SIAM J. Sci. Comput.
Vol. 35, No. 3, pp. C213–C236
© 2013 Society for Industrial and Applied Mathematics

PFFT: AN EXTENSION OF FFTW TO MASSIVELY PARALLEL ARCHITECTURES*

MICHAEL PIPPIG†

Abstract. We present an MPI based software library for computing fast Fourier transforms (FFTs) on massively parallel, distributed memory architectures based on the Message Passing Interface standard (MPI). Similar to established transpose FFT algorithms, we propose a parallel FFT framework that is based on a combination of local FFTs, local data permutations, and global data transpositions. This framework can be generalized to arbitrary multidimensional data and process meshes. All performance-relevant building blocks can be implemented with the help of the FFTW software library. Therefore, our library offers great flexibility and portable performance. Similarly to FFTW, we are able to compute FFTs of complex data, real data, and even- or odd-symmetric real data. All the transforms can be performed completely in place. Furthermore, we propose an algorithm to calculate pruned FFTs more efficiently on distributed memory architectures. For example, we provide performance measurements of FFTs of sizes between 512^3 and 8192^3 up to 262144 cores on a BlueGene/P architecture, up to 32768 cores on a BlueGene/Q architecture, and up to 4096 cores on the Jülich Research on Petaflop Architectures (JuRoPA).

Key words. parallel fast Fourier transform

AMS subject classifications. 65T50, 65Y05

DOI. 10.1137/120885887

1. Introduction.

Without doubt, the fast Fourier transform (FFT) is one of the most important algorithms in scientific computing. It provides the basis of many algorithms, and a tremendous number of applications can be listed. Since the famous divide and conquer algorithm by Cooley and Tukey [4] was published in 1965, many algorithms were derived for computing the discrete Fourier transform in $\mathcal{O}(n \log n)$. This variety of algorithms and the continuous change of hardware architectures made it practically impossible to find one FFT algorithm that is best suitable for all circumstances. Instead, the developers of the FFTW software library proposed another approach. Under the hood, FFTW compares a wide variety of different FFT algorithms and measures their runtimes to find the most appropriate one for the current hardware architecture. The sophisticated implementation is hidden behind an easy interface structure. Therefore, users of FFTW are able to apply highly optimized FFT algorithms without knowing all the details about them. These algorithms have been continuously improved by the developers of FFTW and other collaborators to support new hardware trends, such as SSE, SSE2, graphic processors, and shared memory parallelization. The current release 3.3.3 of FFTW also includes a very flexible distributed memory parallelization based on the Message Passing Interface standard (MPI). However, the underlying parallel algorithm is not suitable for current massively parallel architectures. To give a better understanding, we start with a short introduction to parallel distributed memory FFT implementations and explain the problem for the three-dimensional FFT.

*Submitted to the journal's Software and High-Performance Computing section July 24, 2012; accepted for publication (in revised form) February 7, 2013; published electronically May 14, 2013. This work was supported by the BMBF grant 01IH08001B.
http://www.siam.org/journals/sisc/35-3/88588.html

†Department of Mathematics, Chemnitz University of Technology, 09107 Chemnitz, Germany (michael.pippig@mathematik.tu-chemnitz.de).

There are two main approaches for parallelizing multidimensional FFTs; the first is binary exchange algorithms, and the second is transpose algorithms. An introduction and theoretical comparison can be found in [13]. We want to concentrate on transpose algorithms; i.e., we perform a sequence of local one-dimensional FFTs and two-dimensional data transpositions that are very similar to all-to-all communications. For convenience we consider the three-dimensional input array to be of size $n_0 \times n_1 \times n_2$ with $n_0 \geq n_1 \geq n_2$, and all the dimensions should be divisible by the number of processes.

It is well known that a multidimensional FFT can be efficiently computed by a sequence of lower-dimensional FFTs. For example, a three-dimensional FFT of size $n_0 \times n_1 \times n_2$ can be computed by n_0 two-dimensional FFTs of size $n_1 \times n_2$ along the last two dimensions followed by $n_1 \times n_2$ one-dimensional FFTs of size n_0 along the first dimension. Therefore, the first parallel transpose FFT algorithms were based on one-dimensional data decomposition (also called slab decomposition), which means that the three-dimensional input array is split along n_0 into equal blocks to distribute it on a given number $P \leq n_0$ of MPI processes; i.e., all processes own equal contiguous blocks of size $n_0/P \times n_1 \times n_2$. At the first step, every process is able to compute n_0/P two-dimensional FFTs of size $n_1 \times n_2$ along the last two dimensions, since all required data is locally available. Afterward, only $n_1 n_2$ one-dimensional FFTs of size n_0 along the first dimension are left in order to complete the three-dimensional FFT. However, the required data is distributed along the first dimension among all processes. Therefore, a data transposition (very similar to a call of MPI_Alltoall) is performed that results in a one-dimensional data decomposition of the second dimension; i.e., every process owns a contiguous block of size $n_0 \times n_1/P \times n_2$. At this time the first dimension is local to each process. Therefore, we are able to perform the remaining $n_1 n_2/P$ one-dimensional FFTs of size n_0 on every process. Implementations of the one-dimensional decomposed parallel FFT are, for example, included in the IBM PESSL library [9], the Intel Math Kernel Library [14], and the FFTW [10] software package. Unfortunately, all of these FFT libraries lack high scalability on massively parallel architectures because their data decomposition approach limits the number of efficiently usable MPI processes by n_1. Note that we assumed the dimensions $n_0 \geq n_1 \geq n_2$ to be ordered. Therefore, the resulting data decomposition $n_0 \times n_1/P \times n_2$ implies a stronger upper bound on the number of processes P than the initial data decomposition $n_0/P \times n_1 \times n_2$. Figure 1 shows an illustration of the one-dimensional distributed FFT and an example of its scalability limitation.

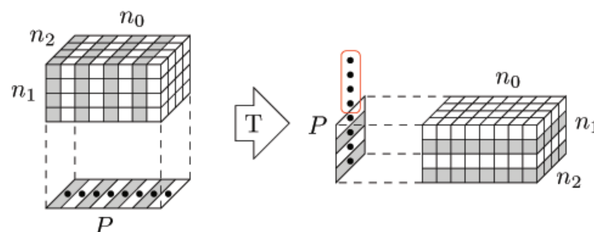

FIG. 1. *Decomposition of a three-dimensional array of size* $n_0 \times n_1 \times n_2 = 8 \times 4 \times 4$ *on a one-dimensional process grid of size* $P = 8$. *After the transposition (T) half of the processes remain idle.*

The main idea in overcoming this scalability bottleneck is to use a two-dimensional

data decomposition. Assume a two-dimensional mesh of $P_0 \times P_1$ MPI processes. Two-dimensional data decomposition (also called rod or pencil decomposition) means that the three-dimensional input array is split along the first two dimensions n_0 and n_1; i.e., each process owns a contiguous block of size $n_0/P_0 \times n_1/P_1 \times n_2$. Now, every process starts with the computation of $n_0/P_0 \times n_1/P_1$ one-dimensional FFTs of size n_2, followed by a communication step that ensures a new two-dimensional data decomposition with blocks of size $n_0/P_0 \times n_1 \times n_2/P_1$. After further $n_0/P_0 \times n_2/P_1$ one-dimensional FFTs of size n_1 and one more communication step, we end up with local blocks of size $n_0 \times n_1/P_0 \times n_2/P_1$. The three-dimensional FFT is finished after further $n_1/P_0 \times n_2/P_1$ one-dimensional FFTs of size n_0. Note that the number of data transpositions increased by one in comparison to the one-dimensional decomposition approach. However, these data transpositions are performed in smaller subgroups along the rows and columns of the process mesh. Figure 2 shows an illustration of the two-dimensional distributed FFT and its improved scalability in comparison to the example above. The two-dimensional data decomposition allows us to increase the number of MPI processes to at most $n_1 n_2$. It was first proposed by Ding, Ferraro, and Gennery [5] in 1995. Eleftheriou et al. [7] implemented a software library [6] for power-of-two FFTs customized to the BlueGene/L architecture based on the two-dimensional data decomposition. Please note that although the so-called volumetric domain decomposition by Eleftheriou et al. [7] looks like a three-dimensional data decomposition at first sight, it turns out that the underlying parallel FFT algorithm still uses a two-dimensional data decomposition. Publicly available implementations of the two-dimensional decomposition approach are the FFT package [22, 21] by Plimpton from Sandia National Laboratories, the P3DFFT library [18, 17] by Pekurovsky, and more recently the 2DECOMP&FFT library [16, 15] by Li. Furthermore, performance evaluations of two-dimensional decomposed parallel FFTs have been published by Fang, Deng, and Martyna [8] and Takahashi [23].

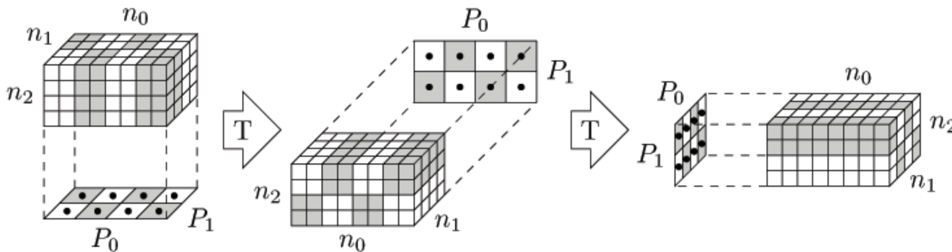

Fig. 2. *Distribution of a three-dimensional array of size $n_0 \times n_1 \times n_2 = 8 \times 4 \times 4$ on a two-dimensional process grid of size $P_0 \times P_1 = 4 \times 2$. None of the processes remains idle in any calculation step.*

All these implementations offer a different set of features and introduce their own interface. Since one dimension of the input array must remain local to all processes, the parallel transpose algorithm of a three-dimensional FFT is restricted to a one- or two-dimensional process mesh. However, this is no longer true if we want to compute FFTs of dimension four or higher. In particular, there are two weak points of the above-mentioned parallel implementations. First, there is no publicly available FFT library that supports process meshes with more then two dimensions for FFTs of dimension four or higher. Second, the two-dimensional data decomposition is implemented only for three-dimensional FFTs—not for four- or higher-dimensional

FFTs. Our parallel FFT framework aims to close this gap and offers one library for all the above-mentioned use cases with an FFTW-like interface. In fact, we extend the distributed memory parallel FFTW to multidimensional data decompositions. Therefore, we are able to compute d-dimensional FFTs in parallel on a process mesh of any dimension less than or equal to $d-1$. In addition, our framework is able to handle parallel FFTs with truncated input and output arrays (also known as over- and undersampling) more efficiently than the above-mentioned parallel FFT libraries. These so-called pruned FFTs save both memory and computational cost in comparison to the straightforward implementation, as we will see later. Last but not least, our framework also supports the computation parallel sine and cosine transforms based on a multidimensional data decomposition. Please note that sine and cosine transforms are also supported by the latest release of the P3DFFT.

