P3DFFT++ Documentation

Release 3.0.0

Dmitry Pekurovsky

May 25, 2022

Contents

CHAPTER 1

P3DFFT Home Page

1.1 Open Source Numerical Library for Scalable Multidimensional Fourier Transforms and related algorithms

This site provides tools for solution of numerical problems in multiscale phenomena in three dimensions (3D). The most common example of such problem is Fast Fourier Transform (FFT), which is an important algorithm for simulations in a wide range of fields, including studies of turbulence, climatology, astrophysics and material science. Other algorithms of importance include Chebyshev transforms and high-order finite difference compact schemes.

Parallel Three-Dimensional Fast Fourier Transforms, dubbed P3DFFT, as well as its extension P3DFFT++, is a library for large-scale computer simulations on parallel platforms.This project was initiated at San Diego Supercomputer Center (SDSC) at UC San Diego by its main author Dmitry Pekurovsky, Ph.D.

This library uses 2D, or pencil, decomposition. This overcomes an important limitation to scalability inherent in FFT libraries implementing 1D (or slab) decomposition: the number of processors/tasks used to run this problem in parallel can be as large as N^2, where N is the linear problem size. This approach has shown good scalability up to 524,288 cores.

P3DFFT

P3DFFT is written in Fortran90 and is optimized for parallel performance. It uses Message Passing Interface (MPI) for interprocessor communication, and starting from v.2.7.5 there is a multithreading option for hybrid MPI/OpenMP implementation. C/C++ interface is available, as are detailed documentation and examples in both Fortran and C. A configure script is supplied for ease of installation. This package depends on a serial FFT library such as Fastest Fourier Transform in the West (FFTW) or IBM's ESSL. The library is available from its github page.

P3DFFT++

P3DFFT++ is the next generation of P3DFFT (versions starting with 3.0). It extends the interface of P3DFFT to allow a wider range of use scenarios. It provides the user with a choice in defining their own data layout formats beyond the predefined 2D pencil blocks. It is written in C++ with C and Fortran interfaces, and currently uses MPI. The library can be found at P3DFFT++ github space. See P3DFFT++ Tutorial and P3DFFT++ reference pages in C++, C and Fortran.

The following table compares P3DFFT family 2.7.6 and 3.1.0 (P3DFFT++).

Feature	P3DFFT 2.x	P3DFFT++
real-to-complex and complex-to-real FFT	Yes	Yes
complex FFT	No	Yes
sine and cosine transforms	In 1 dimension	Yes
pruned transforms	Yes	No
In-place and out-of-place	Yes	Yes
Multiple grids	No	Yes
Hybrid MPI/OpenMP	Yes	No

License of use

This software is provided for free for educational and not-for-profit use under a UCSD license. License terms can be seen here. Users are requested to complete optional registration when downloading this software, and also acknowledge the use as below.

Citation information

Please acknowledge/cite use of P3DFFT as follows: D. Pekurovsky, P3DFFT: a framework for parallel computations of Fourier transforms in three dimensions, SIAM Journal on Scientific Computing 2012, Vol. 34, No. 4, pp. C192-C209. This paper can be obtained here.

To cite the software you can also use DOI for P3DFFT v. 2.7.9:

Fig. 1: This project is supported by National Science Foundation grant OAC-1835885.

CHAPTER 2

Installing P3DFFT

The latest version of P3DFFT can be found here. Once you have extracted the package, you must take the following steps to complete the setup:

1. Run the `configure` script
2. Run `make`
3. Run `make install`

2.1 How to compile P3DFFT

P3DFFT uses a `configure` script to create `Makefile` for compiling the library as well as several examples. This page will step you through the process of running the `configure` script properly.

Run `configure --help` for complete list of options. Recommended options: `--enable-stride1`. For Cray XT platforms also recommended `--enable-useeven`.

Currently the package supports four compiler suites: PGI, Intel, IBM and GNU. Some examples of compiling on several systems are given below. Users may need to customize as needed. If you wish to share more examples or to request or contribute in support for other compilers, please write to dmitry@sdsc.edu. If you give us an account on your system we will work with you to customize the installation.

Argument	Notes	Description	Example
`--prefix=PREFIX`	Mandatory for users without access to `/usr/local`	This argument will install P3DFFT to `PREFIX` when you run `make install`. By default, configure will install to `/usr/local`.	`--prefix=$HOME/local/`
`--enable-gnu`, `--enable-ibm`, `--enable-intel`, `--enable-pgi`, `--enable-cray`	Mandatory	These arguments will prepare P3DFFT to be built by a specific compiler. You must only choose one option.	`--enable-pgi`
`--enable-fftw`, `--enable-essl`	Mandatory	These arguments will prepare P3DFFT to be used with either the FFTW or ESSL library. You must only choose one option.	`--enable-fftw`
`--with-fftw=FFTWLOCATION`	Mandatory if `--enable-fftw` is used	This argument specifies the path location for the FFTW library; it is mandatory if you are planning to use P3DFFT with the FFTW library.	`--with-fftw=$FFTW_H`
`--enable-openmp`	Mandatory if using multithreaded version	This argument adds the appropriate compiler flags to enable OpenMP.	`--enable-openmp`
`--enable-openmpi`	Optional	This argument uses the OpenMPI implementation of MPI.	`--enable-openmpi`
`--enable-oned`	Optional	This argument is for 1D decomposition. The default is 2D decomposition but can be made to 1D by setting up a grid 1xN when running the code.	`--enable-oned`
`--enable-estimate`	Optional, use only with `--enable-fftw`	If this argument is passed, the FFTW library will not use run-time tuning to select the fastest algorithm for computing FFTs.	`--enable-estimate`
`--enable-measure`	Optional, enabled by default, use only with `--enable-fftw`	For search-once-for-the-fast algorithm (takes more time on `p3dfft_setup()`).	`--enable-measure`
`--enable-patient`	Optional, use only with `--enable-fftw`	For search-once-for-the-fastest-algorithm (takes much more time on `p3dfft_setup()`).	`--enable-patient`
`--enable-dimsc`	Optional	To assign processor rows and columns in the Cartesian processor grid according to C convention. The default is Fortran convention which is recommended. This option does not affect the order of storage of arrays in memory.	`--enable-dimsc`
`--enable-useeven`	Optional, recommended for Cray XT	This argument is for using `MPI_Alltoall` instead of `MPI_Alltotallv`. This will pad the send buffers with zeros to make them of equal size; not needed on most architecture but may lead to better results on Cray XT.	`--enable-useeven`
`--enable-stride1`	Optional, recommended	To enable stride-1 data structures on output (this may in some cases give some advantage in performance). You can define loop blocking factors `NLBX` and `NBLY` to experiment, otherwise they are set to default values.	`--enable-stride1`
`--enable-nblx`	Optional	To define loop blocking factor `NBL_X`	`--enable-nblx=32`
`--enable-nbly1`	Optional	To define loop blocking factor `NBL_Y1`	`--enable-nbly1=32`
`--enable-nbly2`	Optional	To define loop blocking factor `NBL_Y2`	`--enable-nbly2=32`
`--enable-nblz`	Optional	To define loop blocking factor `NBL_Z`	`--enable-nblz=32`
`--enable-single`	Optional	This argument will compile P3DFFT in single-precision. By default, configure will setup P3DFFT to be compiled in double-precision.	`--enable-single`
`FC=<Fortran compiler>`	Strongly recommended	Fortran compiler	`FC=mpif90`

2.1.1 Compiling on Comet (XSEDE/SDSC)

Choose a MPI.

Compiler	Modules	Arguments
Intel	intel, fftw	./configure –enable-intel –enable-fftw –with-fftw=$FFTWHOME FC=mpif90 CC=mpicc
GNU	gnu, fftw	./configure –enable-gnu –enable-fftw –with-fftw=$FFTWHOME FC=mpif90 CC=mpicc LDFLAGS=-lm
PGI	pgi, fftw	./configure –enable-pgi –enable-fftw –with-fftw=$FFTWHOME FC=mpif90 CC=mpicc

Compiler	Modules	Arguments
Intel	intel, fftw	./configure –enable-intel –enable-fftw –with-fftw=$FFTWHOME FC=mpif90 CC=mpicc LDFLAGS=-lmpifort
GNU	gnu, fftw	./configure –enable-gnu –enable-fftw –with-fftw=$FFTWHOME FC=mpif90 CC=mpicc LDFLAGS="-lm -lmpichf90"
PGI	pgi, fftw	./configure –enable-pgi –enable-fftw –with-fftw=$FFTWHOME FC=mpif90 CC=mpicc LDFLAGS=-lmpichf90

Compiler	Modules	Arguments
Intel	intel, fftw	./configure –enable-intel –enable-fftw –enable-openmpi –with-fftw=$FFTWHOME FC=mpif90 CC=mpicc
GNU	gnu, fftw	./configure –enable-gnu –enable-fftw –enable-openmpi –with-fftw=$FFTWHOME FC=mpif90 CC=mpicc LDFLAGS=-lm
PGI	pgi, fftw	./configure –enable-pgi –enable-fftw –enable-openmpi –with-fftw=$FFTWHOME FC=mpif90 CC=mpicc

2.1.2 Compiling on Stampede2 (XSEDE/TACC)

Choose a MPI.

Compiler	Modules	Arguments
Intel	intel, fftw3	./configure –enable-intel –enable-fftw –with-fftw=$TACC_FFTW3_DIR FC=mpif90 CC=mpicc
GNU	gcc	./configure –enable-gnu –enable-fftw –with-fftw=/PATH/TO/FFTW/LIBRARY FC=mpif90 CC=mpicc LDFLAGS=-lm

Note: User must install their own FFTW library for GNU compilers while using Intel MPI due to technical difficulties.

Compiler	Modules	Arguments
Intel	intel	./configure –enable-intel –enable-fftw –with-fftw=/PATH/TO/FFTW/LIBRARY FC=mpif90 CC=mpicc LDFLAGS=-lmpifort

Note: Stampede2's FFTW module is not compatible with its MVAPICH2 module yet. Users must install their own FFTW library.

2.1.3 Compiling on Bridges (PSC)

Choose a MPI.

Compiler	Modules	Arguments
Intel	intel, fftw3	./configure –enable-intel –enable-fftw –with-fftw=$FFTW3_LIB/.. FC=mpiifort CC=mpicc LDFLAGS=-lm

Compiler	Modules	Arguments
Intel	intel, fftw3	./configure –enable-intel –enable-fftw –with-fftw=$FFTW3_LIB/.. FC=mpif90 CC=mpicc LDFLAGS=-lmpifort
GNU	gcc, fftw3	./configure –enable-gnu –enable-fftw –with-fftw=$FFTW3_LIB/.. FC=mpif90 CC=mpicc LDFLAGS="-lm -lmpichf90"

Compiler	Modules	Arguments
Intel	intel, fftw3	./configure –enable-intel –enable-fftw –enable-openmpi –with-fftw=$FFTW3_LIB/.. FC=mpif90 CC=mpicc
GNU	fftw3	./configure –enable-gnu –enable-fftw –enable-openmpi –with-fftw=$FFTW3_LIB/.. FC=mpif90 CC=mpicc LDFLAGS=-lm
PGI	pgi, fftw3	./configure –enable-pgi –enable-fftw –enable-openmpi –with-fftw=$FFTW3_LIB/.. FC=mpif90 CC=mpicc

CHAPTER 3

Download P3DFFT

The latest release of P3DFFT can be downloaded here or accessed through its GitHub page.

Be sure to familiarize yourself with the installation instructions in order to build the library.

For more detail and usage instructions, see *P3DFFT User Guide*.

CHAPTER 4

P3DFFT User Guide

Version 2.7.5

4.1 Acknowledgements

- Prof. P.K.Yeung
- Dr. Diego Donzis
- Dr. Giri Chukkapalli
- Dr. Geert Brethouwer

Citation: when reporting results obtained with P3DFFT, please cite the following:

D. Pekurovsky, "P3DFFT: a framework for parallel computations of Fourier transforms in three dimensions", **SIAM Journal on Scientific Computing** 2012, Vol. 34, No. 4, pp. C192-C209.

4.2 Introduction

P3DFFT is a scalable software library implementing three-dimensional spectral transforms. It has been used in a variety of codes from many areas of computational science. It has been tested and used on many high-end computational system. It uses two-dimensional domain decomposition in order to overcome a scaling bottleneck of one-dimensional decomposition. This allows the programs with this library to scale well on a large number of cores, consistent with bisection bandwidth scaling of interconnect of the underlying hardware system.

Below are the main features of P3DFFT v. 2.7.5:

- Real-to-complex and complex-to-real Fast Fourier Transforms (FFT) in 3D.
- Cosine, sine, and combined Fourier-Chebyshev transform (FFT in 2D and Chebyshev in the third dimension). Alternatively users can substitute their own transform in the third dimension, for example a compact scheme.
- Fortran and C interfaces
- Built for performance at all ranges of core counts
- Hybrid MPI/OpenMP implementation
- In-place and out-of-place transforms
- Pruned transforms
- Multivariable transforms

4.3 1. Directory Structure and Files

The following is a directory listing for what you should find in the `p3dfft` package:

Table 1: Directory structure of `p3dfft` package

Directory	Description
toplevel	The `configure` script is located here. Running the `configure` script is essential for properly building P3DFFT. Please refer to section 2 of this guide for more information.
`build`	The library files are contained here. Building the library is required before it can be used. In order to build the library, you must run `./configure` from the top level directory. Then type `make` and then `make install`. For further details on building the library see section 2 of this guide.
`include`	The library is provided as a Fortran module. After installation this directory will have `p3dfft.mod` (for Fortran interface), `p3dfft.h` (C wrapper/include file), and `config.h` (header that contains all arguments used when configure script was executed).
`sample/`	This directory has example programs in both FORTRAN and C, in separate subdirectories. Tests provided include out-of-place and in-place transforms 3D FFT, with error checking. Also provided is an example of power spectrum calculation. Example programs will be compiled automatically with the library during make.

