Cineca TRAINING High Performance Computing 2017

Introduction to Scientific Libraries

Piero Lanucara and Massimiliano Guarrasi, Nicola Spallanzani, Simone Bnà

p.lanucara@cineca.it

CINECA

…..or….

- ``One thing to keep in mind when working at the algorithm level is that ***you do not need to reinvent the wheel***. If there is a library available that supports the features of the system under consideration, you should use it. A good example is the standard linear algebra operations. Nobody should program a matrix-matrix multiplication or an eigenvalue solver if it is not absolutely necessary and known to deliver a great benefit. The vector-vector, matrix-vector and matrix-matrix operations are standardized in the so-called Basic Linear Algebra System (BLAS), while the solvers can be addressed via the Linear Algebra Package (LAPACK) interfaces, for which many implementations are available. One of them is the Intel Math Kernel library (Intel MKL), which is, of course, fully vectorized for all available Intel architectures and additionally offers shared memory parallelization."

Scientific libraries at a glance

- A (complete?) set of function implementing different numerical algorithms
- A set of basic function (e.g. Fast Fourier Transform, scalar product, …)
- A set of low level function (e.g. scalar products or random number generator), or more complex algorithms (Fourier Transform or Matrix diagonalization)
- (Usually) Faster than hand made code (i.e. sometimes it is written in assembler)
- Proprietary or Open Source
- Sometimes developed for a particular compiler/architecture

Pros:

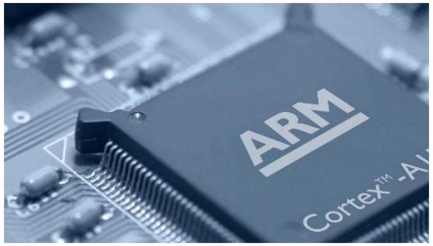
- Helps to modularize the code
- Portability
- Efficiency
- Ready to use

Cons:

- Some details are hidden (e.g. Memory requirements)
- You don't have the complete control
- You have to read carefully the documentation (complex syntax, error prone....)

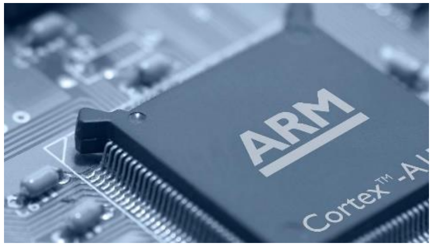
Which library?

- It is hard to have a complete overview of Scientific libraries
- many different libraries
- still evolving…
- …expecially for ``new architectures'' (e.g GPU, Intel Xeon PHI...)



Which PARALLEL library?

- Complex Grand Challenge projects (or even less…) runs on Top500 machines (or even less…)
- High number of nodes, multicore (or manycores) sockets within node
- … use PARALLEL libraries built on top of MPI, OpenMP, CUDA, OpenCL (a mixed combination of…)

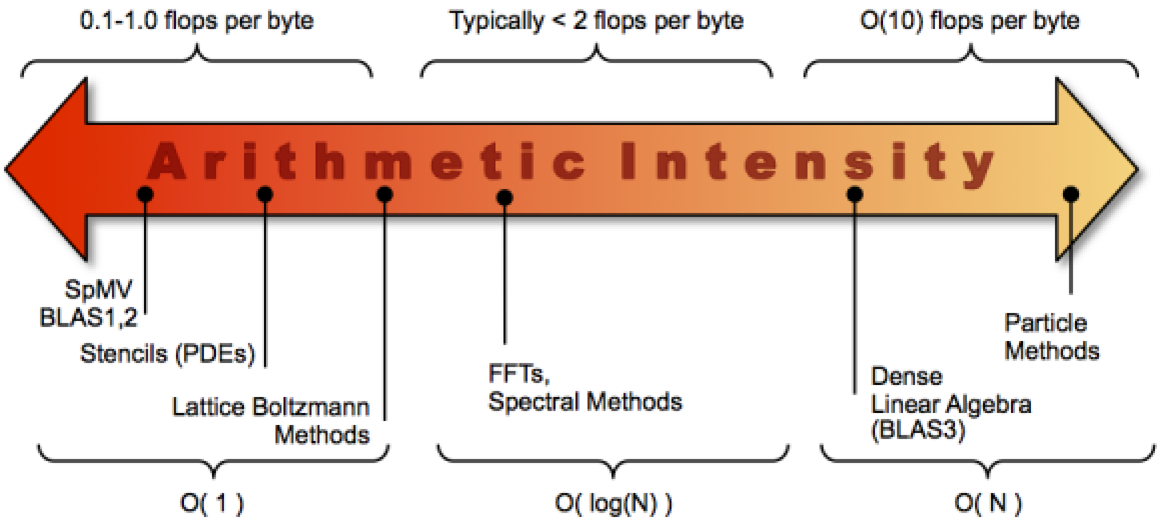


Some Example

1. Parallel FFT libraries:
   - FFTW
   - Other Libs:
     - 2Decomp&FFT
     - P3DFFT
   - Examples
2. Parallel Dense Linear Algebra libraries:
   - BLACS
   - ScaLAPACK
3. Parallel Sparse Linear Algebra libraries:
   - PETSc

Some Example

1. Parallel FFT libraries:
   - FFTW
   - O
     -
     -
   - E
2. Para
   - Bl
   - ScaLAPACK
3. Parallel Sparse Linear Algebra libraries:
   - PETSc



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)

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

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)

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

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 fn

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

Scale like N*logN (binary tree)

Parallel Domain Decomposition

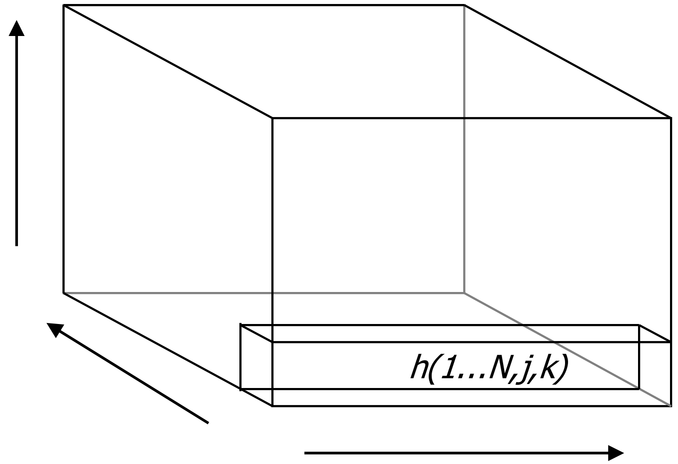
How to compute a FFT on a distributed memory system

- 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***

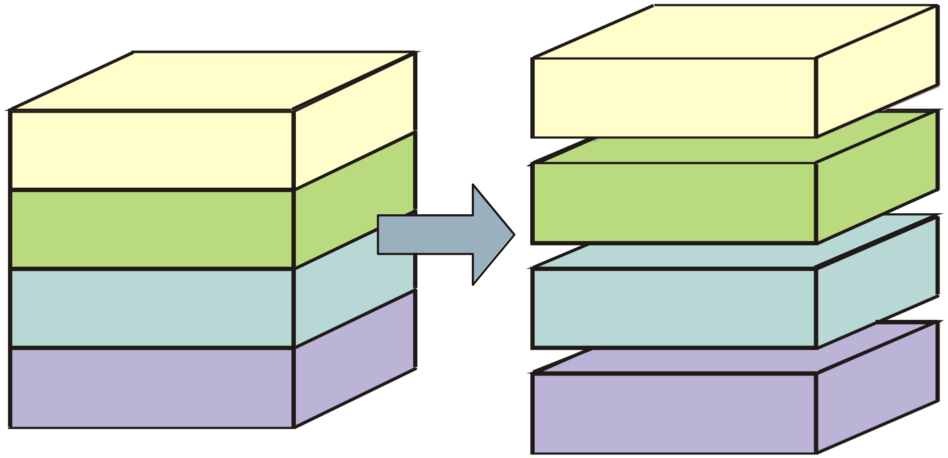
Apply FFT to h(1...N,j,k)

