fftMPI Users Manual

6 Mar 2018 version

http://fftmpi.sandia.gov - Sandia National Laboratories

Table of Contents

Manual for the fftMPI library

6 Mar 2018 version

The Parallel FFT library is designed to perform 2d and 3d complex-to-complex Fast Fourier Transforms (FFTs) efficiently on distributed-memory parallel computers using the message passing interface (MPI) library.

The fftMPI library was developed by Steve Plimpton at Sandia National Laboratories with help from several collaborators, listed below. Sandia is a US Department of Energy facility, with funding from the DOE through its Exascale Computing Program (ECP). It is an open-source code, distributed freely under the terms of the Lesser GNU Public License (LGPL). See the LICENSE file in the distribution.

Additional collaborators:

- Axel Kohlmeyer, Temple University
- Paul Coffman, Argonne Leadership Computing Facility
- Phil Blood, Pittsburgh Supercomputing Center

SJP: these links do not exist yet

- fftMPI website = documentation and tarball downloads
- GitHub site = clone or report bugs, pull requests, etc
- Questions, bugs, suggestions: email to sjplimp at sandia.gov or post to GitHub

My thanks to Bruce Hendrickson and Sue Minkoff at Sandia for useful discussions about parallel FFT strategies and to the FFTW authors for the idea of using "plans" as an object-oriented tool for hiding FFT and remap details from the user.

Introduction

The Parallel FFT library was developed at Sandia National Laboratories, a US Department of Energy facility, with funding from the DOE. It is an open-source code, distributed freely under the terms of the Lesser GNU Public License (LGPL). See the LICENSE file in the distribution.

Include some history about original vs current version?

The primary author of this package is Steve Plimpton, who can be contacted at sjplimp at sandia.gov. This download site is where the latest version of this package can be downloaded.

This library is designed for distributed-memory parallel machines using MPI. The library only performs the data movement tasks necessary to compute multi-dimensional FFTs in parallel; the transforms themselves are computed by on-processor 1d FFT routines provided by vendors or the freely available FFTW package. The data remapping routines can also be called directly (independent of the FFTs) to change the layout of 2d or 3d arrays across processors.

The library is written in C++, but a C-style library interface is provided, along with Fortran and Python wrappers. So the library can be called by C++ or C or any language that interfaces with C, e.g. Fortran or Python.

The library should run on any parallel machine that supports MPI. It also works with any size 2d or 3d arrays, so long as the native 1d FFT routines support the array dimensions. It can be run on any number of processors, including in serial. The library allows considerable flexibility in the choice of initial and final data layout of the FFT grid across processors.

Compiling the library

From the src directory, simply type

```
make
```

This should build the library, so long as you have MPI installed and available in your path. The standard Makefile uses mpicxx for compiling and linking. The resulting library file is libfft.a. There are also several Makefile.machine files provided that work on different machines. You can use one of them or created your own edited file and invoke it as

```
make -f Makefile.machine
```

You should only need to edit the "compiler/linker settings" section at the top of the Makefile.

There are 2 options that can be set via the make command, via the "fft" and/or "p" variables:

```
make fft=FFT1d p=PRECISION
```

The "fft" variable determines which 1d FFT library is used. The options for "FFT1d" are FFTW, FFTW3, FFTW2, MKL, or KISS, in either lower- or upper-case. FFTW is the same as FFTW3. KISS is the default, if "fft" is not specified.

If you specify FFTW2 or FFTW3 or MKL the Makefile assumes the directories for their respective include files are in your path. These files are fftw.h or sfftw.h for double/single precision for FFTW2, fftw3.h for FFTW3, and mkl_dfti.h for MKL. The KISS library (kissfft.h) is included in the src dir.

If the appropriate include file is not found, you will need to add a setting for the directory to FFT_INC in the Makefile, something like -I/home/me/include.

The "p" variable determines whether double or single precision FFTs are performed. The options for PRECISION are single or double, in either lower- or upper-case. Double is the default, if "p" is not specified.

Compiling the test programs

The test directory has several programs that illustrate how to use the Parallel FFT library and can also be used to benchmark its performance. These files invoke 3d FFTs.

- test3d.cpp : use the library as a C++ class
- test3d_c.c : use the library thru its C interface
- test3d_f90.f90 : use the library thru its Fortran interface

And likewise for the test2d* files which invoke 2d FFTs.

