fftMPI Users Manual

1 Oct 2018 version

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

Copyright (2018) Sandia Corporation. This software and manual is distributed under the GNU Lesser General Public License.

Table of Contents

Manual for the fftMPI library

1 Oct 2018 version

The fftMPI library is designed to perform parallel 2d or 3d complex-to-complex Fast Fourier Transforms (FFTs) efficiently on distributed-memory machines using MPI (message passing interface library) to move data between processors.

The fftMPI 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 BSD Public License (BSD). See the LICENSE file in the distribution.

The primary developer of the fftMPI library is Steve Plimpton at Sandia National Laboratories, who can be contacted at sjplimp at sandia.gov. Additional collaborators are listed on the fftMPI website.

Additional info is available at these links:

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

IMPORTANT NOTE: the GitHub site will not be stood up until ~Oct 2018.

The fftMPI library replaces the older Parallel FFT Package, which was developed around 1997 for the LAMMPS molecular dynamics code. Both fftMPI and the older package are available from this download site.

My thanks to Bruce Hendrickson and Sue Minkoff at Sandia for useful discussions about parallel FFT strategies when I was creating the original Parallel FFT Package. Thanks also to the FFTW authors for the idea of using "plans" as an object-oriented tool for hiding data remapping and FFT details from the user.

fftMPI documentation

Quick tour

You can quickly try out fftMPI if you

- build the fftMPI library
- build the test programs that use it
- run the test programs (apps)

You can do all 3 steps in one command from the test dir, as follows:

```
% cd fftmpi/test
% sh Run.sh
```

Or you can do it step-by-step (below).

The Run.sh script builds the apps in double precision for all languages (C++, C, Fortran, Python) and uses the provided KISS library for 1d FFTs. If you want to use single precision or another 1d FFT library, see the 3-step process below. If support for some language is not available on your system, those test apps will simply not build.

It then runs each of the apps for one FFT size in all the languages, both in serial and parallel. The apps allow for a variety of command-line args, which are discussed on the runtest doc page.

IMPORTANT NOTE: To run the Python apps, you must enable Python to find the fftMPI shared library and src/fftmpi.py wrapper script. To run the Python apps in parallel, you must have mpi4py installed in your Python. See the usage doc page for details on both these topics.

You can examine the source code of any of the apps in the test dir to see the logic and syntax for using fftMPI from your application. They invoke all the methods provided by fftMPI.

IMPORTANT NOTE: The test3d_f90 and test2d_f90 (F90) apps only currently use a subset of the fftMPI methods.

Step 1

Build fftMPI with one of these make commands. This should produce two lib*.a files, for 2d and 3d FFTs. They will use the default KISS FFT library for 1d FFTs, and operate on double-precision complex data.

```
% cd fftmpi/src
% make
```

The builds will be performed with mpicxx, which uses whatever MPI you have installed on your system, and wraps the C++ compiler installed on your system.

The compile doc page explains how to choose a different 1d FFT library and change to single precision.

Step 2

Build the test apps, written in C++, C, and Fortran 90. This should produce 6 executables: test3d, test2d, test3d_c, test2d_c, test3d_f90, test2d_f90. Test3d and test2d are C++ executables. Two Python test apps, test2d.py and test3d.py, are also in the test dir.

```
% cd fftmpi/test
% make
```

As with step 1, The builds will be performed with mpicxx, which uses whatever MPI you have installed on your system, and wraps the C++, C, or Fortran compilers installed on your system.

The buildtest doc page explains how to choose a different 1d FFT library and change to single precision.

IMPORTANT NOTE: You must use the same values for the "p" and "fft" make variables as were used when building the fftMPI library in the src directory.

Step 3

Run the test programs from the test dir.

See the Run.sh file for examples of how to launch each of the apps in the various languages. See the runtest doc page for a list of command-line arguments recognized by all the apps.

fftMPI documentation

Introduction

The fftMPI library computes multi-dimensional FFTs in parallel where the FFT grid is distributed across processors. Features and limitations of the library are as follows:

- 2d or 3d FFTs
- complex-to-complex FFTs
- single or double precision
- compute FFTs in place, or output to separate memory
- runs on any number of processors, including 1 proc
- allowed grid size in each dimension = any product of 2,3,5 prime factors
- grid decomposition = arbitrary tiling across MPI tasks (explained below)
- initial/final decompositions of grid can be different (useful for convolutions)
- auto-tuning option for a few parameters which affect performance
- 1d FFTs computed by an external library: FFTW, MKL, or KISS
- data movement/reordering methods can be used without FFTs
- invoke multiple instances of the library (e.g. with MPI sub-communicators)
- callable from C++ or any language via a C interface (e.g. C, Fortran, Python)
- test programs and interface files included for all 4 of these languages
- CPU only execution, currently no OpenMP or GPU support

In the fftMPI context, a "tiling" of the 2d or 3d FFT grid is how it is distributed across processors. Imagine a N1 x N2 or N1 x N2 x N3 grid partitioned into P tiles, where P is the number of MPI tasks (processors). Each tile is a "rectangle" of grid points in 2d or "brick" of grid points in 3d. Each processor's tile can be any size or shape, including empty. The P individual tiles cannot overlap; their union is the entire grid. This means each point of the global FFT grid is owned by a unique processor.

A 2d FFT is performed as a set of N2 1d FFTs in the first dimension, followed by N1 1d FFTs in the 2nd dimension. A 3d FFT is performed as N2*N3 1d FFTs in the first dimension, then N1*N3 1d FFTs in the 2nd, then N1*N2 1d FFTs in the third dimension.

The FFT result can overwrite the input values (in-place FFT) or be written to new memory. However note that fftMPI also allocates additional memory internally to buffer MPI send and recv messages.

The 1d FFTs are not computed by fftMPI itself, but rather by calls each processor makes to an external library. As listed above, fftMPI has support for these libraries:

- FFTW (version 3 or 2)
- Intel MKL
- KISS FFT (provided with fftMPI)

What fftMPI encodes is the parallel communication necessary to remap grid data between processors. This involves both sending/receiving data between processors and re-ordering data on-processor, between each stage of 1d FFT computations. This distributes the sets of 1d FFT computations across processors, and stores the data for individual 1d FFTs contiguously on each processor.

fftMPI documentation

Compiling the library

From the src directory, simply type

```
% make
```

This should build fftMPI, so long as you have MPI installed and available in your path. The standard Makefile uses mpicxx for compiling, using whatever C++ compiler it wraps. By default, two static library are created, for 2d and 3d FFTs: libfft2dmpi.a and libfft3dmpi.a.

There are also a few provided Makefile.machine files that work on different supercomputers. You can use one of them or create 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 any Makefile.machine.

By default, fftMPI is built to use the provided KISS FFT library for 1d FFTs. And by default it computes double-precision FFTs (one complex datum = 2 64-bit floating point values). Two make options will change those settings. You can add either or both the "p" and "fft" variables to any make command:

% make p=single # single precision, p=double is default % make fft=fftw # use the FFTW3 lib for 1d FFTs # options: fftw, fftw3, fftw2, mkl, kiss (default) % make lib # build only 2 static libs (2d and 3d) % make shlib fft=mkl # build only 2 shared libs # make all # build both static and shared libs % make help # see syntax for all options

The shared library build will create libfft2dmpi.so and libfft3dmpi.so. The fftw setting is the same as fftw3.

Note that the Makefile assumes the source directory for the 1d FFT library you select is in your path, so that it can find the appropriate include file. 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 file is included in the src dir.

If the appropriate include file is not found, you will need to add a setting for its directory to FFT_INC in the Makefile, something like -I/home/me/pathforFFTW3/include.

You may wish to install both the fftMPI header files and library files in a location where your system can easily find them, e.g. under /usr/local. This can be particulary useful for *.so shared library files, since your LD_LIBRARY_PATH environment variable is often already set to look in this location.

```
cd fftmpi/src
cp fft3d.h fft3d_wrap.h /usr/local/include      # include files, ditto for 2d
cp libfft2dmpi.a libfft2dmpi.so /usr/local/lib  # library files, ditto for 2d
```

Note that you typically need super-user or sudo priveleges to copy files into these dirs.

fftMPI documentation

Building the test programs

The test directory has several programs that illustrate how to use fftMPI. They can also be used to benchmark its performance. These files invoke 3d FFTs.

- test3d.cpp : use fftMPI as a C++ class
- test3d_c.c : use fftMPI thru its C interface
- test3d_f90.f90 : use fftMPI thru its Fortran interface
- test3d.py : use fftMPI thru its Python interface

Likewise for the test2d* files which invoke 2d FFTs.

You can build all the test programs (apps) from the test directory, by typing:

```
% make
```

The resulting executables are

- C++: test3d and test2d
- C: test3d_c and test2d_c
- F90: test3d_f90 and test2d_f90
- Python: test3d.py (no build necessary)

The standard Makefile uses mpicxx for compiling and linking the C++ and C programs, and mpifort for the Fortran 90 programs, using whatever C++ and Fortran compilers they wrap. 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 a few provided Makefile.machine files that work on different supercomputers. You can use one of them or create 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 any Makefile.machine.

As with the fftMPI library build, there are 2 options for precision and a choice of 1d FFT library, that can be set in the make command, via the "p" and "fft" variables. You can set either or both variables in a single make command:

% make p=single # single precision, p=double is default % make fft=fftw # use the FFTW3 lib for 1d FFTs # options: fftw, fftw3, fftw2, mkl, kiss (default) % make test2d p=single # build just the C++ test2d app % make test3d_f90 fft=mkl # build just the Fortran test3d app % make help # see syntax for all options

The fftw setting is the same as fftw3.

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

The Makefile has two variables FFT_INC and FFT_PATH which point to the fftMPI src dir. This is so the test apps can find the fftMPI header files and library files created when you compiled fftMPI. If you use fftMPI from your own program, built in another location, you will need to either install fftMPI in a standard place (e.g. under /usr/local), or you will need settings like for these variables to point to your fftMPI src directory.

The Makefile also assumes the library file for the 1d FFT library you select is in your LD_LIBRARY_PATH, as defined in your environment, so that it can link to the library correctly. These libraries are as follows:

- FFTW2: libfftw.a
- FFTW3 double precision: libfftw3.a
- FFTW3 single precision: libfftw3f.a
- MKL built with Intel compiler: libmkl_intel_lp64.a, libmkl_sequential.a, libmkl_core.a
- MKL built with GNU compiler: libmkl_gf_lp64.a, libmkl_sequential.a, libmkl_core.a
- KISS: only an include file (no library file)

If the appropriate library file is not found, you will need to add a setting for the its directory to FFT_PATH in the Makefile, something like -L/home/me/pathforMKL/lib.

You can also use shared library (*.so) versions of these files. This requires that those files exist on your system (e.g. you built FFTW as a shared library) and that your system can find those files at run time to load them. This requires that your LD_LIBRARY_PATH environment variable includes the path for these 1d FFT libraries. See the usage doc page for details on how to augment the LD_LIBRARY_PATH variable if needed with new directories.

fftMPI documentation

Running the test programs

The test programs (apps) illustrate how to use fftMPI. They also enable testing and benchmarking of the various methods and options that fftMPI supports. These options are specified via command-line arguments to the apps as indicated below. All of the apps can be launched on any number of processors (including a single processor) via mpirun.

