Introduction to PETSc
Portable, Extensible Toolkit for Scientific Computation

Roberto Gori - r.gori@cineca.it
SuperComputing Applications and Innovation Department
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.

- Essentially Object Oriented code written in C
- Usable from Fortran, C++ and Python (via petsc4py)
- Uses MPI for message-passing
- Uses BLAS and LAPACK for low-level data manipulation
- Can be configured for single or double precision, real or complex scalars
- Interfaces with many other numerical packages
- PETSc has been used for modeling 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 class hierarchy

Level of Abstraction

Application Codes

SNES
(Nonlinear Equations Solvers)

TS
(Time Stepping)

PC
(Preconditioners)

KSP
(Krylov Subspace Methods)

Matrices

Vectors

Index Sets

BLAS

MPI
## PETSc numerical components

### Parallel Numerical Components of PETSc

#### Nonlinear Solvers

<table>
<thead>
<tr>
<th>Newton-based Methods</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line Search</td>
<td>Trust Region</td>
</tr>
</tbody>
</table>

#### Time Steppers

<table>
<thead>
<tr>
<th>General Linear</th>
<th>IMEX</th>
<th>Pseudo–Time Stepping</th>
<th>Runge–Kutta</th>
</tr>
</thead>
</table>

#### Krylov Subspace Methods

<table>
<thead>
<tr>
<th>GMRES</th>
<th>CG</th>
<th>CGS</th>
<th>Bi–CG–Stab</th>
<th>TFQMR</th>
<th>Richardson</th>
<th>Chebychev</th>
<th>Other</th>
</tr>
</thead>
</table>

#### Preconditioners

<table>
<thead>
<tr>
<th>Additive Schwarz</th>
<th>Block Jacobi</th>
<th>Jacobi</th>
<th>ILU</th>
<th>ICC</th>
<th>LU (sequential only)</th>
<th>Other</th>
</tr>
</thead>
</table>

#### Matrices

<table>
<thead>
<tr>
<th>Compressed Sparse Row (AIJ)</th>
<th>Block Compressed Sparse Row (BAIJ)</th>
<th>Symmetric Block Compressed Row (SBAIJ)</th>
<th>Dense</th>
<th>Other</th>
</tr>
</thead>
</table>

#### Vectors

#### Index Sets

<table>
<thead>
<tr>
<th>Indices</th>
<th>Block Indices</th>
<th>Stride</th>
<th>Other</th>
</tr>
</thead>
</table>
PETSc model

Goals
- Portability
- Performance
- Scalable parallelism

Approach
- Object Oriented Delegation Pattern
- Many specific implementations of the same object
- Shared interface
- Command line customization

Benefit
- Most of linear and nonlinear algebra techniques implemented
- Flexibility: easy switch among different implementations
- Nasty details of implementation hidden

Ongoing (development version only)
- GPU, MIC (via OpenCL) and pthread low-level implementations
PETSc object oriented model

- (Almost) all PETSc objects are essentially **delegator objects**
- From Wikipedia: “...an object, instead of performing one of its stated tasks, delegates that task to an associated helper object...”
  

- Example with a XXX object

```c
#include <petscxxx.h>  //Includes the public interface for XXX and other stuff
PetscXXX xxx;
XXXCreate(....,&xxx);  //Initializes the XXX object (no implementation yet)
XXXSetType(xxx,ANY_XXX_TYPE);  //DELEGATION: Sets specific implementation
XXXSetOption(xxx,ANY_XXX_OPTION,XXX_OPTION_VALUE);  //Sets options in DB
XXXAnyCustom(xxx,...);  //Any XXX customization available through the interface
XXXSetFromOptions(xxx);  //Allows options and command line customization
XXXSetUp(xxx);  //Calls specific setup (not all objects need it)
```

- **XXXSetType** calls the specific creation routine `XXXCreate_ANYXXXTYPE(...)`.  
- If `XXXSetType` is called at a later time, the old delegate is freed and `xxx` can be reused with a different low-level implementation.  
- `XXXSetUp`, if needed, closes the setup procedure: `xxx` can then be used.  
- Users can register their own delegates/classes using  
  ```c
  XXXRegister(...,XXXCreate_MYTYPE)
  ```
PETSc from a user perspective

- Home page

- User manual

- Public functions for XXX class (Vec, Mat, KSP, …) accessible at

- Each class has its own set of tutorials which can be compiled and ran
  USE THEM TO LEARN HOW TO DEVELOP WITH PETSc!