You can build all the test programs from the test directory, by simply typing

```
make
```

The standard Makefile uses mpicxx for compiling and linking the C++ and C programs, and mpifort for the Fortran 90 programs. If your MPI installation does not include Fortran support, then you will not have an mpifort and the Fortran test programs will not build.

There are also several Makefile.machine files provided that work on different machines. You can use one of them or create your own edited file and invoke it as

```
make -f Makefile.machine
```

The resulting executables are

- C++: test3d and test2d
- C: test3d_c and test2d_c
- F90: test3d_f90 and test2d_f90

You can build an individual test program by including its name as the target, e.g.

```
make test2d
```

As with the library, there are 2 options that can be set via the make command, via the "fft" and/or "p" variables:

```
make fft=FFT1d p=PRECISION
```

IMPORTANT NOTE: You must use the same values for these two variables as were used when building the library in the src directory. For the "fft" variable this is to insure the build of the calling program links to the appropriate 1d FFT library, e.g. FFTW2 or MKL. For the "p" variable this is to insure the single- or double-precision data types used in the calling program for the FFT data match what the library uses. If you do not use the same variable settings, compile-time errors or a run-time error message will be generated. See additional comments in Using the library from your program for how to insure this consistency when using the library with your program.

NOTE: fft also needed when using C++ classes, to find the right include file, so -I settings discussed above also needed??

The Makefile assumes the directories for the respective 1d FFT libraries are in your LD_LIBRARY_PATH, as defined in your environment. These libraries are libfftw.a for FFTWs, libfftw3.a or libfftw3f.a for double/single

precision for FFTW3, libmkl_intel_lp64.a and libmkl_sequential.a and libmkl_core.a for MKL built with the Intel compiler, and libmkl_gf_lp64.a and libmkl_sequential.a and libmkl_core.a for MKL built with the g++ compiler. If the appropriate library is not found, you will need to add a setting for the directory to FFT_PATH in the Makefile, something like -L/home/me/lib.

Running the test programs

The test programs illustrate how to use the Parallel FFT library. They also enable testing and benchmarking of all the options the library supports. These options are specified via command-line arguments, all of which have default values, as indicated below. All of the programs can be launched serially (one processor) or via mpirun on multiple processors.

Here are examples:

```
test3d -g 100 100 100 -n 50
test3d_c -g 100 100 100 -n 50 -m 1
mpirun -np 8 test3d_f90 -g 100 100 100 -n 50
mpirun -np 16 test2d -g 1000 2000 -n 100
mpirun -np 4 test2d_c -g 500 500 -n 1000 -i 492893 -v
test2d_f90 -g 256 256 -n 1000
```

The test3d (C++) and test3d_c (C) programs perform 3d FFTs and use identical command-line arguments, as listed here. The test3d_f90 (F90) program only currently supports the -h, -g, and -n options from this list. All of the 3d programs should produce identical numerical results, and give very similar performance.

```
Syntax: test3d switch args switch args ...
  -h = print help message
  -g Nx Ny Nz = grid size (default = 8 8 8)
  -pin Px Py Pz = proc grid (default = 0 0 0)
     specify 3d grid of procs for initial partition
     0 0 0 = code chooses Px Py Pz, will be bricks
  -pout Px Py Pz = proc grid (default = 0 0 0)
     specify 3d grid of procs for final partition
     0 0 0 = code chooses Px Py Pz
             will be bricks for mode = 0/2
             will be z pencils for mode = 1/3
  -n Niter = iteration count (default = 1)
  -m 0/1/2/3 = FFT mode (default = 0)
     0 = 1 iteration = forward full FFT, backward full FFT
     1 = 1 iteration = forward convolution FFT, backward convolution FFT
     2 = 1 iteration = just forward full FFT
     3 = 1 iteration = just foward convolution FFT
     full FFT returns data to original layout
     forward convolution FFT is brick -> z-pencil
     backward convolution FFT is z-pencil -> brick
  -i zero/step/index/82783 = initialization (default = zero)
     zero = initialize grid with 0.0
     step = initialize with 3d step function
     index = ascending integers 1 to Nx+Ny+Nz
     digits = random number seed
  -c point/all/combo = communication flag (default = point)
     point = point-to-point comm
     all = use MPI_all2all collective
     combo = point for pencil2brick, all2all for pencil2pencil
  -e pencil/brick = exchange flag (default = pencil)
     pencil = pencil to pencil data exchange (4 stages for full FFT)
     brick = brick to pencil data exchange (6 stages for full FFT)
  -p array/ptr/memcpy
     pack/unpack methods for data remapping (default = array)
     array = array based
     ptr = pointer based
     memcpy = memcpy based
  -t summary/details = timing info (default = summary)
```

```
     summary timings or detailed timing breakdown
  -r = remap only, no 1d FFTs
     useful for debugging
  -o output initial/final grid
     only useful for small problems
  -v = verify correctness of answer (default = no -v)
     only possible init = step/digits, and FFT mode 0/1
```

