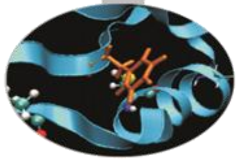
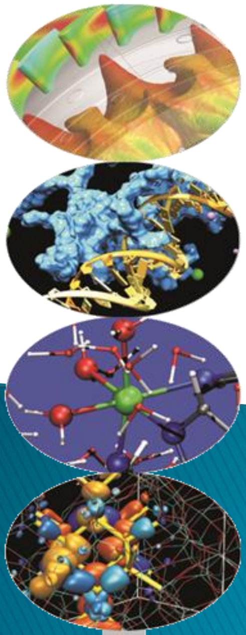


Parallel Fast Fourier Transforms

Theory, Methods and Libraries.
A small introduction.



HPC Numerical Libraries
10-12 March 2014
CINECA – Casalecchio di Reno (BO)

Massimiliano Guarrasi
m.guarrasi@cineca.it

Agenda

- Introduction to F.T.:
  - Theorems
  - DFT
  - FFT
- Parallel Domain Decomposition
  - Slab Decomposition
  - Pencil Decomposition
- Some Numerical libraries:
  - FFTW
    - Some useful commands
    - Some Examples
  - 2Decomp&FFT
    - Some useful commands
    - Example
  - P3DFFT
    - Some useful commands
    - Example
  - Performance results

Introduction to Fourier Transforms

Fourier Transforms

$$H(f) = \int_{-\infty}^{\infty} h(t) e^{2\pi i f t} dt$$

$$h(t) = \int_{-\infty}^{\infty} H(f) e^{-2\pi i f t} df$$

Frequency Domain ⟺ **Time Domain**

Real Space ⟺ **Reciprocal Space**

Some useful Theorems

Convolution Theorem

$$g(t) * h(t) \Leftrightarrow G(f) \cdot H(f)$$

Correlation Theorem

$$Corr(g,h) \equiv \int_{-\infty}^{+\infty} g(\tau) h(t+\tau) d\tau \Leftrightarrow G(f) H^{*}(f)$$

Power Spectrum

$$Corr(h,h) \equiv \int_{-\infty}^{+\infty} h(\tau) h(t+\tau) d\tau \Leftrightarrow |H(f)|^{2}$$

Discrete Fourier Transform (DFT)

In many application contexts the Fourier transform is approximated with a Discrete Fourier Transform (DFT):

$$H(f_n) = \int_{-\infty}^{\infty} h(t) e^{2\pi i f_n t} dt \approx \sum_{k=0}^{N-1} h_k e^{2\pi i f_n t_k} \Delta = \Delta \sum_{k=0}^{N-1} h_k e^{2\pi i f_n t_k}$$

$f_n = n / \Delta$

$$\begin{cases} t_k = \Delta k / N \\ f_n = n / \Delta \end{cases}$$

$$H(f_n) = \Delta \sum_{k=0}^{N-1} h_k e^{2\pi i k n / N}$$

The last expression is periodic, with period N. It define a between 2 sets of numbers , $\mathbf{H_n}$ & $\mathbf{h_k}$ ($\mathbf{H(f_n) = \Delta H_n}$)

Discrete Fourier Transforms (DFT)

$$H_n = \sum_{k=0}^{N-1} h_k e^{2\pi i k n / N} \qquad h_k = \frac{1}{N} \sum_{n=0}^{N-1} H_n e^{-2\pi i k n / N}$$

frequencies from **0** to **fc** (maximum frequency) are mapped in the values with index from **0** to **N/2-1**, while negative ones are up to **-fc** mapped with index values of **N / 2** to **N**

Scale like N*N

Fast Fourier Transform (FFT)

The DFT can be calculated very efficiently
using the algorithm known as the FFT, which uses
symmetry properties of the DFT s

$$
\begin{aligned}
F_k &= \sum_{j=0}^{N-1} e^{2\pi ijk/N} f_j \\
&= \sum_{j=0}^{N/2-1} e^{2\pi ik(2j)/N} f_{2j} + \sum_{j=0}^{N/2-1} e^{2\pi ik(2j+1)/N} f_{2j+1} \\
&= \sum_{j=0}^{N/2-1} e^{2\pi ikj/(N/2)} f_{2j} + W^k \sum_{j=0}^{N/2-1} e^{2\pi ikj/(N/2)} f_{2j+1} \\
&= F_k^e + W^k\, F_k^o
\end{aligned}
$$

Fast Fourier Transform (FFT)

exp(2πi/N)

$$\boxed{F_k = F_k^e + W^k F_k^o}$$

DFT of even terms

DFT of odd terms

Fast Fourier Transform (FFT)

Now Iterate:

$$F^{e} = F^{ee} + W^{k/2}\, F^{eo}$$

$$F^{o} = F^{oe} + W^{k/2}\, F^{oo}$$

You obtain a series for each value of f_n

$$F^{oeoeooeo..oe} = f_n$$

Scale like N*logN (binary tree)

Parallel Domain Decomposition

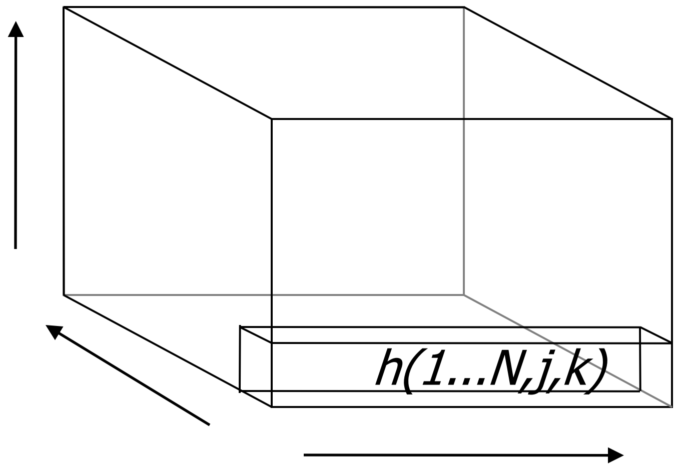
How to compute a FFT on a distributed memory system

Introduction

- On a 1D array:
  - Algorithm limits:
    - All the tasks must know the whole initial array
    - No advantages in using distributed memory systems
  - Solutions:
    - Using OpenMP it is possible to increase the performance on shared memory systems
- On a Multi-Dimensional array:
  - It is possible to use distributed memory systems

Multi-dimensional FFT(an example)

$$H(n_1, n_2) = \text{FFT-on-index-1}\left(\text{FFT-on-index-2}\left[h(k_1, k_2)\right]\right)$$
$$= \text{FFT-on-index-2}\left(\text{FFT-on-index-1}\left[h(k_1, k_2)\right]\right)$$



*1) For each value of **j** and **k***

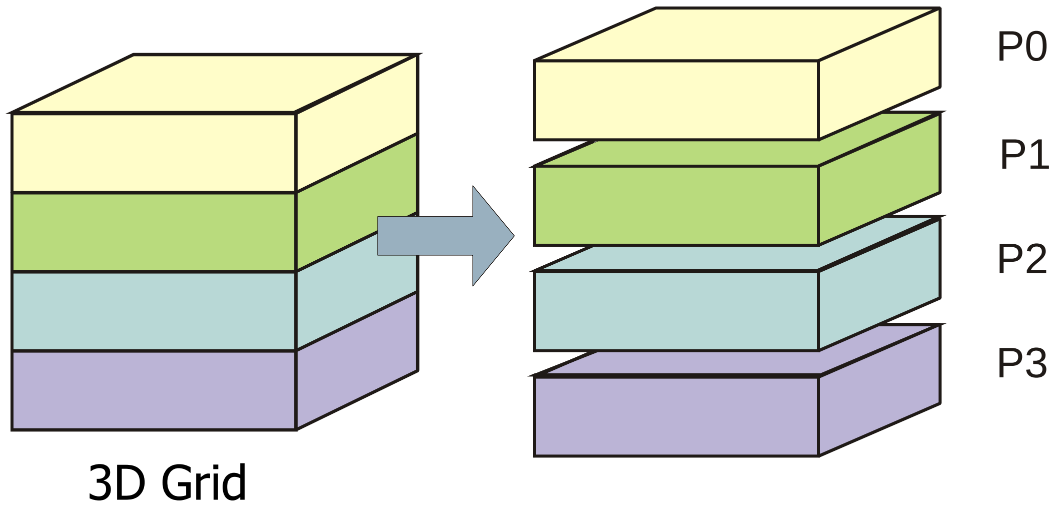
Apply FFT to h(1...N,j,k)

*2) For each value of **i** and **k***

Apply FFT to h(i,1...N,k)

*3) For each value of **i** and **j***
Apply FFT to h(i,j,1...N)