This paper is structured as follows. First, we introduce the notation that is used throughout the remainder of this paper. In section 3 we describe the building blocks that will be plugged together in section 4 to form a flexible parallel FFT framework. Section 5 provides an overview of our publicly available, parallel FFT implementation. Runtime measurements are presented in section 6. Finally, we close with a conclusion.

2. Definitions and assumptions.

In this section, we define the supported one-dimensional transforms of our framework. These can be serial FFTs with either real or complex input. Our aim is to formulate a unique parallel FFT framework that is independent of the underlying one-dimensional transform. But this implies that we have to keep in mind that, depending on the transform type, the input array will consist of real or complex data. Whenever it is important to distinguish the array type, we mention it explicitly.

2.1. One-dimensional FFT of complex data.

Consider n complex numbers $f_k \in \mathbb{C}$, $k = 0, \ldots, n-1$. The one-dimensional forward discrete Fourier transform (DFT) of size n is defined as

$$\hat{f}_l := \sum_{k=0}^{n-1} f_k \, \mathrm{e}^{-2\pi \mathrm{i} lk/n} \in \mathbb{C}, \quad l = 0, \ldots, n-1.$$

Evaluating all $\hat{f}_l$ by direct summation requires $\mathcal{O}(n^2)$ arithmetic operations. In 1965 Cooley and Tukey published an algorithm called Fast Fourier Transform (FFT) [4] that reduces the arithmetic complexity to $\mathcal{O}(n \log n)$. Furthermore, we define the backward discrete Fourier transform of size n by

$$g_k := \sum_{l=0}^{n-1} \hat{f}_l \, \mathrm{e}^{+2\pi \mathrm{i} lk/n} \in \mathbb{C}, \quad k = 0, \ldots, n-1.$$

Note that with these two definitions the backward transform inverts the forward transform only up to the scaling factor n, e.g., $g_k = n f_k$ for $k = 0, \ldots, n-1$. We refer to fast algorithms for computing the DFT of complex data by the abbreviation c2c-FFT, since they transform complex inputs into complex outputs.

2.2. One-dimensional FFT of real data.

Consider n real numbers $g_k \in \mathbb{R}$, $k = 0, \ldots, n-1$. The one-dimensional forward DFT of real data is given by

$$\hat{f}_l := \sum_{k=0}^{n-1} f_k \, \mathrm{e}^{-2\pi \mathrm{i} lk/n} \in \mathbb{C}, \quad l = 0, \ldots, n-1.$$

Since the outputs satisfy the Hermitian symmetry

$$\hat{f}_{n-l} = \hat{f}_l^*, \quad l = 0, \dots, n/2,$$

it is sufficient to store the first $n/2 + 1$ complex outputs (division rounded down for odd n). We define the backward DFT of Hermitian symmetric data of size n by

$$g_k := \sum_{l=0}^{n-1} \hat{f}_l \, \mathrm{e}^{+2\pi \mathrm{i} lk/n} \in \mathbb{R}, \quad k = 0, \dots, n-1.$$

Corresponding to their input and output data types, we abbreviate fast $\mathcal{O}(n \log n)$ algorithms for computing the forward DFT of real data with r2c-FFT and the backward transform with c2r-FFT.

2.3. One-dimensional FFT of even- or odd-symmetric real data. Depending on the symmetry of the input data, there exist 16 different definitions of DFTs of even- or odd-symmetric real data. At this point, we only give the definition of the most commonly used discrete cosine transform of the second kind (DCT-II). The definitions of the other transforms can be found, for example, in the FFTW manual [11].

Consider n real numbers $f_k \in \mathbb{R}$, $k = 0, \dots, n-1$. The one-dimensional DCT-II is given by

$$\hat{f}_l = 2 \sum_{k=0}^{n-1} f_k \, \cos\left(\pi(l + 1/2)k/n\right) \in \mathbb{R}, \quad l = 0, \dots, n-1.$$

Again, the DCT-II can be computed in $\mathcal{O}(n \log n)$. We summarize all fast algorithms to compute the DFT of even- or odd-symmetric real data under the acronym r2r-FFT.

2.4. Pruned FFTs. Let $N \le n$ and $\hat{N} \le n$. For N complex numbers $h_k \in \mathbb{C}$, $k = 0, \dots, N-1$, we define the one-dimensional pruned forward DFT by

$$\hat{h}_l = \sum_{k=0}^{N-1} h_k \mathrm{e}^{-2\pi \mathrm{i} kl/n}, \quad l = 0, \dots, \hat{N} - 1.$$

This means that we are interested in only the first $\hat{N}$ outputs of an oversampled FFT. Obviously, we can calculate the pruned DFT with complexity $\mathcal{O}(n \log n)$ in the following three steps. First, pad the input vector with zeros to the given DFT size n, i.e.,

$$f_k = \begin{cases} h_k & : k = 0, \dots, N-1, \\ 0 & : k = N, \dots, n-1. \end{cases}$$

Second, calculate the sums

$$\hat{f}_l := \sum_{k=0}^{n-1} f_k \, \mathrm{e}^{-2\pi \mathrm{i} lk/n} \in \mathbb{C}, \quad l = 0, \dots, n-1,$$

with a c2c-FFT on size n in $\mathcal{O}(n \log n)$. Afterward, truncate the output vector of length n to the needed length $\hat{N}$, i.e.,

$$\hat{h}_l = \hat{f}_l, \quad l = 0, \dots, \hat{N} - 1.$$

We use a similar three-step algorithm to compute the pruned r2c-FFT and pruned r2r-FFT. In the r2c-case the truncation slightly changes to

$$\hat{h}_l = \hat{f}_l, \quad l = 0, \dots, \hat{N}/2 + 1,$$

in order to respect the Hermitian symmetry of the output array.

2.5. Multidimensional FFTs. Assume a multidimensional input array of $n_0 \times \cdots \times n_{d-1}$ real or complex numbers. We define the multidimensional FFT as the consecutive calculation of the one-dimensional FFTs along the dimensions of the input array. Whenever we want to calculate several multidimensional FFTs of the same size, we use the notation $n_0 \times \cdots \times n_{d-1} \times h$, where the multiplier h tells us how many FFTs of size $n_0 \times \cdots \times n_{d-1}$ are supposed to be calculated simultaneously.

Again, we have to pay special attention to r2c-transforms. Here, we first compute the one-dimensional r2c-FFTs along the last dimension of the multidimensional array. Because of Hermitian symmetry the output array consists of $n_0 \times \cdots \times n_{d-2} \times (n_{d-1}/2+1)$ complex numbers. Afterward, we calculate the separable one-dimensional c2c-FFTs along the first $d-1$ dimensions. For c2r-transforms we do it the other way around.

2.6. Parallel data decomposition. Assume a multidimensional array of size $N_0 \times \cdots \times N_{d-1}$. Furthermore, for $r < d$ assume an r-dimensional Cartesian communicator, which includes a mesh of $P_0 \times \cdots \times P_{r-1}$ MPI processes. Our parallel algorithms are based on a simple block structured domain decomposition; i.e., every process owns a block of $N_0/P_0 \times \cdots \times N_{r-1}/P_{r-1} \times N_r \times \cdots \times N_{d-1}$ local data elements. The data elements may be real or complex numbers depending on the FFT we want to compute. For the sake of clarity, we claim that the dimensions of the data set should be divisible by the dimensions of the process grid, i.e., $P_i | N_j$ for all $i = 0, \dots, r-1$ and $j = 0, \dots, d-1$. This ensures that the data will be distributed equally among the processes in every step of our algorithm. In order to make the following algorithms more flexible, we can easily overcome these requirements. Note also that our implementation does not depend on this restriction. Nevertheless, unequal blocks lead to load imbalances of the parallel algorithm and should be avoided whenever possible. Since we claimed that the rank r of the process mesh is less than the rank d of the data array, at least one dimension of the data array is local to the processes.

Depending on the context we interpret the notation N_i/P_j either as a simple division or as a splitting of the data array along dimension N_i on P_j processes in equal blocks of size N_i/P_j for all $i = 0, \dots, d-1$ and $j = 0, \dots, r-1$. This notation allows us to compactly represent the main characteristics of parallel block data distribution, namely, the local transposition of dimensions and the global array decomposition into blocks. For example, in the case $d = 3, r = 2$ we would interpret the notation $N_2/P_1 \times N_0/P_0 \times N_1$ as an array of size $N_0 \times N_1 \times N_2$ that is distributed on P_0 processes along the first dimension and on P_1 processes along the last dimension. Additionally, the local array blocks are transposed such that the last array dimension comes first. We assume such multidimensional arrays to be stored in C typical row major order; i.e., the last dimension lies consecutively in memory. Therefore, cutting the occupied memory of a multidimensional array into equal pieces corresponds to a splitting of the array along the first dimension.

3. The modules of our parallel FFT framework. The three major ingredients of a parallel transpose FFT algorithm are serial FFTs, serial array transposition, and global array transpositions. All of these are somehow already implemented in

the current release 3.3.2 of the FFTW software library. Our parallel FFT framework builds upon several modules that are more or less wrappers to these FFTW routines. We now describe the modules from bottom to top. In the next section we combine the modules into our parallel FFT framework.

