21st Summer School of PARALLEL COMPUTING

July 2 - 13, 2012 (Italian)
September 3 - 14, 2012 (English)

Introduction to PETSc

Portable, Extensible Toolkit for Scientific Computation

Ambra Giovannini – a.giovannini@cineca.it
SuperComputing Applications and Innovation Department

CINECA

PETSc main features

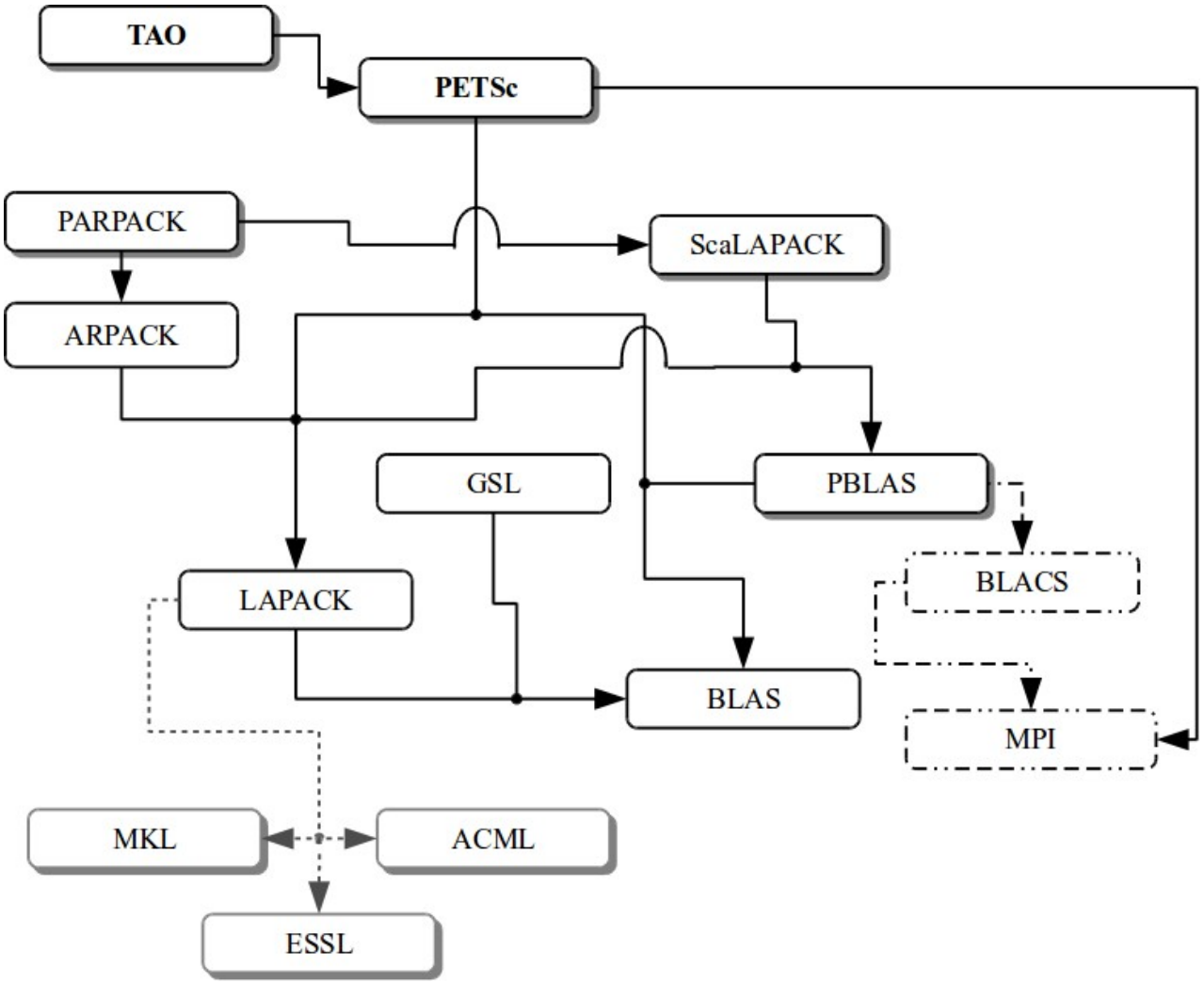
PETSc – Portable, Extensible Toolkit for Scientific Computation

