PFFT - An Extension of FFTW to Massively Parallel Architectures

Michael Pippig
Departement of Mathematics
Chemnitz University of Technology
09107 Chemnitz, Germany
Email: michael.pippig@mathematik.tu-chemnitz.de

***Abstract*—We present a MPI based software library for computing the fast Fourier transforms on massively parallel, distributed memory architectures. 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 multi-dimensional 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. Likewise 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 size 512^3 and 1024^3 up to 262144 cores on a BlueGene/P architecture.**

I. 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 J. W. Cooley and J. Tukey [1] has been published in 1965, a lot of algorithms were derived for computing the discrete Fourier transform in $\mathcal{O}(n \log n)$. This variety of algorithms and the continuously 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 best 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 3.3.1 of FFTW also includes a very flexible distributed memory parallelization based on MPI. However, the underlying parallel algorithm is not suitable for current massive 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.

There are two main approaches for parallelizing multi-dimensional FFTs, first binary exchange algorithms and second transpose algorithms. An introduction and theoretical comparison can be found in [2]. We want to concentrate on transpose algorithms, i.e., we perform a sequence of local one-dimensional FFTs and two-dimensional data transpositions. For convenience we consider the three-dimensional input array to be of size $n_0 \times n_1 \times n_2$ with $n_0 \geq n_1 \geq n_2$.

First parallel transpose FFT algorithms were based on so-called slab decomposition, which means that the three-dimensional input array is split along n_0 to distribute it on a given number $P \leq n_0$ of MPI processes. At the first step, n_0 two-dimensional FFTs of size $n_1 \times n_2$ can be computed, since all required data reside locally on the processes. Only $n_1 n_2$ one-dimensional FFTs of size n_0 remain to complete the three-dimensional FFT but the required data is distributed among all processes. Therefore, a data transposition is performed first that corresponds to a call of MPI Alltoall. Implementations of the one-dimensional decomposed parallel FFT are for example included in the IBM PESSL library [3], the Intel Math Kernel Library [4] and the FFTW [5] software package. Unfortunately, all of these FFT libraries lack high scalability on massively parallel architectures because their data distribution approach limits the number of efficiently usable MPI processes by n_1. Fig. 1 shows an illustration of the one-dimensional distributed FFT and an example of its scalability limitation.

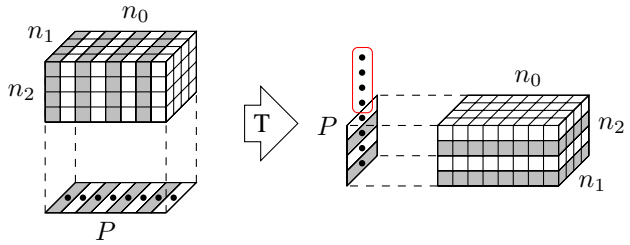

Fig. 1. Decomposition of a three-dimensional array of size $n_0 \times n_1 \times n_2 = 8 \times 4 \times 4$ on a one-dimensional process grid of size $P = 8$. After the transposition (T) half of the processes remain idle.

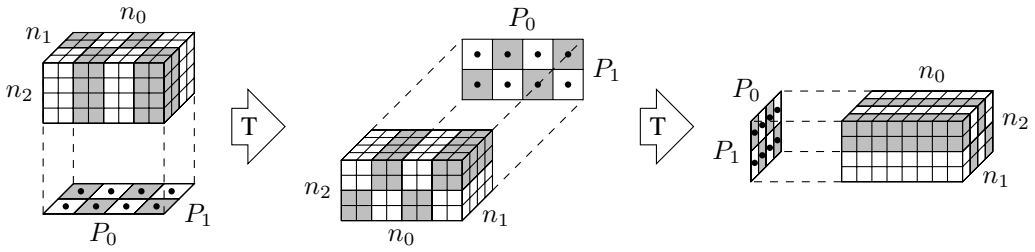

Fig. 2. Distribution of a three-dimensional array of size $n_0 \times n_1 \times n_2 = 8 \times 4 \times 4$ on a two-dimensional process grid of size $P_0 \times P_1 = 4 \times 2$. None of the processes remains idle in any calculation step.