The test2d (C++) and test2d_c (C) programs perform 2d FFTs and use identical command-line arguments, as listed here. The test2d_f90 (F90) program only currently supports the -h, -g, and -n options from this list. All of the 2d programs should produce identical numerical results, and give very similar performance.

```
Syntax: test2d switch args switch args ...
  -h = print help message
  -g Nx Ny = grid size (default = 8 8)
  -pin Px Py = proc grid (default = 0 0)
     specify 2d grid of procs for initial partition
     0 0 = code chooses Px Py, will be bricks
  -pout Px Py = proc grid (default = 0 0)
     specify 2d grid of procs for final partition
     0 0 = code chooses Px Py
           will be bricks for mode = 0/2
           will be y pencils for mode = 1/3
  -n Niter = iteration count (default = 1)
  -m 0/1/2/3 = FFT mode (default = 0)
     0 = 1 iteration = forward full FFT, backward full FFT
     1 = 1 iteration = forward convolution FFT, backward convolution FFT
     2 = 1 iteration = just forward full FFT
     3 = 1 iteration = just foward convolution FFT
     full FFT returns data to original layout
     forward convolution FFT is brick -> y-pencil
     backward convolution FFT is y-pencil -> brick
  -i zero/step/index/82783 = initialization (default = zero)
     zero = initialize grid with 0.0
     step = initialize with 2d step function
     index = ascending integers 1 to Nx+Ny
     digits = random number seed
  -c point/all/combo = communication flag (default = point)
     point = point-to-point comm
     all = use MPI_all2all collective
     combo = point for pencil2brick, all2all for pencil2pencil
  -e pencil/brick = exchange flag (default = pencil)
     pencil = pencil to pencil data exchange (3 stages for full FFT)
     brick = brick to pencil data exchange (4 stages for full FFT)
  -p array/ptr/memcpy
     pack/unpack methods for data remapping (default = array)
     array = array based
     ptr = pointer based
     memcpy = memcpy based
  -t summary/details = timing info (default = summary)
     summary timings or detailed timing breakdown
  -r = remap only, no 1d FFTs
     useful for debugging
  -o output initial/final grid
     only useful for small problems
  -v = verify correctness of answer (default = no -v)
     only possible init = step/digits, and FFT mode 0/1
```

Here is an overview of the different command-line options.

The -g option is for the FFT grid size.

The -pin and -pout options determine how the FFT grid is partitioned across processors before and after the FFT.

The -n option is the iteration count.

The -m option selects between full FFTs versus convolution FFTs. For full FFTs, a forward FFT returns data to its original decomposition. Likewise for an inverse FFT. For convolution FFTS, the data is left in a z-pencil decomposition after a 3d forward FFT or a y-pencil decomposition after a 2d forward FFT. The inverse FFT starts from the z- or y-pencil decomposition and returns the data to its original decomposition.

The -i option determines how the values in the FFT grid are initialized.

The -c option determines whether point2point or all2all communication is performed when the FFT grid data is trasposed between sucessive 1d FFTs.

The -e option is specified as pencil or brick. For 3d FFTs with pencil, there are 4 communication stages for a full FFT: brick -> x-pencil -> y-pencil -> z-pencil -> brick. Or 3 for a 3d convoultion FFT (last stage is skipped). Or 3 for a full 2d FFT (no z-pencil decomposition). Or 2 for a 2d convolution FFT. For 3d FFTs with brick, there are 6 communication stages for a full FFT: brick -> x-pencil -> brick -> y-pencil -> brick -> z-pencil -> brick. Or 5 for a 3d convoultion FFT (last stage is skipped). Or 4 for a full 2d FFT (no z-pencil decomposition). Or 3 for a 2d convolution FFT.