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

1_petsc_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;
}
```

1_petsc_hello (Fortran)

```
program main

integer ierr, rank

#include "include/finclude/petsc.h“

call PetscInitialize( PETSC_NULL_CHARACTER,ierr )

call MPI_Comm_rank( PETSC_COMM_WORLD, rank, ierr )
if (rank .eq. 0) then
    print *, ‘Hello World’
endif

call PetscFinalize(ierr)

end
```

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**

1. Each process tells PETSc what values to set or add to a vector component. Once *all* values provided,
2. begin communication between processes to ensure that values end up where needed (allow other operations, such as some computation, to proceed).
3. 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

```
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

Name	Operation
Vec y,PetscScalar a,Vec x);	$y = y + a * x$
Vec y,PetscScalar a,Vec x);	$y = x + a * y$
(Vec w,PetscScalar a,Vec x,Vec y);	$w = a * x + y$
(Vec y,PetscScalar a,PetscScalar b,Vec x);	$y = a * x + b * y$
ec x, PetscScalar a);	$x = a * x$
x, Vec y, PetscScalar *r);	$r = \bar{x}' * y$
ec x, Vec y, PetscScalar *r);	$r = x' * y$
ec x,NormType type, PetscReal *r);	$r = \|\|x\|\|_{type}$
c x, PetscScalar *r);	$r = \sum x_i$
ec x, Vec y);	$y = x$
ec x, Vec y);	$y = x$ while $x = y$
seMult(Vec w,Vec x,Vec y);	$w_i = x_i * y_i$
seDivide(Vec w,Vec x,Vec y);	$w_i = x_i / y_i$
ec x,int n,Vec y[],PetscScalar *r);	$r[i] = \bar{x}' * y[i]$
(Vec x,int n,Vec y[],PetscScalar *r);	$r[i] = x' * y[i]$
(Vec y,int n, PetscScalar *a, Vec x[]);	$y = y + \sum_i a_i * x[i]$
c x, int *idx, PetscReal *r);	$r = \max x_i$
c x, int *idx, PetscReal *r);	$r = \min x_i$
x);	$x_i = \|x_i\|$
cal(Vec x);	$x_i = 1/x_i$
c x,PetscScalar s);	$x_i = s + x_i$
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

Name	Operation
Y(Mat Y, PetscScalar a,Mat X,MatStructure);	$Y = Y + a * X$
(Mat A,Vec x, Vec y);	$y = A * x$
Add(Mat A,Vec x, Vec y,Vec z);	$z = y + A * x$
Transpose(Mat A,Vec x, Vec y);	$y = A^T * x$
TransposeAdd(Mat A,Vec x, Vec y,Vec z);	$z = y + A^T * x$
(Mat A,NormType type, double *r);	$r = \|\|A\|\|_{type}$
onalScale(Mat A,Vec l,Vec r);	$A = \text{diag}(l) * A * \text{diag}(r)$
(Mat A,PetscScalar a);	$A = a * A$
ert(Mat A,MatType type,Mat *B);	$B = A$
(Mat A,Mat B,MatStructure);	$B = A$
iagonal(Mat A,Vec x);	$x = \text{diag}(A)$
pose(Mat A,MatReuse,Mat* B);	$B = A^T$
Entries(Mat A);	$A = 0$
(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.