3.1. The serial FFT module. The guru interface of FFTW offers a very general way to compute multidimensional vector loops of multidimensional FFTs [10]. However, we do not need the full generality and therefore wrote a wrapper that enables us to compute multidimensional FFTs of the following form. Assume a three-dimensional array of $h_0 \times n \times h_1$ real or complex numbers. Our wrapper allows us to compute the separable one-dimensional FFTs along the second dimension, i.e.,

$$h_0 \times n \times h_1 \overset{\mathsf{FFT}}{\to} h_0 \times \hat{n} \times h_1.$$

Thereby, we denote Fourier transformed dimensions by hats. Note that we do not compute the one-dimensional FFTs along the first dimension h_0. Later, we will use this dimension to store the parallel distributed dimensions. The additional dimension h_1 at the end of the array allows us to compute a set of h_1 serial FFTs at once. The serial FFTs can be any of the serial FFTs that we introduced in section 2, e.g., c2c-FFT, r2c-FFT, c2r-FFT, or r2r-FFT.

In addition, our wrapper allows the input array to be transposed in the first two dimensions

$$n \times h_0 \times h_1 \underset{\mathsf{TI}}{\overset{\mathsf{FFT}}{\to}} h_0 \times \hat{n} \times h_1$$

and the output array to be transposed in the first two dimensions

$$h_0 \times n \times h_1 \underset{\mathsf{TO}}{\overset{\mathsf{FFT}}{\to}} \hat{n} \times h_0 \times h_1.$$

This is a crucial feature, since the local data blocks must be locally transposed before the global communication step can be performed. Experienced FFTW users may have noticed that the FFTW guru interface allows us to calculate local array transpositions and serial FFTs in one step. Computation of a local array transposition is indeed a nontrivial task because one has to think of many details about the memory hierarchy of current computer architectures. FFTW implements cache oblivious array transpositions [12], which aim to minimize the asymptotic number of cache misses independently of the cache size. Unfortunately, we experienced that the performance of an FFT combined with the local transposition is sometimes quite poor. Under some circumstances it is even better to do the transposition and the FFT in two separate steps. In addition, it is not possible to combine the transposition with a multidimensional r2c FFT. Therefore, we decided to implement an additional planning step into the wrapper. Our serial FFT plan now consists of two FFTW plans. The planner decides whether the first FFTW plan performs a transposition, a serial FFT, or both of them. The second FFTW plan performs the outstanding task to complete the serial transposed FFT. In contrast to the FFTW planner, our additional planner is very time consuming, since it has to plan and execute several serial FFTs and data transpositions. The user can decide whether it is worth the effort when he calls the PFFT planning interface. Additionally, we can switch off the serial FFT in order to perform the local transpositions

$$n \times h_0 \times h_1 \underset{\mathsf{TI}}{\to} h_0 \times n \times h_1$$

and

$$h_0 \times n \times h_1 \underset{\mathsf{TO}}{\rightarrow} n \times h_0 \times h_1$$

solely.

Remark 1. In addition to the order of transposition and serial FFT our planner also decides which plan should be executed in place or out of place to reach the minimal runtime.

Remark 2. All of these steps can be performed in place. This is one of the great benefits we get from using FFTW.

3.2. The serial pruned FFT module. The serial FFTs can be easily generalized to pruned FFTs with the three-step algorithm from section 2.4. The padding with zeros and the truncation steps have been implemented as modules in PFFT. To keep notation simple, we do not introduce further symbols to mark a serial FFT as a pruned FFT. Instead, we declare that every one-dimensional FFT of size n can be pruned to N inputs and $\hat{N}$ outputs. This means

$$h_0 \times N \times h_1 \overset{\mathsf{FFT}}{\rightarrow} h_0 \times \hat{N} \times h_1 \tag{3.1}$$

abbreviates the three-step pruning algorithm

$$h_0 \times N \times h_1 \rightarrow h_0 \times n \times h_1 \overset{\mathsf{FFT}}{\rightarrow} h_0 \times \hat{n} \times h_1 \rightarrow h_0 \times \hat{N} \times h_1.$$

This holds analogously if the first two dimensions of the FFT input or output are transposed, e.g.,

$$\begin{aligned} N \times h_0 \times h_1 &\underset{\mathsf{TI}}{\overset{\mathsf{FFT}}{\rightarrow}} h_0 \times \hat{N} \times h_1, \\ h_0 \times N \times h_1 &\underset{\mathsf{TO}}{\overset{\mathsf{FFT}}{\rightarrow}} \hat{N} \times h_0 \times h_1. \end{aligned} \tag{3.2}$$

3.3. The global data transposition module. Suppose a three-dimensional array of size $N_0 \times N_1 \times h$ is mapped on P processes, such that every process holds a block of size $N_0/P \times N_1 \times h$. The MPI interface of FFTW version 3.3.2 includes a parallel matrix transposition (T) to remap the array into blocks of size $N_1/P \times N_0 \times h$. This algorithm is also used for the one-dimensional decomposed parallel FFT implementations of FFTW. In addition, the global transposition algorithm of FFTW supports the local transposition of the first two dimensions of the input array (TI) or the output array (TO). This allows us to handle the following global transpositions:

$$\begin{aligned} N_0/P \times N_1 \times h &\overset{\mathsf{T}}{\rightarrow} N_1/P \times N_0 \times h, \\ N_1 \times N_0/P \times h &\underset{\mathsf{TI}}{\overset{\mathsf{T}}{\rightarrow}} N_1/P \times N_0 \times h, \\ N_0/P \times N_1 \times h &\underset{\mathsf{TO}}{\overset{\mathsf{T}}{\rightarrow}} N_0 \times N_1/P \times h. \end{aligned} \tag{3.3}$$

There are great advantages of using the parallel transposition algorithms of FFTW instead of direct calls to corresponding MPI functions. FFTW does not use only one algorithm to perform an array transposition. Instead different transposition algorithms are compared in the planning step to get the fastest one. This provides us with portable hardware adaptive communication functions. There are three different

transpose algorithms implemented in the current release of FFTW. All of them use some local transpositions in order to get contiguous chunks of memory and a global transposition that is equivalent to a call of MPI_Alltoall or MPI_Alltoallv. Indeed, the first variant is based on MPI_Alltoallv. A second algorithm uses scheduled pointwise communication in order to substitute MPI_Alltoallv. This algorithm can be performed in place, which means that per process only one buffer of size $(N_0 \times N_1 \times h)/P^2$ is necessary. Note that it is impossible to implement such a memory-efficient global transpose with the help of the standard MPI_Alltoall functions. FFTW can also use a third procedure, a recursive transposition pattern, as a substitute for MPI_Alltoall. In summary, we see that the global transposes of FFTW will be at least as good as an implementation based on MPI_Alltoall or even better if the planner finds a faster algorithm. All of these details are hidden behind the easy to use interface of FFTW. However, we need a slight generalization of the transpositions that are available in FFTW to make them suitable for our parallel FFT framework. If we set

$$N_0 = L_1 \times h_1, \quad N_1 = L_0 \times h_0, \quad h = h_2,$$

the transpositions (3.3) turn into

$$\begin{aligned} &L_1/P \times h_1 \times L_0 \times h_0 \times h_2 \overset{\mathsf{T}}{\rightarrow} L_0/P \times h_0 \times L_1 \times h_1 \times h_2, \\ &L_0 \times h_0 \times L_1/P \times h_1 \times h_2 \underset{\mathsf{TI}}{\overset{\mathsf{T}}{\rightarrow}} L_0/P \times h_0 \times L_1 \times h_1 \times h_2, \\ &L_1/P \times h_1 \times L_0 \times h_0 \times h_2 \underset{\mathsf{TO}}{\overset{\mathsf{T}}{\rightarrow}} L_1 \times h_1 \times L_0/P \times h_0 \times h_2. \end{aligned} \tag{3.4}$$

Remark 3. Although this substitution looks straightforward, we must choose the block sizes carefully. Whenever P does not divide L_0 or L_1, we cannot use default block sizes $(L_0 \times h_0)/P$ and $(L_1 \times h_1)/P$ of FFTW. Instead we must ensure that only L_0 and L_1 are distributed on P processes. This corresponds to the block sizes $L_0/P \times h_0$ and $L_1/P \times h_1$.

Remark 4. Similar to FFTW, our global data transpositions operate on real numbers only. However, complex arrays that store real and imaginary parts in the typical interleaved way can be seen as arrays of real pairs. Therefore, we need only double h_2 to initiate the communication for complex arrays.

4. The parallel FFT framework. Now, we have collected all the ingredients to formulate the parallel FFT framework that allows us to calculate h pruned multidimensional FFTs of size

$$N_0 \times \cdots \times N_{d-1} \overset{\mathsf{FFT}}{\rightarrow} \hat{N}_0 \times \cdots \times \hat{N}_{d-1}$$

on a process mesh of size $P_0 \times \cdots \times P_{r-1}$, $r < d$. Our forward FFT framework starts with the r-dimensional decomposition given by

$$N_0/P_0 \times \cdots \times N_{r-1}/P_{r-1} \times N_r \times \cdots \times N_{d-1} \times h.$$

For convenience, we introduce the notation

$$\bigtimes_{s=l}^{u} N_s := \begin{cases} N_l \times \cdots \times N_u & : l \le u, \\ 1 & : l > u. \end{cases}$$

Figure 3 lists the pseudocode of the parallel forward FFT framework.

