Cineca TRAINING High Performance Computing 2017

Introduction to Numerical Libraries

Theory, Methods and Libraries.

Massimiliano Guarrasi, Nicola Spallanzani, Simone Bnà

(m.guarrasi, n.spallanzani, s.bn)@cineca.it

CINECA

WELCOME!!

The goal of this course is to show you how to get advantage of some of the most important numerical libraries for improving the performance of your HPC applications. We will focus on:

FFTW

FFTW, a subroutine library for computing the discrete Fourier transform (DFT) in one or more dimensions, of arbitrary input size, and of both real and complex data (as well as of even/odd data, i.e. the discrete cosine/sine transforms or DCT/DST)

2DECOMP&FFT

The 2DECOMP&FFT library is a software framework in Fortran to build large-scale parallel applications. It is designed for applications using three-dimensional structured mesh and spatially implicit numerical algorithms. At the foundation it implements a general-purpose 2D pencil decomposition for data distribution on distributed-memory platforms.

ScaLAPACK

A good number of libraries for Linear Algebra operations, including BLAS, LAPACK, SCALAPACK, MKL and MAGMA

PETSc

PETSc, a suite of data structures and routines for the scalable (parallel) solution of scientific applications modeled by partial differential equations

Part 0:

Before to start….

ABOUT THIS LECTURE

This first lecture won't be about numerical libraries…

Its purpose is to teach you the very basics of how to interact with CINECA's HPC cluster, where exercises will take place.

You will learn how to access to our system, how to compile, how to launch batch jobs, and everything you need in order to complete the exercises succesfully

…don't worry, it won't last long!! ;-)

WORK ENVIRONMENT

Workstation: User → corsi Password → corsi_2013!

Open a terminal: ***ssh username@login.galileo.cineca.it***

Once you're logged on a cluster, you are on your ***home*** *space.*

It is best suited for ***programming*** *environment (compilation, small debugging sessions…)*

Environment variable: $HOME

Another space you can access to is your ***scratch*** *space.*

It is best suited for ***production*** *environment (launch your jobs from there)*

Environment variable: $CINECA_SCRATCH

WARNING: is active a ***cleaning procedure****, that deletes your files older than 30 days!*

Use the command "cindata" for a quick briefing about your space occupancy

ACCOUNTING

*As an user, you have access to a limited number of CPU hours to spend. They are not assigned to users, but to **projects** and are shared between the users who are working on the same project (i.e. your research partners). Such projects are called **accounts** and are a different concept from your username.*

You can check the state of your account with the command "*saldo –b*", which tells you how many CPU hours you have already consumed for each account you're assigned at

(a more detailed report is provided by "*saldo –r*").

```
[amarani0@fen08 ~]$ saldo -b

---------------------------------------------------------------------------------------------------------
account              start         end         total        localCluster   totConsumed     totConsumed
                                            (local h)   Consumed(local h)    (local h)               %
---------------------------------------------------------------------------------------------------------
cin_staff         20110323    20200323    1000000000            30365762      30527993             3.1
cin_totview       20130123    20130213         50000                   0             0             0.0
train_sc32013     20130211    20130411       1250000               87458         87458             7.0
train_cnl12013    20130311    20130411        100000                   0             0             0.0
```

ACCOUNTING

The account provided for this course is “**train_cnl2016**” (you have to specify it on your job scripts). It expires Monday the 15th and is shared between all the students; there are plenty of hours for everybody, but don’t waste them!

MODULES

CINECA's work environment is organized with modules, a set of installed compilers, libraries, tools and applications available for all users.

"loading" a module means defining all the environment variables that point to the path of what you have loaded.

After a module is loaded, an environment variable is set of the form "MODULENAME_HOME"

```
[amarani0@fen07 ~]$ module load namd
[amarani0@fen07 ~]$ ls $NAMD_HOME
backup  flipbinpdb  flipdcd  namd2  namd2_plumed  namd2_remd  psfgen  sortreplicas
```

MODULE COMMANDS

> module available (or just “> module av”)
Shows the full list of the modules available in the profile you’re into, divided by: environment, libraries, compilers, tools, applications

> module load <module_name>

Loads a specific module

> module show <module_name>
Shows the environment variables set by a specific module

> module help <module_name>
Gets all informations about how to use a specific module

> module list
Shows the loaded modules

LIBRARY MODULES

The Numerical Libraries you will learn about and use during the course are also available via module system

```
------------ /cineca/prod/modulefiles/base/libraries ------------
blas/2007--bgq-xl--1.0(default)       libjpeg/8d--bgq-gnu--4.4.6
essl/5.1                              mass/7.3--bgq-xl--1.0
fftw/2.1.5--bgq-xl--1.0               mpi4py/1.3--bgq-gnu--4.4.6
fftw/3.3.2--bgq-xl--1.0               netcdf/4.1.3--bgq-xl--1.0
fftw/3.3.3--bgq-xl--1.0(default)      numpy/1.6.2--bgq-gnu--4.4.6
gsl/1.15--bgq-xl--1.0                 papi/4.4.0--bgq-gnu--4.4.6
hdf5/1.8.9_par--bgq-xl--1.0           petsc/3.3-p2--bgq-xl--1.0
hdf5/1.8.9_ser--bgq-xl--1.0           scalapack/2.0.2--bgq-xl--1.0(default)
lapack/3.4.1--bgq-xl--1.0(default)    szip/2.1--bgq-xl--1.0
libint/2.0--bgq-xl--1.0(default)      zlib/1.2.7--bgq-gnu--4.4.6
```

Once loaded, they set the environment variable LIBRARYNAME_LIB .

If needed, there is also LIBRARYNAME_INC for the header files.

More on that during the course…

COMPILING ON GALILEO

In EURORA you can choose between three different compiler families: ***gnu***, ***intel*** *and* ***pgi***

You can take a look at the versions available with "*module av*" and then load the module you want.

Defaults are: gnu 4.9.2, intel xe 2015, pgi 16.3

module load intel   # loads default intel compilers suite

module load intel/pe-xe-2016--binary #loads specific compilers suite

Compiler's name	**GNU**	**INTEL**	**PGI**
Fortran	gfortran	ifortran	pgf77
C	gcc	icc	pgcc
C++	g++	icpc	pgCC

Get a list of the compilers flags with the command *man*

PARALLEL COMPILING ON GALILEO

For parallel programming, two families of compilers are available: ***openmpi*** *(recommended) and* ***intelmpi*** *.*

There are different versions of openmpi, depending on which compiler has been used for creating them. Default is openmpi/1.8.4--gnu--4.9.2

module load autoload openmpi # loads default openmpi compilers suite

module load autoload openmpi/1.8.4--intel--cs-xe-2015--binary # loads specific compilers suite

Warning: mpi compiler needs to be loaded after the corresponding basic compiler suite. You can load both compilers at the same time with "autoload"

```
[cin0955a@node342 ~]$ module load openmpi
WARNING: openmpi/1.4.4--gnu--4.5.2 cannot be loaded due to missing prereq.
HINT: the following modules must be loaded first: gnu/4.5.2
[cin0955a@node342 ~]$ module load autoload openmpi
### auto-loading modules gnu/4.5.2
```

If another type of compiler was previously loaded, you may get a "conflict error". Unload the previous module with "module unload"

COMPILING WITH LIBRARIES

Once you have loaded the proper library module, specify its linking by adding a reference in the compiling command.

