PFFT User Manual*

for version 1.0.7-alpha

December 30, 2018

Michael Pippig
michael.pippig.tuc@gmail.com

Download **P**arallel **F**ast **F**ourier **T**ransform Software Library at
www.tu-chemnitz.de/~mpip/software.php
https://github.com/mpip/pfft.git

*This work was supported by the BMBF grant 01IH08001B from 01.01.2010 until 31.03.2013.

Todo list

Contents

This user manual describes the usage of PFFT 1.0.7-alpha [18, 20], a MPI-based, parallel software library for the computation of equispaced fast Fourier transforms (FFT) on parallel, distributed memory architectures. The reader of this manual should familiar with the basic usage of FFTW and MPI. For further information we refer to the well written FFTW user manual [1] and the MPI Standard [15], see also [12] for detailed explanations.

1 Introduction

A popular software library for computing FFTs is FFTW [11, 10]. This library also includes a parallel FFT implementation (FFTW-MPI) based on the Message Passing Interface (MPI). FFTW-MPI parallelizes multi-dimensional FFTs by a mixture of serial lower-dimensional FFTs and parallel data transpositions. However, FFTW-MPI makes use of a one-dimensional data decomposition, which shows to be a scalability bottleneck on large scale, parallel computers. For example, a three-dimensional FFT of size 1024^3 can be computed with at most 1024 MPI processes. In contrast, using a two-dimensional data decomposition would increase the maximum number of MPI processes to 1024^2 in this case.

The main goal of PFFT is to extend the MPI part of the FFTW software library to multi-dimensional data decompositions, i.e., d-dimensional FFTs of size N^d can be computed in parallel with at most N^{d-1} MPI processes. In addition, PFFT offers several extra features that are particular usefull for parallel, distributed memory FFTs but are not yet present in FFTW-MPI. We refer to the publication [20] for a closer look on the different data decompositions and the underlying algorithms of the PFFT library.

The interface of PFFT is as close as possible to the FFTW-MPI interface. In fact, we consider every difference between PFFT and FFTW that is not explicitly mentioned within this manual as a bug that should be reported to `https://github.com/mpip/pfft.git`. Therefore, porting code that uses FFTW-MPI to PFFT is almost trivial, e.g. see Section 2.2.

Most features of PFFT are inherited from FFTW or similarily implemented. 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. However, parallel data decomposition must be at least one dimension smaller.
- 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, see Section **??**.
- FFTW includes shared memory parallelism for all serial transforms. This enables us to benefit from hybrid parallelism to a certain amount, see Section **??**.
- 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.
- PFFT supports single, double and long double precision.
- PFFT supports new-array execution, i.e., a PFFT plan can be planned and executed on different plans up to some restrictions, see Section **??** for details. Thanks to Yu Feng for the new-array execute patch.

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 6.7.
- 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 **??**.
- PFFT explicitly supports the parallel calculation of pruned FFTs. Details are given in Section **??**.

Finally, we complete this overview with a list of features that are (not yet) implemented in PFFT.

- Parallel one-dimensional FFT based on MPI. FFTW-MPI uses another parallelization strategy for one-dimensional FFTs, which is not implemented in PFFT. The reason is that we can not achive a scalability benefit due to higher dimensional data decomposition if the FFT has only one dimension. Therefore, one can also call FFTW directly in this case.
- There is no equivalent of FFTW *wisdom* in PFFT, i.e., you can not save a PFFT plan to disk and restore it for later use.
- PFFT does not have full OpenMP support. All serial FFT computations and global communications are implemented with FFTW, which offers OpenMP support, see Section **??**. However, most of the PFFT-only features, such as pruned FFT, ghost cell send and 3d decompostion of 3d FFTs are not yet parallelized with OpenMP.
- PFFT does not have full SIMD support. All serial FFT computations and global communications are implemented with FFTW, which offers SIMD support, see Section **??**. However, most of the PFFT-only features, such as pruned FFT, ghost

cell send and 3d decompostion of 3d FFTs are not yet parallelized with SIMD.
- PFFT does not overlap communication and computation. The code of PFFT is build in a very modularized structure. Most of these modules consist of FFTWs routines. Therefore, the global transposition does not support non blocking communication.
- Similar to FFTW, we do not provide any parallel IO routines. The user is responsible of load and store of parallel data.
- PFFT depends on FFTW to perform its serial transforms and does not support different vendor FFTs (such as Intel's MKL or IBM's ESSL). However, this is not assumed to be a big drawback, since FFTW seems to perform very well on most platforms.
- The global communication routines can not be called separately. However, it should be possible to implement a user interface to our global transposition routines.
- PFFT does not support GPU parallelization.

You are welcome to propose new PFFT features at https://github.com/mpip/pfft.git.

1.1 Alternative parallel FFT implementations

There have been several FFT implementations that aim to circumvent the scalability bottleneck for at least three dimensional FFTs by using two-dimensional decomposition approach. However, these implementations are often fitted to special problems and where not published as a stand alone software library. Remarkable exceptions are the parallel FFT software library by S. Plimpton [23, 22], the P3DFFT software library by D. Pekurovsky [17, 16] and the 2DECOMP&FFT software library by N. Li [14, 13].

1.2 Parallel nonequispaced FFT

If your are interested in a parallel implementation of nonequispaced fast Fourier transforms (NFFT) for distributed memory architectures, you should have a look at our PNFFT software library [19, 21] that is also available at https://github.com/mpip/pnfft.git.

2 Tutorial

The following chapter describes the usage of the PFFT library at the example of a simple test file in the first section, followed by the more advanced features of PFFT in the next sections.

2.1 A first parallel transform - Three-dimensional FFT with two-dimensional data decomposition

We explain the basic steps for computing a parallel FFT with the PFFT library at the example of the short test program given by Listing 2.1. This test computes a three-dimensional c2c-FFT on a two-dimensional process mesh. The source code `manual_c2c_3d.c` can be found in directory `tests/` of the library's source code tree.

After initializing MPI with `MPI_Init` and before calling any other PFFT routine initialize the parallel FFT computations via

```
void pfft_init(void);
```

MPI introduces the concept of communicators to store all the topological information of the physical process layout. PFFT requires to be called on a process mesh that corresponds to a periodic, Cartesian communicator. We assist the user in creating such a communicator with the following routine

```
int pfft_create_procmesh_2d(
    MPI_Comm comm, int np0, int np1,
    MPI_Comm *comm_cart_2d);
```

This routine uses the processes within the communicator `comm` to create a two-dimensional process grid of size `np0` x `np1` and stores it into the Cartesian communicator `comm_cart_2d`. Note that `comm_cart_2d` is allocated by the routine and must be freed with `MPI_Comm_free` after usage. The input parameter `comm` is a communicator, indicating which processes will participate in the transform. Choosing `comm` as `MPI_COMM_WORLD` implies that the FFT is computed on all available processes.

At the next step we need to know the data decomposition of the input and output array, that depends on the array sizes, the process grid and the chosen parallel algorithm. Therefore, we call

```
ptrdiff_t pfft_local_size_3d(
    ptrdiff_t *n, MPI_Comm comm_cart_2d, unsigned pfft_flags,
```

Listing 2.1: Minimal parallel c2c-FFT test program.

```c
#include <pfft.h>

int main(int argc, char **argv){
  int np[2];
  ptrdiff_t n[3];
  ptrdiff_t alloc_local;
  ptrdiff_t local_ni[3], local_i_start[3];
  ptrdiff_t local_no[3], local_o_start[3];
  pfft_complex *in, *out;
  pfft_plan plan=NULL;
  MPI_Comm comm_cart_2d;

  /* Set size of FFT and process mesh */
  n[0] = 2; n[1] = 2; n[2] = 4;
  np[0] = 2; np[1] = 2;

  /* Initialize MPI and PFFT */
  MPI_Init(&argc, &argv);
  pfft_init();

  /* Create two-dimensional process grid of size np[0] x np[1] */
  pfft_create_procmesh_2d(MPI_COMM_WORLD, np[0], np[1],
      &comm_cart_2d);

  /* Get parameters of data distribution */
  alloc_local = pfft_local_size_dft_3d(
      n, comm_cart_2d, PFFT_TRANSPOSED_NONE,
      local_ni, local_i_start, local_no, local_o_start);

  /* Allocate memory */
  in  = pfft_alloc_complex(alloc_local);
  out = pfft_alloc_complex(alloc_local);

  /* Plan parallel forward FFT */
  plan = pfft_plan_dft_3d(n, in, out, comm_cart_2d,
      PFFT_FORWARD, PFFT_TRANSPOSED_NONE);

  /* Initialize input with random numbers */
  pfft_init_input_complex_3d(n, local_ni, local_i_start,
      in);

  /* Execute parallel forward FFT */
  pfft_execute(plan);

  /* free mem and finalize MPI */
  pfft_destroy_plan(plan);
  MPI_Comm_free(&comm_cart_2d);
  pfft_free(in); pfft_free(out);
  MPI_Finalize();
  return 0;
}
```

```
    ptrdiff_t *local_ni, ptrdiff_t *local_i_start,
    ptrdiff_t *local_no, ptrdiff_t *local_o_start);
```

Hereby, `n`, `local_ni`, `local_i_start`, `local_no`, `local_o_start` are arrays of length 3 that must be allocated. The return value of this function equals the size of the local complex array that needs to be allocated by every process. In most cases, this coincides with the product of the local array sizes – but may be bigger, whenever the parallel algorithm needs some extra storage. The input value `n` gives the three-dimensional FFT size and the flag `pfft_flags` serves to adjust some details of the parallel execution. For the sake of simplicity, we restrict our self to the case `pfft_flags=PFFT_TRANSPOSED_NONE` for a while and explain the more sophisticated flags at a later point. The output arrays `local_ni` and `local_i_start` give the size and the offset of the local input array that result from the parallel block distribution of the global input array, i.e., every process owns the input data `in[k[0],k[1],k[2]]` with `local_i_start[t] <= k[t] < local_i_start[t] + local_ni[t]` for `t=0,1,2`. Analogously, the output parameters `local_o_start` and `local_no` contain the size and the offset of the local output array.

Afterward, the input and output arrays must be allocated. Hereby,

```
pfft_complex* pfft_alloc_complex(size_t size);
```

is a simple wrapper of `fftw_alloc_complex`, which in turn allocates the memory via `fftw_malloc` to ensure proper alignment for SIMD. Have a look at the FFTW user manual [9] for more details on SIMD memory alignment and `fftw_malloc`. Nevertheless, you can also use any other dynamic memory allocation.

The planning of a single three-dimensional parallel FFT of size `n[0]` x `n[1]` x `n[2]` is done by the function

```
pfft_plan pfft_plan_dft_3d(
    ptrdiff_t *n, pfft_complex *in, pfft_complex *out,
    MPI_Comm comm_cart_2d, int sign, unsigned pfft_flags);
```

We provide the address of the input and output array by the pointers `in` and `out`, respectively. An inplace transform is assumed if these pointers are equal. The integer `sign` gives the sign in the exponential of the FFT. Possible values are `PFFT_FORWARD` (-1) and `PFFT_BACKWARD` $(+1)$. Flags passed to the planner via `pfft_flags` must coincide with the flags that were passed to `pfft_local_size_3d`. Otherwise the data layout of the parallel execution may not match calculated local array sizes. As return value we get a PFFT plan, some structure that stores all the information needed to perform a parallel FFT.

Once the plan is generated, we are allowed to fill the input array `in`. Note, that per default the planning step `pfft_plan_dft_3d` will overwrite input array `in`. Therefore, you should not write any sensitive data into `in` until the plan was generated. For simplicity, our test program makes use of the library function

```
void pfft_init_input_complex_3d(
```

```
    ptrdiff_t *n, ptrdiff_t *local_ni, ptrdiff_t *local_i_start,
    pfft_complex *in);
```

to fill the input array with some numbers. Alternatively, one can fill the array with a function `func` of choice and the following loop that takes account of the parallel data layout

```
ptrdiff_t m=0;
for(ptrdiff_t k0=0; k0 < local_ni[0]; k0++)
  for(ptrdiff_t k1=0; k1 < local_ni[1]; k1++)
    for(ptrdiff_t k2=0; k2 < local_ni[2]; k2++)
      in[m++] = func(k0 + local_i_start[0],
                     k1 + local_i_start[1],
                     k2 + local_i_start[2]);
```

The parallel FFT is computed when we execute the generated plan via

```
void pfft_execute(const pfft_plan plan);
```

Now, the results can be read from `out` with an analogous three-dimensional loop. If we do not want to execute another parallel FFT of the same type, we free the allocated memory of the plan with

```
void pfft_destroy_plan(pfft_plan plan);
```

Additionally, we use

```
int MPI_Comm_free(MPI_Comm *comm);
```

to free the communicator allocated by `pfft_create_procmesh_2d` and

```
void pfft_free(void *ptr);
```

to free memory allocated by `pfft_alloc_complex`. Finally, we exit MPI via

```
int MPI_Finalize(void);
```

2.2 Porting FFTW-MPI based code to PFFT

finish FFTW2PFFT porting example

We illustrate the close connection between FFTW-MPI and PFFT at a three-dimensional MPI example analogous to the example given in the FFTW manual [2].