1: **for** $t \leftarrow 0, \ldots, d-r-2$ **do**
2: $h_0 \leftarrow \bigtimes_{s=0}^{r-1} N_s/P_s \times \bigtimes_{s=r}^{d-2-t} N_s$
3: $N \leftarrow N_{d-1-t}$
4: $h_1 \leftarrow \bigtimes_{s=d-t}^{d-1} \hat{N}_s \times h$
5: $h_0 \times N \times h_1 \overset{\mathsf{FFT}}{\rightarrow} h_0 \times \hat{N} \times h_1$
6: **end for**
7: **for** $t \leftarrow 0, \ldots, r-1$ **do**
8: $h_0 \leftarrow \bigtimes_{s=r-t}^{r-1} \hat{N}_{s+1}/P_s \times \bigtimes_{s=0}^{r-t-1} N_s/P_s$
9: $N \leftarrow N_{r-t}$
10: $h_1 \leftarrow \bigtimes_{s=r+1}^{d-1} \hat{N}_s \times h$
11: $h_0 \times N \times h_1 \underset{\mathsf{TO}}{\overset{\mathsf{FFT}}{\rightarrow}} \hat{N} \times h_0 \times h_1$
12:
13: $L_0 \leftarrow \hat{N}_{r-t}$
14: $h_0 \leftarrow \bigtimes_{s=r-t}^{r-1} \hat{N}_{s+1}/P_s \times \bigtimes_{s=0}^{r-t-2} N_s/P_s$
15: $L_1 \leftarrow N_{r-t-1}$
16: $h_1 \leftarrow 1$
17: $h_2 \leftarrow \bigtimes_{s=r+1}^{d-1} \hat{N}_s \times h$
18: $P \leftarrow P_{r-t-1}$
19: $L_0 \times h_0 \times L_1/P \times h_1 \times h_2 \underset{\mathsf{TI}}{\overset{\mathsf{T}}{\rightarrow}} L_0/P \times h_0 \times L_1 \times h_1 \times h_2$
20: **end for**
21: $h_0 \leftarrow \bigtimes_{s=0}^{r-1} \hat{N}_{s+1}/P_s$
22: $N \leftarrow N_0$
23: $h_1 \leftarrow \bigtimes_{s=r+1}^{d-1} \hat{N}_s \times h$
24: $h_0 \times N \times h_1 \overset{\mathsf{FFT}}{\rightarrow} h_0 \times \hat{N} \times h_1$

FIG. 3. *Parallel forward FFT framework.*

Within the first loop we use the serial FFT module (3.1) to calculate the one-dimensional (pruned) FFTs along the last $d-r-1$ array dimensions. In the second loop we calculate r one-dimensional pruned FFTs with transposed output (3.2) interleaved by global data transpositions with transposed input (3.4). Finally, a single nontransposed FFT (3.1) must be computed to finish the full d-dimensional FFT. The data decomposition of the output is then given by

$$\hat{N}_1/P_0 \times \cdots \times \hat{N}_{r-2}/P_{r-1} \times \hat{N}_r \times \cdots \times \hat{N}_{d-1} \times h.$$

Note that the dimensions of the output array are slightly transposed.

Now, the parallel backward FFT framework can be derived very easy since we need only revert all the steps of the forward framework. The backward framework starts with the output decomposition of the forward framework

$$\hat{N}_1/P_0 \times \cdots \times \hat{N}_{r-2}/P_{r-1} \times \hat{N}_r \times \cdots \times \hat{N}_{d-1} \times h$$

and ends with the initial data decomposition

$$N_0/P_0 \times \cdots \times N_{r-1}/P_{r-1} \times N_r \times \cdots \times N_{d-1} \times h.$$

Figure 4 lists the parallel backward FFT framework in pseudocode.

1: $h_0 \leftarrow \bigtimes_{s=0}^{r-1} \hat{N}_{s+1}/P_s$
2: $N \leftarrow \hat{N}_0$
3: $h_1 \leftarrow \bigtimes_{s=r+1}^{d-1} \hat{N}_s \times h$
4: $h_0 \times \hat{N} \times h_1 \overset{\mathsf{FFT}}{\rightarrow} h_0 \times N \times h_1$
5: **for** $t \leftarrow r-1, \ldots, 0$ **do**
6: $L_1 \leftarrow \hat{N}_{r-t}$
7: $h_1 \leftarrow \bigtimes_{s=r-t}^{r-1} \hat{N}_{s+1}/P_s \times \bigtimes_{s=0}^{r-t-2} N_s/P_s$
8: $L_0 \leftarrow N_{r-t-1}$
9: $h_0 \leftarrow 1$
10: $h_2 \leftarrow \bigtimes_{s=r+1}^{d-1} \hat{N}_s \times h$
11: $P \leftarrow P_{r-t-1}$
12: $L_1/P \times h_1 \times L_0 \times h_0 \times h_2 \underset{\mathsf{TO}}{\overset{\mathsf{T}}{\rightarrow}} L_1 \times h_1 \times L_0/P \times h_0 \times h_2$
13:
14: $h_0 \leftarrow \bigtimes_{s=r-t}^{r-1} \hat{N}_{s+1}/P_s \times \bigtimes_{s=0}^{r-t-1} N_s/P_s$
15: $N \leftarrow \hat{N}_{r-t}$
16: $h_1 \leftarrow \bigtimes_{s=r+1}^{d-1} \hat{N}_s \times h$
17: $\hat{N} \times h_0 \times h_1 \underset{\mathsf{TI}}{\overset{\mathsf{FFT}}{\rightarrow}} h_0 \times N \times h_1$
18: **end for**
19: **for** $t \leftarrow d-r-2, \ldots, 0$ **do**
20: $h_0 \leftarrow \bigtimes_{s=0}^{r-1} N_s/P_s \times \bigtimes_{s=r}^{d-2-t} N_s$
21: $N \leftarrow \hat{N}_{d-1-t}$
22: $h_1 \leftarrow \bigtimes_{s=d-t}^{d-1} \hat{N}_s \times h$
23: $h_0 \times \hat{N} \times h_1 \overset{\mathsf{FFT}}{\rightarrow} h_0 \times N \times h_1$
24: **end for**

FIG. 4. *Parallel backward FFT framework.*

Remark 5. A common use case for parallel FFT is the fast convolution of two signals. Therefore, we need to compute the parallel FFT of both signals, multiply both signals pointwise, and compute the backward FFT. The pointwise multiplication can be performed trivially with transposed order of dimensions.

Remark 6. For some applications it might be unacceptable to work with transposed output after the forward FFT. As we have already seen, the backward framework reverts all transpositions of the forward framework. Therefore, execution of the forward framework followed by the backward framework, where we switch off the calculation of all one-dimensional FFTs, gives an FFT framework with nontransposed output. However, this comes at the cost of extra communication and local data transpositions.

The structure of our parallel frameworks can be easily overlooked by the flow of data distribution. Therefore, we repeat the algorithm for the important special cases of a three-dimensional FFT with one-dimensional and two-dimensional process meshes.

4.1. Example: Three-dimensional FFT with one-dimensional data decomposition. Assume a three-dimensional array of size $N_0 \times N_1 \times N_2$ that is distributed on a one-dimensional process mesh of size P_0. For this setting the parallel

forward FFT framework becomes

$$\begin{aligned} &N_0/P_0 \times N_1 \times N_2 \overset{\mathsf{FFT}}{\rightarrow} N_0/P_0 \times N_1 \times \hat{N}_2 \underset{\mathsf{TO}}{\overset{\mathsf{FFT}}{\rightarrow}} \hat{N}_1 \times N_0/P_0 \times \hat{N}_2 \\ \underset{\mathsf{TI}}{\overset{\mathsf{T}}{\rightarrow}}\ &\hat{N}_1/P_0 \times N_0 \times \hat{N}_2 \overset{\mathsf{FFT}}{\rightarrow} \hat{N}_1/P_0 \times \hat{N}_0 \times \hat{N}_2. \end{aligned}$$

The parallel backward FFT framework starts with the transposed input data and returns to the initial data distribution

$$\begin{aligned} &\hat{N}_1/P_0 \times \hat{N}_0 \times \hat{N}_2 \overset{\mathsf{FFT}}{\rightarrow} \hat{N}_1/P_0 \times N_0 \times \hat{N}_2 \underset{\mathsf{TO}}{\overset{\mathsf{T}}{\rightarrow}} \hat{N}_1 \times N_0/P_0 \times \hat{N}_2 \\ \underset{\mathsf{TI}}{\overset{\mathsf{FFT}}{\rightarrow}}\ &N_0/P_0 \times N_1 \times \hat{N}_2 \overset{\mathsf{FFT}}{\rightarrow} N_0/P_0 \times N_1 \times N_2. \end{aligned}$$

4.2. Example: Three-dimensional FFT with two-dimensional data decomposition. Assume a three-dimensional array of size $N_0 \times N_1 \times N_2$ that is distributed on a two-dimensional process mesh of size $P_0 \times P_1$. For this setting the parallel forward FFT framework becomes

$$\begin{aligned} &N_0/P_0 \times N_1/P_1 \times N_2 \underset{\mathsf{TO}}{\overset{\mathsf{FFT}}{\rightarrow}} \hat{N}_2 \times N_0/P_0 \times N_1/P_1 \\ \underset{\mathsf{TI}}{\overset{\mathsf{T}}{\rightarrow}}\ &\hat{N}_2/P_1 \times N_0/P_0 \times N_1 \underset{\mathsf{TO}}{\overset{\mathsf{FFT}}{\rightarrow}} \hat{N}_1 \times \hat{N}_2/P_1 \times N_0/P_0 \\ \underset{\mathsf{TI}}{\overset{\mathsf{T}}{\rightarrow}}\ &\hat{N}_1/P_0 \times \hat{N}_2/P_1 \times N_0 \overset{\mathsf{FFT}}{\rightarrow} \hat{N}_1/P_0 \times \hat{N}_2/P_1 \times \hat{N}_0. \end{aligned}$$

The parallel backward FFT framework starts with the transposed input data and returns to the initial data distribution

$$\begin{aligned} &\hat{N}_1/P_0 \times \hat{N}_2/P_1 \times \hat{N}_0 \overset{\mathsf{FFT}}{\rightarrow} \hat{N}_1/P_0 \times \hat{N}_2/P_1 \times N_0 \\ \underset{\mathsf{TO}}{\overset{\mathsf{T}}{\rightarrow}}\ &\hat{N}_1 \times \hat{N}_2/P_1 \times N_0/P_0 \underset{\mathsf{TI}}{\overset{\mathsf{FFT}}{\rightarrow}} \hat{N}_2/P_1 \times N_0/P_0 \times N_1 \\ \underset{\mathsf{TO}}{\overset{\mathsf{T}}{\rightarrow}}\ &\hat{N}_2 \times N_0/P_0 \times N_1/P_1 \underset{\mathsf{TI}}{\overset{\mathsf{FFT}}{\rightarrow}} N_0/P_0 \times N_1/P_1 \times N_2. \end{aligned}$$

5. The PFFT software library. We implemented the parallel FFT frameworks given by Figures 3 and 4 in a publicly available software library called PFFT. The source code is distributed under the GNU GPL at [20]. PFFT follows the philosophy of FFTW. In fact, it can be understood as an extension of FFTW to multidimensional process grids. Similar to the parallel distributed memory interface of FFTW, the user interface of PFFT splits into two layers. The basic interface depends only on the essential parameters of parallel FFT and is intended to provide an easy start with PFFT. More sophisticated adjustments of the algorithm are possible with the advanced user interface. This includes block size adjustment, automatic ghost cell creation, pruned FFTs, and the calculation of multiple FFTs with one plan. Most features of FFTW are directly inherited by our PFFT library. These include the following:

- We employ fast $\mathcal{O}(N \log N)$ algorithms of FFTW to compute arbitrary-size discrete Fourier transforms of complex data, real data, and even- or odd-symmetric real data.
- The dimension of the FFT can be arbitrary.
- PFFT offers portable performance; e.g., it will perform well on most platforms.
- The application of PFFT is split into a time consuming planning step and a high performance execution step.

