An Introduction to PETSc

A user's point of view

Serge Van Criekingen[1]

[1]Maison de la Simulation, Saclay

Grenoble, January 22nd, 2013

Outline

Summary

What is PETSc ?

1. Portable, Extensible Toolkit for Scientific Computation

What is PETSc ?

1. Portable, Extensible Toolkit for Scientific Computation
2. open-source set of C tools for the parallel solution of PDEs, with an emphasis on scalability ($\rightsquigarrow$ 130,000 procs @ANL) specialized in large sparse iterative parallel solvers.

What is PETSc ?

1. Portable, Extensible Toolkit for Scientific Computation
2. open-source set of C tools for the parallel solution of PDEs, with an emphasis on scalability ($\rightsquigarrow$ 130,000 procs @ANL) specialized in large sparse iterative parallel solvers.
3. Interface for C/C++, Fortran, Python (and Matlab)

What is PETSc ?

1. Portable, Extensible Toolkit for Scientific Computation
2. open-source set of C tools for the parallel solution of PDEs, with an emphasis on scalability ($\rightsquigarrow$ 130,000 procs @ANL) specialized in large sparse iterative parallel solvers.
3. Interface for C/C++, Fortran, Python (and Matlab)
4. supports MPI parallelism (i.e., distributed memory), but latest versions: pthreads (shared-memory) and GPU.

What is PETSc ?

1. Portable, Extensible Toolkit for Scientific Computation
2. open-source set of C tools for the parallel solution of PDEs, with an emphasis on scalability ($\rightsquigarrow$ 130,000 procs @ANL) specialized in large sparse iterative parallel solvers.
3. Interface for C/C++, Fortran, Python (and Matlab)
4. supports MPI parallelism (i.e., distributed memory), but latest versions: pthreads (shared-memory) and GPU.
5. Features:
   - parallel vector and matrices
   - data and grid management tools
   - Krylov iterative solvers
   - parallel preconditioners
   - interfaces with external packages
   - Newton-based nonlinear solvers
   - time-stepping ODE solvers

Related Tools

1. Employed in an impressive list of scientific applications and in other packages (a.o.):
   - eigenproblems: SLEPc
   - FEM and FVM: OpenFVM, OOFEM, DEAL.II, libMesh, Feel++

Related Tools

1. Employed in an impressive list of scientific applications and in other packages (a.o.):
   - eigenproblems: SLEPc
   - FEM and FVM: OpenFVM, OOFEM, DEAL.II, libMesh, Feel++
2. Interfaces with (a. o.)
   - direct solvers: PaStiX, MUMPS and SuperLU
   - preconditioner libraries: Hypre, Euclid, Trilinos/ML (multi-level)
   - automatic differentiation tools: ADIC / ADIFOR
   - graph partitioner: ParMeTiS, Party, PTScotch

Related Tools

1. Employed in an impressive list of scientific applications and in other packages (a.o.):
   - eigenproblems: SLEPc
   - FEM and FVM: OpenFVM, OOFEM, DEAL.II, libMesh, Feel++
2. Interfaces with (a. o.) automatic install
   - direct solvers: PaStiX, MUMPS and SuperLU
   - preconditioner libraries: Hypre, Euclid, Trilinos/ML (multi-level)
   - automatic differentiation tools: ADIC / ADIFOR
   - graph partitioner: ParMeTiS, Party, PTScotch

Related Tools

1. Employed in an impressive list of scientific applications and in other packages (a.o.):
   - eigenproblems: SLEPc
   - FEM and FVM: OpenFVM, OOFEM, DEAL.II, libMesh, Feel++
2. Interfaces with (a. o.) automatic install
   - direct solvers: PaStiX, MUMPS and SuperLU
   - preconditioner libraries: Hypre, Euclid, Trilinos/ML (multi-level)
   - automatic differentiation tools: ADIC / ADIFOR
   - graph partitioner: ParMeTiS, Party, PTScotch
3. (Sca)Lapack: for dense matrices

