Introduction to PETSc

Remi.Lacroix@idris.fr, Dimitri.Lecas@idris.fr, Serge.Van.Criekingen@idris.fr

CNRS — IDRIS

June 2019

Outline I

Outline

What is PETSc ?

- Portable, Extensible Toolkit for Scientific Computation
- Open-source set of C tools for the parallel solution of PDEs, with an emphasis on scalability and specialized in large sparse iterative parallel solvers.
- Developed at Argonne National Laboratory since 1991
- Currently about 15 active developers
- Interface for C/C++, Fortran, Python
- Supports MPI parallelism (no multithreading). GPU version available.

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
- Support for profiling, debugging and graphical output

Features (continued)

- Debugging and optimized versions available through the compilation option `--with-debugging=0/1`
- Supports real/complex floating-point arithmetic, single/double precision, with the type `PetscScalar` depending on compilation options:
  `--with-scalar-type=real/complex`
  `--with-precision=single/double` (also `__float128` / `__fp16`)
- The type `PetscReal` is the real part of `PetscScalar`.
- The type `PetscInt` can be used to represent size of arrays and indexing into arrays. Its size is 32-bit by default, and 64-bit with the compilation option `--with-64-bit-indices`.

Ecosystem

- Interfaces with (among others)
  - direct solvers: `PaStiX, MUMPS, SuperLU`
  - preconditioner libraries: `Hypre, Trilinos/ML` (multi-level)
  - graph partitioner: `ParMeTiS, PTScotch`
- Alternative: `Trilinos` (`C++`; bigger, less integrated package)
- Employed in many scientific applications and in other packages. For instance:
  - Eigenproblems: `SLEPc`
  - Finite Element packages: `Feel++, FEniCS, Firedrake, ...`

Support

- Website: http://www.mcs.anl.gov/petsc/
  - Manual page for all routines
  - Examples
  - Introduction and tutorials by developers
- Mailing lists:
  - for maintainance/bug-report: petsc-maint@mcs.anl.gov
  - for users: petsc-users@mcs.anl.gov
  - for developers: petsc-dev@mcs.anl.gov

Hello World (C)

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


Hello World (C)

- Do not forget the **#include** <petsc.h>.
- Error code should be checked (CHKERRQ(ierr)) after each PETSc call.
- PetscInitialize executes MPI_Init if not done before.
  In this case PetscFinalize also executes MPI_Finalize.
- PETSC_COMM_WORLD can be a subset of MPI_COMM_WORLD.
- In general: MPI calls are "hidden" by PETSc.
- Exceptions: MPI_Comm_size and MPI_Comm_rank
- PetscPrintf prints to standard output, only from the first processor in the communicator.
  To have output from several processors, use PetscSynchronizedPrintf and PetscSynchronizedFlush.

Hello World (C)

```c
#include <petsc.h>
int main( int argc, char* argv[] ){
  PetscErrorCode ierr;
  int rank, size;
  PetscInitialize( &argc, &argv, PETSC_NULL, PETSC_NULL );
  MPI_Comm_rank(PETSC_COMM_WORLD, &rank);
  MPI_Comm_size(PETSC_COMM_WORLD, &size);
  ierr = PetscSynchronizedPrintf(PETSC_COMM_WORLD,
                "Rank %d out of %d says hello \n", rank, size );
  CHKERRQ(ierr);
  ierr = PetscSynchronizedFlush(PETSC_COMM_WORLD, PETSC_STDOUT);
  CHKERRQ(ierr);
  PetscFinalize();
  return 0;
}
```

Yields with mpirun -np 3 :

```
Rank 0 out of 3 says hello
Rank 1 out of 3 says hello
Rank 2 out of 3 says hello
```


Hello World (Fortran)

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

Hello World (`Fortran`)

- Error code must be present as last argument of each `Fortran` call.
- Difference `C`/`Fortran` relatively minor. Only significant ones will be detailed in this class.

Makefile

```
ALL: myCode

PETSC_DIR=/.../petsc-3.11.2/real-debug-mumps-hypre
PETSC_ARCH=

SOURCES     = myCode.c
OBJ         = $(SOURCES:.c=.o)
EXE         = myCode.exe
CLEANFILES = ${OBJ} ${EXE}

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

myCode: ${OBJ} chkopts
        ${CLINKER} -o ${EXE} ${OBJ} ${PETSC_LIB}
```

PETSc components

PETSc objects to be handled through functions : Vec, Mat, KSP, PC,...

For instance:

- Create
- SetType
- Destroy
- ...

Outline

PETSc Vectors: Types & Create

A vector in PETSc is an object of type Vec.

Two basic types: sequential and parallel (MPI-based)

```
VecCreateSeq(MPI_Comm comm, PetscInt m, Vec* x);
```

where

- comm = PETSC_COMM_SELF
- m = (local) size

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

where

- comm = the MPI communicator (often PETSC_COMM_WORLD)
- m = local size, or PETSC_DECIDE if M given
- M = global size, or PETSC_DETERMINE if m given for all ranks

PETSc Vectors: Destroy/Duplicate/Copy

```
VecDuplicate(Vec x, Vec* y)
```

y created with same type as *x*; storage allocated for *y* but values *not copied*.

```
VecCopy(Vec x, Vec y)
```

$y \leftarrow x$ (y pre-existing)

```
VecDestroy(Vec* v)
```

PETSc Vectors: GetSize/GetLocalSize/GetOwnershipRange

```
VecGetSize(Vec x, PetscInt* size)
```

```
VecGetLocalSize(Vec x, PetscInt* size)
```

```
VecGetOwnershipRange(Vec x, PetscInt* istart, PetscInt* iend)
```

Get the range of indices owned by each processor.

Warning: 0-based indices even in `Fortran`.

PETSc Vectors: Set Value(s)

```
VecSet(Vec x, PetscScalar value)
```

```
VecSetValue(Vec x, PetscInt row, PetscScalar value,
            INSERT_VALUES or ADD_VALUES);
```

```
VecSetValues(Vec x, PetscInt n,
             const PetscInt indices[], const PetscScalar values[],
             INSERT_VALUES or ADD_VALUES);
```

PETSc Vectors: Set Value(s) (continued)

Notes:

- *Global* indices have to be used in `VecSetValue` and `VecSetValues`.
  To use *local* indices:
  `VecSetValueLocal` and `VecSetValuesLocal`.
- Always 0-based indices in `C` and `Fortran`.
- `VecSetValues` faster than `VecSetValue`.
  `VecSetValues` fastest if `n` large.

PETSc Vectors: Assemble

After using `VecSetValue` or `VecSetValues`, one must assemble the vector:

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

Note: allows overlap of communication and computation.

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

PETSc Vectors: Get Value(s)

One can pull only local values from a vector.

- Specific values → `VecGetValues` (use global numbering)

```
VecGetValues(Vec x, PetscInt n, const PetscInt indices[],
             PetscScalar y[])
```

Note: values are copied in y; y must be pre-allocated.