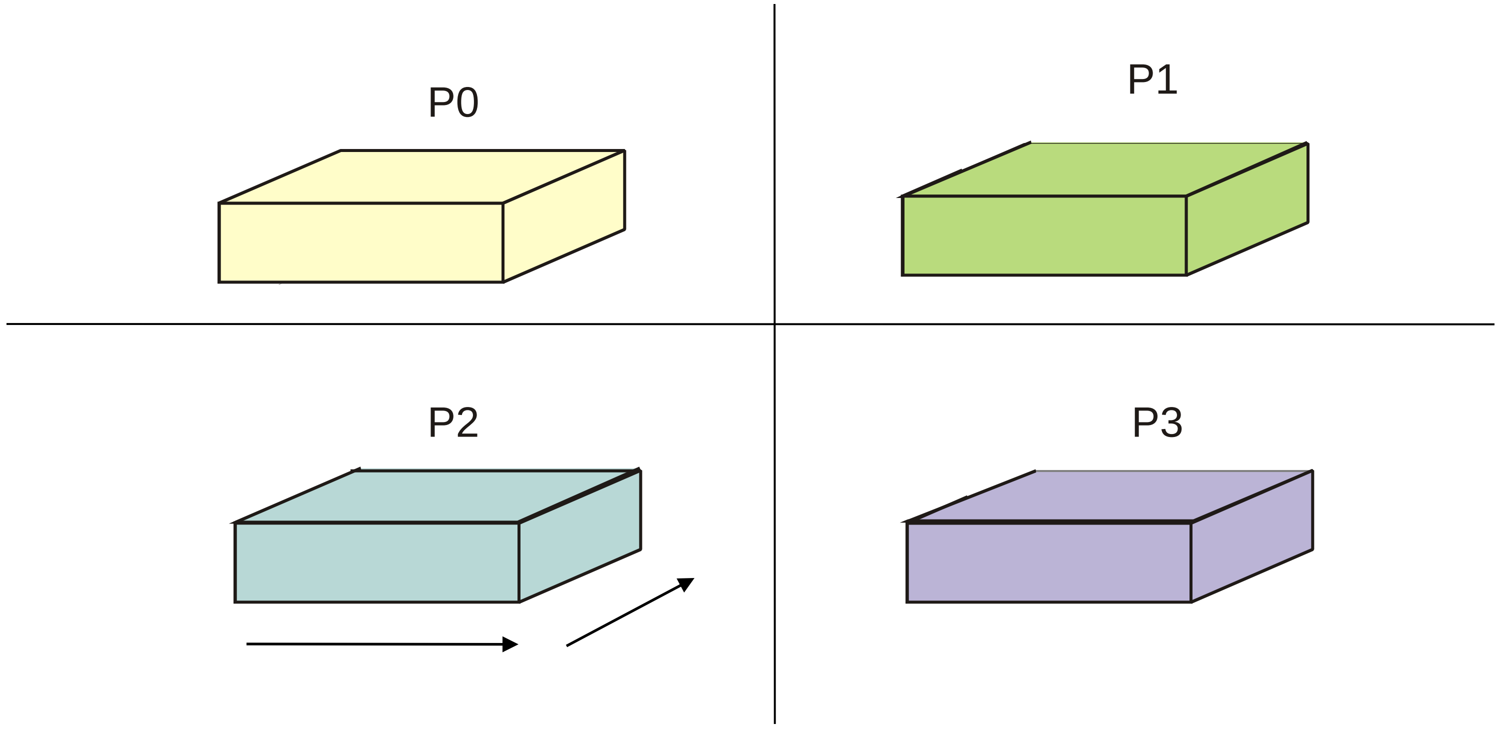
Parallel FFT Data Distribution



Distribute data along one coordinate (e.g. *Z*)

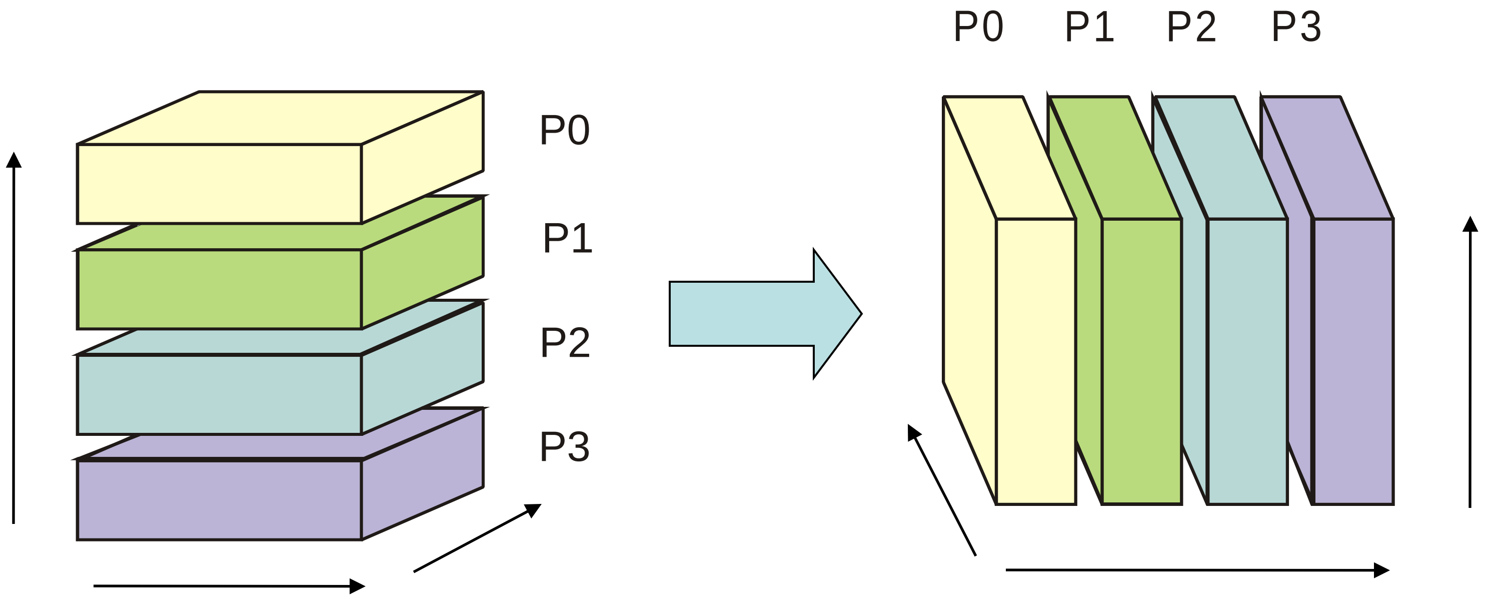
This is know as "Slab Decomposition" or 1D Decomposition

Transform along x and y



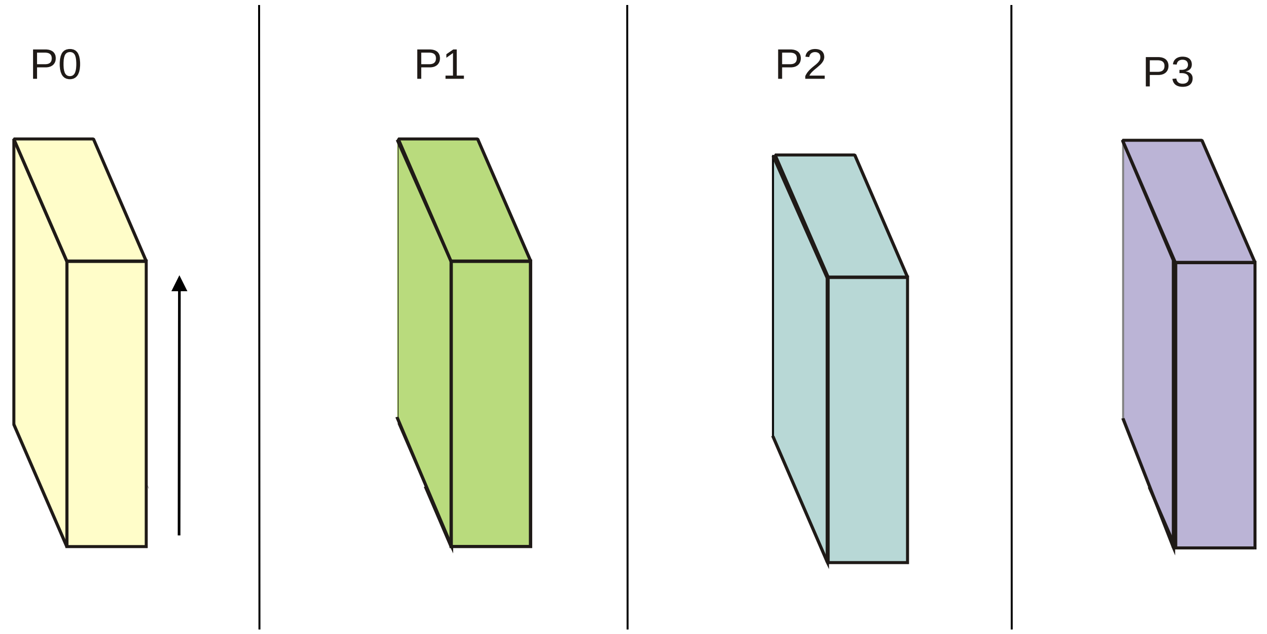
each processor trasform its own sub-grid along the x and y independently of the other