Related Tools

1. Employed in an impressive list of scientific applications and in other packages (a.o.):
   - eigenproblems: SLEPc
   - FEM and FVM: OpenFVM, OOFEM, DEAL.II, libMesh, Feel++
2. Interfaces with (a. o.) automatic install
   - direct solvers: PaStiX, MUMPS and SuperLU
   - preconditioner libraries: Hypre, Euclid, Trilinos/ML (multi-level)
   - automatic differentiation tools: ADIC / ADIFOR
   - graph partitioner: ParMeTiS, Party, PTScotch
3. (Sca)Lapack: for dense matrices
4. Trilinos: C++, “bigger” package, less integrated (“loose confederation of interoperable packages”)

Related Tools

1. Employed in an impressive list of scientific applications and in other packages (a.o.):
   - eigenproblems: SLEPc
   - FEM and FVM: OpenFVM, OOFEM, DEAL.II, libMesh, Feel++
2. Interfaces with (a. o.) automatic install
   - direct solvers: PaStiX, MUMPS and SuperLU
   - preconditioner libraries: Hypre, Euclid, Trilinos/ML (multi-level)
   - automatic differentiation tools: ADIC / ADIFOR
   - graph partitioner: ParMeTiS, Party, PTScotch
3. (Sca)Lapack: for dense matrices
4. Trilinos: C++, “bigger” package, less integrated (“loose confederation of interoperable packages”)
5. ACTS (Advanced CompuTational Software) collection (U.S. DOE) includes PETSc, SuperLU, SLEPc, Hypre, ...

Documentation on PETSc

1. Extensive documentation on `www.mcs.anl.gov/petsc/`:
   - Manual page for all routines
   - Examples
   - Introduction and tutorials by developers

   ⇒ This talk will focus on my user experience.

Documentation on PETSc

1. Extensive documentation on `www.mcs.anl.gov/petsc/`:
   - Manual page for all routines
   - Examples
   - Introduction and tutorials by developers
     ⇒ This talk will focus on my user experience.

2. Fast support from `petsc-maint@mcs.anl.gov`

Summary

Hello World in C (only from first processor)

```
#include "petsc.h"
int main( int argc, char *argv[] ){
  PetscErrorCode ierr;
  PetscInitialize( &argc, &argv, 0, 0 );
  ierr = PetscPrintf( PETSC_COMM_WORLD, "Hello World\n" );
                                                CHKERRQ(ierr);
  PetscFinalize();
  return 0;
}
```

Hello World in Fortran (only from first processor)

```
program test
  implicit none
#include "finclude/petsc.h"
  PetscErrorCode :: ierr
  call PetscInitialize(PETSC_NULL_CHARACTER,ierr)
  call PetscPrintf(PETSC_COMM_WORLD,"Hello World \n",ierr);
                                                  CHKERRQ(ierr)
  call PetscFinalize(ierr)
end program test
```

Hello World in C and Fortran (only from first processor)

```
#include "petsc.h"
int main( int argc, char *argv[] ){
  PetscInitialize( &argc, &argv, 0, 0 );
  PetscPrintf( PETSC_COMM_WORLD, "Hello World\n" );
  PetscFinalize();
  return 0;
}
```

Hello World in C and Fortran (only from first processor)

```c
#include "petsc.h"
int main( int argc, char *argv[] ){
  PetscInitialize( &argc, &argv, 0, 0 );
  PetscPrintf( PETSC_COMM_WORLD, "Hello World\n" );
  PetscFinalize();
  return 0;
}
```

```
program test
  implicit none
#include "finclude/petsc.h"
  PetscErrorCode :: ierr
  call PetscInitialize(PETSC_NULL_CHARACTER,ierr)
  call PetscPrintf(PETSC_COMM_WORLD,"Hello World \n",ierr)
  call PetscFinalize(ierr)
end program test
```