- All local elements → `VecGetArray` / `VecRestoreArray`

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

Notes:

- values are NOT copied; provides direct acces to `Vec` values
- more time-efficient than `VecGetValues`
- can be used to set `Vec` values (more time-efficient than `VecSetValue(s)` - see exercice 2)

PETSc Vectors: Get Value(s)

In `Fortran`:

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

In `Fortran90`:

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

PETSc Vectors: View

```
VecView(Vec x, PETSC_VIEWER_STDOUT_WORLD);
```

PETSC_VIEWER_STDOUT_WORLD $\equiv$ synchronized standard output.

Other visualization contexts: see on-line documentation.

PETSc Vectors: Operations

VecScale	$x = a * x,$
VecAXPY	$y = a * x + y,$
VecDot	$x \cdot y,$
VecPointwiseMult	$w_i = x_i * y_i$
VecNorm	$\|A\|...$
$\vdots$	$\vdots$

PETSc Vectors: Types & Create (other ways)

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

Other way:

```
VecCreate(MPI_Comm comm, Vec* x);
VecSetType(Vec x, VECSEQ/VECMPI);
VecSetSizes(Vec x, PetscInt m, PetscInt M);
```

Yet another way (enter choice at runtime):

```
VecCreate(MPI_Comm comm, Vec* x);
VecSetSizes(Vec x, PetscInt m, PetscInt M);
VecSetFromOptions(Vec x);
```

and use `-vec_type seq` or `-vec_type mpi` at runtime.

PETSc Vectors: Exercise 1

- Create a parallel vector with
  - each local size equals to one plus the corresponding MPI rank,
  - all the vector values set to half the MPI size,
  and print the resulting vector on a various number of cores.
  Result on 3 cores:

```
Vector Object: 3 MPI processes
  type: mpi
Process [0]
1.5
Process [1]
1.5
1.5
Process [2]
1.5
1.5
1.5
```

- Duplicate the resulting vector to create a second vector, and copy the same values into it. Compute the dot product of the two vectors and check that the result equals the square of the 2-norm.

PETSc Vectors: Exercise 2

i. Create a parallel vector of global size 600,000,000 and let PETSc decide the parallel distribution.
ii. Get the range of indices owned by each MPI process and use three different ways to set each vector value equal to its global index. (Build the vec2a.c, vec2b.c and vec2c.c files.)
iii. Compare the performance of those three variants (using the Linux time command or the PetscTime function from PETSc).

Note: for this exercice, use the batch scheduler (LoadLeveler submission script provided).

PETSc Vectors: Exercise 2 - LoadLeveler Submission Script

```
# @ job_name = petscVec
# @ output = $(job_name).$(jobid)
# @ error = $(job_name).$(jobid)
# @ job_type = mpich
# @ total_tasks = 4
# @ environment = NB_TASKS=$(total_tasks)
# @ wall_clock_limit = 0:10:00
# @ queue

module load intel/2019.2

echo "-------------- Run of Vec2a ----------------"
time mpirun -np $NB_TASKS ./vec2a.exe
echo "-------------- Run of Vec2b ----------------"
time mpirun -np $NB_TASKS ./vec2b.exe
echo "-------------- Run of Vec2c ----------------"
time mpirun -np $NB_TASKS ./vec2c.exe
```

LoadLeveler Basic Commands

Submit a job:

```
$ llsubmit jobLL.sh
```

Follow a job:

```
$ llq -u $USER
Id                   Owner    Submitted   ST PRI Class Running On
------------------ ------   ----------- -- --- ----- ----------
ada337-ib.225870.0 userName 6/11 14:53  R  100 c8t1  ada174-ib
```

Cancel a job:

```
$ llcancel Id
```


Outline

PETSc Matrices: Types & Create

A matrix in PETSc is an object of type Mat.

Various types: Sequential or Distributed, Sparse or Dense, ...

```
MatCreate(MPI_Comm comm, Mat* A)
```

where comm = PETSC_COMM_SELF for sequential matrices
or any MPI communicator for distributed matrices (often PETSC_COMM_WORLD)

```
MatSetType(Mat A, MatType type)
```

where types = MATAIJ, MATDENSE, MATBAIJ, MATSBAIJ, ...

Note: MATAIJ = MATSEQAIJ if comm = PETSC_COMM_SELF
= MATMPIAIJ otherwise

Similarly with MATSEQDENSE and MATMPIDENSE

Also: MatSetFromOptions(Mat A) and at runtime

```
-mat_type seqaij/mpiaij/...
```


PETSc Matrices: Types & Create (continued)

```
MatSetSizes(Mat A, PetscInt m, PetscInt n,
                   PetscInt M, PetscInt N)
```

- m: local number of rows (or PETSC_DECIDE)
- n: local number of columns (or PETSC_DECIDE)
- M: global number of rows (or PETSC_DETERMINE)
- N: global number of columns (or PETSC_DETERMINE)

Before actually using the matrix, it should be set up:

```
MatSetUp(Mat A)
```

PETSc Matrices: Destroy/Duplicate/Copy

```
MatDuplicate(Mat A, MatDuplicateOption op,  Mat* B)
```

B created with same type as *A*.
Non-zero pattern duplicated and numerical values

- initialized to 0 if `op = MAT_DO_NOT_COPY_VALUES`
- copied if `op = MAT_COPY_VALUES`

```
MatCopy(Mat A, Mat B, MatStructure str)
```

$B \leftarrow A$ (*B* pre-existing)
`str = DIFFERENT_NONZERO_PATTERN` or `SAME_NONZERO_PATTERN` (optimization)

```
MatDestroy(Mat* A)
```


PETSc Matrices: GetSize/GetLocalSize/GetOwnershipRange

```
MatGetSize(Mat A, PetscInt* M, PetscInt* N)
```

where

- `M`: global number of rows
- `N`: global number of columns

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

where

- `m`: local number of rows
- `n`: local number of columns

```
MatGetOwnershipRange(Mat A, PetscInt* istart, PetscInt* iend)
```

Get the range of indices (matrix lines) owned by each processor.
Warning: 0-based indices even in `Fortran`.

PETSc Matrices: Set Value(s)

```
MatSetValue(Mat A, PetscInt row, PetscInt col, PetscScalar value,
            INSERT_VALUES or ADD_VALUES)
```

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

This routine inserts a *mxn* block of values in the matrix.

- *m*: number of rows
- *idxm*: global indexes of rows (0-based in `C` and `Fortran`)
- *n*: number of columns
- *idxn*: global indexes of columns (0-based in `C` and `Fortran`)
- *values*: array containing values to be inserted.

The value to be put in row *idxm[i]* and column *idxn[j]* is located in *values[i*n+j]*.

Note: If negative indices are passed in *idxm[i]* or *idxn[j]*, these rows and columns are simply ignored.

PETSc Matrices: Assemble

After using `MatSetValue` or `MatSetValues`, one must assemble the matrix:

```
MatAssemblyBegin(Mat A, MAT_FLUSH_ASSEMBLY or MAT_FINAL_ASSEMBLY)
MatAssemblyEnd(Mat A, MAT_FLUSH_ASSEMBLY or MAT_FINAL_ASSEMBLY)
```

Remarks:

1. Use `MAT_FLUSH_ASSEMBLY` when switching between `INSERT_VALUES` and `ADD_VALUES` in `MatSetValues`.
2. Use `MAT_FINAL_ASSEMBLY` for the final assembly before using the matrix.
3. Allows overlap of communication and computation.