- Installing the library is easy. It is based on the common sequence of configure, make, and make install.
- The interface of PFFT is very close to the MPI interface of FFTW. In fact, we tried to add as few extra parameters as possible.
- PFFT is written in C but also offers a Fortran interface.
- FFTW includes shared memory parallelism for all serial transforms. This enables us to benefit from hybrid parallelism.
- All steps of our parallel FFT can be performed completely in place. This is especially remarkable for the global transposition routines.
- Confirming to good MPI programming practice, all PFFT transforms can be performed on user defined communicators. In other words, PFFT does not enforce the user to work with `MPI_COMM_WORLD`.
- PFFT uses the same algorithm to compute the size of the local array blocks as FFTW. This implies that the FFT size need not be divisible by the number of processes.

Furthermore, we added some special features to support repeated tasks that often occur in practical application of parallel FFTs.

- PFFT includes a very flexible ghost cell exchange module. A detailed description of this module is given in section 5.1.
- PFFT accepts three-dimensional data decomposition even for three-dimensional FFTs. However, the underlying parallel FFT framework is still based on two-dimensional decomposition. A more detailed description can be found in section 5.2.
- As we already described in section 2.4, PFFT explicitly supports the parallel calculation of pruned FFTs. In section 6.3 we present some performance results of pruned FFTs.

5.1. The ghost cell module. In algorithms with block based domain decomposition processes often need to operate on data elements, which are not locally available on the current process but on one of the next nearest neighbors. PFFT assists the creation of ghost cells with a flexible module. The number of ghost cells can be chosen arbitrarily and differently in every dimension of the multidimensional array. In contrast to many other libraries, PFFT also handles the case in which the number of ghost cells exceeds the block size of the next neighboring process. This is especially important for unequal block sizes, where some processes get less data then others. PFFT uses the information about the block decomposition to determine the origin of all requested ghost cells. Furthermore, we implemented a module for the adjoint ghost cell send. The adjoint ghost cell send reduces all ghost images to their original owner and sums them up. This feature is especially useful in the case in which different processes are expected to update their ghost cells.

5.2. Remap of three-dimensional into two-dimensional decomposition. Many applications that use three-dimensional FFTs are based on a three-dimensional data decomposition throughout the rest of their implementation. Therefore, the application of our two-dimensional decomposed parallel FFT framework requires nontrivial data movement before and after every FFT. To simplify this task, we used the same ideas as in section 4 to derive a framework for the data reordering. Assume h three-dimensional arrays of total size $N_0 \times N_1 \times N_2 \times h$ to be distributed on a three-dimensional process mesh of size of size $P_0 \times P_1 \times (Q_0 \times Q_1)$ with block size $N_0/P_0 \times N_1/P_1 \times N_2/(Q_0 \times Q_1) \times h$. We do not want to calculate a serial FFT along h. Therefore, it does not count as a fourth dimension of the input array. Note that

1: $h_0 \leftarrow N_0/P_0 \times N_1/P_1$
2: $N \leftarrow N_2/(Q_0 \times Q_1)$
3: $h_1 \leftarrow h$
4: $h_0 \times N \times h_1 \underset{\mathsf{TO}}{\rightarrow} N \times h_0 \times h_1$
5:
6: $L_0 \leftarrow N_1/P_1$
7: $h_0 \leftarrow 1$
8: $L_1 \leftarrow N_2/Q_0$
9: $h_1 \leftarrow N_0/P_0$
10: $h_2 \leftarrow h$
11: $P \leftarrow Q_1$
12: $L_1/P \times h_1 \times L_0 \times h_0 \times h_2 \overset{\mathsf{T}}{\underset{\mathsf{TO}}{\rightarrow}} L_1 \times h_1 \times L_0/P \times h_0 \times h_2$
13:
14: $L_0 \leftarrow N_0/P_0$
15: $h_0 \leftarrow N_1/(P_1 \times Q_1)$
16: $L_1 \leftarrow N_2$
17: $h_1 \leftarrow 1$
18: $h_2 \leftarrow h$
19: $P \leftarrow Q_0$
20: $L_1/P \times h_1 \times L_0 \times h_0 \times h_2 \overset{\mathsf{T}}{\underset{\mathsf{TO}}{\rightarrow}} L_1 \times h_1 \times L_0/P \times h_0 \times h_2$
21:
22: $h_0 \leftarrow N_0/(P_0 \times Q_0) \times N_1/(P_1 \times Q_1)$
23: $N \leftarrow N_2$
24: $h_1 \leftarrow h$
25: $N \times h_0 \times h_1 \underset{\mathsf{TI}}{\rightarrow} h_0 \times N \times h_1$

FIG. 5. *Parallel framework for remapping three-dimensional data decomposition to two-dimensional data decomposition.*

the number of processes along the last dimension of the process mesh is assumed to be of size $Q_0 \times Q_1$. The main idea is to distribute the processes of the last dimension equally on the first two dimensions. The short notation of our data reordering framework is given by

$$\begin{aligned} & N_0/P_0 \times N_1/P_1 \times N_2/(Q_0 \times Q_1) \times h \\ \underset{\mathsf{TO}}{\rightarrow}\ & N_2/(Q_0 \times Q_1) \times N_0/P_0 \times N_1/P_1 \times h \\ \overset{\mathsf{T}}{\underset{\mathsf{TO}}{\rightarrow}}\ & N_2/Q_0 \times N_0/P_0 \times N_1/(P_1 \times Q_1) \times h \\ \overset{\mathsf{T}}{\underset{\mathsf{TO}}{\rightarrow}}\ & N_2 \times N_0/(P_0 \times Q_0) \times N_1/(P_1 \times Q_1) \times h \\ \underset{\mathsf{TI}}{\rightarrow}\ & N_0/(P_0 \times Q_0) \times N_1/(P_1 \times Q_1) \times N_2 \times h, \end{aligned}$$

and the more expressive pseudocode is listed in Figure 5. Since this framework is based on the modules that we proposed in section 3, we again benefit from cache-oblivious transpositions that are implemented within FFTW. Furthermore, this framework can be performed completely in place. To derive a framework for reordering data from two-dimensional decomposition to three-dimensional decomposition, we just need to revert all the steps of the framework from Figure 5, and so we omit the pseudocode

for this framework.

6. Numerical results/runtime measurements. In this section we show the runtime behavior of our PFFT software library in comparison to the FFTW and P3DFFT software libraries. In addition, we give some performance measurement of the pruned FFTs. The runtime tests have been performed on three different hardware architectures.

1. *BlueGene/P in Research Center Jülich (JuGene)* [1]: One node of a BlueGene/P consists of 4 IBM PowerPC 450 cores that run at 850 MHz. These 4 cores share 2 GB of main memory. Therefore, we have 0.5 GB RAM per core whenever all the cores per node are used. The nodes are connected by a three-dimensional torus network with 425 MB/s bandwidth per link. In total JuGene consists of 73728 nodes, i.e., 294912 cores.
2. *BlueGene/Q in Research Center Jülich (JuQueen)* [2]: One node of a BlueGene/Q consists of 16 IBM PowerPC A2 cores that run at 1.6 GHz. These 16 cores share 16 GB SDRAM-DDR3. Therefore, we have 1 GB RAM per core whenever all the cores per node are used. The nodes are connected by a five-dimensional torus network. In total JuQueen consists of 24576 nodes, i.e., 393216 cores.
3. *Jülich Research on Petaflop Architectures (JuRoPA)* [3]: One node of Juropa consists of 2 Intel Xeon X5570 (Nehalem-EP) quad-core processors that run at 2.93 GHz. These 8 cores share 24 GB DDR3 main memory. Therefore, we have 3 GB RAM per core whenever all the cores per node are used. The nodes are connected by an Infiniband QDR with nonblocking fat tree topology. In total JuRoPA consists of 2208 nodes, i.e., 17664 cores.

6.1. Strong scaling behavior of PFFT on BlueGene/P. We investigated the strong scaling behavior of PFFT [20] and P3DFFT [17] on the BlueGene/P machine in Research Center Jülich. Complex to complex FFTs of size 512^3 and 1024^3 have been run out-of-place with 64 of the available 72 racks, i.e., 262144 cores. Since P3DFFT supports only real to complex FFTs, we applied P3DFFT to the real and imaginary parts of a complex input array to get times comparable to those of the complex to complex FFTs of the PFFT package. The test runs consisted of 10 alternate calculations of forward and backward FFTs. Since these two transforms are inverse except for a constant factor, it is easy to check the results after each run. The average wall clock time and the average speedup of one forward and backward transformation can be seen in Figure 6 for an FFT of size 512^3 and in Figure 7 for an FFT of size 1024^3. Memory restrictions force P3DFFT to utilize at least 32 cores on BlueGene/P to calculate an FFT of size 512^3 and 256 cores to perform an FFT of size 1024. Therefore, we chose the associated wall clock times as references for speedup and efficiency calculations. Note that PFFT can perform these FFTs on half the cores because of less memory consumption. However, we only recorded times on core counts which both algorithms were able to utilize to get comparable results. Unfortunately, the PFFT test run of size 1024^3 on 64 racks died due to a hardware failure, and we were not able to repeat this large test. Nevertheless, our measurements show that the scaling behavior of PFFT and P3DFFT are quite similar. Therefore, we expect roughly the same runtime for PFFT of size 1024^3 on 64 racks as we observed for P3DFFT. It turns out that both libraries are comparable in speed. However, from our point of view the flexibility of PFFT is a great advantage over P3DFFT. See also [19] for more details.