Two ways to link a library:

-L$LIBRARY_LIB **-l**name *– or –* **-L**$LIBRARY_LIB/**lib**name**.a**

For some libraries, it may be necessary to include the header path

-I$LIBRARY_INC

```
$ mpicc -I$HDF5_INC input.c -L$HDF5_LIB -lhdf5 \
        -L$SZIP_LIB -lsz -LZLIB_LIB -lz
$ mpicc -I$HDF5_INC input.c -L$HDF5_LIB/libhdf5.a \
        -L$SZIP_LIB/libsz.a -L$ZLIB_LIB/libz.a
```

COMPILING WITH LIBRARIES

*Galileo lets you choose between **static** and **dynamic linking**, with the latter one as a default.*

Static linking means that the library references are resolved at compile time, so the necessary functions and variables are already contained in the executable produced. It means a bigger executable but no need for linking the library paths at runtime.

Dynamic linking means that the library references are resolved at run time, so the executable searches for them in the paths provided. It means a lighter executable and no need to recompile the program after every library update, but need a lot of environment variables to define at runtime.

For enabling static linking: -static (gnu), -intel-static (intel), -Bstatic (pgi)

LAUNCHING JOBS

Now that we have our GALILEO program, it's time to learn how to prepare a job for its execution

GALILEO uses a scheduler called **PBS**.

The job script scheme is:

- #!/bin/bash
- PBS keywords
- variables environment
- execution line

PBS KEYWORDS


```
#PBS –N jobname   # name of the job
#PBS -o job.out   # output file
#PBS -e job.err   # error file
#PBS -l select=1:ncpus=8:mpiprocs=2:mem=8GB   # resources requested *
#PBS -l walltime=1:00:00       # max 24h, depending on the queue
#PBS -q parallel        # queue desired
#PBS -A <my_account>      # name of the account
#PBS -m abe      # mail events
#PBS -M <email1,email2,...>      # list of emails
```

* select = number of chunks (not exactly the nodes) requested

ncpus = number of cpus per chunk requested

mpiprocs = number of mpi tasks per chunk

mem = amount of RAM per chunk

For pure MPI jobs, ncpus = mpiprocs

For OpenMP jobs, mpiprocs < ncpus

KEYWORDS SPECIFIC FOR THE COURSE

#PBS -A train_cnl2016 *# your account name*

#PBS -q R1620674 *# special queue reserved for you*

#PBS -W group_list=train_cnl2016 *# needed for entering in private queue*

“R1620674” queue is a reserved queue composed by a node equipped with 2 GPUs and a node equipped with 2 MICs.

In order to grant fast runs to all the students, we ask you to not launch too big jobs (you won’t need them, anyways). Please don’t request more than half a node at a time!

ENVIRONMENT SETUP AND EXECUTION LINE

The command that "split" the executable on the processes is mpirun:

`mpirun -n 8 ./myexe arg_1 arg_2`

-n *is the number of cores you want to use.*

In order to use mpirun, openmpi (or intelmpi) has to be loaded. Also, if you linked dynamically, you have to remember to load every library module you need.

The environment setting usually start with "cd $PBS_O_WORKDIR". That's because by default you are launching on your home space and may not find the executable you want to launch.

$PBS_O_WORKDIR points at the folder you're submitting the job from.

Galileo JOB SCRIPT EXAMPLE


```
#!/bin/bash
#PBS -l walltime=0:10:00
#PBS -l select=1:ncpus=4:mpiprocs=4:mem=4GB
#PBS -o job.out
#PBS -e job.err
#PBS -q R1620674
#PBS -A train_cnl2016
#PBS -W group_list=train_cnl2016

cd $PBS_O_WORKDIR
module load autoload openmpi
module load somelibrary

mpirun ./myprogram < myinput > myoutput
```

For GPU accelerators
add this to the select line:
:ngpus=2

For MIC accelerators
add this to the select line:
:nmics=2

PBS COMMANDS

qsub

qsub <job_script>

Your job will be submitted to the PBS scheduler and executed when there will be nodes available (according to your priority and the queue you requested)

qstat

qstat -u $USER

Shows the list of all your scheduled jobs, along with their status (idle, running, closing,…)

Also, shows you the job id required for other qstat options

PBS COMMANDS

qstat -f <job_id>

Provides a long list of informations for the job requested.
In particular, if your job isn’t running yet, you'll be notified about its estimated start time or, if you made an error on the job script, you will learn that the job won’t ever start

qdel

qdel <job_id>

Removes the job from the scheduler, killing it

USEFUL DOCUMENTATION

Exercises on Numerical Librearies:

Slides:

Bologna/HPC_Numerical_Libraries/

Part 1:

Introduction to Numerical 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**

Discrete Fourier Transforms (DFT)

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

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

$$F_k = \sum_{j=0}^{N-1} e^{2\pi ijk/N} f_j$$

$$= \sum_{j=0}^{N/2-1} e^{2\pi ik(2j)/N} f_{2j} + \sum_{j=0}^{N/2-1} e^{2\pi ik(2j+1)/N} f_{2j+1}$$

$$= \sum_{j=0}^{N/2-1} e^{2\pi ikj/(N/2)} f_{2j} + W^k \sum_{j=0}^{N/2-1} e^{2\pi ikj/(N/2)} f_{2j+1}$$

$$= F_k^e + W^k F_k^o$$

Fast Fourier Transform (FFT)

$$F_k = F_k^e + W^k F_k^o$$

exp(2πi/N)

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 fn

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

Scale like N*logN (binary tree)

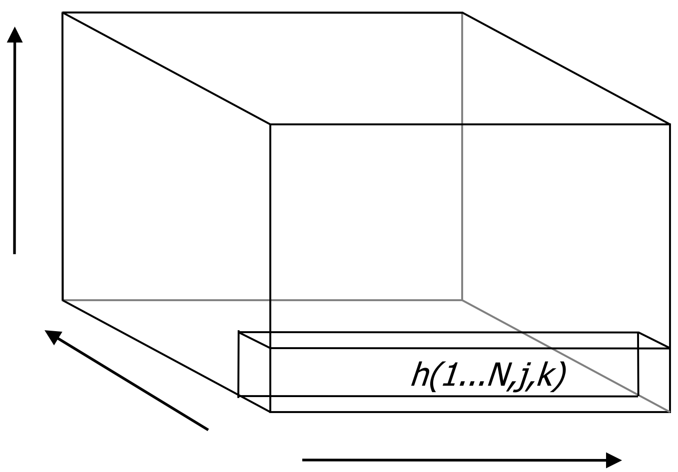
Parallel Domain Decomposition

How to compute a FFT on a distributed memory system

- 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***

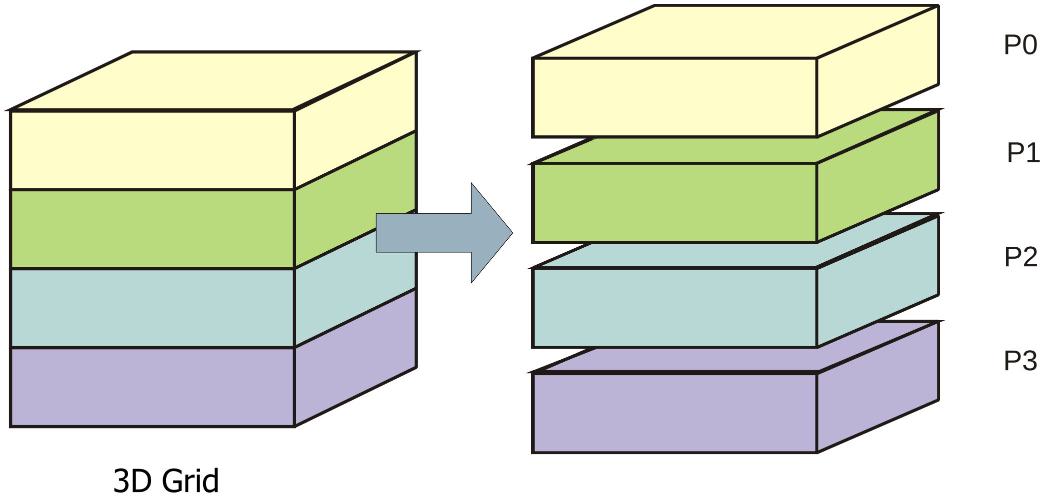
Apply FFT to h(1...N,j,k)