*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

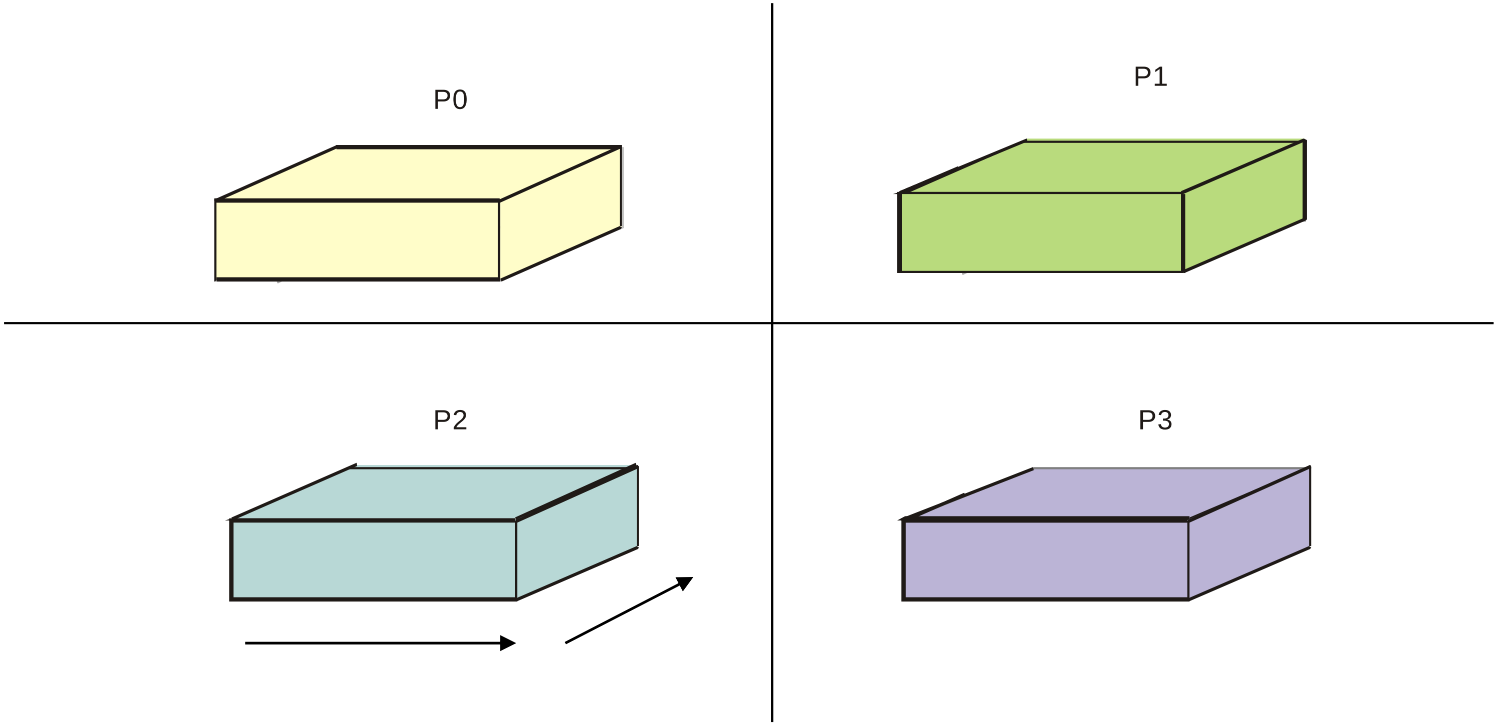


3D Grid

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

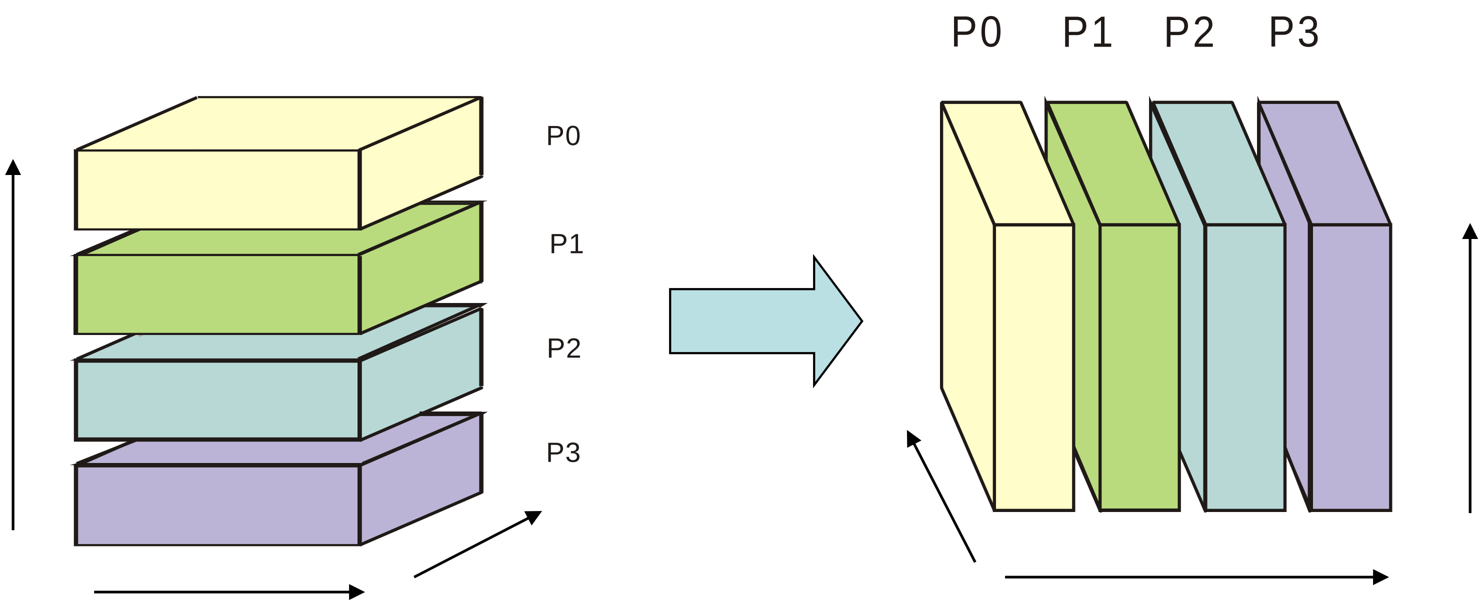
This is known as "Slab Decomposition" or 1D Decomposition