The -p option uses different low-level methods for packing and unpacking FFT data this is sent as messages via MPI.

The -t option determines how much timing output is generated.

The -r option performs no 1d FFTs, it only moves the FFT grid data between processors.

The -o option prints out grid values before and after the FFTs. It should only be used for small grids, else the volume of output can be immense.

The -v option verifies a correct result after a forward and inverse FFT is performed. The results for each grid point should be within epsilon of the starting values.

Data layout

Before giving details on how to use the library, we discuss data layout across processors and within a processor's memory. Consider 2d and 3d arrays which are input to and output by 2d and 3d FFT and remap operations. For parallel efficiency, an application may choose to distribute an array across processors in a variety of ways. The FFT and remap operations allow specification of how an array is mapped to processors on input and also on output. The two mappings can be the same or different. The only requirement for a mapping is that each processor own a contiguous block-shaped subsection of the 2d or 3d array. In 2d this is a rectangular-shaped section of the 2d array; in 3d this is a brick-shaped section of the 3d array. Or in array notation, each processor must own a subsection (ilo:ihi,jlo:jhi) of the global 2d array and similarly for 3d arrays. Note that each array value (FFT grid point) must be uniquely owned by a single processor; the union of all subsections thus tiles the entire array.

On each processor, the array values for its subsection must be stored contiguously in memory with a fast-varying index, a mid-varying index (for 3d), and a slow-varying index. The FFT library does not know or care which indices correspond to which spatial dimensions (x,y,z); it treats the data as effectively a 1d vector and only needs to know which 2d or 3d indices vary fastest or slowest. This means the library can be passed either C-style (last index varies fastest) or Fortran-style (first index varies fastest) arrays. The caller simply specifies the fast, mid, and slow indices in different orders for the two cases. It also means that C-style arrays of pointers to pointers are not allowed as arguments, unless they store the underlying 2d or 3d data contiguous in memory. In which case, the address of the underlying data is passed, not the top-level pointer.

Each array element that is input/output to/from the FFT methods is a complex value. For double-precision FFTs, a complex value must be stored as two contiguous 64-bit doubles (real and imaginary). For single-precision FFTs it is two contiguous 32-bit floats. Thus for double-precision FFTs, if a processor owns 1024 values in its subsection of a 2d or 3d array, it would allocate space for 2048 contiguous doubles to store its values.

The FFT and remap routines allow the option of permuting the order of the fast, mid, and slow indices on output. For example, in a 2d FFT, a processor can own data in row-wise ordering on input to the FFT and in column-wise ordering on output.

It is also permissible for a particular processor to own no data on input and/or output, e.g. if there are more processors than grid points in a particular dimension. In this case the processor subsection is input as (ilo:ihi,jlo:jhi) with ilo > ihi and/or jlo > jhi.

Here are examples of data layouts that the FFT and remap methods allow:

* Each processor initially owns a few rows (or columns) of a 2d or planes of a 3d array and the transformed data is returned in the same layout.

* Each processor initally owns a few rows of a 2d array or planes or pencils of a 3d array. To save communication inside the FFT, it is returned with each processor owning a few columns (2d) or planes or pencils in a different dimension (3d). Then a convolution can be performed by the application, followed by an inverse FFT that returns the data to its original decomposition.

* Each processor initially owns a 2d or 3d subsection of the grid (rectangles or bricks) and the transformed data is returned in the same layout. Or it could be returned in a column-wise or pencil layout as in the previous convolution example.

What is NOT allowed in a data layout is for a procsesor to own a scattered or random set of rows, columns, array subsections, or array values. Such a data distribution might be natural, for example, in a torus-wrap mapping of a

matrix to processors. If this is the case in your application, you will need to write your own remapping method that puts the data in an acceptable layout for input to the FFT or remap operations.

While the FFT routines allow for a wide variety of input and output data layouts, they work fastest with layouts directly usable by the parallel FFTs, without additional remappings being necessary. This is discussed in the optimization section.

Using the library from your program

The discussion in this section and the following C++, C, Fortran sections assumes the program is performing 3d FFTs. Just change 3d to 2d for programs that perform 2d FFTs.