*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)

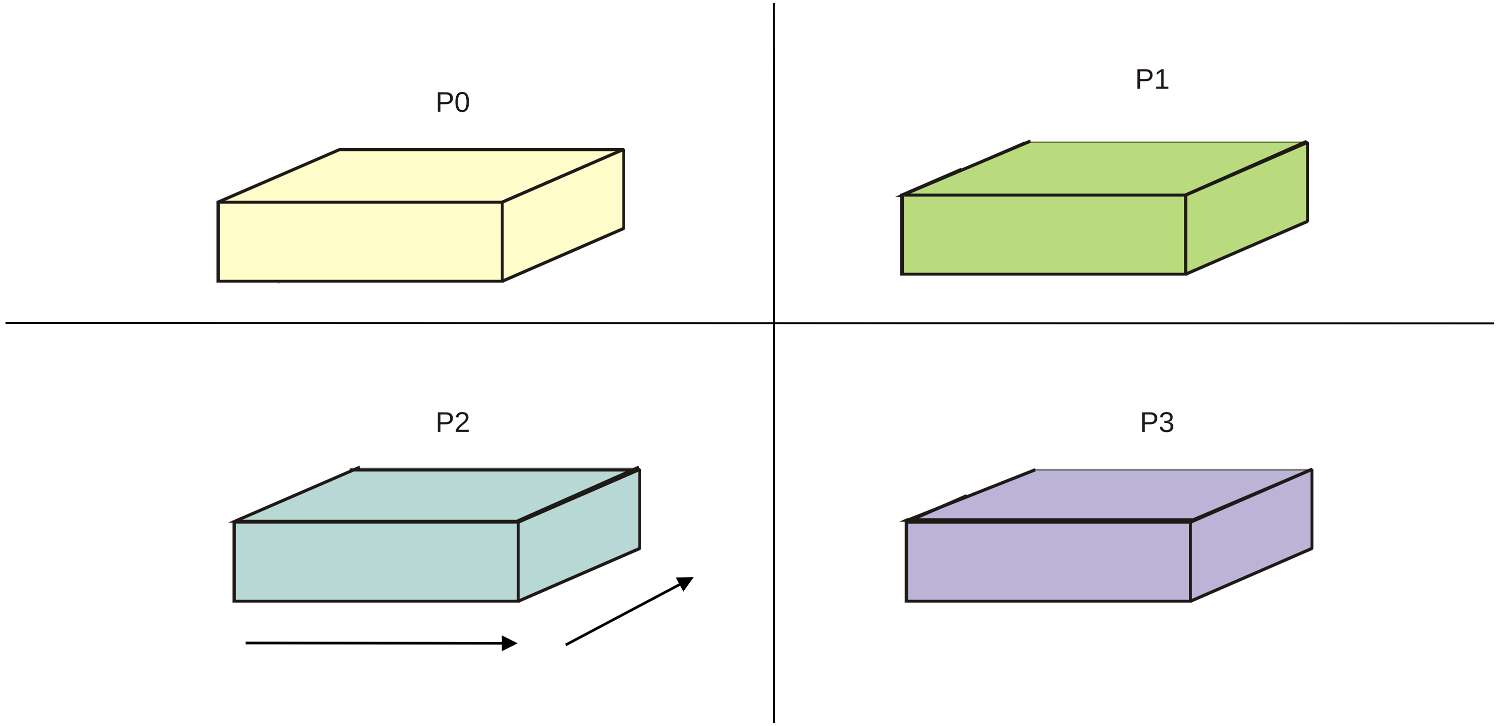
Parallel FFT Data Distribution



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

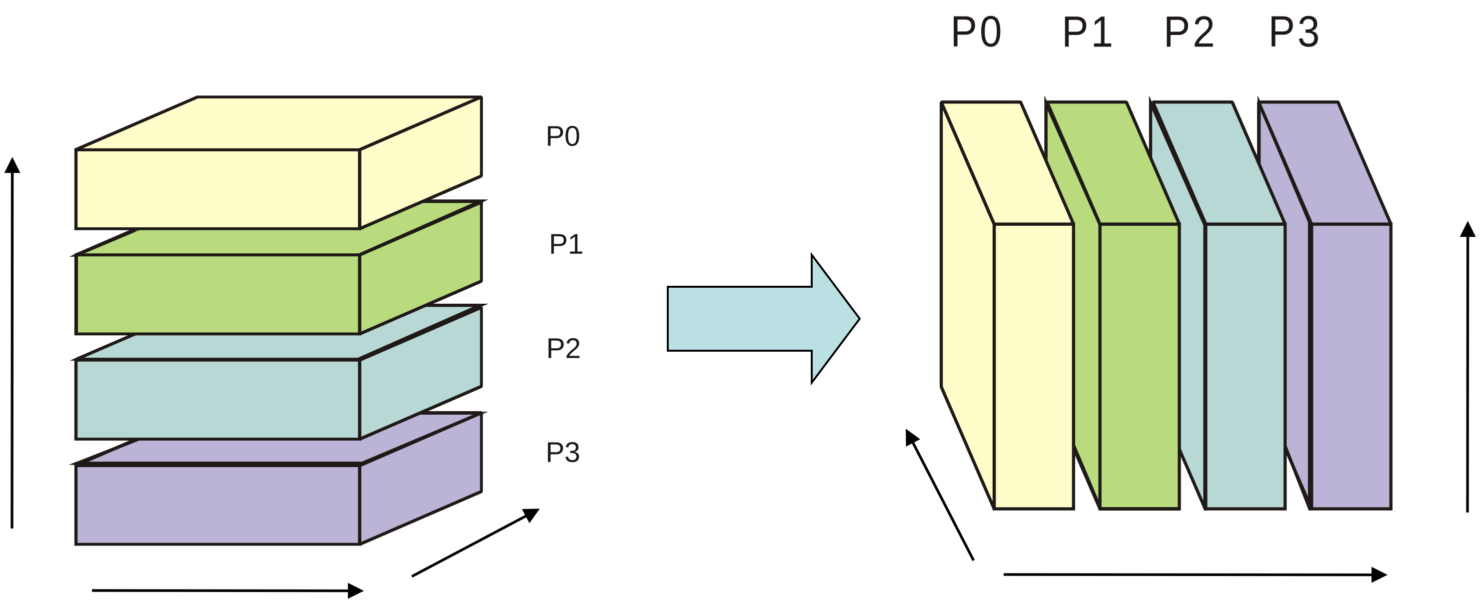
This is know as "Slab Decomposition" or 1D Decomposition