Transform along x and y



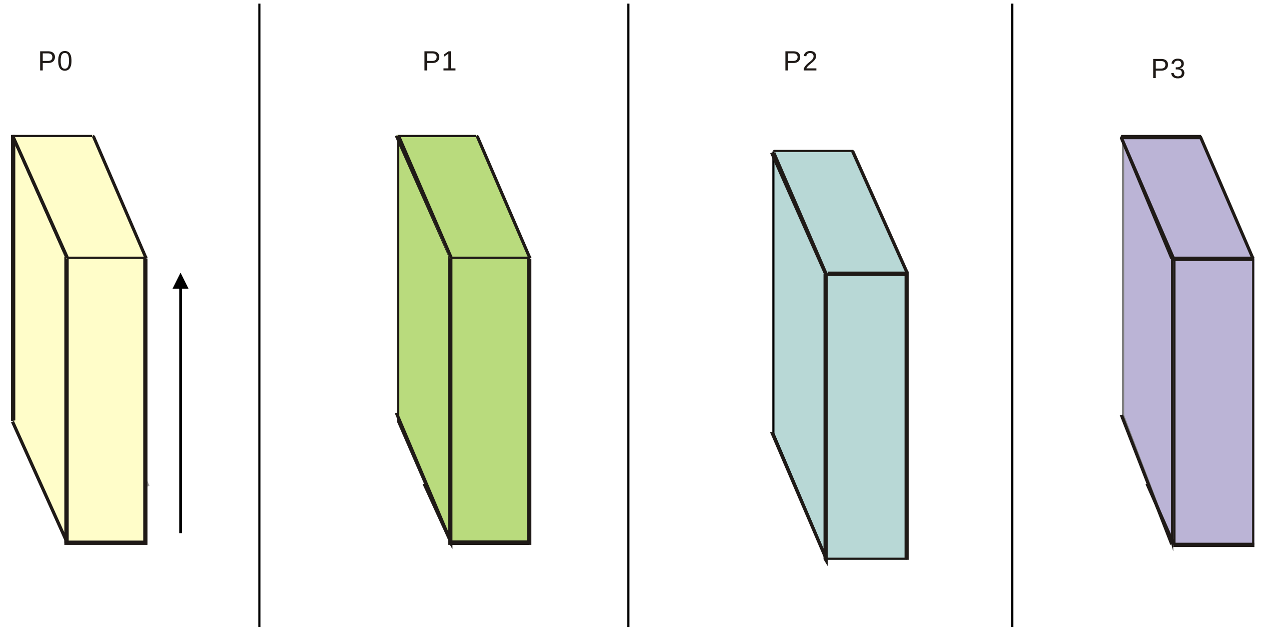
each processor trasform its own sub-grid along the x and y independently of the other

Data redistribution involving x and z



The data are now distributed along x

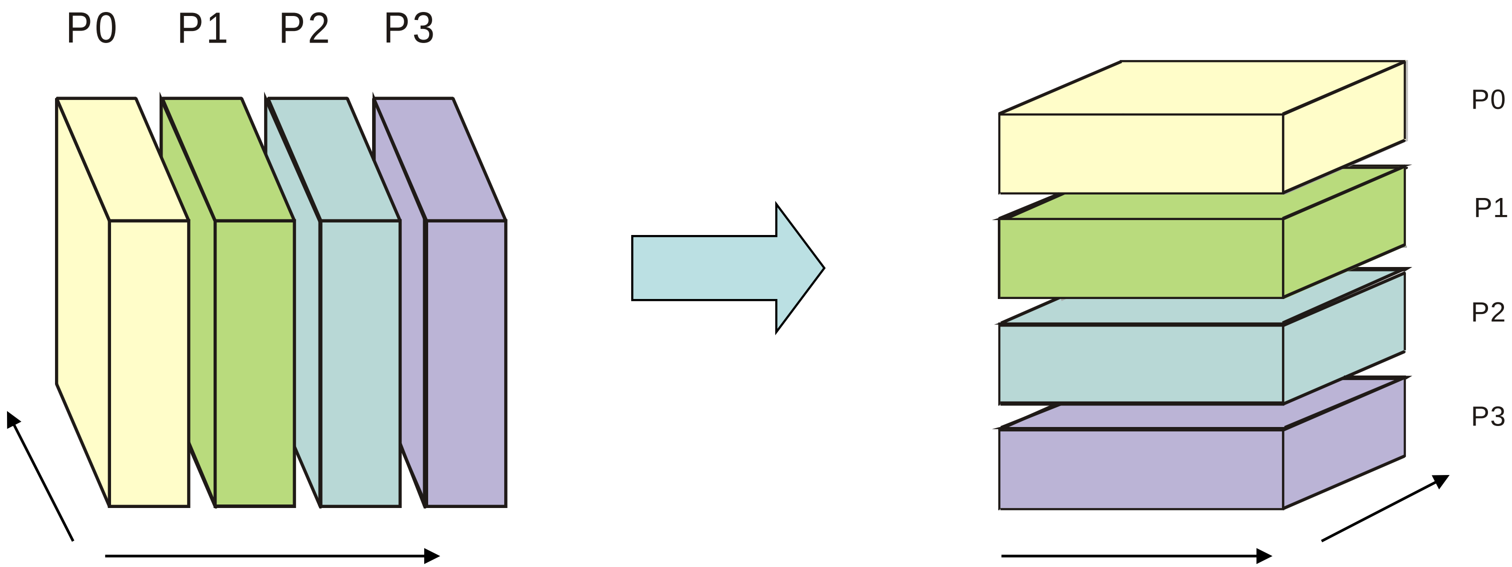
FFT along z



each processor transform its own sub-grid

along the z dimension independently of the other

Data are re-distributed, back from x to z



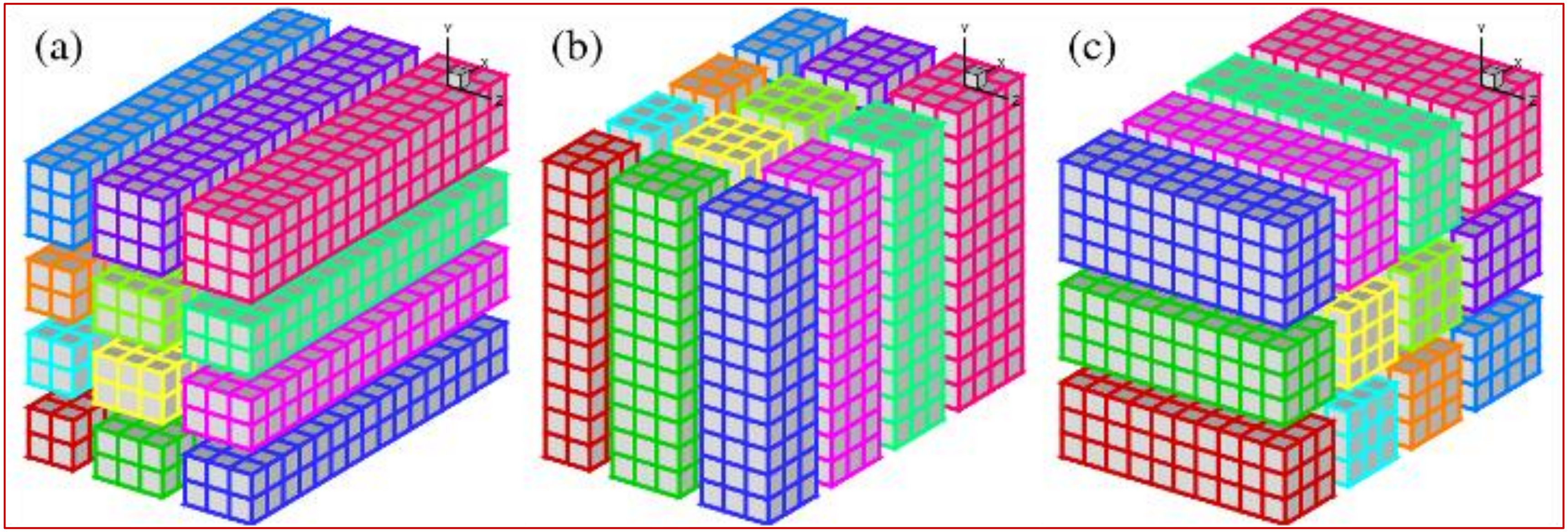
The 3D array now has the original layout, but each element

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)