The main idea of overcome this scalability bottleneck is to use a two-dimensional data decomposition. It was first proposed by Ding et al. [6] in 1995. Eleftheriou et al. [7] introduced a volumetric domain decomposition and implemented a software library [8] for power of two FFTs customized to the BlueGene/L architecture. However, it turns out that the underlying parallel FFT algorithm still uses a two-dimensional data decomposition. 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 and therefore allows to increase the number of MPI processes to at most $n_1 n_2$. The three-dimensional FFT is performed via three successive one-dimensional FFTs. Since only one dimension of the data array remains local, two data transpositions must be performed between. Fig. 2 shows an illustration of the two-dimensional distributed FFT and its improved scalability in comparison to the example above.

Public available implementations of the two-dimensional decomposition approach are the FFT package [9] by Plimpton from Sandia National Laboratories, the P3DFFT [10] library by Pekurovsky and more recently the 2DECOMP&FFT library [11], [12] by Li. Furthermore, performance evaluations of two-dimensional decomposed parallel FFTs have been published by Fang et al. [13] and Takahashi [14].

All these implementations offer a different set of features and introduce their own interface. By our knowledge, there is no public available FFT library, that supports process meshes with more then two dimensions. Our parallel FFT framework aims to close this gap and offer one library for all the above mentioned use cases with an FFTW-like interface. In fact we sort of extended the distributed memory parallel FFTW to multi-dimensional data decompositions. To our knowledge, PFFT is the first public available parallel FFT library that supports two-dimensional decomposition for arbitrary multi-dimensional FFTs. Last but not least, apart from PFFT we do not know any public available FFT library that is able to compute parallel sine and cosine transforms based on a multi-dimensional data decomposition.

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

II. 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 my 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.

A. 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 of size n is defined as

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

Evaluating all $\hat{f}_l$ by direct summation requires $\mathcal{O}(n^2)$ arithmetic operations. In 1965 J. W. Cooley and J. Tukey published an algorithm called Fast Fourier Transform (FFT) [1] that reduces the arithmetic complexity to $\mathcal{O}(n \log n)$. Furthermore, we define the backward discrete Fourier transform of size n by

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

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 discrete Fourier transform of complex data by the abbreviation c2c-FFT, since they transform complex inputs into complex outputs.

B. One-dimensional FFT of real data

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

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

Since the outputs satisfy the Hermitian symmetry

$$\hat{f}_{n-l} = \hat{f}_l^*, \quad l = 0, \ldots, n/2 - 1 \,,$$

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

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

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

C. One-dimensional FFT of even- or odd-symmetric real data

Depending on the symmetry of the input data, there exist 16 different definitions of discrete Fourier transforms of even- or odd-symmetric real data. At this point, we only give the definition of the most commonly used discrete cosine transform of second kind. The definitions of the other transforms can be found for example in the FFTW manual [15].

Consider n real numbers $f_k \in \mathbb{R}$, $k = 0, \ldots, n-1$. The one-dimensional discrete cosine transform of second kind (DCT-II) is given by

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

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

D. Pruned FFTs

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

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

This means, that we are only interested in 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.,

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

Second, calculate the sums

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

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.,

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

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

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

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

E. Multi-dimensional FFTs

Assume an multi-dimensional input array of $n_0 \times \ldots \times n_{d-1}$ real or complex numbers. We define the multi-dimensional FFT as the consecutive calculation of the one-dimensional FFTs along the dimensions of the input array.

Again, we have to pay special attention on r2c-transforms. Here, we first compute the one-dimensional r2c-FFTs along the last dimension of the multi-dimensional array. Because of Hermitian symmetry the output array consist of $n_0 \times \ldots \times n_{d-2} \times (n_{d-1}/2 + 1)$ 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.

