Introduction to PETSc: Vectors

Serge Van Criekingen

Maison de la Simulation

May 13, 2013

PETSc Vectors : Types & Create

A vector in PETSc is an object of type Vec.

Two basic types : sequential and parallel (MPI based)

In parallel, the vector is distibuted over all processes :
each process (i.e., MPI rank) stores its part of the vector.

PETSc Vectors : Types & Create

A vector in PETSc is an object of type Vec.

Two basic types : sequential and parallel (MPI based)

In parallel, the vector is distibuted over all processes :
each process (i.e., MPI rank) stores its part of the vector.

```
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 : Types & Create (cont'd)

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

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

Other way :

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

PETSc Vectors : Types & Create (cont'd)

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

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

Other way :

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

Yet another way :

```
VecCreate(MPI_COMM comm, Vec* x);
VecSetSizes(Vec x, int m, int M);
VecSetFromOptions(Vec x);
```

and use for instance `-vec_type mpi` at runtime.

PETSc Vectors : Types & Create (cont'd)

```
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,integer m,Vec x,
                                    PetscErrorCode ierr)
```

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

PETSc Vectors : Set

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

PETSc Vectors : Set

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

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

PETSc Vectors : Set

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

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

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

In fortran :

```
call VecSetValues(Vec x, integer n, integer(n) indices,
                                    PetscScalar(n) values,
                                    INSERT_VALUES or ADD_VALUES,
                                    PetscErrorCode ierr)
```

PETSc Vectors : Set (cont'd)

Notes :

- `VecSetValues` faster than `VecSetValue`.
  `VecSetValues` fastest if `n` large.

PETSc Vectors : Set (cont'd)

Notes :

- `VecSetValues` faster than `VecSetValue`.
  `VecSetValues` fastest if `n` large.

- *Global* indices have to be used in `VecSetValue` and `VecSetValues`.
  To use *local* indices :
  `VecSetValueLocal` and `VecSetValuesLocal`.

PETSc Vectors : Set (cont'd)

Notes :

- `VecSetValues` faster than `VecSetValue`.
  `VecSetValues` fastest if `n` large.

- *Global* indices have to be used in `VecSetValue` and `VecSetValues`.
  To use *local* indices :
  `VecSetValueLocal` and `VecSetValuesLocal`.

- Always 0-based indices in C and <u>`fortran`</u>.

PETSc Vectors : Assemble

After setting values, one must assemble the vector :

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

Note : allow overlap of communication and calculation.

PETSc Vectors : Assemble

After setting values, one must assemble the vector :

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

Note : allow overlap of communication and calculation.

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

PETSc Vectors : Viewing

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

In `fortran` :

```
call VecView(Vec x,PETSC_VIEWER_STDOUT_WORLD,
                              PetscErrorCode ierr)
```

PETSC_VIEWER_STDOUT_WORLD ≡ synchronized standard output :
All processors send their data to the first processor to print.

Other visualization contexts : see on-line documentation.

PETSc Vectors : Operations

- $$\begin{array}{cc} \texttt{VecScale} & x = a * x, \\ \texttt{VecAXPY} & y = a * x + y, \\ \texttt{VecDot} & x \cdot y, \\ \texttt{VecPointwiseMult} & w_i = x_i * y_i \\ \texttt{VecNorm} & \|A\|_{\dots} \\ \vdots & \vdots \end{array}$$

PETSc Vectors : Operations

- $$\begin{array}{cc} \texttt{VecScale} & x = a * x, \\ \texttt{VecAXPY} & y = a * x + y, \\ \texttt{VecDot} & x \cdot y, \\ \texttt{VecPointwiseMult} & w_i = x_i * y_i \\ \texttt{VecNorm} & \|A\| \ldots \\ \vdots & \vdots \end{array}$$

- `VecDuplicate` : y created with same type as x ;
  storage allocated for y but values *not copied* .
  `VecCopy` : $y \leftarrow x$ (y pre-existing),

PETSc Vectors : Operations

- $$\begin{array}{cc} \texttt{VecScale} & x = a * x, \\ \texttt{VecAXPY} & y = a * x + y, \\ \texttt{VecDot} & x \cdot y, \\ \texttt{VecPointwiseMult} & w_i = x_i * y_i \\ \texttt{VecNorm} & \|A\| \ldots \\ \vdots & \vdots \end{array}$$

- `VecDuplicate` : y created with same type as x ;
  storage allocated for y but values *not copied* .
  `VecCopy` : $y \leftarrow x$ (y pre-existing),

- `VecDestroy`

PETSc Vectors : Get

One can pull only local values from a vector.

- Single Value → `VecGetValues` (use global numbering)

PETSc Vectors : Get

One can pull only local values from a vector.

- Single Value → `VecGetValues` (use global numbering)
- All local elements → `VecGetArray` (no copy made, time-efficient) :

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

PETSc Vectors : Get

One can pull only local values from a vector.

- Single Value → `VecGetValues` (use global numbering)
- All local elements → `VecGetArray` (no copy made, time-efficient) :

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

In `fortran` :

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

PETSc Vectors : Get

One can pull only local values from a vector.

- Single Value → `VecGetValues` (use global numbering)
- All local elements → `VecGetArray` (no copy made, time-efficient) :

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

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 : get ownership range

- To obtain the range of indices owned by each processor :