PETSc Matrices: Memory Preallocation

Matrix preallocation is necessary for good matrix assembly performance:
use `MatMPIAIJSetPreallocation` (or `MatSeqAIJSetPreallocation`).
before assembling the matrix.

For parallel matrices (`MATMPIAIJ`), one has to provide the number of nonzeros per row in *diagonal* and *off-diagonal* submatrices on each processor, defined as follows.

PETSc Matrices: Memory Preallocation (example)

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

Diagonal submatrix on the first process:

$$\begin{pmatrix} 1 & 2 & 0 \\ 0 & 5 & 6 \\ 9 & 0 & 10 \end{pmatrix}$$

Off-diagonal submatrix on the first process:

$$\begin{pmatrix} 0 & 3 & 0 & 0 & 4 \\ 7 & 0 & 0 & 8 & 0 \\ 11 & 0 & 0 & 12 & 0 \end{pmatrix}$$

PETSc Matrices: Memory Preallocation

```
MatMPIAIJSetPreallocation(Mat A,
                          PetscInt d_nz, const PetscInt d_nnz[],
                          PetscInt o_nz, const PetscInt o_nnz[])
```

where

- `d_nz`: number of nonzeros per row in *diagonal* portion of local submatrix (same value is used for all local rows).
- `d_nnz`: array containing the number of nonzeros in the various rows of the *diagonal* portion of the local submatrix (possibly different for each row) or `NULL` (`PETSC_NULL_INTEGER` in Fortran) if `d_nz` is used to specify the nonzero structure.
- `o_nz` and `o_nnz`: same for *off-diagonal* portions of local submatrix.

N.B.: If the `*_nnz` parameter is given then the `*_nz` parameter is ignored

PETSc Matrices: Memory Preallocation (example)

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

Processor 0: $d\_nz = 2$ (or $d\_nnz = \{2, 2, 2\}$) and $o\_nz = 2$ (or $o\_nnz = \{2, 2, 2\}$).
Processor 1: $d\_nz = 3$ (or $d\_nnz = \{3, 3, 2\}$) and $o\_nz = 2$ (or $o\_nnz = \{2, 1, 1\}$).
Processor 2: $d\_nz = 1$ (or $d\_nnz = \{1, 1\}$) and $o\_nz = 4$ (or $o\_nnz = \{4, 4\}$).

PETSc Matrices: Memory Preallocation

Remarks:

1. matrix memory preallocation is critical for achieving good performance during matrix assembling, as this reduces the number of allocations and copies required
2. using the option *-info* during execution will print information about the success of preallocation during matrix assembly
3. when preallocation is used, calling `MatSetUp` is optionnal.

PETSc Matrices: Get value(s)

Local portions of a matrix can be examined (but not altered) with

- `MatGetValues`: returns a local block
- `MatGetRow`/`MatRestoreRow`: obtain a row associated with the given processor

It is recommended to use high-level routines such as:

- `MatGetRowMax`/`MatGetRowMin`
- `MatGetRowSum`
- `MatGetDiagonal` (only for square matrices)
- ...

PETSc Matrices: View

```
MatView(Mat A, PetscViewer viewer)
```

where for `viewer` one uses in general `PETSC_VIEWER_STDOUT_WORLD`.

There are additional viewers like `PETSC_VIEWER_DRAW_WORLD` which draws the non-zero structure of the matrix in X-default window.

PETSc Matrices: Operations

Matrix-Vector product `y = A x`:

```
MatMult(Mat A, Vec x, Vec y)
```

By default if the user lets `PETSc` decide the number of components to be stored locally (by using `PETSC_DECIDE`), vectors and matrices of the same dimension are automatically compatible for parallel matrix-vector operations.

To create vectors compatible with a given matrix:

```
MatCreateVecs(Mat A, Vec* right, Vec* left)
```

creates the two vectors

- `right`: a vector that the matrix can be multiplied against
- `left`: a vector that can be used to store the result of the matrix-vector product

(such that `left = A right` makes sense).

PETSc Matrices: Operations

Other Matrix operations:

MatAXPY	$Y = Y + a * X$
MatMultAdd	$z = y + A * x$
MatMultTranspose	$y = A^T * x$
MatNorm	$r = \|A\|_{type}$
MatDiagonalSet	$A = D$ or $A = A + D$
MatDiagonalScale	$A = D_l * A * D_r$
MatScale	$A = a * A$
MatConvert	$B = A$
MatCopy	$B = A$
MatGetDiagonal	$x = diag(A)$
MatTranspose	$B = A^T$
MatZeroEntries	$A = 0$
MatShift	$Y = Y + a * I$

PETSc Matrices: Exercice 1

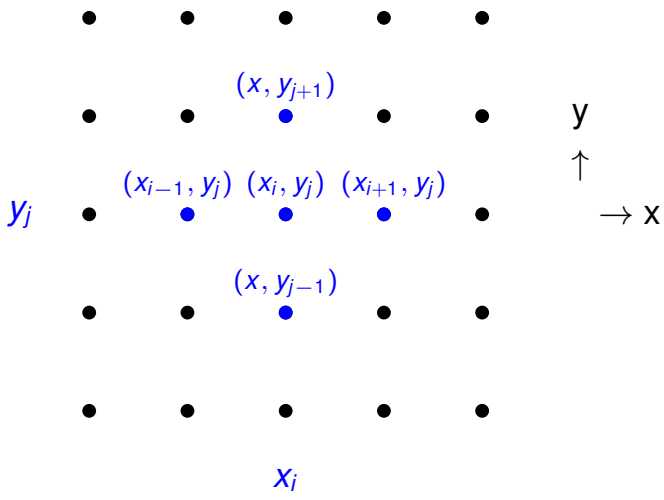
- Create the identity matrix
- Scale the matrix with a double value
- Create a vector u that the matrix can be multiplied against
- Initialize it with values $u[i] = (i + 1) * 10$
- Multiply the matrix and the vector
- Print the resulting vector and check that it is correct

PETSc Matrices: Exercice 2

Create the matrix representing the Laplace operator in 2-D

$$\Delta u = \frac{\partial^2 u}{\partial^2 x} + \frac{\partial^2 u}{\partial^2 y}$$

discretized using a 5-point finite difference scheme:



PETSc Matrices: Exercice 2

With $u_{i,j} = u(x_i, y_j)$ and h_x the mesh step size in x, one has

$$\begin{aligned}\frac{\partial^2 u}{\partial^2 x}(x_i, y_j) &\approx \frac{1}{h_x}\left(\frac{u_{i+1,j} - u_{i,j}}{h_x} - \frac{u_{i,j} - u_{i-1,j}}{h_x}\right) \\ &\approx \frac{1}{h_x^2}\left(u_{i+1,j} - 2u_{i,j} + u_{i-1,j}\right)\end{aligned}$$