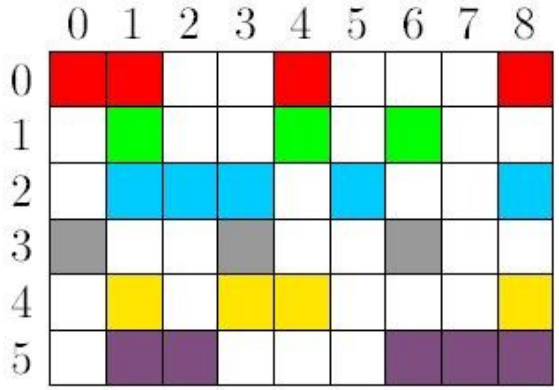
Can **add additional nonzeros freely**.

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

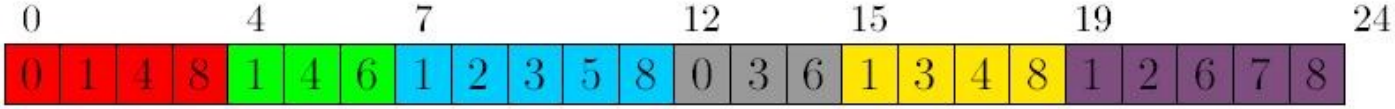


The default matrix representation within PETSc is the general sparse **AIJ format** (Yale sparse matrix or Compressed Sparse Row, CSR)

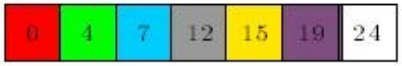
- The nonzero elements are stored by rows
- Array of corresponding column numbers
- Array of pointers to the beginning of each row

value

index



row pointer



Note: The **diagonal matrix entries** are stored with the rest of the nonzeros

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

2. 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

$$\left(\begin{array}{ccc|ccc|cc} 1 & 2 & 0 & 0 & 3 & 0 & 0 & 4 \\ 0 & 5 & 6 & 7 & 0 & 0 & 8 & 0 \\ 9 & 0 & 10 & 11 & 0 & 0 & 12 & 0 \\ \hline 13 & 0 & 14 & 15 & 16 & 17 & 0 & 0 \\ 0 & 18 & 0 & 19 & 20 & 21 & 0 & 0 \\ 0 & 0 & 0 & 22 & 23 & 0 & 24 & 0 \\ \hline 25 & 26 & 27 & 0 & 0 & 28 & 29 & 0 \\ 30 & 0 & 0 & 31 & 32 & 33 & 0 & 34 \end{array}\right)$$

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

2. 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);
[…]
```

PETSc on

plx.cineca.it

fermi.cineca.it

Try it!

PETSc on plx.cineca.it

```
# module load profile/advanced autoload
      PETSc/3.2-p6--openmpi--1.4.4--intel--co-2011.6.233--binary

# echo $PETSC_DIR
/cineca/prod/libraries/PETSc/3.2-p6/openmpi--1.4.4--intel--co-
   2011.6.233--binary

# echo $PETSC_ARCH
linux-intel
```

```
# make
```

```
# cp 1_petsc_hello $CINECA_SCRATCH/dir
# cd $CINECA_SCRATCH/dir

# qsub petscSubmissionScript
```

makefile (plx.cineca.it)

```
PETSC_DIR = /cineca/prod/libraries/PETSc/3.2-p6/openmpi--1.4.4--intel--co-
    2011.6.233--binary

PETSC_ARCH = linux-intel

ALL: 1_petsc_hello

CFLAGS   = -g -O0
CPPFLAGS =

CLEANFILES = 1_petsc_hello

include ${PETSC_DIR}/conf/variables
include ${PETSC_DIR}/conf/rules

1_petsc_hello: 1_petsc_hello.o  chkopts
        -${CLINKER} -o 1_petsc_hello 1_petsc_hello.o  ${PETSC_LIB}
        ${RM} 1_petsc_hello.o
```

PETSc on fermi.cineca.it