Warning: In order to use P3DFFT with C programs, you must include the `p3dfft.h` header file in your program. This header file defines an interface that allows C programs to call Fortran functions from the P3DFFT library.

In addition to the library itself, the package includes several sample programs to illustrate how to use P3DFFT. These sample programs can be found in the `sample/` directory:

Table 2: Filename and description of samples

Source filename	Binary filename	Description
`driver_inverse.c`, `driver_inverse.F90`	`test_inverse_c.x`, `test_inverse_f.x`	This program initializes a 3D array of complex numbers with a 3D sine/cosine wave, then performs inverse FFT transform, and checks that the results are correct. This sample program also demonstrates how to work with complex arrays in wavenumber space, declared as real.
`driver_rand.c`, `driver_rand.F90`	`test_rand_c.x`, `test_rand_f.x`	This program initializes a 3D array with random numbers, then performs forward 3D Fourier transform, then backward transform, and checks that the results are correct, namely the same as in the start except for a normalization factor. It can be used both as a correctness test and for timing the library functions.
`driver_sine.c`, `driver_sine_inplace.c`, `driver_sine.F90`, `driver_sine_inplace.F90`	`test_sine_c.x`, `test_sine_inplace_c.x`, `test_sine_f.x`, `test_sine_inplace_f.x`	This program initializes a 3D array with a 3D sine wave, then performs 3D forward Fourier transform, then backward transform, and checks that the results are correct, namely the same as in the start except for a normalization factor. It can be used both as a correctness test and for timing the library functions.
`driver_sine_many_f.F90`, `driver_sine_inplace_many_f.F90`, `driver_rand_many_f.F90`	`test_sine_many_f.x`, `test_sine_inplace_many_f.x`, `test_rand_many_f.x`	Same as above, but these program tests the multivariable transform feature. There is an extra parameter in the input file specifying the number of variables to transform (`nv`).
`driver_spec.c`, `driver_spec.F90`	`test_spec_c.x`, `test_spec_f.x`	This program initializes a 3D array with a 3D sine wave, then performs 3D FFT forward transform, and computes power spectrum.
`driver_cheby.f90`	`test_cheby_f.x`	This program initializes a 3D array with a sine wave, employing a non-uniform grid in the Z dimension with coordinates given by cos(k/N). Then Chebyshev routine is called (`p3dfft_cheby`) which uses Fourier transform in X and Y and a cosine transform in Z ("ffc"), followed by computation of Chebyshev coefficients. Then backward "cff" transform is called and the results are compared with the expected output after Chebyshev differentiation in Z. This program can be used both as correctness and as a timing test.
`driver_noop.c`, `driver_noop.F90`	`test_noop_c.x`, `test_noop_f.x`	Similar to the above but instead of Chebyshev transform nothing is done; i.e. only 2D FFT is performed and then the data is laid out in a format suitable for a custom transform of the user's choice in the third dimension (i.e. data is local for each processor in that dimension).

4.4 2. Installing p3dfft

In order to prepare the P3DFFT for compiling and installation, you must run the included `configure` script. Here is a simple example on how to run the `configure` script:

```
$ ./configure --enable-pgi --enable-fftw --with-fftw=/usr/local/fftw/ LDFLAGS="-lmpi_
↪f90 -lmpi_f77"
```

The above will prepare P3DFFT to be compiled by the PGI compiler with FFTW support. There are more arguments included in the `configure` script that will allow you to customize P3DFFT to your requirements:

Table 3: Arguments of `configure` script

Argument	Notes	Description	Example
`--prefix=PREFIX`	Mandatory for users without access to `/usr/local`	This argument will install P3DFFT to `PREFIX` when you run `make install`. By default, configure will install to `/usr/local`.	`--prefix=$HOME/local/`
`--enable-gnu`, `--enable-ibm`, `--enable-intel`, `--enable-pgi`, `--enable-cray`	Mandatory	These arguments will prepare P3DFFT to be built by a specific compiler. You must only choose one option.	`--enable-pgi`
`--enable-fftw`, `--enable-essl`	Mandatory	These arguments will prepare P3DFFT to be used with either the FFTW or ESSL library. You must only choose one option.	`--enable-fftw`
`--with-fftw=FFTWLOCATION`	Mandatory if `--enable-fftw` is used	This argument specifies the path location for the FFTW library; it is mandatory if you are planning to use P3DFFT with the FFTW library.	`--with-fftw=$FFTW_H`
`--enable-openmp`	Mandatory if using multithreaded version	This argument adds the appropriate compiler flags to enable OpenMP.	`--enable-openmp`
`--enable-openmpi`	Optional	This argument uses the OpenMPI implementation of MPI.	`--enable-openmpi`
`--enable-oned`	Optional	This argument is for 1D decomposition. The default is 2D decomposition but can be made to 1D by setting up a grid 1xN when running the code.	`--enable-oned`
`--enable-estimate`	Optional, use only with `--enable-fftw`	If this argument is passed, the FFTW library will not use run-time tuning to select the fastest algorithm for computing FFTs.	`--enable-estimate`
`--enable-measure`	Optional, enabled by default, use only with `--enable-fftw`	For search-once-for-the-fast algorithm (takes more time on `p3dfft_setup()`).	`--enable-measure`
`--enable-patient`	Optional, use only with `--enable-fftw`	For search-once-for-the-fastest-algorithm (takes much more time on `p3dfft_setup()`).	`--enable-patient`
`--enable-dimsc`	Optional	To assign processor rows and columns in the Cartesian processor grid according to C convention. The default is Fortran convention which is recommended. This option does not affect the order of storage of arrays in memory.	`--enable-dimsc`
`--enable-useeven`	Optional, recommended for Cray XT	This argument is for using `MPI_Alltoall` instead of `MPI_Alltotallv`. This will pad the send buffers with zeros to make them of equal size; not needed on most architecture but may lead to better results on Cray XT.	`--enable-useeven`
`--enable-stride1`	Optional, recommended	To enable stride-1 data structures on output (this may in some cases give some advantage in performance). You can define loop blocking factors `NLBX` and `NBLY` to experiment, otherwise they are set to default values.	`--enable-stride1`
`--enable-nblx`	Optional	To define loop blocking factor `NBL_X`	`--enable-nblx=32`
`--enable-nbly1`	Optional	To define loop blocking factor `NBL_Y1`	`--enable-nbly1=32`
`--enable-nbly2`	Optional	To define loop blocking factor `NBL_Y2`	`--enable-nbly2=32`
`--enable-nblz`	Optional	To define loop blocking factor `NBL_Z`	`--enable-nblz=32`
`--enable-single`	Optional	This argument will compile P3DFFT in single-precision. By default, configure will setup P3DFFT to be compiled in double-precision.	`--enable-single`
`FC=<Fortran compiler>`	Strongly recommended	Fortran compiler	`FC=mpif90`

More information on how to customize the `configure` script can be found by calling:

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

For a up-to-date list of configure commands for various platforms please refer to *Installing P3DFFT* page.

After you have successfully run the `configure` script, you are ready to compile and install P3DFFT. Simply run:

```
$ make
$ make install
```

4.5 3. p3dfft module

The `p3dfft` module declares important variables. It should be included in any code that calls `p3dfft` routines (via using `p3dfft` statement in Fortran).

The `p3dfft` module also specifies `mytype`, which is the type of real and complex numbers. You can choose precision at compile time through a preprocessor flag (see *Installing P3DFFT* page).

4.6 4. Initialization

Before using the library it is necessary to call an initialization routine `p3dfft_setup`.

Usage:

```
p3dfft_setup(proc_dims, nx, ny, nz, mpi_comm_in, nx_cut, ny_cut, nz_cut, overwrite,␣
↪memsize)
```

Table 4: Arguments of `p3dfft_setup`

Argument	Intent	Description
proc_dims	Input	An array of two integers, specifying how the processor grid should be decomposed. Either 1D or 2D decomposition can be specified. For example, when running on 12 processors, (4,3) or (2,6) can be specified as proc_dims to indicate a 2D decomposition, or (1,12) can be specified for 1D decomposition. *proc_dims* values are used to initialize P1 and P2.
nx, *ny*, *nz*	Input	(Integer) Dimensions of the 3D transform (also the global grid dimensions)
mpi_comm_in	Input	(Integer) MPI Communicator containing all MPI tasks that participate in the partition (in most cases this will be `MPI_COMM_WORLD`).
nx_cut, *ny_cut*, *nz_cut*	Input (optional)	(Integer) Pruned dimensions on output/input (default is same as *nx*, *ny*, *nz*)
overwrite	Input (optional)	(Logical) When set to `true`. (or `1` in C) this argument indicates that it is safe to overwrite the input of the btran (backward transform) routine. This may speed up performance of FFTW routines in some cases when non-stride-1 transforms are made.
memsize	Output (optional)	Optional argument (array of 3 integers). Memsize can be used to allocate arrays. It contains the dimensions of real-space array that are large enough to contain both input and output of an in-place 3D FFT real-to-complex transform defined by *nx*, *ny*, *nz*, *nx_cut*, *ny_cut*, *nz_cut*.

4.7 5. Array Decomposition

The `p3dfft_setup` routine sets up the two-dimensional (2D) array decomposition. P3DFFT employs 2D block decomposition whereby processors are arranged into a 2D grid P1 x P2, based on their MPI rank. Two of the dimensions of the 3D grid are block-distributed across the processor grid, by assigning the blocks to tasks in the rank order. The third dimension of the grid remains undivided, i.e. contained entirely within local memory (see Fig. 1). This scheme is sometimes called pencils decomposition.

A block decomposition is defined by dimensions of the local portion of the array contained within each task, as well as the beginning and ending indices for each dimension defining the array's location within the global array. This information is returned by `p3dfft_get_dims` routine which should be called before setting up the data structures of your program (see `sample/` subdirectory for example programs).

In P3DFFT, the decompositions of the output and input arrays, while both being two-dimensional, differ from each other. The reason for this is as follows. In 3D Fourier Transform it is necessary to transpose the data a few times (two times for two-dimensional decomposition) in order to rearrange the data so as to always perform one-dimensional FFT on data local in memory of each processing element. It would be possible to transpose the data back to the original form after the 3D transform is done, however it often makes sense to save significant time by forgoing this final transpose. All the user has to do is to operate on the output array while keeping in mind that the data are in a transposed form. The backward (complex-to-real) transform takes the array in a transposed form and produces a real array in the original form. The rest of this section clarifies exactly the original and transposed form of the arrays.

Starting with v. 2.7.5 P3DFFT features optional hybrid MPI/OpenMP implementation. In this case the MPI decomposition is the same as above, and each MPI task now has Nthr threads. This essentially implements 3D decomposition, however the results are global arrays (in the OpenMP sense) so they can be used either with multi- or single-threaded program. The number of threads is specified through the environment variable `OMP_NUM_THREADS`.

Usage:

```
p3dfft_get_dims(start,end,size,ip)
```

Table 5: Arguments of `p3dfft_get_dims()`

Argument	Intent	Description
start	Output	An array containing 3 integers, defining the beginning indices of the local array for the given task within the global grid.
end	Output	An array containing 3 integers, defining the ending indices of the local array within the global grid (these can be computed from *start* and *size* but are provided for convenience).
size	Output	An array containing 3 integers, defining the local array's dimensions.
mypad	Output/Optional	This argument is optional and is used in in-place transforms, to obtain the value of padding that should be used in the third dimension of the input array (since input and output arrays may not have the same memory size)
ip	Input	`ip=1`: "Original": a "physical space" array of real numbers, local in X, distributed among P1 tasks in Y dimension and P2 tasks in Z dimension, where P1 and P2 are processor grid dimensions defined in the call to p3dfft_setup. Usually this type of array is an input to real-to-complex (forward) transform and an output of complex-to-real (backward) transform. `ip=2`: "Transposed": a "wavenumber space" array of complex numbers, local in Z, distributed among P1 tasks in X dimension, P2 tasks in Y dimension. Usually this type of array is an output of real-to- complex (forward) transform and an input to complex-to-real, backward transform. `ip=3`: the routine returns three numbers corresponding to "padded" dimensions in the physical space, i.e. an array with these dimensions will be large enough both for physical and wavenumber space. Example of use of this feature can be found in `driver_sine_inplace.F90` sample program.

Warning: The layout of the 2D processor grid on the physical network is dependent on the architecture and software of the particular system, and can have some impact on efficiency of communication. By default, rows have processors with adjacent task IDs (this corresponds to "FORTRAN" type ordering). This can be changed to "C" ordering (columns have adjacent task IDs) by building the library with `-DDIMS_C` preprocessor flag. The former way is recommended on most systems.

P3DFFT uses 2D block decomposition to assign local arrays for each task. In many cases decomposition will not be entirely even: some tasks will get more array elements than others. P3DFFT attempts to minimize load imbalance. For example, the grid dimensions are 128x256x256 and the processor grid is defined as 3x4, the original (`ip=1`) decomposition calls for splitting 256 elements in Y dimension into three processor row. In this case, P3DFFT will break it up into pieces of 86, 85 and 85 elements. The transposed (`ip=2`) decomposition will have local arrays with X dimensions 22, 22 and 21 respectively for processor rows 1 through 3 (the sum of these numbers is 65=(nx+2)/2 since these are now complex numbers instead of reals, and an extra mode for Nyquist frequency is needed – see Section 5 for an explanation).

It should be clear that the user's choice of P1 and P2 can make a difference on how balanced is the decomposition. Obviously the greater load imbalance, the less performance can be expected.