To use the Parallel FFT library from any program (C++, C, Fortran) the following steps are necessary.

- Link the program with the Parallel FFT library in src/libfft.a and also with the 1d FFT library that corresponds to the settings made for 1d FFTs and precision (double vs single) when the FFT library was built. The latter may require a -Ldir setting in the link command.
- Insure that the precision of data allocation for FFT grids matches the precision setting (double vs single) used when the FFT library was built. As discussed above in the Data layout section, this means allocation of doubles for double precision FFTs (2 doubles per complex value) and allocation of floats for single precision FFTs.

To use the Parallel FFT library from a C++ program, the following additional steps are necessary. See the test/test3d.cpp file as an example:

- Include src/fft3d.h in program files which instantiate or call methods from the FFT3d class. The compilation of these files will need to be able to find the include file used by the 1d FFT library specified when the library was built. This may require a -Idir setting in the compile command.
- Add the line "using namespace FFT_NS;" at the top of the same program files.
- See the section Syntax for calling the library from C++ below for specifics of how to make library calls from C++.

To use the Parallel FFT library from a C program, the following additional steps are necessary. See the test/test3d_c.c file as an example:

- Include src/fftlib.h in program files which call methods from the library C-interface.
- Insure the pointer returned from fft3d_create() is stored in an 8-byte variable (void * pointer) by the program. This pointer is passed in all subsequent calls to the library.

To use the Parallel FFT library from a Fortran program, the following additional steps are necessary. See the test/test3d_f90.f90 file as an example:

- Insure the pointer returned from fft3d_create() is stored in an 8-byte variable (integer or floating point) by the program. This pointer is passed in all subsequent calls to the library.

Syntax for calling the library from C++

These are the methods that can be called for the FFT3d class. Those for the FFT2d class are identical, expect for the setup() method, which is described below. See the test/test3d.cpp and test/test2d.cpp files for examples of how all these methods are called.

```
FFT3d(MPI_Comm comm, int precision);
FFT3d(int precision);   // NOTE: need to add this method to lib
~FFT3d();
```

These methods instantiate an instance of the FFT3d class.

Multiple instances can be instantiated by the calling program, e.g. each with its own input/output layout of data. The MPI communicator defines the set of processors which will share the FFT data and perform the parallel FFT.

```
void setup(int nfast, int nmid, int nslow,
           int in_ilo, int in_ihi, int in_jlo    ,
           int in_jhi, int in_klo, int in_khi,
           int out_ilo, int out_ihi, int out_jlo,
           int out_jhi, int out_klo, int out_khi,
           int permute, int *fftsize, int *sendsize, int *recvsize);

void setup(int nfast, int nslow,
           int in_ilo, int in_ihi, int in_jlo, int in_jhi,
           int out_ilo, int out_ihi, int out_jlo, int out_jhi,
           int permute, int *fftsize, int *sendsize, int *recvsize);

void setup_memory(FFT_SCALAR *sendbuf, FFT_SCALAR *recvbuf);
```

These methods are called one time to setup the FFT. The first setup() is for 3d FFTs, the second setup() is for 2d FFTs.

Nfast, nmid, nslow are the dimensions of the global FFT grid. As described in the Data layout section, they refer to how the data is stored contigously by each processor, as the fast-varyind, mid-varying, slow-varying dimension. This is independent of the x,y,z spatial dimensions of the physical problem.

The in/out ijk lo/hi indices are the extent of the subsection of the global grid that this processor owns. Each index can be from 0 to N-1, where N is the global grid dimension. Again, i,j,k correspond to fast,mid,slow, not to x,y,z.

Permute requests a permutation in storage order of fast/mid/slow on output. A vakue of 0 = no permutation. A value of 1 = permute once = mid->fast, slow->mid, fast->slow. A value of 2 = permute twice = slow->fast, fast->mid, mid->slow.

Three value are retured by setup(). Fftsize is the max number of FFT grid points the processor will own at any stage of the FFT (start, intermediate, end). Thus it is the size of the FFT array it must allocate. The returned value N is the # of complex datums the processor owns. Thus the allocation should be 2*N doubles for double-precision FFTs, and 2*N floats for single-precision FFTs.

