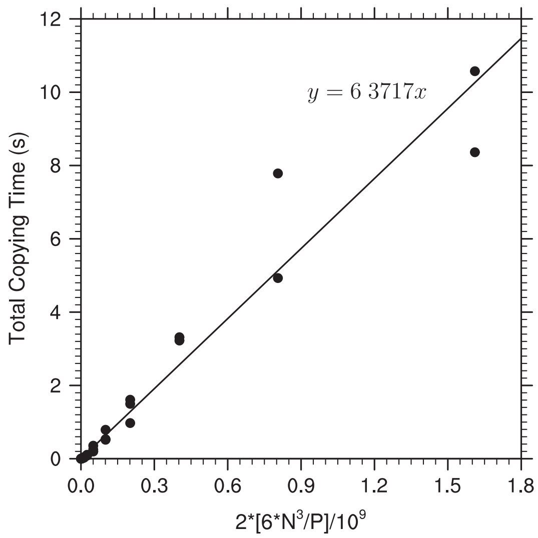

Fig. A.2. The measured memory-to-memory copy time as a function of data size. The slope provides an estimate for t_a.

problem sizes. For the 2048^3 problem size, the differences between x—y and y—z transposes are larger, with an average $t_w \approx 11.60$ ns/word.

These elemental communication times are larger than what is reported by NCAR under optimal conditions for Bluefire ($t_w \approx 0.43$ ns/word for double precision data, and $t_s \approx 1.3$ μs). However, when performing our experiments, the communication bandwidth was shared with other users, so the times we obtained are representative of what a user can actually achieve.

In summary, the elemental times for Bluefire are given in Table A.3.

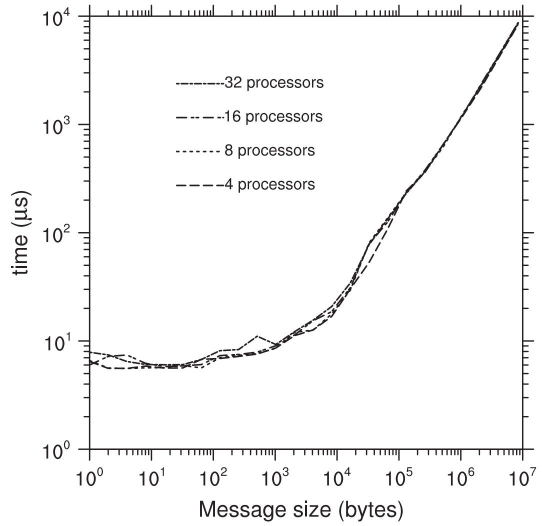

Fig. A.3. The communication time between a number of processors as a function of transmitted data size.

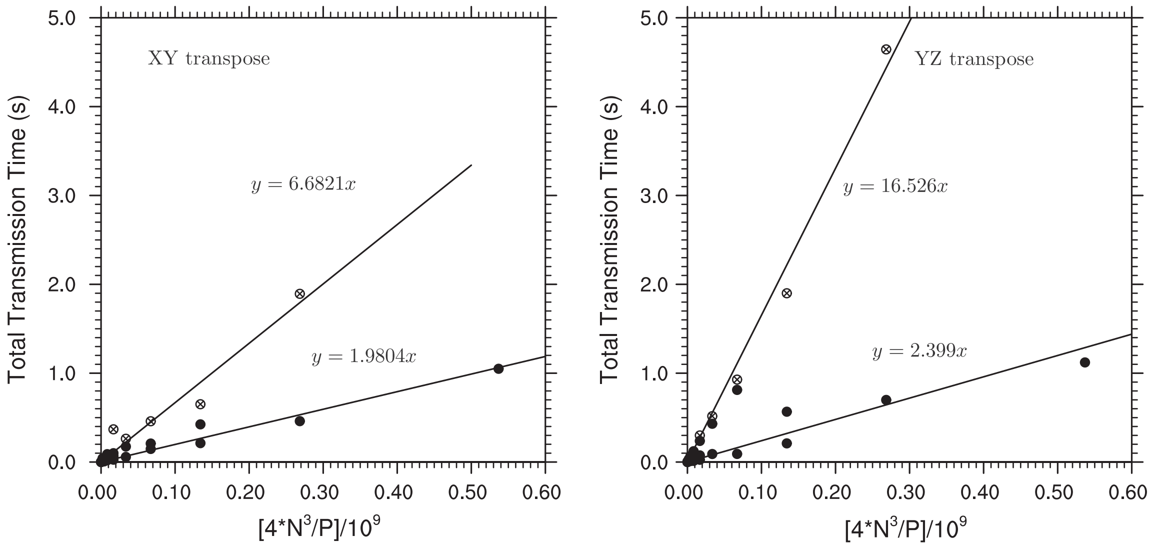

Fig. A.4. Measured data transmission time as a function of data size. The linear fitting provides an estimate of t_w. The left panel shows results for x—y transpositions, while the right panel corresponds to y—z transpositions. The circle-cross symbols are data for 2048^3 problem size and filled circles for smaller problem sizes.