Note: The two array types are distributed among processors in a different way from each other, but this does not automatically imply anything about the ordering of the elements in memory. Memory layout of the original (`ip=1`) array uses the "Fortran" ordering. For example, for an array A(lx,ly,lz) the index corresponding to lx runs fastest. Memory layout for the transposed (`ip=2`) array type depends on how the P3DFFT library was built. By default, it preserves the ordering of the real array, i.e. (X,Y,Z). However, in many cases it is advisable to have Z dimension contiguous, i.e. a memory layout (Z,Y,X). This can speed up some computations in the wavenumber space by improving cache utilization through spatial locality in Z, and also often results in better performance of P3DFFT transforms themselves. The (Z,Y,X) layout can be triggered by building the library with `-DSTRIDE1` preprocessor flag in the makefile. For more information, see performance section below.

Table 6. Mapping of the data array onto processor grid and memory layout

	Physical space	Fourier space
`STRIDE1` defined	Nx, Ny/M1, Nz/M2	Nz, Ny/M2, (Nx+2)/(2M1)
`STRIDE1` undefined	Nx, Ny/M1, Nz /M2	(Nx+2)/(2M1), Ny/M2 ,Nz

4.8 6. Forward (real-to-complex) and backward (complex-to-real) 3D Fourier transforms

P3DFFT versions 2.7.1 and higher implement transforms for one or more than one independent arrays/variables simultaneously. An example of this is 3 components of a velocity field. Multivariable transforms achieve greater speed than single-variable transforms, especially for grids of smaller size, due to buffer aggregation in inter-processor exchanges.

Forward transform is implemented by the `p3dfft_ftran_r2c` subroutine using the following format:

```
p3dfft_ftran_r2c(IN,OUT,op)
```

The input **IN** is an array of real numbers with dimensions defined by array type with `ip=1` (see Table 2 above), with X dimension contained entirely within each task, and Y and Z dimensions distributed among P1 and P2 tasks correspondingly. The output **OUT** is an array of complex numbers with dimensions defined by array type with `ip=2`, i.e. Z dimension contained entirely, and X and Y dimensions distributed among P1 and P2 tasks respectively. The **op**

argument is a 3- letter character string indicating the type of transform desired. Currently only Fourier transforms are supported in X and Y (denoted by symbol *f*) and the following transforms in Z:

Table 7. Suuported types of transforms in Z

t or *f*	Fourier Transform
c	Cosine Transform
s	Sine Transform
n or *0*	Empty transform (no operation takes place, output is the same as input)

Empty transform can be useful for someone implementing custom transform in Z dimension. Example: `op='ffc'` means Fourier transform in X and Y, and a cosine transform in Z. The DCT-I kind of transform is performed (DST-I for sine), the definition of which can be found here.

Backward transform is implemented by the `p3dfft_btran_c2r` subroutine using the following format:

```
p3dfft_btran_c2r(IN,OUT,op)
```

The input **IN** is an array of complex numbers with dimensions defined by array type with `ip=2` (see Table 2 above), i.e. Z dimension is contained entirely, and X and Y dimensions are distributed among P1 and P2 tasks correspondingly. The output **OUT** is an array of real numbers with dimensions defined by array type with `ip=1`, i.e. X dimension is contained entirely within each task, and Y and Z are dimensions distributed among P1 and P2 tasks respectively. The **op** argument is similar to forward transform, with the first character of the string being one of *t*, *c*, *s*, *n*, or *0*, and the second and third being *f*. Example: `op='nff'` means no operation in Z, backward Fourier transforms in Y and X.

4.9 7. Complex array storage definition

Since Fourier transform of a real function has the property of conjugate symmetry, only about half of the complex Fourier coefficients need to be kept. To be precise, if the input array has n real elements, Fourier coefficients F(k) for k=n/2+1,..,n can be dropped as they can be easily restored from the rest of the coefficients. This saves both memory and time. In this version we do not attempt to further pack the complex data. Therefore the output array for the forward transform (and the input array of the backward transform) contains (*nx*/2+1) times *ny* times *nz* complex numbers, with the understanding that *nx*/2-1 elements in X direction are missing and can be restored from the remaining elements. As mentioned above, the *nx*/2+1 elements in the X direction are distributed among P1 tasks in the transposed layout.

4.10 8. Multivariable transforms

Sometime communication performance of transposes such as those included in P3DFFT can be improved by combining several transforms into a single operation. (This allows us to aggregate messages during interprocessor data exchange). This is especially important when transforming small grids and/or when using systems with high interconnect latencies. P3DFFT provides multivariable transforms to make use of this idea. Instead of an 3D array as input parameter these subroutines accept a 4D array, with the extra dimension being the index of independent variables to be transformed (for example this could be 3 velocity components). The following is the syntax for multivariable transforms:

```
p3dfft_ftran_many_r2c(IN,dim_in,OUT,dim_out,nv,op)

p3dfft_btran_many_c2r(IN,dim_in,OUT,dim_out,nv,op)
```

The multivariable transform routines for both forward and backward transforms have an additional argument **nv** (integer) representing the number of independent variables in the input/output arrays. The spacing between these independent variables is defined by **dim_in** and **dim_out** (integer) arguments for input/output arrays respectively. Both

dim_in and **dim_out** should not be less than the size of the grid returned by `get_dims` routine. See sample program `driver_sine_many.F90`, `driver_sine_inplace_many.F90`, or `driver_rand_many.F90` for an example of such use.

4.11 9. Pruned transforms

Sometimes only a subset of output modes is needed to be retained (for forward transform), or a subset of input modes is used, the rest being zeros (for backward transform). Such transforms are called pruned transforms. Leaving off redundant modes can lead to significant savings of time and memory. The reduced dimensions **nx_cut**, **ny_cut**, and **nz_cut** are arguments to `p3dfft_setup`. By default they are equal to `nx`, `ny`, `nz`. If they are different from the above (smaller) the output of forward transforms will be reduced in size correspondingly. The input for backward transform will also be smaller in size. It will be automatically padded with zeros until it reaches *nx*, *ny*, *nz*.

4.12 10. In-place transforms

In and Out arrays can occupy the same space in memory (in-place transform). In this case, it is necessary to make sure that they start in the same location, otherwise the results are unpredictable. Also it is important to remember that the sizes of input and output arrays in general are not equal. The complex array is usually bigger since it contains the Nyquist frequency mode in X direction, in addition to the *nx*/2 modes that equal in space to *nx* real numbers. However when decomposition is not even, sometimes the real array can be bigger than the complex one, depending on the task ID. Therefore to be safe one must make sure the common-space array is large enough for both input and output. This can be done by using memsize argument when calling `p3dfft_setup`. It returns the maximum array size for both input and output. Alternatively, one can call `p3dfft_get_dims` two times with `ip=1` and `ip=2`.

In Fortran using in-place transforms is a bit tricky due to language restrictions on subroutine argument types (i.e., one of the arrays is expected to be real and the other complex). In order to overcome this problem wrapper routines are provided, named `ftran_r2c` and `btran_c2r` respectively for forward and backward transform (without `p3dfft` prefix). There are examples for such in-place transform in the `sample/` subdirectory. These wrappers can be also used for out-of-place transforms just as well.

4.13 11. Memory requirements

Besides the input and output arrays (which can occupy the same space, as mentioned above) P3DFFT allocates temporary buffers roughly 3 times the size of the input or output array.

4.14 12. Performance considerations

P3DFFT was created to compute 3D FFT in parallel with high efficiency. In particular it is aimed for applications where the data volume is large. It is especially useful when running applications on ultra-scale parallel platforms where one-dimensional decomposition is not adequate. Since P3DFFT was designed to be portable, no effort is made to do architecture-specific optimization. However, the user is given some choices in setting up the library, mentioned below, that may affect performance on a given system. Current version of P3DFFT uses ESSL or FFTW library for it 1D FFT routines. ESSL [1] provides FFT routines highly optimized for IBM platforms it is built on. The FFTW [2], while being generic, also makes an effort to maximize performance on many kinds of architectures. Some performance data will be uploaded at the P3DFFT Web site. For more questions and comments please contact dmitry@sdsc.edu.

Optimal performance on many parallel platforms for a given number of cores and problem size will likely depend on the choice of processor decomposition. For example, given a processor grid P1 x P2 (specified in the first argument

to `p3dfft_setup`) performance will generally be better with smaller P1 (with the product P1 x P2 kept constant). Ideally P1 will be equal or less than the number of cores on an SMP node or a multi-core chip. In addition, the closer a decomposition is to being even, the better load balancing.

Beginning with v.2.7.5 P3DFFT is equipped with MPI/OpenMP capability. If use of this feature is needed simply set the desired number of threads through environment variable `OMP_NUM_THREADS`. The optimal number of threads, just like the processor grid, depends on specific platform and problem.

Performance is likely to be better when P3DFFT is built using `--enable-stride1` during configure. This implies stride-1 data ordering for FFTs. Note that using this argument changes the memory layout of the transposed array (see section 3 for explanation). To help tune performance further, two more arguments can be used: `--enable-dnblx=...` and `--enable-dnbly=...`, which define block sizes in X and Y when doing local array reordering. Choosing suitable block sizes allows the program to optimize cache performance, although by default P3DFFT chooses these values based on a good guess according to cache size.

Finally, performance will be better if overwrite parameter is set to `true`. (or `1` in C) when initializing P3DFFT. This allows the library to overwrite the input array, which results in significantly faster execution when not using the `--enable-stride1` argument.

4.15 13. References

1. ESSL library, IBM, http://publib.boulder.ibm.com/infocenter/clresctr/vxrx/index.jsp?topic=/com.ibm.cluster.essl.doc/esslbooks.html
2. Matteo Frigo and Steven G. Johnson, "The Design and Implementation of FFTW3", Proceedings of the IEEE 93 (2), 216–231 (2005). Invited paper, Special Issue on Program Generation, Optimization, and Platform Adaptation.
3. D. Pekurovsky, “P3DFFT: a framework for parallel computations of Fourier transforms in three dimensions”, SIAM Journal on Scientific Computing 2012, Vol. 34, No. 4, pp. C192-C209.

CHAPTER 5

Installing P3DFFT++

The latest version of P3DFFT++ can be found here. Once you have extracted the package, you must take the following steps to complete the setup:

1. Run the `configure` script
2. Run `make`
3. Run `make install`

5.1 How to compile P3DFFT++

P3DFFT++ uses a `configure` script to create `Makefile` for compiling the library as well as several examples. This page will step you through the process of running the `configure` script properly.

Run `configure --help` for complete list of options. For Cray XT platforms also recommended `--enable-useeven`.

Currently the package supports four compiler suites: PGI, Intel, IBM and GNU. Some examples of compiling on several systems are given below. Users may need to customize as needed. If you wish to share more examples or to request or contribute in support for other compilers, please write to dmitry@sdsc.edu. If you give us an account on your system we will work with you to customize the installation.

Argument	Notes	Description	Example
`--prefix=PREFIX`	Mandatory for users without access to `/usr/local`	This argument will install P3DFFT++ to `PREFIX` when you run `make install`. By default, `configure` will install to `/usr/local`.	`--prefix=$HOME/local/` .
`--enable-gnu`, `--enable-ibm`, `--enable-intel`, `--enable-pgi`, `--enable-cray`	Mandatory	These arguments will prepare P3DFFT++ to be built by a specific compiler. You must only choose one option.	`--enable-pgi`
`--enable-fftw`, `--enable-essl`	Mandatory	These arguments will prepare P3DFFT++ to be used with either the FFTW or ESSL library. You must only choose one option.	`--enable-fftw`
`--with-fftw=FFTW_DIR`	Mandatory if `--enable-fftw` is used	This argument specifies the path location for the FFTW librarys directory; it is mandatory if you are planning to use P3DFFT++ with the FFTW library.	`--with-fftw=$FFTW_D`
`--enable-estimate`	Optional, use only with `--enable-fftw`	If this argument is passed, the FFTW library will not use run-time tuning to select the fastest algorithm for computing FFTs.	`--enable-estimate`
`--enable-measure`	Optional, enabled by default, use only with `--enable-fftw`	For search-once-for-the-fast algorithm (takes more time on `p3dfft_setup()`).	`--enable-measure`
`--enable-patient`	Optional, use only with `--enable-fftw`	For search-once-for-the-fastest-algorithm (takes much more time on `p3dfft_setup()`).	`--enable-patient`
`--enable-dimsc`	Optional	To assign processor rows and columns in the Cartesian processor grid according to C convention. The default is Fortran convention which is recommended. This option does not affect the order of storage of arrays in memory.	`--enable-dimsc`
`FC=<Fortran compiler>`	Strongly recommended	Fortran compiler	`FC=mpif90`
`FCFLAGS="<Fortran compiler flags>"`	Optional, recommended	Fortran compiler flags	`FCFLAGS="-O3"`
`CC=<C compiler>`	Strongly Recommended	C compiler	`CC=mpicc`
`CFLAGS="<C compiler flags>"`	Optional, recommended	C compiler flags	`CFLAGS="-O3"`
`CXX=<C++ compiler>`	Strongly Recommended	C++ compiler	`CXX=mpicxx`
`CXXFLAGS="<C++ compiler flags>"`	Optional, recommended	C++ compiler flags	`CXXFLAGS="-O3"`
`LDFLAGS="<linker flags>"`	Optional	Linker flags	

5.1.1 Compiling on Comet (XSEDE/SDSC)

Compiler	Modules	Arguments
Intel	intel, fftw	./configure –enable-intel –enable-fftw –with-fftw=$FFTWHOME FC=mpif90 CC=mpicc CXX=mpicxx
GNU	gnu, fftw	./configure –enable-gnu –enable-fftw –with-fftw=$FFTWHOME FC=mpif90 CC=mpicc CXX=mpicxx
PGI	pgi, fftw	./configure –enable-pgi –enable-fftw –with-fftw=$FFTWHOME FC=mpif90 CC=mpicc CXX=mpicxx

5.1.2 Compiling on Stampede2 (XSEDE/TACC)

Compiler	Modules	Arguments
Intel	intel, fftw3	./configure –enable-intel –enable-fftw –with-fftw=$TACC_FFTW3_DIR FC=mpif90 CC=mpicc CXX=mpicxx
GNU	gcc, fftw3	./configure –enable-gnu –enable-fftw –with-fftw=$TACC_FFTW3_DIR FC=mpif90 CC=mpicc CXX=mpicxx

5.1.3 Compiling on Bridges (PSC)

Compiler	Modules	Arguments
Intel	intel, fftw3	./configure –enable-intel –enable-fftw –with-fftw=$FFTW3_LIB/.. FC=mpiifort CC=mpiicc CXX=mpiicpc
GNU	gcc, fftw3	./configure –enable-gnu –enable-fftw –with-fftw=$FFTW3_LIB/.. FC=mpif90 CC=mpicc CXX=mpicxx
PGI	pgi, fftw3	./configure –enable-pgi –enable-fftw –with-fftw=$FFTW3_LIB/.. FC=mpif90 CC=mpicc CXX=mpicxx

5.1.4 Compiling on Summit (OLCF)

Compiler	Modules	Arguments
GNU	gcc, fftw	./configure –enable-gnu –enable-fftw –with-fftw=$OLCF_FFTW_ROOT FC=mpif90 CC=mpicc CXX=mpiCC
PGI	pgi, fftw	./configure –enable-pgi –enable-fftw –with-fftw=$OLCF_FFTW_ROOT FC=mpif90 CC=mpicc CXX=mpiCC

CHAPTER 6

Download P3DFFT++

The latest release of P3DFFT++ can be downloaded here.