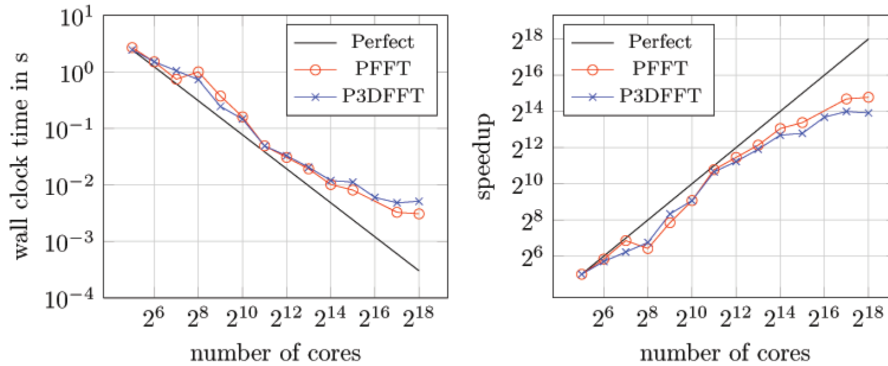

FIG. 6. *Wall clock time (left) and speedup (right) for FFT of size* 512^3 *up to* 262144 *cores on BlueGene/P.*

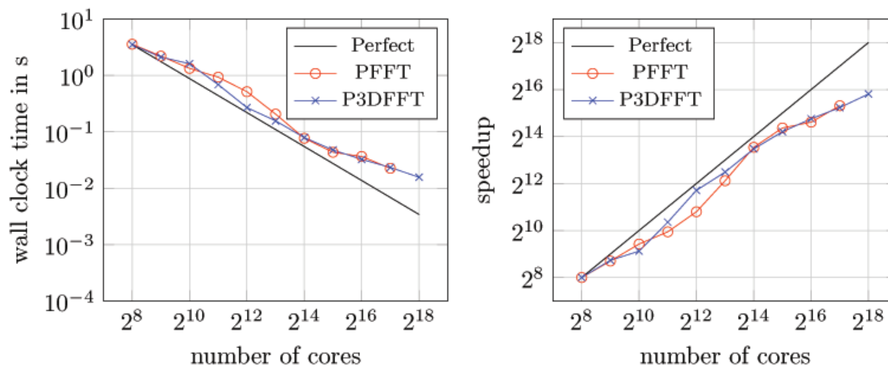

FIG. 7. *Wall clock time (left) and speedup (right) for FFT of size* 1024^3 *up to* 262144 *cores on BlueGene/P.*

6.2. Comparision of PFFT and FFTW on JuRoPA. We performed our PFFT library on the Jülich Research on Petaflop Architectures (JuRoPA) and compared the scaling behavior with the one-dimensional decomposed parallel FFTW. The runtimes of a three-dimensional FFT of size 256^3 given in Figure 8 show a good scaling behavior of our two-dimensional decomposed PFFT up to 2048 cores, while the one-dimensional data decomposition of FFTW cannot make use of more than 256 cores.

6.3. Parallel pruned FFT. As already mentioned, our parallel FFT algorithm includes the calculation of pruned multidimensional FFTs. Most of the time serial FFT libraries do not support the calculation of pruned FFTs, since the user can easily pad the input array with zeros and calculate the full FFT with the library. However, the zero padding step is not that easy in the parallel case. There we need to redistribute the data first in order to decompose the larger, zero padded input array. In addition, the parallel computation of zero padded multidimensional FFTs leads to serious load imbalance since some processes calculate one-dimensional FFTs on vectors that are full of zeros. This phenomenon is getting even worse for higher-dimensional FFTs. PFFT completely avoids the data redistribution, since it applies the one-dimensional pruned FFT algorithm (3.1) rowwise whenever the corresponding

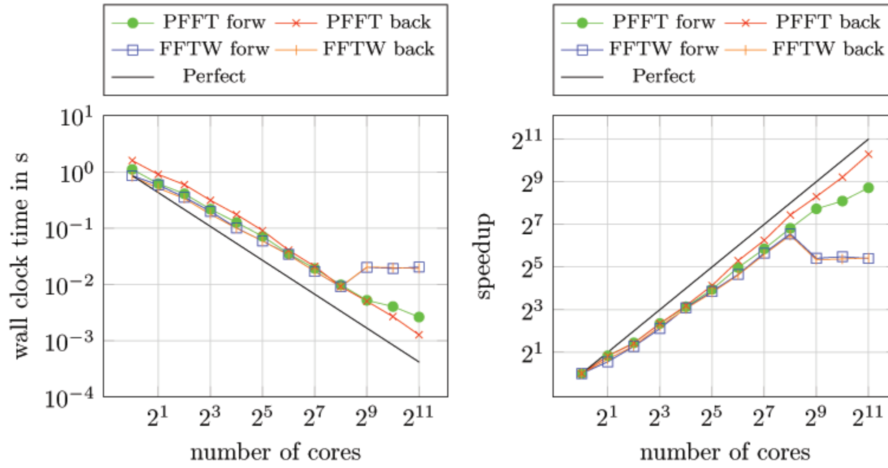

FIG. 8. *Wall clock time (left) and speedup (right) for FFT of size* 256^3 *up to* 2048 *cores on JuRoPA.*

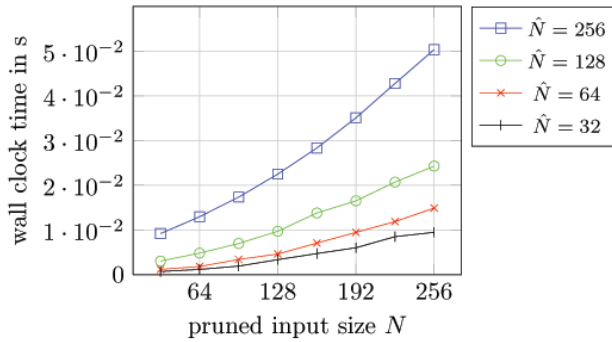

FIG. 9. *Pruned FFT with underlying FFT size* 256^3 *on* 16^2 *cores on BlueGene/P.*

data dimension is locally available on the processes.

We want to illustrate the possible performance gain with an example. Therefore, we compute a three-dimensional pruned FFT of size 256^3 on 256 cores of a BlueGene/P architecture. The data decomposition scheme is based on a two-dimensional process mesh of size 16×16. We alter the pruned input size $N \times N \times N$ and the pruned output size $\hat{N} \times \hat{N} \times \hat{N}$ between 32 and 256. Figure 9 shows the runtime of pruned PFFT for different values of N and $\hat{N}$. We observe an increasing performance benefit for decreasing input array size N and also for decreasing output array size $\hat{N}$. Without the pruned FFT support, we would have to pad the input array of size $N \times N \times N$ with zeros to the full three-dimensional FFT size $n \times n \times n$ and calculate this FFT in parallel. The time for computing an FFT of size 256^3 corresponds to the time in Figure 9 for $N = \hat{N} = 256$.

6.4. Weak scaling behavior of PFFT on BlueGene/Q and JuRoPA. In order to investigate the weak scaling behavior on BlueGene/Q we performed parallel FFTs of size 512^3, 1024^3, 2048^3, 4096^3, and 8192^3 on 8, 64, 512, 4096, and 32768 cores, respectively. This gives a constant local array size of 256^3 per process. We measured

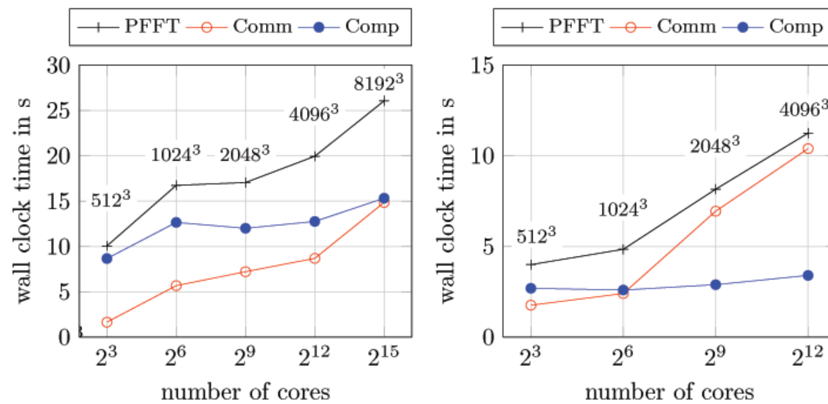

Fig. 10. *Wall clock time for FFT of constant local array size* 256^3 *per core up to* $P = 32768$ *cores on BlueGene/Q (left) and up to* $P = 2048$ *cores on JuRoPA (right). The figure includes the whole runtime of one forward and one backward FFT (PFFT) and the time spent for communication (Comm) and computation (Comp). The numbers next to data points indicate the the total FFT size.*

the average time of 10 forward and backward FFTs with transposed input/output on each process and plotted the maximum over all processes in Figure 10. We used exactly the same setting on JuRoPA but stopped at 4096 cores. The results are also given in Figure 10. In addition, we show the time that is spent for communication and computation. Note that the computational part also includes the local transposes of our serial FFT module.

6.5. Strong scaling behavior of PFFT on BlueGene/Q and JuRoPA. Finally, we compare the strong scaling behavior of our parallel in-place and out-of-place FFTs for different FFT sizes on BlueGene/Q and JuRoPA. Again, we performed 10 loops of a forward and backward FFT with transposed input/output. The maximum average time for FFTs of size 512^3, 1024^3, 2048^3, 4096^3, and 8192^3 with up to 32768 cores on BlueGene/Q are given in Figures 11, 12, 13, 14, and 15, respectively. In addition, we show the time that is spent for communication and computation. Note that the computational part also includes the local transposes of our serial FFT module. For every test run we chose the minimal possible core count to start the benchmark. We observe that the in-place transforms are indeed more memory efficient, since they allow us to run the benchmarks with smaller core counts. The out-of-place transforms are slighlty faster for large core counts. However, the in-place transforms are most important for small numbers of cores, where less memory is available. There is no difference in the performance of in-place and out-of-place FFTs for small core counts. Our parallel FFT framework provides an overall good scaling behavior. For large numbers of cores we observe some jumps of the runtimes due to the communication part. This shall be investigated in future research.