```
module purge
module load profile/advanced
module load autoload

module load petsc
module load bgq-xl

echo  ' # $PETSC_DIR:'  $PETSC_DIR
echo  ' # $PETSC_ARCH:'  $PETSC_ARCH
```

```
# make
```

```
# cp 1_petsc_hello $CINECA_SCRATCH/dir
# cd $CINECA_SCRATCH/dir

# llsubmit petscSubmissionScript
```

makefile (fermi.cineca.it)

```
PETSC_DIR = /cineca/prod/libraries/petsc/3.3-p2/bgq-xl--1.0

PETSC_ARCH = bgq-power

ALL: 1_petsc_hello

CFLAGS   = -g -O0
CPPFLAGS =

CLEANFILES = 1_petsc_hello

include ${PETSC_DIR}/conf/variables
include ${PETSC_DIR}/conf/rules

1_petsc_hello: 1_petsc_hello.o  chkopts
        -${CLINKER} -o 1_petsc_hello 1_petsc_hello.o  ${PETSC_LIB}
        ${RM} 1_petsc_hello.o
```

TRY IT

1_petsc_hello

- Download the provided source code and data from the course directory.
- Compile the first example.
- Run it!

2_petsc_vec

- Download, compile and run `2_petsc_vec.c`
- Use PETSc vectors:
- Duplicate a vector
- Create a vector
- Set some vector values (remember `VecAssemblyXXX()` calls!)
- Try some numerical operations

3_petsc_mat

- Download, compile and run 3_petsc_mat.c
- Use PETSc matrices:
- Duplicate a matrix
- Create a matrix
- Set some matrix values (remember MatAssemblyXXX() calls!)
- Try some numerical operations

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

SNES: nonlinear solvers

The SNES class includes methods for solving systems of nonlinear equations of the form

$$F(x) = 0;$$

where F: Rn → Rn.

Newton-like methods provide the core of the package, including both line search and trust region techniques.