To install, modify the `Makefile` to set C++ MPI-enabled compiler appropriate to your system. Also edit the location of FFTW or MKL libraries. Several site-specific makefiles are provided as examples. `make lib` will build the library.

In addition, if you wish to build example programs in C++, C and/or Fortran, follow the same steps for `Makefile` in each of the subdirectories. Then type `make samples` in the top directory to build all 3 language examples. `make all` or `make builds` both library and examples.

CHAPTER 7

P3DFFT++ Documentation

The new generation of P3DFFT library (dubbed P3DFFT++ or P3DFFT v.3) is a generalization of the concept of P3DFFT for more arbitrary data formats and transform functions. It is now in beta testing and is available from its GitHub page. Language-specific reference is provided in Tutorial as well as in separate Reference documents (see C++, C and Fortran). Makefile examples are provided for each language to illustrate how P3DFFT++ is to be linked with a user program.

7.1 P3DFFT++ Tutorial

7.1.1 General considerations

P3DFFT++ is written in C++ and contains wrappers providing easy interfaces with C and Fortran.

For C++ users all P3DFFT++ objects are defined within the `p3dfft` namespace, in order to avoid confusion with user-defined objects. For example, to initialize P3DFFT++ it is necessary to call the function `p3dfft::setup()`, and to exit P3DFFT++ one should call `p3dfft::cleanup()` (alternatively, one can use namespace `p3dfft` and call `setup()` and `cleanup()`). From here on in this document we will omit the implicit `p3dfft::` prefix from all C++ names.

In C and Fortran these functions become `p3dfft_setup` and `p3dfft_cleanup`. While C++ users can directly access `P3DFFT` objects such as `DataGrid` class, C and Fortran users will access these through handles provided by corresponding wrappers (see more details below).

7.1.2 Data types

P3DFFT++ currently operates on four main data types:

1. `float` (single precision floating point)
2. `double` (double precision floating point)
3. `mycomplex` (single precision complex number) (equivalent to `complex<float>`)

4. `complex_double` (double precision complex number) (equivalent to `complex<double>`)

7.1.3 Data layout

While P3DFFT had the assumption of predetermined 2D pencils in X and in Z dimensions as the primary data storage, P3DFFT++ relaxes this assumption to include more general formats, such as arbitrary shape and memory order 2D pencils as well as 3D blocks. Below is the technical description of how to specify the data layout formats.

Specification of data layout consists of data and processor grid descriptors. `ProcGrid` decsriptor contains dimensions of the procesor grid (in up to 3 dimensions) and the associated MPI Cartesian sub-communicators. `DataGrid` is a metadata descriptor, containing information about data grid dimensions and their mapping onto processor grid and local memory. In C++ each of these descriptors is defined as a class, while in C and in Fortran it is defined through handles to a C++ object through inter-language wrappers. Below is the technical description of the definition for each language.

C++

The following is a main constructor call for the `ProcGrid` class:

```
ProcGrid(int pdims[3],MPI_Comm mpicomm)
```

The following is the main constructor call for the `DataGrid` class:

```
DataGrid(int gdims[3],int dim_conj_sym, ProcGrid *pgrid,int dmap[3],int mem_order[3])
```

Argument	Description
gdims	The three global dimensions of the grid to be decomposed. Here the Fortran-inspired convention is followed: the first of the three numbers specifies the dimension with the fastest changing index, i.e. the first logical dimension (X).
dim_conj_sym	The dimension of conjugate symmetry where we store N/2+1 of the data after Real-to-complex transform due to conjugate symmety;(-1 for none)
pgrid	The processor grid to be used in decomposition. See above.
dmap	The mapping of data grid dimensions into processor grid dimensions. For example, dmap=(1,0,2) implies second data dimension being spanned by the first processor grid dimension, first data dimension being spanned by the second processor grid dimension, and the third data dimension is mapped onto third processor dimension.
mem_order	The relative ordering of the three dimensions in memory within the local portion of the grid. Here C-style indexing is used (indices start with 0). The simplest ordering {0,1,2} corresponds to the first logical dimension being stored with the fastest changing index (memory stride=1), followed by the second (stride=L0) and the third dimension (stride=L0*L1), where Li is the size of local grid in i's dimension for a given MPI task. This corresponds to a C array A[L2][L1][L0]. As another example, ordering {2,0,1} means that the second dimension (L1) is stored with the fastest-changing index (memory stride=1), the third dimension dimension (L2) with the medium stride =L1, and the first dimension is stored with the slowest index, stride=L1*L2. This would correspond to a C array A[L0][L2][L1].

Here is an example where we first define a 2D processor grid with dimensions 2x4 and then define a data grid mapping onto the processor grid.

```
main() {

...

    int gdims= {128, 128, 128};

    int pdims[]={1,2,4};

    int dmap[] = {0,1,2};    //X-pencil

    int mem_order={0,1,2};

    ProcGrid *pgrid = new ProcGrid(pdims,MPI_COMM_WORLD);

    DataGrid mygrid(gdims, -1, pgrid, dmap, mem_order);
}
```

Upon construction the `DataGrid` object defines several useful parameters, available by accessing the following public class members of `DataGrid`:

Member	Descripton
int Ldims[3]	Dimensions of the local portion of the `DataGrid` (`ldims[0]=gdims[0]/pdims[0]` etc). Note: these dimensions are specified in the order of logical grid dimensions and may differ from memory storage order, which is defined by *mem_order*.
int nd	Number of dimensions of the processor grid (1, 2 or 3).
int L[3]	0 to 3 local dimensions (i.e. not split).
int D[3]	0 to 3 split dimensions.
int GlobStart[3]	Coordinates of the lowest element of the local grid within the global array. This is useful for reconstructing the global grid from grid pieces for each MPI task.

and other useful information. The `DataGrid` class also provides a copy constructor.

To release a `DataGrid` object, simply `delete` it.

C

For C users, grid initialization is accomplished by a call to `p3dfft_init_proc_grid` and `p3dfft_init_data_grid`. the latter returns a pointer to an object of type `Grid`. This type is a C structure containing a large part of the C++ class `DataGrid`. Calling `p3dfft_init_data_grid` initializes the C++ `DataGrid` object and also copies the information into a `Grid` object accessible from C, returning its pointer. For example:

```
int xdim,pgrid;

int dmap[] = {0,1,2};
int mem_order[] = {0,1,2};
int pdims[] = {1,2,4};

Grid *grid1;
```

(continues on next page)

(continued from previous page)

```
pgrid = p3dfft_init_proc_grid(pdims,MPI_COMM_WORLD);

grid1 = p3dfft_init_data_grid(gdims, dim_conj_sym, pgrid, dmap, mem_order, mpicomm);

xdim = grid1->Ldims[0]; /* Size of zero logical dimension of the local portion of the␣
↪grid for a given processor */
```

To release a grid object simply execute:

```
p3dfft_free_data_grid(Grid *gr);
```

Fortran

For Fortran users the `ProcGrid` and `DataGrid` objects are represented as handles of type `integer(C_INT)`. For example:

```
integer(C_INT) pgrid,grid1

integer ldims(3),glob_start(3),gdims(3),dim_conj_sym,pgrid,pdims(3),dmap(3),mem_
↪order(3)

pgrid = p3dfft_init_proc_grid(pdims,MPI_COMM_WORLD)

grid1 = p3dfft_init_data_grid(ldims, glob_start, gdims, dim_conj_sym, pgrid, dmap,␣
↪mem_order)
```

This call initializes a C++ `DataGrid` object as a global variable and assigns an integer ID, returned in this example as `grid1`. In addition this call also returns the dimensions of the local portion of the `DataGrid` (`Ldims`) and the position of this portion within the global array (`GlobStart`).

Other elements of the C++ `DataGrid` object can be accessed through respective functions, such as `p3dfft_grid_get_....`

To release a `DataGrid` object, simply call:

```
p3dfft_free_data_grid_f(gr)
```

where `gr` is the `DataGrid` handle.

7.1.4 P3DFFT++ Transforms

P3DFFT++ aims to provide a versatile toolkit of algorithms/transforms in frequent use for solving multiscale problems. To give the user maximum flexibility there is a range of algorithms from top-level algorithms operating on the entire 3D array, to 1D algorithms which can function as building blocks the user can arrange to suit his/her needs. In addition, inter-processor exchanges/transposes are provided, so as to enable the user to rearrange the data from one orientation of pencils to another, as well as other types of exchanges. In P3DFFT++ the one-dimensional transforms are assumed to be expensive in terms of memory bandwidth, and therefore such transforms are performed on local data (i.e. in the dimension that is not distributed across processor grid). Transforms in three dimensions consist of three transforms in one dimension, interspersed by inter-processor interchange as needed to rearrange the data. The 3D transforms are high-level functions saving the user work in arranging the 1D transforms and transposes, as well as often providing superior performance. **We recommend to use 3D transforms whenever they fit the user's algorithm.**

Although syntax for C++, C and Fortran is different, using P3DFFT++ follows the same logic. P3DFFT++ functions in a way similar to FFTW: first the user needs to plan a transform, using a planner function once per each transform

type. The planner function initializes the transform, creates a plan and stores all information relevant to this transform inside P3DFFT++. The users gets a handle referring to this plan (the handle is a class in C++, and an integer variable in C or Fortran) that can be later used to execute this transform, which can be applied multiple times. The handles can be released after use.

In order to define and plan a transform (whether 1D or 3D, in C++, C or Fortran) one needs to first define initial and final `DataGrid` objects. They contain all the necessary grid decomposition parameters. P3DFFT++ figures out the optimal way to transpose the data between these two `DataGrid` configurations, assuming they are consistent (i.e. same grid size, number of tasks etc).

7.1.5 One-dimensional (1D) Transforms

1D transforms is the smaller building block for higher dimensional transforms in P3DFFT++. They include different flavors of Fast Fourier Transforms (FFTs), empty transform (provided for convenience, as in the case where a user might want to implement their own 1D transform, but is interested in memory reordering to arrange the transform dimension for stride-1 data access), and (in the future) other transforms that share the following property: they are memory bandwidth and latency intensive, and are optimally done when the dimension the transform operates on is entirely within one MPI task's domain.

1D transforms can be done with or without data exchange and/or memory reordering. In general, combining a transform with an exchange/reordering can be beneficial for performance due to cache reuse, compared to two separate calls to a transform and an exchange.

The following predefined 1D transforms are available (in C++ the `P3DFFT_` prefix can be omitted if used within `P3DFFT` namespace).

Transform	Description
`P3DFFT_EMPTY_TYPE`	Empty transform.
`P3DFFT_R2CFFT_S`, `P3DFFT_R2CFFT_D`	Real-to-complex forward FFT (as defined in FFTW manual), in single and double precision respectively.
`P3DFFT_C2RFFT_S`, `P3DFFT_C2RFFT_D`	Complex-to-real backward FFT (as defined in FFTW manual), in single and double precision respectively.
`P3DFFT_CFFT_FORWARD_S`, `P3DFFT_CFFT_FORWARD_D`	Complex forward FFT (as defined in FFTW manual), in single and double precision respectively.
`P3DFFT_CFFT_BACKWARD_S`, `P3DFFT_CFFT_BACKWARD_D`	Complex backward FFT (as defined in FFTW manual), in single and double precision respectively.
`P3DFFT_DCT<x>_REAL_S`, `P3DFFT_DCT1_REAL_D`	Cosine transform for real-numbered data, in single and double precision, where <*x*> stands for the variant of the cosine transform, such as `DCT1`, `DCT2`, `DCT3`, or `DCT4`.
`P3DFFT_DST<x>_REAL_S`, `P3DFFT_DST1_REAL_D`	Sine transform for real-numbered data, in single and double precision, where <*x*> stands for the variant of the cosine transform, such as `DST1`, `DST2`, `DST3`, or `DST4`.
`P3DFFT_DCT<x>_COMPLEX_S`, `P3DFFT_DCT1_COMPLEX_D`	Cosine transform for complex-numbered data, in single and double precision, where <*x*> stands for the variant of the cosine transform, such as `DCT1`, `DCT2`, `DCT3`, or `DCT4`.
`P3DFFT_DST<x>_COMPLEX_S`, `P3DFFT_DST1_COMPLEX_D`	Sine transform for complex-numbered data, in single and double precision, where <*x*> stands for the variant of the cosine transform, such as `DST1`, `DST2`, `DST3`, or `DST4`.