Data redistribution involving x and z



The data are now distributed along x

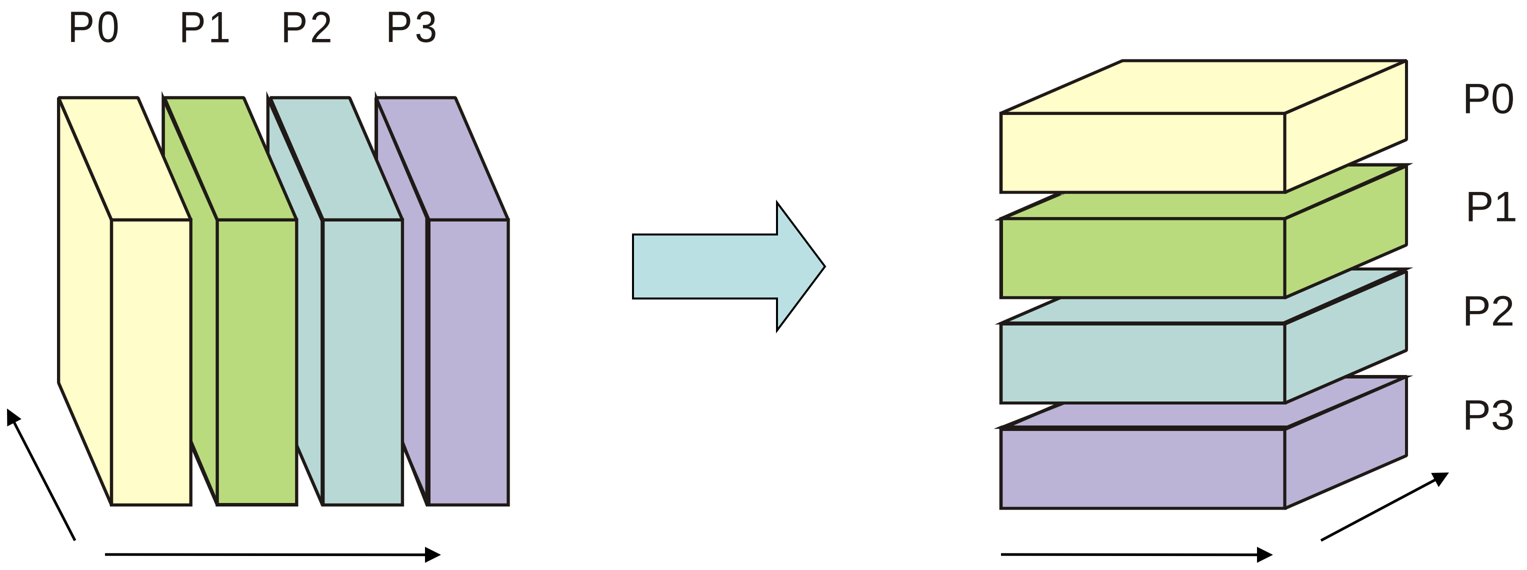
FFT along z



each processor transform its own sub-grid

along the z dimension independently of the other

Data are re-distributed, back from x to z



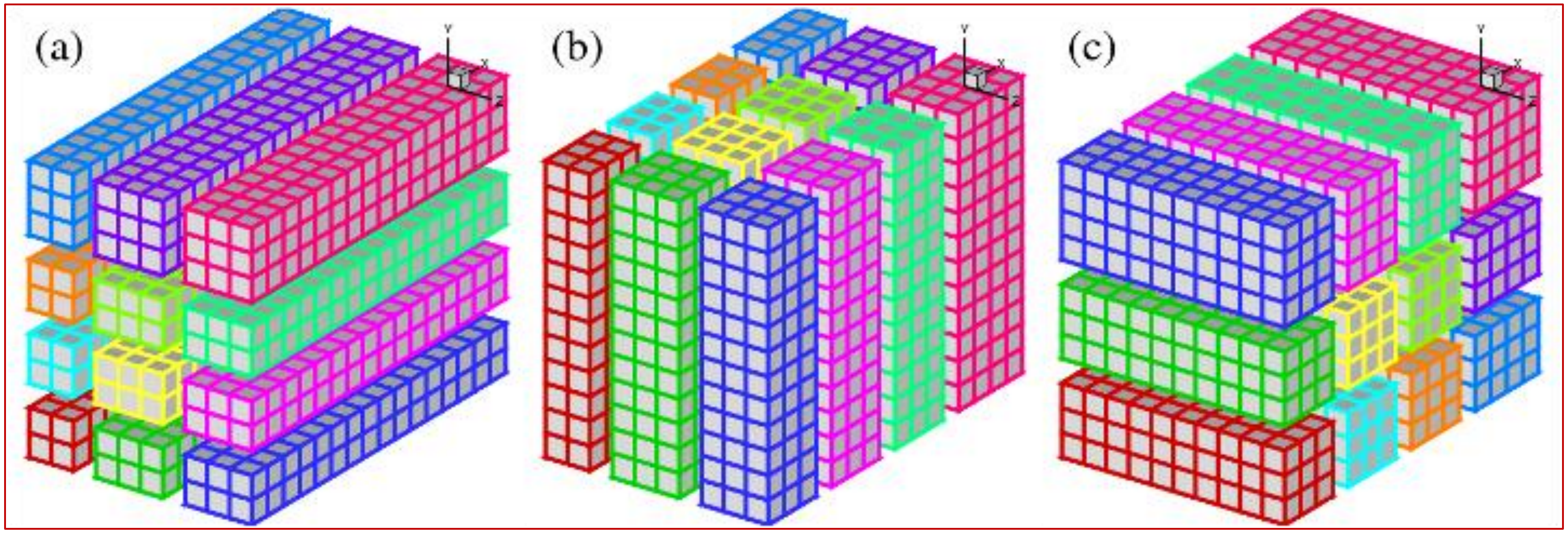
The 3D array now has the original layout, but each element

Has been substituted with its FFT.

Limit of Slab Decomposition

- Pro:
  - Simply to implement
  - Moderate communications
- Con:
  - Parallelization only along one direction
  - Maximum number of MPI tasks bounded by the size of the larger array index
- Possible Solutions:
  - 2D (Pencil) Decomposition

2D Domain Decomposition



Slab vs Pencil Decomposition

- Slab (1D) decomposition:
  - Faster on a limited number of cores
  - Parallelization is limited by the length of the largest axis of the 3D data array used
- Pencil (2D) decomposition:
  - Faster on massively parallel supercomputers
  - Slower using large size arrays on a moderate number of cores (more MPI communications)