Notes

- `ierr` (type `PetscErrorCode`) required at the end of each Fortran call
- `PETSC_COMM_WORLD` can be a subset of `MPI_COMM_WORLD`

Notes

- `ierr` (type `PetscErrorCode`) required at the end of each Fortran call
- `PETSC_COMM_WORLD` can be a subset of `MPI_COMM_WORLD`
- `PetscInitialize` contains `MPI_Init`

  `PetscFinalize` contains `MPI_Finalize`

Notes

- `ierr` (type `PetscErrorCode`) required at the end of each Fortran call
- `PETSC_COMM_WORLD` can be a subset of `MPI_COMM_WORLD`
- `PetscInitialize` contains `MPI_Init`
  `PetscFinalize` contains `MPI_Finalize`

  In general: MPI calls are "hidden" by PETSc

Notes

- `ierr` (type `PetscErrorCode`) required at the end of each Fortran call
- `PETSC_COMM_WORLD` can be a subset of `MPI_COMM_WORLD`
- `PetscInitialize` contains `MPI_Init`
  `PetscFinalize` contains `MPI_Finalize`

  In general: MPI calls are "hidden" by PETSc

  Exceptions: `MPI_Comm_size` and `MPI_Comm_rank`

Back to Hello World (with parallel output)

```
[...]
MPI_Comm_rank(PETSC_COMM_WORLD, &myrank);
PetscSynchronizedPrintf(PETSC_COMM_WORLD,
                "Rank %d says hello \n", myrank );
PetscSynchronizedFlush(PETSC_COMM_WORLD);
[...]
```

yields with `mpirun -np 3`:

```
Rank 1 says hello
Rank 2 says hello
Rank 3 says hello
```

Back to Hello World (with parallel output)

```
[...]
MPI_Comm_rank(PETSC_COMM_WORLD, &myrank);
PetscSynchronizedPrintf(PETSC_COMM_WORLD,
                "Rank %d says hello \n", myrank );
PetscSynchronizedFlush(PETSC_COMM_WORLD);
[...]
```

yields with `mpirun -np 3`:

```
Rank 1 says hello
Rank 2 says hello
Rank 3 says hello
```

(More complicated in Fortran)

Summary

PETSc Vectors

1. Create

```
VecCreateSeq(PETSC_COMM_SELF, int m, Vec *x);
```

```
VecCreateMPI(MPI_Comm comm, int m, int M, Vec *x);
```

where

- `m` = local size, or `PETSC_DECIDE` if `M` given
- `M` = global size, or `PETSC_DETERMINE` if `m` given for all ranks

PETSc Vectors

1. Create

```
VecCreateSeq(PETSC_COMM_SELF, int m, Vec *x);
```

```
VecCreateMPI(MPI_Comm comm, int m, int M, Vec *x);
```

In Fortran:

```
call VecCreateSeq(PETSC_COMM_SELF, int m, Vec x,
                                        PetscErrorCode ierr)
```

```
call VecCreateMPI(MPI_Comm comm, int m, int M, Vec x,
                                        PetscErrorCode ierr)
```

PETSc Vectors

1. Create
2. Set

```
VecSetValues(Vec x,int n,int *indices,
                    PetscScalar *values,INSERT_VALUES);
```

```
VecSetValues(Vec x,int n,int *indices,
                    PetscScalar *values,ADD_VALUES);
```

PETSc Vectors

1. Create
2. Set

```
VecSetValues(Vec x,int n,int *indices,
                    PetscScalar *values,INSERT_VALUES);
```

```
VecSetValues(Vec x,int n,int *indices,
                    PetscScalar *values,ADD_VALUES);
```

Notes: • 0-based global indices in C and Fortran

PETSc Vectors

1. Create
2. Set

```
VecSetValues(Vec x,int n,int *indices,
                    PetscScalar *values,INSERT_VALUES);
```

```
VecSetValues(Vec x,int n,int *indices,
                    PetscScalar *values,ADD_VALUES);
```