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

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

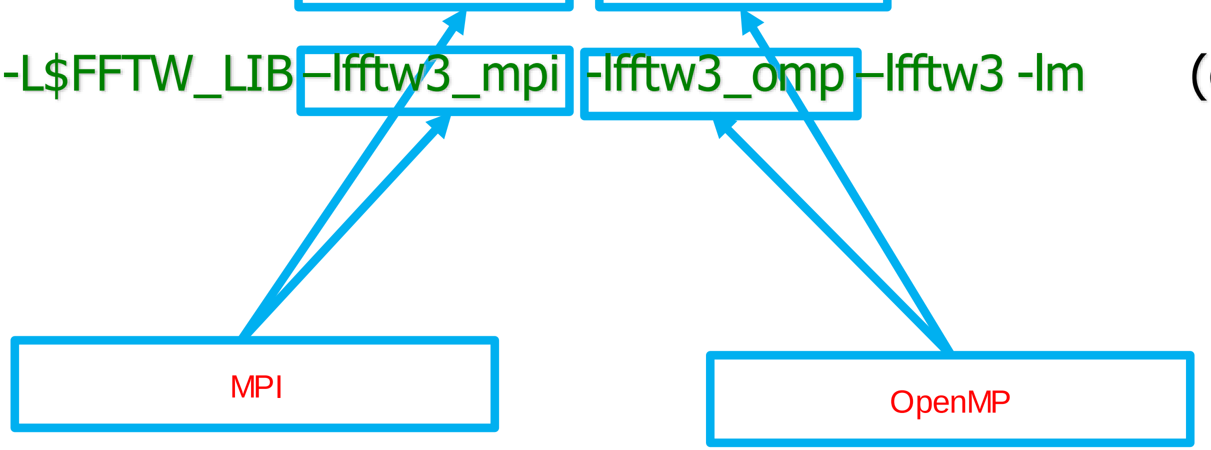
Including header:

•-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_binding 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 FFTW - Fortran

Cineca TRAINING High Performance Computing 2017

```
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
```