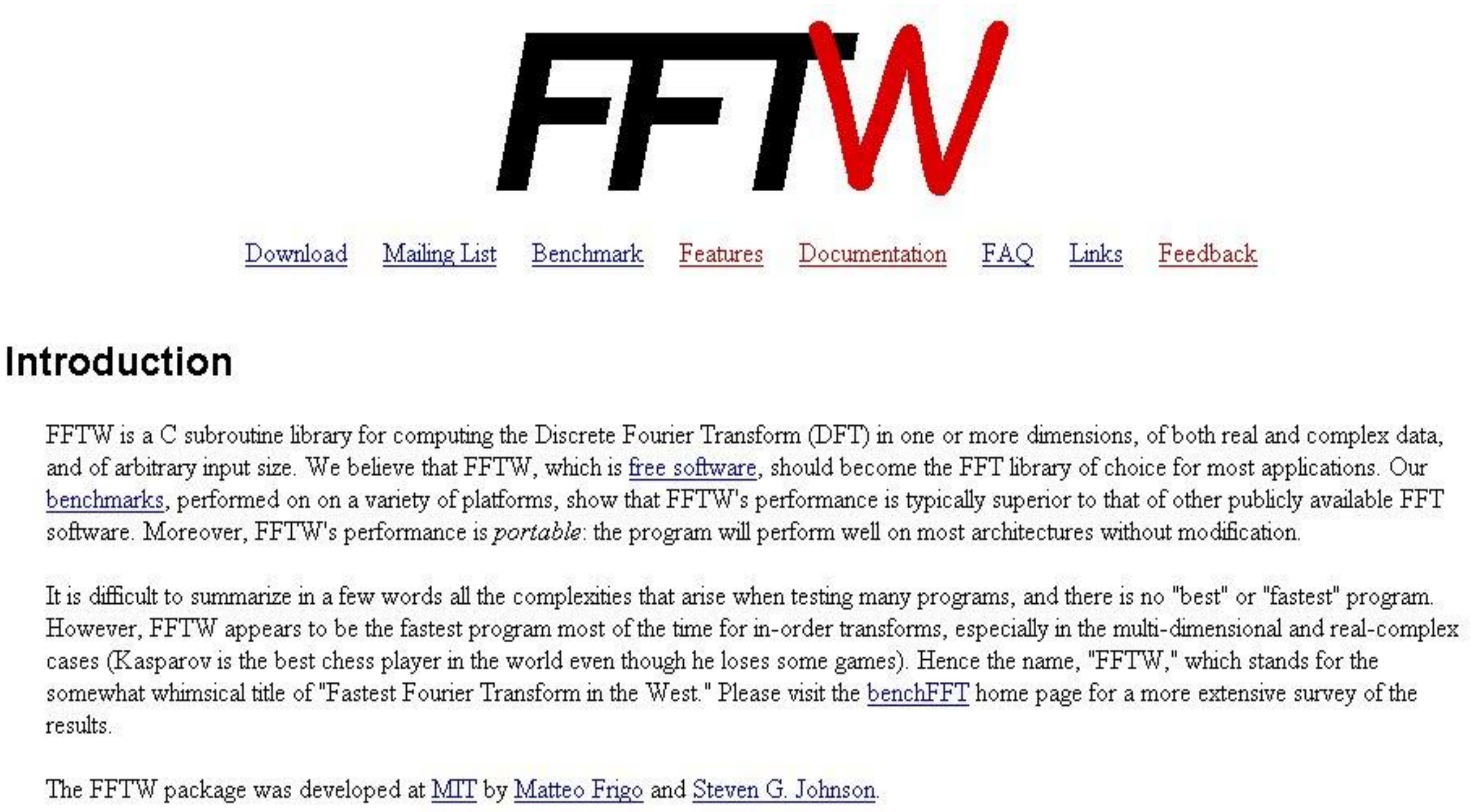
Some useful papers

- Auto-tuning of the FFTW Library for Massively Parallel Supercomputers.
  - M. Guarrasi, G. Erbacci, A. Emerson;
  - 2012, PRACE white paper;
  - Available at this link;
- Scalability Improvements for DFT Codes due to the Implementation of the 2D Domain Decomposition Algorithm.
  - M. Guarrasi, S. Frigio, A. Emerson, G. Erbacci
  - 2013, PRACE white paper;
  - Available at this link
- Testing and Implementing Some New Algorithms Using the FFTW Library on Massively Parallel Supercomputers.
  - M. Guarrasi, N. Li, S. Frigio, A. Emerson, G. Erbacci;
  - Accepted for ParCo 2013 conference proceedings.
- 2DECOMP&FFT – A highly scalable 2D decomposition library and FFT interface.
  - N. Li, S. Laizet;
  - 2010, Cray User Group 2010 conference;
  - Available at this link
- P3DFFT: a framework for parallel computations of Fourier transforms in three dimensions.
  - D. Pekurovsky;
  - 2012, SIAM Journal on Scientific Computing, Vol. 34, No. 4, pp. C192-C209
- The Design and Implementation of FFTW3.
  - M. Frigio, S. G. Johnson;
  - 2005, Proceedings of the IEEE.

FFT Numerical Libraries

The simplest way to compute a FFT on a modern HPC system

http://www.fftw.org



FFTW

Download Mailing List Benchmark Features Documentation FAQ Links Feedback

Introduction

FFTW is a C subroutine library for computing the Discrete Fourier Transform (DFT) in one or more dimensions, of both real and complex data, and of arbitrary input size. We believe that FFTW, which is free software, should become the FFT library of choice for most applications. Our benchmarks, performed on on a variety of platforms, show that FFTW's performance is typically superior to that of other publicly available FFT software. Moreover, FFTW's performance is *portable*: the program will perform well on most architectures without modification.

It is difficult to summarize in a few words all the complexities that arise when testing many programs, and there is no "best" or "fastest" program. However, FFTW appears to be the fastest program most of the time for in-order transforms, especially in the multi-dimensional and real-complex cases (Kasparov is the best chess player in the world even though he loses some games). Hence the name, "FFTW," which stands for the somewhat whimsical title of "Fastest Fourier Transform in the West." Please visit the benchFFT home page for a more extensive survey of the results.

The FFTW package was developed at MIT by Matteo Frigo and Steven G. Johnson.

- Written in C
- Fortran wrapper is also provided
- FFTW adapt itself to your machines, your cache, the size of your memory, the number of register, etc...
- FFTW doesn't use a fixed algorithm to make DFT
  - FFTW chose the best algorithm for your machines
- Computation is split in 2 phases:
  - PLAN creation
  - Execution
- FFTW support transforms of data with arbitrary length, rank, multiplicity, and memory layout, and more....

- Many different versions:
  - FFTW 2:
    - Released in 2003
    - Well tested and used in many codes
    - Includes serial and parallel transforms for both shared and distributed memory system
  - FFTW 3:
    - Released in February 2012
    - Includes serial and parallel transforms for both shared and distributed memory system
    - Hybrid implementation MPI-OpenMP
    - Last version is FFTW 3.3.3

FFTW

Some Useful Instructions

How can I compile a code that uses FFTW on PLX?

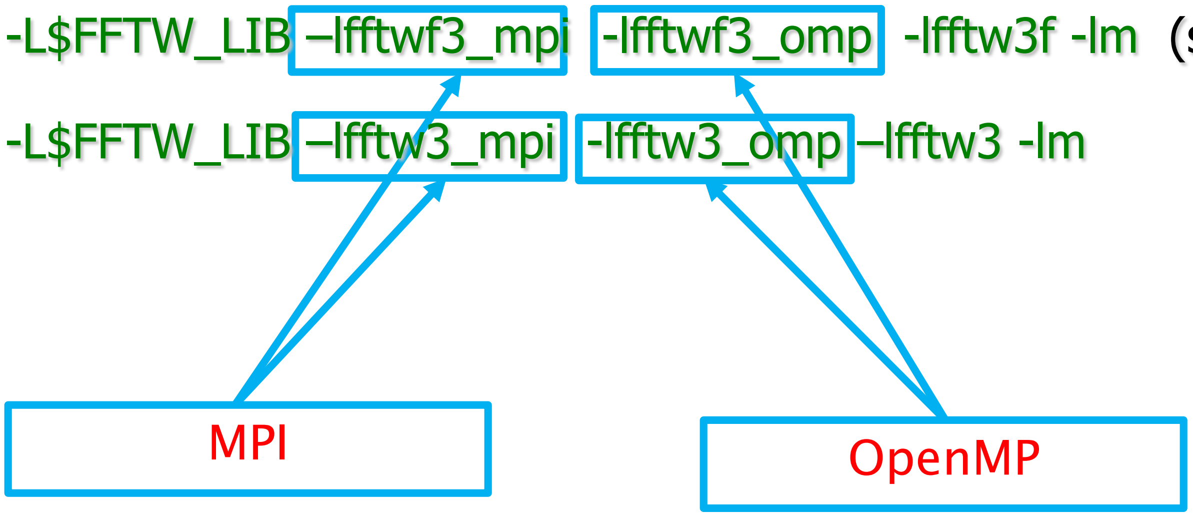
- Module Loading:

    module load autoload fftw/3.3.3--openmpi--1.6.3--intel--cs-xe-2013--binary

- Including header:

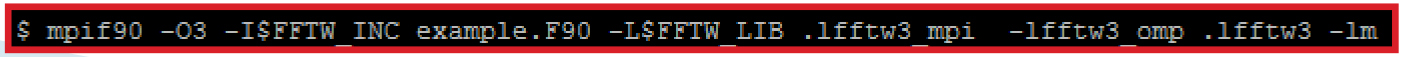
    - -I$FFTW_INC

- Linking:



```
-L$FFTW_LIB –lfftwf3_mpi  -lfftwf3_omp  -lfftw3f -lm   (single precision)
-L$FFTW_LIB –lfftw3_mpi  -lfftw3_omp –lfftw3 -lm       (double precision)
```

- An example:



```
$ mpif90 -O3 -I$FFTW_INC example.F90 -L$FFTW_LIB .lfftw3_mpi  -lfftw3_omp .lfftw3 -lm
```

How can I compile a code that uses FFTW on FERMI?