F. Parallel data decomposition

Assume a multi-dimensional array of size $N_0 \times \ldots \times N_{d-1}$. Furthermore, for $r < d$ assume a r-dimensional Cartesian communicator, which includes a mesh of $P_0 \times \ldots \times P_{r-1}$ MPI processes. Our parallel algorithms are based on a simple block structured domain decomposition, i.e., every process owns a block of $N_0/P_0 \times \ldots \times N_{r-1}/P_{r-1} \times N_r \times \ldots \times N_{d-1}$ 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 $i = 0, \ldots, r-1$ and $j = 0, \ldots, d-1$. 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 that also 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 $i = 0, \ldots, d-1$ and $j = 0, \ldots, r-1$.

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 multi-dimensional 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 multi-dimensional array into equal pieces corresponds to a splitting of the array along the first dimension.

III. 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 them are somehow already implemented in the current release 3.3.1 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.

A. The serial FFT module

The guru interface of FFTW offers a very general way to compute multi-dimensional vector loops of multi-dimensional FFTs [5]. However, we do not need the full generality and therefore wrote a wrapper that enables us to compute multi-dimensional 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.,

$$h_0 \times n \times h_1 \overset{\mathsf{FFT}}{\to} h_0 \times \hat{n} \times h_1 \,. \tag{1}$$

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 on, 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 FFT can be any of the serial FFTs that we introduced in Section II, 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 non-trivial task because one has to think of many details about the memory hierarchy of current computer architectures. FFTW implemented cache oblivious array transpositions [16], which aim to minimize the asymptotic number of cache misses independent of the cache size. Unfortunately, we experienced that the performance of an FFT combined with the local transposition is sometimes quiet 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 multi-dimensional 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 of the serial FFT to performe the local transpositions

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

and

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

solely.

Remark 1: Beside the sequence 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.

B. The serial pruned FFT module

The serial FFTs can be easily generalized to pruned FFTs with the three-step algorithm from Section II-D. The padding with zeros and the truncation step have been implemented as modules in PFFT. To keep notation simple, we do not introduce further symbols to mark a serial FFT as 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

$$h_0 \times N \times h_1 \overset{\mathsf{FFT}}{\to} h_0 \times \hat{N} \times h_1 \tag{2}$$

abbreviates the three-step pruning algorithm

$$\begin{array}{rcl} h_0 \times N \times h_1 & \to & h_0 \times n \times h_1 \\ \overset{\mathsf{FFT}}{\to} \quad h_0 \times \hat{n} \times h_1 & \to & h_0 \times \hat{N} \times h_1 \,. \end{array}$$

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

$$N \times h_0 \times h_1 \underset{\mathsf{TI}}{\overset{\mathsf{FFT}}{\to}} h_0 \times \hat{N} \times h_1 \,, \tag{3}$$

$$h_0 \times N \times h_1 \underset{\mathsf{TO}}{\overset{\mathsf{FFT}}{\to}} \hat{N} \times h_0 \times h_1 \,. \tag{4}$$

C. 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 FFTW3.3.1 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 FFTWs one-dimensional decomposed parallel FFT implementations. In addition, FFTWs global transposition algorithm 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

$$\begin{aligned} N_0/P \times N_1 \times h &\overset{\mathsf{T}}{\rightarrow} N_1/P \times N_0 \times h\,, \\ N_1 \times N_0/P \times h &\underset{\mathsf{TI}}{\overset{\mathsf{T}}{\rightarrow}} N_1/P \times N_0 \times h\,, \\ N_0/P \times N_1 \times h &\underset{\mathsf{TO}}{\overset{\mathsf{T}}{\rightarrow}} N_0 \times N_1/P \times h\,. \end{aligned} \tag{5}$$