- Always use a debug version of PETSc when developing.

- No need to download and install supported external packages separately: PETSc will do this for you if any if the packages is requested at configure time.

- An example:

  ```bash
  $ ./configure --download-mpich=1 --download-mumps=1
  ```
PETSc from a user perspective

**VecCreate**

Creates an empty vector object. The type can then be set with `VecSetType()`, or `VecSetFromOptions()`.

**Synopsis**

```c
#include "petscvec.h"
PetscErrorCode VecCreate(MPI_Comm comm, Vec *vec)
```

If you never call `VecSetType()` or `VecSetFromOptions()` it will generate an error when you try to use the vector.

Collective on `MPI_Comm`

**Input Parameter**

*comm* -The communicator for the vector object

**Output Parameter**

*vec* -The vector object

**Keywords**

vector, create

**See Also**

`VecSetType()`, `VecSetSizes()`, `VecCreateMPIWithArray()`, `VecCreateMPI()`, `VecDuplicate()`,
`VecDuplicateVecs()`, `VecCreateGhost()`, `VecCreateSeq()`, `VecPlaceArray()`

**Level:** beginner
**Location:** `src/vec/vec/interface/veccreate.c`
**Index of all Vec routines**
**Table of Contents for all manual pages**
**Index of all manual pages**

**Examples**

`src/sys/threadcomm/examples/tutorials/ex4.c.html`
`src/vec/vec/examples/tutorial/ex1.c.html`
`src/vec/vec/examples/tutorial/ex2.c.html`
Writing PETSc programs: initialization and finalization

**PetscInitialize**(int *argc, char ***args, const char options_file[], const char help_string[])

- Setup of static data
- Registers all PETSc specific implementations (of all classes)
- Setup of services (logging, error-handling, profiling)
- Setup of MPI (if it is not already been initialized)

**PetscFinalize()**

- Calculates logging summary
- Checks for memory leaks (already allocated mem, if req'ed)
- Finalizes MPI (if PetscInitialize() began MPI)
- Shutdowns all PETSc services
A simple hello world program

```c
#include "petscsys.h"

int main(int argc, char **args) {
    PetscErrorCode ierr;
    PetscMPIInt rank;

    ierr = PetscInitialize(&argc, &args,(char *)0, NULL);CHKERRQ(ierr);
    ierr = MPI_Comm_rank(PETSC_COMM_WORLD, &rank);CHKERRQ(ierr);
    ierr = PetscPrintf(PETSC_COMM_SELF, "Hello by process %d!\n",rank);CHKERRQ(ierr);
    ierr = PetscFinalize();
    return 0;
}
```
A simple hello world program

```fortran
#include "finclude/petsc.h"

program main

PetscErrorCode :: ierr
PetscMPIInt :: rank
character(len=6) :: num
character(len=30) :: hello

call PetscInitialize( PETSC_NULL_CHARACTER, ierr ); CHKERRQ(ierr)
call MPI_Comm_rank( PETSC_COMM_WORLD, rank, ierr ); CHKERRQ(ierr)
write(num,*): rank
hello = 'Hello from process '//num
call PetscPrintf( PETSC_COMM_SELF, hello//uchar(10), ierr ); CHKERRQ(ierr)
call PetscFinalize(ierr)

end program
```
Vec and Mat
Vectors

What are PETSc vectors?