Notes:
- 0-based global indices in C and Fortran
- In Fortran: `int indices(n), PetscScalar values(n)`

PETSc Vectors

1. Create
2. Set

```
VecSetValues(Vec x,int n,int *indices,
                    PetscScalar *values,INSERT_VALUES);
```

```
VecSetValues(Vec x,int n,int *indices,
                    PetscScalar *values,ADD_VALUES);
```

Notes:
- 0-based global indices in C and Fortran
- In Fortran: `int indices(n), PetscScalar values(n)`
- Faster if `n` large

PETSc Vectors

1. Create
2. Set

```
VecSetValues(Vec x,int n,int *indices,
                    PetscScalar *values,INSERT_VALUES);
```

```
VecSetValues(Vec x,int n,int *indices,
                    PetscScalar *values,ADD_VALUES);
```

3. Assemble (allow overlap of communication and calculation)

```
VecAssemblyBegin(Vec x);
VecAssemblyEnd(Vec x);
```

PETSc Vectors

1. Create
2. Set

```
VecSetValues(Vec x,int n,int *indices,
                    PetscScalar *values,INSERT_VALUES);
```

```
VecSetValues(Vec x,int n,int *indices,
                    PetscScalar *values,ADD_VALUES);
```

3. Assemble (allow overlap of communication and calculation)

```
VecAssemblyBegin(Vec x);
VecAssemblyEnd(Vec x);
```

Caution: `INSERT_VALUES` and `ADD_VALUES` can not be mixed
(call assembly routines inbetween).

PETSc Vectors

4. Get (local only!)
    1. Single Value: `VecGetValues` (use global numbering)

PETSc Vectors

4. Get (local only!)
    1. Single Value: `VecGetValues` (use global numbering)
    2. All local elements: `VecGetArray` (no copy made, time-efficient)

PETSc Vectors

4. Get (local only!)
    1. Single Value: `VecGetValues` (use global numbering)
    2. All local elements: `VecGetArray` (no copy made, time-efficient)

```
VecGetArray(Vec v,PetscScalar **array);
...
VecRestoreArray(Vec v,PetscScalar **array);
```

PETSc Vectors

4. Get (local only!)
   1. Single Value: `VecGetValues` (use global numbering)
   2. All local elements: `VecGetArray` (no copy made, time-efficient)

```
VecGetArray(Vec v,PetscScalar **array);
...
VecRestoreArray(Vec v,PetscScalar **array);
```

```
call VecGetArray(Vec v,PetscScalar vv(1),
                 PetscOffset offset, PetscErrorCode ierr)
...  vv(offset + i) ...
call VecRestoreArray(...)
```

PETSc Vectors

4. Get (local only!)
   1. Single Value: `VecGetValues` (use global numbering)
   2. All local elements: `VecGetArray` (no copy made, time-efficient)

```
VecGetArray(Vec v,PetscScalar **array);
...
VecRestoreArray(Vec v,PetscScalar **array);
```

```
call VecGetArray(Vec v,PetscScalar vv(1),
                 PetscOffset offset, PetscErrorCode ierr)
...  vv(offset + i) ...
call VecRestoreArray(...)
```

```
call VecGetArrayF90(Vec v, PetscScalar pointer vv,
                                    PetscErrorCode ierr)
...
call VecRestoreArrayF90(...)
```

PETSc Vectors

4. Get (local only!)
   1. Single Value: `VecGetValues` (use global numbering)
   2. All local elements: `VecGetArray` (no copy made, time-efficient)

5. Operations

`VecAXPY`	$y = a * x + y,$
`VecDot`	$x \cdot y,$
`VecNorm`	$\|A\|_{...}$
$\vdots$	$\vdots$

PETSc Vectors

4. Get (local only!)
   1. Single Value: `VecGetValues` (use global numbering)
   2. All local elements: `VecGetArray` (no copy made, time-efficient)
5. Operations
6. `VecView, VecDuplicate, VecGetOwnershipRange, VecDestroy, ...`