There are great advantages of using FFTWs parallel transposition algorithms instead of direct calls to corresponding MPI functions. FFTW does not only use one algorithm to perform a 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. Furthermore, all transpositions can be performed in place, which is impossible by calls to MPIs standard Alltoall functions and hard to program in an efficient way with point to point communications. However, we need a slightly generalization of FFTWs transpositions to make it suitable to 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 (5) turn into

$$\begin{aligned} & L_1/P \times h_1 \times L_0 \times h_0 \times h_2 \\ \overset{\mathsf{T}}{\rightarrow}\; & L_0/P \times h_0 \times L_1 \times h_1 \times h_2\,, \end{aligned} \tag{6}$$

$$\begin{aligned} & L_0 \times h_0 \times L_1/P \times h_1 \times h_2 \\ \underset{\mathsf{TI}}{\overset{\mathsf{T}}{\rightarrow}}\; & L_0/P \times h_0 \times L_1 \times h_1 \times h_2\,, \end{aligned} \tag{7}$$

$$\begin{aligned} & L_1/P \times h_1 \times L_0 \times h_0 \times h_2 \\ \underset{\mathsf{TO}}{\overset{\mathsf{T}}{\rightarrow}}\; & L_1 \times h_1 \times L_0/P \times h_0 \times h_2\,. \end{aligned} \tag{8}$$

Remark 1: Although this substitution looks straightforward, we must choose the block sizes carefully. Whenever P does not divide L_0 or L_1, we can not use FFTWs default block sizes $(L_0 \times h_0)/P$ and $(L_1 \times h_1)/P$. Instead we must assure, 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 2: Similar to FFTW, our global data transpositions operate on real numbers only. However, complex arrays that store real and imaginary part in the typical interleaved way can be seen as arrays of real pairs. Therefore, we only need to double h_2 to initiate the communication for complex arrays.

IV. The parallel FFT framework

Now, we have collected all the ingredients to formulate the parallel FFT framework that allows us to calculate h pruned multi-dimensional FFTs of size

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

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

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

For convenience, we introduce the notation

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

Fig. 3 lists the pseudo code of the parallel forward FFT framework.

1: **for** $t \leftarrow 0, \ldots, d-r-2$ **do**
2: $\quad h_0 \leftarrow \bigtimes_{s=0}^{r-1} N_s/P_s \times \bigtimes_{s=r}^{d-2-t} N_s$
3: $\quad N \leftarrow N_{d-1-t}$
4: $\quad h_1 \leftarrow \bigtimes_{s=d-t}^{d-1} \hat{N}_s \times h$
5: $\quad h_0 \times N \times h_1 \overset{\mathsf{FFT}}{\rightarrow} h_0 \times \hat{N} \times h_1$
6: **end for**
7: **for** $t \leftarrow 0, \ldots, r-1$ **do**
8: $\quad h_0 \leftarrow \bigtimes_{s=r-t}^{r-1} \hat{N}_{s+1}/P_s \times \bigtimes_{s=0}^{r-t-1} N_s/P_s$
9: $\quad N \leftarrow N_{r-t}$
10: $\quad h_1 \leftarrow \bigtimes_{s=r+1}^{d-1} \hat{N}_s \times h$
11: $\quad h_0 \times N \times h_1 \underset{\mathsf{TO}}{\overset{\mathsf{FFT}}{\rightarrow}} \hat{N} \times h_0 \times h_1$
12:
13: $\quad L_0 \leftarrow \hat{N}_{r-t}$
14: $\quad h_0 \leftarrow \bigtimes_{s=r-t}^{r-1} \hat{N}_{s+1}/P_s \times \bigtimes_{s=0}^{r-t-2} N_s/P_s$
15: $\quad L_1 \leftarrow N_{r-t-1}$
16: $\quad h_1 \leftarrow 1$
17: $\quad h_2 \leftarrow \bigtimes_{s=r+1}^{d-1} \hat{N}_s \times h$
18: $\quad P \leftarrow P_{r-t-1}$
19: $\quad L_0 \times h_0 \times L_1/P \times h_1 \times h_2 \underset{\mathsf{TI}}{\overset{\mathsf{T}}{\rightarrow}} L_0/P \times h_0 \times L_1 \times h_1 \times h_2$
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 (2) 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 (4) interleaved by global data transpositions with transposed input (7). Finally, a single non-transposed FFT (2) must be computed to finish the full d-dimensional FFT. The data decomposition of the output is then given by

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

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

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

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

and ends with the initial data decomposition

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

Fig. 4 lists the parallel backward FFT framework in pseudo code.