- Roughly represent elements of a Banach space
- Usually they store solutions and right-hand sides.
- Vector elements are **PetscScalars**
- Each process locally owns a subvector of contiguously numbered global indices

Features

- Vector types: STANDARD, PTHREAD and CUSP (dev only)
- Supports all vector space operations
  
  - *VecDot()*, *VecNorm()*, *VecScale()*, ...
- Also unusual ops, like e.g. *VecSqrt()*, *VecReciprocal()*
- Hidden communication of vector values during assembly
- Communications between different parallel vectors
Vector basic interface 1/2

\textbf{VecCreate}(MPI\_Comm \textit{comm}, Vec *\textit{v})

– Automatically generates the appropriate vector type (sequential or parallel) over all processes in \textit{comm}

\textbf{VecSetSizes}(Vec \textit{v}, PetscInt \textit{m}, PetscInt \textit{M})

– Sets local and global sizes

\textbf{VecSetType}(Vec \textit{v}, VecType \textit{type})

– Sets vector type (defines the delegated object)

\textbf{VecSetFromOptions}(Vec \textit{v})

– Configures the vector from the options database

\textbf{VecDuplicate}(Vec \textit{old}, Vec *\textit{new})

– Duplicates the vector (doesn't copy values)
Vector basic interface 2/2

**VecGetSize(Vec v, PetscInt *size)**
- Gets global size of v

**VecGetLocalSize(Vec v, PetscInt *size)**
- Gets local size of v

**VecView(Vec x, PetscViewer v)**
- Prints the content of the vector using the viewer object

**VecCopy(Vec x, Vec y)**
- Copies vector values

**VecSet(Vec x, PetscScalar value)**
- Sets all values of the vector to a specific value

**VecDestroy(Vec *x)**
- Destroys the Vec object
Vector assembly

VecSetValue(Vec x, PetscInt idx, PetscScalar v, InsertMode mode)

VecSetValues(Vec x, PetscInt n, PetscInt *idx, PetscScalar *v, InsertMode mode)

VecAssemblyBegin(Vec x)

VecAssemblyEnd(Vec x)

A three step process

- VecSetValues can be called as many times as the user wants to tell PETSc what values are to be inserted (or added to existing ones) and where
- VecAssemblyBegin starts communications to ensure that values end up where needed (allow other operations, such as some independent computation, to proceed).
- VecAssemblyEnd completes the communication
Vector - Example 1

```
#include "petscvec.h"
...
Vec x;
PetscInt i,N;
PetscMPIInt rank;
PetscScalar value=1.0;
PetscErrorCode ierr;
...
ierr = VecGetSize(x, &N);CHKERRQ(ierr); /* Global size */
ierr = MPI_Comm_rank(PETSC_COMM_WORLD, &rank);CHKERRQ(ierr)
if (rank == 0) { /* Only rank 0 sets all values into the vector */
   for (i=0; i<N; i++) {
      ierr = VecSetValue(x,i,value,INSERT_VALUES);CHKERRQ(ierr);
   }
}
/* data is distributed to the other processes */
ierr = VecAssemblyBegin(x);CHKERRQ(ierr);
ierr = VecAssemblyEnd(x);CHKERRQ(ierr);
/* the vector can then be used */
```
Vector - Example 2

```c
#include "petscvec.h"
...
Vec x;
PetscInt i, low, high;
PetscScalar value=1.0;
PetscErrorCode ierr;
...
ierr = VecGetOwnershipRange(x, &low, &high); CHKERRQ(ierr);
for (i=low; i<high; i++) {
    ierr = VecSetValue(x, i, value, INSERT_VALUES); CHKERRQ(ierr);
}
ierr = VecAssemblyBegin(x); CHKERRQ(ierr);
ierr = VecAssemblyEnd(x); CHKERRQ(ierr);
/* the vector can then be used */
```
### Numerical vector operations