Transform along x and y



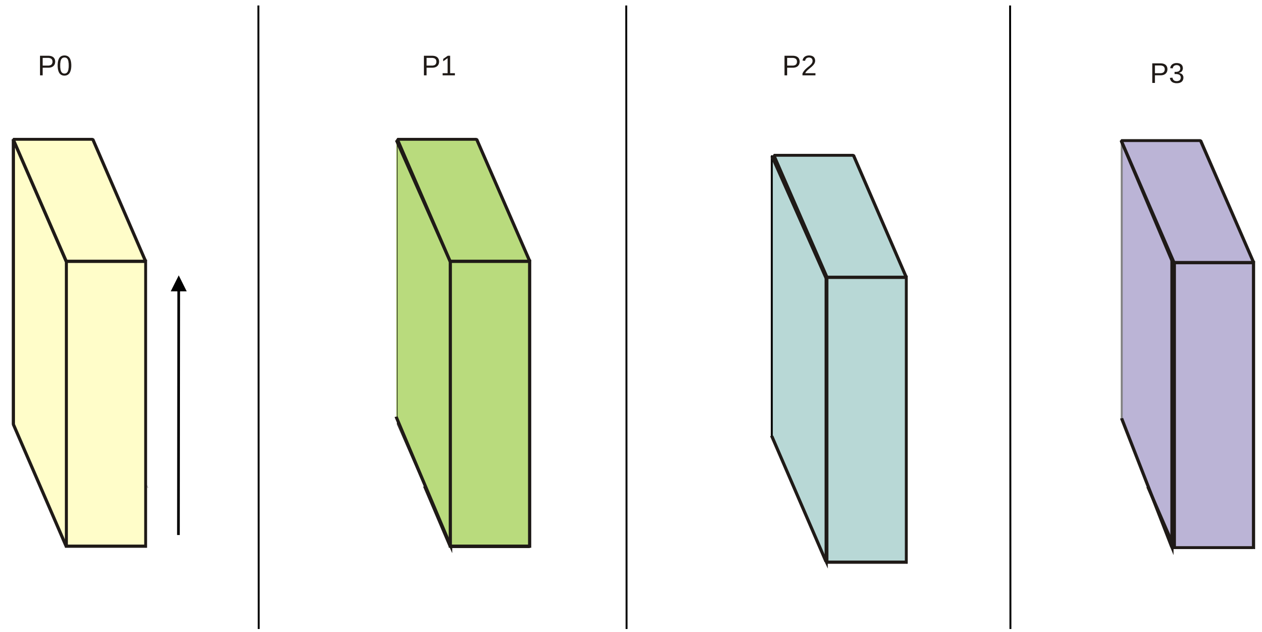
each processor trasform its own sub-grid along the x and y independently of the other

Data redistribution involving x and z



The data are now distributed along x

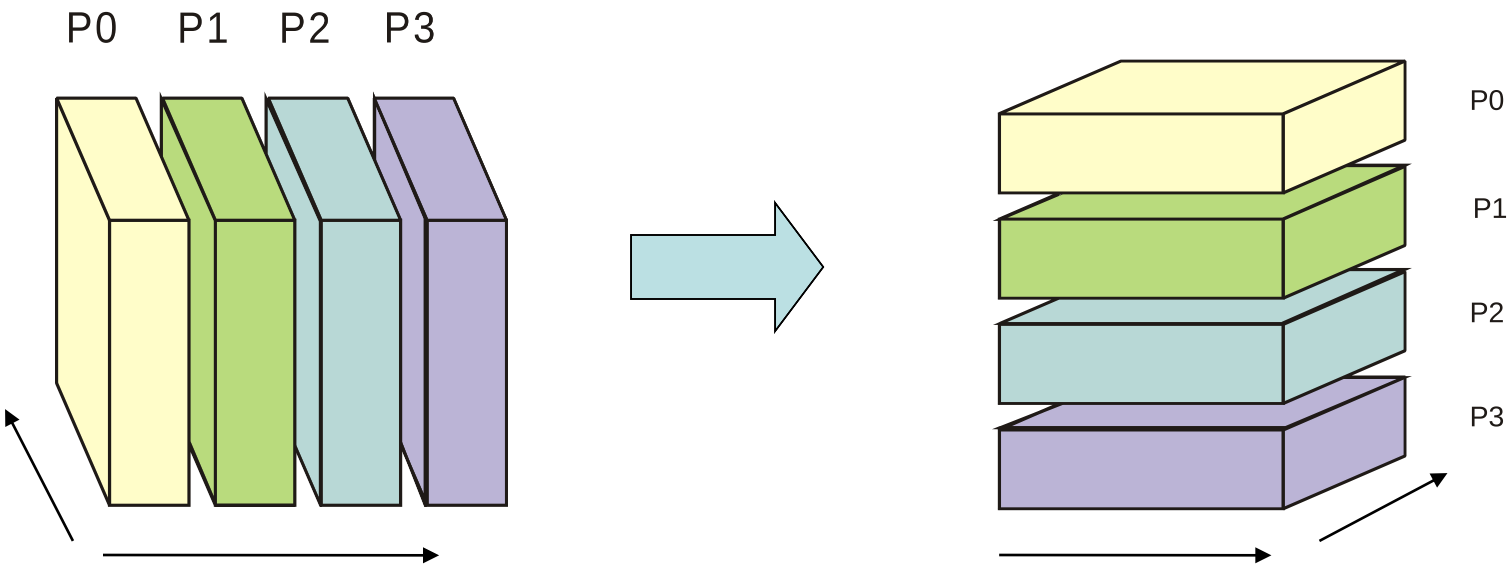
FFT along z



each processor transform its own sub-grid

along the z dimension independently of the other

Data are re-distributed, back from x to z



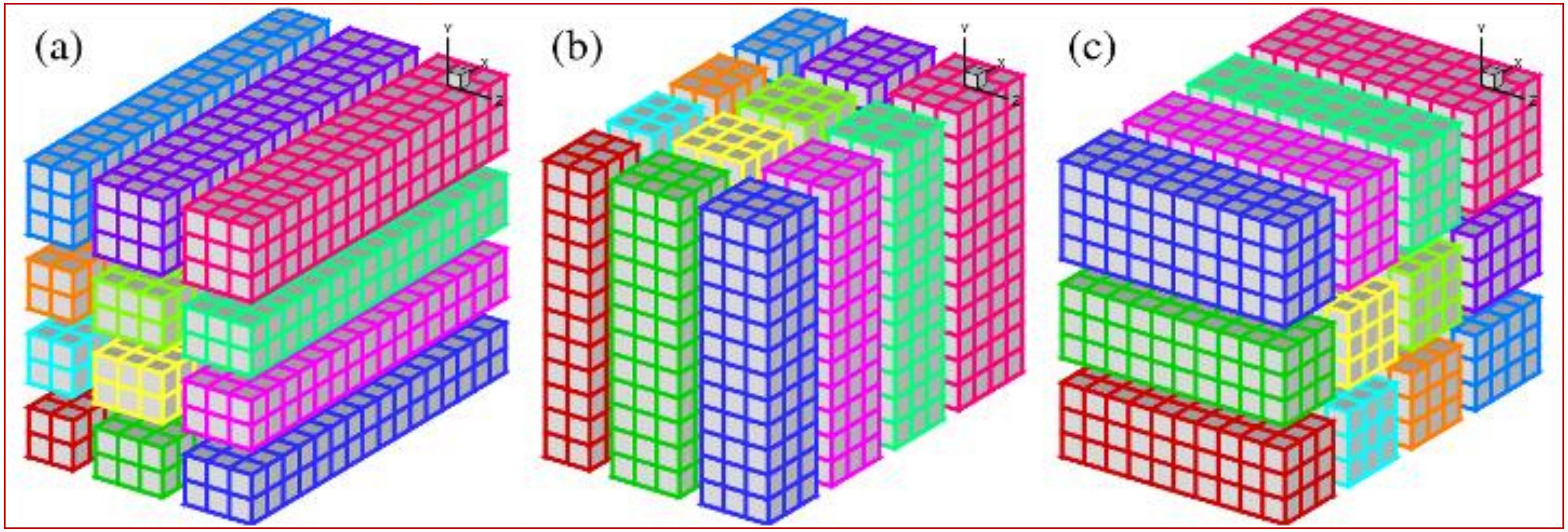
The 3D array now has the original layout, but each element

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)

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?