```
VecGetOwnershipRange(Vec x, int* istart, int* iend);
```

In `fortran` :

```
call VecGetOwnershipRange(Vec x, integer istart,
                                 integer iend,
                                 PetscErrorCode ierr)
```

PETSc Vectors : Index Set

Index Set $\equiv$ generalization of a set of integer indices.

Type : `IS`

PETSc Vectors : Index Set

Index Set $\equiv$ generalization of a set of integer indices.

Type : `IS`

Two ways to create :

```
ISCreateGeneral(MPI_Comm comm, int n, int idx[],
                               PETSC_COPY_VALUES, IS* is);
```

```
ISCreateStride(MPI_Comm comm, int n, int first, int step,
                                                  IS* is);
```

PETSc Vectors : Index Set

Index Set ≡ generalization of a set of integer indices.

Type : `IS`

Two ways to create :

```
ISCreateGeneral(MPI_Comm comm, int n, int idx[],
                               PETSC_COPY_VALUES, IS* is);
```

```
ISCreateStride(MPI_Comm comm, int n, int first, int step,
                                                  IS* is);
```

To visualize :

```
ISView(IS is, PETSC_VIEWER_STDOUT_SELF
           or PETSC_VIEWER_STDOUT_WORLD);
```

PETSc Vectors : Scatters and Gathers

The `VecScatter` type decribes a context for both scatters and gathers.

To exchange data between the indices `isX` of `vecX`
and the indices `isY` of `vecY`
(length of `isX` = length of `isY`) :

```
VecScatterCreate(Vec vecX, IS isX, Vec vecY, IS isY,
                              VecScatter* the_context);
```

PETSc Vectors : Scatters and Gathers

The `VecScatter` type decribes a context for both scatters and gathers.

To exchange data between the indices `isX` of `vecX`
and the indices `isY` of `vecY`
(length of `isX` = length of `isY`) :

```
VecScatterCreate(Vec vecX, IS isX, Vec vecY, IS isY,
                                 VecScatter* the_context);
```

⇒ to copy elements from `X` to `Y` :

```
VecScatterBegin(the_context, vecX, vecY, INSERT_VALUES,
                                          SCATTER_FORWARD);
VecScatterEnd(the_context, vecX, vecY, INSERT_VALUES,
                                          SCATTER_FORWARD);
```

PETSc Vectors : Scatters and Gathers

The `VecScatter` type decribes a context for both scatters and gathers.

To exchange data between the indices `isX` of `vecX`
and the indices `isY` of `vecY`
(length of `isX` = length of `isY`) :

```
VecScatterCreate(Vec vecX, IS isX, Vec vecY, IS isY,
                                   VecScatter* the_context);
```

⇒ to copy elements from `X` to `Y` :

```
VecScatterBegin(the_context, vecX, vecY, INSERT_VALUES,
                                            SCATTER_FORWARD);
VecScatterEnd(the_context, vecX, vecY, INSERT_VALUES,
                                            SCATTER_FORWARD);
```

Notes : • Conventional scatter if `isX` = stride with step 1.
Conventional gather if `isY` = stride with step 1.

PETSc Vectors : Scatters and Gathers

The `VecScatter` type decribes a context for both scatters and gathers.

To exchange data between the indices `isX` of `vecX`
and the indices `isY` of `vecY`
(length of `isX` = length of `isY`) :

```
VecScatterCreate(Vec vecX, IS isX, Vec vecY, IS isY,
                                     VecScatter* the_context);
```

⇒ to copy elements from `X` to `Y` :

```
VecScatterBegin(the_context, vecX, vecY, INSERT_VALUES,
                                              SCATTER_FORWARD);
VecScatterEnd(the_context, vecX, vecY, INSERT_VALUES,
                                              SCATTER_FORWARD);
```

Notes :
- Conventional scatter if `isX` = stride with step 1.
  Conventional gather if `isY` = stride with step 1.
- `ADD_VALUES`, `SCATTER_BACKWARD` also available.

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
```

PETSc Vectors : Exercise 2

Duplicate the vector resulting from exercise 1
and copy the same values into it.

Use `VecDot` to compute the dot product of the two vectors and
verify that the result equals the square of the 2-norm (computed using
`VecNorm`).

PETSc Vectors : Exercise 3

Create a parallel vector of global size 100,000,000
and let PETSc decide the parallel distribution.

Use `VecGetOwnershipRange` to get the local indices
and set each vector value equal to its index
using first `VecSetValue`, then `VecSetValues`.

Compare the speeds of both options (using the Linux `time` command
or the `PetscGetTime` function from PETSc).

PETSc Vectors : Exercise 4

Use `VecScatter` to get off-process values :

- Create a parallel vector `theVecMPI` of global size 100
and set each vector value equal to its index as in exercice 3.

- Create a sequential vector `theVecSeq` of size 3
(in fact one sequential vector is created on each process).

- Create an index set for both the parallel and sequential vectors,
and use it to gather the elements 6, 45 and 97 from `theVecMPI`
on each local copy of the sequential vector `theVecSeq`.