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>VecAXPY(Vec y, PetscScalar a, Vec x);</td>
<td>( y = y + a \times x )</td>
</tr>
<tr>
<td>VecAYPX(Vec y, PetscScalar a, Vec x);</td>
<td>( y = x + a \times y )</td>
</tr>
<tr>
<td>VecWAXPY(Vec w, PetscScalar a, Vec x, Vec y);</td>
<td>( w = a \times x + y )</td>
</tr>
<tr>
<td>VecAXPYB(Vec y, PetscScalar a, PetscScalar b, Vec x);</td>
<td>( y = a \times x + b \times y )</td>
</tr>
<tr>
<td>VecScale(Vec x, PetscScalar a);</td>
<td>( x = a \times x )</td>
</tr>
<tr>
<td>VecDot(Vec x, Vec y, PetscScalar *r);</td>
<td>( r = x' \times y )</td>
</tr>
<tr>
<td>VecTDot(Vec x, Vec y, PetscScalar *r);</td>
<td>( r = x' \times y )</td>
</tr>
<tr>
<td>VecNorm(Vec x, NormType type, PetscReal *r);</td>
<td>( r = |x|_\text{type} )</td>
</tr>
<tr>
<td>VecSum(Vec x, PetscScalar *r);</td>
<td>( r = \sum x_i )</td>
</tr>
<tr>
<td>VecCopy(Vec x, Vec y);</td>
<td>( y = x )</td>
</tr>
<tr>
<td>VecSwap(Vec x, Vec y);</td>
<td>( y = x \text{ while } x = y )</td>
</tr>
<tr>
<td>VecPointwiseMult(Vec w, Vec x, Vec y);</td>
<td>( w_i = x_i \times y_i )</td>
</tr>
<tr>
<td>VecPointwiseDivide(Vec w, Vec x, Vec y);</td>
<td>( w_i = x_i / y_i )</td>
</tr>
<tr>
<td>VecMDot(Vec x, int n, Vec y[], PetscScalar *r);</td>
<td>( r[i] = x' \times y[i] )</td>
</tr>
<tr>
<td>VecMTDot(Vec x, int n, Vec y[], PetscScalar *r);</td>
<td>( r[i] = x' \times y[i] )</td>
</tr>
<tr>
<td>VecMAXPY(Vec y, int n, PetscScalar *a, Vec x[]);</td>
<td>( y = y + \sum_i a_i \times x[i] )</td>
</tr>
<tr>
<td>VecMax(Vec x, int *idx, PetscReal *r);</td>
<td>( r = \max_i x_i )</td>
</tr>
<tr>
<td>VecMin(Vec x, int *idx, PetscReal *r);</td>
<td>( r = \min_i x_i )</td>
</tr>
<tr>
<td>VecAbs(Vec x);</td>
<td>( x_i =</td>
</tr>
<tr>
<td>VecReciprocal(Vec x);</td>
<td>( x_i = 1 / x_i )</td>
</tr>
<tr>
<td>VecShift(Vec x, PetscScalar s);</td>
<td>( x_i = s + x_i )</td>
</tr>
<tr>
<td>VecSet(Vec x, PetscScalar alpha);</td>
<td>( x_i = \alpha )</td>
</tr>
</tbody>
</table>
Working with local vectors

Sometimes it is more efficient to directly access local storage of a PETSc Vec (e.g. in finite difference computations involving vector elements)

VecGetArray(Vec x, PetscScalar *[])
- Access the local storage

VecRestoreArray(Vec x, PetscScalar *[])
- You must return the array to PETSc when you have done computing with local data

PETSc handles data structure conversions (e.g. if data resides on GPU)
- For most common uses, these routines are inexpensive and do not involve a copy of local data.
Vector - Example 3

```
#include "petscvec.h"
...
Vec  vec;
PetscMPIInt  rank;
PetscScalar  *avec;
...
ierr = VecCreate(PETSC_COMM_WORLD,&vec); CHKERRQ(ierr);
ierr = VecSetSizes(vec,PETSC_DECIDE,100); CHKERRQ(ierr);
ierr = VecSetType(vec,VECSTANDARD); CHKERRQ(ierr);
...
ierr = VecGetArray(vec, &avec); CHKERRQ(ierr);
ierr = MPI_Comm_rank(PETSC_COMM_WORLD, &rank); CHKERRQ(ierr);
ierr = PetscPrintf(PETSC_COMM_SELF,"First element of local array for rank %d is %f\n",rank,avec[0]); CHKERRQ(ierr);
ierr = VecRestoreArray(vec, &avec); CHKERRQ(ierr);
...
```
Matrices

What are PETSc matrices?
- Roughly represent linear operators in Banach spaces
- In most of the PETSc low-level implementations, each process logically owns a submatrix of contiguous rows

Features
- Supports many storage formats
  - AIJ, BAIJ, SBAIJ, DENSE, CUSP (GPU, dev-only) ...
- Data structures for many external packages
  - MUMPS (parallel), SuperLU_dist (parallel), SuperLU, UMFPack
- Hidden communications in parallel matrix assembly
- Matrix operations are defined from a common interface
- Shell matrices via user defined MatMult and other ops
Parallel sparse matrices

Each process **logically owns** a matrix subset of contiguously numbered global rows. Each subset consists of two sequential matrices corresponding to **diagonal** and **off-diagonal** parts.