Exactly the same task can be performed with PFFT as given in Listing **??**.

```
#include <pfft.h>

int main(int argc, char **argv)
{
    const ptrdiff_t n[3] = {..., ..., ...};
```

Listing 2.2: Minimal parallel c2c-FFT test program.

```
#include <fftw3-mpi.h>

int main(int argc, char **argv)
{
    const ptrdiff_t n0 = 4, n1 = 4, n2 = 4;
    fftw_plan plan;
    fftw_complex *data;
    ptrdiff_t alloc_local, local_n0, local_0_start, i, j, k;

    MPI_Init(&argc, &argv);
    fftw_mpi_init();

    /* get local data size and allocate */
    alloc_local = fftw_mpi_local_size_3d(n0, n1, n2, MPI_COMM_WORLD,
                                         &local_n0, &local_0_start);
    data = fftw_alloc_complex(alloc_local);

    /* create plan for in-place forward DFT */
    plan = fftw_mpi_plan_dft_3d(n0, n1, n2, data, data,
       MPI_COMM_WORLD,
                                FFTW_FORWARD, FFTW_ESTIMATE);

    /* initialize data to some function my_function(x,y) */
    for (i = 0; i < local_n0; ++i)
      for (j = 0; j < n1; ++j)
        for (k = 0; k < n2; ++k)
          data[i*n1*n2 + j*n2 + k] = my_function(local_0_start + i,
              j, k);

    ptrdiff_t local_ni[3] = {local_n0, n1, n2}, local_i_start[3] =
       {local_0_start, 0, 0};
    pfft_apr_complex_3d(data, local_ni, local_i_start, "input:",
       MPI_COMM_WORLD);

    /* compute transforms, in-place, as many times as desired */
    fftw_execute(plan);

    ptrdiff_t local_no[3] = {local_n0, n1, n2}, local_o_start[3] =
       {local_0_start, 0, 0};
    pfft_apr_complex_3d(data, local_no, local_o_start, "output:",
       MPI_COMM_WORLD);

fftw_destroy_plan(plan);

    MPI_Finalize();
}
```

```
    pfft_plan plan;
    pfft_complex *data;
    ptrdiff_t alloc_local, local_ni[3], local_i_start[3],
        local_no[3], local_o_start[3], i, j, k;
    unsigned pfft_flags = 0;

    MPI_Init(&argc, &argv);
    pfft_init();

    /* get local data size and allocate */
    alloc_local = pfft_local_size_dft_3d(n, MPI_COMM_WORLD,
        pfft_flags,
                                         local_ni, local_i_start,
                                         local_no, local_o_start);
    data = pfft_alloc_complex(alloc_local);

    /* create plan for in-place forward DFT */
    plan = pfft_plan_dft_3d(n, data, data, MPI_COMM_WORLD,
                            PFFT_FORWARD, PFFT_ESTIMATE);

    /* initialize data to some function my_function(x,y,z) */
    for (i = 0; i < local_n[0]; ++i)
      for (j = 0; j < n[1]; ++j)
        for (k = 0; k < n[2]; ++k)
          data[i*n[1]*n[2] + j*n[2] + k] =
              my_function(local_i_start[0] + i, j, k);

    /* compute transforms, in-place, as many times as desired */
    pfft_execute(plan);

    pfft_destroy_plan(plan);

    MPI_Finalize();
}
```

- substitute `fftw3-mpi.h` by `pfft.h`
- substitute all prefixes `fftw_` and `fftw_mpi_` by `pfft_`
- substitute all prefixes `FFTW_` by `PFFT_`
- the integers `N`, `local_n0`, `local_0_start` become arrays of length 3
- `dft_` in `pfft_local_size_dft_3d`
- `pfft_local_size_dft_3d` has additional input `pfft_flags` and additional outputs `local_no`, `local_o_start`
- The loop that inits `data` becomes splitted along all three dimensions. We could also use

First, All prefixes `fftw_` are substituted by `pfft_`

Now, the changes in order to use a two-dimensional process mesh are marginal as can be seen in Listing **??**.

```
#include <pfft.h>

int main(int argc, char **argv)
{
    const ptrdiff_t n[3] = {..., ..., ...};
    const int np0 = ..., np1 = ...;
    pfft_plan plan;
    pfft_complex *data;
    ptrdiff_t alloc_local, local_ni[3], local_i_start[3],
        local_no[3], local_o_start[3], i, j, k;
    unsigned pfft_flags = 0;
    MPI_Comm comm_cart_2d;

    MPI_Init(&argc, &argv);
    pfft_init();

    /* create two-dimensional process grid of size np0 x np1 */
    pfft_create_procmesh_2d(MPI_COMM_WORLD, np0, np1,
        &comm_cart_2d);

    /* get local data size and allocate */
    alloc_local = pfft_local_size_dft_3d(n, comm_cart_2d, pfft_flags,
                                         local_ni, local_i_start,
                                         local_no, local_o_start);
    data = pfft_alloc_complex(alloc_local);

    /* create plan for in-place forward DFT */
    plan = pfft_plan_dft_3d(n, data, data, MPI_COMM_WORLD,
                            PFFT_FORWARD, PFFT_ESTIMATE);

    /* initialize data to some function my_function(x,y,z) */
    for (i = 0; i < local_n[0]; ++i)
      for (j = 0; j < local_n[1]; ++j)
        for (k = 0; k < local_n[2]; ++k)
          data[i*local_n[1]*local_n[2] + j*local_n[2] + k] =
              my_function(local_i_start[0] + i,
                          local_i_start[1] + j,
                          local_i_start[2] + k);

    /* compute transforms, in-place, as many times as desired */
    pfft_execute(plan);

    pfft_destroy_plan(plan);
```

```
    MPI_Finalize();
}
```

2.3 Errorcode for communicator creation

As we have seen the function

```
int pfft_create_procmesh_2d(
    MPI_Comm comm, int np0, int np1,
    MPI_Comm *comm_cart_2d);
```

creates a two-dimensional, periodic, Cartesian communicator. The **int** return value (not used in Listing 2.1) is the forwarded error code of `MPI_Cart_create`. It is equal to zero if the communicator was created successfully. The most common error is that the number of processes within the input communicator `comm` does not fit `np0 x np1`. In this case the Cartesian communicator is not generated and the return value is unequal to zero. Therefore, a typical sanity check might look like

```
/* Create two-dimensional process grid of size np[0] x np[1],
   if possible */
if( pfft_create_procmesh_2d(MPI_COMM_WORLD, np[0], np[1],
        &comm_cart_2d) )
{
  pfft_fprintf(MPI_COMM_WORLD, stderr,
      "Error: This test file only works with %d processes.\n",
      np[0]*np[1]);
  MPI_Finalize();
  return 1;
}
```

Hereby, we use the PFFT library function

```
void pfft_fprintf(
    MPI_Comm comm, FILE *stream, const char *format, ...);
```

to print the error message. This function is similar to the standard C function `fprintf` with the exception, that only the process with MPI rank 0 within the given communicator `comm` will produce some output; see Section 6.8.7 for details.

2.4 Inplace transforms

Similar to FFTW, PFFT is able to compute parallel FFTs completely in place, which means that beside some constant buffers, no second data array is necessary. Especially, the global data communication can be performed in place. As far as we know, there is

no other parallel FFT library beside FFTW and PFFT that supports this feature. This feature is enabled as soon as the pointer to the output array `out` is equal to the pointer to the input array `in`. E.g., in Listing 2.1 we would call

```
/* Plan parallel forward FFT */
plan = pfft_plan_dft_3d(n, in, in, comm_cart_2d,
    PFFT_FORWARD, PFFT_TRANSPOSED_NONE);
```

2.5 Higher dimensional data decomposition

The test program given in Listing 2.1 used a two-dimensional data decomposition of a three-dimensional data set. Moreover, PFFT support the computation of any d-dimensional FFT with r-dimensional data decomposition as long as $r \leq d - 1$. For example, one can use a one-dimensional data decomposition for any two- or higher-dimensional data set, while the data set must be at least four-dimensional to fit to a three-dimensional data decomposition. The case $r = d$ is not supported efficiently, since during the parallel computations there is always at least one dimension that remains local, i.e., one dimensions stays non-decomposed. The only exception from this rule is the case $d = r = 3$ that is supported by PFFT in a special way, see Section 2.6.2 for details.

The dimensionality of the data decomposition is given by the dimension of the Cartesian communicator that goes into the PFFT planing routines. Therefore, we present a generalization of communicator creation function

```
int pfft_create_procmesh(
    int rnk_np, MPI_Comm comm, const int *np,
    MPI_Comm *comm_cart);
```

Hereby, the array `np` of length `rnk_np` gives the size of the Cartesian communicator `cart_comm`.

2.6 Parallel data decomposition

In the following, we use the notation $\frac{n}{P}$ to symbolize that an array of length n is broken into disjoint blocks and distributed on P MPI processes. Hereby, the data is distributed in compliance to the FFTW-MPI data decompostion [5], i.e., the first `P/block` (rounded down) processes get a contiguous chunk of `block` elements, the next process gets the remaining `n - block * (n/block)` data elements, and all remaining processes get nothing. Thereby, the block size `block` defaults to `n/P` (rounded down) but can also be user defined.

2.6.1 Non-transposed and transposed data layout

In the following, we use the notation $\frac{n}{P}$ to symbolize that an array of length n is distributed on P MPI processes. The standard PFFT data decomposition of h interleaved d-dimensional arrays of equal size $n_0 \times n_1 \times \ldots \times n_{d-1}$ on a r-dimensional process mesh of size $P_0 \times \ldots \times P_{r-1}$ is given by the blocks

$$\frac{n_0}{P_0} \times \frac{n_1}{P_1} \times \ldots \times \frac{n_{r-1}}{P_{r-1}} \times n_r \times n_{r+1} \times \ldots \times n_{d-1} \times h.$$

A PFFT created with planning flag `PFFT_TRANSPOSED_NONE` requires the inputs to be decomposed in this standard way and produces outputs that are decomposed in the same way.

PFFT can save half of the global communication amount, if the data reordering to standard decomposition is omitted. The transposed data decomposition is given by

$$\frac{n_1}{P_0} \times \frac{n_2}{P_1} \times \ldots \times \frac{n_r}{P_{r-1}} \times n_0 \times n_{r+1} \times \ldots \times n_{d-1} \times h$$

A PFFT plan created with planning flag `PFFT_TRANSPOSED_OUT` produces outputs with transposed data decomposition. Analogously, a PFFT plan created with planning flag `PFFT_TRANSPOSED_IN` requires its inputs to be decomposed in the transposed way. Typically, one creates a forward plan with `PFFT_TRANSPOSED_OUT` and a backward plan with planning flag `PFFT_TRANSPOSED_IN`.

Note that the flags `PFFT_TRANSPOSED_OUT` and `PFFT_TRANSPOSED_IN` must be passed to the array distribution function (see Section 6.4) *as well as* to the planner (see Section 6.5).

2.6.2 Three-dimensional FFTs with three-dimensional data decomposition

Many applications work with three-dimensional block decompositions of three-dimensional arrays. PFFT supports decompositions of the kind

$$\frac{n_0}{P_0} \times \frac{n_1}{P_1} \times \frac{n_2}{P_2} \times h.$$

However, PFFT applies a parallel algorithms that needs at least one non-distributed transform dimension (we do not transform along h), Therefore, we split the number of processes along the last dimension into two factors $P_2 = Q_1 Q_2$, remap the data to the two-dimensional decomposition

$$\frac{n_0}{P_0 Q_0} \times \frac{n_1}{P_1 Q_1} \times n_2 \times h,$$

and compute the parallel FFT with this two-dimensional decomposition. Note that the 3d to 2d remap implies some very special restrictions on the block sizes for n_0 and

n_1, i.e., the blocks must be divisible by Q_0 and Q_1. More precisely, the default blocks of the 2d-decomposition are given by `n0/(P0*Q0)` and `n1/(P1*Q1)` (both divisions rounded down). This implies that the default blocks of the 3d-decomposition must be `n0/(P0*Q0)* Q0`, `n1/(P1*Q1)* Q1`, and `n2/(Q0*Q1)` (all divisions rounded down).

2.7 Planning effort

Pass one of the following flags

- `PFFT_ESTIMATE`,
- `PFFT_MEASURE`,
- `PFFT_PATIENT`, or,
- `PFFT_EXHAUSIVE`

to the PFFT planner in order to plan all internal FFTW plans with `FFTW_ESTIMATE`, `FFTW_MEASURE`, `FFTW_PATIENT`, or `FFTW_EXHAUSIVE`, respectively. The default value is `PFFT_MEASURE`.

PFFT uses FFTW plans for parallel array transposition and the serial transforms. In fact, every serial transform is a combination of strided lower-dimensional FFTs and a serial array transposition (necessary to prepare the global transposition) which can be done by a single FFTW plan. However, it turns out that FFTW sometimes performs better if the serial transposition and the strided FFTs are executed separately. Therefore, PFFT introduces the flag `PFFT_TUNE` that enables extensive run time tests in order to find the optimal sequence of serial strided FFT and serial transposition for every serial transform. These tests are disable on default which corresponds to the flag `PFFT_NO_TUNE`.

2.8 Preserving input data

The following flags

- `PFFT_PRESERVE_INPUT`,
- `PFFT_DESTROY_INPUT`, and,
- `PFFT_BUFFERED_INPLACE`

only take effect for out-of-place transforms. The first one behaves analogously to the FFTW flag `FFTW_PRESERVE_INPUT` and ensures that the input values are not overwritten. In fact, this flag implies that only the first serial transform is executed out-of-place and all successive steps are performed in-place on the output array. In compliance to FFTW, this is the default behaviour for out-of-place plans.

The second flag behaves analogously to the FFTW flag `FFTW_DESTROY_INPUT` and tells the planner that the input array can be used as scratch array. This may give some speedup for out-of-place plans, because all the intermediate transforms and transposition steps can be performed out-of-place.

Finally, the flag `PFFT_BUFFERED_INPLACE` can be used for out-of-place plans that store its inputs and outputs in the same array, i.e., array `out` is used for intermediate out-of-place transforms and transpositions but the PFFT inputs and outputs are stored in array `in`.

2.9 FFTs with shifted index sets

Describe shifted input and output

- `PFFT_SHIFTED_IN`
- `PFFT_SHIFTED_OUT`

2.10 Pruned FFT and Shifted Index Sets

Describe pruned FFT with shifted input and output

2.10.1 Pruned FFT

For pruned r2r- and c2c-FFT are defined as

$$g_l = \sum_{k=0}^{n_i-1} \hat{g}_k \mathrm{e}^{-2\pi \mathrm{i} k l/n}, \quad l = 0, \ldots, n_o - 1,$$

where $n_i \leq n$ and $n_o \leq n$.

2.10.2 Shifted Index Sets

For $N \in 2\mathbb{N}$ we define the FFT with shifted inputs

For $K, L, N \in 2\mathbb{N}$, $L < N$, $L < N$ we define

2.11 Precisions

PFFT handles multiple precisions exactly in the same way as FFTW. Therefore, we quote part [8] of the FFTW manual in the context of PFFT:

You can install single and long-double precision versions of PFFT, which replace double with float and long double, respectively; see **??**. To use these interfaces, you must

- Link to the single/long-double libraries; on Unix, `-lpfftf` or `-lpfftl` instead of (or in addition to) `-lpfft`. (You can link to the different-precision libraries simultaneously.)
- Include the same `<pfft.h>` header file.