C++

Below is an example of how a 1D transform can be called from C++. In this example, real-to-complex transform in double precision is planned and then performed. First a constructor for class transplan is called:

```
transplan<double,complex_double> trans_f(gridIn, gridOut, R2C_FFT_D, dim, false);
```

Here **gridIn** and **gridOut** are initial and final `DataGrid` objects, describing, among other things, initial and final memory ordering of the grid storage array (ordering can be the same or different for input and output). **dim** is the dimension/rank to be transformed. Note that this is the logical dimension rank (0 for X, 1 for Y, 2 for Z), and may not be the same as the storage dimension, which depends on `mem_order` member of **gridIn** and **gridOut**. The transform dimension of the `DataGrid` is assumed to be MPI task-local. The second last parameter is a bool variable telling P3DFFT++ whether this is an in-place or out-of-place transform. Note that in C++ the `P3DFFT_` prefix for transform types is optional.

When a `transplan` constructor is called as above, P3DFFT++ stores the parameters of the 1D transform and if needed, plans its execution (i.e. as in FFTW planning) and stores the plan handle. This needs to be done once per transform type. In order to execute the transform, simply call `exec` member of the class, e.g.:

```
trans_f.exec((char *) In,(char *) Out);
```

Here **In** and **Out** are pointers to input and output arrays. In this case they are of type `double` and `complex_double`, however in this call they are cast as `char*`, as required by P3DFFT++. They contain the local portion of the 3D input and output arrays, arranged as a contiguous sequence of numbers according to local grid dimensions and the memory order of **gridIn** and **gridOut** classes, respectively. If the transform is out-of-place, then these arrays must be non-overlapping. The execution can be performed many times with the same handle and same or different input and output arrays.This call will perform the 1D transform specified when the `transplan` object was constructed, along the dimension **dim**. Again, the logical dimension specified as **dim** in the planning stage must be MPI-local for both input and output arrays. Other utilities allow the user to transpose the grid arrays in MPI/processor space (see `MPIplan` and `transMPIplan`).

To release the transform handle simply `delete` the `transplan` class object.

C

Here is an example of initializing and executing a 1D transform (again, a real-to-complex double precision FFT) in a C program.

```
Grid *gridIn, *gridOut;

Plan3D trans_f;

...

gridIn = p3dfft_init_data_grid(gdimsIn, pgrid, DmapIn, mem_orderIn);
gridOut = p3dfft_init_data_grid(gdimsOut, pgrid, DmapOut, mem_orderOut);

trans_f = p3dfft_plan_1Dtrans(gridIn, gridOut, P3DFFT_R2CFFT_D, dim, 0);
```

Here `gridIn` and `gridOut` are pointers to the C equivalent of P3DFFT++ `DataGrid` object (initial and final), `trans_f` is the handle for the 1D transform after it has been initialized and planned, `dim` is the logical dimension of the transform (`0`, `1`, or `2`), and the last argument indicates that this is not an in-place transform (a non-zero argument would indicate in-place). This initialization/planning needs to be done once per transform type.

```
p3dfft_exec_1Dtrans_double(trans_f,IN,OUT);
```

This statement executes the 1D transformed planned and handled by `trans_f`. **IN** and **OUT** are pointers to one-dimensional input and output arrays containing the 3D grid stored contiguously in memory based on the local grid dimensions and storage order of `gridIn` and `gridOut`. The execution can be performed many times with the same handle and same or different input and output arrays. In case of out-of-place transform the input and output arrays must be non-overlapping.

Fortran

Here is an example of initializing and executing a 1D transform (again, a real-to-complex double precision FFT) in a Fortran program:

```
integer(C_INT) gridIn,gridOut
integer trans_f

gridIn = p3dfft_init_grid(ldimsIn, glob_startIn, gdimsIn, pgrid, dmapIn, mem_orderIn)
gridOut = p3dfft_init_grid(ldimsOut, glob_startOut, gdimsOut, pgrid, dmapOut, mem_
↪orderOut)
trans_f = p3dfft_plan_1Dtrans_f(gridIn, gridOut, P3DFFT_R2CFFT_D, dim-1, 0)
```

These statement set up initial and final grids (`gridIn` and `gridOut`), initialize and plan the 1D real-to-complex double FFT and use `trans_f` as its handle. This needs to be done once per transform type. Note that we need to translate the transform dimension dim into C convention (so that X corresponds to 0, Y to 1 and Z to 2). The last argument is `0` for out-of-place and non-zero for in-place transform.

```
call p3dfft_1Dtrans_double(trans_f,Gin,Gout)
```

This statement executes the 1D transform planned before and handled by `trans_f`. **Gin** and **Gout** are 1D contiguous arrays of values (double precision and double complex) of the 3D grid array, according to the local grid dimensions and memory storage order of `gridIn` and `gridOut`, respectively. After the previous planning step is complete, the execution can be called many times with the same handle and same or different input and output arrays. If the transform was declared as out-of-place then **Gin** and **Gout** must be non-overlapping.

7.1.6 Three-dimensional Transforms

As mentioned above, three-dimensional (3D) transforms consist of three one-dimensional transforms in sequence (one for each dimension), interspersed by inter-processor transposes. In order to specify a 3D transform, five main things are needed:

1. Initial `DataGrid` (as described above, `DataGrid` object defines all of the specifics of grid dimensions, memory ordering and distribution among processors).
2. Final `DataGrid`.
3. The type of 3D transform.
4. Whether this is in-place transform
5. Whether this transform can overwrite input

The final `DataGrid` may or may not be the same as the initial `DataGrid`. First, in real-to-complex and complex-to-real transforms the global grid dimensions change for example from (n0, n1, n2) to (n0/2+1, n1, n2), since most applications attempt to save memory by using the conjugate symmetry of the Fourier transform of real data. Secondly, the final `DataGrid` may have different processor distribution and memory ordering, since for example many applications with convolution and those solving partial differential equations do not need the initial `DataGrid` configuration in Fourier space. The flow of these applications is typically 1) transform from physical to Fourier space, 2) apply convolution or derivative calculation in Fourier space, and 3) inverse FFT to physical space. Since forward FFT's last step is 1D FFT in the third dimension, it is more efficient to leave this dimension local and stride-1, and since the first step of the inverse FFT is to start with the third dimension 1D FFT, this format naturally fits the algorithm and results in big savings of time due to elimination of several extra transposes.

In order to define the 3D transform type one needs to know three 1D transform types comprising the 3D transform. Usage of 3D transforms is different depending on the language used and is described below.

C++

In C++ 3D transform type is interfaced through a class `trans_type3D`, which is constructed as in the following example:

```
trans_type3D name_type3D(int types1D[3]);
```

Here **types1D** is the array of three 1D transform types which define the 3D transform (empty transforms are permitted). Copy constructor is also provided for this class.

For example:

```
int type_rcc, type_ids[3];

type_ids[0] = P3DFFT_R2CFFT_D;
type_ids[1] = P3DFFT_CFFT_FORWARD_D;
type_ids[2] = P3DFFT_CFFT_FORWARD_D;

trans_type3D mytype3D(type_ids);
```

3D transforms are provided as the class template:

```
template<class TypeIn,class TypeOut> class transform3D;
```

Here **TypeIn** and **TypeOut** are initial and final data types. Most of the times these will be the same, however some transforms have different types on input and output, for example real-to-complex FFT. In all cases the floating point precision (single/double) of the initial and final types should match.

The constructor of `transform3D` takes the following arguments:

```
transform3D<TypeIn,TypeOut> my_transform_name(gridIn,gridOut,type,inplace,overwrite);
```

Here type is a 3D transform type (constructed as shown above), **inplace** is a bool variable indicating whether this is an in-place transform, and **overwrites** (also boolean) defines if the input can be rewritten (default is false). **gridIn** and **gridOut** are initial and final `DataGrid` objects. Calling a `transform3D` constructor creates a detailed step-by-step plan for execution of the 3D transform and stores it in the `my_transform_name` object.

Once a 3D transform has been defined and planned, execution of a 3D transform can be done by calling:

```
my_transform_name.exec(TypeIn *in,TypeOut *out);
```

Here **in** and **out** are initial and final data arrays of appropriate types. These are assumed to be one-dimensional contiguous arrays containing the three-dimensional grid for input and output, local to the memory of the given MPI task, and stored according to the dimensions and memory ordering specified in the **gridIn** and **gridOut** objects, respectively. For example, if `grid1.ldims={2,2,4}` and `grid1.mem_order={2,1,0}`, then the in array will contain the following sequence: G000, G001, G002, G003, G010, G011, G012, G013, G100, G101, G102, G103, G110, G111, G112, G113. Again, we follow the Fortran convention that the fastest running index is the first, (i.e. G012 means the grid element at X=0, Y=1, Z=2).

C

In C a unique datatype `Type3D` is used to define the 3D transform needed. `p3dfft_init_3Dtype` function is used to initialize a new 3D transform type, based on the three 1D transform types, as in the following example:

```
int type_rcc,  type_ids[3];

```

(continues on next page)

(continued from previous page)

```
type_ids[0] = P3DFFT_R2CFFT_D;
type_ids[1] = P3DFFT_CFFT_FORWARD_D;
type_ids[2] = P3DFFT_CFFT_FORWARD_D;

type_rcc = p3dfft_init_3Dtype(type_ids);
```

In this example `type_rcc` will describe the real-to-complex (R2C) 3D transform (R2C in 1D followed by two complex 1D transforms).

To define and plan the 3D transform, use `p3dfft_plan_3Dtrans` function as follows:

```
int mytrans;

mytrans = p3dfft_plan_3Dtrans(gridIn,gridOut,type,inplace,overwrite);
```

Here **gridIn** and **gridOut** are pointers to initial and final `DataGrid` objects (of type `Grid`); **type** is the 3D transform type defined as above; **inplace** is an integer indicating an in-place transform if it's non-zero, out-of-place otherwise. **overwrite** is an integer defining if the input can be overwritten (non-zero; default is zero). In this example `mytrans` contains the handle to the 3D transform that can be executed (many times) as follows:

```
p3dfft_exec_3Dtrans_double(mytrans,in,out);
```

Here **in** and **out** are pointers to input and output arrays, as before, assumed to be the local portion of the 3D grid array stored according to **gridIn** and **gridOut** descriptors. For single precision use `p3dfft_exec_3Dtrans_single`.

Fortran

In Fortran, similar to C, to define a 3D transform the following routine is used:

```
mytrans = p3dfft_plan_3Dtrans_f(gridIn,gridOut,type,inplace, overwrite)
```

Here **gridIn** and **gridOut** are handles defining the initial and final `DataGrid` configurations; **type** is the 3D transform type, defined as above; and **inplace** is the integer whose non-zero value indicates this is an in-place transform (or 0 for out-of-place). Non-zero **overwrite** indicates it is OK to overwrite input (default is no). Again, this planner routine is called once per transform. Execution can be called multiple times as follows:

```
call p3dfft_3Dtrans_double(mytrans,IN,OUT)
```

Here **IN** and **OUT** are the input and output arrays. For single precision use `p3dfft_3Dtrans_single_f`.

7.2 P3DFFT++ C++ Reference

Contents

7.2.1 Introduction

For C++ users all P3DFFT++ objects are defined within the `p3dfft` namespace, in order to avoid confusion with user-defined objects. For example, to initialize P3DFFT++ it is necessary to call the function `p3dfft::setup()`, and to exit P3DFFT++ one should call `p3dfft::cleanup()` (alternatively, one can use namespace `p3dfft` and call `setup()` and `cleanup()`). From here on in this document we will omit the implicit `p3dfft::` prefix from all C++ names.

7.2.2 Setup and Grid layout

ProcGrid constructor

The public portion of the `ProcGrid` class is below:

```
class ProcGrid {

 public:
     int taskid,numtasks;
     int nd;  //number of dimensions the volume is split over
     int ProcDims[3];  //Processor grid size (in inverse order of split dimensions\
          , i.e. rows first, then columns etc
     int grid_id_cart[3];
     MPI_Comm mpi_comm_glob; // Global MPi communicator we are starting from
     MPI_Comm mpi_comm_cart;
     MPI_Comm mpicomm[3]; //MPI communicators for each dimension

     ProcGrid(int procdims[3],MPI_Comm mpi_comm_init);
```

DataGrid constructor

The public portion of the `DataGrid` class is below:

```
class DataGrid {

...

  public :


    int nd; //number of dimensions the volume is split over

    int Gdims[3]; //Global dimensions

    dim_conj_sym; // dimension of the array in which a little less than half of the␣
↪elements are omitted due to conjugate symmetry. This argument should be non-
↪negative only for complex-valued arrays resulting from real-to-complex FFT in the␣
↪given dimension.

    int MemOrder[3]; //Memory ordering inside the data volume
    int Ldims[3]; //Local dimensions on THIS processor
    int Pdims[3]; // Dimensions of Processor grid (as mapped onto data dimensions)
    int Dmap[3]; // Mapping of data dimensions onto processor dimensions
    int D[3]; //Ranks of Dimensions of physical grid split over rows and columns␣
↪correspondingly
    int L[3]; //Rank of Local dimension (p=1)
    ProcGrid *Pgrid;
    int grid_id[3]; //Position of this pencil/cube in the processor grid
    int GlobStart[3]; // Starting coords of this cube in the global grid
    // int (*st)[3],(*sz)[3],(*en)[3]; // Lowest, size and uppermost location in 3D,␣
↪for each processor in subcommunicator
    int **st[3],**sz[3],**en[3]; // Lowest, size and uppermost location in 3D, for␣
↪each processor in subcommunicator

    bool is_set;
    DataGrid(int gdims_[3],ProcGrid *pgrid,int dmap[3],int mem_order[3]);
    DataGrid(const DataGrid &rhs);
    DataGrid() {};
    ~DataGrid();
    void set_mo(int mo[3]) {for(int i=0;i<3;i++) mem_order[i] = mo[i];};
...
};
```

grid constructor

```
DataGrid::DataGrid(int gdims[3],int dim_conj_sym,ProcGrid *pgrid,int dmap[3],int mem_
↪order[3])
```

Function: Initializes a new `DataGrid` with specified parameters.