CINECA

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)


```fortran
program FFT_MPI_2D
      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

Parallel Three-Dimensional Fast Fourier Transforms (P3DFFT)

- General-purpose 3D 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

Linear algebra constitutes the core of most technical-scientific applications

Scalar products $$s = \sum_i a_i \cdot b_i$$

Linear Systems $$A_{ij} x_j = b_i$$

Eigenvalue Equations $$A_{ij} x_j = \alpha x_i$$

Linear Algebra is Hierarchical

Linear systems, Eigenvalue equations

3 M x M products

2 M x V products

1 V x V products

Standard Linear Algebra Libraries

- BLAS
- LAPACK
- ARPACK
- ScaLAPACK
- P_ARPACK
- PETSc

BLAS, LAPACK, ARPACK → Serial Linear Algebra Packages

essl (IBM AIX)
mkl (Intel)
acml (AMD)
plasma (ICL – Univ. Tennessee)
magma (ICL – Univ. Tennessee)

→ threaded

→ hybrid

ScaLAPACK, P_ARPACK → Parallel (distributed) Linear Algebra Packages (for dense matrices)

dplasma (ICL – Univ. Tennessee)

PETSc → Sparse (non-)Linear Systems

- **Level 1 : Vector - Vector operations**
- **Level 2 : Vector - Matrix operations**
- **Level 3 : Matrix - Matrix operations**

(Scalable) Linear Algebra PACKage (LAPACK and ScaLAPACK)

- **Matrix Decomposition**
- **Linear Equation Systems**
- **Eigenvalue Equations**
- **Linear Least Square Equations**
- **for dense, banded, triangular, real and complex matrices**

BLAS/LAPACK subroutines

Routines name scheme: XYYZZZ

X data type → S = REAL
D = DOUBLE PRECISION
C = COMPLEX
Z = DOUBLE COMPLEX

YY matrix type (GE = general, SY = symmetric, HE = hermitian)

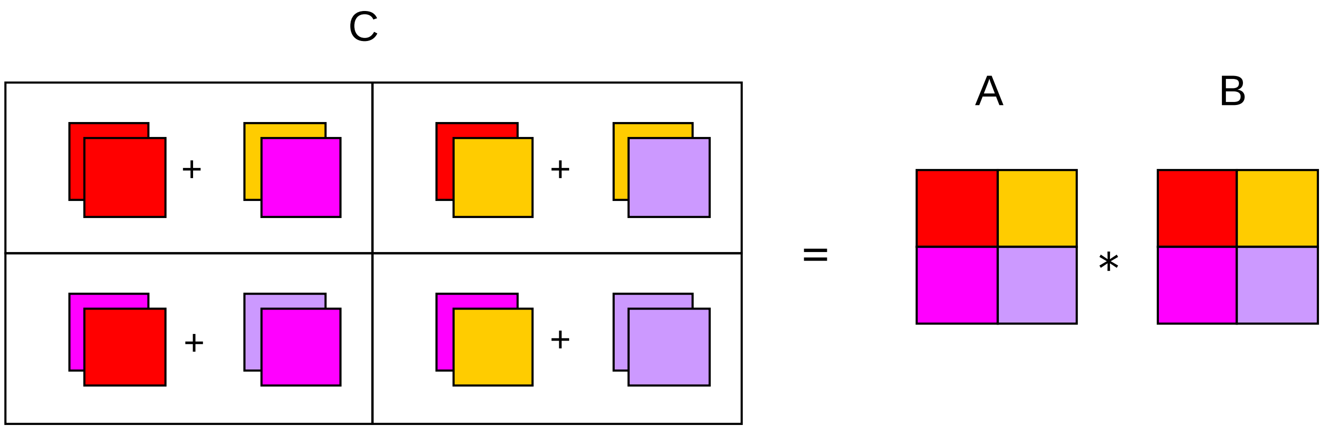
ZZZ algorithm used to perform computation

Some auxiliary functions don't make use of this naming scheme!

Block Operations

A block representation of a matrix operation constitutes the basic parallelization strategy for dense matrices.

For instance, a matrix-matrix product can be split in a sequence of smaller operations of the same type acting on subblocks of the original matrix



$$c_{ij} = \sum_{k=1}^{N} a_{ik} \cdot b_{kj}$$

Example: Partitioning into 2x2 Blocks

a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	a_{16}	a_{17}	a_{18}	a_{19}
a_{21}	a_{22}	a_{23}	a_{24}	a_{25}	a_{26}	a_{27}	a_{28}	a_{29}
a_{31}	a_{32}	a_{33}	a_{34}	a_{35}	a_{36}	a_{37}	a_{38}	a_{39}
a_{41}	a_{42}	a_{43}	a_{44}	a_{45}	a_{46}	a_{47}	a_{48}	a_{49}
a_{51}	a_{52}	a_{53}	a_{54}	a_{55}	a_{56}	a_{57}	a_{58}	a_{59}
a_{61}	a_{62}	a_{63}	a_{64}	a_{65}	a_{66}	a_{67}	a_{68}	a_{69}
a_{71}	a_{72}	a_{73}	a_{74}	a_{75}	a_{76}	a_{77}	a_{78}	a_{79}
a_{81}	a_{82}	a_{83}	a_{84}	a_{85}	a_{86}	a_{87}	a_{88}	a_{89}
a_{91}	a_{92}	a_{93}	a_{94}	a_{95}	a_{96}	a_{97}	a_{98}	a_{99}

B_{11}	B_{12}	B_{13}	B_{14}	B_{15}
B_{21}	B_{22}	B_{23}	B_{24}	B_{25}
B_{31}	B_{32}	B_{33}	B_{34}	B_{35}
B_{41}	B_{42}	B_{43}	B_{44}	B_{45}
B_{51}	B_{52}	B_{53}	B_{54}	B_{55}

Block Representation

Next Step: distribute blocks among processors

Process Grid

N processes are organized into a logical 2D mesh with p rows and q columns, such that p x q = N

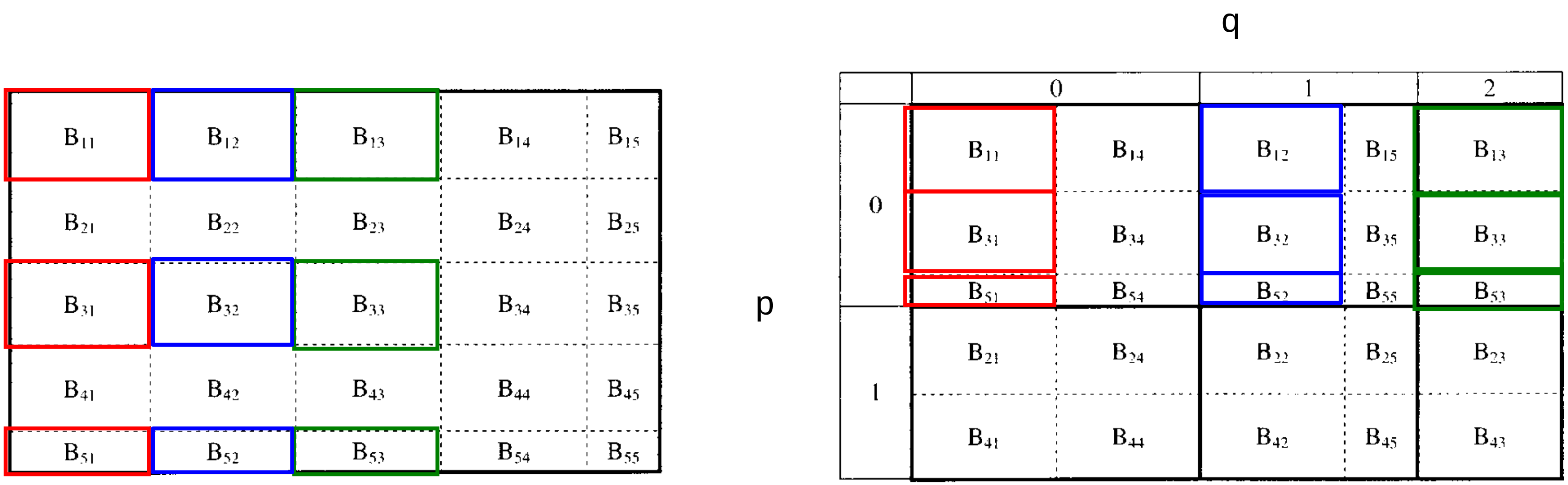
p

q

	0	1	2
0	rank = 0	rank = 1	rank = 2
1	rank = 3	rank = 4	rank = 5

A process is referenced by its coordinates within the grid rather than a single number

Cyclic Distribution of Blocks



Blocks are distributed on processors in a cyclic manner
on each index

Some routine can help us to make the best distribution

Distribution of matrix elements

	0		1		2
0	B_{11}	B_{14}	B_{12}	B_{15}	B_{13}
	B_{31}	B_{34}	B_{32}	B_{35}	B_{33}
	B_{51}	B_{54}	B_{52}	B_{55}	B_{53}
1	B_{21}	B_{24}	B_{22}	B_{25}	B_{23}
	B_{41}	B_{44}	B_{42}	B_{45}	B_{43}

The indexes of a single element can be traced back to the processor

	0				1			2	
0	a_{11}	a_{12}	a_{17}	a_{18}	a_{13}	a_{14}	a_{19}	a_{15}	a_{16}
	a_{21}	a_{22}	a_{27}	a_{28}	a_{23}	a_{24}	a_{29}	a_{25}	a_{26}
	a_{51}	a_{52}	a_{57}	a_{58}	a_{53}	a_{54}	a_{59}	a_{55}	a_{56}
	a_{61}	a_{62}	a_{67}	a_{68}	a_{63}	a_{64}	a_{69}	a_{65}	a_{66}
	a_{91}	a_{92}	a_{97}	a_{98}	a_{93}	a_{94}	a_{99}	a_{95}	a_{96}
1	a_{31}	a_{32}	a_{37}	a_{38}	a_{33}	a_{34}	a_{39}	a_{35}	a_{36}
	a_{41}	a_{42}	a_{47}	a_{48}	a_{43}	a_{44}	a_{49}	a_{45}	a_{46}
	a_{71}	a_{72}	a_{77}	a_{78}	a_{73}	a_{74}	a_{79}	a_{75}	a_{76}
	a_{81}	a_{82}	a_{87}	a_{88}	a_{83}	a_{84}	a_{89}	a_{85}	a_{86}

myid=0	myid=1	myid=2	myid=3	myid=4	myid=5
p=0 q=0	p=0 q=1	p=0 q=2	p=1 q=0	p=1 q=1	p=1 q=2

Distribution of matrix elements

a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	a_{16}	a_{17}	a_{18}	a_{19}
a_{21}	a_{22}	a_{23}	a_{24}	a_{25}	a_{26}	a_{27}	a_{28}	a_{29}
a_{31}	a_{32}	a_{33}	a_{34}	a_{35}	a_{36}	a_{37}	a_{38}	a_{39}
a_{41}	a_{42}	a_{43}	a_{44}	a_{45}	a_{46}	a_{47}	a_{48}	a_{49}
a_{51}	a_{52}	a_{53}	a_{54}	a_{55}	a_{56}	a_{57}	a_{58}	a_{59}
a_{61}	a_{62}	a_{63}	a_{64}	a_{65}	a_{66}	a_{67}	a_{68}	a_{69}
a_{71}	a_{72}	a_{73}	a_{74}	a_{75}	a_{76}	a_{77}	a_{78}	a_{79}
a_{81}	a_{82}	a_{83}	a_{84}	a_{85}	a_{86}	a_{87}	a_{88}	a_{89}
a_{91}	a_{92}	a_{93}	a_{94}	a_{95}	a_{96}	a_{97}	a_{98}	a_{99}

Logical View (Matrix)

a_{11}	a_{12}	a_{17}	a_{18}	a_{13}	a_{14}	a_{19}	a_{15}	a_{16}
a_{21}	a_{22}	a_{27}	a_{28}	a_{23}	a_{24}	a_{29}	a_{25}	a_{26}
a_{51}	a_{52}	a_{57}	a_{58}	a_{53}	a_{54}	a_{59}	a_{55}	a_{56}
a_{61}	a_{62}	a_{67}	a_{68}	a_{63}	a_{64}	a_{69}	a_{65}	a_{66}
a_{91}	a_{92}	a_{97}	a_{98}	a_{93}	a_{94}	a_{99}	a_{95}	a_{96}
a_{31}	a_{32}	a_{37}	a_{38}	a_{33}	a_{34}	a_{39}	a_{35}	a_{36}
a_{41}	a_{42}	a_{47}	a_{48}	a_{43}	a_{44}	a_{49}	a_{45}	a_{46}
a_{71}	a_{72}	a_{77}	a_{78}	a_{73}	a_{74}	a_{79}	a_{75}	a_{76}
a_{81}	a_{82}	a_{87}	a_{88}	a_{83}	a_{84}	a_{89}	a_{85}	a_{86}

Local View (CPUs)

http://acts.nersc.gov/scalapack/hands-on/datadist.html
http://acts.nersc.gov/scalapack/hands-on/addendum.html

BLACS

(**B**asic **L**inear **A**lgebra **C**ommunication **S**ubprograms)

The BLACS project is an ongoing investigation whose purpose is to create a linear algebra oriented message passing interface that may be implemented efficiently and uniformly across a large range of distributed memory platforms

ScaLAPACK
BLACS
Communication Library (MPI)

BLACS Process Grid

Processes are distributed on a 2D mesh using row-order or column-order (ORDER='R' or 'C'). Each process is assigned a row/column ID as well as a scalar ID

column ID

row ID

	0	*1*	*2*	*3*
0	**0**	**1**	**2**	**3**
1	**4**	**5**	**6**	**7**

COMM_ROW
Communications along rows

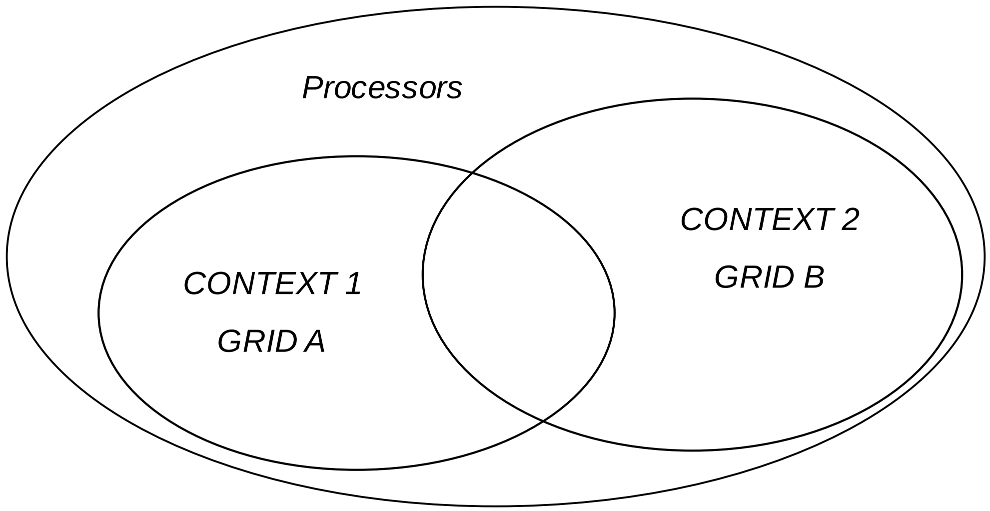
COMM_COL
Communications along columns

MPI task ID (rank)