- Replace all lowercase instances of '`pfft_`' with '`pfftf_`' or '`pfftl_`' for single or long-double precision, respectively. (**`pfft_complex`** becomes **`pfftf_complex`**, `pfft_execute` becomes `pfftf_execute`, etcetera.)
- Uppercase names, i.e. names beginning with '`PFFT_`', remain the same.
- Replace **`double`** with **`float`** or **`long double`** for subroutine parameters.

2.12 Ghost cell communication

explain ghost cell communication with a test file

2.13 Fortran interface

explain F03 interface with a test file

3 Installation and linking

The install of PFFT is based on the Autotools and follows the typical workflow

```
./configure
make
make install
```

3.1 Install of the latest official FFTW release

PFFT depends on Release 3.3.3 of the FFTW library [11]. For the sake of completeness, we show the command line based install procedure in the following. However, note that we provide install scripts on www.tu-chemnitz.de/~mpip/software.php that simplify the install a lot. We highly recommend to use these install scripts, since they additionally apply several performance patches and bugfixes that have been submitted to the FFTW developers but are not yet included in the official FFTW releases.

```
wget http://www.fftw.org/fftw-3.3.3.tar.gz
tar xzvf fftw-3.3.3.tar.gz
cd fftw-3.3.3
./configure --enable-mpi --prefix=$HOME/local/fftw3_mpi
make
make install
```

The MPI algorithms of FFTW must be build with a MPI C compiler. Add the statement `MPICC=$MPICCOMP` at the end of line 4 if the `configure` script fails to determine the right MPI C compiler `$MPICCOMP`. Similarly, the MPI Fortran compiler `$MPIFCOMP` is set by `MPIFC=$MPIFCOMP`.

3.2 Install of the PFFT library

In the simplest case, the hardware platform and the FFTW-3.3.3 library are recognized by the PFFT configure script automatically, so all we have to do is

```
wget http://www.tu-chemnitz.de/~mpip/software/pfft-1.0.7-alpha.tar.gz
tar xzvf pfft-1.0.7-alpha.tar.gz
cd pfft-1.0.7-alpha
./configure
make
```

```
make check
make install
```

Hereby, the optional call `make check` builds the test programs. If the FFTW-3.3.3 software library is already installed on your system but not found by the configure script, you can provide the FFTW installation directory `$FFTWDIR` to configure by

```
./configure --with-fftw3=$FFTWDIR
```

This call implies that the FFTW header files are located in `$FFTWDIR/include` and the FFTW library files are located in `$FFTWDIR/lib`. Otherwise, one should specify the FFTW include path `$FFTWINC` and the FFTW library path `$FFTWLIB` separately by

```
./configure --with-fftw3-includedir=$FFTWINC \
   --with-fftw3-libdir=$FFTWLIB
```

At the end, this is equivalent to

```
./configure CPPFLAGS=-I$FFTWINC LDFLAGS=-L$FFTWLIB
```

which is more common to experienced users of the Autotools. To install PFFT in a user specified directory `$PFFTINSTDIR` call configure with the option

```
./configure --prefix=$PFFTINSTDIR
```

However, this option is mandatory whenever you do not have root permissions on your machine, since the default install paths of `configure` are not accessible by standard users. The PFFT library must be built with a MPI compiler. In Section 3.1 we already described how to hand the right compilers to the `configure` script. Some more options are

- `--enable-float`: Produces a single-precision version of PFFT (float) instead of the default double-precision (double); see 2.11.
- `--enable-long-double`: Produces a long-double precision version of PFFT (long double) instead of the default double-precision (double); see 2.11.
- `--disable-fortran`: Disables inclusion of Fortran wrapper routines in the standard PFFT libraries.
- `--disable-tests`: Disables build of test programs.

For more details on the options of the `configure` script call

```
./configure --help
```

3.3 How to include PFFT in your program

All programs using PFFT should include its header file

```
#include <pfft.h>
```

This header includes the FFTW headers `fftw.h`, `fftw-mpi.h` automatically. Make sure that the compiler can find them by setting the include flags appropriately. You must also link to the PFFT, FFTW and FFTW-MPI libraries. On Unix, this means adding `-lpfft -lfftw3_mpi -lfftw3 -lm` at the end of the link command. For example, to build `pfft_test.c` use the following compiler invocation

```
mpicc pfft_test.c -I$PFFTINC -I$FFTWINC -L$PFFTLIB -L$FFTWLIB \
   -lpfft -lfftw3_mpi -lfftw3 -lm
```

Substitute `mpicc` by any other MPI C compiler if you like. `$PFFTINC`, `$FFTWINC`, `$PFFTLIB`, and `$FFTWLIB` denote the PFFT and FFTW include and library paths, respectively. If you use the install scripts mentioned in Sect. 3.2, these paths will be

```
PFFTINC = $HOME/local/pfft-1.0.7-alpha/include
FFTWINC = $HOME/local/fftw-3.3.3/include
PFFTLIB = $HOME/local/pfft-1.0.7-alpha/lib
FFTWLIB = $HOME/local/fftw-3.3.3/lib
```

4 Advanced Features

4.1 How to Deal with FFT Index Shifts in Parallel

Let $n \in 2\mathbb{N}$. A common problem is that the index of the FFT input and/or output array runs between $-n/2, \ldots, n/2-1$, but the FFT library requires them to run between $0, \ldots, n-1$. With serial program execution one can easily remap the input data $\hat{g}_k$ in a way that is suitable for the library, i.e.,

$$\hat{f}_k := \hat{g}_{(k-n/2 \bmod n)}, \quad k = 0, \ldots, n-1.$$

Similarly, one could remap the outputs of the library f_l, $l = 0, \cdots, n-1$ in the opposite direction in order to get the required outputs, i.e.,

$$g_l := f_{l \bmod n}, \quad l = -n/2, \ldots, n/2-1.$$

These shifts are also known as `fftshift` in Matlab.

However, with distributed memory these `fftshift` operations require more complex data movements and result in a global communication. For example, the first index of the array moves to the middle and, therefore, the corresponding data move to another MPI process. Fortunately, this communication can be avoided at the cost of little extra computation. At the end of the section we present two PFFT library functions that perform the necessary pre- and postprocessing for shifted input and output index sets.

4.1.1 Shift with half the FFT size

The special case of input shift $k_s = -n/2$ and/or output shift $l_s = -n/2$ is supported by PFFT. User can choose to shift the input (`PFFT_SHIFTED_IN`) and/or to shift the output (`PFFT_SHIFTED_OUT`).

this flag can be used for `local_size` and planning

Here, we are interested in the computation of

$$g_l = \sum_{k=-n_i/2}^{n_i/2-1} \hat{g}_k \mathrm{e}^{-2\pi \mathrm{i} k l/n}, \quad l = -n_o/2, \ldots, n_o/2-1$$

with $n, n_i, n_o \in 2\mathbb{N}$ and $n > n_i$, $n > n_o$.

With an index shift of $n/2$ both in k and l this equivalent to the computation of

$$\begin{aligned} g_{(l-n/2)} &= \sum_{k=n/2-n_i/2}^{n/2+n_i/2-1} \hat{g}_{(k-n/2)} \mathrm{e}^{-2\pi \mathrm{i}(k-n/2)(l-n/2)/n} \\ &= \mathrm{e}^{+\pi \mathrm{i} l} \sum_{k=n/2-n_i/2}^{n/2+n_i/2-1} \left(\hat{g}_{(k-n/2)} \mathrm{e}^{+\pi \mathrm{i}(k-n/2)} \right) \mathrm{e}^{-2\pi \mathrm{i} k l/n} \\ &= \mathrm{e}^{+\pi \mathrm{i}(l-n/2)} \underbrace{\sum_{k=n/2-n_i/2}^{n/2+n_i/2-1} \underbrace{\left(\hat{g}_{(k-n/2)} \mathrm{e}^{+\pi \mathrm{i} k} \right)}_{\hat{f}_k} \mathrm{e}^{-2\pi \mathrm{i} k l/n}}_{f_l} \end{aligned}$$

for $l = n/2 - n_o/2, \ldots, n/2 + n_o/2 - 1$. Therefore, we get the following algorithm

$$f_l = \sum_{k=0}^{n} \hat{g}_k \mathrm{e}^{-2\pi \mathrm{i} k l/n}, \quad l = -n_o/2, \ldots, n_o/2 - 1$$

The special case $k_s = -\frac{n_i}{2}, l_s = -\frac{n_o}{2}$ corresponds to the shifts the arrays (FFTSHIFT)

1: For $k = 0, \ldots, n-1$ set $\hat{f}_k = 0$.

2: For $k = -n_i/2, \ldots, n_i/2 - 1$ compute $\hat{f}_{(k+n/2)} = (-1)^{(k+n/2)} \hat{g}_k$.

3: For $l = 0, \ldots, n-1$ compute $f_l = \sum_{k=0}^{n} \hat{f}_k \mathrm{e}^{-2\pi \mathrm{i} k l/n}$ using PFFT.

4: For $l = -n_o/2, \ldots, n_o/2 - 1$ compute $g_l = (-1)^l f_{(l+n/2)}$.

Note, that this shift implies that the library deals with pruned FFTs in a special way, i.e., half of the zeros are added at the beginning of the inputs and the other half is added at the end.

4.1.2 Arbitrary shifts

More general shifts must be done by the user.

In a more general setting, we are interested in the computation of FFTs with shifted index sets, i.e., assume $k_s, l_s \in \mathbb{Z}$ and compute

$$g_l = \sum_{k=k_s}^{n_i+k_s-1} \hat{g}_k \mathrm{e}^{-2\pi \mathrm{i} k l/n}, \quad l = l_s, \ldots, n_o + l_s - 1\,.$$

Because of the periodicity of the FFT this can be easily performed by Alg. 4.1.2. How-

Algorithm 4.1 Shifted FFT with explicit data movement.

1: For $k = 0, \ldots, n_i - 1$ assign $\hat{f}_k = \hat{g}_{(k+k_s \bmod n_i)}$.

2: For $l = 0, \ldots, n_o - 1$ compute $f_l = \sum_{k=0}^{n_i} \hat{f}_k \mathrm{e}^{-2\pi \mathrm{i} kl/n}$ using PFFT.

3: For $l = 0, \ldots, n_o - 1$ assign $g_l = f_{(l-l_s \bmod n_o)}$.

ever, this involves explicit data movement since the sequence of data changes. For a our parallel data decomposition the change of data layout requires data communication. A simple index shift results in the computation of

$$g_{l+l_s} = \sum_{k=k_s}^{n_i+k_s-1} \hat{g}_k \mathrm{e}^{-2\pi \mathrm{i} k(l+l_s)/n} = \sum_{k=0}^{n_i-1} \hat{g}_{k+k_s} \mathrm{e}^{-2\pi \mathrm{i}(k+k_s)(l+l_s)/n}$$
$$= \mathrm{e}^{-2\pi \mathrm{i} k_s l/n} \sum_{k=0}^{n_i-1} \underbrace{\left(\hat{g}_{k+k_s} \mathrm{e}^{-2\pi \mathrm{i}(k+k_s) l_s/n} \right)}_{=:\hat{f}_k} \mathrm{e}^{-2\pi \mathrm{i} kl/n}$$

for all $l = 0, \ldots, n_o - 1$. The resulting Alg. 4.1.2 preserves the sequence of data at the price of some extra computation.

Algorithm 4.2 Shifted FFT without explicit data movement.

1: For $k = 0, \ldots, n_i - 1$ compute $\hat{f}_k = \hat{g}_{(k+k_s)} \mathrm{e}^{-2\pi \mathrm{i}(k+k_s) l_s/n}$.

2: For $l = 0, \ldots, n_o - 1$ compute $f_l = \sum_{k=0}^{n_i} \hat{f}_k \mathrm{e}^{-2\pi \mathrm{i} kl/n}$ using PFFT.

3: For $l = 0, \ldots, n_o - 1$ compute $g_{(l+l_s)} = f_l \mathrm{e}^{-2\pi \mathrm{i} k_s l/n}$.

The special case $k_s = -\frac{n_i}{2}, l_s = -\frac{n_o}{2}$ corresponds to the shifts the arrays (FFTSHIFT)

1: For $k = 0, \ldots, n_i - 1$ compute $\hat{f}_k = \hat{g}_{(k-n_i/2)} \mathrm{e}^{+\pi \mathrm{i}(k-n_i/2) n_o/n}$.

2: For $l = 0, \ldots, n_o - 1$ compute $f_l = \sum_{k=0}^{n_i} \hat{f}_k \mathrm{e}^{-2\pi \mathrm{i} kl/n}$ using PFFT.

3: For $l = 0, \ldots, n_o - 1$ compute $g_{(l-n_o/2)} = f_l \mathrm{e}^{+\pi \mathrm{i} n_i l/n}$.

4.2 Parallel pruned FFT

Within PFFT we define a pruned FFT as

$$g_l = \sum_{k=0}^{n_i-1} \hat{g}_k \mathrm{e}^{-2\pi \mathrm{i} kl/n}, \quad l = 0, \ldots, n_o - 1.$$

Formally, this is equivallent to the following regular size n FFT

$$f_l = \sum_{k=0}^{n-1} \hat{f}_k \mathrm{e}^{-2\pi \mathrm{i} kl/n}, \quad l = 0, \ldots, n,$$

with

$$\hat{g}_k := \begin{cases} \hat{f}_k, & : k = 0, \ldots, n_1 - 1, \\ 0 & : k = n_i, \ldots, n - 1, \end{cases}$$

and $f_l := g_l$, $k = 0, \ldots, n_o - 1$. I.e., we add $n - n_i$ zeros at the end of the input array and throw away $n - n_o$ entries at the end of the output array.

The definition of pruned FFT changes for `PFFT_SHIFTED_IN` and `PFFT_SHIFTED_OUT`.

5 Interface Layers of the PFFT Library

We give a quick overview of the PFFT interface layers in the order of increasing flexibility at the example of c2c-FFTs. For r2c-, c2r-, and r2r-FFT similar interface layer specifications apply. A full reference list of all PFFT functions is given in Chapter 6.