<table>
<thead>
<tr>
<th></th>
<th>P0</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>10</td>
<td>11</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

|     | 13               | 0              | 14             | 15             | 16             | 17             | 0              | 0              |
|     | 0                | 18             | 0              | 19             | 20             | 21             | 0              | 0              |
|     | 0                | 0              | 0              | 22             | 23             | 0              | 24             | 0              |

|     | 25               | 26             | 27             | 0              | 0              | 28             | 29             | 0              |
|     | 30               | 0              | 0              | 31             | 32             | 33             | 0              | 34             |
Matrix operations 1/2

\textbf{MatCreate} (MPI\_Comm \texttt{comm}, \texttt{Mat *A})

- Automatically generates the appropriate matrix type (sequential or parallel) over all processes in \texttt{comm}.

\textbf{MatSetSizes} (\texttt{Mat A}, \texttt{PetscInt m}, \texttt{PetscInt n},
\hspace{1cm} \texttt{PetscInt M}, \texttt{PetscInt N})

- Sets the local and global sizes

\textbf{MatSetType} (\texttt{Mat A}, \texttt{MatType type})

- Sets matrix type (defines the delegated object)

\textbf{MatSetFromOptions} (\texttt{Mat A})

- Configures the matrix from the options database.

\textbf{MatDuplicate} (\texttt{Mat B}, \texttt{MatDuplicateOption op}, \texttt{Mat *A})

- Duplicates a matrix (including or not its nonzeros).
Matrix operations 2/2

MatView(Mat A, PetscViewer v)

– Prints matrix content using the viewer object

MatGetOwnershipRange(Mat A, PetscInt *m, PetscInt* n)

– Gets the first and last (+1) of locally owned rows

MatGetOwnershipRanges(Mat A, const PetscInt ***ranges)

– Gets start and end rows of each process sharing the matrix

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

– Gets global number of rows and columns

MatDestroy(Mat *A)

– Destroys the Mat object
Matrix assembly

Like PETSc vectors, Mat assembling process involves calls to

\textbf{MatSetValue}(\texttt{Mat A, PetscInt idxm, PetscInt idxn, PetscScalar value, InsertMode mode})

\textbf{MatSetValues}(\texttt{Mat A, PetscInt m, PetscInt idxm[], PetscInt n, PetscInt idxn[], PetscScalar values[], InsertMode mode})

\textbf{MatAssemblyBegin}(\texttt{Mat A, MatAssemblyType type})

\textbf{MatAssemblyEnd}(\texttt{Mat A, MatAssemblyType type})
Matrix - Example

#include "petscmat.h"
...
Mat A;
PetscInt cols[3], i, istart, iend;
PetscScalar vals[3];
PetscErrorCode ierr;
...
/* suppose A has been already created and have its type set */
ierr = MatGetOwnershipRange(A,&istart,&iend);CHKERRQ(ierr);
...
vals[0] = -1.0; vals[1] = 2.0; vals[2] = -1.0; /* 1D laplacian stencil */
for (i=istart; i<iend; i++) {
    cols[0] = i-1; cols[1] = i; cols[2] = i+1; /* 1D laplacian stencil */
    ierr = MatSetValues(A,1,&i,3,cols,value,INSERT_VALUES);CHKERRQ(ierr);
}
ierr = MatAssemblyBegin(A,MAT_FLUSH_ASSEMBLY);CHKERRQ(ierr);
ierr = MatAssemblyEnd(A,MAT_FLUSH_ASSEMBLY);CHKERRQ(ierr);
/* all processes contribute to 0,0 entry */
ierr = MatSetValue(A,0,0,vals[0],ADD_VALUES);CHKERRQ(ierr);
ierr = MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
ierr = MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
### Numerical matrix operations