Summary

PETSc Matrices

1. Create (sparse)

```
MatCreateAIJ(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);
```

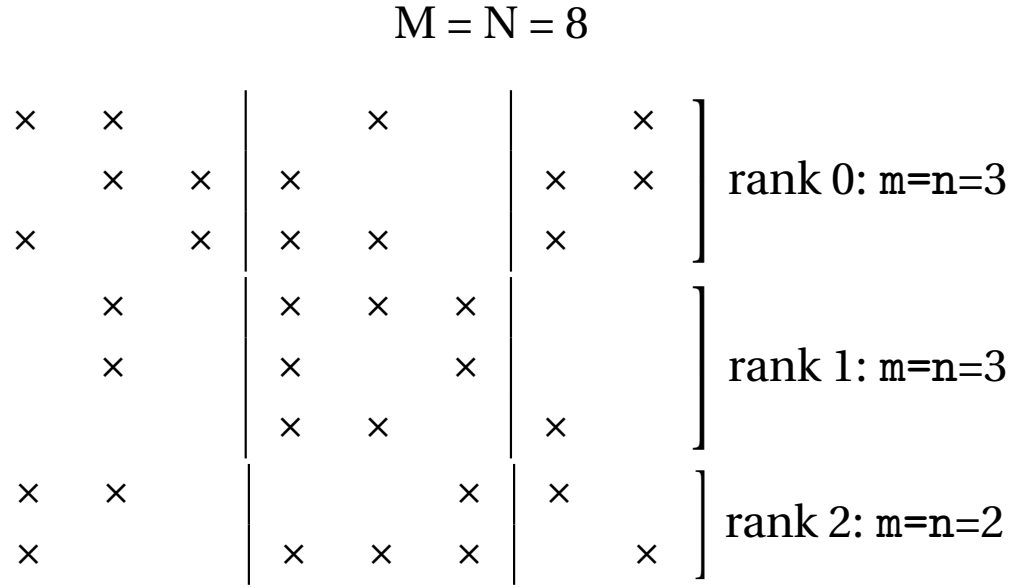
where (Matrix stored by rows)

- `m` = local number of rows, or `PETSC_DECIDE` if `M` given
- `M` = global number of rows, or `PETSC_DETERMINE` if `m` given for all ranks
- `n` = `m` if square (in general: such that Mat-Vec product OK)
- `N` = global number of columns, or `PETSC_DETERMINE` if `n` given for all ranks