5.1 Basic Interface

The `_3d` interface is the simplest interface layer. It is suitable for the planning of three-dimensional FFTs.

```
ptrdiff_t pfft_local_size_dft_3d(
    const ptrdiff_t *n, MPI_Comm comm_cart, unsigned pfft_flags,
    ptrdiff_t *local_ni, ptrdiff_t *local_i_start,
    ptrdiff_t *local_no, ptrdiff_t *local_o_start);
void pfft_local_block_dft_3d(
    const ptrdiff_t *n, MPI_Comm comm_cart,
    int pid, unsigned pfft_flags,
    ptrdiff_t *local_ni, ptrdiff_t *local_i_start,
    ptrdiff_t *local_no, ptrdiff_t *local_o_start);
pfft_plan pfft_plan_dft_3d(
    const ptrdiff_t *n,
    pfft_complex *in, pfft_complex *out, MPI_Comm comm_cart,
    int sign, unsigned pfft_flags);
```

Hereby, `n`, `local_ni`, `local_i_start`, `local_no`, and `local_o_start` are **ptrdiff_t** arrays of length `3`.

The basic interface generalizes the `_3d` interface to FFTs of arbitrary dimension `rnk_n`.

```
ptrdiff_t pfft_local_size_dft(
    int rnk_n, const ptrdiff_t *n,
    MPI_Comm comm_cart, unsigned pfft_flags,
    ptrdiff_t *local_ni, ptrdiff_t *local_i_start,
    ptrdiff_t *local_no, ptrdiff_t *local_o_start);
void pfft_local_block_dft(
    int rnk_n, const ptrdiff_t *n,
    MPI_Comm comm_cart, int pid, unsigned pfft_flags,
    ptrdiff_t *local_ni, ptrdiff_t *local_i_start,
    ptrdiff_t *local_no, ptrdiff_t *local_o_start);
pfft_plan pfft_plan_dft(
```

```
    int rnk_n, const ptrdiff_t *n,
    pfft_complex *in, pfft_complex *out, MPI_Comm comm_cart,
    int sign, unsigned pfft_flags);
```

Therefore, `n`, `local_ni`, `local_i_start`, `local_no`, and `local_o_start` become arrays of length `rnk_n`.

5.2 Advanced Interface

The advanced interface introduces the arrays `ni` and `no` of length `rnk_n` that give the pruned FFT input and output size. Furthermore, the arrays `iblock` and `oblock` of length `rnk_pm` (`rnk_pm` being the dimension of the process mesh) serve to adjust the block size of the input and output block decomposition. The additional parameter `howmany` gives the number of transforms that will be computed simultaneously.

```
ptrdiff_t pfft_local_size_many_dft(
    int rnk_n, const ptrdiff_t *n,
    const ptrdiff_t *ni, const ptrdiff_t *no, ptrdiff_t howmany,
    const ptrdiff_t *iblock, const ptrdiff_t *oblock,
    MPI_Comm comm_cart, unsigned pfft_flags,
    ptrdiff_t *local_ni, ptrdiff_t *local_i_start,
    ptrdiff_t *local_no, ptrdiff_t *local_o_start);
void pfft_local_block_many_dft(
    int rnk_n, const ptrdiff_t *ni, const ptrdiff_t *no,
    const ptrdiff_t *iblock, const ptrdiff_t *oblock,
    MPI_Comm comm_cart, int pid, unsigned pfft_flags,
    ptrdiff_t *local_ni, ptrdiff_t *local_i_start,
    ptrdiff_t *local_no, ptrdiff_t *local_o_start);
pfft_plan pfft_plan_many_dft(
    int rnk_n, const ptrdiff_t *n,
    const ptrdiff_t *ni, const ptrdiff_t *no, ptrdiff_t howmany,
    const ptrdiff_t *iblock, const ptrdiff_t *oblock,
    pfft_complex *in, pfft_complex *out, MPI_Comm comm_cart,
    int sign, unsigned pfft_flags);
```

5.3 Skip Serial Transformations Along Single Dimensions

The `_skipped` interface extends the `_many` interface by adding the possibility to skip some of the serial FFTs.

```
pfft_plan pfft_plan_many_dft_skipped(
    int rnk_n, const ptrdiff_t *n,
    const ptrdiff_t *ni, const ptrdiff_t *no, ptrdiff_t howmany,
    const ptrdiff_t *iblock, const ptrdiff_t *oblock,
```

```
    const int *skip_trafos,
    pfft_complex *in, pfft_complex *out, MPI_Comm comm_cart,
    int sign, unsigned pfft_flags);
```

Hereby, `skip_trafos` is an **int** array of length `rnk_n`. However, at the moment only the first `rnk_pm+1` (`rnk_pm` being the mesh dimension of the communicator `comm_cart`) dimensions are recognized by PFFT. For `t=0,...,rnk_pm` set `skip_trafos[t]=0` if the `t`-th serial transformation should be computed, otherwise set `skip_trafos[t]=1`. Note that the local transpositions are always performed, since they are a prerequisite for the global communication to work. At the moment it is only possible to skip the whole serial transform along the last `rnk_n-rnk_pm-1` dimensions (those dimensions are not distributed in the initial data layout and PFFT computes the serial FFT along these dimensions at once). However, this behavior can be realized by a call of a `(rnk_pm+1)`-dimensional PFFT with

```
for(int t=rnk_pm+1; t<rnk_n; t++)
  howmany *= n[t];
```

and manual computation of the desired serial transforms along the last `rnk_n-rnk_pm-1` dimensions. Please inform the author if this feature is desired to be included in the PFFT interface – at the moment this feature has low priority since it can be done by the user manually.

6 PFFT Reference

6.1 Files and Data Types

You must include the PFFT header file by

```
#include <pfft.h>
```

in the preamble of each source file that calls PFFT. This header automatically includes `fftw.h` and `fftw3-mpi.h`. Therefore, PFFT can use the **`fftw_complex`** data type defined in `fftw.h`, see [3]. Note that **`fftw_complex`** is defined to be the C99 native complex whenever `<complex.h>` is included *before* `<fftw.h>`, `<fftw-mpi.h>` and `<pfft.h>`. Otherwise it is defined as

```
typedef double fftw_complex[2];
```

For the sake of a clean namespace we define the wrapper data type **`pfft_complex`** as

```
typedef fftw_complex pfft_complex;
```

that can be used equivallently to **`fftw_complex`**. Futhermore, we define the wrapper functions

```
void *pfft_malloc(size_t n);
double *pfft_alloc_real(size_t n);
pfft_complex *pfft_alloc_complex(size_t n);
void pfft_free(void *p);
```

as substitues for their corresponding FFTW equivalents, see [4]. Note that memory allocated by one of these functions must be freed with `pfft_free` (or its equivalent `fftw_free`). Because of the performance reasons given in [9] we recommend to use one of the `pfft_` (or its equivalent `fftw_`) allocation functions for all arrays containing FFT inputs and outputs. However, PFFT will also work (possibly slower) with any other memory allocation method.

Different precisions are handled as in FFTW: That is `pfft_` functions and datatypes become `pfftf_` (single precision) or `pfftl_` (long double precision) prefixed. Quadruple precision is not yet supported. The main problem is that we do not know about a suitable MPI datatype to represent `__float128`.

6.2 MPI Initialization

Initialization and cleanup of PFFT in done in the same way as for FFTW-MPI, see [6]. In order to keep a clean name space, PFFT offers the wrapper functions

```
void pfft_init(void);
void pfft_cleanup(void);
```

that can be used as substitutes for `fftw_mpi_init` and `fftw_mpi_cleanup`, respectively.

6.3 Using PFFT Plans

PFFT follows exactly the same workflow as FFTW-MPI. A plan created by one of the functions given in Section 6.5 is executed with

```
void pfft_execute(const pfft_plan plan);
```

and freed with

```
void pfft_destroy_plan(const pfft_plan plan);
```

Note, that you can *not* apply `fftw_mpi_execute` or `fftw_destroy` on PFFT plans.

The new array execute functions are given by

```
void pfft_execute_dft(const pfft_plan plan, pfft_complex *in,
    pfft_complex *out);
void pfft_execute_dft_r2c(const pfft_plan plan, double *in,
    pfft_complex *out);
void pfft_execute_dft_c2r(const pfft_plan plan, pfft_complex *in,
    double *out);
void pfft_execute_r2r(const pfft_plan plan, double *in, double *out);
```

The arrays given by `in` and `out` must have the correct size and the same alignement as the array that were used to create the plan, just as it is the case for FFTW, see **??**.

6.4 Data Distribution Functions

6.4.1 Complex-to-Complex FFT

```
ptrdiff_t pfft_local_size_dft_3d(
    const ptrdiff_t *n, MPI_Comm comm_cart, unsigned pfft_flags,
    ptrdiff_t *local_ni, ptrdiff_t *local_i_start,
    ptrdiff_t *local_no, ptrdiff_t *local_o_start);
ptrdiff_t pfft_local_size_dft(
    int rnk_n, const ptrdiff_t *n,
    MPI_Comm comm_cart, unsigned pfft_flags,
```

```
    ptrdiff_t *local_ni, ptrdiff_t *local_i_start,
    ptrdiff_t *local_no, ptrdiff_t *local_o_start);
ptrdiff_t pfft_local_size_many_dft(
    int rnk_n, const ptrdiff_t *n, const ptrdiff_t *ni, const
       ptrdiff_t *no,
    ptrdiff_t howmany, const ptrdiff_t *iblock, const ptrdiff_t
       *oblock,
    MPI_Comm comm_cart, unsigned pfft_flags,
    ptrdiff_t *local_ni, ptrdiff_t *local_i_start,
    ptrdiff_t *local_no, ptrdiff_t *local_o_start);
```

Compute the data distribution of a parallel, complex input/output discrete Fourier transform (DFT) in two or more dimensions, returning the number of *complex* numbers that must be allocated to hold the parallel transform.

Arguments:

- `rnk_n` is the rank of the transform (typically the size of the arrays `n`, `ni`, `no`) that can be any integer ≥ 2. The `_3d` planner corresponds to a `rnk_n` of 3.
- The array `n` of size `rnk_n` specifies the transform dimensions. They can be any positive integer.
- The array `ni` of size `rnk_n` specifies the input array dimensions. They can be any positive integer with `ni[t] <= n[t]` for all dimensions `t=0,...,rnk_n-1`. For `ni[t]<n[t]` the inputs will be padded with zeros up to size `n[t]` along the `t`-th dimension before the transform, see Section **??**.
- The array `no` of size `rnk_n` specifies the output array dimensions. They can be any positive integer with `no[t] <= n[t]` for all dimensions `t=0,...,rnk_n-1`. For `no[t]<n[t]` the outputs will be pruned to size `no[t]` along the `t`-th dimension after the transform, see Section **??**.
- `howmany` is the number of transforms to compute. The resulting plan computes howmany transforms, where the input of the k-th transform is at location in+k (in C pointer arithmetic) with stride `howmany`, and its output is at location out+k with stride `howmany`. The basic `pfft_plan_dft` interface corresponds to howmany=1.
- `comm_cart` is a Cartesian communicator of dimension `rnk_pm` that specifies the parallel data decomposition, see Section **??**. Most of the time, PFFT requires `rnk_pm < rnk_n`. The only exception is the case `rnk_pm == rnk_n == 3`, see Section 2.6.2. If an ordinary (i.e. non-Cartesian) communicator is passed, PFFT internally converts it into a one-dimensional Cartesian communicator while retaining the MPI ranks (this results in the FFTW-MPI data decomposition).
- The arrays `iblock` and `oblock` of size `rnk_pm+1` specify the block sizes for the first `rnk_pm+1` dimensions of the input and output data, respectively. These must be the same block sizes as were passed to the corresponding `local_size` function. You can pass `PFFT_DEFAULT_BLOCKS` to use PFFT's default block sizes. Furthermore, you can use `PFFT_DEFAULT_BLOCK` to set the default block size in separate

dimensions, e.g., `iblock[t]=PFFT_DEFAULT_BLOCK`.

- `pfft_flags` is a bitwise OR ('|') of zero or more planner flags, as defined in Section **??**.
- The array `local_ni` of size `rnk_n` returns the size of the local input array block in every dimension (counted in units of complex numbers).
- The array `local_i_start` of size `rnk_n` returns the offset of the local input array block in every dimension (counted in units of complex numbers).
- The array `local_no` of size `rnk_n` returns the size of the local output array block in every dimension (counted in units of complex numbers).
- The array `local_o_start` of size `rnk_n` returns the offset of the local output array block in every dimension (counted in units of complex numbers).

In addition, the following `local_block` functions compute the local data distribution of the process with MPI rank `pid`. The `local_size` interface can be understood as a call of `local_block` where `pid` is given by `MPI_Comm_rank(comm_cart, &pid)`, i.e., each MPI process computes its own data block. However, `local_block` functions have a **`void`** return type, i.e., they omit the computation of the local array size that is necessary to hold the parallel transform. This makes `local_block` functions substantially faster in exectuion.

```
void pfft_local_block_dft_3d(
    const ptrdiff_t *n, MPI_Comm comm_cart, int pid, unsigned
       pfft_flags,
    ptrdiff_t *local_ni, ptrdiff_t *local_i_start,
    ptrdiff_t *local_no, ptrdiff_t *local_o_start);
void pfft_local_block_dft(
    int rnk_n, const ptrdiff_t *n,
    MPI_Comm comm_cart, int pid, unsigned pfft_flags,
    ptrdiff_t *local_ni, ptrdiff_t *local_i_start,
    ptrdiff_t *local_no, ptrdiff_t *local_o_start);
void pfft_local_block_many_dft(
    int rnk_n, const ptrdiff_t *ni, const ptrdiff_t *no,
    const ptrdiff_t *iblock, const ptrdiff_t *oblock,
    MPI_Comm comm_cart, int pid, unsigned pfft_flags,
    ptrdiff_t *local_ni, ptrdiff_t *local_i_start,
    ptrdiff_t *local_no, ptrdiff_t *local_o_start);
```

6.4.2 Real-to-Complex FFT

```
ptrdiff_t pfft_local_size_dft_r2c_3d(
    const ptrdiff_t *n, MPI_Comm comm_cart, unsigned pfft_flags,
    ptrdiff_t *local_ni, ptrdiff_t *local_i_start,
    ptrdiff_t *local_no, ptrdiff_t *local_o_start);
ptrdiff_t pfft_local_size_dft_r2c(
```

```
    int rnk_n, const ptrdiff_t *n,
    MPI_Comm comm_cart, unsigned pfft_flags,
    ptrdiff_t *local_ni, ptrdiff_t *local_i_start,
    ptrdiff_t *local_no, ptrdiff_t *local_o_start);
ptrdiff_t pfft_local_size_many_dft_r2c(
    int rnk_n, const ptrdiff_t *n, const ptrdiff_t *ni, const
       ptrdiff_t *no,
    ptrdiff_t howmany, const ptrdiff_t *iblock, const ptrdiff_t
       *oblock,
    MPI_Comm comm_cart, unsigned pfft_flags,
    ptrdiff_t *local_ni, ptrdiff_t *local_i_start,
    ptrdiff_t *local_no, ptrdiff_t *local_o_start);
```

Compute the data distribution of a parallel, real-input/complex-output discrete Fourier transform (DFT) in two or more dimensions, returning the number of *complex* numbers that must be allocated to hold the parallel transform.