Argument	Description
gdims	Three global grid dimensions (logical order - X, Y, Z)
dim_conjugate	Dimension of conjugate symmetry, non-negative only for complex arrays resulting from real-to-complex FFT in the given dimension. This is logical, not storage, dimension, with valid numbers 0 - 2, and -1 implying no conjugate symmetry.
pgrid	A pointer to a processor grid this data grid is living on.
dmap	A permutation of the 3 integers: `0`, `1` and `2`. Specifies mapping of data dimensions onto processor grid dimensions. For example, dmap=(1,0,2) implies second data dimension being spanned by the first processor grid dimension, first data dimension being spanned by the second processor grid dimension, and the third data dimension is mapped onto third processor dimension.
mem_order	A permutation of the 3 integers: 0, 1 and 2. Specifies mapping of the logical dimension and memory storage dimensions for local memory for each MPI task. `mem_order[i0] = 0` means that the i0's logical dimension is stored with `stride=1` in memory. Similarly, `mem_order[i1] = 1` means that i1's logical dimension is stored with `stride=ldims[i0]` etc
mpicomm	The MPI communicator in which this `DataGrid` lives

7.2.3 P3DFFT++ Transforms

P3DFFT++ functions in a way similar to FFTW: first the user needs to plan a transform, using a planner function once per each transform type. The planner function initializes the transform, creates a plan and stores all information relevant to this transform inside P3DFFT++. The users gets a handle referring to this plan (which is a class in C++) that can be later used to execute this transform, and can be applied multiple times. The handles can be released after use.

In order to define and plan a transform (whether 1D or 3D) one needs to first define initial and final `DataGrid` objects. They contain all the necessary grid decomposition parameters. P3DFFT++ figures out the optimal way to transpose the data between these two grid configurations, assuming they are consistent (i.e. same grid size, number of tasks etc).

One-Dimensional (1D) Transforms

The following predefined 1D transforms are available:

Transform	Description
EMPTY_TYPE	Empty transform.
R2CFFT_S, P3DFFT_R2CFFT_D	Real-to-complex forward FFT (as defined in FFTW manual), in single and double precision respectively.
C2RFFT_S, P3DFFT_C2RFFT_D	Complex-to-real backward FFT (as defined in FFTW manual), in single and double precision respectively.
CFFT_FORWARD_S, CFFT_FORWARD_D	Complex forward FFT (as defined in FFTW manual), in single and double precision respectively.
CFFT_BACKWARD_S, CFFT_BACKWARD_D	Complex backward FFT (as defined in FFTW manual), in single and double precision respectively.
DCT<x>_REAL_S, DCT1_REAL_D	Cosine transform for real-numbered data, in single and double precision, where <x> stands for the variant of the cosine transform, such as DCT1, DCT2, DCT3, or DCT4.
DST<x>_REAL_S, DST1_REAL_D	Sine transform for real-numbered data, in single and double precision, where <x> stands for the variant of the cosine transform, such as DST1, DST2, DST3, or DST4.
DCT<x>_COMPLEX_S, DCT1_COMPLEX_D	Cosine transform for complex-numbered data, in single and double precision, where <x> stands for the variant of the cosine transform, such as DCT1, DCT2, DCT3, or DCT4.
DST<x>_COMPLEX_S, DST1_COMPLEX_D	Sine transform for complex-numbered data, in single and double precision, where <x> stands for the variant of the cosine transform, such as DST1, DST2, DST3, or DST4.

Custom transform types

Custom 1D transforms can be defined by the user through `trans_type1D` class template.

```
template <class Type1,class Type2> class trans_type1D : public gen_trans_type{

    int ID;

  public :

    typedef long (*doplan_type)(const int *n,int howmany,Type1 *in,const int *inembed,
↪int istride,int idist,Type2 *out,const int *onembed,int ostride,int odist,...);

    long (*doplan)(...);
    void (*exec)(...);

    trans_type1D(const char *name, long (*doplan_)(...),void (*exec)(...)=NULL,int␣
↪isign=0);
    inline int getID() {return(ID);}
    trans_type1D(const trans_type1D &rhs);
    ~trans_type1D();
};
```

This class template is a derivative of `gen_trans_type1D` class, defined as follows:

```
class gen_trans_type {
  public :
    char *name;
    int isign; // forward (-1) or backward (+1), in case this is complex FFT
    bool is_set,is_empty;
    int dt1,dt2; //Datatype before and after
    int prec; // precision for a real value in bytes (4 or 8)
    gen_trans_type(const char *name_,int isign_=0);
    ~gen_trans_type();
```

(continues on next page)

(continued from previous page)

```
    bool operator==(const gen_trans_type &) const;
};
```

In order to define a custom transform type, the user needs to provide planning and execution functions (`doplan` and `exec`). For example, in case of a complex FFT implemented through FFTW, the following is how the transform type is constructed:

```
char *name = "Complex-to-complex Fourier Transform, forward transform, double␣
↪precision";
int isign = FFTW_FORWARD;
trans_type1D<complex_double,complex_double> *mytype = new trans_type1D<complex_double,
↪complex_double>(name,(long (*)(...) ) fftw_plan_many_dft,(void (*)(...)) exec_c2c_d,
↪isign);
```

where `exec_c2c_d` is defined as follows:

```
void exec_c2c_d(long plan,complex_double *in,complex_double *out)
{
    fftw_execute_dft((fftw_plan) plan,(fftw_complex *) in,(fftw_complex *) out);
}
```

Planning 1D transform

1D transform in C++ is realized through `transplan` template class. `TypeIn` and `TypeOut` are the datatypes for input and output.

Two constructors are provided.

```
template <class TypeIn,class TypeOut> class transplan::transplan(const grid &gridIn,
↪const grid &gridOut,const gen_trans_type *type,const int d, const bool inplace_);

template <class TypeIn,class TypeOut> class transplan::transplan(const grid &gridIn,
↪const grid &gridOut,const int type,const int d, const bool inplace_);
```

Function: Defines and plans a 1D transform of a 3D array.

Argument	Description
gridIn	Initial `DataGrid` descriptor
gridOut	Final `DataGrid` descriptor
type	The type of the 1D transform (either as a predefined integer parameter, or as a class `gen_trans_type`.
d	The dimension to be transformed. Note that this is the logical dimension rank (0 for X, 1 for Y, 2 for Z), and may not be the same as the storage dimension, which depends on `mem_order` member of *gridIn* and *gridOut*. The transform dimension of the `DataGrid` is assumed to be MPI task-local.
inplace	`True` for in-place transform, `false` for out-of-place.

Releasing 1D transform handle

To release a 1D transform handle, simply `delete` the corresponding `transplan` class.

Executing 1D transform

```
template <class TypeIn,class TypeOut> class transplan::exec(char *In, char *Out);
```

Function: Executes the pre-planned 1D transform of a 3D array.

Argument	Description
In, *Out*	Pointers to input and output arrays, cast as `pointer` to `char`. They contain the local portion of the 3D input and output arrays, arranged as a contiguous sequence of numbers according to local grid dimensions and the memory order of initial and final `DataGrid` objects respectively.

Note: If the transform is out-of-place, then these arrays must be non-overlapping. The execution can be performed many times with the same handle and same or different input and output arrays.

Three-dimensional Transforms

Three-dimensional (3D) transforms consist of three one-dimensional transforms in sequence (one for each dimension), interspersed by inter-processor transposes. In order to specify a 3D transform, three main things are needed:

1. Initial `DataGrid` (as described above, `DataGrid` object defines all of the specifics of grid dimensions, memory ordering and distribution among processors).
2. Final `DataGrid`.
3. The type of 3D transform.

The final `DataGrid` may or may not be the same as the initial `DataGrid`. First, in real-to-complex and complex-to-real transforms the global grid dimensions change for example from (n0, n1, n2) to (n0/2+1 ,n1, n2), since most applications attempt to save memory by using the conjugate symmetry of the Fourier transform of real data. Secondly, the final `DataGrid` may have different processor distribution and memory ordering, since for example many applications with convolution and those solving partial differential equations do not need the initial `DataGrid` configuration in Fourier space. The flow of these applications is typically 1) transform from physical to Fourier space, 2) apply convolution or derivative calculation in Fourier space, and 3) inverse FFT to physical space. Since forward FFT's last step is 1D FFT in the third dimension, it is more efficient to leave this dimension local and stride-1, and since the first step of the inverse FFT is to start with the third dimension 1D FFT, this format naturally fits the algorithm and results in big savings of time due to elimination of several extra transposes.

In order to define the 3D transform type one needs to know three 1D transform types comprising the 3D transform. In C++ 3D transform type is interfaced through a class `trans_type3D`.

trans_type3D constructor

Two constructors are provided for `trans_type3D` (in addition to a copy constructor):

```
trans_type3D::trans_type3D(const gen_trans_type *types_[3]);
trans_type3D::trans_type3D(const int types[3]);
```

Types is an array of 3 1D transform types, either as integer type IDs, or `gen_trans_type` classes.

`trans_type3D` class has the following public members:

```
char *name;
int dtIn,dtOut; // Datatypes for input and output: 1 is real, 2 is complex
int prec; // Datatype precision for a real value in bytes: 4 for single, 8 for double precision

bool is_set;
int types[3]; // 3 1D transform types
```

Transform3D constructor

In C++ 3D transforms are handled through class template `transform3D`, with input and output datatypes `TypeIn` and `TypeOut`. Often these will be the same, however some transforms have different types on input and output, for example real-to-complex FFT. In all cases the floating point precision (single/double) of the initial and final types should match.

```
template<class TypeIn,class TypeOut> class transform3D::transform3D( const grid &grid_in, const grid &grid_out, const trans_type3D *type, const bool inplace, const bool Overwrite);
```

Function: Defines and plans a 3D transform.

Argument	Description
gridIn	Initial `DataGrid` configuration
gridOut	Final `DataGrid` configuration
type	pointer to a 3D transform type class
inplace	`true` is this is an in-place transform; `false` if an out-of-place transform.
Overwrite (optional)	Indicates whether input can be overwritten (`true` = yes, default is no)

Transform3D Execution

```
template<class TypeIn,class TypeOut> class transform3D::exec(TypeIn *In,TypeOut *Out);
```

Function: Executes a 3D transform.

Argument	Description
In, *Out*	Pointers to input and output arrays. In case of in-place transform they can point to the same location. For out-of-place transforms the arrays must be non-overlapping.

Spectral Derivative for 3D array

```
template<class TypeIn,class TypeOut> class transform3D::exec_deriv(TypeIn *In,TypeOut *Out, int idir);
```

Function: Executes 3D real-to-complex FFT, followed by spectral derivative calculation, i.e. multiplication by (ik), where i is the complex imaginary unit, and k is the wavenumber. This function is defined only for complex-valued output arrays (single or double precision), i.e. `TypeOut` must be either `mycomplex` or `complex_double`.

Argument	Description
In, *Out*	Pointers to input and output arrays, assumed to be the local portion of the 3D grid array stored contiguously in memory, consistent with definition of grids in planning stage.
idir	The dimension where derivative is to be taken in (this is logical dimension, NOT storage mapped). Valid values are 0 - 2.

Note:

1) Unless inplace was defined in the planning stage of mytrans, In and Out must be non-overlapping

2) This function can be used multiple times after the 3D transform has been defined and planned.

7.3 P3DFFT++ C Reference

Contents

7.3.1 Setup and Grid Layout

p3dfft_setup

```
void p3dfft_setup()
```

Function: Called once in the beginning of use to initialize P3DFFT++.

p3dfft_cleanup

```
void p3dfft_cleanup()
```

Function: Called once before exit and after the use to free up P3DFFT++ structures.

p3dfft_init_proc_grid

```
int p3dfft_init_proc_grid(int pdims[3],MPI_Comm mpicomm)
```

Function: Initializes a new grid with specified parameters.

Argument	Description
pdims	The dimensions of the 3D processor grid. Value of 1 implies the corresponding dimension is local. These are stored in Fortran (row-major) order, i.e. adjacent MPI tasks are mapped onto the lowest index of the processor grid.
mpicomm	The MPI communicator this processor grid is living on. The library makes it own copy of the communicator, in order to avoid interference with the user program communication.

p3dfft_init_data_grid

```
Grid *p3dfft_init_data_grid(int gdims[3],int dim_conj_sym,int pgrid,int dmap[3],int␣
↪mem_order[3])
```

Function: Initializes a new data grid with specified parameters.