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>MatAXPY(Mat Y, PetscScalar a, Mat X, MatStructure);</code></td>
<td>$Y = Y + a \times X$</td>
</tr>
<tr>
<td><code>MatMult(Mat A, Vec x, Vec y);</code></td>
<td>$y = A \times x$</td>
</tr>
<tr>
<td><code>MatMultAdd(Mat A, Vec x, Vec y, Vec z);</code></td>
<td>$z = y + A \times x$</td>
</tr>
<tr>
<td><code>MatMultTranspose(Mat A, Vec x, Vec y);</code></td>
<td>$y = A^T \times x$</td>
</tr>
<tr>
<td><code>MatMultTransposeAdd(Mat A, Vec x, Vec y, Vec z);</code></td>
<td>$z = y + A^T \times x$</td>
</tr>
<tr>
<td><code>MatNorm(Mat A, NormType type, double *r);</code></td>
<td>$r =</td>
</tr>
<tr>
<td><code>MatDiagonalScale(Mat A, Vec l, Vec r);</code></td>
<td>$A = \text{diag}(l) \times A \times \text{diag}(r)$</td>
</tr>
<tr>
<td><code>MatScale(Mat A, PetscScalar a);</code></td>
<td>$A = a \times A$</td>
</tr>
<tr>
<td><code>MatConvert(Mat A, MatType type, Mat *B);</code></td>
<td>$B = A$</td>
</tr>
<tr>
<td><code>MatCopy(Mat A, Mat B, MatStructure);</code></td>
<td>$B = A$</td>
</tr>
<tr>
<td><code>MatGetDiagonal(Mat A, Vec x);</code></td>
<td>$x = \text{diag}(A)$</td>
</tr>
<tr>
<td><code>MatTranspose(Mat A, MatReuse, Mat* B);</code></td>
<td>$B = A^T$</td>
</tr>
<tr>
<td><code>MatZeroEntries(Mat A);</code></td>
<td>$A = 0$</td>
</tr>
<tr>
<td><code>MatShift(Mat Y, PetscScalar a);</code></td>
<td>$Y = Y + a \times I$</td>
</tr>
</tbody>
</table>
Matrix AIJ format

The default matrix representation within PETSc is the AIJ format, where:

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

<table>
<thead>
<tr>
<th>Value</th>
<th>Index</th>
<th>Row Pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

The diagram illustrates the AIJ format, where:

- Red indicates nonzero elements.
- Yellow indicates the corresponding column numbers.
- Purple indicates the row pointers.
Matrix memory preallocation

Memory **preallocation** is critical for achieving **good performance** during matrix assembly, as this reduces the number of allocations and copies required during the assembling process.

Private representations of PETSc sparse matrices are dynamic data structures: **additional nonzeros can be freely added** (if no preallocation has been explicitly provided).

Dynamically adding many nonzeros
- requires additional memory allocations
- requires copies
→ **kills performances!**
Preallocation of a sequential sparse matrix (1/2)

MatSeqAIJSetPreallocation(Mat A, Petscint nz, PetscInt *nnz)

- Dynamic preallocation if $(nz == 0 \land nnz == PETSC_NULL)$

- Quick and dirty preallocation if $nz$ is set to the maximum number of nonzeros in any row.
  - Fine if the number of nonzeros per row is roughly the same throughout the matrix
Preallocation of sequential sparse matrix (2/2)

- A finer preallocation
  \[ \text{nnz}[0] = \text{<nonzeros in row 0>} \]
  
  \[ \ldots \]
  
  \[ \text{nnz}[m] = \text{<nonzeros in row m>} \]

- If one **underestimates** the actual number of nonzeros in a given row, then during the assembly process PETSc will complain unless otherwise told.
Preallocation of a parallel sparse matrix (1/2)

\textbf{MatMPIAIJSetPreallocation}\texttt{(Mat A,}
\begin{align*}
\text{PetscInt } & \text{dnz,} \\
\text{PetscInt } & \*dnnz, \\
\text{PetscInt } & \text{onz,} \\
\text{PetscInt } & \*onnz)
\end{align*}