```
BLACS_GRIDINIT( CONTEXT, ORDER, NPROW, NPCOL )
```

Initialize a 2D grid of NPROW x NPCOL processes with an order specified by ORDER in a given CONTEXT

CONTEXT



Context ⟷ *MPI Communicators*

BLACS_PINFO(MYPNUM, NPROCS)

Query the system for process ID MYPNUM (output) and number of processes NPROCS (output).

BLACS_GET(ICONTEXT, WHAT, VAL)

Query to BLACS environment based on WHAT (input) and ICONTEXT (input)
If WHAT=0, ICONTEX is ignored and the routine returns in VAL (output) a value indicating the default system context

BLACS_GRIDINIT(CONTEXT, ORDER, NPROW, NPCOL)

Initialize a 2D mesh of processes

BLACS_GRIDINFO(CONTEXT, NPROW, NPCOL, MYROW, MYCOL)

Query CONTEXT for the dimension of the grid of processes (NPROW, NPCOL) and for row-ID and col-ID (MYROW, MYCOL)

BLACS_GRIDEXIT(CONTEXT)

Release the 2D mesh associated with CONTEXT

BLACS_EXIT(CONTINUE) Exit from BLACS environment

BLACS: Subroutines

Point to Point Communication

DGESD2D(ICONTEX,M,N,A,LDA,RDEST,CDEST)

Send matrix A(M,N) to process (RDEST,CDEST)

DGERV2D(ICONTEX,M,N,A,LDA,RSOUR,CSOUR)

Receive matrix A(M,N) from process (RSOUR,CSOUR)

Broadcast

DGEBS2D(ICONTEX,SCOPE,TOP,M,N,A,LDA)

Execute a Broadcast of matrix A(M,N)

DGEBR2D(ICONTEX,SCOPE,TOP,M,N,A,LDA,RSRC,CSRC)

Receive matrix A(M,N) sent from process (RSRC,CSRC) with a broadcast operation

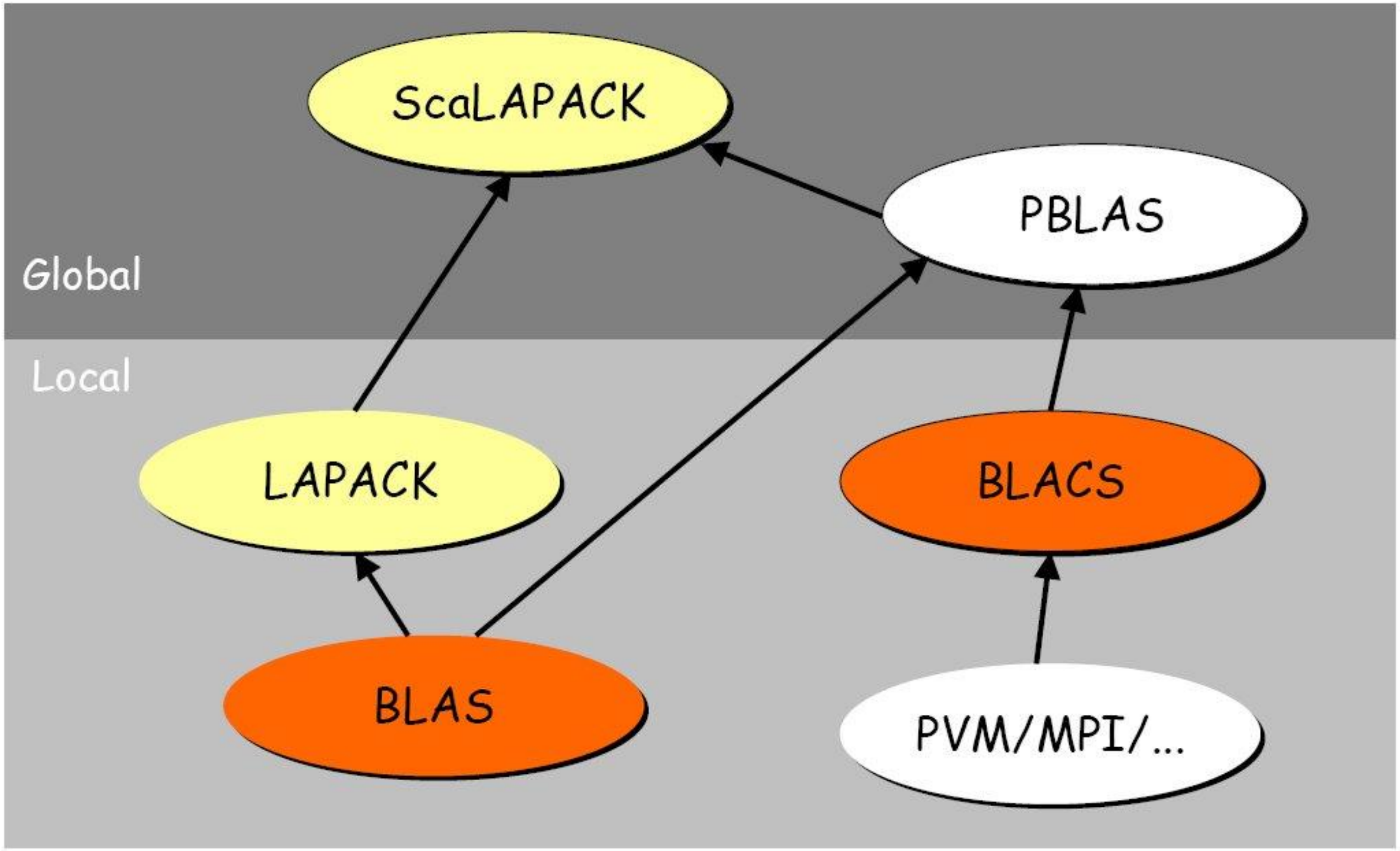
Global reduction

DGSUM2D(ICONTXT,SCOPE,TOP,M,N,A,LDA,RDST,CDST)

Execute a parallel element-wise sum of matrix A(M,N) and store the result in process (RDST,CDST) buffer

http://www.netlib.org/blacs/BLACS/QRef.html

Dependencies



ScaLAPACK and PBLAS: template

1. *Initialize BLACS*
2. *Initialize BLACS grids*
3. *Distribute matrix among grid processes (cyclic block distribution)*
4. *Calls to ScaLAPACK/PBLAS routines*
5. *Harvest results*
6. *Release BLACS grids*
7. *Close BLACS environment*

Example:


```
       !      Initialize the BLACS

       CALL BLACS_PINFO( IAM, NPROCS )

       !      Set the dimension of the 2D processors grid

       CALL GRIDSETUP( NPROCS, NPROW, NPCOL ) ! User defined

       write (*,100) IAM, NPROCS, NPROW, NPCOL
100    format(' MYPE ',I3,', NPE ',I3,', NPE ROW ',I3,', NPE COL
',I3)

       !      Initialize a single BLACS context

       CALL BLACS_GET( -1, 0, CONTEXT )
       CALL BLACS_GRIDINIT( CONTEXT, 'R', NPROW, NPCOL )
       CALL BLACS_GRIDINFO( CONTEXT, NPROW, NPCOL, MYROW, MYCOL )
       …………
       …………
   CALL BLACS_GRIDEXIT( CONTEXT )
   CALL BLACS_EXIT( 0 )
```

The Descriptor is an integer array that stores the information required to establish the mapping between each global array entry and its corresponding process and memory location.

Each matrix MUST be associated with a Descriptor. Anyhow it's responsibility of the programmer to distribute the matrix coherently with the Descriptor.

```
DESCA( 1 ) = 1        DESCA( 2 ) = ICTXT
DESCA( 3 ) = M        DESCA( 4 ) = N
DESCA( 5 ) = MB              DESCA( 6 ) = NB
DESCA( 7 ) = RSRC            DESCA( 8 ) = CSRC
DESCA( 9 ) = LDA
```

Descriptor Initialization

DESCINIT(DESCA, M, N, MB, NB, RSRC, CSRC, ICTXT, LDA, INFO)

DESCA(9) (global output) matrix A ScaLAPACK Descriptor

M, N (global input) global dimensions of matrix A

MB, NB (global input) blocking factors used to distribute matrix A

RSRC, CSRC (global input) process coordinates over which the first element of A is distributed

ICTXT (global input) BLACS context handle, indicating the global context of the operation on matrix

LDA (local input) leading dimension of the local array (depends on process!)

ScaLAPACK tools

http://www.netlib.org/scalapack/tools

Computation of the local matrix size for a M x N matrix distributed over processes in blocks of dimension MB x NB

```
Mloc = NUMROC( M, MB, ROWID, 0, NPROW )
Nloc = NUMROC( N, NB, COLID, 0, NPCOL )
allocate( Aloc( Mloc, Nloc ) )
```

Computation of local and global indexes

```
iloc = INDXG2L( i, MB, ROWID, 0, NPROW )
jloc = INDXG2L( j, NB, COLID, 0, NPCOL )

i = INDXL2G( iloc, MB, ROWID, 0, NPROW )
j = INDXL2G( jloc, NB, COLID, 0, NPCOL )
```

ScaLAPACK tools

Compute the process to which a certain global element **(i,j)** belongs

```
iprow = INDXG2P( i, MB, ROWID, 0, NPROW )
jpcol = INDXG2P( j, NB, COLID, 0, NPCOL )
```

Define/read a local element, knowing global indexes

```
CALL PDELSET( A, i, j, DESCA, aval )
```

local array

input value

```
CALL PDELGET( SCOPE, TOP, aval, A, i, j, DESCA )
```

output value

character*1 topology of the broadcast 'D' or 'I'

character*1 scope broadcast 'R', 'C' or 'A'

PBLAS/ScaLAPACK subroutines

Routines name scheme: PXYYZZZ

Parallel (P)

X data type → S = REAL
D = DOUBLE PRECISION
C = COMPLEX
Z = DOUBLE COMPLEX

YY matrix type (GE = general, SY = symmetric, HE = hermitian)

ZZZ algorithm used to perform computation

Some auxiliary functions don't make use of this naming scheme!

Calls to ScaLAPACK routines

- It's responsibility of the programmer to correctly distribute a global matrix before calling ScaLAPACK routines
- ScaLAPACK routines are written using a message passing paradigm, therefore each subroutine access directly ONLY local data
- Each process of a given CONTEXT must call the same ScaLAPACK routine...
- … providing in input its local portion of the global matrix
- Operations on matrices distributed on processes belonging to different contexts are not allowed

PBLAS subroutines

matrix multiplication: C = A * B (level 3)

```
PDGEMM('N', 'N', M, N, L, 1.0d0, A, 1, 1, DESCA, B, 1, 1, DESCB, 0.0d0, C, 1,
1, DESCC)
```

matrix transposition: C = A' (level 3)

```
PDTRAN( M, N, 1.0d0, A, 1, 1, DESCA, 0.0d0, C, 1, 1, DESCC )
```

matrix times vector: Y = A * X (level 2)

```
PDGEMV('N', M, N, 1.0d0, A, 1, 1, DESCA, X, 1, JX, DESCX, 1, 0.0d0, Y, 1, JY,
DESCY, 1)
```

X(1:N,JX:JX)   Y(1:M,JY:JY)

row / column swap: X ⇔ Y (level 1)

```
PDSWAP( N, X, IX, JX, DESCX, INCX, Y, IY, JY, DESCY, INCY )
        X(IX,JX:JX+N-1) if INCX = M_X,  X(IX:IX+N-1,JX) if INCX = 1 and INCX <> M_X,
        Y(IY,JY:JY+N-1) if INCY = M_Y,  Y(IY:IY+N-1,JY) if INCY = 1 and INCY <> M_Y.
```

scalar product: p = X' ·Y (level 1)

```
PDDOT( N, p, X, IX, JX, DESCX, INCX, Y, IY, JY, DESCY, INCY )
        X(IX,JX:JX+N-1) if INCX = M_X, X(IX:IX+N-1,JX) if INCX = 1 and INCX <> M_X,
        Y(IY,JY:JY+N-1) if INCY = M_Y, Y(IY:IY+N-1,JY) if INCY = 1 and INCY <> M_Y.
```

Eigenvalues and, optionally, eigenvectors: A Z = w Z

```
PDSYEV( 'V', 'U', N, A, 1, 1, DESCA, W, Z, 1, 1, DESCZ, WORK, LWORK, INFO )
```

'U' use upper triangular part of A
'L' use lower triangular part of A

if **lwork = -1**, compute workspace dimension.
Return it in **work(1)**

'V' compute eigenvalues and eigenvectors
'N' compute eigenvalues only

Print matrix

```
PDLAPRNT( M, N, A, 1, 1, DESCA, IR, IC, CMATNM, NOUT, WORK)
```

M global first dimension of A

IR, IC coordinates of the printing process

N global second dimension of A

CMATNM character*(*) title of the matrix

A local part of matrix A

NOUT output fortran units (0 stderr, 6 stdout)

DESCA descriptor of A

WORK workspace

BLACS/ScaLAPACK + MPI

It is quite tricky to write a program using BLACS as a communication library, therefore:

MPI and BLACS must be used consistently!

Initialize MPI + BLACS