Sendsize and recvsize are the size of buffers needed to perform the MPI sends and receives for the data remapping operations. If the internal memoryflag value is set to 1 (the default, see description below), the FFT library will allocate these buffers. So the returned values can be ignored. If the internal memoryflag value is set to 0 by the caller, the caller must allocate the memory and pass it to the FFT library via the setup_memory() method. This must be done before any FFT is computed. The returned sendsize and recvsize values are NOT a count of

complex values, but are the # of doubles or floats that the two buffers must be allocated for (double- or single-precision FFTs).

```
void compute(FFT_SCALAR *in, FFT_SCALAR *out, int flag);
```

This method is called as many times as desired to perform forward and inverse FFTs.

Flag is set to 1 for forward FFTs and to -1 for inverse FFTs. In and out are the FFT array data allocated by the caller. If in = out, an in-place FFT is performed. The size of the pointer should be allocated to the fftsize value returned by setup(), even if that is larger than the input or output data size. If in != out, then "in" only need be large enough for the initial data owned by the processor, but "out" should be allocated to the fftsize value returned by setup().

As discussed in the Data layout section, in and out are effectively pointers to 1d vectors of contiguous memory. For double-precision FFTs, FFT_SCALAR is a typedef to a "double". For single-precision FFTs, FFT_SCALAR is a typedef to a "float". So the caller can pass a double * or float * pointer directly.

```
void only_1d_ffts(FFT_SCALAR *in, int flag);
void only_remaps(FFT_SCALAR *in, FFT_SCALAR *out, int flag);
void only_one_remap(FFT_SCALAR *in, FFT_SCALAR *out, int flag, int which);
```

These methods are generally only useful for performance testing or debugging. They perform low-level operations within the FFT. See the test3d.cpp and its timing() method for examples of how they can be used. NOTE: provide some more documentation here??

In addition, both the FFT3d and FFT2d classes have 6 public variables which can be set directly to enable various FFT options. Assume an instance of the FFT3d (or FFT2d) class is named "fft"; the settings can be performed in this manner:

```
fft->collective = 0/1/2 (default = 2), -c point/all/combo
fft->exchange = 0/1 (default = 0), -e pencil/brick
fft->packflag = 0/1/2 (default = 0), -p array/ptr/memcpy
fft->memoryflag = 0/1 (default = 1)
fft->scaled = 0/1 (default = 1)
fft->remaponly = 0/1 (default = 0)
```

The first 4 settings must be made after the class is instantiated, but before the setup() method is called. The last 2 settings can be used after setup(), but before the compute() method.

For the first 3 settings, the possible numeric values correspond to the character strings shown for the -c, -e, -p switch options explained above for the test program arguments. E.g. collective = 0 is the same as -c point, collective = 1 is the same as -c all, and collective = 2 is the same as -c combo.

If memoryflag = 1 is set, then the caller must allocate a sendbuf and recvbuf and pass them to the library via a setup_memory() call before the compute() method is invoked. The required length of these buffers is returned by the setup() method as sendsize and recvsize.

If scaled = 1 is set, then a forward FFT followed by an inverse FFT will return values equal to the initial FFT grid values. If scaled = 0 is set, then each final value will be a factor of N larger than the initial value, where N = total # of points in the 2d or 3d FFT grid.

Syntax for calling the library from C

These are the methods that can be called via the C interface to the FFT3d class. Those for the FFT2d class are identical, expect for the setup() method, as indicated in the list. See the test/test3d_c.c and test/test2d_c.c files for examples of how all these methods are called.

```
void fft3d_create(MPI_Comm, int, void **ptr);
void fft3d_create_no_mpi(int, void **ptr);
void fft3d_destroy(void *ptr);
void fft3d_set(void *ptr, char *keyword, int value);
void fft3d_get(void *ptr, char *keyword, int *value);
void fft3d_setup(void *ptr, int, int, int,
                 int, int, int, int, int, int, int, int, int, int, int, int,
                 int, int *, int *, int *);
void fft2d_setup(void *ptr, int, int,
                 int, int, int, int, int, int, int, int,
                 int, int *, int *, int *);
void fft3d_setup_memory(void *ptr, FFT_SCALAR *, FFT_SCALAR *);
void fft3d_compute(void *ptr, FFT_SCALAR *, FFT_SCALAR *, int);
void fft3d_only_1d_ffts(void *ptr, FFT_SCALAR *, int);
void fft3d_only_remaps(void *ptr, FFT_SCALAR *, FFT_SCALAR *, int);
void fft3d_only_one_remap(void *ptr, FFT_SCALAR *, FFT_SCALAR *, int, int);
```

All these methods correspond one-to-one to the C++ class methods described in the preceeding section. See the discussion there for how the different methods are used and details of their arguments.