PETSc Matrices

1. Create (sparse)

```
MatCreateAIJ(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);
```

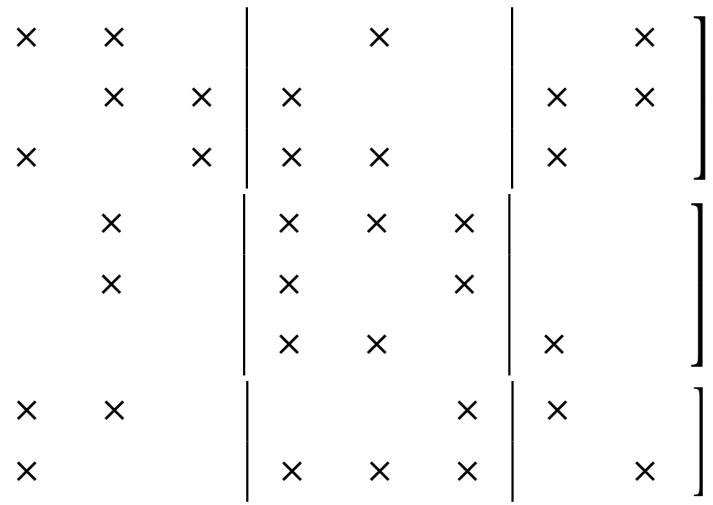


PETSc Matrices

1. Create (sparse)

```
MatCreateAIJ(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);
```

$M = N = 8$



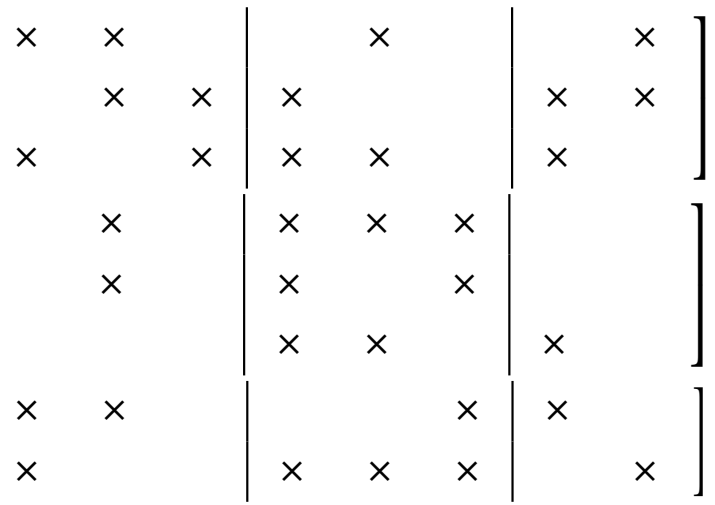
```
d_nnz={2,2,2}, o_nnz={2,3,3}
d_nnz={3,2,2}, o_nnz={1,1,1}
d_nnz={1,1},   o_nnz={3,4}
```

PETSc Matrices

1. Create (sparse)

```
MatCreateAIJ(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);
```

$M = N = 8$



`d_nnz={2,2,2}, o_nnz={2,3,3}`

`d_nnz={3,2,2}, o_nnz={1,1,1}`

`d_nnz={1,1}, o_nnz={3,4}`

- `d_nnz` specified ⇒ `d_nz` ignored
- `d_nnz = PETSC_NULL` & `d_nz` ≥ max{ `d_nnz` }

PETSc Matrices

1. Create

```
MatCreateAIJ(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);
```

Instead of `AIJ`, one can also use `Dense` and `BAIJ`

PETSc Matrices

1. Create

```
MatCreateAIJ(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);
```

Instead of `AIJ`, one can also use `Dense` and `BAIJ`
Sequential versions also available

PETSc Matrices

1. Create
2. Set

```
MatSetValues(..., INSERT_VALUES);
```

```
MatSetValues(..., ADD_VALUES);
```

PETSc Matrices

1. Create
2. Set

```
MatSetValues(..., INSERT_VALUES);
```

```
MatSetValues(..., ADD_VALUES);
```

If `BAIJ`, use `MatSetValuesBlocked`

PETSc Matrices

1. Create
2. Set

```
MatSetValues(..., INSERT_VALUES);
```

```
MatSetValues(..., ADD_VALUES);
```

3. Assemble

```
MatAssemblyBegin(Mat A, MatAssemblyType maType);
MatAssemblyEnd(Mat A, MatAssemblyType maType);
```

where `maType` is

MAT_FLUSH_ASSEMBLY between `INSERT_VALUES` and `ADD_VALUES`,
MAT_FINAL_ASSEMBLY before using the matrix.

PETSc Matrices

4. Operations

`MatAXPY`	$Y = a * X + Y,$
`MatMult`	$y = Ax,$
`MatMultAdd`	$x_3 = x_2 + Ax_1,$
`MatNorm`	$\|A\|_{...}$
`MatTranspose`	A^T
$\vdots$	$\vdots$

PETSc Matrices

4. Operations
5. Matrix-Free

```
extern int mult(Mat,Vec,Vec);
MatCreateShell(comm,m,n,M,N,ctx,&mat);
MatShellSetOperation(mat,MATOP_MULT,(void(*)(void))mult);
```

PETSc Matrices

4. Operations
5. Matrix-Free

```
extern int mult(Mat,Vec,Vec);
MatCreateShell(comm,m,n,M,N,ctx,&mat);
MatShellSetOperation(mat,MATOP_MULT,(void(*)(void))mult);
```

6. `MatView`, `MatDuplicate`, `MatGetOwnershipRange`, `MatDestroy`, ...