Argument	Description
gdims	Three global grid dimensions (logical order - X, Y, Z).
dim_conj_sym	Dimension of the array in which a little less than half of the elements are omitted due to conjugate symmetry. This argument should be non-negative only for complex-valued arrays resulting from real-to-complex FFT in the given dimension.
pgrid	The processor grid ID, on which this data grid is living on.
dmap	A permutation of the 3 integers: `0`, `1` and `2`. Specifies mapping of data dimensions onto processor grid dimensions. For example, dmap=(1,0,2) implies second data dimension being spanned by the first processor grid dimension, first data dimension being spanned by the second processor grid dimension, and the third data dimension is mapped onto third processor dimension.
mem_order	A permutation of the 3 integers: 0, 1 and 2. Specifies mapping of the logical dimension and memory storage dimensions for local memory for each MPI task. `mem_order[i0] = 0` means that the i0's logical dimension is stored with `stride=1` in memory. Similarly, `mem_order[i1] = 1` means that i1's logical dimension is stored with `stride=ldims[i0]` etc.

Return value: A pointer to the newly initialized `DataGrid` structure that can later be used for grid operations and to get information about the grid.

The `DataGrid` structure is defined as follows:

```
struct Grid_struct {
    int nd; //number of dimensions the volume is split over
    int Gdims[3]; //Global dimensions

    int dim_conj_sym; // Dimension of conjugate symmetry where we store N/2+1 of the␣
↪data after Real-to-complex transform due to conjugate symmety;(-1 for none)

    int MemOrder[3]; //Memory ordering inside the data volume
    int Ldims[3]; //Local dimensions on THIS processor
    int pgrid; //Processor grid
    int dmapr[3]; //Mapping of the data grid dimensions onto processor grid
    int D[3]; //Ranks of Dimensions of physical grid split over rows and columns␣
↪correspondingly
    int L[3]; //Rank of Local dimension (p=1)
    int grid_id[3]; //Position of this pencil/cube in the processor grid
    int ProcDims[3]; // Processor grid dimensions, as mapped onto data dimensions
int GlobStart[3]; // Starting coords of this cube in the global grid
} ;
typedef struct Grid_struct Grid;
```

p3dfft_free_proc_grid

```
void p3dfft_free_proc_grid(int pgrid)
```

Function: Frees up a processor grid, specified by its handle.

Argument	Description
pgrid	Handle of the `ProcGrid` structure to be freed.
p3dfft_free_grid	

void p3dfft_free_data_grid(Grid *gr)

Function: Frees up a data grid.

Argument	Description
gr	pointer to `DataGrid` structure.

7.3.2 One-dimensional transforms

1D transforms can be done with or without data exchange and/or memory reordering. In general, combining a transform with an exchange/reordering can be beneficial for performance due to cache reuse, compared to two separate calls to a transform and an exchange.

The following predefined 1D transforms are available:

Transform	Description
P3DFFT_EMPTY_TYPE	Empty transform.
P3DFFT_R2CFFT_S, P3DFFT_R2CFFT_D	Real-to-complex forward FFT (as defined in FFTW manual), in single and double precision respectively.
P3DFFT_C2RFFT_S, P3DFFT_C2RFFT_D	Complex-to-real backward FFT (as defined in FFTW manual), in single and double precision respectively.
P3DFFT_CFFT_FORWARD_S, P3DFFT_CFFT_FORWARD_D	Complex forward FFT (as defined in FFTW manual), in single and double precision respectively.
P3DFFT_CFFT_BACKWARD_S, P3DFFT_CFFT_BACKWARD_D	Complex backward FFT (as defined in FFTW manual), in single and double precision respectively.
P3DFFT_DCT<x>_REAL_S, P3DFFT_DCT1_REAL_D	Cosine transform for real-numbered data, in single and double precision, where <*x*> stands for the variant of the cosine transform, such as DCT1, DCT2, DCT3, or DCT4.
P3DFFT_DST<x>_REAL_S, P3DFFT_DST1_REAL_D	Sine transform for real-numbered data, in single and double precision, where <*x*> stands for the variant of the cosine transform, such as DST1, DST2, DST3, or DST4.
P3DFFT_DCT<x>_COMPLEX_S, P3DFFT_DCT1_COMPLEX_D	Cosine transform for complex-numbered data, in single and double precision, where <*x*> stands for the variant of the cosine transform, such as DCT1, DCT2, DCT3, or DCT4.
P3DFFT_DST<x>_COMPLEX_S, P3DFFT_DST1_COMPLEX_D	Sine transform for complex-numbered data, in single and double precision, where <*x*> stands for the variant of the cosine transform, such as DST1, DST2, DST3, or DST4.

1D transform planning

```
int p3dfft_plan_1Dtrans(Grid *gridIn, Grid *gridOut, int type1D, int dim, int inplace)
```

Function: Defines and plans a 1D transform of a 3D array. This planning stage must precede execution of 3D transform.

Argument	Description
gridIn, *gridOut*	Pointers to the C equivalent of P3DFFT++ DataGrid object (initial and final)
dim	The logical dimension of the transform (0, 1 or 2). Note that this is the logical dimension rank (0 for X, 1 for Y, 2 for Z), and may not be the same as the storage dimension, which depends on mem_order member of **gridIn** and **gridOut**. The transform dimension of the grid is assumed to be MPI task-local.
inplace	Indicates that this is not an in-place transform (a non-zero argument would indicate in-place).

Return value: The function returns a handle for the transform that can be used in other function calls.

Note: This initialization/planning needs to be done once per transform type.

1D transform execution

```
void p3dfft_exec_1Dtrans_double(int mytrans, double *IN, double *OUT)

void p3dfft_exec_1Dtrans_single(int mytrans, float *IN, float *OUT)
```

Function: Executes double or single precision 1D transform, respectively, of a 3D array

Argument	Description
mytrans	The handle for the 1D transform.
IN, *OUT*	Pointers to one-dimensional input and output arrays containing the 3D grid stored contiguously in memory based on the local grid dimensions and storage order of **gridIn** and **gridOut**.

Note:

1) The execution can be performed many times with the same handle and same or different input and output arrays.
2) In case of out-of-place transform the input and output arrays must be non-overlapping.
3) Both input and output arrays must be local in the dimension of transform

7.3.3 Three-dimensional Transforms

p3dfft_init_3Dtype

```
int p3dfft_init_3Dtype(int type_ids[3])
```

Function: Defines a 3D transform type.

Argument	Description
type_ids	An array of three 1D transform types.

Return value: A handle for 3D transform type.

Example:

```
int type_rcc, type_ids[3];

type_ids[0] = P3DFFT_R2CFFT_D;
type_ids[1] = P3DFFT_CFFT_FORWARD_D;
type_ids[2] = P3DFFT_CFFT_FORWARD_D;

type_rcc = p3dfft_init_3Dtype(type_ids);
```

In this example `type_rcc` will describe the real-to-complex (R2C) 3D transform (R2C in 1D followed by two complex 1D transforms).

3D transform planning

```
int p3dfft_plan_3Dtrans(Grid *gridIn, Grid *gridOut, int type3D, int inplace, int␣
↪overwrite)
```

Function: Plans a 3D transform. This planning stage must precede execution of 3D transform.

Argument	Description
gridIn, *gridOut*	Pointers to initial and final `DataGrid` objects
type3D	The 3D transform type defined as above
inplace	An integer indicating an in-place transform if it's non-zero, out-of-place otherwise.
overwrite (optional)	Non-zero when it is OK to overwrite the input array (optional argument, default is `0`)

Return value: The function returns an integer handle to the 3D transform that can be called multiple times by an execute function.

Note:

1) This initialization/planning needs to be done once per transform type.

2) The final grid may or may not be the same as the initial grid. First, in real-to-complex and complex-to-real transforms the global grid dimensions change for example from (n0,n1,n2) to (n0/2+1,n1,n2), since most applications attempt to save memory by using the conjugate symmetry of the Fourier transform of real data. Secondly, the final grid may have different processor distribution and memory ordering, since for example many applications with convolution and those solving partial differential equations do not need the initial grid configuration in Fourier space. The flow of these applications is typically 1) transform from physical to Fourier space, 2) apply convolution or derivative calculation in Fourier space, and 3) inverse FFT to physical space. Since forward FFT's last step is 1D FFT in the third dimension, it is more efficient to leave this dimension local and stride-1, and since the first step of the inverse FFT is to start with the third dimension 1D FFT, this format naturally fits the algorithm and results in big savings of time due to elimination of several extra transposes.

3D Transform Execution

```
void p3dfft_exec_3Dtrans_single(int mytrans, float *In, float *Out)

void p3dfft_exec_3Dtrans_double(int mytrans, double *In, double *Out)
```

Function: Execute 3D transform in single or double precision, respectively.

Argument	Description
In, *Out*	Pointers to input and output arrays, assumed to be the local portion of the 3D grid array, stored contiguously in memory, consistent with definition of `DataGrid` in planning stage.

Note:

1) Unless `inplace` was defined in the planning stage of `mytrans`, **In** and **Out** must be non-overlapping

2) These functions can be used multiple times after the 3D transform has been defined and planned.

3D Spectral Derivative

```
void p3dfft_exec_3Dtrans_single(int mytrans, float *In, float *Out, int idir)

void p3dfft_exec_3Dtrans_double(int mytrans, double *In, double *Out, int idir)
```

Function: Execute 3D real-to-complex FFT, followed by spectral derivative calculation, i.e. multiplication by (ik), where i is the complex imaginary unit, and k is the wavenumber; in single or double precision, respectively.

Argument	Description
In, *Out*	Pointers to input and output arrays, assumed to be the local portion of the 3D grid array stored contiguously in memory, consistent with definition of `DataGrid` in planning stage.
idir	The dimension where derivative is to be taken in (this is logical dimension, NOT storage mapped). Valid values are `0` - `2`.

Note:

1) Unless `inplace` was defined in the planning stage of `mytrans`, **In** and **Out** must be non-overlapping

2) These functions can be used multiple times after the 3D transform has been defined and planned.

7.4 P3DFFT++ Fortran Reference

Contents

7.4.1 Setup and Grid Layout

In Fortran grid structures are hidden and is operated on by integer handles.

p3dfft_setup

```
subroutine p3dfft_setup
```

Function: Called once in the beginning of use to initialize P3DFFT++.

p3dfft_cleanup

```
subroutine p3dfft_cleanup
```

Function: Called once before exit and after the use to free up P3DFFT++ structures.

p3dfft_init_proc_grid

```
function p3dfft_init_proc_grid(integer pdims(3),integer mpicomm)

integer(C_INT) p3dfft_proc_data_grid
```

Function: Initializes a new processor grid with specified parameters.

Argument	Description
pdims	The dimensions of the 3D processor grid. Value of 1 implies the corresponding dimension is local. These are stored in Fortran (row-major) order, i.e. adjacent MPI tasks are mapped onto the lowest index of the processor grid.
mpicomm	The MPI communicator this processor grid is living on. The library makes it own copy of the communicator, in order to avoid interference with the user program communication.

Return value: An integer handle of the initialized processor grid, to be used later by various routines accessing the grid.

p3dfft_init_data_grid

function p3dfft_init_data_grid(ldims, glob_start, gdims, dim_conj_sym, pgrid, dmap, mem_order)

integer(C_INT) p3dfft_init_data_grid

Function: Initializes a new data grid. Returns handle of the initialized data grid.

Table 1: **IN**

Argument	Description
integer gdims(3)	Three global grid dimensions (logical order - X, Y, Z).
integer dim_conjto-xx	Dimension of the array in which a little less than half of the elements are omitted due to conjugate symmetry. This argument should be non-negative only for complex-valued arrays resulting from real-to-complex FFT in the given dimension.
integer pgrid	Processor grid handle
integer dmap(3)	A permutation of the 3 integers: `0`, `1` and `2`. Specifies the mapping of data grid onto processor grid. For example, dmap=(1,0,2) implies second data dimension being spanned by the first processor grid dimension, first data dimension being spanned by the second processor grid dimension, and the third data dimension is mapped onto third processor dimension.
integer mem_order(3)	A permutation of the 3 integers: 0, 1 and 2. Specifies mapping of the logical dimension and memory storage dimensions for local memory for each MPI task. `mem_order(i0) = 0` means that the i0's logical dimension is stored with `stride=1` in memory. Similarly, `mem_order(i1) = 1` means that i1's logical dimension is stored with `stride=ldims(i0)` etc.

Table 2: **OUT**

Argument	Description
integer ldims(3)	Local dimensions of the grid for each MPI tasks, in order of logical dimensions numbering (XYZ). Essentially ldims = *gdims* / *pgrid*.
integer glob_start(3)	Starting coordinates of the local portion of the grid within the global grid.

Return value: An integer handle of the initialized grid, to be used later by various routines accessing the grid.

p3dfft_free_data_grid

```
subroutine p3dfft_free_data_grid(grid)
```

Function: Frees the data grid specified by its handle.

Table 3: **IN**

Argument	Description
integer(C_INT) grid	The handle of the data grid to be freed.
p3dfft_free_proc_grid	

subroutine p3dfft_free_proc_grid(Pgrid)

Function: Frees the processor grid specified by its handle.

Table 4: **IN**

Argument	Description
integer(C_INT) Pgrid	The handle of the processor grid to be freed.