- Module Loading:

  module load autoload fftw/3.3.5--intelmpi--2017—binary

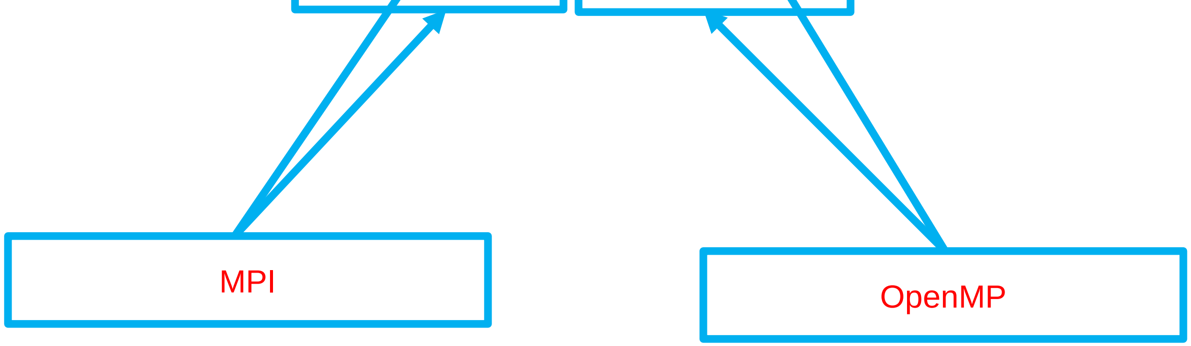
Including header:

- -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
```

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 - 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 - 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

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

Some useful papers

Cineca TRAINING High Performance Computing 2017

- 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.

CINECA

Cineca
TRAINING
High Performance
Computing 2017

Part 3:

Introduction to PETSc

(Portable, Extensible Toolkit for Scientific Computation)

CINECA

PETSc main features

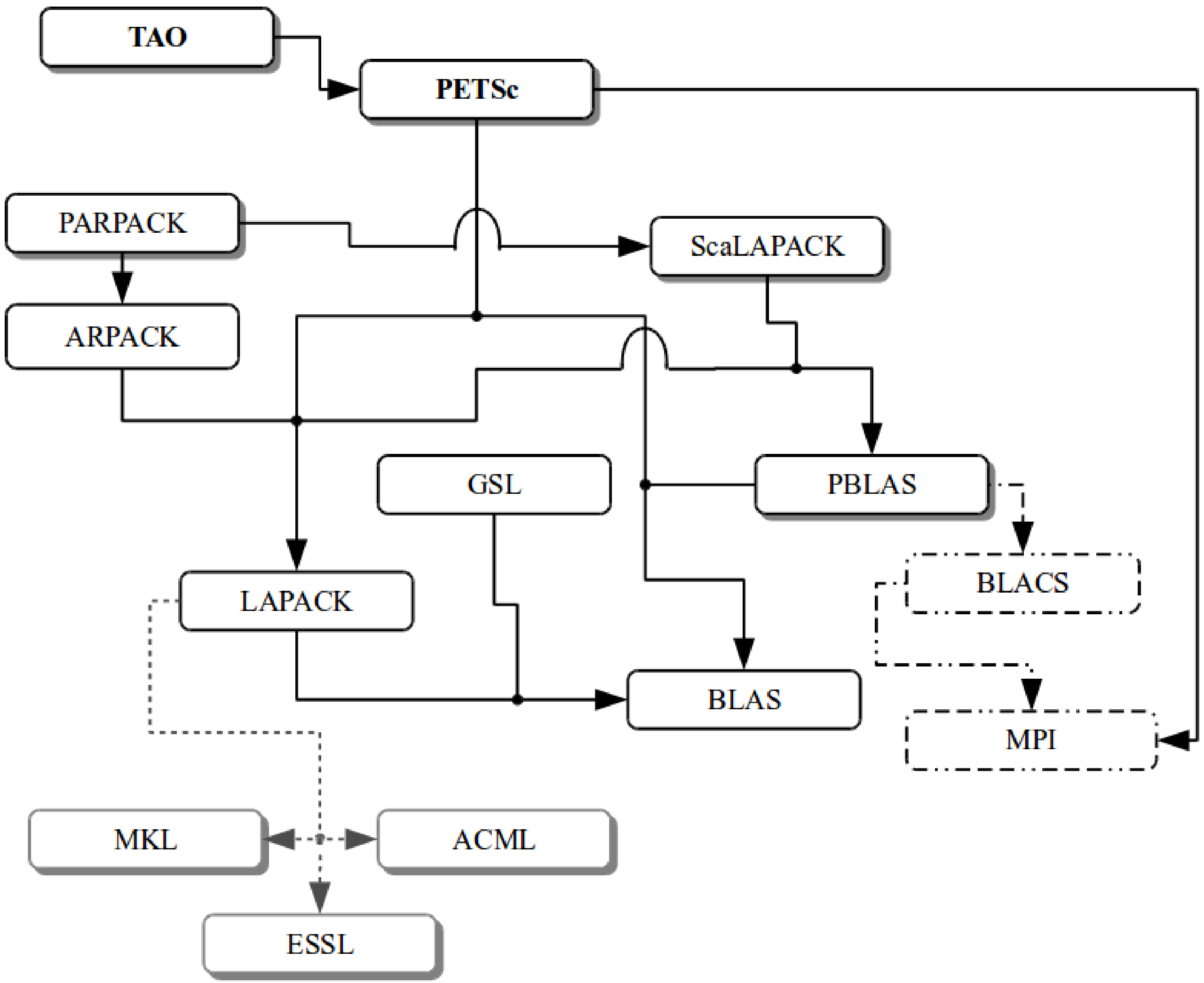
PETSc – Portable, Extensible Toolkit for Scientific Computation

Is a suite of data structures and routines for the scalable (parallel) solution of scientific applications mainly modelled by partial differential equations.

- **ANL** – Argonne National Laboratory
- Begun September **1991**
- Uses the **MPI** standard for all message-passing communication
- **C**, **Fortran**, and **C++**
- Consists of a variety of libraries; each library manipulates a particular family of **objects** and the operations one would like to perform on the objects
- PETSc has been used for modelling in all of these **areas**:

  Acoustics, Aerodynamics, Air Pollution, Arterial Flow, Brain Surgery, Cancer Surgery and Treatment, Cardiology, Combustion, Corrosion, Earth Quakes, Economics, Fission, Fusion, Magnetic Films, Material Science, Medical Imaging, Ocean Dynamics, PageRank, Polymer Injection Molding, Seismology, Semiconductors, ...

Relationship between libraries



PETSc programming model

Goals

- Portable
- Performance
- Scalable parallelism

Approach

- Variety of libraries
  - Objects (One interface – One or more implementations)
  - Operations on the objects

Benefit

- Code reuse
- Flexibility
- Hide within objects the details of the communication

PETSc numerical component

Nonlinear Solvers		
Newton-based Methods		Other
Line Search	Trust Region	

Time Steppers			
Euler	Backward Euler	Pseudo-Time Stepping	Other

Krylov Subspace Methods							
GMRES	CG	CGS	Bi-CG-Stab	TFQMR	Richardson	Chebychev	Other

Preconditioners						
Additive Schwarz	Block Jacobi	Jacobi	ILU	ICC	LU (sequential only)	Other

Matrices				
Compressed Sparse Row (AIJ)	Block Compressed Sparse Row (BAIJ)	Block Diagonal (BDiag)	Dense	Other

Vectors

Index Sets			
Indices	Block Indices	Stride	Other

Writing PETSc programs: initialization and finalization