Arguments are the same as for c2c transforms (see Section 6.4.1) with the following exceptions:

- The logical input array size `ni` will differ from the physical array size of the real inputs if the flag `PFFT_PADDED_R2C` is included in `pfft_flags`. This results from the padding at the end of the last dimension that is necessary to align the real valued inputs and complex valued outputs for inplace transforms, see [7]. In contrast to FFTW-MPI, PFFT does not pad the r2c inputs per default.
- `local_ni` is counted in units of real numbers. It will include padding
- `local_i_start` is counted in units of real numbers.

The corresponding `local_block` functions compute the local data distribution of the process with MPI rank `pid`.

```
void pfft_local_block_dft_r2c_3d(
    const ptrdiff_t *n, MPI_Comm comm_cart, int pid, unsigned
       pfft_flags,
    ptrdiff_t *local_ni, ptrdiff_t *local_i_start,
    ptrdiff_t *local_no, ptrdiff_t *local_o_start);
void pfft_local_block_dft_r2c(
    int rnk_n, const ptrdiff_t *n,
    MPI_Comm comm_cart, int pid, unsigned pfft_flags,
    ptrdiff_t *local_ni, ptrdiff_t *local_i_start,
    ptrdiff_t *local_no, ptrdiff_t *local_o_start);
void pfft_local_block_many_dft_r2c(
    int rnk_n, const ptrdiff_t *ni, const ptrdiff_t *no,
    const ptrdiff_t *iblock, const ptrdiff_t *oblock,
    MPI_Comm comm_cart, int pid, unsigned pfft_flags,
    ptrdiff_t *local_ni, ptrdiff_t *local_i_start,
    ptrdiff_t *local_no, ptrdiff_t *local_o_start);
```

6.4.3 Complex-to-Real FFT

```
ptrdiff_t pfft_local_size_dft_c2r_3d(
    const ptrdiff_t *n, MPI_Comm comm_cart, unsigned pfft_flags,
    ptrdiff_t *local_ni, ptrdiff_t *local_i_start,
    ptrdiff_t *local_no, ptrdiff_t *local_o_start);
ptrdiff_t pfft_local_size_dft_c2r(
    int rnk_n, const ptrdiff_t *n,
    MPI_Comm comm_cart, unsigned pfft_flags,
    ptrdiff_t *local_ni, ptrdiff_t *local_i_start,
    ptrdiff_t *local_no, ptrdiff_t *local_o_start);
ptrdiff_t pfft_local_size_many_dft_c2r(
    int rnk_n, const ptrdiff_t *n, const ptrdiff_t *ni, const
       ptrdiff_t *no,
    ptrdiff_t howmany, const ptrdiff_t *iblock, const ptrdiff_t
       *oblock,
    MPI_Comm comm_cart, unsigned pfft_flags,
    ptrdiff_t *local_ni, ptrdiff_t *local_i_start,
    ptrdiff_t *local_no, ptrdiff_t *local_o_start);
```

Compute the data distribution of a parallel, complex-input/real-output discrete Fourier transform (DFT) in two or more dimensions, returning the number of *complex* numbers that must be allocated to hold the parallel transform.

Arguments are the same as for c2c transforms (see Section 6.4.1) with the following exceptions:

- The logical output array size `no` will differ from the physical array size of the real outputs if the flag `PFFT_PADDED_C2R` is included in `pfft_flags`. This results from the padding at the end of the last dimension that is necessary to align the real valued outputs and complex valued inputs for inplace transforms, see [7]. In contrast to FFTW-MPI, PFFT does not pad the c2r outputs per default.
- `local_no` is counted in units of real numbers.
- `local_o_start` is counted in units of real numbers.

The corresponding `local_block` functions compute the local data distribution of the process with MPI rank `pid`.

```
void pfft_local_block_dft_c2r_3d(
    const ptrdiff_t *n, MPI_Comm comm_cart, int pid, unsigned
       pfft_flags,
    ptrdiff_t *local_ni, ptrdiff_t *local_i_start,
    ptrdiff_t *local_no, ptrdiff_t *local_o_start);
void pfft_local_block_dft_c2r(
    int rnk_n, const ptrdiff_t *n,
    MPI_Comm comm_cart, int pid, unsigned pfft_flags,
    ptrdiff_t *local_ni, ptrdiff_t *local_i_start,
    ptrdiff_t *local_no, ptrdiff_t *local_o_start);
```

```
void pfft_local_block_many_dft_c2r(
    int rnk_n, const ptrdiff_t *ni, const ptrdiff_t *no,
    const ptrdiff_t *iblock, const ptrdiff_t *oblock,
    MPI_Comm comm_cart, int pid, unsigned pfft_flags,
    ptrdiff_t *local_ni, ptrdiff_t *local_i_start,
    ptrdiff_t *local_no, ptrdiff_t *local_o_start);
```

6.4.4 Real-to-Real FFT

```
ptrdiff_t pfft_local_size_r2r_3d(
    const ptrdiff_t *n, MPI_Comm comm_cart, unsigned pfft_flags,
    ptrdiff_t *local_ni, ptrdiff_t *local_i_start,
    ptrdiff_t *local_no, ptrdiff_t *local_o_start);
ptrdiff_t pfft_local_size_r2r(
    int rnk_n, const ptrdiff_t *n,
    MPI_Comm comm_cart, unsigned pfft_flags,
    ptrdiff_t *local_ni, ptrdiff_t *local_i_start,
    ptrdiff_t *local_no, ptrdiff_t *local_o_start);
ptrdiff_t pfft_local_size_many_r2r(
    int rnk_n, const ptrdiff_t *n, const ptrdiff_t *ni, const
       ptrdiff_t *no,
    ptrdiff_t howmany, const ptrdiff_t *iblock, const ptrdiff_t
       *oblock,
    MPI_Comm comm_cart, unsigned pfft_flags,
    ptrdiff_t *local_ni, ptrdiff_t *local_i_start,
    ptrdiff_t *local_no, ptrdiff_t *local_o_start);
```

Compute the data distribution of a parallel, complex input/output discrete Fourier transform (DFT) in two or more dimensions, returning the number of *real* numbers that must be allocated to hold the parallel transform.

Arguments are the same as for c2c transforms (see Section 6.4.1) with the following exceptions:

- `local_ni` is counted in units of real numbers.
- `local_i_start` is counted in units of real numbers.
- `local_no` is counted in units of real numbers.
- `local_o_start` is counted in units of real numbers.

The corresponding `local_block` functions compute the local data distribution of the process with MPI rank `pid`.

```
void pfft_local_block_r2r_3d(
    const ptrdiff_t *n, MPI_Comm comm_cart, int pid, unsigned
       pfft_flags,
    ptrdiff_t *local_ni, ptrdiff_t *local_i_start,
    ptrdiff_t *local_no, ptrdiff_t *local_o_start);
void pfft_local_block_r2r(
```

```
    int rnk_n, const ptrdiff_t *n,
    MPI_Comm comm_cart, int pid, unsigned pfft_flags,
    ptrdiff_t *local_ni, ptrdiff_t *local_i_start,
    ptrdiff_t *local_no, ptrdiff_t *local_o_start);
void pfft_local_block_many_r2r(
    int rnk_n, const ptrdiff_t *ni, const ptrdiff_t *no,
    const ptrdiff_t *iblock, const ptrdiff_t *oblock,
    MPI_Comm comm_cart, int pid, unsigned pfft_flags,
    ptrdiff_t *local_ni, ptrdiff_t *local_i_start,
    ptrdiff_t *local_no, ptrdiff_t *local_o_start);
```

6.5 Plan Creation

6.5.1 Complex-to-Complex FFT

```
pfft_plan pfft_plan_dft_3d(
    const ptrdiff_t *n, pfft_complex *in, pfft_complex *out,
       MPI_Comm comm_cart,
    int sign, unsigned pfft_flags);
pfft_plan pfft_plan_dft(
    int rnk_n, const ptrdiff_t *n, pfft_complex *in, pfft_complex
       *out, MPI_Comm comm_cart,
    int sign, unsigned pfft_flags);
pfft_plan pfft_plan_many_dft(
    int rnk_n, const ptrdiff_t *n, const ptrdiff_t *ni, const
       ptrdiff_t *no,
    ptrdiff_t howmany, const ptrdiff_t *iblock, const ptrdiff_t
       *oblock,
    pfft_complex *in, pfft_complex *out, MPI_Comm comm_cart,
    int sign, unsigned pfft_flags);
pfft_plan pfft_plan_many_dft_skipped(
    int rnk_n, const ptrdiff_t *n, const ptrdiff_t *ni, const
       ptrdiff_t *no,
    ptrdiff_t howmany, const ptrdiff_t *iblock, const ptrdiff_t
       *oblock,
    const int *skip_trafos, pfft_complex *in, pfft_complex *out,
       MPI_Comm comm_cart,
    int sign, unsigned pfft_flags);
```

Plan a parallel, complex input/output discrete Fourier transform (DFT) in two or more dimensions, returning an **pfft_plan**. The planner returns NULL if the plan cannot be created.

Arguments:

- `rnk_n`, `n`, `ni`, `no`, `howmany`, `iblock`, `oblock`, `comm_cart` must be the same as passed to the corresponding `pfft_local_size_dft` function, see Section 6.4.1.

- The array `skip_trafos` of size `rnk_pm+1` specifies the serial transforms that will be omitted. For `t=0,...,rnk_pm` set `skip_trafos[t]=1` if the `t`-th serial transformation should be computed, otherwise set `skip_trafos[t]=0`, see Section 5.3 for more details.
- `in` and `out` point to the complex valued input and output arrays of the transform, which may be the same (yielding an in-place transform). These arrays are overwritten during planning, unless `PFFT_ESTIMATE | PFFT_NO_TUNE` is used in the flags. (The arrays need not be initialized, but they must be allocated.)
- `sign` is the sign of the exponent in the formula that defines the Fourier transform. It can be -1 (= `PFFT_FORWARD`) or +1 (= `PFFT_BACKWARD`).
- `pfft_flags` is a bitwise OR (`'|'`) of zero or more planner flags, as defined in Section **??**.

PFFT computes an unnormalized transform: computing a forward followed by a backward transform (or vice versa) will result in the original data multiplied by the size of the transform (the product of the dimensions `n[t]`).

6.5.2 Real-to-Complex FFT

```
pfft_plan pfft_plan_dft_r2c_3d(
    const ptrdiff_t *n, double *in, pfft_complex *out, MPI_Comm
       comm_cart,
    int sign, unsigned pfft_flags);
pfft_plan pfft_plan_dft_r2c(
    int rnk_n, const ptrdiff_t *n, double *in, pfft_complex *out,
       MPI_Comm comm_cart,
    int sign, unsigned pfft_flags);
pfft_plan pfft_plan_many_dft_r2c(
    int rnk_n, const ptrdiff_t *n, const ptrdiff_t *ni, const
       ptrdiff_t *no,
    ptrdiff_t howmany, const ptrdiff_t *iblock, const ptrdiff_t
       *oblock,
    double *in, pfft_complex *out, MPI_Comm comm_cart,
    int sign, unsigned pfft_flags);
pfft_plan pfft_plan_many_dft_r2c_skipped(
    int rnk_n, const ptrdiff_t *n, const ptrdiff_t *ni, const
       ptrdiff_t *no,
    ptrdiff_t howmany, const ptrdiff_t *iblock, const ptrdiff_t
       *oblock,
    const int *skip_trafos, double *in, pfft_complex *out, MPI_Comm
       comm_cart,
    int sign, unsigned pfft_flags);
```

Plan a parallel, real-input/complex-output discrete Fourier transform (DFT) in two or more dimensions, returning an **`pfft_plan`**. The planner returns NULL if the plan cannot

be created.

Arguments:

- `rnk_n`, `n`, `ni`, `no`, `howmany`, `iblock`, `oblock`, `comm_cart` must be the same as passed to the corresponding `pfft_local_size_dft_r2c` function, see Section 6.4.2.
- `in` and `out` point to the real valued input and complex valued output arrays of the transform, which may be the same (yielding an in-place transform). These arrays are overwritten during planning, unless `PFFT_ESTIMATE | PFFT_NO_TUNE` is used in the flags. (The arrays need not be initialized, but they must be allocated.)
- `sign` is the sign of the exponent in the formula that defines the Fourier transform. It can be -1 (= `PFFT_FORWARD`) or +1 (= `PFFT_BACKWARD`). Note that this parameter is not part of the FFTW-MPI interface, where r2c transforms are defined to be forward transforms. However, the backward transform can be easily realized by an additional conjugation of the complex outputs as done by PFFT.