•Module Loading:

```
module load autoload fftw/3.3.2--bgq-gnu--4.4.6
```

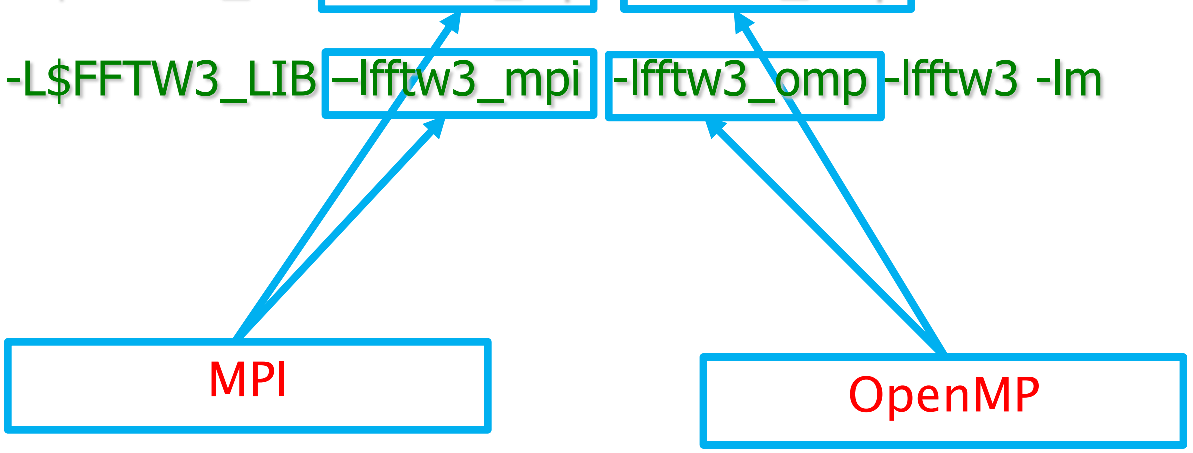
•Including header:

•-I$FFTW3_INC

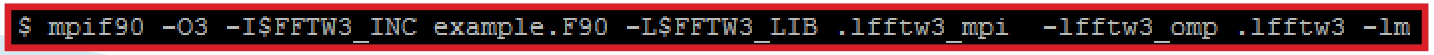
•Linking:

-L$FFTW3_LIB –lfftw3f_mpi -lfftw3f_omp -lfftw3f -lm (single precision)

-L$FFTW3_LIB –lfftw3_mpi -lfftw3_omp -lfftw3 -lm (double precision)



•An example:



```
$ mpif90 -O3 -I$FFTW3_INC example.F90 -L$FFTW3_LIB .lfftw3_mpi  -lfftw3_omp .lfftw3 -lm
```

Some important Remarks for FORTRAN users

• Function in C became function in FORTRAN if they have a return value, and subroutines otherwise.
• All C types are mapped via the iso_c_binning standard.
• FFTW plans are type(C_PTR) in FORTRAN.
•The ordering of FORTRAN array dimensions must be reversed when they are passed to the FFTW plan creation

Initialize FFTW

Including FFTW Lib:

- C:
    - Serial:

      `#include <fftw.h>`
    - MPI:

      `#include <fftw-mpi.h>`
- FORTRAN:
    - Serial:

      `include 'fftw3.f03'`
    - MPI:

      `include 'fftw3-mpi.f03'`

MPI initializzation:

- C:

  `void fftw_mpi_init(void)`
- FORTRAN:

  `fftw_mpi_init()`

Array creation

C:

- Fixed size array:

```
fftx_complex data[n0][n1][n2]
```

- Dynamic array:

```
data = fftw_alloc_complex(n0*n1*n2)
```

- MPI dynamic arrays:

```
fftw_complex *data
ptrdiff_t alloc_local, local_no, local_no_start
alloc_local= fftw_mpi_local_size_3d(n0, n1, n2, MPI_COMM_WORLD, &local_n0,&local_n0_start)
data = fftw_alloc_complex(alloc _local)
```

FORTRAN:

- Fixed size array (simplest way):

```
complex(C_DOUBLE_COMPLEX), dimension(n0,n1,n2) :: data
```

- Dynamic array (simplest way):

```
complex(C_DOUBLE_COMPLEX), allocatable, dimension(:, :, :) :: data
allocate (data(n0, n1, n2))
```

- Dynamic array (fastest method):

```
complex(C_DOUBLE_COMPLEX), pointer :: data(:, :, :) )
type(C_PTR) :: cdata
cdata = fftw_alloc_complex(n0*n1*n2)
call c_f_pointer(cdata, data, [n0,n1,n2])
```

- MPI dynamic arrays:

```
complex(C_DOUBLE_COMPLEX), pointer :: data(:, :, :)
type(C_PTR) :: cdata
integer(C_INTPTR_T) :: alloc_local, local_n2, local_n2_offset
alloc_local = fftw_mpi_local_size_3d(n2, n1, n0, MPI_COMM_WORLD, local_n2, local_n2_offset)
cdata = fftw_alloc_complex(alloc_local)
call c_f_pointer(cdata, data, [n0,n1,local_n2])
```

Plan Creation (C2C)

1D Complex to complex DFT:

• C:

```
fftw_plan = fftw_plan_dft_1d(int nx, fftw_complex *in, fftw_complex *out, fftw_direction dir, unsigned flags)
```

•FORTRAN:

```
plan = ftw_plan_dft_1d(nz, in, out, dir, flags)
```

FFTW_FORWARD
FFTW_BACKWARD

FFTW_ESTIMATE
FFTW_MEASURE

2D Complex to complex DFT:

• C:

```
fftw_plan = fftw_plan_dft_2d(int nx, int ny, fftw_complex *in, fftw_complex *out, fftw_direction dir, unsigned flags)

fftw_plan = fftw_mpi_plan_dft_2d(int nx, int ny, fftw_complex *in, fftw_complex *out,MPI_COMM_WORLD, fftw_direction dir, int flags)
```

•FORTRAN:

```
plan = ftw_plan_dft_2d(ny, nx, in, out, dir, flags)

plan = ftw_mpi_plan_dft_2d(ny, nx, in, out, MPI_COMM_WORLD, dir, flags)
```

3D Complex to complex DFT:

• C:

```
fftw_plan = fftw_plan_dft_3d(int nx, int ny, int nz, fftw_complex *in, fftw_complex *out, fftw_direction dir, unsigned flags)

fftw_plan = fftw_mpi_plan_dft_3d(int nx, int ny, int nz, fftw_complex *in, fftw_complex *out,MPI_COMM_WORLD, fftw_direction dir, int flags)
```

•FORTRAN:

```
plan = ftw_plan_dft_3d(nz, ny, nx, in, out, dir, flags)

plan = ftw_mpi_plan_dft_3d(nz, ny, nx, in, out, MPI_COMM_WORLD, dir, flags)
```

Plan Creation (R2C)

1D Real to complex DFT:

• C:

fftw_plan = fftw_plan_dft_r2c_1d(int nx, double *in, fftw_complex *out, fftw_direction dir, unsigned flags)

•FORTRAN:

ftw_plan_dft_r2c_1d(nz, in, out, dir, flags)

FFTW_FORWARD
FFTW_BACKWARD

FFTW_ESTIMATE
FFTW_MEASURE

2D Real to complex DFT:

• C:

fftw_plan = fftw_plan_dft_r2c_2d(int nx, int ny, double *in, fftw_complex *out, fftw_direction dir, unsigned flags)

fftw_plan = fftw_mpi_plan_dft_r2c_2d(int nx, int ny, double *in, fftw_complex *out,MPI_COMM_WORLD, fftw_direction dir, int flags)

•FORTRAN:

ftw_plan_dft_r2c_2d(ny, nx, in, out, dir, flags)

ftw_mpi_plan_dft_r2c_2d(ny, nx, in, out, MPI_COMM_WORLD, dir, flags)

3D Real to complex DFT:

• C:

fftw_plan = fftw_plan_dft_r2c_3d(int nx, int ny, int nz, fftw_complex *in, fftw_complex *out, fftw_direction dir, unsigned flags)

fftw_plan = fftw_mpi_plan_dft_r2c_3d(int nx, int ny, int nz, fftw_complex *in, fftw_complex *out,MPI_COMM_WORLD, fftw_direction dir, int flags)

•FORTRAN:

ftw_plan_dft_r2c_3d(nz, ny, nx, in, out, dir, flags)

ftw_mpi_plan_dft_r2c_3d(nz, ny, nx, in, out, MPI_COMM_WORLD, dir, flags)

Plan Execution

Complex to complex DFT:

- C:

void fftw_execute_dft(fftw_plan plan, fftw_complex *in, fftw_complex *out)
void fftw_mpi_execute_dft (fftw_plan plan, fftw_complex *in, fftw_complex *out)

- FORTRAN:

fftw_execute_dft (plan, in, out)
fftw_mpi_execute_dft (plan, in, out)

Real to complex DFT:

- C:

void fftw_execute_dft (fftw_plan plan, double *in, fftw_complex *out)
void fftw_mpi_execute_dft (fftw_plan plan, double *in, fftw_complex *out)

- FORTRAN:

fftw_execute_dft (plan, in, out)
Fftw_mpi_execute_dft (plan, in, out)

Finalizing FFTW

Destroying PLAN:

• C:

```
void  fftw_destroy_plan(fftw_plan plan)
```

• FORTRAN:

```
fftw_destroy_plan(plan)
```

FFTW MPI cleanup:

• C:

```
void  fftw_mpi_cleanup ()
```

• FORTRAN:

```
fftw_mpi_cleanup ()
```

Deallocate data:

• C:

```
void  fftw_free (fftw_complex data)
```

• FORTRAN:

```
fftw_free (data)
```

FFTW

Some Useful Examples

1D Serial FFT for a fixed size array – Fortran

```
 program FFTW1D
       use, intrinsic :: iso_c_binding
       implicit none
       include 'fftw3.f03'
       integer(C_INTPTR_T):: L = 1024
       integer(C_INT) :: LL
       type(C_PTR) :: plan1
       complex(C_DOUBLE_COMPLEX), dimension(1024) :: idata, odata
       integer :: i
       character(len=41), parameter :: filename='serial_data.txt'
       LL = int(L,C_INT)
!! create MPI plan for in-place forward DF
         plan1 = fftw_plan_dft_1d(LL, idata, odata, FFTW_FORWARD, FFTW_ESTIMATE)
!! initialize data
         do i = 1, L
             if (i .le. (L/2)) then
                 idata(i) = (1.,0.)
             else
                 idata(i) = (0.,0.)
             endif
         end do
!! compute transform (as many times as desired)
         call fftw_execute_dft(plan1, idata, odata)
!! deallocate and destroy plans
         call fftw_destroy_plan(plan1)
     end
```

1D Serial FFT for a variable size array – Fortran

```
 program FFTW1D
 use, intrinsic :: iso_c_binding
       implicit none
       include 'fftw3.f03'
       integer(C_INTPTR_T):: L = 1024
       integer(C_INT) :: LL
       type(C_PTR) :: plan1
       type(C_PTR) :: p_idata, p_odata
       complex(C_DOUBLE_COMPLEX), dimension(:), pointer :: idata,odata
       integer :: i
!! Allocate
          LL = int(L,C_INT)
          p_idata = fftw_alloc_complex(L)
          p_odata = fftw_alloc_complex(L)
          call c_f_pointer(p_idata,idata,(/L/))
          call c_f_pointer(p_odata,odata,(/L/))
!!   create MPI plan for in-place forward DF
          plan1 = fftw_plan_dft_1d(LL, idata, odata, FFTW_FORWARD, FFTW_ESTIMATE)
!! initialize data
         do i = 1, L
             if (i .le. (L/2)) then
                 idata(i) = (1.,0.)
             else
                 idata(i) = (0.,0.)
             endif
         end do
!! compute transform (as many times as desired)
         call fftw_execute_dft(plan1, idata, odata)
!! deallocate and destroy plans
         call fftw_destroy_plan(plan1)
         call fftw_free(p_idata)
         call fftw_free(p_odata)
     end
```

1D Serial FFT for a variable size array – C

```
# include <stdlib.h>
# include <stdio.h>
# include <math.h>
# include <fftw3.h>

int main ( void )

{
  ptrdiff_t i;
  const ptrdiff_t n = 1024;
  fftw_complex *in;
  fftw_complex *out;
  fftw_plan plan_forward;
/* Create arrays. */
   in = fftw_malloc ( sizeof ( fftw_complex ) * n );
   out = fftw_malloc ( sizeof ( fftw_complex ) * n );
/* Initialize data */
   for ( i = 0; i < n; i++ )  {
      if (i <= (n/2-1))   {
          in[i][0] = 1.;
          in[i][1] = 0.;
        }
      else {
          in[i][0] = 0.;
          in[i][1] = 0.;
        }
   }
/* Create plans. */
   plan_forward = fftw_plan_dft_1d ( n, in, out, FFTW_FORWARD, FFTW_ESTIMATE );
/* Compute transform (as many times as desired) */
   fftw_execute ( plan_forward );
/* deallocate and destroy plans */
   fftw_destroy_plan ( plan_forward );
   fftw_free ( in );
   fftw_free ( out );
  return 0;
}
```

2D Parallel FFT – Fortran (part1)

```
program FFT_MPI_3D
        use, intrinsic :: iso_c_binding
        implicit none
                 include 'mpif.h'
        include 'fftw3-mpi.f03'
        integer(C_INTPTR_T), parameter :: L = 1024
        integer(C_INTPTR_T), parameter :: M = 1024
        type(C_PTR) :: plan, cdata
        complex(C_DOUBLE_COMPLEX), pointer :: fdata(:,:)
        integer(C_INTPTR_T) :: alloc_local, local_M, local_j_offset
        integer(C_INTPTR_T) :: i, j
        complex(C_DOUBLE_COMPLEX) :: fout
        integer :: ierr, myid, nproc
! Initialize
                    call mpi_init(ierr)
                    call MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
                    call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
                    call fftw_mpi_init()
!   get local data size and allocate (note dimension reversal)
                    alloc_local = fftw_mpi_local_size_2d(M, L, MPI_COMM_WORLD, local_M, local_j_offset)
                    cdata = fftw_alloc_complex(alloc_local)
                    call c_f_pointer(cdata, fdata, [L,local_M])
!   create MPI plan for in-place forward DFT (note dimension reversal)
            plan = fftw_mpi_plan_dft_2d(M, L, fdata, fdata, MPI_COMM_WORLD, FFTW_FORWARD,
FFTW_MEASURE)
```

2D Parallel FFT – Fortran (part2)

```
! initialize data to some function my_function(i,j)
            do j = 1, local_M
              do i = 1, L
                call initial(i, (j + local_j_offset), L, M, fout)
                fdata(i, j) = fout
              end do
            end do
! compute transform (as many times as desired)
            call fftw_mpi_execute_dft(plan, fdata, fdata)!
! deallocate and destroy plans
            call fftw_destroy_plan(plan)
            call fftw_mpi_cleanup()
            call fftw_free(cdata)
            call mpi_finalize(ierr)
      end
```

2D Parallel FFT – C (part1)

```
# include <stdlib.h>
# include <stdio.h>
# include <math.h>
# include <mpi.h>
# include <fftw3-mpi.h>

int main(int argc, char **argv)
{
    const ptrdiff_t L = 1024, M = 1024;
    fftw_plan plan;
    fftw_complex *data ;
    ptrdiff_t alloc_local, local_L, local_L_start, i, j, ii;
    double xx, yy, rr, r2, t0, t1, t2, t3, tplan, texec;
    const double amp = 0.25;
    /* Initialize */
    MPI_Init(&argc, &argv);
    fftw_mpi_init();

    /* get local data size and allocate */
    alloc_local = fftw_mpi_local_size_2d(L, M, MPI_COMM_WORLD, &local_L, &local_L_start);
    data = fftw_alloc_complex(alloc_local);
    /* create plan for in-place forward DFT */
    plan = fftw_mpi_plan_dft_2d(L, M, data, data, MPI_COMM_WORLD, FFTW_FORWARD, FFTW_ESTIMATE);
```

2D Parallel FFT – C (part2)

```
/* initialize data to some function my_function(x,y) */
/*……..*/
/* compute transforms, in-place, as many times as desired */
        fftw_execute(plan);
/* deallocate and destroy plans */
        fftw_destroy_plan(plan);
        fftw_mpi_cleanup();
        fftw_free ( data );
        MPI_Finalize();
    }
```

2DECOMP & FFT

The most important FFT Fortran Library that use 2D (Pencil) Domain Decomposition

- General-purpose 2D pencil decomposition module to support building large-scale parallel applications on distributed memory systems.
- Highly scalable and efficient distributed Fast Fourier Transform module, supporting three dimensional FFTs (both complex-to-complex and real-to-complex/complex-to-real).
- Halo-cell support allowing explicit message passing between neighbouring blocks.
- Parallel I/O module to support the handling of large data sets.
- Shared-memory optimisation on the communication code for multi-code systems.
- Written in Fortran
- Best performance using Fortran 2003 standard
- No C wrapper is already provided
- Structure: Plan Creation – Execution – Plan Destruction
- Uses FFTW lib (or ESSL) to compute 1D transforms
- More efficient on massively parallel supercomputers.
- Well tested
- Additional features

How can I compile a code that uses 2Decomp&FFT on PLX?

- Module Loading:

module load autoload profile/advanced

Module load 2Decomp_FFT/1.5.847--openmpi--1.6.3--intel--cs-xe-2013—binary

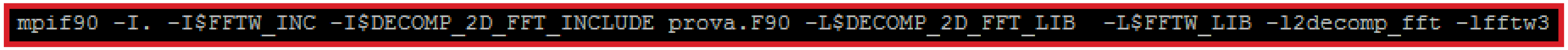
- Including header:

-I$FFTW_INC -I$DECOMP_2D_FFT_INC

- Linking (double precision):

-L$DECOMP_2D_FFT_LIB -L$FFTW_LIB -l2decomp_fft -lfftw3_mpi -lfftw3

- Example:



```
mpif90 -I. -I$FFTW_INC -I$DECOMP_2D_FFT_INCLUDE prova.F90 -L$DECOMP_2D_FFT_LIB  -L$FFTW_LIB -l2decomp_fft -lfftw3
```

How can I compile a code that uses 2Decomp&FFT on FERMI?