7.4.2 One-dimensional (1D) Transforms

The following predefined 1D transforms are available:

Transform	Description
P3DFFT_EMPTY_TYPE	Empty transform.
P3DFFT_R2CFFT_S, P3DFFT_R2CFFT_D	Real-to-complex forward FFT (as defined in FFTW manual), in single and double precision respectively.
3DFFT_C2RFFT_S, 3DFFT_C2RFFT_D	Complex-to-real backward FFT (as defined in FFTW manual), in single and double precision respectively.
3DFFT_CFFT_FORWARD_S, 3DFFT_CFFT_FORWARD_D	Complex forward FFT (as defined in FFTW manual), in single and double precision respectively.
3DFFT_CFFT_BACKWARD_S, 3DFFT_CFFT_BACKWARD_D	Complex backward FFT (as defined in FFTW manual), in single and double precision respectively.
3DFFT_DCT<x>_REAL_S, 3DFFT_DCT1_REAL_D	Cosine transform for real-numbered data, in single and double precision, where <*x*> stands for the variant of the cosine transform, such as DCT1, DCT2, DCT3, or DCT4.
P3DFFT_DST<x>_REAL_S, P3DFFT_DST1_REAL_D	Sine transform for real-numbered data, in single and double precision, where <*x*> stands for the variant of the cosine transform, such as DST1, DST2, DST3, or DST4.
P3DFFT_DCT<x>_COMPLEX_S, P3DFFT_DCT1_COMPLEX_D	Cosine transform for complex-numbered data, in single and double precision, where <*x*> stands for the variant of the cosine transform, such as DCT1, DCT2, DCT3, or DCT4.
P3DFFT_DST<x>_COMPLEX_S, P3DFFT_DST1_COMPLEX_D	Sine transform for complex-numbered data, in single and double precision, where <*x*> stands for the variant of the cosine transform, such as DST1, DST2, DST3, or DST4.

1D transform planning

```
function p3dfft_plan_1Dtrans_f(gridIn, gridOut, type, dim, inplace)

integer p3dfft_plan_1Dtrans
```

Function: Defines and plans a 1D transform of a 3D array in a given dimension.

Table 5: **IN**

Argument	Description
integer gridIn	Initial data grid handle.
integer gridOut	Destination data grid handle.
integer type	1D transform type.
integer dim	Dimension rank of the 3D array which should be transformed. valid values are `0`, `1`, or `2`. Note that this is the logical dimension rank (`0` for X, `1` for Y, `2` for Z), and may not be the same as the storage dimension, which depends on `mem_order` member of **gridIn** and **gridOut**. The transform dimension of the grid is assumed to be MPI task-local.
integer inplace	Nonzero value if the transform is in-place.

1D transform execution

```
subroutine p3dfft_exec_1Dtrans_single(mytrans,in,out)

subroutine p3dfft_exec_1Dtrans_double(mytrans,in,out)
```

Function: Executes a 1D transform of a 3D array, in single or double precision.

Table 6: **IN**

Argument	Description
mytrans	The handle of a 1D transform predefined earlier with `p3dfft_plan_1Dtrans`.
in	3D array to be transformed
out	Destination array (can be the same if inplace was nonzero when defining `mytrans`)

Note:

1) If `inplace` was not defined the input and output arrays must be non-overlapping.
2) This transform is done in the dimension specified in `p3dfft_plan_1Dtrans`, and this dimension should be local for both input and output arrays.
3) This subroutine can be called multiple times with the same mytrans and same or different *in*/*out*.

7.4.3 Three-dimensional transforms

3D transform planning

```
function p3dfft_plan_3Dtrans_f(gridIn,gridOut,type,inplace, overwrite)

integer p3dfft_plan_3Dtrans_f
```

Function: Defines and plans a 3D transform.

Argument	Description
integer gridIn	Initial data grid handle
integer gridOut	Destination data grid handle
integer type(3)	Three 1D transform types making up the desired 3D transform
integer inplace	If nonzero, the transform takes place in-place
integer overwrite	Nonzero if the input can be overwritten

Return value: A handle of the 3D transform

Note: The final grid may or may not be the same as the initial grid. First, in real-to-complex and complex-to-real transforms the global grid dimensions change for example from (n0,n1,n2) to (n0/2+1,n1,n2), since most applications attempt to save memory by using the conjugate symmetry of the Fourier transform of real data. Secondly, the final grid may have different processor distribution and memory ordering, since for example many applications with convolution and those solving partial differential equations do not need the initial grid configuration in Fourier space. The flow of these applications is typically 1) transform from physical to Fourier space, 2) apply convolution or derivative calculation in Fourier space, and 3) inverse FFT to physical space. Since forward FFT's last step is 1D FFT in the third dimension, it is more efficient to leave this dimension local and stride-1, and since the first step of the inverse FFT is to start with the third dimension 1D FFT, this format naturally fits the algorithm and results in big savings of time due to elimination of several extra transposes.

3D transform execution

```
subroutine p3dfft_exec_3Dtrans_single(mytrans,in,out)

subroutine p3dfft_exec_3Dtrans_double(mytrans,in,out)
```

Function: Executes a predefined 3D transform in single or double precision.

Argument	Description
mytrans	The handle of the predefined 3D transform
in	Input array
out	Output array

Note: This subroutine can be called multiple times for the same `mytrans` and same or different *in*/*out*.Input and output arrays are local portions of the global 3D array, assumed to be stored contiguously in memory following the definition of the grids in planning stage.

3D Spectral Derivative

```
p3dfft_exec_3Dtrans_single(mytrans, In, Out, idir)

p3dfft_exec_3Dtrans_double(mytrans, In, Out, idir)
```

Function: Execute 3D real-to-complex FFT, followed by spectral derivative calculation, i.e. multiplication by (ik), where i is the complex imaginary unit, and k is the wavenumber; in single or double precision, respectively.

Argument	Description
In, *Out*	Input and output arrays, assumed to be the local portion of the 3D grid array stored contiguously in memory, consistent with definition of `grid` in planning stage.
integer idir	The dimension where derivative is to be taken in (this is logical dimension, NOT storage mapped). Valid values are `0` - `2`.

Note:

1) Unless `inplace` was defined in the planning stage of `mytrans`, **In** and **Out** must be non-overlapping.

2) These functions can be used multiple times after the 3D transform has been defined and planned.

7.5 P3DFFT++ C++ examples

`Makefile.user.C++`	An example Makefile showing how P3DFFT++ should be linked with a user's code
`test1D_r2c.C`	This program exemplifies the use of 1D transforms in P3DFFT++, using cosine 1D transform (DCT-1), for real-valued arrays. 1D transforms are performed on 3D arrays, in the dimension specified as an argument. This could be an isolated 1D transform or a stage in a multidimensional transform. This function can do local transposition, i.e. arbitrary input and output memory ordering. However it does not do an inter-processor transpose (see test_transMPI for that).
`test1D_r2c_cmplx.C`	This program exemplifies the use of 1D transforms in P3DFFT++, using cosine 1D transform (DCT-1), for complex-valued arrays. 1D transforms are performed on 3D arrays, in the dimension specified as an argument. This could be an isolated 1D transform or a stage in a multidimensional transform. This function can do local transposition, i.e. arbitrary input and output memory ordering. However it does not do an inter-processor transpose (see test_transMPI for that).
`test1D_sin.C`	This program exemplifies the use of 1D transforms in P3DFFT++, using cosine 1D transform (DST-1), for real-valued arrays. 1D transforms are performed on 3D arrays, in the dimension specified as an argument. This could be an isolated 1D transform or a stage in a multidimensional transform. This function can do local transposition, i.e. arbitrary input and output memory ordering. However it does not do an inter-processor transpose (see test_transMPI for that).
`test3D_c2c.C`	This program exemplifies the use of P3DFFT++ for 3D complex-to-complex FFT using 2D domain decomposition (1D is a specific case).
`test3D_c2c_inplace.C`	This program exemplifies the use of P3DFFT++ for 3D complex-to-complex using 2D domain decomposition (1D is a specific case). This is an in-place transform example (output overwrites input, which is in the same array).
`test3D_r2c.C`	This program exemplifies the use of P3DFFT++ for 3D real-to-complex and complex-to-real FFT using 2D domain decomposition (1D is a specific case).
`test3D_r2c_sngl.C`	This program exemplifies the use of P3DFFT++ for 3D real-to-complex and complex-to-real FFT using 2D domain decomposition (1D is a specific case). This is a single precision version of test3D_r2c.C example.
`test_deriv.C`	This program exemplifies using P3DFFT++ library for taking a spectral derivative of a 3D array in a given dimension.
`test_trans.C`	This program exemplifies the use of 1D transforms in P3DFFT++, using real-to-complex (R2C) 1D transform. 1D transforms are performed on 3D arrays, in the dimension specified as an argument. This could be an isolated 1D transform or a stage in a multidimensional transform. This function can do local transposition, i.e. arbitrary input and output memory ordering. However it does not do an inter-processor transpose (see test_transMPI for that).

7.6 P3DFFT++ C examples

Makefile_user.C	An example Makefile showing how P3DFFT++ should be linked with a user's code
test1D_c2c.c	This program exemplifies the use of 1D transforms in P3DFFT++, using cosine 1D transform (DCT-1), for real-valued arrays. 1D transforms are performed on 3D arrays, in the dimension specified as an argument. This could be an isolated 1D transform or a stage in a multidimensional transform. This function can do local transposition, i.e. arbitrary input and output memory ordering. However it does not do an inter-processor transpose (see test_transMPI for that).
test1D_r2c.c	This program exemplifies the use of 1D transforms in P3DFFT++, using cosine 1D transform (DCT-1), for complex-valued arrays. 1D transforms are performed on 3D arrays, in the dimension specified as an argument. This could be an isolated 1D transform or a stage in a multidimensional transform. This function can do local transposition, i.e. arbitrary input and output memory ordering. However it does not do an inter-processor transpose (see test_transMPI for that).
test1D_r2c.c	This program exemplifies the use of 1D transforms in P3DFFT++, using real-to-complex (R2C) 1D transform. 1D transforms are performed on 3D arrays, in the dimension specified as an argument. This could be an isolated 1D transform or a stage in a multidimensional transform. This function can do local transposition, i.e. arbitrary input and output memory ordering. However it does not do an inter-processor transpose (see test_transMPI for that).
test3D_c2c.c	This program exemplifies using P3DFFT++ library for 3D complex-to-complex FFT.
test3D_c2c_inplace.c	This program exemplifies using P3DFFT++ library for 3D complex-to-complex FFT, as an in-place transform (output overwrites input, at the same array).
test3D_r2c.c	This program exemplifies using P3DFFT++ library for 3D real-to-complex FFT.
test3D_r2c_overwrite.c	This program exemplifies using P3DFFT++ library for 3D real-to-complex FFT.
test3D_r2c_sp.c	This program exemplifies using P3DFFT++ library for 3D real-to-complex FFT. This is a single precision version of test3D_r2c.c example.
test_deriv.c	This program exemplifies using P3DFFT++ library for taking a spectral derivative in a given dimension.
test2D_r2c.c	This program exemplifies using P3DFFT++ library for 2D Fourier Transform (real-to-complex) on a 3D grid. this implies that one dimension is not transformed.

7.7 P3DFFT++ Fortran examples

`Makefile.user.Fortran`	An example Makefile showing how P3DFFT++ should be linked with a user's code
`test1D_r2c.f90`	This program exemplifies the use of 1D transforms in P3DFFT++, for a 1D cosine transform, for real-valued arrays. 1D transforms are performed on 3D arrays, in the dimension specified as an argument. This could be an isolated 1D transform or a stage in a multidimensional transform. This function can do local transposition, i.e. arbitrary input and output memory ordering. However it does not do an inter-processor transpose (see test_transMPI for that).
`test1D_c2c.f90`	This program exemplifies the use of 1D transforms in P3DFFT++, for a 1D cosine transform, for complex-valued arrays. 1D transforms are performed on 3D arrays, in the dimension specified as an argument. This could be an isolated 1D transform or a stage in a multidimensional transform. This function can do local transposition, i.e. arbitrary input and output memory ordering. However it does not do an inter-processor transpose (see test_transMPI for that).
`test1D_r2c.f90`	This program exemplifies the use of 1D transforms in P3DFFT++, for a 1D real-to-complex FFT. 1D transforms are performed on 3D arrays, in the dimension specified as an argument. This could be an isolated 1D transform or a stage in a multidimensional transform. This function can do local transposition, i.e. arbitrary input and output memory ordering. However it does not do an inter-processor transpose (see test_transMPI for that).
`test3D_c2c.f90`	This sample program illustrates the use of P3DFFT++ library for highly scalable parallel 3D FFT, for a 3D complex FFT.
`test3D_c2c_inplace.f90`	This sample program illustrates the use of P3DFFT++ library for highly scalable parallel 3D FFT, for a 3D complex FFT. This is an in-place version (output overwrites input at the same location).
`test3D_r2c.f90`	This sample program illustrates the use of P3DFFT++ library for highly scalable parallel 3D FFT, for a 3D real-to-complex FFT.
`test3D_r2c_single.f90`	This sample program illustrates the use of P3DFFT++ library for highly scalable parallel 3D FFT, for a 3D real-to-complex FFT. This is a single precision version of test3D_r2c.f90 example.
`test_deriv.f90`	This program exemplifies using P3DFFT++ library for taking a spectral derivative of a 3D array in a given dimension.

CHAPTER 8

Selected publications and presentations

1. Pekurovsky D. P3DFFT: A Framework for Parallel Computations of Fourier Transforms in Three Dimensions. SIAM Journal on Scientific Computing. 2012 January; 34(4):C192-C209. DOI: 10.1137/11082748X PDF
2. Pekurovsky, D. Designing an adaptable framework for highly scalable multidimensional spectral transforms. Presented at SIAM Conference on Parallel Processing for Scientific Computing (PP20), February 12 - 15, 2020, Seattle, Washington, U.S PDF
3. Pekurovsky D. A Portable Framework for Multidimensional Spectral-like Transforms At Scale. Proceedings of the 2021 Improving Scientific Software Conference (No. NCAR/TN567+PROC). 2021 August 12. DOI: 10.26024/p6mv-en77 PDF

CHAPTER 9

Contact

You can reach the main author of P3DFFT Dmitry Pekurovsky at dmitry@sdsc.edu. Also be sure to subscribe to the project mailing list where you can discuss topics of interest with other users and developers and get timely news regarding this library.

CHAPTER 10

License of use

Title P3DFFT++

Authors Dmitry Pekurovsky