```
SNESCreate(MPI Comm comm, SNES *snes)
SNESSetType(SNES snes, SNESType method)
SNESSetFunction(SNES snes, Vec f,
    PetscErrorCode (*FormFunction)
    (SNES snes, Vec x, Vec f, void *ctx), void *ctx)
SNESSolve(SNES snes, Vec b, Vec x)
```

PETSc SNES methods

Method	SNES Type	Options Name	Default Convergence Test
Line search	SNESLS	ls	SNESConverged_LS()
Trust region	SNESTR	tr	SNESConverged_TR()
Test Jacobian	SNESTEST	test	

Debugging and Profiling

Debugging

PETSc programs may be debugged using one of the two options:

`-start_in_debugger` - start all processes in debugger
`-on_error_attach_debugger` - start debugger only on error

Profiling and performance tuning

Profiling:

- Integrated profiling of:
  - time
  - floating-point performance
  - memory usage
  - communication
- User-defined events
- Profiling by stages of an application

Performance Tuning:

- Matrix optimizations
- Application optimizations
- Algorithmic tuning

PETSc profiling options

The profiling options include the following:

`-log_summary` - Prints an ASCII version of performance data at program's conclusion. These statistics are comprehensive and concise and require little overhead; thus, `-log_summary` is intended as the primary means of monitoring the performance of PETSc codes.

`-info [infofile]` - Prints verbose information about code to `stdout` or an optional file. This option provides details about algorithms, data structures, etc. Since the overhead of printing such output slows a code, this option should not be used when evaluating a program's performance.

`-log_trace [logfile]` - Traces the beginning and ending of all PETSc events. This option, which can be used in conjunction with -info, is useful to see where a program is hanging without running in the debugger.

If configured with `--with-debugging=1`; Activated at runtime

Using PETSc with other packages: MATLAB

Using PETSc with other packages: MATLAB

- Dumping files to be read into Matlab
  - **`-vec_view_matlab`** or **`-mat_view_matlab`**

- Automatically sending data from a running PETSc program to a Matlab process where you may interactively type Matlab commands
  - **`PetscViewerSocketOpen()`**
  - **`VecView()`**, **`MatView()`**, **`PetscIntView()`**, ecc

- Automatically sending data back and forth between PETSc and Matlab where Matlab commands are issued not interactively but from a script or the PETSc program.
  - Using the **Matlab Compute Engine**

  - *http://www.mathworks.com*

Using PETSc with other packages: TAO (Toolkit for Advanced Optimization)

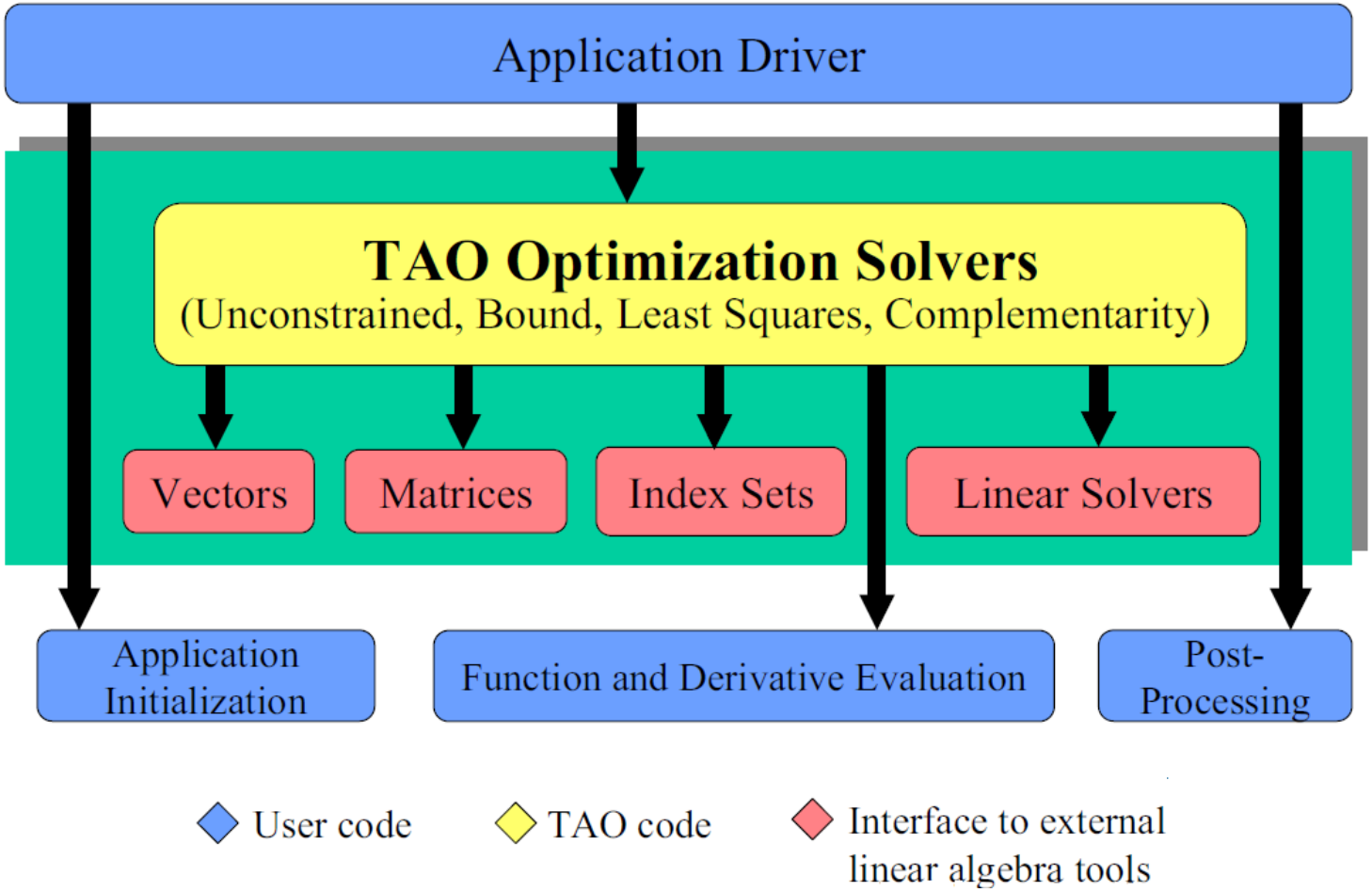
TAO main features

TAO – Toolkit for Advanced Optimization

Is aimed at the solution of large-scale optimization problems on high-performance architectures.

- **ANL** – Argonne National Laboratory
- **Goals**: portability, performance, scalable parallelism, and an interface independent of the architecture
- **C**, **Fortran**, and **C++**
- **Impact**: cancer treatment, laser treatment, artificial intelligence, modelling of polymeric materials, page rank, learning algorithm, I/O tracing, hydraulic conductivities, digital photography, risk minimization, ...

TAO design


Application Driver
TAO Optimization Solvers
(Unconstrained, Bound, Least Squares, Complementarity)
Vectors
Matrices
Index Sets
Linear Solvers
Application Initialization
Function and Derivative Evaluation
Post-Processing
User code
TAO code
Interface to external linear algebra tools

TAO execution flow (1/3)