6.5.3 Complex-to-Real FFT

```
pfft_plan pfft_plan_dft_c2r_3d(
    const ptrdiff_t *n, pfft_complex *in, double *out, MPI_Comm
       comm_cart,
    int sign, unsigned pfft_flags);
pfft_plan pfft_plan_dft_c2r(
    int rnk_n, const ptrdiff_t *n, pfft_complex *in, double *out,
       MPI_Comm comm_cart,
    int sign, unsigned pfft_flags);
pfft_plan pfft_plan_many_dft_c2r(
    int rnk_n, const ptrdiff_t *n, const ptrdiff_t *ni, const
       ptrdiff_t *no,
    ptrdiff_t howmany, const ptrdiff_t *iblock, const ptrdiff_t
       *oblock,
    pfft_complex *in, double *out, MPI_Comm comm_cart,
    int sign, unsigned pfft_flags);
pfft_plan pfft_plan_many_dft_c2r_skipped(
    int rnk_n, const ptrdiff_t *n, const ptrdiff_t *ni, const
       ptrdiff_t *no,
    ptrdiff_t howmany, const ptrdiff_t *iblock, const ptrdiff_t
       *oblock,
    const int *skip_trafos, pfft_complex *in, double *out, MPI_Comm
       comm_cart,
    int sign, unsigned pfft_flags);
```

Plan a parallel, complex-input/real-output discrete Fourier transform (DFT) in two or more dimensions, returning an **`pfft_plan`**. The planner returns NULL if the plan cannot be created.

Arguments:

- rnk_n, n, ni, no, howmany, iblock, oblock, comm_cart must be the same as passed to the corresponding pfft_local_size_dft_c2r function, see Section 6.4.3.
- in and out point to the complex valued input and real valued output arrays of the transform, which may be the same (yielding an in-place transform). These arrays are overwritten during planning, unless PFFT_ESTIMATE | PFFT_NO_TUNE is used in the flags. (The arrays need not be initialized, but they must be allocated.)
- sign is the sign of the exponent in the formula that defines the Fourier transform. It can be -1 (= PFFT_FORWARD) or +1 (= PFFT_BACKWARD). Note that this parameter is not part of the FFTW-MPI interface, where c2r transforms are defined to be backward transforms. However, the forward transform can be easily realized by an additional conjugation of the complex inputs as done by PFFT.

6.5.4 Real-to-Real FFT

```
pfft_plan pfft_plan_r2r_3d(
    const ptrdiff_t *n, double *in, double *out, MPI_Comm comm_cart,
    const pfft_r2r_kind *kinds, unsigned pfft_flags);
pfft_plan pfft_plan_r2r(
    int rnk_n, const ptrdiff_t *n, double *in, double *out, MPI_Comm
       comm_cart,
    const pfft_r2r_kind *kinds, unsigned pfft_flags);
pfft_plan pfft_plan_many_r2r(
    int rnk_n, const ptrdiff_t *n, const ptrdiff_t *ni, const
       ptrdiff_t *no,
    ptrdiff_t howmany, const ptrdiff_t *iblock, const ptrdiff_t
       *oblock,
    double *in, double *out, MPI_Comm comm_cart,
    const pfft_r2r_kind *kinds, unsigned pfft_flags);
pfft_plan pfft_plan_many_r2r_skipped(
    int rnk_n, const ptrdiff_t *n, const ptrdiff_t *ni, const
       ptrdiff_t *no,
    ptrdiff_t howmany, const ptrdiff_t *iblock, const ptrdiff_t
       *oblock,
    const int *skip_trafos, double *in, double *out, MPI_Comm
       comm_cart,
    const pfft_r2r_kind *kinds, unsigned pfft_flags);
```

Plan a parallel, real input/output (r2r) transform in two or more dimensions, returning an **pfft_plan**. The planner returns NULL if the plan cannot be created.

Arguments:

- rnk_n, n, ni, no, howmany, iblock, oblock, comm_cart must be the same as passed to the corresponding pfft_local_size_r2r function, see Section 6.4.4.

- `in` and `out` point to the real valued input and output arrays of the transform, which may be the same (yielding an in-place transform). These arrays are overwritten during planning, unless `PFFT_ESTIMATE | PFFT_NO_TUNE` is used in the flags. (The arrays need not be initialized, but they must be allocated.)
- The array `kinds` of length `rnk_n` specifies the kind of r2r transform that is computed in the corresponding dimensions. Just like FFTW-MPI we compute the separable product formed by taking each transform kind along the corresponding dimension, one dimension after another.

6.6 FFT Execution Timer

PFFT offers an easy way to perform run time measurements and print/write the results.

6.6.1 Basis Run Time Measurements

PFFT-plans automatically accumulate the local run times of every call to `pfft_execute`. For most applications it is sufficient to print run time of a plan `ths` averaged over all runs with

```
void pfft_print_average_timer(
    const pfft_plan ths, MPI_Comm comm);
```

Note, that for each timer the maximum time over all processes is reduced to rank `0` of communicator `comm`, i.e., a call to `MPI_Reduce` is performed and the output is only printed on this process. The following function works in the same way but prints more verbose output

```
void pfft_print_average_timer_adv(
    const pfft_plan ths, MPI_Comm comm);
```

To write the averaged run time of plan `ths` into a file called `name` use

```
void pfft_write_average_timer(
    const pfft_plan ths, const char *name, MPI_Comm comm);
void pfft_write_average_timer_adv(
    const pfft_plan ths, const char *name, MPI_Comm comm);
```

Again, the output is only written on rank `0` of communicator `comm`.

Discard all the recorded run times with

```
void pfft_reset_timer(
    pfft_plan ths);
```

This function is called per default at the end of every PFFT plan creation function.

6.6.2 Advanced Timer Manipulation

In order to access the run times directly a new typedef **pfft_timer** is introduced. The following function returns a copy of the timer corresponding to PFFT plan `ths`

```
pfft_timer pfft_get_timer(
    const pfft_plan ths);
```

Note that the memory of the returned **pfft_timer** must be released with

```
void pfft_destroy_timer(
    pfft_timer ths);
```

as soon as the timer is not needed anymore.

In the following we introduce some routines to perform basic operations on timers. For all functions with a **pfft_timer** return value you must use `pfft_destroy_timer` in order to release the allocated memory of the timer. Create a copy of a PFFT-timer `orig` with

```
pfft_timer pfft_copy_timer(
    const pfft_timer orig);
```

Compute the average, local time over all runs of a timer `ths` with

```
void pfft_average_timer(
    pfft_timer ths);
```

Create a new timer that contains the sum of two timers `sum1` and `sum2` with

```
pfft_timer pfft_add_timers(
    const pfft_timer sum1, const pfft_timer sum2);
```

Create a timer that contains the maximum times of all the timers `ths` from all processes belonging to communicator `comm` with

```
pfft_timer pfft_reduce_max_timer(
    const pfft_timer ths, MPI_Comm comm);
```

Since this function calls `MPI_Reduce`, only the first process (rank 0) of `comm` will get the desired data while all the other processes have timers with undefined values.

Note, that you can not access the elements of a timer directly, since it is only a pointer to a **struct**. However, PFFT offers a routine that creates an array and copies all the entries of the timer into it

```
double* pfft_convert_timer2vec(
    const pfft_timer ths);
```

Remember to use `free` in order to release the allocated memory of the returned array at the moment it is not needed anymore. The entries of the returned array are ordered as follows:

- dimension of the process mesh `rnk_pm`

- number of serial trafos `rnk_trafo`
- number of global remaps `rnk_remap`
- number of `pfft_execute` runs `iter`
- local run time of all runs
- `rnk_n` local times of the serial trafos
- `rnk_remap` local times of the global remaps
- 2 times of the global remaps that are only necessary for three-dimensional FFTs on three-dimensional process meshes
- time for computing twiddled input (as needed for `PFFT_SHIFTED_OUT`)
- time for computing twiddled output (as needed for `PFFT_SHIFTED_IN`)

The complementary function

```
pfft_timer pfft_convert_vec2timer(
    const double *times);
```

creates a timer and fills it's entries with the data from array `times`. Thereby, the entries of `times` must be in the same order as above.

implement getters and setters for pfft timer

6.7 Ghost Cell Communication

In the following we describe the PFFT ghost cell communication module. At the moment, PFFT ghost cell communication is restricted to three-dimensional arrays.

Does anybody need non-3d ghost cell communication?

Assume a three-dimensional array `data` of size `n[3]` that is distributed in blocks such that each process has a local copy of `data[k[0],k[1],k[2]]` with

```
local_start[t] <= k[t] < local_start[t] + local_n[t]
```

Here and in the following, we assume `t=0,1,2`. The "classical" ghost cell exchange communicates all the necessary data between neighboring processes, such that each process gets a local copy of `data[k[0],k[1],k[2]]` with

```
local_gc_start[t] <= k[t] < local_gc_start[t] + local_ngc[t]
```

where

```
local_gc_start[t] = local_start[t] - gc_below[t];
local_ngc[t] = local_n[t] + gc_below[t] + gc_above[t];
```

I.e., the local array block is increased in every dimension by `gc_below` elements below and `gc_above` elements above. Hereby, the `data` is wrapped periodically whenever `k[t]` exceeds the array dimensions. The number of ghost cells in every dimension can be chosen independently and can be arbitrary large, i.e., PFFT ghost cell communication also handles the case where the requested data exceeds next neighbor communication.

The number of ghost cells can even be bigger than the array size, which results in multiple local copies of the same data elements at every process. However, the arrays `gc_below` and `gc_above` must be equal among all MPI processes.

PFFT ghost cell communication can work on both, the input and output array distributions. Substitute `local_n` and `local_start` by `local_ni` and `local_i_start` if you are interested in ghost cell communication of the input array. For ghost cell communication of the output array, substitute `local_n` and `local_start` by `local_no` and `local_o_start`.

6.7.1 Using Ghost Cell Plans

We introduce a new datatype **`pfft_gcplan`** that stores all the necessary information for ghost cell communication. Using a ghost cell plan follows the typical workflow: At first, determine the parallel data distribution; cf. Section 6.7.2. Next, create a ghost cell plan; cf. Section 6.7.4 and Section 6.7.5. Then, execute the ghost cell communication with one of the following two collective functions

```
void pfft_exchange(
    pfft_gcplan ths);
void pfft_reduce(
    pfft_gcplan ths);
```

Hereby, a ghost cell exchange creates duplicates of local data elements on next neighboring processes, while a ghost cell reduce is the adjoint counter part of the exchange, i.e., it adds the sum of all the duplicates of a local data element to the original data element. Finally, free the allocated memory with

```
void pfft_destroy_gcplan(
    pfft_gcplan ths);
```

if the plan is not needed anymore. Passing a freed plan to `pfft_exchange` or `pfft_reduce` results in undefined behavior.

6.7.2 Data Distribution

Corresponding to the three interface layers for FFT planning, there are the following three layers for computing the ghost cell data distribution:

```
ptrdiff_t pfft_local_size_gc_3d(
    const ptrdiff_t *local_n, const ptrdiff_t *local_start,
    const ptrdiff_t *gc_below, const ptrdiff_t *gc_above,
    ptrdiff_t *local_ngc, ptrdiff_t *local_gc_start);
ptrdiff_t pfft_local_size_gc(
    int rnk_n,
    const ptrdiff_t *local_n, const ptrdiff_t *local_start,
    const ptrdiff_t *gc_below, const ptrdiff_t *gc_above,
```

```
    ptrdiff_t *local_ngc, ptrdiff_t *local_gc_start);
ptrdiff_t pfft_local_size_many_gc(
    int rnk_n,
    const ptrdiff_t *local_n, const ptrdiff_t *local_start,
    ptrdiff_t howmany,
    const ptrdiff_t *gc_below, const ptrdiff_t *gc_above,
    ptrdiff_t *local_ngc, ptrdiff_t *local_gc_start);
```

Hereby, `rnk_n` and `howmany` must be the exactly same variables that were used for the PFFT plan creation. However, only the case `rnk_n==3` is completely implemented at the moment. The local array size `local_n` must be equal to `local_ni` or `local_no` (computed by an appropriate call of `pfft_local_size`; cf. Section 6.4) depending on whether the ghost cell plan works on the FFT input or output array. Analogously, `local_start` becomes `local_i_start` or `local_o_start`. The number of ghost cells is given by the two arrays `gc_below` and `gc_above` that must be equal among all MPI processes. All the ghost cell data distribution functions return the local array plus ghost cell size `local_ngc` and the corresponding offset `local_gc_start` as two arrays of length `rnk_n`. In addition, the **`ptrdiff_t`** return value gives the number of data elements that are necessary in order to store the array plus ghost cells.

Note, that the array distribution functions do not distinguish between real and complex valued data. That is because `local_n` and `local_start` count array elements in units of complex or real depending on the transform. In addition, it does not matter if the local array is transposed or not, i.e., it is not necessary to pass the flags `PFFT_TRANSPOSED_IN` and `PFFT_TRANSPOSED_OUT` to the ghost cell distribution function. In constrast, the ghost cell plan creation depends on the transform type as well as the transposition flags.

6.7.3 Memory Allocation

In most applications we must ensure that the data array is large enough to suit the memory requirements of a parallel FFT and the ghost cell communication. The following two code snippets illustrate the correct allocation of memory in for complex valued and real valued arrays.

```
/* Get parameters of data distribution */
/* alloc_local, local_no, local_o_start are given in complex units */
/* local_ni, local_i_start are given in real units */
alloc_local = pfft_local_size_dft_r2c_3d(n, comm_cart_2d,
   PFFT_TRANSPOSED_NONE,
    local_ni, local_i_start, local_no, local_o_start);

/* alloc_local_gc, local_ngc, local_gc_start are given in complex
   units */
alloc_local_gc = pfft_local_size_gc_3d(
```

```
    local_no, local_o_start, gc_below, gc_above,
    local_ngc, local_gc_start);

/* Allocate enough memory for FFT and ghost cells */
pfft_complex *cdata = pfft_alloc_complex(alloc_local_gc >
   alloc_local ? alloc_local_gc : alloc_local);
```

Here, `alloc_local` gives the number of data elements that are necessary to hold all steps of the parallel FFT, while `alloc_local_gc` gives the number of data elements that are necessary to hold all steps of the ghost cell communication. Note that we took the maximum of these both numbers as argument for `pfft_alloc_complex`. The code snippet for real valued arrays looks very similar.

```
/* Get parameters of data distribution */
/* alloc_local, local_no, local_o_start are given in complex units */
/* local_ni, local_i_start are given in real units */
alloc_local = pfft_local_size_dft_r2c_3d(n, comm_cart_2d,
   PFFT_TRANSPOSED_NONE,
    local_ni, local_i_start, local_no, local_o_start);

/* alloc_local_gc, local_ngc, local_gc_start are given in real units
   */
alloc_local_gc = pfft_local_size_gc_3d(
    local_ni, local_i_start, gc_below, gc_above,
    local_ngc, local_gc_start);

/* Allocate enough memory for FFT and ghost cells */
double *rdata = pfft_alloc_real(alloc_local_gc > 2*alloc_local ?
   alloc_local_gc : 2*alloc_local);
```

Note that the number of real valued data elements is given by two times `alloc_local` for r2c transforms, whereas the last line would change into

```
double *rdata = pfft_alloc_real(alloc_local_gc > alloc_local ?
   alloc_local_gc : alloc_local);
```

for r2r transforms.