Similarly in y

$$\frac{\partial^2 u}{\partial^2 y}(x_i, y_j) \approx \frac{1}{h_y^2}\left(u_{i,j+1} - 2u_{i,j} + u_{i,j-1}\right)$$

such that if $h = h_x = h_y$

$$\Delta u = \frac{1}{h^2}\left(u_{i,j+1} + u_{i+1,j} - 4u_{i,j} + u_{i-1,j} + u_{i,j-1}\right)$$

PETSc Matrices: Exercice 2 (continued)

We consider homogeneous Dirichlet boundary conditions and do not include the boundary points in our matrix. For a 4×4 grid numbered line by line, the resulting matrix looks like this:

$$A = \frac{-1}{h^2} \left(\begin{array}{cccc|cccc|cccc|c} 4 & -1 & & & -1 & & & & & & & & \\ -1 & 4 & -1 & & & -1 & & & & & & & \\ & -1 & 4 & -1 & & & -1 & & & & & & \\ & & -1 & 4 & & & & -1 & & & & & \\ \hline -1 & & & & 4 & -1 & & & -1 & & & & \\ & -1 & & & -1 & 4 & -1 & & & -1 & & & \\ & & -1 & & & -1 & 4 & -1 & & & -1 & & \\ & & & -1 & & & -1 & 4 & & & & -1 & \\ \hline & & & & \ddots & & & & \ddots & & & & \ddots \end{array} \right)$$

Use `MatView` with `PETSC_VIEWER_STDOUT_WORLD` to check your matrix.

Use it also with `PETSC_VIEWER_DRAW_WORLD` (and `-draw_pause <sec>`) to visualize the nonzero structure.

Outline

Linear Algebra prerequisites: direct vs. iterative methods

To solve $Ax = b$:

- Direct methods:
  1. Factorization: $A = LU$ [hard!]
  2. Solve: $L(Ux) = b \Rightarrow Ux = L^{-1}b \Rightarrow x = U^{-1}(L^{-1}b)$ [easy!]
- Iterative methods: start from x_0 and define

$$x_{n+1} = x_n + B^{-1}(b - A x_n)$$

  with B the preconditioner such that $B^{-1} \simeq A^{-1}$, and proceed until convergence (residual $< \epsilon$).

  Motivation: if $B^{-1} = A^{-1}$ we have

$$\begin{aligned} x_{n+1} &= x_n + A^{-1}(b - A x_n) \\ &= x_n + A^{-1} b - x_n \\ &= A^{-1} b \text{ exact solution} \end{aligned}$$

Linear Algebra prerequisites: choosing a preconditioner

- Identity: $B = I$ (no preconditioning)
- Jacobi (or diagonal): $B = diag(A)$
- (S)SOR (successive over relaxation): B = lower/upper triangular part of A
- ILU (incomplete LU factorization): $B = \tilde{L}\tilde{U}$
  ICC (incomplete Cholesky factorization) if symmetric ($\tilde{L} = \tilde{U}^T$)
- Block Jacobi (or block diagonal): $B = blockDiag(A)$; domain decomposition with each (local) block solved directly or iteratively
- ASM (Additive Schwarz Method): domain decomposition with possible overlap
- Multigrid or algebraic multigrid method
- ...

If the preconditioner is “better”, the convergence is faster.

Linear Algebra prerequisites: more iterative acceleration

We have seen that

$$\begin{aligned} x_{n+1} &= x_n + B^{-1}(b - A x_n) \\ &= f(x_n) \end{aligned}$$

This is known as *Richardson* (or *stationary*) iterations.
More generally, to accelerate convergence, one can do

$$x_{n+1} = f(x_n, x_{n-1}, \ldots)$$

Most famous in this view are the Krylov-type methods:

- CG - Conjugate Gradient (for Symmetric Positive Definite matrices)
- GMRES - Generalized Minimal Residual
- BiCGStab - Biconjugate Gradient Stabilized
- ...

Combine "good" preconditioner with Krylov-type acceleration scheme
for fast convergence

PETSc Solvers

PETSc specializes in Krylov-type iterative solvers and preconditioners,
and also offers interfaces for external direct solvers (`Mumps, PaStiX, SuperLU`)
and external preconditioners (`Hypre, Trilinos/ML`,...).

PETSc types:

- `KSP` ≡ Krylov solver
- `PC` ≡ Preconditioner

List of available KSP and PC types:

https://www.mcs.anl.gov/petsc/petsc-current/docs/manualpages/KSP/KSPType.html
https://www.mcs.anl.gov/petsc/petsc-current/docs/manualpages/PC/PCType.html

Summary of available sparse linear solvers (with matrix types):

https://www.mcs.anl.gov/petsc/documentation/linearsolvertable.html

Note: no specific type for direct solver, in fact handled as “special case” of KSP (!) - see later.

PETSc Solvers: Create & Set Matrix

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

```
KSPSetOperators(KSP ksp, Mat A, Mat precondBase);
```

where

- `A` = system matrix
- `precondBase` = base matrix to derive the preconditioner (typically *A* itself)

PETSc Solvers: Set Solution Method

```
KSPSetType(KSP ksp, KSPType kspType);
KSPSetTolerances(KSP ksp, PetscReal rtol, PetscReal atol,
                          PetscReal dtol, PetscInt maxits);
```

where

- `kspType = KSPCG, KSPGMRES, KSPBCGS, ...` (full list on the `KSPType` documentation page above)
- `rtol, atol, dtol` = relative, absolute, divergence tolerance (resp. default values: 1e-5, 1e-50, 1e5)
- `maxits` = maximum number of iterations (default 10,000)

or for runtime specification:

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

and program launched using: `-ksp_type <method> -ksp_rtol <rtol> ...`

where `<method> = cg, gmres, bcgs, ...`

PETSc Solvers: Set Preconditioner

```
KSPGetPC(KSP ksp, PC* pc);
PCSetType(PC pc, PCType pcType);
PCSetUp(PC pc);
```

where `pcType = PCNONE, PCJACOBI, PCSOR, PCILU, PCICC, ...`
(full list on the `PCType` documentation page above)

or for runtime specification (before `PCSetUp`) call

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

and program launched using: `-pc_type <method>`

where `<method> = none, jacobi, sor, ilu, icc, ...`

NB: `PCSetFromOptions` included in `KSPSetFromOptions`

PETSc Solvers: Solve & After

Before solving, one must call:

```
KSPSetUp(KSP ksp);
```

To solve `A x = b`:

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

- `x` overwritten with answer.
- initial guess `x=0` unless `KSPSetInitialGuessNonzero` before solve.