```
PetscInitialize(int *argc, char ***args, const char
file[], const char help[])
```

- Setup static data and services
- Setup MPI if it is not already

```
PetscFinalize()
```

- Calculates logging summary
- Finalize MPI (if `PetscInitialize()` began MPI)
- Shutdown and release resources

Example: hello.c


```
#include "petsc.h"

#undef __FUNCT__
#define __FUNCT__ "main"
int main(int argc,char **args)
{
   PetscErrorCode ierr;
   PetscMPIInt    rank;

   PetscInitialize(&argc, &args,(char *)0, PETSC_NULL);

   MPI_Comm_rank(PETSC_COMM_WORLD, &rank);
   ierr = PetscPrintf(PETSC_COMM_SELF,"Hello by procs %d!\n",
                      rank); CHKERRQ(ierr);

   ierr = PetscFinalize();
   return 0;
}
```

Example: hello.F90


```
program main

integer :: ierr, rank
character(len=6)  :: num
character(len=30) :: hello

#include "finclude/petsc.h"

call PetscInitialize( PETSC_NULL_CHARACTER,ierr )

call MPI_Comm_rank( PETSC_COMM_WORLD, rank, ierr )
write(num,*) rank
hello = 'Hello by process '//num
call PetscPrintf( PETSC_COMM_SELF, hello//achar(10), ierr )

call PetscFinalize(ierr)

end program
```

Vec and Mat

Vectors

What are PETSc vectors?

- Fundamental objects for storing field solutions, right-hand sides, etc.
- Each process locally owns a subvector of contiguously numbered global indices

Features

- Has a direct interface to the values
- Supports all vector space operations
  - `VecDot()`, `VecNorm()`, `VecScale()`, …
- Also unusual ops, e.g. `VecSqrt()`, `VecInverse()`
- Automatic communication during assembly
- Customizable communication (scatters)

Creating a vector

- `VecCreate(MPI_Comm comm, Vec *v)`
  - Vector types: sequential and parallel (MPI based)
  - Automatically generates the appropriate vector type (sequential or parallel) over all processes in `comm`
- `VecSetSizes(Vec v, int m, int M)`
  - Sets the local and global sizes, and checks to determine compatibility
- `VecSetFromOptions(Vec v)`
  - Configures the vector from the options database

- `VecDuplicate(Vec old, Vec *new)`
  - Does not copy the values

Vector basic operations


```
VecGetSize(Vec v, int *size)
VecGetLocalSize(Vec v, int *size)
VecGetOwnershipRange(Vec vec, int *low, int *high)

VecView(Vec x, PetscViewer v)

VecCopy(Vec x, Vec y)

VecSet(Vec x, PetscScalar value)
VecSetValues(Vec x, int n, int *idx,
             PetscScalar *v, INSERT_VALUES)

VecDestroy(Vec *x)
```

Vector assembly

Once all of the values have been inserted with `VecSetValues()`,

one must call

`VecAssemblyBegin(Vec x)`

`VecAssemblyEnd(Vec x)`

to perform any needed message passing of nonlocal components.

A **three step process**

- Each process tells PETSc what values to set or add to a vector component. Once *all* values provided,
- begin communication between processes to ensure that values end up where needed (allow other operations, such as some computation, to proceed).
- Complete the communication

Vector – Example 1


```
VecGetSize(x, &N);  /* Global size */
MPI_Comm_rank(PETSC_COMM_WORLD, &rank);

if (rank == 0) {
  for (i=0; i<N; i++)
    VecSetValues(x, 1, &i, &i, INSERT_VALUES);
}

/* These two routines ensure that the data is
distributed to the other processes */
VecAssemblyBegin(x);
VecAssemblyEnd(x);
```

Vector – Example 2 (Do it in Parallel!)


```
VecGetOwnershipRange(x, &low, &high);

for (i=low; i<high; i++)
   VecSetValues(x, 1, &i, &i, INSERT_VALUES);

/* These routines must be called in case some other
process contributed a value owned by another process
*/
VecAssemblyBegin(x);
VecAssemblyEnd(x);
```

Numerical vector operations

Function Name	Operation
VecAXPY(Vec y,PetscScalar a,Vec x);	$y = y + a * x$
VecAYPX(Vec y,PetscScalar a,Vec x);	$y = x + a * y$
VecWAXPY(Vec w,PetscScalar a,Vec x,Vec y);	$w = a * x + y$
VecAXPBY(Vec y,PetscScalar a,PetscScalar b,Vec x);	$y = a * x + b * y$
VecScale(Vec x, PetscScalar a);	$x = a * x$
VecDot(Vec x, Vec y, PetscScalar *r);	$r = \bar{x}' * y$
VecTDot(Vec x, Vec y, PetscScalar *r);	$r = x' * y$
VecNorm(Vec x,NormType type, PetscReal *r);	$r = \|\|x\|\|_{type}$
VecSum(Vec x, PetscScalar *r);	$r = \sum x_i$
VecCopy(Vec x, Vec y);	$y = x$
VecSwap(Vec x, Vec y);	$y = x$ while $x = y$
VecPointwiseMult(Vec w,Vec x,Vec y);	$w_i = x_i * y_i$
VecPointwiseDivide(Vec w,Vec x,Vec y);	$w_i = x_i / y_i$
VecMDot(Vec x,int n,Vec y[],PetscScalar *r);	$r[i] = \bar{x}' * y[i]$
VecMTDot(Vec x,int n,Vec y[],PetscScalar *r);	$r[i] = x' * y[i]$
VecMAXPY(Vec y,int n, PetscScalar *a, Vec x[]);	$y = y + \sum_i a_i * x[i]$
VecMax(Vec x, int *idx, PetscReal *r);	$r = \max x_i$
VecMin(Vec x, int *idx, PetscReal *r);	$r = \min x_i$
VecAbs(Vec x);	$x_i = \|x_i\|$
VecReciprocal(Vec x);	$x_i = 1/x_i$
VecShift(Vec x,PetscScalar s);	$x_i = s + x_i$
VecSet(Vec x,PetscScalar alpha);	$x_i = \alpha$

Working with local vector

It is sometimes more efficient to directly access the storage for the local part of a PETSc `Vec`.

- E.g., for finite difference computations involving elements of the vector

`VecGetArray(Vec, double *[])`

- Access the local storage

`VecRestoreArray(Vec, double *[])`

- You must return the array to PETSc when you finish

Allows PETSc to handle data structure conversions

- For most common uses, these routines are inexpensive and do *not* involve a copy of the vector.

Vector – Example 3


```
Vec vec;
Double *avec;
[…]
VecCreate(PETSC_COMM_WORLD,&vec);
VecSetSizes(vec,PETSC_DECIDE,n);
VecSetFromOptions(vec);
[…]
VecGetArray(vec, &avec);

/* compute with avec directly, e.g.: */
PetscPrintf(PETSC_COMM_WORLD,
            "First element of local array of vec in
             each process is %f\n", avec[0] );

VecRestoreArray(vec, &avec);
```

2_petsc_vec.c


```
[…]
PetscViewer viewer_fd;
Vec va;
[…]
ierr = PetscViewerBinaryOpen(PETSC_COMM_WORLD, "data/va_200.bin",
                             FILE_MODE_READ, &viewer_fd );CHKERRQ(ierr);
ierr = VecCreate(PETSC_COMM_WORLD, &va);                   CHKERRQ(ierr);
ierr = VecLoad(va, viewer_fd);                             CHKERRQ(ierr);
ierr = PetscViewerDestroy(&viewer_fd);                     CHKERRQ(ierr);
CHKMEMQ;