6.7.4 Plan Creation for Complex Data

The following functions create ghost cell plans that operate on complex valued arrays, i.e.,

- c2c inputs,
- c2c outputs,
- r2c outputs (use flag `PFFT_GC_C2R`), and
- c2r inputs (use flag `PFFT_GC_R2C`).

Corresponding to the three interface layers for FFT planning, there are the following three layers for creating a complex valued ghost cell plan:

```
pfft_gcplan pfft_plan_cgc_3d(
    const ptrdiff_t *n,
    const ptrdiff_t *gc_below, const ptrdiff_t *gc_above,
    pfft_complex *data, MPI_Comm comm_cart, unsigned gc_flags);
pfft_gcplan pfft_plan_cgc(
    int rnk_n, const ptrdiff_t *n,
    const ptrdiff_t *gc_below, const ptrdiff_t *gc_above,
    pfft_complex *data, MPI_Comm comm_cart, unsigned gc_flags);
pfft_gcplan pfft_plan_many_cgc(
    int rnk_n, const ptrdiff_t *n,
    ptrdiff_t howmany, const ptrdiff_t *block,
    const ptrdiff_t *gc_below, const ptrdiff_t *gc_above,
    pfft_complex *data, MPI_Comm comm_cart, unsigned gc_flags);
```

Hereby, `rnk_n`, `n`, `howmany` and `comm_cart` must be the variables that were used for the PFFT plan creation. However, only the case `rnk_n==3` is completely implemented at the moment. Remember that `n` is the logical FFT size just as it is the case for FFT planning. The block size `block` must be equal to `iblock` or `oblock` depending on whether the ghost cell plan works on the FFT input or output array. Analogously, `data` becomes `in` or `out`. Set the number of ghost cells by `gc_below` and `gc_above` as described in Section 6.7. The flags `gc_flags` must be set appropriately to the flags that were passed to the FFT planner. Table 6.1 shows the ghost cell planner flags that must be set in dependence on the listed FFT planner flags. In addition, we introduce the flag

FFT flag	ghost cell flag
PFFT_TRANSPOSED_NONE	PFFT_GC_TRANSPOSED_NONE
PFFT_TRANSPOSED_IN	PFFT_GC_TRANSPOSED
PFFT_TRANSPOSED_OUT	PFFT_GC_TRANSPOSED

Table 6.1: Mapping of FFT and complex valued ghost cell planner flags.

`PFFT_GC_R2C` (and its equivalent `PFFT_GC_C2R`) to handle the complex array storage format of r2c and c2r transforms. In fact, these two flags imply an ordinary complex valued ghost cell communication on an array of size `n[0] x ... x n[rnk_n-2] x (n[rnk_n-1]/2+1)`. Please note that we wrongly assume periodic boundary conditions in this case. Therefore, you should ignore the data elements with the last index behind `n[rnk_n-1]/2`.

Does anybody need r2c ghost cell communication with correct boundary conditions?

6.7.5 Plan Creation for Real Data

The following functions create ghost cell plans that operate on real valued arrays, i.e.,

- r2r inputs,
- r2r outputs,
- r2c inputs, and
- c2r outputs.

Corresponding to the three interface layers for FFT planning, there are the following three layers for creating a real valued ghost cell plan:

```
pfft_gcplan pfft_plan_rgc_3d(
    const ptrdiff_t *n,
    const ptrdiff_t *gc_below, const ptrdiff_t *gc_above,
    double *data, MPI_Comm comm_cart, unsigned gc_flags);
pfft_gcplan pfft_plan_rgc(
    int rnk_n, const ptrdiff_t *n,
    const ptrdiff_t *gc_below, const ptrdiff_t *gc_above,
    double *data, MPI_Comm comm_cart, unsigned gc_flags);
pfft_gcplan pfft_plan_many_rgc(
    int rnk_n, const ptrdiff_t *n,
    ptrdiff_t howmany, const ptrdiff_t *block,
    const ptrdiff_t *gc_below, const ptrdiff_t *gc_above,
    double *data, MPI_Comm comm_cart, unsigned gc_flags);
```

Hereby, `rnk_n`, `n`, `howmany` and `comm_cart` must be the variables that were used for the PFFT plan creation. Remember that `n` is the logical FFT size just as it is the case for FFT planning. The block size `block` must be equal to `iblock` or `oblock` depending on whether the ghost cell plan works on the FFT input or output array. Analogously, `data` becomes `in` or `out`. Set the number of ghost cells by `gc_below` and `gc_above` as described in Section 6.7.2. The flags `gc_flags` must be set appropriately to the flags that were passed to the FFT planner. Table 6.2 shows the ghost cell planner flags that must be set in dependence on the listed FFT planner flags. Note that the

FFT flag	ghost cell flag
PFFT_TRANSPOSED_NONE	PFFT_GC_TRANSPOSED_NONE
PFFT_TRANSPOSED_IN	PFFT_GC_TRANSPOSED
PFFT_TRANSPOSED_OUT	PFFT_GC_TRANSPOSED
PFFT_PADDED_R2C	PFFT_GC_PADDED_R2C
PFFT_PADDED_C2R	PFFT_GC_PADDED_C2R

Table 6.2: Mapping of FFT and real valued ghost cell planner flags.

flag `PFFT_GC_PADDED_R2C` (or its equivalent `PFFT_GC_PADDED_C2R`) implies an ordinary real valued ghost cell communication on an array of size `n[0] x ... x n[rnk_n-2]`

`x 2*(n[rnk_n-1]/2+1)`. Especially, the padding elements will be handles as normal data points, i.e., you must we aware that the numbers of ghost cells `gc_below[rnk_n-1]` and `gc_above[rnk_n-1]` include the number of padding elements.

6.7.6 Inofficial Flags

explain `PFFT_GC_SENDRECV` and `PFFT_GC_RMA`

6.7.7 Ghost Cell Execution Timer

PFFT ghost cell plans automatically accumulate the local run times of every call to `pfft_exchange` and `pfft_reduce`. For most applications it is sufficient to print run time of a plan `ths` averaged over all runs with

```
void pfft_print_average_gctimer(
    const pfft_gcplan ths, MPI_Comm comm);
```

Note, that for each timer the maximum time over all processes is reduced to rank `0` of communicator `comm`, i.e., a call to `MPI_Reduce` is performed and the output is only printed on this process. The following function works in the same way but prints more verbose output

```
void pfft_print_average_gctimer_adv(
    const pfft_gcplan ths, MPI_Comm comm);
```

To write the averaged run time of a ghost cell plan `ths` into a file called `name` use

```
void pfft_write_average_gctimer(
    const pfft_gcplan ths, const char *name, MPI_Comm comm);
void pfft_write_average_gctimer_adv(
    const pfft_gcplan ths, const char *name, MPI_Comm comm);
```

Again, the output is only written on rank `0` of communicator `comm`.

Discard all the recorded run times with

```
void pfft_reset_gctimers(
    pfft_gcplan ths);
```

This function is called per default at the end of every ghost cell plan creation function.

In order to access the run times directly a new typedef **`pfft_timer`** is introduced. The following functions return a copy of the timer corresponding to ghost cell plan `ths` that accumulated the time for ghost cell exchange or ghost cell reduce, respectively:

```
pfft_gctimer pfft_get_gctimer_exg(
    const pfft_gcplan ths);
pfft_gctimer pfft_get_gctimer_red(
    const pfft_gcplan ths);
```

Note that the memory of the returned **pfft_gctimer** must be released with

```
void pfft_destroy_gctimer(
    pfft_gctimer ths);
```

as soon as the timer is not needed anymore.

In the following we introduce some routines to perform basic operations on timers. For all functions with a **pfft_gctimer** return value you must use `pfft_destroy_gctimer` in order to release the allocated memory of the timer. Create a copy of a ghost cell timer `orig` with

```
pfft_gctimer pfft_copy_gctimer(
    const pfft_gctimer orig);
```

Compute the average, local time over all runs of a timer `ths` with

```
void pfft_average_gctimer(
    pfft_gctimer ths);
```

Create a new timer that contains the sum of two timers `sum1` and `sum2` with

```
pfft_gctimer pfft_add_gctimers(
    const pfft_gctimer sum1, const pfft_gctimer sum2);
```

Create a timer that contains the maximum times of all the timers `ths` from all processes belonging to communicator `comm` with

```
pfft_gctimer pfft_reduce_max_gctimer(
    const pfft_gctimer ths, MPI_Comm comm);
```

Since this function calls `MPI_Reduce`, only the first process (rank 0) of `comm` will get the desired data while all the other processes have timers with undefined values.

Note, that you can not access the elements of a timer directly, since it is only a pointer to a **struct**. However, PFFT offers a routine that creates an array and copies all the entries of the timer into it

```
void pfft_convert_gctimer2vec(
    const pfft_gctimer ths, double *times);
```

Remember to use `free` in order to release the allocated memory of the returned array at the moment it is not needed anymore. The entries of the returned array are ordered as follows:

- number of `pfft_execute` runs `iter`
- local run time of all runs
- local run time of zero padding (make room for incoming ghost cells and init with zeros)
- local run time of the ghost cell exchange or reduce (depending on the timer)

The complementary function

```
pfft_gctimer pfft_convert_vec2gctimer(
    const double *times);
```

creates a timer and fills it's entries with the data from array `times`. Thereby, the entries of `times` must be in the same order as above.

Do we need getters and setters for ghost cell timers?

6.8 Useful Tools

The following functions are useful tools but are not necessarily needed to perform parallel FFTs.

6.8.1 Initializing Complex Inputs and Checking Outputs

To fill a complex array `data` with reproducible, complex values you can use one of the functions

```
void pfft_init_input_complex_3d(
    const ptrdiff_t *n,
    const ptrdiff_t *local_n, const ptrdiff_t *local_n_start,
    pfft_complex *data);
void pfft_init_input_complex(
    int rnk_n, const ptrdiff_t *n,
    const ptrdiff_t *local_n, const ptrdiff_t *local_start,
    pfft_complex *data);
```

Hereby, the arrays `n`, `local_n` and `local_n_start` of length `rnk_n` (`rnk_n==3` for `_3d`) give the size of the FFT, the local array size and the local array offset as computed by the array distribution functions described in Section 6.4 The functions

```
double pfft_check_output_complex_3d(
    const ptrdiff_t *n,
    const ptrdiff_t *local_n, const ptrdiff_t *local_n_start,
    const pfft_complex *data, MPI_Comm comm);
double pfft_check_output_complex(
    int rnk_n, const ptrdiff_t *n,
    const ptrdiff_t *local_n, const ptrdiff_t *local_start,
    const pfft_complex *data, MPI_Comm comm);
```

compute the l_1-norm between the elements of array `data` and values produced by `pfft_init_input_complex_3d`, `pfft_init_input_complex`. In addition, we supply the following functions for setting all the input data to zero at once

```
void pfft_clear_input_complex_3d(
    const ptrdiff_t *n,
    const ptrdiff_t *local_n, const ptrdiff_t *local_n_start,
```

```
    pfft_complex *data);
void pfft_clear_input_complex(
    int rnk_n, const ptrdiff_t *n,
    const ptrdiff_t *local_n, const ptrdiff_t *local_start,
    pfft_complex *data);
```

Note, that these functions can be combined for a quick consistency check of the FFT. Since a forward FFT followed by a backward FFT reproduces the inputs up to a scaling factor, the following code snippet should give a result equal to zero up to machine precision.

```
/* Initialize input with random numbers */
pfft_init_input_complex_3d(n, local_ni, local_i_start,
    in);

/* execute parallel forward FFT */
pfft_execute(plan_forw);

/* clear the old input */
if(in != out)
  pfft_clear_input_complex_3d(n, local_ni, local_i_start, in);

/* execute parallel backward FFT */
pfft_execute(plan_back);

/* Scale data */
for(ptrdiff_t l=0; l < local_ni[0] * local_ni[1] * local_ni[2]; l++)
  in[l] /= (n[0]*n[1]*n[2]);

/* Print error of back transformed data */
err = pfft_check_output_complex_3d(n, local_ni, local_i_start, in,
   comm_cart_2d);
pfft_printf(comm_cart_2d, "Error after one forward and backward
   trafo of size n=(%td, %td, %td):\n", n[0], n[1], n[2]);
pfft_printf(comm_cart_2d, "maxerror = %6.2e;\n", err);
```

Hereby, we set all inputs equal to zero after the forward FFT in order to be sure that all the final results are actually computed by the backward FFT instead of being a buggy relict of the forward transform.

6.8.2 Initializing Real Inputs and Checking Outputs

To fill a real array `data` with reproducible, real values use one of the functions

```
void pfft_init_input_real_3d(
    const ptrdiff_t *n,
    const ptrdiff_t *local_n, const ptrdiff_t *local_n_start,
```

```
    double *data);
void pfft_init_input_real(
    int rnk_n, const ptrdiff_t *n,
    const ptrdiff_t *local_n, const ptrdiff_t *local_start,
    double *data);
```

Hereby, the arrays `n`, `local_n` and `local_n_start` give the size of the FFT, the local array size and the local array offset as computed by the array distribution functions described in Section 6.4 The functions

```
double pfft_check_output_real_3d(
    const ptrdiff_t *n,
    const ptrdiff_t *local_n, const ptrdiff_t *local_n_start,
    const pfft_complex *data, MPI_Comm comm);
double pfft_check_output_real(
    int rnk_n, const ptrdiff_t *n,
    const ptrdiff_t *local_n, const ptrdiff_t *local_start,
    const pfft_complex *data, MPI_Comm comm);
```

compute the l_1-norm between the elements of array `data` and values produced by `pfft_init_input_real_3d`, `pfft_init_input_real`. In addition, we supply the following functions for setting all the input data to zero at once

```
void pfft_clear_input_real_3d(
    const ptrdiff_t *n,
    const ptrdiff_t *local_n, const ptrdiff_t *local_n_start,
    double *data);
void pfft_clear_input_real(
    int rnk_n, const ptrdiff_t *n,
    const ptrdiff_t *local_n, const ptrdiff_t *local_start,
    double *data);
```

Note, that both `pfft_init_input_real*` functions will set all array elements to zero were `local_n + local_n_start >= n`. In addition, both `pfft_check_output_real*` function will ignore all the errors resulting from these elements. Therefore, it is safe to use all these functions for a consistency check of a r2c transform followed by a c2r transform since all padding elements will be ignored.