Here are example launch commands. Those without a leading mpirun will run on a single processor:

```
% test3d -g 100 100 100 -n 50
% test3d_c -g 100 100 100 -n 50 -m 1 -i step
% mpirun -np 8 test3d_f90 -g 100 100 100 -n 50 -c all
% mpirun -np 16 test2d -g 1000 2000 -n 100 -t details
% mpirun -np 4 test2d_c -g 500 500 -n 1000 -i 492893 -v
% test2d_f90 -g 256 256 -n 1000
% python test3d.py -g 100 100 100 -n 50
% mpirun -np 16 python test3d.py -g 100 100 100 -n 50
```

IMPORTANT NOTE: To run the Python apps, you must enable Python to find the fftMPI shared library and src/fftmpi.py wrapper script. To run the Python apps in parallel, you must have mpi4py installed in your Python.

See the usage doc page for details on both these topics.

Command-line arguments:

All possible command-line arguments are listed here. All the setttings have default values, so you only need to specify those you wish to change. Additional details are explained below. An explanation of the output of the test programs is also given below.

The test3d (C++) and test3d_c (C) and test3d.py (Python) apps perform 3d FFTs and use identical commane-line arguments. The test2d (C++) and test2d_c (C) and test2d.py (Python) apps perform 2d FFTs and take identical arguments except as noted below for -g, -pin, or -pout.

IMPORTANT NOTE: The test3d_f90 and test2d_f90 (F90) apps only currently support the -h, -g, and -n options from this list.

Note that all the 3d apps should produce identical numerical results, and give very similar performance to each other, since the work is being done by the fftMPI library. Ditto for the 2d apps.

Command-line syntax:

```
% test3d switch args switch args ...
% test2d switch args switch args ...

  -h = print help message
  -g Nx Ny Nz = grid size (default = 8 8 8) (no Nz for 2d)
  -pin Px Py Pz = proc grid (default = 0 0 0) (no Pz for 2d)
     specify 3d grid of procs for initial partition
     0 0 0 = code chooses Px Py Pz, will be bricks (rectangles for 2d)
  -pout Px Py Pz = proc grid (default = 0 0 0) (no Pz for 2d)
     specify 3d grid of procs for final partition
     0 0 0 = code chooses Px Py Pz
             will be bricks for mode = 0/2 (rectangles for 2d)
             will be z pencils for mode = 1/3 (y pencils for 2d)
  -n Nloop = iteration count (default = 1)
             can use 0 if -tune enabled, then will be set by tuning operation
  -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 (y-pencil in 2d)
     backward convolution FFT is z-pencil -> brick (y-pencil in 2d)
  -i zero/step/index/82783 = initialization (default = zero)
     zero = initialize grid with 0.0
     step = initialize with 3d step function (2d step function in 2d)
     index = ascending integers 1 to Nx+Ny+Nz (Nx+Ny in 2d)
     digits = random number seed
  -tune nper tmax extra
     nper = # of FFTs per trial run
     tmax = tune within tmax CPU secs, 0.0 = unlimited
     extra = 1 for detailed timing of trial runs, else 0
  -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) (3 for 2d)
     brick = brick to pencil data exchange (6 stages for full FFT) (4 for 2d)
```

```
-p array/ptr/memcpy
   pack/unpack methods for data remapping (default = memcpy)
   array = array based
   ptr = pointer based
   memcpy = memcpy based
-t = provide more timing details (not set by default)
   include timing breakdown, not just summary
-r = remap only, no 1d FFTs (not set by default)
   useful for debugging
-o = output initial/final grid (not set by default)
   only useful for small problems
-v = verify correctness of answer (not set by default)
   only possible for FFT mode = 0/1
```

More details on the command-line arguments:

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 number of iterations (FFTs) to perform. Note that for modes 0 and 1, each iteration will involve 2 FFTs (forward and inverse).

The -m option selects the mode, which chooses between full FFTs versus convolution FFTs and determines what 1 iteration means. 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 -tune option performs an auto-tuning operation before it performs the FFTs. It sets the values that could be otherwise specified by the -c, -e, -p options. So if those options are also specified, they are ignored if -tune is specified. See the library tune API for details on the auto-tuning procedure.

The -c option is specified as point or all or combo. It determines whether point2point or all2all communication is performed when the FFT grid data is moved to new processors between successive 1d FFTs. See the library setup API for details.

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. See the library setup API for details.

The -p option is specified as array or ptr or memcpy. This setting determins what low-level methods are used for packing and unpacking FFT data that is sent as messages via MPI. See the library setup API for details.

The -t option produces more timing output by the test app.

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

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

The -v option verifies a correct result after all the FFTs are complete. FFT is performed. The results for each grid point should be within epsilon of the starting values. It can only be used for modes 0 and 1, when both forward and inverse FFTs are performed.

Output from test apps:

This run on a desktop machine (dual-socket Broadwell CPU):

```
% mpirun -np 16 test3d -g 128 128 128 -n 10 -t -tune 5 10.0 0
```

proudced the following output:

```
3d FFTs with KISS library, precision = double
Grid size: 128 128 128
  initial proc grid: 2 2 4
  x pencil proc grid: 1 4 4
  y pencil proc grid: 4 1 4
  z pencil proc grid: 4 4 1
  3d brick proc grid: 2 2 4
  final proc grid: 2 2 4
Tuning trials & iterations: 9 5
  coll exch pack 3dFFT 1dFFT remap r1 r2 r3 r4: 0 0 2 0.030088 0 0 0 0 0 0
  coll exch pack 3dFFT 1dFFT remap r1 r2 r3 r4: 0 1 2 0.0400121 0 0 0 0 0 0
  coll exch pack 3dFFT 1dFFT remap r1 r2 r3 r4: 1 0 2 0.0360526 0 0 0 0 0 0
  coll exch pack 3dFFT 1dFFT remap r1 r2 r3 r4: 1 1 2 0.0448938 0 0 0 0 0 0
  coll exch pack 3dFFT 1dFFT remap r1 r2 r3 r4: 2 0 2 0.0250025 0 0 0 0 0 0
  coll exch pack 3dFFT 1dFFT remap r1 r2 r3 r4: 2 1 2 0.0238482 0 0 0 0 0 0
  coll exch pack 3dFFT 1dFFT remap r1 r2 r3 r4: 2 1 0 0.0225584 0 0 0 0 0 0
  coll exch pack 3dFFT 1dFFT remap r1 r2 r3 r4: 2 1 1 0.0203406 0 0 0 0 0 0
  coll exch pack 3dFFT 1dFFT remap r1 r2 r3 r4: 2 1 2 0.018153 0 0 0 0 0 0
10 forward and 10 back FFTs on 16 procs
Collective, exchange, pack methods: 2 1 2
Memory usage (per-proc) for FFT grid = 2 MBytes
Memory usage (per-proc) by FFT lib = 3.0008 MBytes
Initialize grid = 0.00136495 secs
FFT setup = 0.000128984 secs
FFT tune = 2.7151 secs
Time for 3d FFTs = 0.35673 secs
  time/fft3d = 0.0178365 secs
  flop rate for 3d FFTs = 11.4977 Gflops
Time for 1d FFTs only = 0.127282 secs
  time/fft1d = 0.0063641 secs
  fraction of time in 1d FFTs = 0.356802
Time for remaps only = 0.219896 secs
  fraction of time in remaps = 0.616422
Time for remap #1 = 0.028487 secs
  fraction of time in remap #1 = 0.0798558
Time for remap #2 = 0.065825 secs
  fraction of time in remap #2 = 0.184523
Time for remap #3 = 0.0849571 secs
  fraction of time in remap #3 = 0.238155
Time for remap #4 = 0.0452759 secs
  fraction of time in remap #4 = 0.126919
```

Annotated output from test apps:

```
3d FFTs with KISS library, precision = double
```

What 1d FFT library was used, also single vs double precision

```
Grid size: 128 128 128
  initial proc grid: 2 2 4
  x pencil proc grid: 1 4 4
  y pencil proc grid: 4 1 4
  z pencil proc grid: 4 4 1
  3d brick proc grid: 2 2 4
  final proc grid: 2 2 4
```

The global FFT grid size and how many processors the grid was partitioned by in each dimension (xyz in this case). The initial and final proc grids are the partitioning for input and output to/from the FFTs. The xyz pencil proc grids are the partitioning for intermediate steps when "-e pencil" is used. The 3d brick proc grid is the intermediate step when "-e brick" is used.

Tuning trials & iterations: 9 5 coll exch pack 3dFFT 1dFFT remap r1 r2 r3 r4: 0 0 2 0.030088 0 0 0 0 0 0 coll exch pack 3dFFT 1dFFT remap r1 r2 r3 r4: 0 1 2 0.0400121 0 0 0 0 0 0 coll exch pack 3dFFT 1dFFT remap r1 r2 r3 r4: 1 0 2 0.0360526 0 0 0 0 0 0 coll exch pack 3dFFT 1dFFT remap r1 r2 r3 r4: 1 1 2 0.0448938 0 0 0 0 0 0 coll exch pack 3dFFT 1dFFT remap r1 r2 r3 r4: 2 0 2 0.0250025 0 0 0 0 0 0 coll exch pack 3dFFT 1dFFT remap r1 r2 r3 r4: 2 1 2 0.0238482 0 0 0 0 0 0 coll exch pack 3dFFT 1dFFT remap r1 r2 r3 r4: 2 1 0 0.0225584 0 0 0 0 0 0 coll exch pack 3dFFT 1dFFT remap r1 r2 r3 r4: 2 1 1 0.0203406 0 0 0 0 0 0 coll exch pack 3dFFT 1dFFT remap r1 r2 r3 r4: 2 1 2 0.018153 0 0 0 0 0 0

This output is only produced when the "-tune" option is used. In this case 9 tuning trials were run, each for 5 iterations (5 or 10 FFTs depending on the -m mode). The "coll exch pack" settings for the 9 runs are listed next. These correspond to the "-c", "-e", "-p" settings described above:

- -c: 0 = point, 1 = all, 2 = combo
- -e: 0 = pencil, 1 = brick
- -p: 0 = array, 1 = ptr, 2 = memcpy

The timings for each trial are listed following the coll/exch/pack settings. The 3dFFT timing is the total time for the trial. In this case the other timings are 0.0 because the -tune extra setting was 0. If it had been 1, then an additional timing breakdown for each trial run would be output.

In this case all 6 permutations of "-c" and "-e" were tried. Then the 3 -p options were used with the fastest of the previous 6 runs. The fastest run of all was with -c = 2 (combo), -e = 1 (brick), and -p = 2 (memcpy).