The maximum average time of 10 forward and backward FFTs of size 512^3, 1024^3, 2048^3, and 4096^3 with up to 2048 cores on JuRoPA are given in Figures 16, 17, 18, and 19, respectively. Here we see nearly the same behavior. There is even less difference in the performance of in-place and out-of-place FFTs on JuRoPA. The big jump in Figure 16 results from the fact that an in-place transposition with one single core can be totally omitted, while the out-of-place transposition needs at least one copy of the local memory.

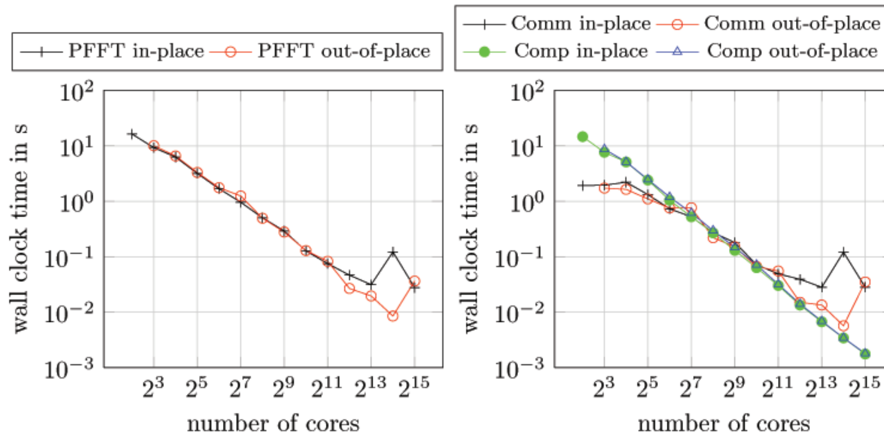

FIG. 11. *Wall clock time for in-place and out-of-place FFT of size* 512^3 *up to* $P = 32768$ *cores on BlueGene/Q. The figure includes the whole run time of one forward and one backward FFT (PFFT) and the time spent for communication (Comm) and computation (Comp).*

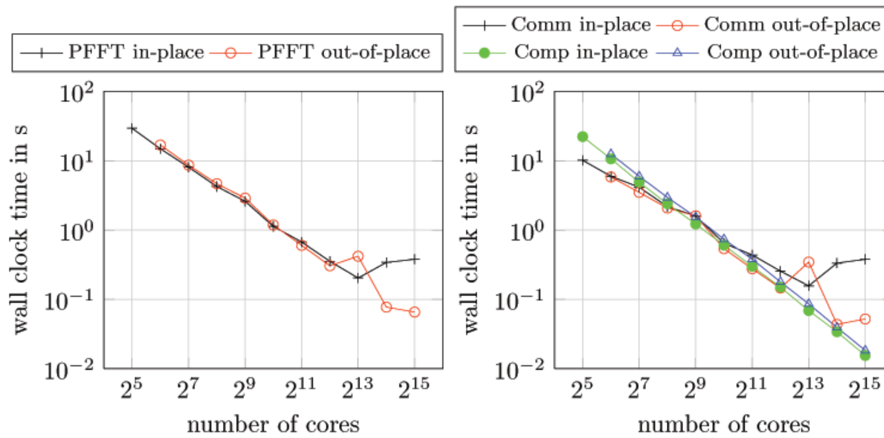

FIG. 12. *Wall clock time for in-place and out-of-place FFT of size* 1024^3 *up to* $P = 32768$ *cores on BlueGene/Q. The figure includes the whole run time of one forward and one backward FFT (PFFT) and the time spent for communication (Comm) and computation (Comp).*

7. Conclusion. We developed a parallel framework for computing arbitrary multidimensional FFTs on multidimensional process meshes. This framework has been implemented on top of the FFTW software library within a parallel FFT software library called PFFT. Our algorithms can be computed completely in place and use the hardware adaptivity of FFTW in order to achieve high performance on a wide variety of different architectures. Runtime tests up to 262144 cores of the BlueGene/P supercomputer proved PFFT to be as fast as the well-known P3DFFT software package. Therefore, PFFT is a very flexible, high performance library for computing multidimensional FFTs on massively parallel architectures.

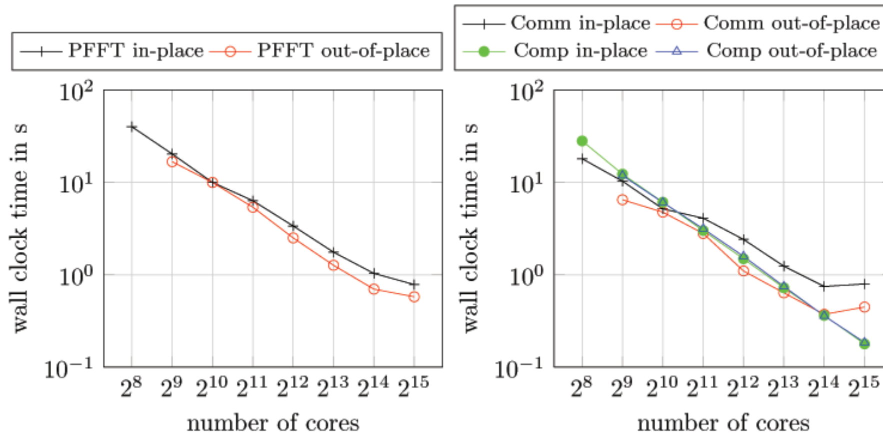

FIG. 13. *Wall clock time for in-place and out-of-place FFT of size* 2048^3 *up to* $P = 32768$ *cores on BlueGene/Q. The figure includes the whole runtime of one forward and one backward FFT (PFFT) and the time spent for communication (Comm) and computation (Comp).*

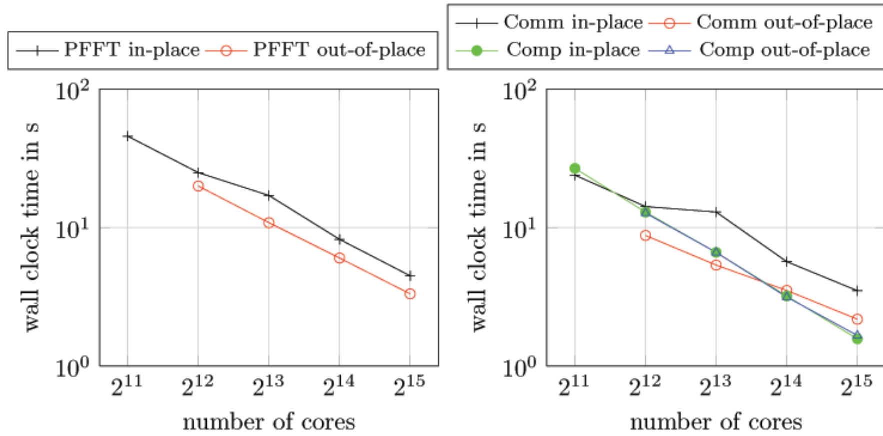

FIG. 14. *Wall clock time for in-place and out-of-place FFT of size* 4096^3 *up to* $P = 32768$ *cores on BlueGene/Q. The figure includes the whole runtime of one forward and one backward FFT (PFFT) and the time spent for communication (Comm) and computation (Comp).*

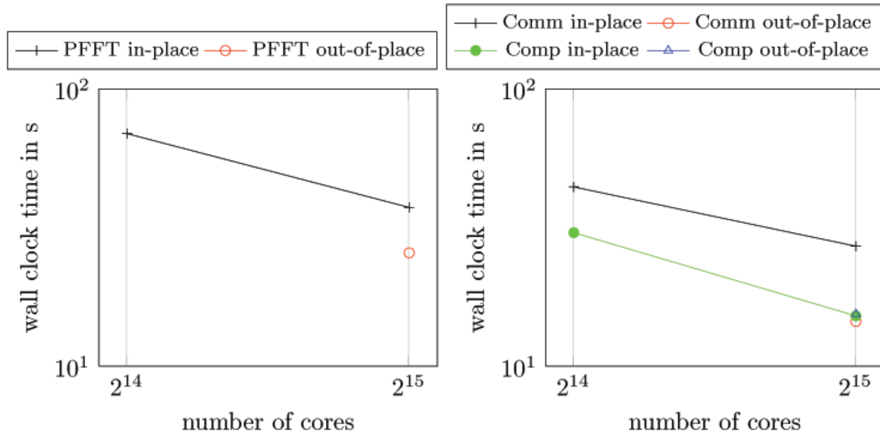

FIG. 15. *Wall clock time for in-place and out-of-place FFT of size* 8192^3 *up to* $P = 32768$ *cores on BlueGene/Q. The figure includes the whole runtime of one forward and one backward FFT (PFFT) and the time spent for communication (Comm) and computation (Comp).*

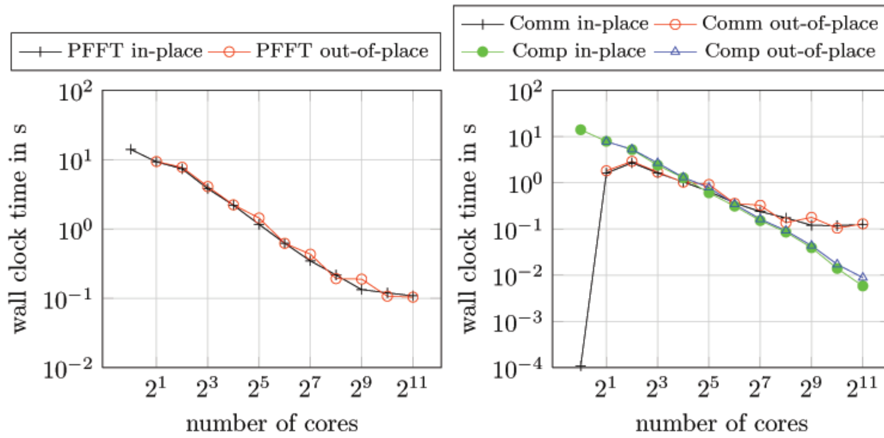

FIG. 16. *Wall clock time for in-place and out-of-place FFT of size* 512^3 *up to* $P = 2048$ *cores on JuRoPA. The figure includes the whole runtime of one forward and one backward FFT (PFFT) and the time spent for communication (Comm) and computation (Comp).*

