P3DFFT - Highly scalable parallel 3D Fast Fourier Transforms library

USER GUIDE

Version 2.4

San Diego Supercomputer Center/UC San Diego

Acknowledgements

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

Feedback

More information about p3dfft can be obtained at http://code.google.com/p/p3dfft/. In addition you may submit tickets regarding issues you've faced with p3dfft. For further questions and comments, please send emails to `dmitry@sdsc.edu`

Table of Contents

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.

**** IMPORTANT:** 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_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.

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**. The following files will be installed: libp3dfft.a -> PREFIX/lib/ p3dfft.h -> PREFIX/include config.h -> PREFIX/include samples -> PREFIX/share By default, configure will install to /usr/local	--prefix= $HOME/local/
--enable-gcc --enable-ibm --enable-intel --enable-pgi	Mandatory	These arguments will prepare `p3dfft` to be built by a specific compiler. You must only choose one option.	--enable-pgi
--enable-fftw --enable-essl	Mandatory	These arguments will prepare `p3dfft` to be used with either the FFTW or ESSL library. You must only choose one option.	--enable-fftw
--with-fftw=*PATH*	Mandatory if --enable-fftw is used.	This argument specifies the path location for the FFTW library; it is mandatory if you are planning to use `p3dfft` with the FFTW library.	--with-fftw=$FFTW
--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, only use 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, only use with --enable-fftw	For search-once-for-the-fast algorithm (takes more time on p3dfft_setup()).	--enable-measure
--enable-patient	Optional, only use 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	--enable-dimsc

		grid according to the C convention. The default is Fortran convention (recommended). This argument does not affect the order of storage of arrays in memory.	
--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-nbly=32
--enable-nbly2	Optional	To define loop blocking factor NBL_Y2	--enable-nbly=32
--enable-nblz	Optional	To define loop blocking factor NBL_Z	--enable-nbly=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>	Optional	Fortran compiler	FC=mpfort
FCFLAGS="<Fortran compiler flags>"	Optional, recommended	Fortran compiler flags	FCFLAGS="-Mextend"
CC=<C compiler>	Optional	C compiler	CC=mpcc
CFLAGS="<C compiler flags>"	Optional, recommended	C compiler flags	CFLAGS="-fastsse"
LDFLAGS="<linker flags>"	Mandatory (depending on platform)	Linker flags	LDFLAGS="-lmpi_f90 -lmpi_f77"

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 such as Kraken, Ranger, and Triton, please refer to the `p3dfft` wiki page at http://code.google.com/p/p3dfft/wiki/Install

After you have run the configure script, you are ready to compile and install p3dfft. Simply run:

```
$ make
$ make install
```

3. `p3dfft` module

The `p3dfft` module declares important variables. It should be included in any code that calls P3DFFT routines (via `use 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 Installation Guide).

4. Initialization

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

Usage: *p3dfft_setup(proc_dims,nx,ny,nz,overwrite)*

Table 4: Arguments of p3dfft_setup

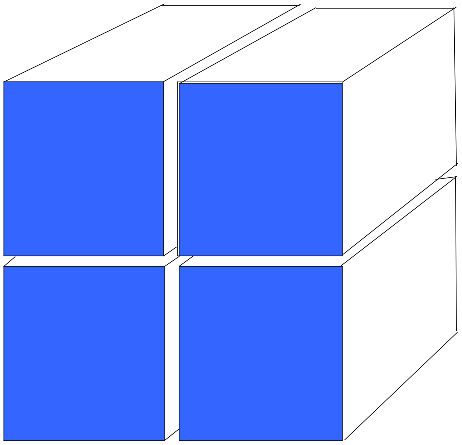
Argument	Intent	Description
proc_dims	Input	An array of two integers, specifying how the processor grid should be decomposed. Either 1D or 2D decomposition can be specified. For example, when running on 12 processors, (4,3) or (2,6) can be specified as proc_dims to indicate a 2D decomposition, or (1,12) can be specified for 1D decomposition. proc_dims values are used to initialize P_1 and P_2.
nx,ny,nz	Input	(Integer) Dimensions of the 3D transform (also the global grid dimensions)
Overwrite	Input	(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.

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 P_1 x P_2, 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).

Figure 1: An example of 2D block(a.k.a. *pencils*) decomposition of a 3D grid.



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.

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
ip	Input	An integer argument specifying one of the two choices for array types: ip=1: "Original": a "physical space" array of real numbers, local in X, distributed among P_1 tasks in Y dimension and P_2 tasks in Z dimension, where P_1 and P_2 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 P_1 tasks in X dimension, P_2 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.

Note: 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 is the grid dimensions are 128 x 256 x 256 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. P3DFFT in this case 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 `–-enable-stride1` argument when calling `configure`. For more information, see performance section below.

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

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

p3dfft_ftran_r2c(IN,OUT)

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 P_1 and P_2 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 P_1 and P_2 tasks respectively.

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

p3dfft_btran_c2r(IN,OUT)

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 P_1 and P_2 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 P_1 and P_2 tasks respectively.

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) * Ny * 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 P_1 tasks in the transposed layout.

8. 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 finding out dimensions of both original and transposed arrays y calling `p3dfft_get_dims` two times with ip=1 and 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.

9. 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.

10. 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 P_1 x P_2 (specified in the first argument to `p3dfft_setup`) performance will generally be better with smaller P_1 (with the product P_1 x P_2 kept constant). Ideally P_1 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.

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.

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.

==

11. 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.