```
CALL MPI_INIT(IERR)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NPROC,IERR)
CALL MPI_COMM_RANK(MPI_COMM_WORLD,MPIME,IERR)
!
comm_world = MPI_COMM_WORLD
!
ndims = 2
dims  = 0
CALL MPI_DIMS_CREATE( NPROC, ndims, dims, IERR)

NPROW = dims(1)   !  cartesian direction 0
NPCOL = dims(2)   !  cartesian direction 1

!     Get a default BLACS context
!
CALL BLACS_GET( -1, 0, ICONTEXT )

!     Initialize a default BLACS context
CALL BLACS_GRIDINIT(ICONTEXT, 'R', NPROW, NPCOL)
CALL BLACS_GRIDINFO(ICONTEXT, NPROW, NPCOL, ROWID, COLID)

CALL MPI_COMM_SPLIT(comm_world, COLID, ROWID, COMM_COL, IERR)
CALL MPI_COMM_RANK(COMM_COL, coor(1), IERR)
!
CALL MPI_COMM_SPLIT(comm_world, ROWID, COLID, COMM_ROW, IERR)
CALL MPI_COMM_RANK(COMM_ROW, coor(2), IERR)
```

Initialize MPI environment

Compute the dimensions of a 2D mesh compatible with NPROCS processes

Initialize BLACS process grid of size nprow x npcol

Create a row and a column communicator using BLACS indexes rowid and colid

Matrix redistribution


```
! Distribute matrix A0 (M x N) from root node to all processes in context ictxt.
!
call SL_INIT(ICTXT, NPROW, NPCOL)
call SL_INIT(rootNodeContext, 1, 1) ! create 1 node context
                                    ! for loading matrices
call BLACS_GRIDINFO( ICTXT, NPROW, NPCOL, MYROW, MYCOL)
!
! LOAD MATRIX ON ROOT NODE AND CREATE DESC FOR IT
!
if (MYROW == 0 .and. MYCOL == 0) then
   NRU = NUMROC( M, M, MYROW, 0, NPROW )
   call DESCINIT( DESCA0, M, N, M, N, 0, 0, rootNodeContext, max(1, NRU), INFO )
else
   DESCA0(1:9) = 0
   DESCA0(2) = -1
end if
!
!  CREATE DESC FOR DISTRIBUTED MATRIX
!
NRU = NUMROC( M, MB, MYROW, 0, NPROW )
CALL DESCINIT( DESCA, M, N, MB, NB, 0, 0, ICTXT, max(1, NRU), INFO )
!
!  DISTRIBUTE DATA
!
if (debug) write(*,*) "node r=", MYROW, "c=", MYCOL, "M=", M, "N=", N
call PDGEMR2D( M, N, A0, 1, 1, DESCA0, A, 1, 1, DESCA, DESCA( 2 ) )
```

How To Compile (INTEL MKL)

- *# load these modules on MARCONI:*
- **`module load autoload profile/advanced`**
- **`module load scalapack/2.0.2--intelmpi--2017--binary`**
- **`MKL="-I${MKL_INC} -L${MKL_LIB} -lmkl_scalapack_lp64 \`**
- **`     -lmkl_intel_lp64 -lmkl_core -lmkl_sequential \`**
- **`     -lmkl_blacs_intelmpi_lp64"`**
- **`LALIB="-L${SCALAPACK_LIB} -lscalapack"`**

- *C:*
- *(remember to include mkl.h, mkl_scalapack.h, mkl_blacs.h)*
- **`mpicc -o program.x program.c ${MKL} ${LALIB}`**

- *FORTRAN:*
- **`mpif90 -o program.x program.f90 ${MKL} ${LALIB}`**

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, ...

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
  - Line Search
  - Trust Region
- Other

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);	$A = \mathrm{diag}(l) * A * \mathrm{diag}(r)$
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);	$x = \mathrm{diag}(A)$
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.

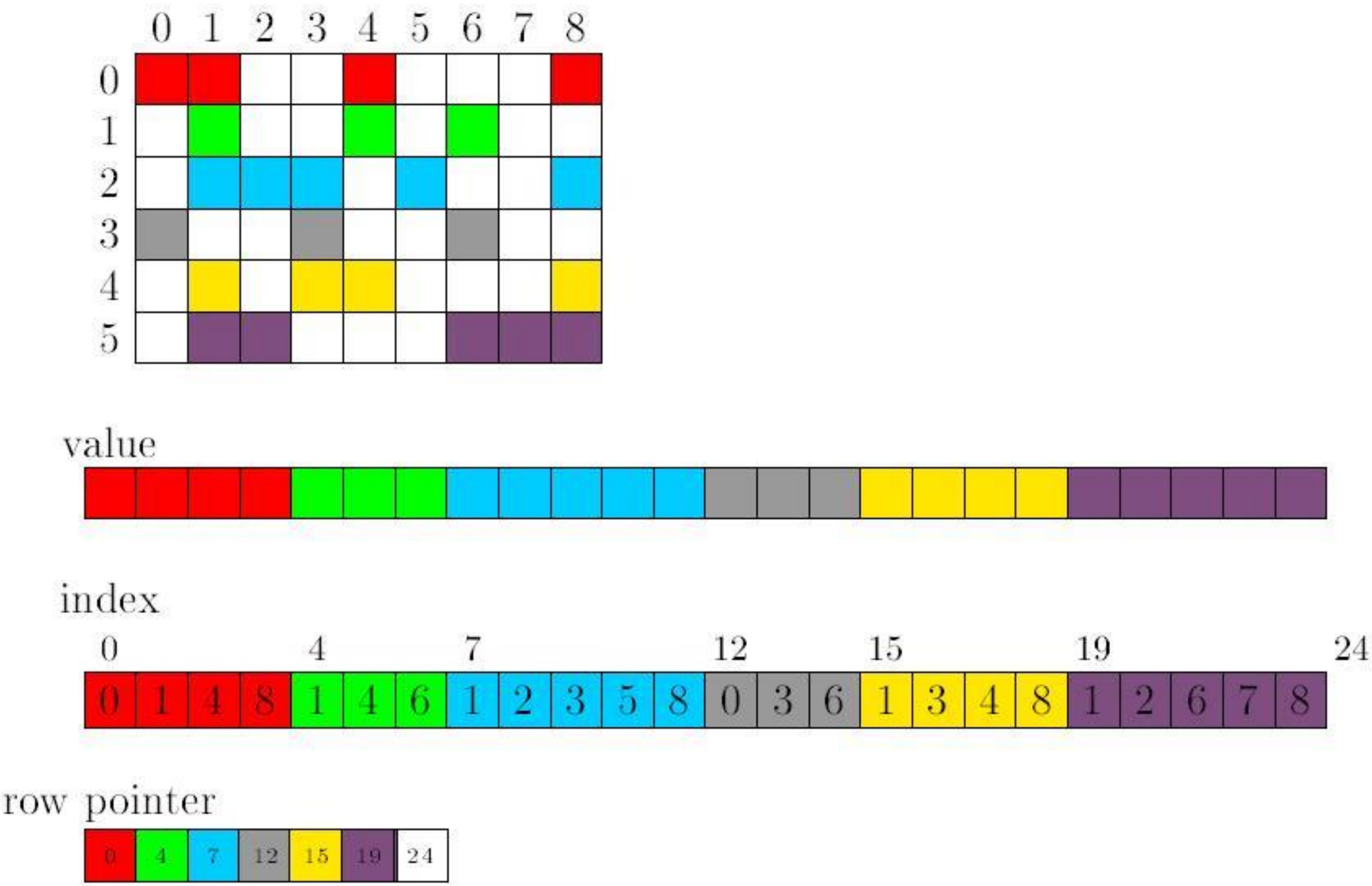
PETSc sparse matrices are dynamic data structures.
Can **add additional nonzeros freely**.

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