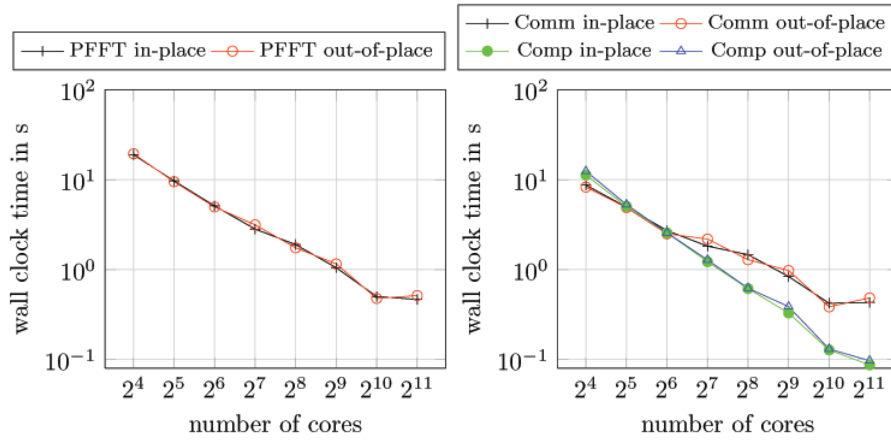

Fig. 17. *Wall clock time for in-place and out-of-place FFT of size* 1024^3 *up to* $P = 2048$ *cores on JuRoPA. The figure includes the whole runtime of one forward and one backward FFT (PFFT) and the time spent for communication (Comm) and computation (Comp).*

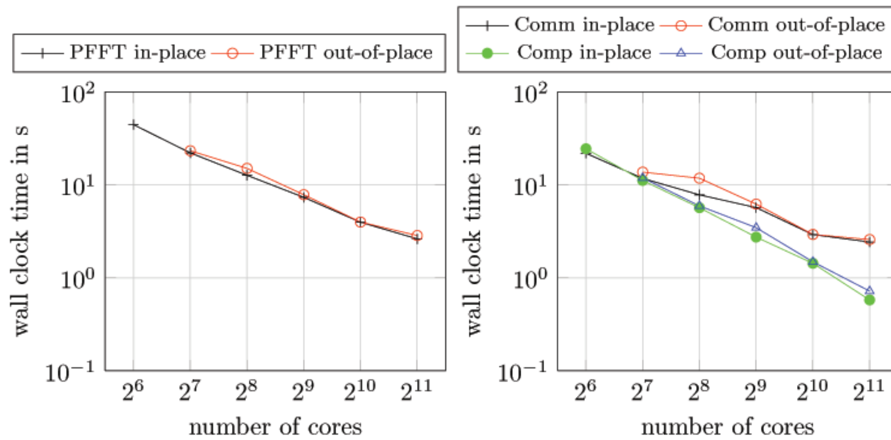

Fig. 18. *Wall clock time for in-place and out-of-place FFT of size* 2048^3 *up to* $P = 2048$ *cores on JuRoPA. The figure includes the whole runtime of one forward and one backward FFT (PFFT) and the time spent for communication (Comm) and computation (Comp).*

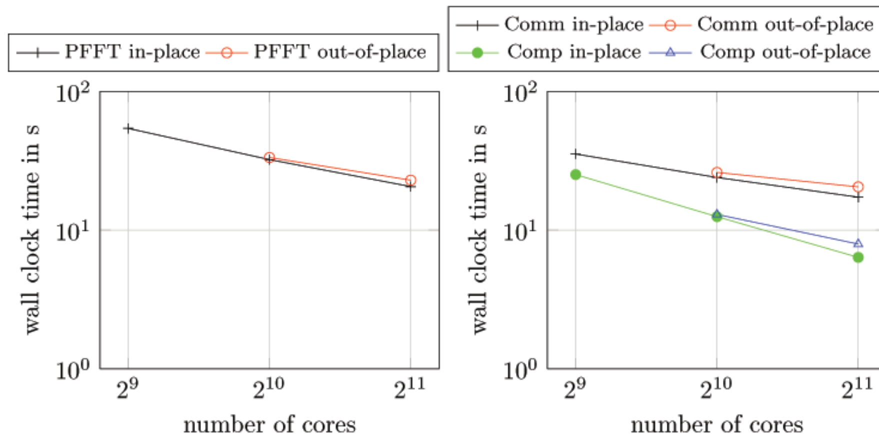

Fig. 19. *Wall clock time for in-place and out-of-place FFT of size* 4096^3 *up to* $P = 2048$ *cores on JuRoPA. The figure includes the whole runtime of one forward and one backward FFT (PFFT) and the time spent for communication (Comm) and computation (Comp).*

Acknowledgments. We are grateful to the Jülich Supercomputing Center for providing the computational resources on Jülich BlueGene/P (JuGene) and Jülich Research on Petaflop Architectures (JuRoPA). We wish to thank Sebastian Banert, who did some of the runtime measurements on JuRoPA and Jugene. Furthermore, we gratefully acknowledge the help of Dr. Ralf Wildenhues and Dr. Michael Hofmann on the PFFT build system. Last but not least, we thank the anonymous reviewers for their helpful suggestions.

REFERENCES

[1] *JuGene: Jülich Blue Gene/P*, http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUGENE/JUGENE_node.html.

[2] *JuQeen: Jülich Blue Gene/Q*, http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUQUEEN/JUQUEEN_node.html.

[3] *JuRoPA: Jülich Research on Petaflop Architectures*, http://www.fz-juelich.de/ias/jsc/EN/ Expertise/Supercomputers/JUROPA/JUROPA_node.html.

[4] J. W. Cooley and J. W. Tukey, *An algorithm for machine calculation of complex Fourier series*, Math. Comput., 19 (1965), pp. 297–301.

[5] H. Q. Ding, R. D. Ferraro, and D. B. Gennery, *A portable 3D FFT package for distributed-memory parallel architectures*, in Proceedings of the 7th SIAM Conference on Parallel Processing, SIAM, Philadelphia, 1995, pp. 70–71.

[6] M. Eleftheriou, J. E. Moreira, B. G. Fitch, and R. S. Germain, *Parallel FFT Subroutine Library*, http://www.alphaworks.ibm.com/tech/bgl3dfft.

[7] M. Eleftheriou, J. E. Moreira, B. G. Fitch, and R. S. Germain, *A volumetric FFT for BlueGene/L*, in HiPC, T. M. Pinkston and V. K. Prasanna, eds., Lecture Notes in Comput. Sci. 2913, Springer, Berlin, 2003, pp. 194–203.

[8] B. Fang, Y. Deng, and G. Martyna, *Performance of the 3D FFT on the 6D network torus QCDOC parallel supercomputer*, Comput. Phys. Comm., 176 (2007), pp. 531–538.

[9] S. Filippone, *The IBM parallel engineering and scientific subroutine library*, in PARA, J. Dongarra, K. Madsen, and J. Wasniewski, eds., Lecture Notes in Comput. Sci. 1041, Springer, Berlin, 1995, pp. 199–206.

[10] M. Frigo and S. G. Johnson, *The design and implementation of FFTW3*, Proc. IEEE, 93 (2005), pp. 216–231.

[11] M. Frigo and S. G. Johnson, *FFTW, C Subroutine Library*, http://www.fftw.org, 2009.

[12] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, *Cache-oblivious algorithms*, in Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS), IEEE Computer Society, Washington, DC, 1999, pp. 285–297.

[13] A. Gupta and V. Kumar, *The scalability of FFT on parallel computers*, IEEE Trans. Parallel Distributed Systems, 4 (1993), pp. 922–932.

[14] Intel Corporation, *Intel Math Kernel Library*, http://software.intel.com/en-us/intel-mkl/.

[15] N. Li, *2DECOMP&FFT, Parallel FFT Subroutine Library*, http://www.2decomp.org.

[16] N. Li and S. Laizet, *2DECOMP & FFT - A Highly Scalable 2D Decomposition Library and FFT Interface*, in Cray User Group 2010 Conference, Edinburgh, Scotland, 2010, pp. 1–13.

[17] D. Pekurovsky, *P3DFFT, Parallel FFT Subroutine Library*, http://code.google.com/p/p3dfft.

[18] D. Pekurovsky, *P3DFFT: A framework for parallel computations of Fourier transforms in three dimensions*, SIAM J. Sci. Comput., 34 (2012), pp. C192–C209.

[19] M. Pippig, *An efficient and flexible parallel FFT implementation based on FFTW*, in Competence in High Performance Computing (Schwetzingen, Germany, 2010), C. Bischof, H.-G. Hegering, W. E. Nagel, and G. Wittum, eds., Springer, Berlin, 2000, pp. 125–134.

[20] M. Pippig, *PFFT, Parallel FFT Subroutine Library*, http://www.tu-chemnitz.de/∼mpip/software.php, 2011.

[21] S. J. Plimpton, *Parallel FFT Subroutine Library*, http://www.sandia.gov/∼sjplimp/docs/fft/README.html.

[22] S. J. Plimpton, R. Pollock, and M. Stevens, *Particle-mesh Ewald and rRESPA for parallel molecular dynamics simulations*, in Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing (Minneapolis, 1997), SIAM, Philadelphia, 1997.

[23] D. Takahashi, *An implementation of parallel 3-D FFT with 2-D decomposition on a massively parallel cluster of multi-core processors*, in Parallel Processing and Applied Mathematics, R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, eds., Lecture Notes in Comput. Sci. 6067, Springer, Berlin, 2010, pp. 606–614.