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

Fig. 4. Parallel Backward FFT Framework

Remark: 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 a FFT framework with non-transposed 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.

A. 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

$$\begin{array}{l} N_0/P_0 \times N_1 \times N_2 \overset{\mathsf{FFT}}{\rightarrow} N_0/P_0 \times N_1 \times \hat{N}_2 \\ \overset{\mathsf{FFT}}{\underset{\mathsf{TO}}{\rightarrow}} (\hat{N}_1 \times N_0/P_0) \times \hat{N}_2 \overset{\mathsf{T}}{\underset{\mathsf{TI}}{\rightarrow}} (\hat{N}_1/P_0 \times N_0) \times \hat{N}_2 \\ \overset{\mathsf{FFT}}{\rightarrow} \hat{N}_1/P_0 \times \hat{N}_0 \times \hat{N}_2\,. \end{array}$$

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

$$\begin{array}{l} \hat{N}_1/P_0 \times \hat{N}_0 \times \hat{N}_2 \overset{\mathsf{FFT}}{\rightarrow} (\hat{N}_1/P_0 \times N_0) \times \hat{N}_2 \\ \overset{\mathsf{T}}{\underset{\mathsf{TO}}{\rightarrow}} (\hat{N}_1 \times N_0/P_0) \times \hat{N}_2 \overset{\mathsf{FFT}}{\underset{\mathsf{TI}}{\rightarrow}} N_0/P_0 \times N_1 \times \hat{N}_2 \\ \overset{\mathsf{FFT}}{\rightarrow} N_0/P_0 \times N_1 \times N_2\,. \end{array}$$

B. 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

$$\begin{array}{l} N_0/P_0 \times N_1/P_1 \times N_2 \overset{\mathsf{FFT}}{\underset{\mathsf{TO}}{\rightarrow}} (\hat{N}_2 \times N_0/P_0) \times N_1/P_1 \\ \overset{\mathsf{T}}{\underset{\mathsf{TI}}{\rightarrow}} (\hat{N}_2/P_1 \times N_0/P_0) \times N_1 \overset{\mathsf{FFT}}{\underset{\mathsf{TO}}{\rightarrow}} (\hat{N}_1 \times \hat{N}_2/P_1) \times N_0/P_0 \\ \overset{\mathsf{T}}{\underset{\mathsf{TI}}{\rightarrow}} (\hat{N}_1/P_0 \times \hat{N}_2/P_1) \times N_0 \overset{\mathsf{FFT}}{\rightarrow} \hat{N}_1/P_0 \times \hat{N}_2/P_1 \times \hat{N}_0\,. \end{array}$$

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

$$\begin{array}{l} \hat{N}_1/P_0 \times \hat{N}_2/P_1 \times \hat{N}_0 \overset{\mathsf{FFT}}{\rightarrow} (\hat{N}_1/P_0 \times \hat{N}_2/P_1) \times N_0 \\ \overset{\mathsf{T}}{\underset{\mathsf{TO}}{\rightarrow}} (\hat{N}_1 \times \hat{N}_2/P_1) \times N_0/P_0 \overset{\mathsf{FFT}}{\underset{\mathsf{TI}}{\rightarrow}} (\hat{N}_2/P_1 \times N_0/P_0) \times N_1 \\ \overset{\mathsf{T}}{\underset{\mathsf{TO}}{\rightarrow}} (\hat{N}_2 \times N_0/P_0) \times N_1/P_1 \overset{\mathsf{FFT}}{\underset{\mathsf{TI}}{\rightarrow}} N_0/P_0 \times N_1/P_1 \times N_2\,. \end{array}$$

V. The PFFT software library

We implemented the parallel FFT frameworks given by Fig. 3 and Fig. 4 in a public available software library called PFFT. The source code is distributed under the GNU GPL at [17]. PFFT follows the philosophy of FFTW. In fact it can be understood as an extension of FFTW to multi-dimensional 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 to our PFFT library. This includes the following.