Summary

PETSc Krylov Solvers (KSP)

1. Create

```
KSPCreate(MPI_Comm comm, KSP *ksp);
```

PETSc Krylov Solvers (KSP)

1. Create

```
KSPCreate(MPI_Comm comm, KSP *ksp);
```

2. Set

```
KSPSetType(KSP ksp, KSPType kspType);
KSPSetOperators(KSP ksp, Mat A, Mat PrecondBase,
                                MatStructure flag);
KSPSetUp(KSP ksp);
```

where `kspType` = `KSPCG, KSPGMRES, ...`

`flag` = `SAME_PRECONDITIONER, SAME_NONZERO_PATTERN, DIFFERENT_NONZERO_PATTERN`

PETSc Krylov Solvers (KSP)

1. Create

```
KSPCreate(MPI_Comm comm, KSP *ksp);
```

2. Set

```
KSPSetType(KSP ksp, KSPType kspType);
KSPSetOperators(KSP ksp, Mat A, Mat PrecondBase,
                                MatStructure flag);
KSPSetUp(KSP ksp);
```

3. Solve A x = b

```
KSPSolve(KSP ksp,Vec b,Vec x);
```

PETSc Krylov Solvers (KSP) and Preconditioning (PC)

4. Preconditioning

```
KSPgetPC(KSP ksp,PC *pc);
PCSetType(PC pc, PCType PCJACOBI);
PCSetUp(PC pc);
```

Also: `KSPSetTolerances, KSPGetConvergedReason, KSPGetIterationNumber, KSPGetResidualNorm, ...`

PETSc Krylov Solvers (KSP) and Preconditioning (PC)

4. Preconditioning

```
KSPgetPC(KSP ksp,PC *pc);
PCSetType(PC pc, PCType PCSOR);
PCSetUp(PC pc);
```

Also: `KSPSetTolerances, KSPGetConvergedReason, KSPGetIterationNumber, KSPGetResidualNorm, ...`

PETSc Krylov Solvers (KSP) and Preconditioning (PC)

4. Preconditioning

```
KSPgetPC(KSP ksp,PC *pc);
PCSetType(PC pc, PCType PCILU);
PCFactorSetLevels(PC pc, int level_of_fill);
PCSetUp(PC pc);
```

PETSc Krylov Solvers (KSP) and Preconditioning (PC)

4 Preconditioning

```
KSPgetPC(KSP ksp,PC *pc);
PCSetType(PC pc, PCType PCASM);
PCASMSetOverlap(PC pc, int overlap);
PCSetUp(PC pc);
PCASMGetSubKSP(PC pc, int *n_local, int *first_local,
                                        KSP *subKSP[]);
for (i=0; i<nlocal; i++){
      KSPSetType(subKSP(i), KSPtype GMRES);
      KSPGetPC(KSP subKSP(i),PC *subPC);
      PCSetType(PC subPC, PCtype PCSOR);
}
```

PETSc Krylov Solvers (KSP) and Preconditioning (PC)

4. Preconditioning

```
KSPgetPC(KSP ksp,PC *pc);
PCSetType(PC pc, PCType PCASM);
PCASMSetOverlap(PC pc, int overlap);
PCSetUp(PC pc);
PCASMGetSubKSP(PC pc, int *n_local, int *first_local,
                                        KSP *subKSP[]);
for (i=0; i<nlocal; i++){
      KSPSetType(subKSP(i), KSPtype KSPPREONLY);
      KSPGetPC(KSP subKSP(i),PC *subPC);
      PCSetType(PC subPC, PCtype PCSOR);
}
```