• Same logic as before for dynamic allocation
• \texttt{dnz} and \texttt{dnnz} specify preallocation for the diagonal block
• \texttt{onz} and \texttt{onnz} specify preallocation for the off-diagonal block
Preallocation of parallel sparse matrix (2/2)

Process 0
\[
\begin{pmatrix}
1 & 2 & 0 & | & 0 & 3 & 0 & | & 0 & 4 \\
0 & 5 & 6 & | & 7 & 0 & 0 & | & 8 & 0 \\
9 & 0 & 10 & | & 11 & 0 & 0 & | & 12 & 0
\end{pmatrix}
\]

dnz=2, onz=2
dnnz[0]=2, onnz[0]=2

Process 1
\[
\begin{pmatrix}
13 & 0 & 14 & | & 15 & 16 & 17 & | & 0 & 0 \\
0 & 18 & 0 & | & 19 & 20 & 21 & | & 0 & 0 \\
0 & 0 & 0 & | & 22 & 23 & 0 & | & 24 & 0
\end{pmatrix}
\]

dnz=3, onz=2
dnnz[0]=3, onnz[0]=2

Process 2
\[
\begin{pmatrix}
25 & 26 & 27 & | & 0 & 0 & 28 & | & 29 & 0 \\
30 & 0 & 0 & | & 31 & 32 & 33 & | & 0 & 34
\end{pmatrix}
\]

dnz=1, onz=4
dnnz[0]=1, onnz[0]=4
dnnz[1]=1, onnz[1]=4
PETSc solvers: KSP, SNES and TS
• **KSP** (K stands for *Krylov*) objects are used for solving linear systems by means of direct or iterative methods.
• In the iterative case, convergence can be improved by using a suitable **PC** object (preconditioner).
• Almost all iterative methods are implemented.
• Direct solution for parallel square matrices available through external solvers (MUMPS, SuperLU_dist)
• Linear operators set in KSP by using

```c
KSPSetOperators(KSP ksp, Mat amat, Mat pmat, MatStructure matflag)
```
## PETSc KSP methods

<table>
<thead>
<tr>
<th>Method</th>
<th>KSPType</th>
<th>Options Database Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Richardson</td>
<td>KSPRICHARDSON</td>
<td>richardson</td>
</tr>
<tr>
<td>Chebyshev</td>
<td>KSPCHEBYSHEV</td>
<td>chebyshev</td>
</tr>
<tr>
<td>Conjugate Gradient [12]</td>
<td>KSPCG</td>
<td>cg</td>
</tr>
<tr>
<td>BiConjugate Gradient</td>
<td>KSPBICG</td>
<td>bicg</td>
</tr>
<tr>
<td>Generalized Minimal Residual [16]</td>
<td>KSPGMRES</td>
<td>gmres</td>
</tr>
<tr>
<td>Flexible Generalized Minimal Residual</td>
<td>KSPFGMRES</td>
<td>fgmres</td>
</tr>
<tr>
<td>Deflated Generalized Minimal Residual</td>
<td>KSPDGMRES</td>
<td>dgmres</td>
</tr>
<tr>
<td>Generalized Conjugate Residual</td>
<td>KSPGCR</td>
<td>gcr</td>
</tr>
<tr>
<td>BiCGSTAB [19]</td>
<td>KSPBCGS</td>
<td>b cg s</td>
</tr>
<tr>
<td>Conjugate Gradient Squared [18]</td>
<td>KSPCGS</td>
<td>cgs</td>
</tr>
<tr>
<td>Transpose-Free Quasi-Minimal Residual (1) [8]</td>
<td>KSPTFQMR</td>
<td>tfqmr</td>
</tr>
<tr>
<td>Transpose-Free Quasi-Minimal Residual (2)</td>
<td>KSPTCQMR</td>
<td>tcqmr</td>
</tr>
<tr>
<td>Conjugate Residual</td>
<td>KSPCR</td>
<td>cr</td>
</tr>
<tr>
<td>Least Squares Method</td>
<td>KSPLSQR</td>
<td>lsqr</td>
</tr>
<tr>
<td>Shell for no KSP method</td>
<td>KSPPREONLY</td>
<td>preonly</td>
</tr>
</tbody>
</table>
# PETSc PC methods