To observe convergence at runtime: `-ksp_monitor`

PETSc Solvers: Solve & After

After solve:

```
KSPGetConvergedReason(KSP ksp, KSPConvergedReason* reason)
```

where

- reason = 2 ≡ KSP_CONVERGED_RTOL
- reason = -3 ≡ KSP_DIVERGED_ITS
- ...

```
KSPGetIterationNumber(KSP ksp, PetscInt* its)
```

```
KSPGetResidualNorm(KSP ksp, PetscReal* rnorm)
```

Note: norm of the *preconditioned* residual $B^{-1}(b - Ax)$ by default (if left preconditioning).

```
KSPDestroy(KSP* ksp)
```


PETSc Solvers: Viewing

```
KSPView(KSP ksp, PETSC_VIEWER_STDOUT_WORLD)
```

Example output with `kspType = KSPBCGS` and `pcType = PCSOR`:

```
KSP Object: 8 MPI processes
  type: bcgs
  maximum iterations=1000000, initial guess is zero
  tolerances:  relative=1e-08, absolute=1e-50, divergence=10000
  left preconditioning
  using DEFAULT norm type for convergence test
PC Object: 8 MPI processes
  type: sor
    SOR: type = local_symmetric, iterations = 1, local iterations = 1, omega = 1
  linear system matrix = precond matrix:
  Matrix Object:   8 MPI processes
    type: mpibaij
    rows=57600, cols=57600, bs=4
    total: nonzeros=1142400, allocated nonzeros=1612800
    total number of mallocs used during MatSetValues calls =0
        block size is 4
```

or at runtime: `-ksp_view`

PETSc Solvers: Direct Methods

In PETSc, direct methods are special cases of Krylov methods (KSP), with only the PCLU preconditioner applied:

```
KSPSetType(ksp, KSPPREONLY);
KSPGetPC(ksp, &pc);
PCSetType(pc, PCLU);
```

The PETSc built-in PCLU (and PCILU) works only in sequential.

For parallel direct methods, use external solvers:

```
PCFactorSetMatSolverType(pc, MATSOLVERPASTIX);
```

```
PCFactorSetMatSolverType(pc, MATSOLVERMUMPS);
```

```
PCFactorSetMatSolverType(pc, MATSOLVERSUPERLU_DIST);
```

or at runtime: `-ksp_type preonly -pc_type lu -pc_factor_mat_solver_type pastix/mumps/superlu_dist`

Note: mumps used by default if PCLU invoked in parallel.

PETSc Solvers: Domain Decomposition with PCBJACOBI

Example with PCBJACOBI (here with iterative local solves):

```
KSPGetPC(ksp, &pc);
PCSetType(pc, PCBJACOBI);
PCSetUp(pc);
PCBJacobiGetSubKSP(pc, &n_local, &first_local, &subKSP);
for (i = 0; i < n_local; i++) {
  KSPSetType(subKSP(i), KSPGMRES);
  KSPGetPC(subKSP(i), &subPC);
  PCSetType(subPC, PCSOR);
}
```

where

- subKSP = array containing the local KSP objects on each subdomain
- n_local = number of blocks on this processor
- first_local = global number of the first block on this processor

At runtime, use -sub_ksp_type and -sub_pc_type.

PETSc Solvers: Domain Decomposition with PCASM

Example with PCASM (here with direct local solves):

```
KSPGetPC(ksp, &pc);
PCSetType(pc, PCASM);
PCASMSetOverlap(pc, overlap);
PCSetUp(pc);
PCASMGetSubKSP(pc, &n_local, &first_local, &subKSP);
for (i = 0; i < n_local; i++) {
  KSPSetType(subKSP(i), KSPPREONLY);
  KSPGetPC(subKSP(i), &subPC);
  PCSetType(subPC, PCLU);
}
```

where

- subKSP = array containing the local KSP objects
- n_local = number of blocks on this processor
- first_local = global number of the first block on this processor

At runtime, use -sub_ksp_type and -sub_pc_type.

PETSc Solvers: PETSc default

Default PETSc solver (schematically):

```
KSPGetPC(ksp, &pc);
PCSetType(pc, PCBJACOBI);
PCASMSetOverlap(pc, overlap);
PCSetUp(pc);
PCASMGetSubKSP(pc, &n_local, &first_local, &subKSP);
for (i = 0; i < n_local; i++) {
  KSPSetType(subKSP(i), KSPPREONLY);
  KSPGetPC(subKSP(i), &subPC);
  PCSetType(subPC, PCILU);
}
```

where

- subKSP = array containing the local KSP objects
- n_local = number of blocks on this processor
- first_local = global number of the first block on this processor

At runtime, use -sub_ksp_type and -sub_pc_type.

PETSc Solvers: Exercice

Using the matrix you built previously, solve the Poisson problem

$$-\Delta x = b$$

with a random right-hand-side vector.

Procedure: build a random x_{exact} vector and build $b = Ax_{exact}$.

Then solve using the default `KSP` and `PC` settings.

Compare the solution with x_{exact} by computing the 2-norm and the infinity-norm of $x - x_{exact}$.

PETSc Solvers: Exercice (continued)

Insert timing routines and compare different solution methods by changing the method at runtime:

- replace the default `GMRES` iterative solution method with `CG`, `MinRes` or `BiCGStab`.
- replace the default preconditioner with `Jacobi`, `ASM` or `HYPRE`
- try the `MUMPS` direct solver.

Recommendation: Use a 1000 $\times$ 1000 grid size on 4 cores.

For one of your calculations with the `CG` method, compute the norm of the residual $\|Ax - b\|$ yourself and compare it with the one given by `PETSc`. In this view, have `PETSc` compute an unpreconditioned residual using the runtime option `-ksp_norm_type unpreconditioned`.

Outline

Profiling

Basic profiling options:

- -log_view: to print summary of flop and timing information
- -info: print details about algorithms, data structure,... (slows down the code - debug only!)
- -log_trace: to print traces of all PETSc calls (to see where a program is hanging without running in a debugger)

-log_view output (1/3)

```
************************************************************************************************************************
***             WIDEN YOUR WINDOW TO 120 CHARACTERS.  Use 'enscript -r -fCourier9' to print this document         ***
************************************************************************************************************************

---------------------------------------------- PETSc Performance Summary: ----------------------------------------------

./solver.exe on a  named ada239 with 4 processors, by ssos455 Thu Jun 20 10:12:53 2019
Using Petsc Release Version 3.11.2, May, 18, 2019

                         Max       Max/Min     Avg       Total
Time (sec):           4.814e+01     1.000   4.814e+01
Objects:              6.200e+01     1.000   6.200e+01
Flop:                 5.472e+10     1.000   5.472e+10  2.189e+11
Flop/sec:             1.137e+09     1.000   1.137e+09  4.546e+09
MPI Messages:         5.281e+03     2.000   3.962e+03  1.585e+04
MPI Message Lengths:  4.222e+07     2.000   7.992e+03  1.266e+08
MPI Reductions:       5.224e+03     1.000

Flop counting convention: 1 flop = 1 real number operation of type (multiply/divide/add/subtract)
                            e.g., VecAXPY() for real vectors of length N --> 2N flop
                            and VecAXPY() for complex vectors of length N --> 8N flop

Summary of Stages:   ----- Time ------  ----- Flop ------  --- Messages ---  -- Message Lengths --  -- Reductions --
                        Avg     %Total     Avg     %Total    Count   %Total     Avg         %Total    Count   %Total
 0:      Main Stage: 4.8143e+01 100.0%  2.1888e+11 100.0%  1.585e+04 100.0%  7.992e+03      100.0%  5.215e+03  99.8%

------------------------------------------------------------------------------------------------------------------------
```