VecView(va, PETSC_VIEWER_STDOUT_WORLD);

VecGetSize(va, &size_global);                          CHKERRQ(ierr);
VecGetLocalSize(va, &size_local);                      CHKERRQ(ierr);
VecGetOwnershipRange(va, &low_idx, &high_idx);         CHKERRQ(ierr);
[…]
VecDestroy(&va);
[…]
```

Matrices

What are PETSc matrices?

- Fundamental objects for storing linear operators
- Each process locally owns a submatrix of contiguous rows

Features

- Supports many data types
  - AIJ, Block AIJ, Symmetric AIJ, Block Diagonal, etc.
- Supports structures for many packages
  - Spooles, MUMPS, SuperLU, UMFPack, DSCPack
- A matrix is defined by its interface, the operations that you can perform with it, not by its data structure

Creating a matrix

- **MatCreate(MPI_Comm comm, Mat *A)**
  - Matrices types: sequential and parallel (MPI based).
  - Automatically generates the appropriate matrix type (sequential or parallel) over all processes in comm.

  MatSetSizes(Mat A, int m, int n, int M, int N)
  - Sets the local and global sizes, and checks to determine compatibility

  MatSetFromOptions(Mat A)
  - Configures the matrix from the options database.

- **MatDuplicate(Mat B, MatDuplicateOption op, Mat *A)**
  - Duplicates a matrix including the non-zero structure.

Matrix basic operations


```
MatView(Mat A, PetscViewer v)

MatGetOwnershipRange(Mat A, PetscInt *m, PetscInt* n)
MatGetOwnershipRanges(Mat A, const PetscInt **ranges)
```

– Each process locally owns a submatrix of contiguously numbered global rows.

```
MatGetSize(Mat A, PetscInt *m, PetscInt* n)

MatSetValues(Mat A, int m, const int idxm[],
           int n, const int idxn[],
           const PetscScalar values[],
           INSERT_VALUES| ADD_VALUES)
```

Matrix assembly

Once all of the values have been inserted with `MatSetValues()`, one must call

```
MatAssemblyBegin(Mat A, MatAssemblyType type)
MatAssemblyEnd(Mat A, MatAssemblyType type)
```

to perform any needed message passing of nonlocal components.

Matrix – Example 1


```
Mat     A;
int     column[3], i;
double value[3];
[…]
MatCreate(PETSC_COMM_WORLD,PETSC_DECIDE,PETSC_DECIDE,n,n,&A);
MatSetFromOptions(A);

value[0] = -1.0; value[1] = 2.0; value[2] = -1.0;
if (rank == 0) {
    for (i=1; i<n-2; i++) {
        column[0] = i-1; column[1] = i; column[2] = i+1;
        MatSetValues(A,1,&i,3,column,value,INSERT_VALUES);
}}
MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);
MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);
```

Matrix – Example 2


```
Mat     A;
int     column[3], i, start, end, istart, iend;
double value[3];
[…]
MatCreate(PETSC_COMM_WORLD,PETSC_DECIDE,PETSC_DECIDE,n,n,&A);
MatSetFromOptions(A);
MatGetOwnershipRange(A,&istart,&iend);

value[0] = -1.0; value[1] = 2.0; value[2] = -1.0;
for (i=istart; i<iend; i++) {
    column[0] = i-1; column[1] = i; column[2] = i+1;
    MatSetValues(A,1,&i,3,column,value,INSERT_VALUES);
}
MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);
MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);
```

Numerical matrix operations

Function Name	Operation
MatAXPY(Mat Y, PetscScalar a,Mat X,MatStructure);	$Y = Y + a * X$
MatMult(Mat A,Vec x, Vec y);	$y = A * x$
MatMultAdd(Mat A,Vec x, Vec y,Vec z);	$z = y + A * x$
MatMultTranspose(Mat A,Vec x, Vec y);	$y = A^T * x$
MatMultTransposeAdd(Mat A,Vec x, Vec y,Vec z);	$z = y + A^T * x$
MatNorm(Mat A,NormType type, double *r);	$r = \|\|A\|\|_{type}$
MatDiagonalScale(Mat A,Vec l,Vec r);	$A = \text{diag}(l) * A * \text{diag}(r)$
MatScale(Mat A,PetscScalar a);	$A = a * A$
MatConvert(Mat A,MatType type,Mat *B);	$B = A$
MatCopy(Mat A,Mat B,MatStructure);	$B = A$
MatGetDiagonal(Mat A,Vec x);	$x = \text{diag}(A)$
MatTranspose(Mat A,MatReuse,Mat* B);	$B = A^T$
MatZeroEntries(Mat A);	$A = 0$
MatShift(Mat Y,PetscScalar a);	$Y = Y + a * I$

Matrix memory pre-allocation

Preallocation of memory is critical for achieving **good performance** during matrix assembly, as this reduces the number of allocations and copies required.

PETSc sparse matrices are dynamic data structures.

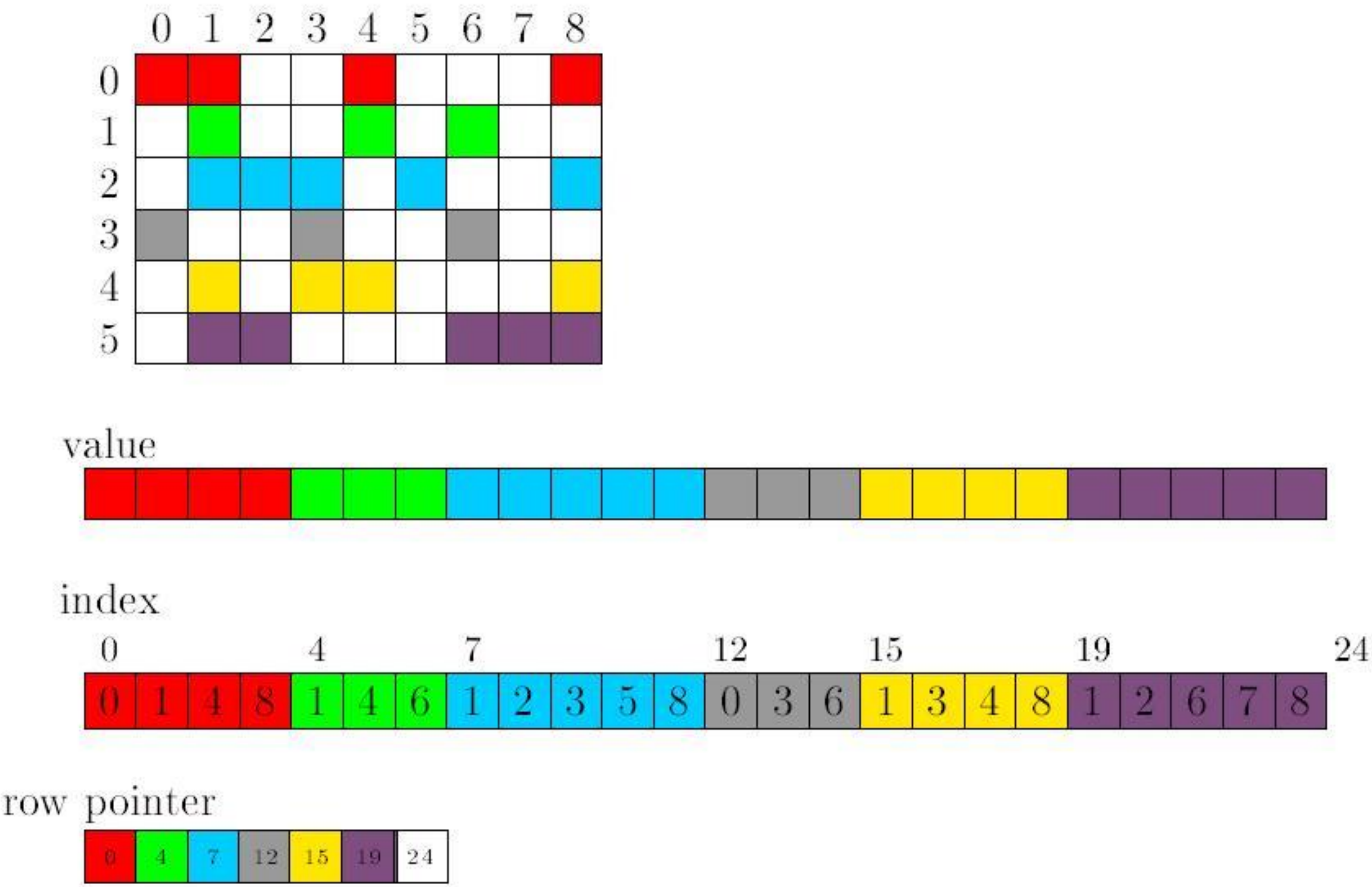
Can **add additional nonzeros freely**.

Dynamically adding many nonzeros

- requires additional memory allocations
- requires copies
- can kill performance

Memory pre-allocation provides the freedom of dynamic data structures plus good performance

Matrix AIJ format



Pre-allocation of sequential sparse matrix (1/2)