6.8.3 Initializing r2c/c2r Inputs and Checking Outputs

The real inputs of a r2c transform can be initialized with the functions decribed in Section 6.8.2. However, generating suitable inputs for a c2r transform requires more caution. In order to get real valued results of a DFT the complex input coefficients need to satisfy an radial Hermitian symmetry, i.e., $X[\boldsymbol{k}] = X^*[-\boldsymbol{k}]$. We use the following trick to generate the complex input values for c2r transforms. Assume any $\boldsymbol{N}$-periodic complex valued function f. It can be easily shown that the values $X[\boldsymbol{k}] := \frac{1}{2}\left(f(\boldsymbol{k}) + f^*(-\boldsymbol{k})\right)$ satisfy the radial Hermitian symmetry.

To fill a complex array `data` with reproducible, complex values that fulfill the radial Hermitian symmetry use one of the functions

```
void pfft_init_input_complex_hermitian_3d(
    const ptrdiff_t *n,
    const ptrdiff_t *local_n, const ptrdiff_t *local_n_start,
    double *data);
void pfft_init_input_complex_hermitian(
    int rnk_n, const ptrdiff_t *n,
    const ptrdiff_t *local_n, const ptrdiff_t *local_start,
    double *data);
```

Hereby, the arrays `n`, `local_n` and `local_n_start` give the size of the FFT, the local array size and the local array offset as computed by the array distribution functions described in Section 6.4 The functions

```
double pfft_check_output_complex_hermitian_3d(
    const ptrdiff_t *n,
    const ptrdiff_t *local_n, const ptrdiff_t *local_n_start,
    const pfft_complex *data, MPI_Comm comm);
double pfft_check_output_complex_hermitian(
    int rnk_n, const ptrdiff_t *n,
    const ptrdiff_t *local_n, const ptrdiff_t *local_start,
    const pfft_complex *data, MPI_Comm comm);
```

compute the l_1-norm between the elements of array `data` and values produced by `pfft_init_input_complex_hermitian_3d`, `pfft_init_input_complex_hermitian`. In addition, we supply the following functions for setting all the input data to zero at once

```
void pfft_clear_input_complex_hermitian_3d(
    const ptrdiff_t *n,
    const ptrdiff_t *local_n, const ptrdiff_t *local_n_start,
    pfft_complex *data);
void pfft_clear_input_complex_hermitian(
    int rnk_n, const ptrdiff_t *n,
    const ptrdiff_t *local_n, const ptrdiff_t *local_start,
    pfft_complex *data);
```

Note, that these functions can also be used in order to generate complex inputs with radial Hermitian symmetry for ordinary c2c transforms. Of course the results of such a c2c DFT will have all imaginary parts equal to zero up to machine precision.

6.8.4 Operations on Arrays of Type `ptrdiff_t`

The following routines are shortcuts for the elementwise manipulation of **`ptrdiff_t`** valued arrays. In the following, all arrays `vec`, `vec1`, and `vec2` are of length `d` and type **`ptrdiff_t`**.

```
ptrdiff_t pfft_prod_INT(
    int d, const ptrdiff_t *vec);
```

Returns the product over all elements of `vec`.

```
ptrdiff_t pfft_sum_INT(
    int d, const ptrdiff_t *vec);
```

Returns the sum over all elements of `vec`.

```
int pfft_equal_INT(
    int d, const ptrdiff_t *vec1, const ptrdiff_t *vec2);
```

Returns 1 if both arrays have equal entries, 0 otherwise.

```
void pfft_vcopy_INT(
    int d, const ptrdiff_t *vec1,
    ptrdiff_t *vec2);
```

Copies the elements of `vec1` into `vec2`.

```
void pfft_vadd_INT(
    int d, const ptrdiff_t *vec1, const ptrdiff_t *vec2,
    ptrdiff_t *sum);
```

Fills `sum` with the componentwise sum of `vec1` and `vec2`.

```
void pfft_vsub_INT(
    int d, const ptrdiff_t *vec1, const ptrdiff_t *vec2,
    ptrdiff_t *sum);
```

Fills `sum` with the componentwise difference of `vec1` and `vec2`.

6.8.5 Print Three-Dimensional Arrays in Parallel

Use the following routine to print the elements of a block decomposed three-dimensional (real or complex valued) array `data` in a nicely formatted way.

```
void pfft_apr_real_3d(
    const double *data,
    const ptrdiff_t *local_n, const ptrdiff_t *local_start,
    const char *name, MPI_Comm comm);
void pfft_apr_complex_3d(
    const pfft_complex *data,
    const ptrdiff_t *local_n, const ptrdiff_t *local_start,
    const char *name, MPI_Comm comm);
```

Obviously, this makes only sense for arrays of moderate size. The block decomposition is given by `local_n`, `local_n_start` as returned by the array distribution function decribed in Section 6.4. Furthermore, some arbitrary string `name` can be added at the beginning of each output - typically this will be the name of the array. Communicator

comm must be suitable to the block decomposition and is used to synchronize the outputs over all processes.

Generalizations for the case where the dimensions of the local arrays are permuted are given by

```
void pfft_apr_real_permuted_3d(
    const double *data,
    const ptrdiff_t *local_n, const ptrdiff_t *local_start,
    int perm0, int perm1, int perm2,
    const char *name, MPI_Comm comm);
void pfft_apr_complex_permuted_3d(
    const pfft_complex *data,
    const ptrdiff_t *local_n, const ptrdiff_t *local_start,
    int perm0, int perm1, int perm2,
    const char *name, MPI_Comm comm);
```

Hereby, perm0, perm1, and perm2 give the array's permutation of dimension.

6.8.6 Reading Command Line Arguments

The following function offers a simple way to read command line arguments into an array parameter.

```
void pfft_get_args(
    int argc, char **argv, const char *name,
    int neededArgs, unsigned type,
    void *parameter);
```

Hereby, argc and argv are the standard argument of the main routine. Furthermore, name, neededAgrs, and type give the name, number of entries and the type of the command line argument. Supported types are PFFT_INT, PFFT_PTRDIFF_T, PFFT_FLOAT, PFFT_DOUBLE, and PFFT_UNSIGNED, which denote the standard C type that is used for typecasting. In addition, you can use the special type PFFT_SWITCH that is an integer type equal to one if the corresponding command line argument is given. The array parameter must be of sufficient size to hold neededArgs elements of the given data type. Special attention is given

For example, a program containing the following code snippet

```
double x=0.1;
pfft_get_args(argc, argv, "-pfft_x", 1, PFFT_DOUBLE, &x);
int np[2]={2,1};
pfft_get_args(argc, argv, "-pfft_np", 2, PFFT_INT, np);
ptrdiff_t n[3]={32,32,32};
pfft_get_args(argc, argv, "-pfft_n", 3, PFFT_PTRDIFF_T, n);
int switch=0;
pfft_get_args(argc, argv, "-pfft_on", 0, PFFT_SWITCH, switch);
```

that is executed via

```
./test -pfft_x 3.1 -pfft_np 2 3 -pfft_n 8 16 32 -pfft_on
```

will read `x=3.1`, `np[2] = {2,3}`, `n[3]={8,16,32}`, and turn on the **`switch`**`=1`. Note the address operator `&` in front of `x` in the second line! Furthermore, note that the initialization of all variables with default values before the call of `pfft_get_args` avoids trouble if the user does not provide all the command line arguments.

6.8.7 Parallel Substitutes for `vprintf`, `fprintf`, and `printf`

The following functions are similar to the standard C function `vfprintf`, `fprintf` and `printf` with the exception, that only rank `0` within the given communicator `comm` will produce output. The intension is to avoid the flood of messages that is produced when simple `printf` statement are run in parallel.

```
void pfft_vfprintf(
    MPI_Comm comm, FILE *stream, const char *format, va_list ap);
void pfft_fprintf(
    MPI_Comm comm, FILE *stream, const char *format, ...);
void pfft_printf(
    MPI_Comm comm, const char *format, ...);
```

6.9 Generating Periodic Cartesian Communicators

Based on the processes that are part of the given communicator `comm` the following routine

```
int pfft_create_procmesh_1d(
    MPI_Comm comm, int np0,
    MPI_Comm *comm_cart_1d);
```

allocates and creates a one-dimensional, periodic, Cartesian communicator `comm_cart_1d` of size `np0`. Thereby, a non-zero error code is returned whenever `np0` does not fit the size of `comm`. The memory of the generated communicator should be released with `MPI_Comm_free` after usage. Analogously, use

```
int pfft_create_procmesh_2d(
    MPI_Comm comm, int np0, int np1,
    MPI_Comm *comm_cart_2d);
```

in order to allocate and create two-dimensional, periodic, Cartesian communicator `comm_cart_2d` of size `np0*np1` or

```
int pfft_create_procmesh(
    int rnk_np, MPI_Comm comm, const int *np,
    MPI_Comm *comm_cart);
```

in order to allocate and create a `rnk_np`-dimensional, periodic, Cartesian communicator of size `np[0]*np[1]*...*np[rnk_np-1]`. Hereby, `np` is an array of length `rnk_np`. Again, the memory of the generated communicator should be released with `MPI_Comm_free` after usage.

7 Developers Guide

7.1 Search and replace patterns

Correct alignment of pfft.h header

```
%s/^\(    [^ ]\+[^\\]*\)  \\/  \1\\/g
```

Expand most macros of pfft.h to generate the function reference of this manual:

```
sed -e 's/ *\\$//g' -e 's/PFFT_EXTERN //g' \
    -e 's/PX(\([^)]*\))/pfft_\1/g' -e 's/ INT/ ptrdiff_t/g' \
    -e 's/ R/ double/g' -e 's/ C/ pfft_complex/g' \
    -e 's/^  //g' pfft.h > pfft.h.expanded
```

8 ToDo

- `PFFT_FORWARD` is defined as `FFTW_FORWARD`
- `FFTW_FORWARD` is defined as -1
- PFFT allows to chose between `FFTW_FORWARD` and `FFTW_BACKWARD`, which is not implemented by FFTW.
- Matlab uses the same sign convention, i.e., -1 for `fft` and $+1$ for `ifftn`

8.1 Measuring parallel run times

Use `MPI_Barrier` in front of every call to `pfft_` function to avoid unbalanced run times.

Acknowledgments

We are thankful to Yu Feng who implemented the new array execute and the clear input functions.

Bibliography

[1] M. Frigo and S.G. Johnson: *FFTW users manual.* http://www.fftw.org/fftw3_doc.

[2] M. Frigo and S.G. Johnson: *FFTW users manual: 2d mpi example.* http://www.fftw.org/fftw3_doc/2d-MPI-example.html#g_t2d-MPI-example.

[3] M. Frigo and S.G. Johnson: *FFTW users manual: Complex numbers.* http://www.fftw.org/fftw3_doc/Complex-numbers.html#Complex-numbers.

[4] M. Frigo and S.G. Johnson: *FFTW users manual: Memory allocation.* http://www.fftw.org/fftw3_doc/Memory-Allocation.html#Memory-Allocation.

[5] M. Frigo and S.G. Johnson: *FFTW users manual: Mpi data distribution.* http://www.fftw.org/fftw3_doc/MPI-Data-Distribution.html#MPI-Data-Distribution.

[6] M. Frigo and S.G. Johnson: *FFTW users manual: MPI initialization.* http://www.fftw.org/fftw3_doc/MPI-Initialization.html#MPI-Initialization.

[7] M. Frigo and S.G. Johnson: *FFTW users manual: MPI initialization.* http://www.fftw.org/fftw3_doc/Real_002ddata-DFT-Array-Format.html#Real_002ddata-DFT-Array-Format.

[8] M. Frigo and S.G. Johnson: *FFTW users manual: Precision.* http://www.fftw.org/fftw3_doc/Precision.html#Precision.

[9] M. Frigo and S.G. Johnson: *FFTW users manual: SIMD alignment and fftw_malloc.* http://www.fftw.org/fftw3_doc/SIMD-alignment-and-fftw_005fmalloc.html#SIMD-alignment-and-fftw_005fmalloc.

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

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

[12] W. Gropp, E. Lusk, and R. Thakur: *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, Cambridge, MA, USA, 1999.

[13] N. Li: *2DECOMP&FFT, Parallel FFT subroutine library*. http://www.2decomp.org.

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

[15] MPI Forum: *MPI: A Message-Passing Interface Standard. Version 2.2*, 2009. http://www.mpi-forum.org.

[16] D. Pekurovsky: *P3DFFT, Parallel FFT subroutine library*. http://code.google.com/p/p3dfft.

[17] D. Pekurovsky: *P3DFFT: A Framework for Parallel Computations of Fourier Transforms in Three Dimensions*. SIAM J. Sci. Comput., 34:C192 – C209, 2012.

[18] M. Pippig: *PFFT, Parallel FFT subroutine library*, 2011. http://www.tu-chemnitz.de/~mpip/software.php.

[19] M. Pippig: *PNFFT, Parallel Nonequispaced FFT subroutine library*, 2011. http://www.tu-chemnitz.de/~mpip/software.php.

[20] M. Pippig: *PFFT - An extension of FFTW to massively parallel architectures*. SIAM J. Sci. Comput., 35:C213 – C236, 2013.

[21] M. Pippig and D. Potts: *Parallel three-dimensional nonequispaced fast Fourier transforms and their application to particle simulation*. SIAM J. Sci. Comput., accepted, 2013.

[22] S.J. Plimpton: *Parallel FFT subroutine library*. http://www.sandia.gov/~sjplimp/docs/fft/README.html.

[23] 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)*, Philadelphia, 1997. SIAM.