-log_view output (2/3)

```
See the 'Profiling' chapter of the users' manual for details on interpreting output.
Phase summary info:
   Count: number of times phase was executed
   Time and Flop: Max - maximum over all processors
                  Ratio - ratio of maximum to minimum over all processors
   Mess: number of messages sent
   AvgLen: average message length (bytes)
   Reduct: number of global reductions
   Global: entire computation
   Stage: stages of a computation. Set stages with PetscLogStagePush() and PetscLogStagePop().
      %T - percent time in this phase         %F - percent flop in this phase
      %M - percent messages in this phase     %L - percent message lengths in this phase
      %R - percent reductions in this phase
   Total Mflop/s: 10e-6 * (sum of flop over all processors)/(max time over all processors)
------------------------------------------------------------------------------------------------------------------------
Event            Count      Time (sec)     Flop                             --- Global ---  --- Stage ----  Total
                   Max Ratio  Max     Ratio   Max  Ratio  Mess   AvgLen  Reduct  %T %F %M %L %R  %T %F %M %L %R Mflop/s
------------------------------------------------------------------------------------------------------------------------

--- Event Stage 0: Main Stage

BuildTwoSidedF     1 1.0 3.1669e-03 5.9 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0      0
MatMult         2638 1.0 7.6309e+00 1.0 5.93e+09 1.0 1.6e+04 8.0e+03 0.0e+00 16 11100100  0  16 11100100  0  3109
MatSolve        2637 1.0 1.0156e+01 1.0 5.92e+09 1.0 0.0e+00 0.0e+00 0.0e+00 21 11  0  0  0  21 11  0  0  0  2332
MatLUFactorNum     1 1.0 1.4307e-02 1.0 2.74e+06 1.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0   766
MatILUFactorSym    1 1.0 1.1605e-02 1.0 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0      0
MatAssemblyBegin   1 1.0 3.2959e-03 5.8 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0      0
MatAssemblyEnd     1 1.0 9.9793e-02 1.0 0.00e+00 0.0 1.2e+01 2.0e+03 8.0e+00  0  0  0  0  0   0  0  0  0  0      0
MatGetRowIJ        1 1.0 3.6001e-0516.8 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0      0
MatGetOrdering     1 1.0 1.4179e-03 1.0 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0      0
MatView            2 2.0 3.8099e-04 1.5 0.00e+00 0.0 0.0e+00 0.0e+00 1.0e+00  0  0  0  0  0   0  0  0  0  0      0
VecMDot         2551 1.0 1.2966e+01 1.0 1.98e+10 1.0 0.0e+00 0.0e+00 2.6e+03 27 36  0  0 49  27 36  0  0 49   6097
VecNorm         2643 1.0 7.7790e-01 1.6 1.32e+09 1.0 0.0e+00 0.0e+00 2.6e+03  1  2  0  0 51   1  2  0  0 51   6790
VecScale        2637 1.0 3.1526e-01 1.0 6.59e+08 1.0 0.0e+00 0.0e+00 0.0e+00  1  1  0  0  0   1  1  0  0  0   8365
VecCopy           87 1.0 4.6980e-02 1.0 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0      0
VecSet          2725 1.0 7.0045e-01 1.0 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  1  0  0  0  0   1  0  0  0  0      0
VecAXPY          173 1.0 7.5669e-02 1.1 8.65e+07 1.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0   4573
...
```

-log_view output (3/3)

```
Memory usage is given in bytes:

Object Type          Creations   Destructions    Memory  Descendants' Mem.
Reports information only for process 0.

--- Event Stage 0: Main Stage

              Matrix     4              4     55986668     0.
              Vector    44             44     80080832     0.
           Index Set     5              5      3007960     0.
         Vec Scatter     1              1         1392     0.
         PetscRandom     1              1          646     0.
       Krylov Solver     2              2        20040     0.
     Preconditioner      2              2         1912     0.
              Viewer     3              2         1680     0.
```


More on -log_view: defining stages

Up to 10 profiling stages can be defined in a code:

```
PetscLogStage stage1, stage2;
...
PetscLogStageRegister("Name of stage 1", &stage1);
PetscLogStageRegister("Name of stage 2", &stage2);
...
PetscLogStagePush(stage1);
...
PetscLogStagePop();
...
PetscLogStagePush(stage2);
...
PetscLogStagePop();
```

-log_view output with stages (1/3)

```
************************************************************************************************************************
***             WIDEN YOUR WINDOW TO 120 CHARACTERS.  Use 'enscript -r -fCourier9' to print this document         ***
************************************************************************************************************************

---------------------------------------------- PETSc Performance Summary: ----------------------------------------------

./solver.exe on a  named ada042 with 4 processors, by ssos455 Thu Jun 20 09:56:08 2019
Using Petsc Release Version 3.11.2, May, 18, 2019

                         Max       Max/Min     Avg       Total
Time (sec):           4.793e+01     1.000   4.793e+01
Objects:              6.200e+01     1.000   6.200e+01
Flop:                 5.472e+10     1.000   5.472e+10  2.189e+11
Flop/sec:             1.142e+09     1.000   1.142e+09  4.567e+09
MPI Messages:         5.281e+03     2.000   3.962e+03  1.585e+04
MPI Message Lengths:  4.222e+07     2.000   7.992e+03  1.266e+08
MPI Reductions:       5.224e+03     1.000

Flop counting convention: 1 flop = 1 real number operation of type (multiply/divide/add/subtract)
                            e.g., VecAXPY() for real vectors of length N --> 2N flop
                            and VecAXPY() for complex vectors of length N --> 8N flop

Summary of Stages:   ----- Time ------  ----- Flop ------  --- Messages ---  -- Message Lengths --  -- Reductions --
                        Avg     %Total     Avg     %Total    Count   %Total     Avg         %Total    Count   %Total
 0:      Main Stage: 4.8826e-01   1.0%  2.9984e+07   0.0%  1.800e+01   0.1%  5.335e+03        0.1%  1.900e+01   0.4%
 1: Fill & assemble: 9.8242e-02   0.2%  0.0000e+00   0.0%  1.200e+01   0.1%  2.002e+03        0.0%  8.000e+00   0.2%
 2:           Solve: 4.7344e+01  98.8%  2.1885e+11 100.0%  1.582e+04  99.8%  8.000e+03       99.9%  5.188e+03  99.3%

------------------------------------------------------------------------------------------------------------------------
```