The number of trials and the number of iterations/trial can be adjusted by the tune() method of fffMPI to limit the tuning to the specified -tune tmax setting.

```
10 forward and 10 back FFTs on 16 procs

Collective, exchange, pack methods: 2 1 2
Memory usage (per-proc) for FFT grid = 2 MBytes
Memory usage (per-proc) by FFT lib = 3.0008 MBytes
```

These are the number of FFTs that were performed for the timing results that follow. They were run with the coll/exch/pack settings shown, which are the optimal settings from the tuning trials in this case. The first memory usage line is for the FFT grid owned by the test app. The second memory usage line is for the MPI send/receive buffers and other data allocated internally for this problem by fftMPI.

```
Initialize grid = 0.00136495 secs
FFT setup = 0.000128984 secs
FFT tune = 2.7151 secs
Time for 3d FFTs = 0.35673 secs
  time/fft3d = 0.0178365 secs
  flop rate for 3d FFTs = 11.4977 Gflops
```

This is the timing breakdown of both the setup (initialize by app, FFT setup by fftMPI), the tuning (if performed), and the FFTs themselves. The time per FFT is also give, as well as the flop rate. There are 5Nlog2(N) flops performed in an FFT, where N is the total number of 2d or 3d grid points and the log is base 2. The flop rate is aggregate across all the processors.

```
Time for 1d FFTs only = 0.127282 secs
  time/fft1d = 0.0063641 secs
  fraction of time in 1d FFTs = 0.356802
Time for remaps only = 0.219896 secs
  fraction of time in remaps = 0.616422
Time for remap #1 = 0.028487 secs
  fraction of time in remap #1 = 0.0798558
Time for remap #2 = 0.065825 secs
  fraction of time in remap #2 = 0.184523
Time for remap #3 = 0.0849571 secs
  fraction of time in remap #3 = 0.238155
Time for remap #4 = 0.0452759 secs
  fraction of time in remap #4 = 0.126919
```

This extra output is only produced if the "-t" option is used. It gives a breakdown of where the total FFT time was spent. The breakdown is roughly: total = 1d FFTs + remaps. In this case 100% = 36% + 62%. And total remap = remap #1 + #2 + #3 + #4. In this case 62% = 8% + 12% + 24% + 13%. The summations are not exact (i.e. they don't sum exactly to 100%), because the breakdown timings are performed by separate runs, and timings are not exactly reproducible from run to run. There can also be load-imbalance effects when full FFTs are timed by themselves, versus individual components being run and timed individually.

The remaps operations include the cost for data movement (from one processor to another) and data reordering (on-processor) operations. In this 3d case, there were 4 flavors of remap. From initial grid to x pencils (#1), from x to y pencils (#2), from y to z pencils (#3), and from z pencils to final grid (same as initial grid). Depending on the choice of mode, some of these remaps may not be performed. Also, for modes 0,1 there are 2 FFTs per iteration, so the per-remap timings are averaged over the communication required in both the forward and inverse directions. E.g. remap #3 would be the average time for y to z pencil for the forward FFT, and z to y pencil for the inverse FFT.

fftMPI documentation

Data layout and optimization

To use fftMPI, an application (app) defines one or more 2d or 3d FFT grids. The data on these grids is owned by the app and distributed across the processors it runs on. To compute an FFT, each processor passes a pointer to the memory which stores its portion of the input grid point values. It also passes a pointer to where it wants the output FFT values stored. The two pointers can be identical to perform an in-place FFT. See the "compute() method API")_api_compute.html for more details.

As explained on the intro doc page, for fftMPI the 2d or 3d FFT grid data is distributed across processors via a "tiling". Imagine a N1 x N2 or N1 x N2 x N3 grid partitioned into P tiles, where P is the number of MPI tasks (processors). Each tile is a "rectangle" of grid points in 2d or "brick" of grid points in 3d. Each processor's tile can be any size or shape, including empty. The P individual tiles cannot overlap; their union is the entire grid. This means each point of the global FFT grid is owned by a unique processor.

The setup() method API has arguments for each processor to specify the global grid size, and the bounds of its tile for input, and also for output. These can be different, which is useful for performing a convolution operation as described below. It also has arguments to specify the ordering of values within each tile for input, and an option to permute the ordering for output.

As a concrete example, assume a global 2d FFT grid is 100x200, and a particular processor owns a 4x8 sub-rectangle of the 2d grid, which could be located anywhere within the global grid. That processor owns data for 4x8 = 32 grid points. Each point is a complex datum, with both a real and imaginary value. For double-precision FFTs, each value is a 64-bit floating point number. For single-precision FFTS, it is a 32-bit floating point number. This processor thus stores 32 complex datums or 64 floating point numbers. For double precision, this would be 8*64 = 512 bytes of grid data.

The processor must store its 64 values contiguously in memory as follows, where R/I are the real/imaginary pair of values for one complex datum:

```
R1, I1, R2, I2, ... R63, I63, R64, I64
```

Conceptually the FFT grid is a 2d matrix, and the values can be ordered one of two ways. With the row-index varying fastest and the column-index slowest, or vice versa:

```
R(1,1), I(1,1), R(2,1), I(2,1), ... R(4,1), I(4,1), R(1,2), I(1,2), ... R(4,8), I(4,8)    # row-inde
R(1,1), I(1,1), R(1,2), I(1,2), ... R(1,8), I(1,8), R(2,1), I(2,1), ... R(4,8), I(4,8)    # column-i
```

The calling app chooses which ordering by specifying which dimension of the 100x200 grid is fast-varying and which is slow-varying. The fftMPI library does NOT know or care which indices correspond to spatial dimensions x or y, but only which is the fast index and which is the slow index. The ordering must be the same on all processors.

The same logic applies to 3d FFTs. There are 6 possible orderings for each processor's input data. One index is specified as fast-varying, a 2nd as mid-varying, and the 3rd as slow-varying. It does NOT matter which indices correspond to spatial dimensions x or y or z.

Note that this means the fftMPI library can be passed either C-style or Python Numpy arrays (last index varies fastest) or Fortran-style arrays (first index varies fastest), so long as the underlying array data is stored contiguously. Here are examples array allocations for a 3d double-precision FFT with nfast = 300, nmid = 200, nslow = 100:

C or C++:

```
double grid1002003002;
double grid100200600;
```

Fortran:

```
real(8), dimension(2,300,200,100) grid
real(8), dimension(600,200,100) grid
```

Python:

```
grid = numpy.zeros(100,200,300,2,np.float64)
grid = numpy.zeros(100,200,600,np.float64)
```

Note that each grid point stores a (real,imaginary) pair of values in consecutive memory locations. So the arrays can be defined as 4d where dim=2 varies fastest, or 3d where the nfast dim=300 is doubled.

Finally, as mentioned above, a permuted ordering can be specified for output of the FFT. For example, for a 2d FFT, all processors can own data in row-wise ordering on input, and in column-wise ordering on output. See the discussion of a convolution operation below.

Here are a few other points worth mentioning:

- C-style arrays of pointers to pointers cannot be passed to fftMPI, unless the underlying data is contiguous in memory. In which case the address of the first datum must be passed to fffMPI. The library treats the data as a 1d vector.
- What is NOT allowed in a data layout is for a procsesor to own a scattered or random set of rows, columns, grid sub-sections, or individual grid points of a 2d or 3d grid. Such a data distribution might be natural, for example, in a torus-wrap mapping of a 2d matrix to processors. If this is the case in your app, you will need to write your own remapping method that puts the data in an acceptable layout for input to fftMPI.
- It's OK 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 in 2d is input as (ilo:ihi,jlo:jhi) with ilo > ihi and/or jlo > jhi. Similarly in 3d.

  Here are examples of data layouts that fftMPI allows:

  - Each processor initially owns a few rows (or columns) of a 2d grid or planes of a 3d grid 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, the output layout is different, 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 after the forward FFT, followed by an inverse FFT that returns the data to its original layout.
  - 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 convolution example of the previous bullet.

Optimization of data layouts

As explained on the intro doc page, a 2d FFT for a N1 x N2 grid is performed as a set of N2 1d FFTs in the first dimension, followed by N1 1d FFTs in the 2nd dimension. A 3d FFT for a N1 x N2 x N3 grid is performed as N2*N3 1d FFTs in the first dimension, then N1*N3 1d FFTs in the 2nd, then N1*N2 1d FFTs in the third dimension.

In the context of the discussion above, this means the 1st set of 1d FFTs is performed in the fast-varying dimension, and the last set of 1d FFTs is performed in the slow-varying dimension. For 3d FFTs, the middle set of 1d FFTs is performed in the mid-varying dimension.

While fftMPI allows for a variety of input and output data layouts, it will run fastest when the input and outputs layout do not require additional data movement before or after performing an FFT.

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 processor 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, so that what was originally the slow-varying dimension is now the fast-varying dimension (for the last set of 1d FFTs). In this case, no final remapping of data is required; the data can be left in the layout used for the final set of 1d FFTs. This is a good way to perform the convolution operation explained above.

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

Using the fftMPI library from your program

The test apps in the test dir are examples of how to use fftMPI from C++, C, Fortran, and Python. The details are summarized here for performing 3d FFTs. Just change "3" to "2" to perform 2d FFTs. You can perform both 2d and 3d FFTs from the same app, by including both the 2d and 3d header files and linking to both the 2d and 3d library files.

Calling fftMPI from your source code

Any file that makes a call to fftMPI needs to include a header file that defines the fftMPI API. These code lines also show how to allocate the FFT grid for single or double precision FFTs.

C++:

```
#include "fft3d.h"
using namespace FFTMPI_NS;

work = (float *) malloc(2*fftsize*sizeof(float));   // single precision
work = (double *) malloc(2*fftsize*sizeof(double)); // double precision
```

The header file fft3d.h defines a typedef for FFT_SCALAR which is set to "float" or "double" depending on the precision you build fftMPI with. So you can define the work vector to be of type FFT_SCALAR if you wish.

C:

```
#include "fft3d_wrap.h"

work = (float *) malloc(2*fftsize*sizeof(float));   // single precision
work = (double *) malloc(2*fftsize*sizeof(double)); // double precision
```

As with C++, you can define the work vector to be of type FFT_SCALAR.

Fortran:

```
use iso_c_binding   ! use these lines in any subroutine that calls fftMPI
use fft3d_wrap

real(4), allocatable, target :: work(:)   ! single precision
real(8), allocatable, target :: work(:)   ! double precision
allocate(work(2*fftsize))
```

Python:

```
import numpy as np
from fftmpi import FFT3dMPI

work = np.zeros(2*fftsize,np.float32)     # single precision
work = np.zeros(2*fftsize,np.float)       # double precision
```

In all these code examples, the fftsize variable is assumed to be set to the total number of FFT grid points owned by a particular processor. As explained on the data layout doc page, the grid data passed to fftMPI from each processor must be a contiguous set of complex values. Thus the allocations are for 2*fftsize values, where 2 is for a "real" value, followed by an "imaginary value".

As a concrete example, assume the global 3d FFT grid is 1024^3, and a particular processor owns a 100x200x50 sub-brick of the 3d grid, which could be located anywhere in the global grid. Then its fftsize = 100*200*50 = 1 million. For double precision FFTs, it would allocate a vector of 2 million 64-bit values. The details for how each processor orders values from a 3d sub-grid as a 1d vector are explained on the data layout doc page.

IMPORTANT NOTE: As explained on the compile doc page, it is a compile-time choice to build fftMPI to perform either single or double precision FFTs. Your application must allocate its FFT grid data to match the library precision setting. You can look at the test apps to see how they do this in a flexible way so that the app can choose to perform its FFTs in either single or double precision.

Building your app with fftMPI

The header files listed above for each language

- C++: fft3d.h
- C: fft3d_wrap.h
- Fortran: fft3d_wrap.f90
- Python: fftmpi.py

are all in the fftMPI src directory. When you compile your app, it must be able to find the appropriate header file.

For C++ and C, the compile and link commands can be something like this:

```
mpicxx -I/home/me/fftmpi/src -c test3d.cpp
mpicxx -L/home/me/fftmpi/src test3d.o -lfft3dmpi -o test3d
```

where the -I and -L switches give the path to the fftMPI src dir.

For Fortran, the fft3d_wrap.f90 file needs to be in the directory with your app files. So you can copy it there, as in the test dir. The compile and link commands can then be something like this:

```
mpif90 -I/home/me/fftmpi/src -c fft3d_wrap.f90
mpif90 -I/home/me/fftmpi/src -c test3d_f90.f90
mpif90 -L/home/me/fftmpi/src test3d_f90.o fft3d_wrap.o -lfft3dmpi -lstdc++ -o test3d_f90
```

where the -I and -L switches give the path to the fftMPI src dir.