```
#include “tao.h”

typedef struct { [...] } UserAppCtx;
int UserFunctionGradient(TAO_APPLICATION, Vec, double*, Vec, void*);
int UserHessian(TAO_APPLICATION, Vec, Mat*, Mat*, MatStructure*, void*);

int main(int argc,char **argv){

        Vec x;                  /* solution vector */
        Mat H;                  /* Hessian matrix */
        TAO_SOLVER tao;         /* TAO_SOLVER solver context */
        TAO_APPLICATION taoapp; /* TAO application context */
        UserAppCtx appCtx;    /* user-defined application context */

        PetscInitialize(&argc, &argv, (char *)0, 0);
        TaoInitialize(&argc, &argv, (char *)0, 0);

        /* cont. */
```

TAO execution flow (2/3)

```
    /* cont. */

    VecCreateSeq(PETSC_COMM_SELF, appCtx.n, &x);
    MatCreateSeqBDiag(PETSC_COMM_SELF, appCtx.n, appCtx.n, 0, 2, 0, 0,
&H);

    TaoCreate(PETSC_COMM_SELF, “tao_cg”, &tao);
    TaoApplicationCreate(PETSC_COMM_SELF, &taoapp);

    VecSet(&zero,x);
    TaoAppSetInitialSolutionVec(taoapp,x);

    TaoAppSetObjectiveAndGradientRoutine(taoapp, UserFunctionGradient,
                                         (void *)&appCtx);
    TaoAppSetHessianMat(taoapp, H, H);
    TaoAppSetHessianRoutine(taoapp, UserHessian, (void *)& appCtx);

    /* cont. */
```

TAO execution flow (3/3)

```
        /* cont. */

        TaoSolveApplication(taoapp, tao);

        TaoView(tao);
        TaoGetSolutionStatus(tao, iterate, f, gnorm, cnorm, xdiff, reason);

        TaoDestroy(tao);
        TaoAppDestroy(taoapp);

        VecDestroy(x);
        MatDestroy(H);

        TaoFinalize();
        PetscFinalize();

        return 0;
}
```

References
PETSc, MPI, TAO

References

PETSc Documentation: http://www.mcs.anl.gov/petsc/docs

- PETSc users manual
- Manual pages
- Many hyperlinked examples
- FAQ, Troubleshooting info, installation info, etc.

PETSc Publications: http://www.mcs.anl.gov/petsc/publications

- Research and publications that make use PETSc

MPI Information: http://www.mpi-forum.org

Using MPI (2nd Edition), by Gropp, Lusk, and Skjellum

Domain Decomposition, by Smith, Bjorstad, and Gropp

TAO

http://www.mcs.anl.gov/research/projects/tao/index.html