- Module Loading:

```
module load autoload profile/advanced

module load 2Decomp_fft/1.5.847--bgq-gnu--4.4.6
```

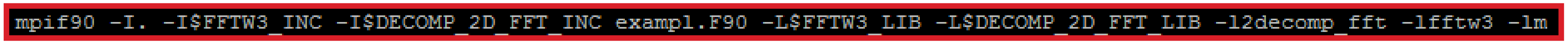
- Including header:

```
-I$FFTW3_INC  -I$DECOMP_2D_FFT_INC
```

- Linking (double precision):

```
-L$DECOMP_2D_FFT_LIB  -L$FFTW3_LIB  -l2decomp_fft  -lfftw3_mpi -lfftw3
```

- Example:



```
mpif90 -I. -I$FFTW3_INC -I$DECOMP_2D_FFT_INC exampl.F90 -L$FFTW3_LIB -L$DECOMP_2D_FFT_LIB -l2decomp_fft -lfftw3 -lm
```

Some useful instructions (FORTRAN only)

Including 2Decomp&FFT Lib:

```
use decomp_2d
use decomp_2d_fft
```

Initial declarations:

```
   integer, parameter :: p_row = ..
   integer, parameter :: p_col = ...
complex(mytype), allocatable, dimension(:,:,:) :: in, out
```

Please note that: p_row*p_col = N_core

Initialization:

```
call decomp_2d_init(n0,n1,n2,p_row,p_col)
call decomp_2d_fft_init
```

Allocation:

```
call decomp_2d_fft_get_size(fft_start,fft_end,fft_size)
 allocate (in(xstart(1):xend(1),xstart(2):xend(2),xstart(3):xend(3)))
 allocate (out(fft_start(1):fft_end(1), fft_start(2):fft_end(2), fft_start(3):fft_end(3))) ! R2C FFT
allocate (out(zstart(1):zend(1),zstart(2):zend(2),zstart(3):zend(3))) ! C2C FFT
```

Execution:

```
call decomp_2d_fft_3d(in, out, DECOMP_2D_FFT_FORWARD)
```

Finalizing:

```
call decomp_2d_fft_finalize
call decomp_2d_finalize
```

2DECOMP & FFT

An example

3D FFT (part 1)

```
 PROGRAM FFT_3D_2Decomp_MPI
       use mpi
       use, intrinsic :: iso_c_binding
       use decomp_2d
       use decomp_2d_fft
      implicit none
      integer, parameter :: L = 128
      integer, parameter :: M = 128
      integer, parameter :: N = 128
      integer, parameter :: p_row = 16
      integer, parameter :: p_col = 16
      integer :: nx, ny, nz
      complex(mytype), allocatable, dimension(:,:,:) :: in, out
      complex(mytype) :: fout
      integer :: ierror, i,j,k, numproc, mype
      integer, dimension(3) :: sizex, sizez
! ===== Initialize
       call MPI_INIT(ierror)
       call decomp_2d_init(L,M,N,p_row,p_col)
       call decomp_2d_fft_init
```

3D FFT (part 2)

```
      allocate (in(xstart(1):xend(1),xstart(2):xend(2),xstart(3):xend(3)))
      allocate (out(zstart(1):zend(1),zstart(2):zend(2),zstart(3):zend(3)))
! ===== each processor gets its local portion of global data =====
      do k=xstart(3),xend(3)
        do j=xstart(2),xend(2)
          do i=xstart(1),xend(1)
            call initial(i, j, k, L, M, N, fout)
            in(i,j,k) = fout
          end do
        end do
       end do
! ===== 3D forward FFT =====
       call decomp_2d_fft_3d(in, out, DECOMP_2D_FFT_FORWARD)
! ==========================
       call decomp_2d_fft_finalize
       call decomp_2d_finalize
       deallocate(in,out)
       call MPI_FINALIZE(ierror)
     end
```

Parallel Three-Dimensional Fast Fourier Transforms (P3DFFT)

- General-purpose 2D pencil decomposition module to support building large-scale parallel applications on distributed memory systems.
- Highly scalable and efficient distributed Fast Fourier Transform module, supporting three dimensional FFTs (both complex-to-complex and real-to-complex/complex-to-real).
- Sine/cosine/Chebyshev/empty transform
- Shared-memory optimisation on the communication code for multi-code systems.
- Written in Fortran 90
- C wrapper is already provided
- Structure: Plan Creation – Execution – Plan Destruction
- Uses FFTW lib (or ESSL) to compute 1D transforms
- More efficient on massively parallel supercomputers.
- Well tested but not stable as 2Decomp&FFT
- Additional features

How can I compile a code that uses P3DFFT on PLX?

- Module Loading:

module load autoload profile/advanced

module load p3dfft/2.5.1--openmpi--1.6.3--intel--cs-xe-2013--binary

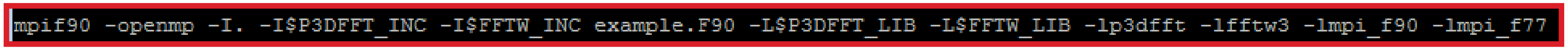
- Including header:

-I$FFTW_INC -I$P3DFFT_INC

- Linking (double precision):

-L$P3DFFT_LIB -L$FFTW_LIB -lp3dfft –lfftw3

- Example:



```
mpif90 -openmp -I. -I$P3DFFT_INC -I$FFTW_INC example.F90 -L$P3DFFT_LIB -L$FFTW_LIB -lp3dfft -lfftw3 -lmpi_f90 -lmpi_f77
```

How can I compile a code that uses P3DFFT on FERMI?

- Module Loading:

```
module load autoload profile/advanced

module load p3dfft/2.6.1.beta--bgq-gnu--4.4.6
```

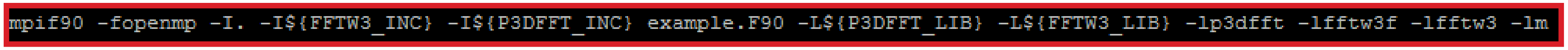
- Including header:

```
-I$FFTW_INC  -I$P3DFFT_INC
```

- Linking (double precision):

```
-L$P3DFFT_INC  -L$FFTW_LIB  -l2decomp_fft  -lfftw3_mpi -lfftw3
```

- Example:



```
mpif90 -fopenmp -I. -I${FFTW3_INC} -I${P3DFFT_INC} example.F90 -L${P3DFFT_LIB} -L${FFTW3_LIB} -lp3dfft -lfftw3f -lfftw3 -lm
```

Some useful instructions (C users)

Including P3DFFT Lib:

```
 #include “p3dfft.h”
```

Initial declarations:

```
unsigned char op_f[3]="fft", op_b[3]="tff";
MPI_Dims_create(nproc,2,dims);
if(dims[0] > dims[1]) {
dims[0] = dims[1];
dims[1] = nproc/dims[0]; }
```

Initialization:

```
p3dfft_setup(dims,nx,ny,nz,1,memsize);
p3dfft_get_dims(istart,iend,isize,1);
p3dfft_get_dims(fstart,fend,isize,2);
```

Allocation:

```
A = (double *) malloc(sizeof(double) * isize[0]*isize[1]*isize[2]);
B = (double *) malloc(sizeof(double) * fsize[0]*fsize[1]*fsize[2]*2);
C = (double *) malloc(sizeof(double) * isize[0]*isize[1]*isize[2]);
```

Execution:

```
  p3dfft_ftran_r2c(A,B,op_f);
 p3dfft_btran_c2r(B,C,op_b);
```

Finalizing:

```
 p3dfft_clean();
 MPI_Finalize();
```

An example

3D FFT (part 1)