For Python, there is no build step. However your Python script needs to be able to find the fftmpi.py file at run time; see the next section.

Note that if using an external 1d FFT library (FFTW or Intel MKL) then the link lines above also need to include the 1d FFT library. See the buildtest doc page for details on how to do this. To use the provided KISS FFT library (just a header file), no additional link arguments are needed.

Running your app with fftMPI

If you build a C++, C, or Fortran app with a static fftMPI library (libfft3dmpi.a file), it should just run.

If you build with a shared fftMPI library, then the system must be able to find the fftMPI library at run time. This can be done in one of two ways.

(1) You can add the fftMPI src dir to your LD_LIBRARY_PATH environment variable, e.g.

```
setenv LD_LIBRARY_PATH $LD_LIBRARY_PATH:/home/me/fftmpi/src   # csh or tcsh
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/me/fftmpi/src   # bash
```

(2) You can "install" the fftMPI library file in a location your system can typically find it, such as /usr/local/lib. See the compile doc page for details on installing fftMPI after you build it. This typically requires super-user or sudo priveleges.

To run a Python app that uses fftMPI, two things must work. If either operation doesn't work, you will get a Python run-time error.

(a) Python finds the src/fftmpi.py file when a statement like this is executed:

```
from fftmpi import FFT3dMPI
```

You can do this by setting the PYTHONPATH environment variable to include the fftMPI src dir, like this:

```
setenv PYTHONPATH $PYTHONPATH:/home/sjplimp/fftmpi/src   # csh or tcsh
export PYTHONPATH=$PYTHONPATH:/home/sjplimp/fftmpi/src   # bash
```

Or from your Python script, you can augment the search path directly:

```
path_fftmpi = "/home/me/fftmpi/src"
sys.path.append(path_fftmpi)
from fftmpi import FFT3dMPI
```

(b) Python finds the fftMPI shared library file (libfft3dmpi.so) when an FFT instance is instantiated, like this:

```
fft = FFT3dMPI(world,precision)
```

You do this the same as explained in (1) or (2) above, either by adding the fftMPI src dir to the LD_LIBRARY_PATH environment variable, or copying the fftMPI shared library into a directory where Python can find it.

Running a Python app in parallel

For a Python app to use fftMPI in parallel, it must pass an MPI communicator to fftMPI. Such an app also typically makes MPI calls in the app itself to work in parallel. To do both of these in Python, use the mpi4py Python package.

To see if mpi4y is already part of your Python type this line:

>>> from mpi4py import MPI

If it works you should be able to run this test.py script like this: mpirun -np 4 python test.py

```
from mpi4py import MPI
world = MPI.COMM_WORLD
me = world.rank
nprocs = world.size
print "Me %d Nprocs %d" % (me,nprocs)
```

and get 4 lines of output "Me N Nprocs 4", where N = 0,1,2,3.

If the import fails, you need to install mpi4py in your Python. Here are two ways to do it.

If you are using anaconda for your Python package management, simply type the following which will download and install it:

```
conda install mpi4py
```

If not, you can download mpi4py from its web site http://www.mpi4py.org, unpack it, and install it via pip:

```
pip install mpi4py
```

Once installed, the test.py example script above should run.

IMPORTANT NOTE: When mpi4py is installed in your Python, it compiles an MPI library (e.g. inside conda) or uses a pre-existing MPI library it finds on your system. This MUST be the same MPI library that fftMPI is built with and links to. If they do not match, you will typically get run-time MPI errors when your app runs.

You can inspect the path to the MPI library that mpi4py uses like this:

```
% python
>>> import mpi4py
>>> mpi4py.get_config()
```

fftMPI documentation

API overview and simple example code

The fftMPI library has 4 classes: FFT3d, FFT2d, Remap3d, Remap2d. The FFT classes perform 3d and 2d FFTs. The Remap classes perform a data remap, which communicates and reorders the data that is distributed 3d and 2d arrays across processors. They can be used if you want to only rearrange data, but not perform an FFT.

The basic way to use either of the FFT classes is to instantiate the class and call setup() once to define how the FFT grid is distributed across processors for both input to and output from the FFT. Then invoke the compute() method as many times as needed to perform forward and/or inverse FFTs. Destruct the class when you are done using it.

Simple example code to do this in C++ is shown below. This code is from the test/simple.cpp file. There are also equivalent files in the test dir for C, Fortran, and Python: simple_c.c, simple_f90.f90, and simple.py.

If a different size FFT or different grid distribution is needed, the FFT classes can be intstantiated as many times as needed. Using the Remap classes is similar, where the remap() method replaces compute().

The choice to operate on single versus double precision data must be made at compile time, as explained on the compile doc page.

These are the methods that can be called for either the FFT3d or FFT2d classes:

- constructor and destructor
- setup() = define grid size and input/output layouts
- setup_memory() = caller provide memory for FFT to use
- set() = set a parameter affecting how FFT is performed
- tune() = auto-tune paramters that affect how FFT is performed
- compute() = compute a single forward or inverse FFT
- only_1d_ffts() = compute just 1d FFTs, no data movement
- only_remaps() = just data movement, no FFTs

- only_one_remap() = just one pass of data movement
- get() = query parameter or timing information

See these apps in the test dir for examples of how all these methods are called from different languages:

- test3d.cpp, test3d_c.cpp, test3d_f90.f90, test3d.py
- test2d.cpp, test2d_c.cpp, test2d_f90.f90, test2d.py

These are methods that can be called for either the Remap3d or Remap2d classes:

- constructor and destructor
- setup() = define grid size and input/output layouts
- set() = set a parameter affecting how Remap is performed
- remap() = perform the the Remap

Simple example code

These files in the text dir of the fftMPI distribution compute a forward/inverse FFT on any number of procs. The size of the FFT is hardcoded at the top of the file. Each file is about 140 lines with comments:

- simple.cpp
- simple_c.c
- simple_f90.f90
- simple.py

You should be able to compile/run any of them as as follows:

cd test make simple mpirun -np 4 simple # run C++ example on 4 procs simple_c # run C example on 1 proc mpirun -np 10 simple_f90 # run Fortran example on 10 procs mpirun -np 6 python simple.py # run Python example on 6 procs

You must link to the fftMPI library built for double-precision FFTs. In the simple source codes, you could replace "3d" by "2d" for 2d FFTs.

The C++ code is in the next section.

```
// Compute a forward/inverse double precision complex FFT using fftMPI
// change FFT size by editing 3 "FFT size" lines
// run on any number of procs

// Run syntax:
// % simple               # run in serial
// % mpirun -np 4 simple  # run in parallel

#include

using namespace FFTMPI_NS;

// FFT size

#define NFAST 128
#define NMID 128
#define NSLOW 128
```

```
int main(int narg, char **args)
{
  // setup MPI

  MPI_Init(&narg,&args);
  MPI_Comm world = MPI_COMM_WORLD;

  int me,nprocs;
  MPI_Comm_size(world,&nprocs);
  MPI_Comm_rank(world,&me);

  // instantiate FFT

  int precision = 2;
  FFT3d *fft = new FFT3d(world,precision);

  // simple algorithm to factor Nprocs into roughly cube roots

  int npfast,npmid,npslow;

  npfast = (int) pow(nprocs,1.0/3.0);
  while (npfast <nprocs) {
    if (nprocs % npfast == 0) break;
    npfast++;
  }
  int npmidslow = nprocs / npfast;
  npmid = (int) sqrt(npmidslow);
  while (npmid  <npmidslow) {
    if (npmidslow % npmid == 0) break;
    npmid++;
  }
  npslow = nprocs / npfast / npmid;

  // partition grid into Npfast x Npmid x Npslow bricks

  int nfast,nmid,nslow;
  int ilo,ihi,jlo,jhi,klo,khi;

  nfast = NFAST;
  nmid = NMID;
  nslow = NSLOW;

  int ipfast = me % npfast;
  int ipmid = (me/npfast) % npmid;
  int ipslow = me / (npfast*npmid);

  ilo = (int) 1.0*ipfast*nfast/npfast;
  ihi = (int) 1.0*(ipfast+1)*nfast/npfast - 1;
  jlo = (int) 1.0*ipmid*nmid/npmid;
  jhi = (int) 1.0*(ipmid+1)*nmid/npmid - 1;
  klo = (int) 1.0*ipslow*nslow/npslow;
  khi = (int) 1.0*(ipslow+1)*nslow/npslow - 1;

  // setup FFT, could replace with tune()

  int fftsize,sendsize,recvsize;
  fft->setup(nfast,nmid,nslow,
             ilo,ihi,jlo,jhi,klo,khi,ilo,ihi,jlo,jhi,klo,khi,
             0,fftsize,sendsize,recvsize);

  // tune FFT, could replace with setup()
```

```
  //fft->tune(nfast,nmid,nslow,
  //          ilo,ihi,jlo,jhi,klo,khi,ilo,ihi,jlo,jhi,klo,khi,
  //          0,fftsize,sendsize,recvsize,0,5,10.0,0);

  // initialize each proc's local grid
  // global initialization is specific to proc count

  FFT_SCALAR *work = (FFT_SCALAR *) malloc(2*fftsize*sizeof(FFT_SCALAR));

  int n = 0;
  for (int k = klo; k <= khi; k++) {
    for (int j = jlo; j <= jhi; j++) {
      for (int i = ilo; i <= ihi; i++) {
        work[n] = (double) n;
        n++;
        work[n] = (double) n;
        n++;
      }
    }
  }

  // perform 2 FFTs

  double timestart = MPI_Wtime();
  fft->compute(work,work,1);        // forward FFT
  fft->compute(work,work,-1);       // inverse FFT
  double timestop = MPI_Wtime();

  if (me == 0) {
    printf("Two %dx%dx%d FFTs on %d procs as %dx%dx%d grid\n",
           nfast,nmid,nslow,nprocs,npfast,npmid,npslow);
    printf("CPU time = %g secs\n",timestop-timestart);
  }

  // find largest difference between initial/final values
  // should be near zero

  n = 0;
  double mydiff = 0.0;
  for (int k = klo; k <= khi; k++) {
    for (int j = jlo; j <= jhi; j++) {
      for (int i = ilo; i <= ihi; i++) {
        if (fabs(work[n]-n) > mydiff) mydiff = fabs(work[n]-n);
        n++;
        if (fabs(work[n]-n) > mydiff) mydiff = fabs(work[n]-n);
        n++;
      }
    }
  }

  double alldiff;
  MPI_Allreduce(&mydiff,&alldiff,1,MPI_DOUBLE,MPI_MAX,world);
  if (me == 0) printf("Max difference in initial/final values = %g\n",alldiff);

  // clean up

  free(work);
  delete fft;
}
```

fftMPI documentation

API for FFT constructor and destructor

These fftMPI methods create and destroy an instance of the FFT3d or FFT2d class. The code examples are for 3d FFTs. Just replace "3d" by "2d" for 2d FFTs.

Multiple instances can be instantiated by the calling program, e.g. if you need to define FFTs with different input or output layouts of data across processors or to run on different subsets of processors. The MPI communicator argument for the constructor defines the set of processors which share the FFT data and perform the parallel FFT.

API:

```
FFT3d(MPI_Comm comm, int precision);     // constructor
~FFT3d();                                // destructor
```

The comm argument in the constructor is an MPI communicator. The precision argument is 1 for single-precision (two 32-bit floating point numbers = 1 complex datum), and 2 for double-precision (two 64-bit floating point numbers = 1 complex datum). The precision is checked by the fftMPI library to insure it was compiled with a matching precision. See the compile doc page for how to compile fftMPI for single versus double precision.