- We employ FFTWs fast $\mathcal{O}(N \log N)$ algorithms 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 good on most platforms.
- The application of PFFT is split into a time consuming planning and a high performance execution step.
- Installing the library is easy. It is based on the common sequence of configure, make, make install.
- The interface of PFFT is very close to the MPI interface of FFTW. In fact, we tried to add as little 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 does not need to 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 V-A.
- 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 V-B.
- As we already described in Section II-D, PFFT explicitly supports the parallel calculation of pruned FFTs. In Section VI-C we present some performance results of PFFTs pruned FFTs.

A. The ghost cell module

In algorithms with block based domain decomposition it is a often necessary that processes need to operate on data elements, which are not locally available on the current process but on one of the next nearest neighbors. PFFT assist the creation of ghost cells with a flexible module. The number of ghost cells can be chosen arbitrary and different in every dimension of the multi-dimensional array. In contrast to many other libraries, PFFT also handles the case where 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 all owner processes of the requested ghost cells. Furthermore, we implemented a module for the adjoint ghost cell send. The adjoint ghost cell send does a reduces of all ghost images to their original owner and sums them up. This feature is especially useful in the case where different processes are expected to update their ghost cells.

B. 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 non-trivial data movement before and after every FFT. To simplify this task, we used the same ideas as in Section IV to derive a framework for the data reordering. Assume a three-dimensional array of 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, is does not account as fourth dimension of the input array. Note, that 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

$$
\begin{aligned}
& N_0/P_0 \times N_1/P_1 \times N_2/(Q_0 \times Q_1) \times h \\
\xrightarrow[\mathsf{TO}]{} \quad & N_2/(Q_0 \times Q_1) \times N_0/P_0 \times N_1/P_1 \times h \\
\xrightarrow[\mathsf{TO}]{\mathsf{T}} \quad & N_2/Q_0 \times N_0/P_0 \times N_1/(P_1 \times Q_1) \times h \\
\xrightarrow[\mathsf{TO}]{\mathsf{T}} \quad & N_2 \times N_0/(P_0 \times Q_0) \times N_1/(P_1 \times Q_1) \times h \\
\xrightarrow[\mathsf{TI}]{} \quad & N_0/(P_0 \times Q_0) \times N_1/(P_1 \times Q_1) \times N_2 \times h
\end{aligned}
$$

and the more expressive pseudo code is listed in Fig. 5. Since this framework is based on the modules that we proposed in Section III, we again benefit of FFTWs cache oblivious transpositions. 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 Fig. 5. We relinquish to list the pseudo code for this framework.

VI. Numerical results/Runtime measurements

In this section we show the runtime behavior or our PFFT software library in comparison to the FFTW and P3DFFT software libraries. In addition, we give some performance measurement of the pruned FFTs.

A. Strong scaling behavior of PFFT on BlueGene/L

In [18] the strong scaling behavior of PFFT [17] and P3DFFT [10] up to the full BlueGene/P machine in Reasearch Center Jülich has been investigated. Complex to complex FFTs

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: $L_1/P \times h_1 \times L_0 \times h_0 \times h_2 \underset{\mathsf{TO}}{\overset{\mathsf{T}}{\rightarrow}} L_1 \times h_1 \times L_0/P \times h_0 \times h_2$
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: $L_1/P \times h_1 \times L_0 \times h_0 \times h_2 \underset{\mathsf{TO}}{\overset{\mathsf{T}}{\rightarrow}} L_1 \times h_1 \times L_0/P \times h_0 \times h_2$
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

of size 513^3 and 1024^3 have been performed on up to 64 of the available 72 racks, i.e., 262144 cores. It turned out, that both libraries are comparable in speed. However, from our point of view the flexibility of PFFT is a great advantage over P3DFFT.

B. Performance measurements on JuRoPA

We also 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 Fig. 6 and Fig. 7 show a good scaling behavior of our two-dimensional decomposed PFFT up to 2048 cores, while the one-dimensional data decomposition of FFTW can not make use of more than 256 cores.