NB: KSPPREONLY = defaut "sub-type"

PETSc Krylov Solvers (KSP) and Preconditioning (PC)

6. Direct Method

```
KSPSetType(KSP ksp, KSPType KSPPREONLY);
KSPgetPC(KSP ksp,PC *pc);
PCSetType(PC pc, PCType PCLU);
```

PETSc Krylov Solvers (KSP) and Preconditioning (PC)

6. Direct Method

```
KSPSetType(KSP ksp, KSPType KSPPREONLY);
KSPgetPC(KSP ksp,PC *pc);
PCSetType(PC pc, PCType PCLU);
```

7. External Solvers

```
PCFactorSetMatSolverPackage(PC pc,
                            MatSolverPackage MATSOLVERPASTIX);
```

PETSc Krylov Solvers (KSP) and Preconditioning (PC)

6. Direct Method

```
KSPSetType(KSP ksp, KSPType KSPPREONLY);
KSPgetPC(KSP ksp,PC *pc);
PCSetType(PC pc, PCType PCLU);
```

7. External Solvers

```
PCFactorSetMatSolverPackage(PC pc,
                            MatSolverPackage MATSOLVERMUMPS);
```

PETSc Krylov Solvers (KSP) and Preconditioning (PC)

6. Direct Method
7. External Solvers
8. For run-time specification:

```
KSPSetFromOptions(KSP ksp);
```

PETSc Krylov Solvers (KSP) and Preconditioning (PC)

6. Direct Method
7. External Solvers
8. For run-time specification:

```
KSPSetFromOptions(KSP ksp);
```

e.g., `-ksp_rtol <rtol> -ksp_type <method>`

PETSc Krylov Solvers (KSP) and Preconditioning (PC)

6. Direct Method
7. External Solvers
8. For run-time specification:

```
KSPSetFromOptions(KSP ksp);
```

```
PCSetFromOptions(PC pc);
```

Summary

PETSc main “categories”

- Vec: vectors
- Mat: matrices
- KSP: Krylov subspace iterative methods
- PC: preconditioners

PETSc main “categories”

- Vec: vectors
- Mat: matrices
- KSP: Krylov subspace iterative methods
- PC: preconditioners
- DM: Data Management between meshes and vectors/matrices, used to build grids, also unstructured.

PETSc main "categories"

- Vec: vectors
- Mat: matrices
- KSP: Krylov subspace iterative methods
- PC: preconditioners
- DM: Data Management between meshes and vectors/matrices, used to build grids, also unstructured.
- SNES: Scalable Nonlinear Equations Solvers (Newton-like methods)

PETSc main “categories”

- Vec: vectors
- Mat: matrices
- KSP: Krylov subspace iterative methods
- PC: preconditioners
- DM: Data Management between meshes and vectors/matrices, used to build grids, also unstructured.
- SNES: Scalable Nonlinear Equations Solvers (Newton-like methods)
- TS: Time-Stepping (for ODE)

Conclusions

- PETSc is powerful and widely used.

Conclusions

- PETSc is powerful and widely used.
- Once set-up, PETSc enables to try a lot of different solution methods.

Conclusions

- PETSc is powerful and widely used.
- Once set-up, PETSc enables to try a lot of different solution methods.
- Want to know more: come to the training!

  By Jérémy Foulon, Loïc Gouarin, Serge Van Criekingen

  - In March: from 18 to 20 at the Maison de la Simulation (Saclay)
  - In May-June: in Lyon area

Conclusions

- PETSc is powerful and widely used.
- Once set-up, PETSc enables to try a lot of different solution methods.
- Want to know more: come to the training!

  By Jérémy Foulon, Loïc Gouarin, Serge Van Criekingen

  - In March: from 18 to 20 at the Maison de la Simulation (Saclay)
  - In May-June: in Lyon area

  "Bilingual" training: exercises in C or Fortran

  More information: www.maisondelasimulation.fr