C++:

```
MPI_Comm world = MPI_COMM_WORLD;
int precision = 2;

FFT3d *fft = new FFT3d(world,precision);
delete fft;
```

C:

```
MPI_Comm world = MPI_COMM_WORLD;
int precision = 2;
void *fft;

fft3d_create(world,precision,&fft);
fft3d_destroy(fft);
```

Fortran:

```
integer world,precision
type(c_ptr) :: fft

world = MPI_COMM_WORLD
precision = 2

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

Python:

```
from mpi4py import MPI

world = MPI.COMM_WORLD
precision = 2
```

```
fft = FFT3dMPI(world,precision)
del fft
```

fftMPI documentation

API for FFT setup() and setup_memory()

These fftMPI methods are invoked once to setup an FFT. They define the global grid size, the input/output layouts of data across processors, and various parameters which can affect how the FFT is computed. The code examples at the bottom of the page are for 3d FFTs. Just replace "3d" by "2d" for 2d FFTs. Note that the setup() method has a 3d and 2d version.

An alternative to the setup() method is the tune() method described on the tune API doc page. One or the other method must be invoked before computing actual FFTs, but not both.

API:

```
int collective = 0/1/2 = point/all/combo (default = 2)    // 6 variables
int exchange = 0/1 = pencil/brick (default = 0)
int packflag = array/ptr/memcpy = 0/1/2 (default = 2)
int memoryflag = 0/1 (default = 1)
int scaled = 0/1 (default = 1)
int remaponly = 0/1 (default = 0)

void setup(int nfast, int nmid, int nslow,                 // 3d version
           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,                           // 2d version
           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)
```

The first 6 lines in the API section above are names of public variables within the FFT class which can be set to enable options. All of them have reasonable default settings. So you typically don't need to reset them.

If reset, the first 4 variables must be set before the setup() call. Once setup() is invoked, changing them has no effect.

The last 2 variables can be set (or changed) anytime before the compute() method is called to perform an FFT.

The "collective" variable = 0/1/2 corresponds to 3 algorithmic choices for performing collective communication when FFT grid data moves to new processors between stages of 1d FFTs.

The "point" setting (0) invokes point-to-point MPI_Send() and MPI_Receive() methods between pairs of processor to send/receive data.

The "all" setting (1) invokes the MPI_All2all() method within subsets of processors that need to exchange data.

The "combo" setting (2) is a combination of the other options. It invokes point-to-point MPI methods for pencil-to-brick data movement, and the all2all MPI method for pencil-to-pencil data movement.

The "exchange" variable = 0/1 corresponds to 2 algorithmic choices for how many times FFT grid data moves to new processors between stages of 1d FFTs.

The "pencil" setting (0) moves data once between each pair of 1d FFT stages. For example, assume the fast dimension corresponds to x, and the slow dimension to y. Then to move data bewteen an x-pencil layout to a y-pencil layout, one data remap is performed.

The "brick" setting (1) moves data twice between each pair of 1d FFT stages. For example, to move data bewteen an x-pencil layout to a y-pencil layout, one data remap is performed to go from an x-pencil layout to a 3d brick (or 2d rectangle) layout, and a second data remap to go from brick (rectangle) layout to a y-pencil layout.

The "packflag" variable = 0/1/2 corresponds to 3 algorithmic choices for packing/unpacking FFT grid data into MPI communication buffers.

The "array" setting (0) accesses the local FFT grid data as a 3d (or 2d) array.

The "ptr" setting (1) accesses the local FFT grid data as a 1d vector using pointers.

The "memcpy" setting (2) is similar to the "ptr" setting, except data is copied via a memcpy() function rather than be looping over it one datum at a time.

If the "memoryflag" variable is 1, then fftMPI will allocate memory internally to use for sending/receiving messages. If "memoryflag" is set to 0, then the caller must allocate the memory and pass pointers 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 the "scaled" variable is 1, then a forward FFT followed by an inverse FFT will return values equal to the initial FFT grid values. If the "scaled" variable is 0, then the same operation would produce final values that are a factor of N larger than the initial values, where N = total # of points in the FFT grid (3d or 2d).

The setup() method can only be called once. Only the 3d case is illustrated below for each language; the 2d analogs should be clear.

The nfast, nmid, nslow arguments are the size of the global 3d FFT grid (nfast, nslow for 2d). As explained on the layout doc page, they do NOT refer to dimensions x or y or z in a spatial sense. Rather they refer to the ordering of grid points in the caller's memory for the 3d "brick" of grid points that each processor owns. The points in the nfast dimension are consecutive in memory, points in the nmid dimension are separated by stride nfast in memory, and points in the nslow dimension are separated by stride nfast*nmid.

The "in/out ijk lo/hi" indices define the tile of the 3d or 2d global grid that each processor owns before and after a forward or inverse FFT is computed. See the compute doc page for details Again, i/j/k correspond to fast/mid/slow, NOT to x/y/z.

As explained on the layout doc page, a tile is a brick in 3d or rectangle in 2d. Each index can range from 0 to N-1 inclusive, where N is the corresponding global grid dimension. The lo/hi indices are the first and last point (in that dimension) that the processor owns. If a processor owns no grid point (e.g. on input), then its lo index (in one or more dimensions) should be one larger than its hi index.

IMPORTANT NOTE: When calling fftMPI from Fortran, the index ranges are from 1 to N inclusive, not 0 to N-1.

Here are three examples for 2d FFT grids:

```
in_ilo = 10, in_ihi = 20
in_jlo = 100, in_jhi = 110

in_ilo = 10, in_ihi = 10
in_jlo = 100, in_jhi = 109

in_ilo = 10, in_ihi = 9
in_jlo = 100, in_jhi = 110
```

The first means the processor owns an 11x11 rectangle of grid points. The second means the processor owns a 1x10 rectangle of grid points. The third means the processor owns no grid points.

IMPORTANT NOTE: It is up to the calling app to insure that a valid tiling of the global grid across all processors is passed to fftMPI. As explained on the layout doc page, "valid" means that every grid point is owned by a unique processor and the union of all the tiles is the global grid.

Finally, the permute argument triggers a permutation in storage order of fast/mid/slow for the FFT output. A vakue of 0 means no permutation. A value of 1 means permute once = mid->fast, slow->mid, fast->slow. A value of 2 means permute twice = slow->fast, fast->mid, mid->slow. For 2d FFTs, the only allowed permute values are 0,1. As explained on the layout doc page, this can be useful when performing convolution operations, to avoid extra communication after the FFT is performed.

Three values 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). Note that it is possible for the output size to be larger than the input size, and an intermediate size can be larger than both the input or output sizes.

Thus fftsize is the size of the FFT array the caller should allocate to store its FFT grid points. Note that fftsize is the # of complex datums the processor owns. Thus the caller allocation should be 2*fftsize doubles for double-precision FFTs, and 2*fftsize floats for single-precision FFTs. As explained on the compute doc page, the caller can either perform an FFT in-place (one FFT grid) or allocate separate input and output grids. In the latter case, the output grid should be of size fftsize. The input grid can be exactly the size of the input data (i.e. possibly smaller than fftsize).

The returned sendsize and recvsize are the length of buffers needed to perform the MPI sends and receives for the data remapping operations for a 2d or 3d FFT. If the memoryflag variable is set to 1 (the default, see description above), fftMPI will allocate these buffers. The caller can ignore sendsize and recvsize. If the memoryflag variable is set to 0, the caller must allocate the two buffers of these lenghts and pass them to fftMPI via the setup_memory() method, as explained next.

The setup_memory() method can only be called if the "memorysize" variable is set to 1, in which case it must be called. The caller allocates two buffers (sendbuf and recvbuf) with lengths sendsize and recvsize respectively, and passes them to fftMPI. Sendsize and recvsize are values returned by the setup() method.

The FFT_SCALAR datatype in the setup_memory() API above, is defined by fftMPI to be "double" (64-bit) or "float" (32-bit) for double-precision or single-precision FFTs.

Note that unlike fftsize, sendsize and recvsize are NOT a count of complex values, but are the number of doubles or floats the two buffers must be able to hold, for double- or single-precision FFTs respectively.

C++:

```
int cflag,eflag,pflag,mflag,sflag,rflag;

fft->collective = cflag;
fft->exchange = eflag;
fft->packflag = pflag;
fft->memoryflag = mflag;

fft->scaled = sflag;
fft->remaponly = rflag;

int nfast,nmid,nslow;
int in_ilo,in_ihi,in_jlo,in_jhi,in_klo,in_khi;
int out_ilo,out_ihi,out_jlo,out_jhi,out_klo,out_khi;
int permute,fftsize,sendsize,recvsize;

fft->setup(nfast,nmid,nslow,
           in_ilo,in_ihi,in_jlo,in_jhi,in_klo,in_khi,
           out_ilo,out_ihi,out_jlo,out_jhi,out_klo,out_khi,
           permute,fftsize,sendsize,recvsize);

FFT_SCALAR *sendbuf = (FFT_SCALAR *) malloc(sendsize*sizeof(FFT_SCALAR));
FFT_SCALAR *recvbuf = (FFT_SCALAR *) malloc(recvsize*sizeof(FFT_SCALAR));
fft->setup_memory(sendbuf,recvbuf);
```

The "fft" pointer is created by instantiating an instance of the FFT3d class.

The "in i/j/k lo/hi" indices range from 0 to N-1 inclusive, where N is nfast, nmid, or nslow.

The FFT_SCALAR datatype is defined by fftMPI to be "double" (64-bit) or "float" (32-bit) for double-precision or single-precision FFTs.

C:

```
void *fft;     // set by fft3d_create()
int cflag,eflag,pflag,mflag,sflag,rflag;

fft3d_set(fft,"collective",cflag);
fft3d_set(fft,"exchange",eflag);
fft3d_set(fft,"pack",pflag);
fft3d_set(fft,"memory",mflag);

fft3d_set(fft,"scale",sflag);
fft3d_set(fft,"remaponly",rflag);

int nfast,nmid,nslow;
int in_ilo,in_ihi,in_jlo,in_jhi,in_klo,in_khi;
int out_ilo,out_ihi,out_jlo,out_jhi,out_klo,out_khi;
int permute,fftsize,sendsize,recvsize;

fft3d_setup(fft,nfast,nmid,nslow,
            in_ilo,in_ihi,in_jlo,in_jhi,in_klo,in_khi,
            out_ilo,out_ihi,out_jlo,out_jhi,out_klo,out_khi,
            permute,&fftsize,&sendsize,&recvsize);
```

```
FFT_SCALAR *sendbuf = (FFT_SCALAR *) malloc(sendsize*sizeof(FFT_SCALAR));
FFT_SCALAR *recvbuf = (FFT_SCALAR *) malloc(recvsize*sizeof(FFT_SCALAR));
fft3d_setup_memory(sendbuf,recvbuf);
```

The "in i/j/k lo/hi" indices range from 0 to N-1 inclusive, where N is nfast, nmid, or nslow.

The FFT_SCALAR datatype is defined by fftMPI to be "double" (64-bit) or "float" (32-bit) for double-precision or single-precision FFTs.

Fortran:

```
type(c_ptr) :: fft    ! set by fft3d_create()
integer cflag,eflag,pflag,mflag,sflag,rflag

call fft3d_set(fft,"collective",cflag)
call fft3d_set(fft,"exchange",eflag)
call fft3d_set(fft,"pack",pflag)
call fft3d_set(fft,"memory",mflag)

call fft3d_set(fft,"scale",sflag)
call fft3d_set(fft,"remaponly",rflag)

integer nfast,nmid,nslow
integer in_ilo,in_ihi,in_jlo,in_jhi,in_klo,in_khi
integer out_ilo,out_ihi,out_jlo,out_jhi,out_klo,out_khi
integer permute,fftsize,sendsize,recvsize

call fft3d_setup(fft,nfast,nmid,nslow, &
                 in_ilo,in_ihi,in_jlo,in_jhi,in_klo,in_khi, &
                 out_ilo,out_ihi,out_jlo,out_jhi,out_klo,out_khi, &
                 permute,fftsize,sendsize,recvsize)

real(4), allocatable, target :: sendbuf(:),recvbuf(:)    ! single precision
real(8), allocatable, target :: sendbuf(:),recvbuf(:)    ! double precision
allocate(sendbuf(sendsize))
allocate(sendbuf(recvsize))
fft3d_setup_memory(fft,c_loc(sendbuf),c_loc(recvbuf))
```

The "in i/j/k lo/hi" indices range from 1 to N inclusive, where N is nfast, nmid, or nslow.

Python:

```
cflag = 1
pflag = 0
...

fft.set("collective",cflag)
fft.set("exchange",eflag)
fft.set("pack",pflag)
fft.set("memory",mflag)

fft.set("scale",sflag)
fft.set("remaponly",rflag)

fftsize,sendsize,recvsize =    fft.setup(nfast,nmid,nslow,in_ilo,in_ihi,in_jlo,in_jhi,in_klo,in_khi,
             out_ilo,out_ihi,out_jlo,out_jhi,out_klo,out_khi,permute)

import numpy as np
sendbuf = np.zeros(sendsize,np.float32)   # single precision
```

```
recvbuf = np.zeros(recvsize,np.float32)
sendbuf = np.zeros(sendsize,np.float)     # double precision
recvbuf = np.zeros(sendsize,np.float)
fft.setup_memory(sendbuf,recvbuf)
```

The "fft" object is created by instantiating an instance of the FFT3dMPI class.

The "in i/j/k lo/hi" indices range from 0 to N-1 inclusive, where N is nfast, nmid, or nslow.

fftMPI documentation

API for FFT tune()

This fftMPI method performs auto-tuning of the collective, exchange, and packflag variables listed on the setup dic page to determine which settings produce the fastest FFT. The code examples at the bottom of the page are for 3d FFTs. Just replace "3d" by "2d" for 2d FFTs. Note that the tune() method has a 3d and 2d version.

An alternative to the tune() method is the setup() method described on the setup API doc page. In this case the 3 variables are set explicitly before calling setup(). One or the other method must be invoked before computing actual FFTs, but not both.

API:

```
void tune(int nfast, int nmid, int nslow,             // 3d verion
          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,
          int flag, int niter, double tmax, int tflag);

void tune(int nfast, int nslow,                       // 2d verion
          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,
          int flag, int niter, double tmax, int tflag);
```

The tune() method sets the internal values of the collective, exchange, packflag variables discussed on the setup API doc page by performing a series of trial runs, where one or more FFTs with the global grid size and data layout.

The trial FFTs use dummy FFT grids allocated internally by fftMPI and filled with zeroes. If memory is an issue in your application, it does not need to allocate its own memory for FFT grids until after tune() is complete.

The final values of the 3 variables can be queried after tuning is complete via the methods discussed on the stats API doc page.

All the arguments from nfast to recvsize have the same meaning for tune() as they do for the setup() method, discssed on the setup API doc page. This means all the FFT trials will be run on the same size global grid and with the same initial/final data tilings as the eventual actual FFTs.

If flag is set to 0, pairs of forward and inverse FFTs are performed in the trials. If flag is set to 1, only forward FFTs are performed in the trials. If flag is set to -1, only inverse FFTs are performed in the trials.

The niter argument sets how many FFTs are performed in each trial (or pairs of FFTs for flag = 0).

The tmax argument sets a time limit (in CPU seconds) for how long the tune operation will take. A value of 0.0 means no time limit is enforced.

If tflag is set to 0, only full FFTs will be timed. If tflag is set to 1, 1d FFTs and data remapping operations will also be timed.

Timing data for all operations (2d/3d FFT, optional 1d FFTs only, optional data remappings) for each timing trial can be queried after tuning is complete via the methods discussed on the stats API doc page.

A sequence of 9 trial runs is performed as follows:

- First, a test run of a single forward FFT is performed using the current values of the collective, exchange, packflag variables. Its CPU cost is used to estimate the time needed for all the trials. If this time exceeds tmax, then less trials are performed, as explained below.
- Six trials are performed using all combinations of the collective and exchange variable. I.e. collective = 0,1,2; exchange = 0,1. The fastest of these 6 trials sets the value of collective and exchange.
- Three trials are performed for packflag = 1,2,3, using the optimal collective and exchange values. The fastest of these 3 trials sets the value packflag.

If performing all 9 trials with niter FFTs per trial will take more time than tmax, the following logic is invoked to limit the tuning time:

- Reduce niter to less than the requested value and perform the 9 trials.
- If more time reduction is needed even with niter = 1, only perform the first 6 trials (to set collective, exchange). Set packflag to its default value of 2.
- If more time reduction is needed, only perform 3 trials (to set collective). Set exchange to its default value of 0 and packflag to its default value of 2.
- If more time reduction is needed, only perform 2 trials with collective = 1,2. Set exchange to its default value of 0 and packflag to its default value of 2.
- If more time reduction is needed, and the initial collective, exchange, packflag are variables do not have default values, perform just 1 trial with default values assigned to the 3 variables.
- If no trials can be performed (i.e. the single forward FFT exceeded tmax/2), just leave the collective, exchange, packflag variables set to their initial values, and use the test run results as the first and only trial.

C++:

```
void *fft;     // set by fft3d_create()
int nfast,nmid,nslow;
int in_ilo,in_ihi,in_jlo,in_jhi,in_klo,in_khi;
int out_ilo,out_ihi,out_jlo,out_jhi,out_klo,out_khi;
int permute,fftsize,sendsize,recvsize;
int flag,niter,tflag;
double tmax;

fft->tune(nfast,nmid,nslow,
          in_ilo,in_ihi,in_jlo,in_jhi,in_klo,in_khi,
          out_ilo,out_ihi,out_jlo,out_jhi,out_klo,out_khi,
          permute,fftsize,sendsize,recvsize,
          flag,niter,tmax,tflag);
```

The "fft" pointer is created by instantiating an instance of the FFT3d class.

The "in i/j/k lo/hi" indices range from 0 to N-1 inclusive, where N is nfast, nmid, or nslow.

C:

```
int nfast,nmid,nslow;
int in_ilo,in_ihi,in_jlo,in_jhi,in_klo,in_khi;
int out_ilo,out_ihi,out_jlo,out_jhi,out_klo,out_khi;
int permute,fftsize,sendsize,recvsize;
int flag,niter,tflag;
double tmax;

fft3d_tune(fft,nfast,nmid,nslow,
           in_ilo,in_ihi,in_jlo,in_jhi,in_klo,in_khi,
           out_ilo,out_ihi,out_jlo,out_jhi,out_klo,out_khi,
           permute,&fftsize,&sendsize,&recvsize,
           flag,niter,tmax,tflag);
```

The "in i/j/k lo/hi" indices range from 0 to N-1 inclusive, where N is nfast, nmid, or nslow.

Fortran:

```
type(c_ptr) :: fft    ! set by fft3d_create()
```

integer nfast,nmid,nslow integer in_ilo,in_ihi,in_jlo,in_jhi,in_klo,in_khi integer out_ilo,out_ihi,out_jlo,out_jhi,out_klo,out_khi integer permute,fftsize,sendsize,recvsize inteter flag,niter,tflag real(8) tmax

```
call fft3d_tune(fft,nfast,nmid,nslow, &
                in_ilo,in_ihi,in_jlo,in_jhi,in_klo,in_khi, &
                out_ilo,out_ihi,out_jlo,out_jhi,out_klo,out_khi, &
                permute,fftsize,sendsize,recvsize,
                flag,niter,tmax,tflag)
```

The "in i/j/k lo/hi" indices range from 1 to N inclusive, where N is nfast, nmid, or nslow.

Python:

```
fftsize,sendsize,recvsize =
   fft.tune(nfast,nmid,nslow,in_ilo,in_ihi,in_jlo,in_jhi,in_klo,in_khi,
            out_ilo,out_ihi,out_jlo,out_jhi,out_klo,out_khi,permute,
            flag,niter,tmax,tflag)
```

The "fft" object is created by instantiating an instance of the FFT3dMPI class.

The "in i/j/k lo/hi" indices range from 0 to N-1 inclusive, where N is nfast, nmid, or nslow.

fftMPI documentation

API for FFT compute()

This fftMPI method can be called as many times as desired to perform forward and inverse FFTs for a specific grid size and layout across processors. The code examples are for 3d FFTs. Just replace "3d" by "2d" for 2d FFTs.

API:

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

The FFT_SCALAR datatype is defined by fftMPI to be "double" (64-bit) or "float" (32-bit) for double-precision or single-precision FFTs.

The "in" pointer is the input data to the FFT, stored as a 1d vector of contiguous memory for the FFT grid points this processor owns.

The "out" pointer is the output data from the FFT, also stored as a 1d vector of contiguous memory for the FFT grid points this processor owns.

What is stored in the in and out 1d vectors are "tiles" of 2d or 3d FFT grid data that each processor owns. The extent of the tiles and the way the data is ordered is defined by the setup() method. The layout doc page explains what tiles are and gives more details on how the FFT data is ordered as a 1d vector.

Note that in and out can be the same pointer, in which case the FFT is computed "in place", although there is additional internal memory allocated by fftMPI to migrate data to new processors and reorder it. The setup() method returns a variable "fftsize" which should be used to allocate the necessary size of the in and out vectors. Because this memory is used by fftMPI at intermediate stages of a 2d or 3d FFT, fftsize may be larger than the number of grid points a processor initially owns.

When flag is set to 1, a forward FFT is performed. When flag is set to -1, an inverse FFT is performed.

For a forward FFT, the input data (in pointer) is a "tile" of grid points defined by the "in i/j/k lo/hi" indices specified in the setup() method and ordered by its "nfast, nmid, nslow" arguments. Similarly, the output data (out pointer) is a "tile" of grid points defined by the "out i/j/k lo/hi" indices and ordered by permutation of the "nfast, nmid, nslow" arguments as specified by the permute flag.

For an inverse FFT, it is the opposite. The input data (in pointer) is a "tile" of grid points defined by the "out i/j/k lo/hi" indices with ordering implied by the permute flag. And the output data (out pointer) is a "tile" of grid points defined by the "in i/j/k lo/hi" indices and orderd by the "nfast, nmid, nslow" arguments of the setup() method.

C++:

```
FFT_SCALAR *work;
work = (FFT_SCALAR *) malloc(2*fftsize*sizeof(FFT_SCALAR));

fft->compute(work,work,1);     // forward FFT
fft->compute(work,work,-1);    // inverse FFT
```

The "fft" pointer is created by instantiating an instance of the FFT3d class.

The FFT_SCALAR datatype is defined by fftMPI to be "double" (64-bit) or "float" (32-bit) for double-precision or single-precision FFTs.

C:

```
void *fft;     // set by fft3d_create()
FFT_SCALAR *work;
work = (FFT_SCALAR *) malloc(2*fftsize*sizeof(FFT_SCALAR));

fft3d_compute(fft,work,work,1);    // forward FFT
fft3d_compute(fft,work,work,-1);   // inverse FFT
```

The FFT_SCALAR datatype is defined by fftMPI to be "double" (64-bit) or "float" (32-bit) for double-precision or single-precision FFTs.

Fortran:

```
type(c_ptr) :: fft    ! set by fft3d_create()
real(4), allocatable, target :: work(:)      ! single precision
real(8), allocatable, target :: work(:)      ! double precision
allocate(work(2*fftsize))

call fft3d_compute(fft,c_loc(work),c_loc(work),1)     ! forward FFT
call fft3d_compute(fft,c_loc(work),c_loc(work),-1)    ! inverse FFT
```

Python:

```
import numpy as np
work = np.zeros(2*fftsize,np.float32)    # single precision
work = np.zeros(2*fftsize,np.float)      # double precision

fft.compute(work,work,1)                 # forward FFT
fft.compute(work,work,-1)                # inverse FFT
```

The "fft" object is created by instantiating an instance of the FFT3dMPI class.

fftMPI documentation

API for FFT only_1d_ffts(), only_remaps(), only_one_remap()

These methods are generally only useful for performance testing or debugging. They perform low-level operations within the FFT. See the test/test3d.cpp and its timing() method for examples of how they can be accessed and output.

The code examples at the bottom of the page are for 3d FFTs. Just replace "3d" by "2d" for 2d FFTs. Note that a few of the variables listed in the API section do not exist for 2d FFTs.

API:

```
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);

int collective,exchange,packflag; // 3 values caller can set
int64_t memusage;                 // memory usage in bytes

int npfast1,npfast2,npfast3;      // size of pencil decomp in fast dim
int npmid1,npmid2,npmid3;         // ditto for mid dim
int npslow1,npslow2,npslow3;      // ditto for slow dim
int npbrick1,npbrick2,npbrick3;   // size of brick decomp in 3 dims

int ntrial;                           // # of tuning trial runs
int npertrial;                        // # of FFTs per trial
int cbest,ebest,pbest;                // fastest setting for coll,exch,pack
int cflags[10],eflags[10],pflags[10]; // same 3 settings for each trial
double besttime;                      // fastest single 3d FFT time
double setuptime;                     // setup() time after tuning
double tfft[10];                    // single 3d FFT time for each trial
double t1d[10];                     // 1d FFT time for each trial
```

```
double tremap[10];                    // total remap time for each trial
double tremap1[10],tremap2[10],tremap3[10],tremap4[10];  // per-remap time for each trial

char *fft1d;                 // name of 1d FFT lib
char *precision;             // precision of FFTs, "single" or "double"
```

The 3 "only" methods perform only portions of an FFT, so that they can be timed seperately by the calling app. The "in" and "out" pointers have the same meaning as for the compute() method. The data they point to should be initialized to zero by the caller.

For only_1d_ffts(), 3 sets of 1d ffts (fast, mid, slow) are performed if the 3d case, and 2 sets for 2d (just fast, slow). No data remapping is performed. Only an "in" buffer is passed to this method, since the 1d FFTs are always done in place. Since a processor may own more data at intermediate stages of the FFT than it does initially, the data buffer should be of size "fftsize" and all be initialized to zero. Fftsize is the buffer length returned by the setup() method or tune() method.

For only_remaps(), all the data remappings for the FFT are performed, but no 1d FFTs. The flag value is 1 for a forward FFT and -1 for an inverse FFT, the same as the flag value for the compute() method.

For only_one_remap(), a single data remappings within the FFT is performed (no 1d FFTs). The flag value is 1 for a forward FFT and -1 for an inverse FFT. The "which" argument is one of 1,2,3,4 for a 3d FFT, and one of 1,2,3 for a 2d FFT. For a forward 3d FFT, 1 = initial remap from input layout to x-pencils, 2 = remap from x to y-pencils, 3 = remap from y to z-pencils, 4 = remap from z-pencils to output layout. For an inverse 3d FFT each of the which = 1,2,3,4 is the same, except the remap is in the other direction. E.g. which = 3 is a remap from z to y-pencils. A 2d FFT is the same except there is no y to z-pencils remap for a forward FFT.

The variable lines in the API section above are names of public variables within the FFT class which can be accessed by the caller. Their data types are one of the following: int (32-bit integer), int64_t (64-bit integer), double (64-bit floating point), char * (string), int *, double *. The latter two are vectors of values.

The collective,exchange,packflag values are set by the caller (or defaults) when using the setup() method They are set by fftMPI when using the tune() method. Memusage is the size (on each processor) of the internal memory allocated by fftMPI for send and receive buffers.

The 4 lines of variables that begin with "np" are info about the processor decompositions of the global FFT grid at different stages of the FFT. Fast, mid, slow refer to the x, y, z-pencil decompositions between stages of 1d FFTs. Brick refer to a 3d brick (or 2d rectangle) decomposition which is used when exchange = 1 (brick). See the setup() method doc page for details.

The large set of variables begining is output generated by the tune() method. Refer to its doc page for details on trials and FFTs/trial (npertrial). The various input flags and timing outputs for each trial are stored in vectors.

C++:

```
FFT_SCALAR *work;
work = (FFT_SCALAR *) malloc(2*fftsize*sizeof(FFT_SCALAR));

fft->only_1d_ffts(work,1);
fft->only_remaps(work,work,1);
fft->only_one_remap(work,work,1,3);

printf("3d FFTs with %s library, precision = %s\n",
  fft->fft1d,fft->precision);
```

```
printf("Memory usage (per-proc) by fftMPI = %g MBytes\n",
       (double) fft->memusage / 1024/1024);

printf("Tuning trials & iterations: %d %d\n",fft->ntrial,fft->npertrial);
for (int i = 0; i <fft->ntrial; i++)
  printf("  coll exch pack 3dFFT 1dFFT remap r1 r2 r3 r4: "
         "%d %d %d %g %g %g %g %g %g %g\n",
         fft->cflags[i],fft->eflags[i],fft->pflags[i],
         fft->tfft[i],fft->t1d[i],fft->tremap[i],
         fft->tremap1[i],fft->tremap2[i],
         fft->tremap3[i],fft->tremap4[i]);
```

The "fft" pointer is created by instantiating an instance of the FFT3d class.

The FFT_SCALAR datatype is defined by fftMPI to be "double" (64-bit) or "float" (32-bit) for double-precision or single-precision FFTs.

C:

```
void *fft;     // set by fft3d_create()
FFT_SCALAR *work;
work = (FFT_SCALAR *) malloc(2*fftsize*sizeof(FFT_SCALAR));

fft3d_only_1d_ffts(fft,work,1);
fft3d_only_remaps(fft,work,work,1);
fft3d_only_one_remap(fft,work,work,1,3);

char *fft1d = (char *) fft3d_get(fft,"fft1d");
char *precision = (char *) fft3d_get(fft,"precision");
printf("3d FFTs with %s library, precision = %s\n",fft1d,precision);

double memusage = (double) *((int64_t *) fft3d_get(fft,"memusage")) / 1024/1024;
printf("Memory usage (per-proc) by fftMPI = %g MBytes\n",memusage);

printf("Tuning trials & iterations: %d %d\n",ntrial,npertrial);
int *cflags = (int *) fft3d_get(fft,"cflags");
int *eflags = (int *) fft3d_get(fft,"eflags");
int *pflags = (int *) fft3d_get(fft,"pflags");
double *tfft = (double *) fft3d_get(fft,"tfft");
double *t1d = (double *) fft3d_get(fft,"t1d");
double *tremap = (double *) fft3d_get(fft,"tremap");
double *tremap1 = (double *) fft3d_get(fft,"tremap1");
double *tremap2 = (double *) fft3d_get(fft,"tremap2");
double *tremap3 = (double *) fft3d_get(fft,"tremap3");
for (int i = 0; i <ntrial; i++)
  printf("  coll exch pack 3dFFT 1dFFT remap r1 r2 r3: "
         "%d %d %d %g %g %g %g %g %g\n",
         cflags[i],eflags[i],pflags[i],
         tfft[i],t1d[i],tremap[i],
         tremap1[i],tremap2[i],tremap3[i]);
```

The FFT_SCALAR datatype is defined by fftMPI to be "double" (64-bit) or "float" (32-bit) for double-precision or single-precision FFTs.

Fortran:

```
type(c_ptr) :: fft    ! set by fft3d_create()
real(4), allocatable, target :: work(:)      ! single precision
real(8), allocatable, target :: work(:)      ! double precision
allocate(work(2*fftsize))
```

```
call fft3d_only_1d_ffts(fft,c_loc(work),1)
call fft3d_only_remaps(fft,c_loc(work),c_loc(work),1)
call fft3d_only_one_remap(fft,c_loc(work),c_loc(work),1,3)
```

NOTE: still need to doc how get() functions work from Fortran

---

Python:

```
import numpy as np
work = np.zeros(2*fftsize,np.float32)     # single precision
work = np.zeros(2*fftsize,np.float)       # double precision

fft.only_1d_ffts(work,1)
fft.only_remaps(work,work,1)
fft.only_one_remap(work,work,1,3)

print "3d FFTs with %s library, precision = %s" %
  (fft.get("fft1d",4,1),fft.get("precision",4,1))

print "Memory usage (per-proc) by fftMPI = %g MBytes" %
  (float(fft.get("memusage",3,0)) / 1024/1024)

print "Tuning trials & iterations: %d %d" % (ntrial,npertrial)
for i in range(ntrial):
  print "  coll exch pack 3dFFT 1dFFT remap r1 r2 r3 r4: " +
        "%d %d %d %g %g %g %g %g %g %g" %
        (fft.get("cflags",1,1)[i],fft.get("eflags",1,1)[i],
         fft.get("pflags",1,1)[i],fft.get("tfft",2,1)[i],
         fft.get("t1d",2,1)[i],fft.get("tremap",2,1)[i],
         fft.get("tremap1",2,1)[i],fft.get("tremap2",2,1)[i],
         fft.get("tremap3",2,1)[i],fft.get("tremap4",2,1)[i])
```

The "fft" object is created by instantiating an instance of the FFT3dMPI class.

fftMPI documentation

API for all Remap methods

These methods work with instances of the Remap3d and Remap2d classes. They just perform data remaps of a 3d or 2d array, but not FFTs. The FFT3d and FFT2d classes intstantiate and use their own Remap classes to perform a data remap, which mean to move data to new processors and reorder it. The Remap classes can be used by themselves if your application needs to remap data for its own purposes.

Currently the Remap classes only work with floating point data (32-bit or 64-bit). Each datum in a distributed 3d or 2d grid can be 1 or more floating point values. E.g. the FFT classes use the Remap classes with 2 values (real, imaginary) per grid point. Note that you could remap a 4 or higher dimension grid of floating point values if you are willing to store it as a distributed 3d grid with multiple contiguous values for the higher dimensions. We may add a capability for remapping generalized data types (e.g. ints or structs) in the future.

All of the Remap methods are similar to corresponding FFT methods. The code examples are for 3d Remaps. Just replace "3d" by "2d" for 2d Remaps.

As with the FFT classes, multiple instances of the Remap can be instantiated by the calling program, e.g. if you need to define Remaps wtih different input or output distributions of data across processors. The MPI communicator argument for the constructor defines the set of processors which share the Remap data and perform the parallel Remap.

API:

```
Remap3d(MPI_Comm comm, int precision);     // constructor
~Remap3d();                                // destructor

int collective = 0/1 = point/all (default = 1)             // 3 variables
int packflag = array/ptr/memcpy = 0/1/2 (default = 2)
int memoryflag = 0/1 (default = 1)

void setup(int in_ilo, int in_ihi, int in_jlo,             // 3d version
           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 nqty, int permute, int memoryflag,
           int &sendsize, int &recvsize)

void setup(int in_ilo, int in_ihi, int in_jlo, int in_jhi,     // 2d version
           int out_ilo, int out_ihi, int out_jlo, int out_jhi,
           int nqty, int permute, int memoryflag,
           int &sendsize, int &recvsize)

void remap(FFT_SCALAR *in, FFT_SCALAR *out, FFT_SCALAR sendbuf, FFT_SCALAR recvbuf);
```

The Remap3d() and ~Remap3d() methods create and destroy an instance of the Remap3d class.

The comm argument is an MPI communicator. The precision argument is 1 for single-precision (32-bit floating point numbers) and 2 for double-precision (64-bit floating point numbers). The precision is checked by the fftMPI library to insure it was compiled with a matching precision. See the compile doc page for how to compile fftMPI for single versus double precision.

The "collective, packflag, memoryflag" lines are public variables within the Remap class which can be set to enable option. All of them have reasonable default settings. So you typically don't need to reset them. If reset, they must be set before the setup() call. Once setup() is invoked, changing them has no effect.

The meaning of the variables is exactly the same as for the FFT classes. See the setup API doc page for an explanation. Note that for remaps, the collective variable has only two settings (0,1). There is no collective = 2 option like there is for FFTs.

The setup() method can only be called once. Only the 3d case is illustrated below for each language; the 2d analogs should be clear.

The meaning of the "in/out ijk lo/hi" indices is exactly the same as for the FFT setup() method, as explained on the setup API doc page (with examples). Note that unlike for FFTs, the sizes of each dimension of the global 3d (nfast,nmid,nslow) or 2d grid (nfast,nslow) are not arguments to the Remap setup() method. However the "in/out ijk lo/hi" indices define the same tiles of the 3d or 2d global grid that each processor owns before and after the remap operation.

As explained on the layout doc page, a tile is a brick in 3d or rectangle in 2d. Each index can range from 0 to N-1 inclusive, where N is the corresponding global grid dimension. The lo/hi indices are the first and last point (in that dimension) that the processor owns. If a processor owns no grid point (e.g. on input), then its lo index (in one or more dimensions) should be one larger than its hi index.

IMPORTANT NOTE: When calling the Remap classes from Fortran, the index ranges are from 1 to N inclusive, not 0 to N-1.

As also explained on the layout doc page, the "in/out ijk lo/hi" indices do NOT refer to dimensions x or y or z in a spatial sense. Rather they refer to the ordering of grid points in the caller's memory for the 3d "brick" of grid points that each processor owns. The points in the nfast dimension are consecutive in memory, points in the nmid dimension are separated by stride nfast in memory, and points in the nslow dimension are separated by stride nfast*nmid. For remaps, the nfast, nmid, nslow global grid size (or nfast, nslow in 2d) are not input arguments, but they are implied by the input and output tilings.

The "nqty" argument is the number of floating point values per grid point. The FFT classes call the Remap classes with nqty=2 for a complex value (real, imaginary) per grid point.

IMPORTANT NOTE: It is up to the calling app to insure that a valid tiling of the global grid across all processors is passed to fftMPI. As explained on the layout doc page, "valid" means that every grid point is owned by a unique processor and the union of all the tiles is the global grid.

Finally, the permute argument triggers a permutation in storage order of fast/mid/slow for the remap output, the same as for FFT output for the FFT setup() method. A value of 0 means no permutation. A value of 1 means permute once = mid->fast, slow->mid, fast->slow. A value of 2 means permute twice = slow->fast, fast->mid, mid->slow. For 2d remaps, the only allowed permute values are 0,1.

The returned sendsize and recvsize are the length of buffers needed to perform the MPI sends and receives for the data remapping operation. If the memoryflag variable is set to 1 (the default), fftMPI will allocate these buffers. The caller can ignore sendsize and recvsize. If the memoryflag variable is set to 0, the caller must allocate the two buffers of these lengths and pass them as arguments to the remap() method, as explained next.

For the remap() method, The FFT_SCALAR datatype is defined by fftMPI to be "double" (64-bit) or "float" (32-bit) for double-precision or single-precision FFTs.

The "in" pointer is the input data to the remap, stored as a 1d vector of contiguous memory for the grid points this processor owns.

The "out" pointer is the output data from the remap, also stored as a 1d vector of contiguous memory for the grid points this processor owns.

C++:

```
#include "remap3d.h"
using namespace FFTMPI_NS;

MPI_Comm world = MPI_COMM_WORLD;
int precision = 2;

Remap3d *remap = new Remap3d(world,precision);
delete remap;

int cflag,pflag,mflag;
int in_ilo,in_ihi,in_jlo,in_jhi,in_klo,in_khi;
int out_ilo,out_ihi,out_jlo,out_jhi,out_klo,out_khi;
int nqty,permute,memoryflag,sendsize,recvsize;

remap->collective = cflag;
remap->packflag = pflag;
remap->memoryflag = mflag;

remap->setup(nfast,nmid,nslow,
```

```
              in_ilo,in_ihi,in_jlo,in_jhi,in_klo,in_khi,
              out_ilo,out_ihi,out_jlo,out_jhi,out_klo,out_khi,
              nqty,permute,memoryflag,sendsize,recvsize);

int insize = (in_ihi-in_ilo+1) * (in_jhi-in_jlo+1) * (in_khi-in_klo+1);
int outsize = (out_ihi-out_ilo+1) * (out_jhi-out_jlo+1) * (out_khi-out_klo+1);
int remapsize = (insize > outsize) ? insize : outsize;
FFT_SCALAR *work = (FFT_SCALAR *) malloc(remapsize*sizeof(FFT_SCALAR));
FFT_SCALAR *sendbuf = (FFT_SCALAR *) malloc(sendsize*sizeof(FFT_SCALAR));
FFT_SCALAR *recvbuf = (FFT_SCALAR *) malloc(recvsize*sizeof(FFT_SCALAR));

remap->remap(work,work,sendbuf,recvbuf);
```

The "in i/j/k lo/hi" indices range from 0 to N-1 inclusive, where N is nfast, nmid, or nslow.

The FFT_SCALAR datatype is defined by fftMPI to be "double" (64-bit) or "float" (32-bit) for double-precision or single-precision FFTs.

C:

```
#include "fft3d_wrap.h"

MPI_Comm world = MPI_COMM_WORLD;
int precision = 2;

void *remap;
remap3d_create(world,precision,&remap);
remap3d_destroy(remap);

int cflag,pflag,mflag;
int in_ilo,in_ihi,in_jlo,in_jhi,in_klo,in_khi;
int out_ilo,out_ihi,out_jlo,out_jhi,out_klo,out_khi;
int nqty,permute,memoryflag,sendsize,recvsize;

remap3d_set(remap,"collective",cflag);
remap3d_set(remap,"pack",pflag);
remap3d_set(remap,"memory",mflag);

remap3d_setup(nfast,nmid,nslow,
              in_ilo,in_ihi,in_jlo,in_jhi,in_klo,in_khi,
              out_ilo,out_ihi,out_jlo,out_jhi,out_klo,out_khi,
              nqty,permute,memoryflag,&sendsize,&recvsize);

int insize = (in_ihi-in_ilo+1) * (in_jhi-in_jlo+1) * (in_khi-in_klo+1);
int outsize = (out_ihi-out_ilo+1) * (out_jhi-out_jlo+1) * (out_khi-out_klo+1);
int remapsize = (insize > outsize) ? insize : outsize;
FFT_SCALAR *work = (FFT_SCALAR *) malloc(remapsize*sizeof(FFT_SCALAR));
FFT_SCALAR *sendbuf = (FFT_SCALAR *) malloc(sendsize*sizeof(FFT_SCALAR));
FFT_SCALAR *recvbuf = (FFT_SCALAR *) malloc(recvsize*sizeof(FFT_SCALAR));

remap3d_remap(remap,work,work,sendbuf,recvbuf);
```

The "in i/j/k lo/hi" indices range from 0 to N-1 inclusive, where N is nfast, nmid, or nslow.

The FFT_SCALAR datatype is defined by fftMPI to be "double" (64-bit) or "float" (32-bit) for double-precision or single-precision FFTs.

Fortran:

```
include 'mpif.h'
use iso_c_binding
use fft3d_wrap

integer world,precision
type(c_ptr) :: remap

world = MPI_COMM_WORLD
precision = 2

call remap3d_create(world,precision,remap)
call remap3d_destroy(remap)

integer cflag,pflag,mflag
integer in_ilo,in_ihi,in_jlo,in_jhi,in_klo,in_khi
integer out_ilo,out_ihi,out_jlo,out_jhi,out_klo,out_khi
integer nqty,permute,memoryflag,sendsize,recvsize

call remap3d_set(remap,"collective",cflag)
call remap3d_set(remap,"pack",pflag)
call remap3d_set(remap,"memory",mflag)

call remap3d_setup(remap,nfast,nmid,nslow, &
                   in_ilo,in_ihi,in_jlo,in_jhi,in_klo,in_khi, &
                   out_ilo,out_ihi,out_jlo,out_jhi,out_klo,out_khi, &
                   nqty,permute,memoryflag,sendsize,recvsize)

integer insize,outsize,remapsize
real(4), allocatable, target :: work(:),sendbuf(:),recvbuf(:)  ! single precision
real(8), allocatable, target :: work(:),sendbuf(:),recvbuf(:)  ! double precision
insize = (in_ihi-in_ilo+1) * (in_jhi-in_jlo+1) * (in_khi-in_klo+1)
outsize = (out_ihi-out_ilo+1) * (out_jhi-out_jlo+1) * (out_khi-out_klo+1)
remapsize = max(insize,outsize)
allocate(work(remapsize))
allocate(sendbuf(sendsize))
allocate(sendbuf(recvsize))

call remap3d_remap(remap,c_loc(work),c_loc(work),c_loc(sendbuf),c_loc(recvbuf))
```

The "in i/j/k lo/hi" indices range from 1 to N inclusive, where N is nfast, nmid, or nslow.

Python:

```
import numpy as np
from fftmpi import Remap3dMPI
from mpi4py import MPI

world = MPI.COMM_WORLD
precision = 2

remap = Remap3dMPI(world,precision)
del remap

cflag = 1
pflag = 0
...

remap.set("collective",cflag)
remap.set("pack",pflag)
remap.set("memory",mflag)
```

```
sendsize,recvsize =    remap.setup(nfast,nmid,nslow,in_ilo,in_ihi,in_jlo,in_jhi,in_klo,in_khi,
               out_ilo,out_ihi,out_jlo,out_jhi,out_klo,out_khi,
               nqty,permute,memoryflag)

insize = (in_ihi-in_ilo+1) * (in_jhi-in_jlo+1) * (in_khi-in_klo+1)
outsize = (out_ihi-out_ilo+1) * (out_jhi-out_jlo+1) * (out_khi-out_klo+1)
remapsize = max(insize,outsize)
work = np.zeros(remapsize,np.float32)     # single precision
sendbuf = np.zeros(sendsize,np.float32)
recvbuf = np.zeros(recvsize,np.float32)
work = np.zeros(remapsize,np.float)       # double precision
sendbuf = np.zeros(sendsize,np.float)
recvbuf = np.zeros(sendsize,np.float)

remap.remap(work,work,sendbuf,recvbuf)
```

The "in i/j/k lo/hi" indices range from 0 to N-1 inclusive, where N is nfast, nmid, or nslow.