-log_view output with stages (2/3)

```
------------------------------------------------------------------------------------------------------------------------
Event                Count      Time (sec)     Flop                             --- Global ---  --- Stage ----  Total
                   Max Ratio  Max     Ratio   Max  Ratio  Mess   AvgLen  Reduct  %T %F %M %L %R  %T %F %M %L %R Mflop/s
------------------------------------------------------------------------------------------------------------------------

--- Event Stage 0: Main Stage

MatMult                2 1.0 6.4082e-03 1.0 4.50e+06 1.0 1.2e+01 8.0e+03 0.0e+00  0  0  0  0  0   1 60 67100  0  2806
MatView                2 2.0 3.7599e-04 1.5 0.00e+00 0.0 0.0e+00 0.0e+00 1.0e+00  0  0  0  0  0   0  0  0  0  5     0
VecNorm                6 1.0 2.0132e-03 1.1 2.00e+06 1.0 0.0e+00 0.0e+00 6.0e+00  0  0  0  0  0   0 27  0  0 32  3974
VecCopy                1 1.0 2.7895e-04 1.0 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0     0
VecAXPY                2 1.0 9.1696e-04 1.0 1.00e+06 1.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0 13  0  0  0  4362
VecScatterBegin        2 1.0 1.2803e-04 1.1 0.00e+00 0.0 1.2e+01 8.0e+03 0.0e+00  0  0  0  0  0   0  0 67100  0     0
VecScatterEnd          2 1.0 2.1887e-04 1.7 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0     0
VecSetRandom           1 1.0 7.5328e-03 1.0 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   2  0  0  0  0     0
KSPSetUp               1 1.0 6.5360e-03 1.1 0.00e+00 0.0 0.0e+00 0.0e+00 2.0e+00  0  0  0  0  0   1  0  0  0 11     0
PCSetUp                1 1.0 3.2440e-02 1.0 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   7  0  0  0  0     0

--- Event Stage 1: Fill & assemble

BuildTwoSidedF         1 1.0 9.5487e-04 9.4 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   1  0  0  0  0     0
MatAssemblyBegin       1 1.0 1.1778e-03 8.5 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   1  0  0  0  0     0
MatAssemblyEnd         1 1.0 6.2296e-02 1.0 0.00e+00 0.0 1.2e+01 2.0e+03 8.0e+00  0  0  0  0  0  63  0100100100     0
VecSet                 1 1.0 5.7220e-05 1.4 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0     0

--- Event Stage 2: Solve

MatMult             2636 1.0 7.6238e+00 1.0 5.93e+09 1.0 1.6e+04 8.0e+03 0.0e+00 16 11100100  0  16 11100100  0  3109
MatSolve            2637 1.0 1.0184e+01 1.0 5.92e+09 1.0 0.0e+00 0.0e+00 0.0e+00 21 11  0  0  0  21 11  0  0  0  2325
MatLUFactorNum         1 1.0 1.4273e-02 1.0 2.74e+06 1.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0   768
MatILUFactorSym        1 1.0 1.1533e-02 1.1 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0     0
MatGetRowIJ            1 1.0 6.2943e-0533.0 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0     0
MatGetOrdering         1 1.0 1.4310e-03 1.1 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0     0
VecMDot             2551 1.0 1.2894e+01 1.0 1.98e+10 1.0 0.0e+00 0.0e+00 2.6e+03 27 36  0  0 49  27 36  0  0 49  6131
VecNorm             2637 1.0 7.8429e-01 1.7 1.32e+09 1.0 0.0e+00 0.0e+00 2.6e+03  1  2  0  0 50   1  2  0  0 51  6725
VecScale            2637 1.0 3.1082e-01 1.0 6.59e+08 1.0 0.0e+00 0.0e+00 0.0e+00  1  1  0  0  0   1  1  0  0  0  8484
VecCopy               86 1.0 4.7260e-02 1.1 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0     0
VecSet              2724 1.0 7.0102e-01 1.0 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  1  0  0  0  0   1  0  0  0  0     0
...
```

-log_view output with stages (3/3)

```
Object Type          Creations   Destructions      Memory  Descendants' Mem.
Reports information only for process 0.

--- Event Stage 0: Main Stage

              Matrix     3              4     55986668     0.
              Vector    13             43     80079168     0.
          Index Set      0              3      3002376     0.
        Vec Scatter      0              1         1392     0.
        PetscRandom      1              1          646     0.
      Krylov Solver      2              2        20040     0.
     Preconditioner      2              2         1912     0.
              Viewer     3              2         1680     0.

--- Event Stage 1: Fill & assemble

              Vector     2              1         1664     0.
          Index Set      2              2         5584     0.
        Vec Scatter      1              0            0     0.

--- Event Stage 2: Solve

              Matrix     1              0            0     0.
              Vector    29              0            0     0.
          Index Set      3              0            0     0.
```


Profiling application codes

PETSc automatically logs object creation, times and floating-point counts for the built-in library routines.
To have your application code monitored as well, use:

```
PetscLogEventRegister(...)
PetscLogEventBegin(...)
...
PetscLogFlops(...)
PetscLogEventEnd(...)
```

Note that `PetscLogFlops` must be defined by the user.

Similarly external solvers (like `MUMPS`, ...) are not logged in `PETSc` profiling.

Other features

- Debugger option: `-start_in_debugger` or `-on_error_attach debugger` (uses `gdb` by default)

- To create your own option(s):

```
PetscInt size;
PetscOptionsGetInt(PETSC_NULL, PETSC_NULL, "-size",
                   &size, PETSC_NULL);
```

  See also `PetscOptionsGetBool`, `PetscOptionsGetReal`, ...

- Matrix-Free methods: one can define a matrix only through its effect in various operations, for instance matrix-vector products (only required operation for Krylov methods), avoiding full matrix assembly.
  See `MatCreateShell` and `MatShellSetOperation`.
  `KSP` supports matrix-free methods, but the matrix-free variant is allowed only in combination with no preconditioning (`PCNONE`), a user-provided preconditioner matrix, or a user-provided preconditioner shell (`PCSHELL`).

Other features

- Data Management (`DM`) tools for communication between algebraic structures (like `Vec` and `Mat`) and mesh data structures:
  - `DMDA` (Distributed Arrays): for cartesian structured meshes
  - `DMPlex`: for unstructured meshes
  - ...

  Defines local portions of a mesh, manages ghost points,...

- Non-linear solvers: see the `SNES` (Scalable Nonlinear Equation Solvers) object. Built on top of `KSP` solvers and data management tools.

- Time-dependent problems: see the `TS` (Time Stepping) object for ODEs.

Exercises

In your solver code:

- add 2 stages, for instance one for the matrix fill-in and assembly, and another one for the solve.
- add an option to be allowed to enter the grid dimension at runtime.

Outline

PETSc DMDA: Introduction

"DM objects are used to manage communication between the algebraic structures in PETSc (like Vec and Mat) and mesh data structures in PDE-based (or other) simulations."

Different types of DM:

- DMDA (Distributed Arrays): for Cartesian structured meshes
- DMPlex: for unstructured meshes
- DMNetwork: for graphs
- ...

Here we restrict ourselves to the DMDA objets.

They are created with DMDACreate1d, DMDACreate2d or DMDACreate3d.

They are destroyed with DMDestroy.

PETSc DMDA: Creation in 2-D

```
DMDACreate2d(MPI_Comm comm, DMBoundaryType bx, DMBoundaryType by,
             DMDAStencilType stencil_type,
             PetscInt M, PetscInt N, PetscInt m, PetscInt n,
             PetscInt dof, PetscInt s,
             const PetscInt lx[], const PetscInt ly[], DM* da)
```

where

- DMBoundaryType describes the choice for fill of ghost cells on physical domain boundaries (not on interfaces between processes!): DM_BOUNDARY_NONE, DM_BOUNDARY_PERIODIC, ...
- DMDAStencilType: DMDA_STENCIL_STAR, DMDA_STENCIL_BOX
- M, N: global dimension in each direction of the array
- m, n: number of processors in each dimension (or PETSC_DECIDE)
- dof: number of degrees of freedom per node
- s: stencil width
- lx, ly: arrays (resp. of length m, n) containing the number of nodes in each cell along the x and y coordinates, or NULL.