C. Parallel pruned FFT

As already mentioned, our parallel FFT algorithm includes the calculation of pruned multi-dimensional 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 zero and calculate the full FFT with the library. However, it is not that easy in the parallel case since data needs to redistributed on all processes. In addition, the computation of zero padded multi-dimensional FFTs leads to serious load imbalance as some processes calculate one-dimensional FFTs

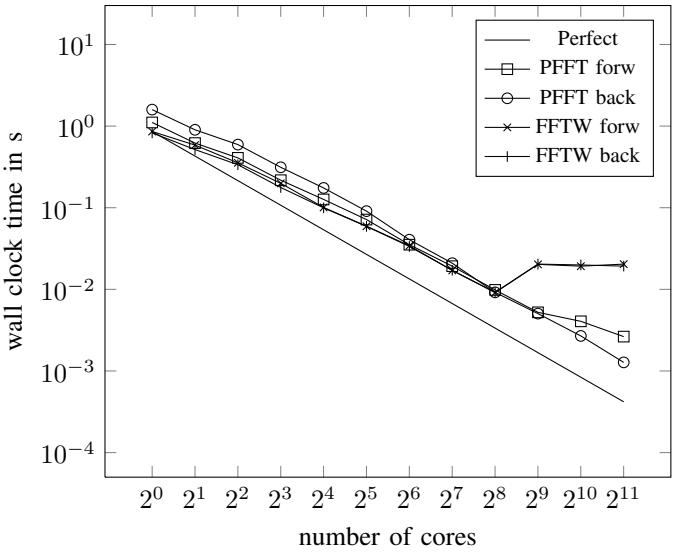

Fig. 6. Walltime for FFT of size 256^3 up to 2048 cores on JuRoPA

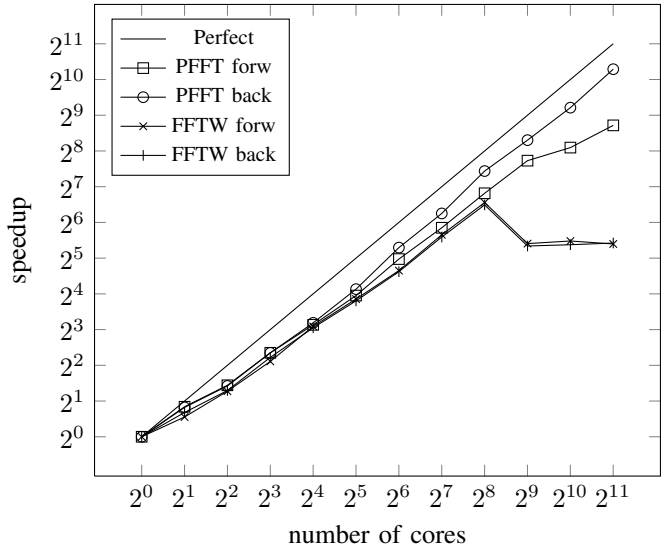

Fig. 7. Speedup for FFT of size 256^3 up to 2048 cores on JuRoPA

on vectors that are full of zeros. This phenomena is getting even worse for higher-dimensional FFTs. PFFT avoid this data reordering, since it applies the one-dimensional pruned FFT algorithm 2 row wise, whenever the corresponding data dimension is locally available on the processes.

We want to illustrate the possible performance gain with an example. Therefore, we computed a three-dimensional pruned FFT of size $n \times n \times n$, $n = 256$, on 256 cores of a BlueGene/P architecture. The data decomposition scheme was based on a two-dimensional process mesh of size 16×16. We altered 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. Fig. 8 shows the runtime of pruned PFFT for different values of N and $\hat{N}$. We can observe an increasing performance benefit for decreasing input array size N and also for decreasing output array size $\hat{N}$. Without