Table A.3
Estimated elemental times on Bluefire for the complexity analysis.

t_c	0.37 ns/FLOP
t_a	6.37 ns/word
t_w	2.19–11.60 ns/word
t_s	7.00 μs

References

[1] E. Bylaska, M. Valiev, R. Kawai, J.H. Weare, Parallel implementation of the projector augmented plane wave method for charged systems, Comput. Phys. Commun. 143 (2002) 11–28.
[2] C. Cavazzoni, G. Chiarotti, A parallel and modular deformable cell CarParrinello code, Comput. Phys. Commun. 123 (1999) 56–76.
[3] H. Calandra, F. Bothorel, P. Vezolle, A massively parallel implementation of the common azimuth pre-stack depth migration, IBM J. Res. Dev. 52 (2008) 83–91.
[4] S. Stellmach, U. Hansen, An efficient spectral method for the simulation of dynamos in Cartesian geometry and its implementation on massively parallel computers, Geochem. Geophys. Geosyst. 9 (2008) 5.
[5] L.P. Wang, S.Y. Chen, J.G. Brasseur, Examination of hypotheses in the Kolmogorov refined turbulence theory through high-resolution simulations. Part 2. Passive scalar field, J. Fluid Mech. 400 (1999) 163–197.
[6] Z. Yin, L. Yuan, T. Tang, A new parallel strategy for two-dimensional incompressible flow simulations using pseudo-spectral methods, J. Comput. Phys. 210 (2005) 325–341.
[7] P. Luchini, M. Quadrio, A low-cost parallel implementation of direct numerical simulation of wall turbulence, J. Comput. Phys. 211 (2006) 551–571.
[8] M. Iovieno, C. Cavazzoni, D. Tordella, A new technique for a parallel dealiased pseudospectral Navier–Stokes code, Comput. Phys. Commun. 141 (2001) 365–374.
[9] S. Poykko, Ab initio electronic structure methods in parallel computers, Appl. Parallel Comput. Large Scale Sci. Ind. Prob. 1541 (1998) 452–459.
[10] J. Wiggs, H. Jonsson, A hybrid decomposition parallel implementation of the Car–Parrinello method, Comput. Phys. Commun. 87 (1995) 319–340.
[11] P. Haynes, M. Cote, Parallel fast Fourier transforms for electronic structure calculations, Comput. Phys. Commun. 130 (2000) 130–136.
[12] M. Eleftheriou, B. Fitch, A. Rayshubskiy, T.J.C. Ward, R.S. Germain, Scalable framework fo 3D FFTs on the Bluegen/l supercomputer: implementation and early performance measurements, IBM J. Res. Develop. 49 (2005) 457.
[13] B. Fang, Y. Deng, G. Martyna, Performance of the 3D FFT on the 6D network torus QCDOC parallel supercomputer, Comput. Phys. Commun. 176 (2007) 531–538.
[14] D. Takahashi, Efficient implementation of parallel three-dimensional FFT on clusters of PCs, Comput. Phys. Commun. 152 (2003) 144–150.
[15] E. Chu, A. George, FFT algorithms and their adaptation to parallel processing, Linear Algebra Appl. 284 (1998) 95–124.
[16] C. Calvin, Implementation of parallel FFT algorithms on distributed memory machines with a minimum overhead of communication, Parallel Comput. 22 (1996) 1255–1279.
[17] M. Inda, R. Bisseling, A simple and efficient parallel FFT algorithm using the BSP model, Parallel Comput. 27 (2001) 1847–1878.
[18] F. Marino, E.E. Swartzlander, Parallel implementation of multidimensional transforms without interprocessor communication, IEEE Trans. Comput. 48 (1999) 951–961.
[19] D. Takahashi, T. Boku, M. Sato, A blocking algorithm for parallel 1-D FFT on clusters of PCs, in: Euro-Par 2005 Parallel Procesing, Procedings, vol. 2400, 2002, pp. 691–700.
[20] R.B. Pelz, The parallel Fourier pseudospectral method, J. Comput. Phys. 92 (1991) 296–312.
[21] P. Dmitruk, L.-P. Wang, W.H. Matthaeus, R. Zhang, D. Seckel, Scalable parallel FFT for spectral simulations on a Beowulf cluster, Parallel Comput. 27 (2001) 1921–1936.
[22] J.S. Nelson, S.J. Plimpton, M.P. Sears, Plane-wave electronic-structure calculations on a parallel supercomputer, Phys. Rev. B 47 (1993) 1765–1774.
[23] D. Pekurovsky, Scaling three-dimensional Fourier transform, San Diego Super Computer Summer Institute, <http://www.sdsc.edu/us/resources/p3dfft/>, San Diego, USA, 2007.
[24] D. Takahashi, An implementation of parallel 3-D FFT with 2-D decomposition on a massively parallel cluster of multi-core processors, in: Proceedings of the Eighth International Conference on Parallel Processing and Applied Mathematics, (PPAM), Wroclaw, Poland, 2009.
[25] N. Li, S. Laizet, 2DECOMP&FFT-A highly scalable 2D decomposition library and FFT interface, Cray User Group 2010 Conference, Edinburgh, 2010, <www.2decomp.org>.
[26] M. Eleftheriou, J.E. Moreira, B.G. Fitch, R.S. Germain, A volumetric FFT for BlueGene/L, High Perform. Comput. HIPC 2913 (2003) 194–203.
[27] M. Eleftheriou, B. Fitch, A. Rayshubskiy, T.J.C. Ward, R. Germain, Performance measurements of the 3D FFT on the BlueGene/L supercomputer, in: Euro-Par 2005 Parallel Processing, Proceedings, vol. 3648, 2005, pp. 795–803.
[28] S. Johnson, M. Frigo, A modified split-radix FFT with fewer arithmetic operations, IEEE Trans. Signal Process. 55 (2007) 111–119, <www.fftw.org>.
[29] W. Press, B. Flannery, S. Teukolsky, W. Vetterling, Numerical Recipes, The Art of Scientific Computing, Cambridge University Press, Cambridge, UK, 1986.