PETSc DMDA: Stencil Type and Width in 2-D

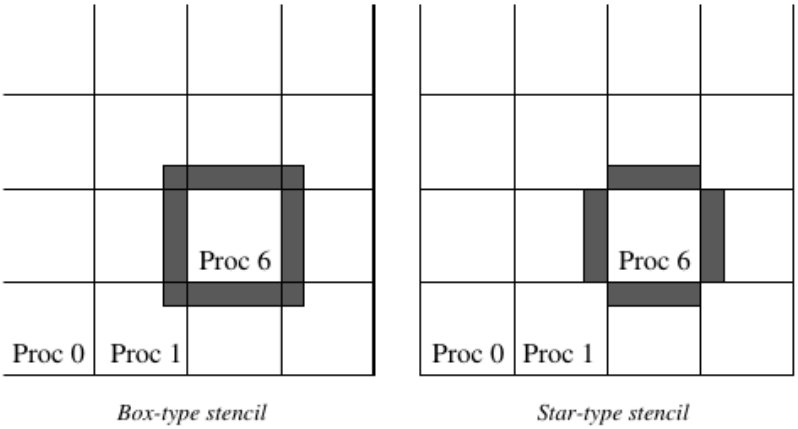


Figure 7: Ghost Points for Two Stencil Types on the Seventh Process

Standard 5-point stencil ⇒ take `DMDAStencilType` = `DMDA_STENCIL_STAR`
and stencil width `s`=1

Standard 9-point stencil ⇒ take `DMDAStencilType` = `DMDA_STENCIL_BOX`
and stencil width `s`=1

PETSc DMDA: SetFromOptions/SetUp

After creation:

```
DMSetFromOptions(DM dm)
DMSetUp(DM dm)
```

`DMSetUp` is necessary.

`DMSetFromOptions` only for runtime specifications. Then command line options available:

```
-da_grid_x <M>   - number of grid points in x direction
-da_grid_y <N>   - number of grid points in y direction
-da_processors_x <m>   - number of processors in x direction
-da_processors_y <n>   - number of processors in y direction
...
```

Note: mesh refinement also possible.

PETSc DMDA: View

To view a DMDA:

```
DMView(DM dm, PETSC_VIEWER_STDOUT_WORLD)
```


PETSc DMDA: Example on 5×4 mesh on 4 processes

	Proc[2]		*Proc*[3]	
13	14	15	18	19
10	11	12	16	17
3	4	5	8	9
0	1	2	6	7
	Proc[0]		*Proc*[1]	

DMView yields:

```
DM Object: 4 MPI processes
  type: da
Processor [0] M 5 N 4 m 2 n 2 w 1 s 1
X range of indices: 0 3, Y range of indices: 0 2
Processor [1] M 5 N 4 m 2 n 2 w 1 s 1
X range of indices: 3 5, Y range of indices: 0 2
Processor [2] M 5 N 4 m 2 n 2 w 1 s 1
X range of indices: 0 3, Y range of indices: 2 4
Processor [3] M 5 N 4 m 2 n 2 w 1 s 1
X range of indices: 3 5, Y range of indices: 2 4
```

PETSc DMDA: Get local information

```
DMDAGetLocalInfo(DM da, DMDALocalInfo* info)
```

where DMDALocalInfo is a C structure:

```
typedef struct {
 PetscInt dim, dof, sw;
 /* global number of grid points in each direction */
 PetscInt mx, my, mz;
 /* starting point of this processor, excluding ghosts */
 PetscInt xs, ys, zs;
 /* number of grid points on this processor, excluding ghosts */
 PetscInt xm, ym, zm;
 /* starting point of this processor including ghosts */
 PetscInt gxs, gys, gzs;
 /* number of grid points on this processor including ghosts */
 PetscInt gxm, gym, gzm;
 /* type of ghost nodes at boundary */
 DMBoundaryType bx, by, bz;
 DMDAStencilType st;
 DM da;
} DMDALocalInfo;
```


PETSc DMDA: Example on 5×4 mesh on 4 processes

Proc[2]			*Proc*[3]	
13	14	15	18	19
10	11	12	16	17
3	4	5	8	9
0	1	2	6	7
Proc[0]			*Proc*[1]	

If DM_BOUNDARY_NONE everywhere, we have:

```
Proc[0]:  info.mx = 5 info.my = 4
Proc[0]:  info.xs = 0 info.ys = 0 info.xm = 3 info.ym = 2
Proc[0]:  info.gxs = 0 info.gys = 0 info.gxm = 4 info.gym = 3

Proc[1]:  info.mx = 5 info.my = 4
Proc[1]:  info.xs = 3 info.ys = 0 info.xm = 2 info.ym = 2
Proc[1]:  info.gxs = 2 info.gys = 0 info.gxm = 3 info.gym = 3
```

PETSc DMDA: Get local information

Note for `Fortran`: `DMDALocalInfo` is hard to use in `Fortran`.

Better to use `DMDAGetCorners` and `DMDAGetInfo` to get the same information.

PETSc DMDA: Creating Vectors & Matrices

To create vectors with the appropriate local/global sizes from a `dm`:

```
DMCreateLocalVector(DM dm, Vec* vec)
DMCreateGlobalVector(DM dm, Vec* vec)
```

To create a matrix from a `dm`:

```
DMCreateMatrix(DM dm, Mat* mat)
```

Then:

- The number of nonzeros in the sparse matrix is automatically preallocated!
- The nonzero structure is automatically set with zero entries put in.
- `MatSetValuesStencil` is recommended to fill-in values (see next slides).

PETSc DMDA: MatStencil

The MatStencil structure stores logical coordinates i,j,k of a point in a grid, i.e., of a single row or column of the associated matrix:

```
typedef struct {
  PetscInt k, j, i, c;
} MatStencil;
```

(c = degrees of freedom at each grid point - ignored if 1 dof per grid point)

PETSc DMDA: MatSetValuesStencil

```
MatSetValuesStencil(Mat mat,
                    PetscInt m, const MatStencil idxm[],
                    PetscInt n, const MatStencil idxn[],
                    const PetscScalar v[], InsertMode addv)
```

where

- m/n = number of rows/columns being entered
- idxm/idxn = grid coordinates for matrix rows/columns being entered
- v = the array of values
- addv = ADD_VALUES or INSERT_VALUES

For an example, see the following tutorial examples:

http://www.mcs.anl.gov/petsc/petsc-current/src/ksp/ksp/examples/tutorials/ex29.c.html

http://www.mcs.anl.gov/petsc/petsc-current/src/ksp/ksp/examples/tutorials/ex22f.F90.html

PETSc DMDA: Exercise

Fill-in the missing commands (look for TBC = “To Be Completed”) in the given code using `DMDA` objects to solve the Poisson problem in 2-D:

$$-\Delta x = 0$$

with homogeneous Dirichlet boundary conditions.

Here the boundary points are included in the matrix.

The solver starts from a random x (`KSPSetInitialGuessNonzero` is set to true) and converges to the zero solution.