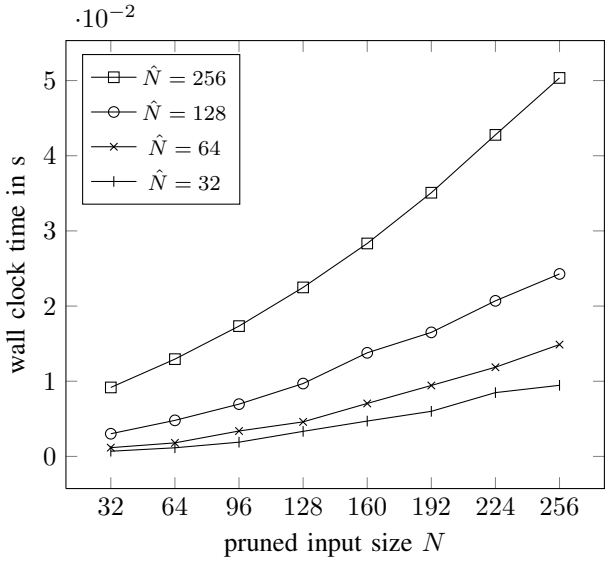

Fig. 8. Pruned FFT with underlying FFT size 256^3 on 16^2 cores of BlueGene/P

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 a FFT of size 256^3 corresponds to the time in Fig. 8 for $N = \hat{N} = 256$.

VII. Conclusion

We developed a parallel framework for computing arbitrary multi-dimensional FFTs on multi-dimensional 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 multi-dimensional FFTs on massively parallel architectures.

Acknowledgment

This work was supported by the BMBF grant 01IH08001B. 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 Ralf Wildenhues and Michael Hofmann on the PFFT build system.

References

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

[2] A. Gupta and V. Kumar, "The scalability of FFT on parallel computers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, pp. 922 – 932, 1993.

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

[4] Intel Corporation, "Intel math kernel library." [Online]. Available: http://software.intel.com/en-us/intel-mkl/

[5] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the* IEEE, vol. 93, pp. 216 – 231, 2005.

[6] H. Q. Ding, R. D. Ferraro, and D. B. Gennery, "A portable 3d FFT package for distributed-memory parallel architectures," in *PPSC*, 1995, pp. 70 – 71.

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

[8] ——, "Parallel FFT subroutine library." [Online]. Available: http://www.alphaworks.ibm.com/tech/bgl3dfft

[9] S. Plimpton, "Parallel FFT subroutine library." [Online]. Available: http://www.sandia.gov/~sjplimp/docs/fft/README.html

[10] D. Pekurovsky, "P3DFFT, Parallel FFT subroutine library." [Online]. Available: http://www.sdsc.edu/us/resources/p3dfft

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

[12] N. Li, "2DECOMP&FFT, Parallel FFT subroutine library." [Online]. Available: http://www.2decomp.org

[13] B. Fang, Y. Deng, and G. Martyna, "Performance of the 3D FFT on the 6D network torus QCDOC parallel supercomputer," *Computer Physics Communications*, vol. 176, no. 8, pp. 531–538, apr 2007. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0010465507000276

[14] 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*, ser. Lecture Notes in Computer Science, R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, Eds. Springer Berlin / Heidelberg, 2010, vol. 6067, pp. 606–614. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-14390-8_63

[15] M. Frigo and S. G. Johnson, "FFTW, C subroutine library," `http://www.fftw.org`, 2009. [Online]. Available: http://www.fftw.org

[16] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *Proc. 40th Ann. Symp. on Foundations of Comp. Sci. (FOCS)*. IEEE Comput. Soc., 1999, pp. 285 – 297.

[17] M. Pippig, "PFFT, Parallel FFT subroutine library." [Online]. Available: http://www.tu-chemnitz.de/~mpip

[18] ——, "An Efficient and Flexible Parallel FFT Implementation Based on FFTW," in *Competence in High Performance Computing*, C. Bischof, H.-G. Hegering, W. E. Nagel, and G. Wittum, Eds. Schwetzingen, Germany: Springer, Jun. 2010, pp. 125 – 134. [Online]. Available: http://www.springerlink.com/content/978-3-642-24025-6