<table>
<thead>
<tr>
<th>Method</th>
<th>PCType</th>
<th>Options Database Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacobi</td>
<td>PCJACOBI</td>
<td>jacobii</td>
</tr>
<tr>
<td>Block Jacobi</td>
<td>PCBJACOBI</td>
<td>bjacobi</td>
</tr>
<tr>
<td>SOR (and SSOR)</td>
<td>PCSOR</td>
<td>sor</td>
</tr>
<tr>
<td>SOR with Eisenstat trick</td>
<td>PCEISENSTAT</td>
<td>eisenstat</td>
</tr>
<tr>
<td>Incomplete Cholesky</td>
<td>PCICC</td>
<td>icc</td>
</tr>
<tr>
<td>Incomplete LU</td>
<td>PCILU</td>
<td>ilu</td>
</tr>
<tr>
<td>Additive Schwarz</td>
<td>PCASM</td>
<td>asm</td>
</tr>
<tr>
<td>Algebraic Multigrid</td>
<td>PCGAMG</td>
<td>gamg</td>
</tr>
<tr>
<td>Linear solver</td>
<td>PCKSP</td>
<td>ksp</td>
</tr>
<tr>
<td>Combination of preconditioners</td>
<td>PCCOMPOSITE</td>
<td>composite</td>
</tr>
<tr>
<td>LU</td>
<td>PCLU</td>
<td>lu</td>
</tr>
<tr>
<td>Cholesky</td>
<td>PCCHOLESKY</td>
<td>cholesky</td>
</tr>
<tr>
<td>No preconditioning</td>
<td>PCNONE</td>
<td>none</td>
</tr>
<tr>
<td>Shell for user-defined PC</td>
<td>PCSHELL</td>
<td>shell</td>
</tr>
</tbody>
</table>
SNES: nonlinear solvers

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

\[ F(x) = 0, \quad F : \mathbb{R}^n \to \mathbb{R}^n. \]

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

\[
\text{SNESSetFunction(SNES \ snes, Vec \ v,} \\
\quad \text{PetscErrorCode (*SNESFunction)(SNES, Vec, Vec, void*),} \\
\quad \text{void *ctx)}
\]

\[
\text{SNESSetJacobian(SNES \ snes, Mat \ amat, Mat \ pmat,} \\
\quad \text{PetscErrorCode (*SNESJacobianFunction)} \\
\quad (\text{SNES, Vec, Mat*, Mat*, MatStructure*,void *),}} \\
\quad \text{void *ctx)}
\]
### PETSc SNES methods

<table>
<thead>
<tr>
<th>Method</th>
<th>SNESType</th>
<th>Options Name</th>
<th>Default Convergence Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line search</td>
<td>SNESNEWTONLS</td>
<td>newtonls</td>
<td>SNESConverged_NEWTONLS()</td>
</tr>
<tr>
<td>Trust region</td>
<td>SNESNEWTONTR</td>
<td>newtontr</td>
<td>SNESConverged_NEWTONTR()</td>
</tr>
<tr>
<td>Test Jacobian</td>
<td>SNESTEST</td>
<td>test</td>
<td></td>
</tr>
</tbody>
</table>
**TS: time steppers**

**TS** class includes methods for solving systems of linear or nonlinear Ordinary Differential Equations (ODEs) or Differential Algebraic Equations (DAEs), i.e. problems which can be written down as

\[ F(t, u, \dot{u}) = G(t, u), \quad u(t_0) = u_0. \]

The class provides explicits, implicits or semi-implicit methods and the user has to provide functions on how to compute the fundamental pieces of equation (F, G and a Jacobian).
Debugging and Profiling
Debugging

If configured in debug mode (default), PETSc provides large support to error handling, backtracing and memory leak detection for C/C++ codes by simply adhering to very basic guidelines for code developing.

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

Also, if configured with MPICH for the message passing interface and with GNU compilers, PETSc code is completely valgrind-free.
Profiling and performance tuning

Profiling:

- Integrated profiling of:
  - time
  - floating-point performance
  - memory usage
  - communication

- User-defined events

- Profiling by stages of an application
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.

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