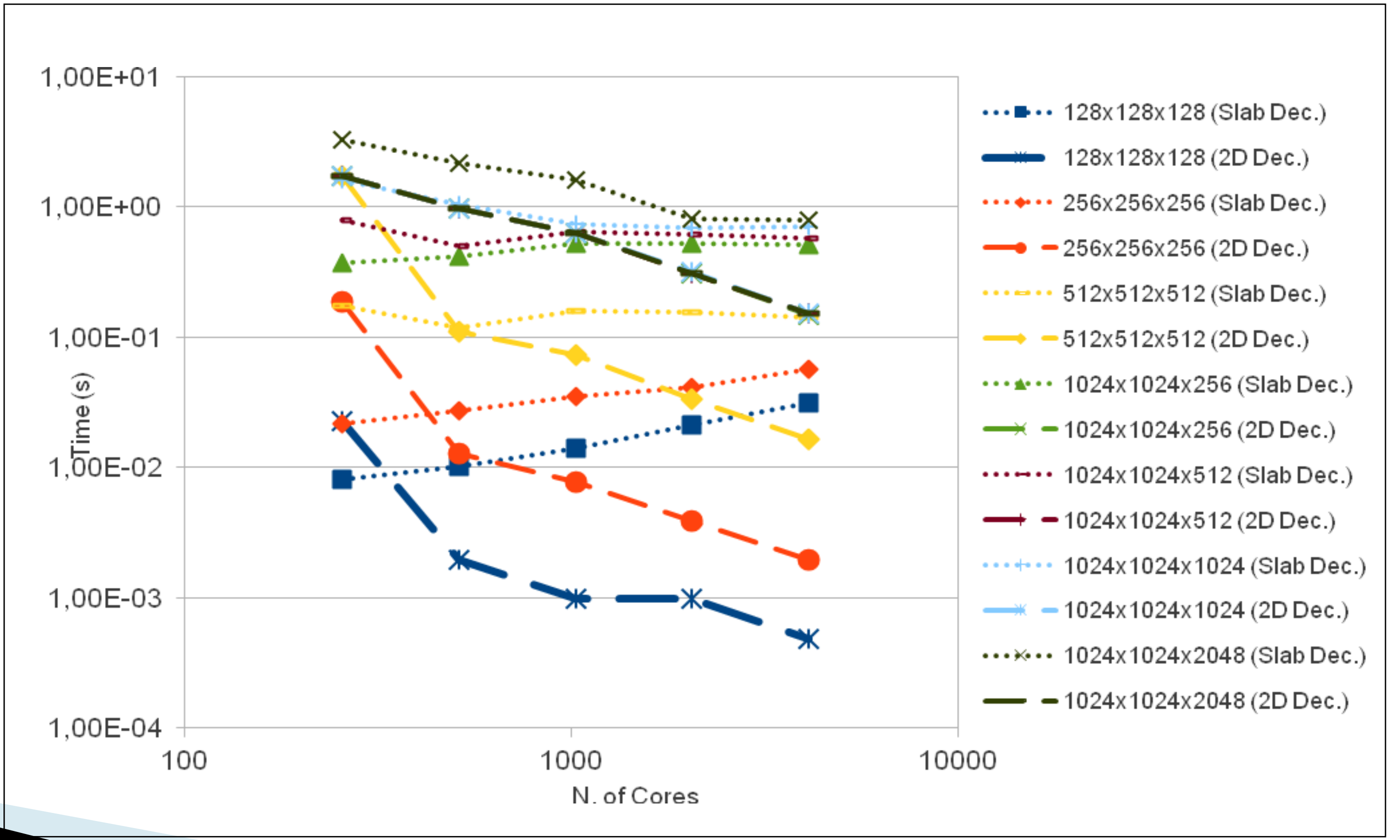
```
#include "p3dfft.h"
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main(int argc,char **argv)
{
   double *A,*B,*p,*C;
   int memsize[3];
   int istart[3], isize[3], iend[3];
   int fstart[3], fsize[3], fend[3];
   unsigned char op_f[3]="fft", op_b[3]="tff";
   /* .... ... ...*/
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&nproc);
   MPI_Comm_rank(MPI_COMM_WORLD,&proc_id);
   dims[0]=dims[1]=0;
   MPI_Dims_create(nproc,2,dims);
   if(dims[0] > dims[1]) {
     dims[0] = dims[1];
     dims[1] = nproc/dims[0];
    }
   /* ... ... ... */
```

3D FFT (part 2)

```
    p3dfft_setup(dims,nx,ny,nz,1,memsize);
    MPI_Barrier(MPI_COMM_WORLD);
    p3dfft_get_dims(istart,iend,isize,1);
    p3dfft_get_dims(fstart,fend,fsize,2);
    /* ……… */
    A = (double *) malloc(sizeof(double) * isize[0]*isize[1]*isize[2]);
    B = (double *) malloc(sizeof(double) * fsize[0]*fsize[1]*fsize[2]*2);
    C = (double *) malloc(sizeof(double) * isize[0]*isize[1]*isize[2]);
    MPI_Barrier(MPI_COMM_WORLD);
    /* … Initializzation of A … */
    p3dfft_ftran_r2c(A,B,op_f);
    /* ……… */
    Ntot = fsize[0]*fsize[1]* fsize[2]*2;
    for(i=0;i < Ntot;i++){
      B[i] *= nx*ny*nz;
    }
    MPI_Barrier(MPI_COMM_WORLD);
    /* ……… */
    p3dfft_btran_c2r(B,C,op_b);
    /* ……… */
    MPI_Barrier(MPI_COMM_WORLD);
    /* ……… */
    p3dfft_clean();
    MPI_Finalize();
}
```

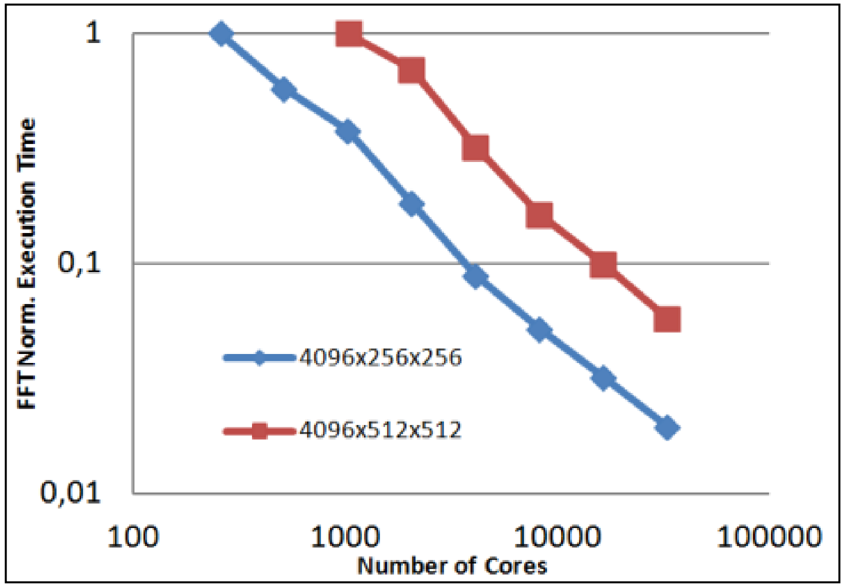
Performance results

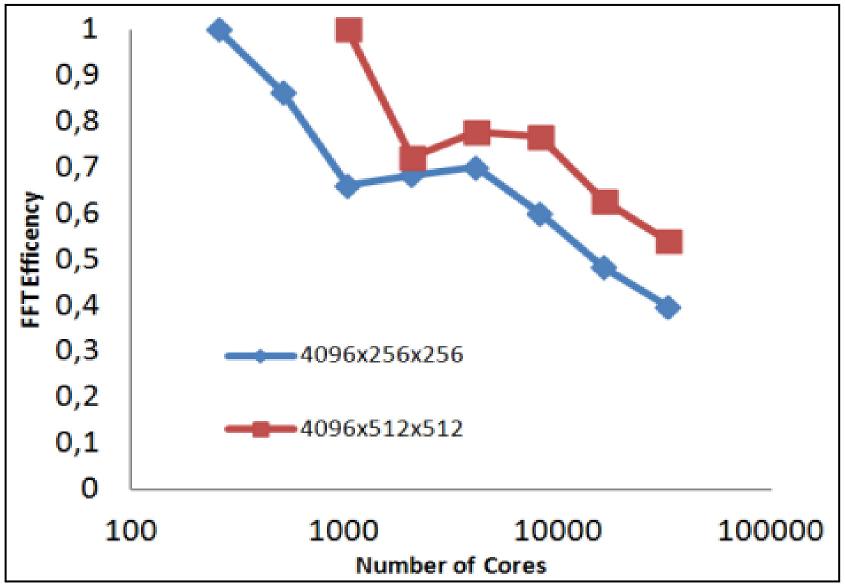
- Simple c2c FFT code based on 2Decomp&FFT library vs standard FFTW library
- Tested on FERMI up to 4096 cores
- Execution times



Performance Results

- FFT Kernel of BlowupNS CFD code based on 2Decomp&FFT library
- Tested on FERMI up to 32768 cores
- Two resolution tested:
  - 4096x256x256
  - 4096x512x512





Links:

FFTW Homepage : http://www.fftw.org/

Download FFTW-3: http://www.fftw.org/fftw-3.3.3.tar.gz

Manual FFTW-3: http://www.fftw.org/fftw3.pdf

2Decomp&FFT homepage: http://www.2decomp.org/

Dowload 2Decomp&FFT: http://www.2decomp.org/download/2decomp_fft-1.5.847.tar.gz

Online Manual 2Decomp&FFT: http://www.2decomp.org/decomp_api.html

P3DFFT homepage: http://code.google.com/p/p3dfft/

Dowload P3DFFT: http://p3dfft.googlecode.com/files/p3dfft-dist.2.6.1.tar

Manual P3DFFT:
http://p3dfft.googlecode.com/files/P3DFFT_User_Guide_2.6.1.pdf/

Thank You

For any other info,
Send an email to
m.guarrasi@cineca.it