Note that the create() methods add a "void **ptr" argument which is returned to the caller when an instance of the FFT3d or FFT2d class is created within the library. All the other methods return this "void *ptr" argument to the library, so the class method can be invoked for the proper class. As with C++ instantiation, the caller can call fft3d_create() multiple times to manage multiple flavors of FFTs, so long as it keeps track of the set of returned pointers.

The only other difference in this list of methods versus the C++ interface are the added fft3d_set() and fft3d_get() methods. These provide the functionality for setting (or querying) the 6 public variables described in the preceeding section. For example, they can be invoked from the caller as follows to set or query various FFT options.

```
fft3d_set(ptr,"collective",2);
fft3d_set(ptr,"exchange",1);
fft3d_set(ptr,"packflag",2);
fft3d_set(ptr,"memoryflag",1);
fft3d_set(ptr,"scaled",0);
fft3d_set(ptr,"remaponly",0);

int exchange;
fft3d_get(ptr,"exchange",&exchange);
```

Syntax for calling the library from Fortran

These are the methods that can be called via the Fortran interface to the FFT3d class. Those for the FFT2d class are identical, expect for the setup() method, as indicated in the list. See the test/test3d_f90.f90 and test/test2d_f90.f90 files for examples of how several of these methods are called.

```
integer world                    // world = MPI_COMM_WORLD (for example)
integer (kind=8) :: fft          // pointer to instance of C++ class
real (kind=8), allocatable :: work(:)      // for double-precision FFTs
real (kind=8), allocatable :: sendbuf(:)
real (kind=8), allocatable :: recvbuf(:)
real (kind=4), allocatable :: work(:)      // for single-precision FFTs
real (kind=4), allocatable :: sendbuf(:)
real (kind=4), allocatable :: recvbuf(:)
integer precision,collective,flag
integer nx,ny,nz,inxlo,inxhi ...    // NOTE: make these var names same as C++
                                    // likewise in test3d_f90.f90
integer permute,fftsize,sendsize,recvsize

call fft3d_create(world,precision,fft)
call fft3d_create(precision,fft)
call fft3d_destroy(fft)

call fft3d_set(fft,"collective",collective);
call fft3d_get(fft,"collective",collective);

call fft3d_setup(fft,nx,ny,nz, &
    inxlo,inxhi,inylo,inyhi,inzlo,inzhi, &
    outxlo,outxhi,outylo,outyhi,outzlo,outzhi, &
    permute,fftsize,sendsize,recvsize)
call fft2d_setup(fft,nx,nz, &
    inxlo,inxhi,inylo,inyhi, &
    outxlo,outxhi,outylo,outyhi, &
    permute,fftsize,sendsize,recvsize)
call fft3d_setup_memory(fft,sendbuf,recvbuf);

call fft3d_compute(fft,work,work,flag)

call fft3d_only_1d_ffts_(fft,work,flag)
call fft3d_only_remaps_(fft,work,work,flag)
call fft3d_only_one_remap_(fft,work,work,flag,which)
```

All these methods correspond one-to-one to the C-interface methods described in the preceeding section, which in turn correspond one-to-one to the C++ class methods described above. See the discussions in those two secions for how the different methods are used and details of their arguments.

Optimization

As noted above, communication cost in the FFTs can be minimized by choosing appropriate input and output data layouts. For both 2d and 3d FFTs an optimal input layout is one where each processor already owns the entire fast-varying dimension of the data array and each processor has (roughly) the same amount of data. In this case, no initial remapping of data is required; the first set of 1d FFTs can be performed immediately.

Similarly, an optimal output layout is one where each proc owns the entire slow-varying dimension and again (roughly) the same amount of data. Additionally it is one where the permutation is specified as 1 for 2d and as 2 for 3d. In this case, no final remapping of data is required; the data is simple left in the layout used for the final set of 1d FFTs.

Note that these input and output layouts may or may not make sense for a specific application. But using either or both of them will reduce the cost of the FFT operation.