```
MatCreateSeqAIJ(PETSC COMM SELF, int m, int n,
                int nz, int *nnz, Mat *A)
```

1. If (`nz` == 0 && `nnz` == `PETSC_NULL`)

→ PETSc to control all matrix memory allocation

1. Set `nz` = <*value*>

   → Specify the expected number of nonzeros for each row.

   – Fine if the number of nonzeros per row is roughly the same throughout the matrix

   – Quick and easy first step for pre-allocation

Pre-allocation of sequential sparse matrix (2/2)


```
MatCreateSeqAIJ(PETSC COMM SELF, int m, int n,
                int nz, int *nnz, Mat *A)
```

3. Set `nnz[0]` = *<nonzeros in row 0>*

   ...

   `nnz[m]` = *<nonzeros in row m>*

   → indicate (nearly) the exact number of elements intended for the various rows

If one **underestimates** the actual number of nonzeros in a given row, then during the assembly process PETSc will **automatically allocate additional needed space**.

This extra memory allocation can **slow** the computation!

Parallel sparse matrices

Each process locally owns a submatrix of contiguously numbered global rows.

Each submatrix consists of **diagonal** and **off-diagonal** parts.

P0

P1

P2

Pre-allocation of parallel sparse matrix (1/2)

```
MatCreateMPIAIJ(MPI_Comm comm,
                int m, int n, int M, int N,
                int d_nz, int *d_nnz,
                int o_nz, int *o_nnz,
                Mat *A)
```

1. If (**d_nz** == **o_nz** == 0 && **d_nnz** == **o_nnz** == **PETSC_NULL**)

→ PETSc to control dynamic allocation of matrix memory space

1. Set **d_nz** = <*value*> and **o_nz** = <*value*>

   → Specify nonzero information for the diagonal (**d_nz**) and off-diagonal (**o_nz**) parts of the matrix.

Pre-allocation of parallel sparse matrix (2/2)


```
MatCreateMPIAIJ(MPI Comm comm,
                int m, int n, int M, int N,
                int d_nz, int *d_nnz,
                int o_nz, int *o_nnz,
                Mat *A)
```

3. Set `d_nnz[0]` = <*nonzeros in row 0, diagonal part*>
   ...
   `d_nnz[m]` = <*nonzeros in row m, diagonal part* >
   `o_nnz[0]` = <*nonzeros in row 0, off-diagonal part*>
   ...
   `o_nnz[m]` = <*nonzeros in row m , off-diagonal part* >

   → Specify nonzero information for the diagonal (`d_nnz`) and off-diagonal (`o_nnz`) parts of the matrix.

Verifying Predictions (1/2)


```
MatGetInfo(Mat mat, MatInfoType flag, MatInfo *info)
```

Or

Runtime option: `-info -mat_view_info`

```
typedef struct {
    PetscLogDouble block_size;
    PetscLogDouble nz_allocated, nz_used, nz_unneeded;
    PetscLogDouble memory;
    PetscLogDouble assemblies;
    PetscLogDouble mallocs;
    PetscLogDouble fill_ratio_given, fill_ratio_needed;
    PetscLogDouble factor_mallocs;
} MatInfo;
```

Verifying Predictions (2/2)


```
[...]

MatInfo info;
Mat A;
double numMal, nz_a, nz_u;

[...]

MatGetInfo(A, MAT_LOCAL, &info);

numMal = info.mallocs;
nz_a   = info.nz_allocated;
nz_u   = info.nz_used;

[...]
```

3_petsc_mat.c


```
[…]
PetscViewer viewr_fd;
Mat mC;
[…]
ierr = PetscViewerBinaryOpen(PETSC_COMM_WORLD, "data/mC.bin",
                         FILE_MODE_READ, &viewr_fd ); CHKERRQ(ierr);
ierr = MatCreate(PETSC_COMM_WORLD, &mC);               CHKERRQ(ierr);
ierr = MatSetType(mC, MATAIJ);                         CHKERRQ(ierr);
ierr = MatLoad(mC, viewr_fd);                          CHKERRQ(ierr);
ierr = PetscViewerDestroy(&viewr_fd);                  CHKERRQ(ierr);
CHKMEMQ;

MatGetSize(mC,&row_global,&col_global);                CHKERRQ(ierr);
MatGetOwnershipRange(mC,&row_local_min,&row_local_max);
[…]
MatDestroy(&mC);
[…]
```

KSP and SNES

KSP: linear equations solvers

The **object KSP** provides uniform and efficient access to all of the package's **linear system solvers**

KSP is intended for solving nonsingular systems of the form

$$Ax = b.$$

```
KSPCreate(MPI_Comm comm, KSP *ksp)
KSPSetOperators(KSP ksp, Mat Amat, Mat Pmat,
                MatStructure flag)
KSPSolve(KSP ksp, Vec b, Vec x)
KSPGetIterationNumber(KSP ksp, int *its)
KSPDestroy(KSP ksp)
```

PETSc KSP methods

Method	KSPType	Options Database Name	Default Convergence Monitor†
Richardson	KSPRICHARDSON	richardson	true
Chebychev	KSPCHEBYCHEV	chebychev	true
Conjugate Gradient [11]	KSPCG	cg	true
BiConjugate Gradient	KSPBICG	bicg	true
Generalized Minimal Residual [15]	KSPGMRES	gmres	precond
BiCGSTAB [18]	KSPBCGS	bcgs	precond
Conjugate Gradient Squared [17]	KSPCGS	cgs	precond
Transpose-Free Quasi-Minimal Residual (1) [7]	KSPTFQMR	tfqmr	precond
Transpose-Free Quasi-Minimal Residual (2)	KSPTCQMR	tcqmr	precond
Conjugate Residual	KSPCR	cr	precond
Least Squares Method	KSPLSQR	lsqr	precond
Shell for no KSP method	KSPPREONLY	preonly	precond

†true - denotes true residual norm, precond - denotes preconditioned residual norm