Introduction to PETSc

Matrix

Jérémy Foulon

Institut du Calcul et de la Simulation - UPMC

14 mai 2013

About Matrices for PETSc

Matrix in PETSc are called *Mat*.
PETSc provides a large variety of matrix implementation because no single formats is appropriate for all problems.
Currently PETSc support dense storage and compressed sparse row storage in sequential and parallel versions, as well as several specialized formats. Additional formats can be added.
We present here the basic use of PETSc matrices involves in the following actions :

1. create a matrix with a particular type
2. insert/add values in the matrix
3. process the matrix
4. use the matrix
5. destroy the matrix

Documentation : all matrix routines : http://www.mcs.anl.gov/petsc/petsc-current/docs/manualpages/Mat/index.html

Creating matrix

Simplest routines for forming a PETSc Matrix :

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

- *comm* : *PETSC_COMM_SELF* (sequential application) or *PETSC_COMM_WORLD* (parallel application)
- *A* : a pointer on the matrix

```
MatSetSizes(Mat* A, int m, int n, int M, int N)
```

- *m* : local number of rows
- *n* : local number of columns
- *M* : global number of rows
- *N* : global number of columns

Creating matrix

Remarks :

1. the user specifies either the global or local matrix dimensions. Useless parameter can be replaced by the keyword *PETSC_DECIDE*
2. PETSc manage memory allocation
3. by default *MatCreate* use sparse AIJ format

Adding or inserting values in the matrix

To insert or add values entries to a matrix, one calls a variant of *MatSetValues*, either :

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

or

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

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

- *m* : number of rows
- *idxm* : global indexes of rows
- *n* : number of columns
- *idxn* : global indexes of columns
- *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]*.

Adding or inserting values in the matrix

Remarks :

1. row and column indices begin with zero (use C convention)
2. to insert values in column major order use the option (not supported by all sparse implementation of matrix) :

```
MatSetOption(Mat A, MAT_COLUMN_ORIENTED, PETSC_TRUE)
```

3. with block compressed sparse row format (*MatSeqBAIJ* or *MatMPIBAIJ*), we can use for more efficiency the routine :

```
MatSetValuesBlocked(...)
```

Assembling matrix

The routines for matrix processing are :

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

Remarks

1. by placing code between these two calls, the user can perform computation while messages are in transit
2. we cannot mixed *INSERT_VALUES ADD_VALUES* operations
3. for such intermediate assembly we can use *MAT_FLUSH_ASSEMBLY*

Sparse Matrix

The default matrix representation within PETSc is the general sparse AIJ format (CSR : Compressed Sparse Row format). An alternative is the Block Compressed Row and Block diagonal storage much more efficiency for problems with multiple degrees of freedom per node.
CSR format : use two arrays of integer (*row*, *col*) and one array of double (*val*).
With *i* the *i*-th indice of row

- *row[i+1] - row[i]* = number of non zeros values on the *i*-th row
- from *col[row[i]]* to *col[row[i+1] -1]* : list of non zeros column indices for the *i*-th row
- from *val[row[i]]* to *val[row[i+1] -1]* : non zeros values in the same order as list of non zeros indices of columns for *i*-th row

Sparse Matrix

Example : CSR format with arrays *row*, *col* and *val*.

$$\begin{pmatrix} 1 & -2 & 0 & 0 & 0 \\ -4 & 1 & -2 & 0 & 0 \\ 0 & 2 & 5 & 0 & 2 \\ 0 & 0 & 1 & -3 & 3 \\ 8 & 0 & 0 & 2 & 1 \end{pmatrix}$$

We obtain :

$$\begin{aligned} row &= \{0, 2, 5, 8, 11, 14\} \\ col &= \{0, 1, 0, 1, 2, 1, 2, 4, 2, 3, 4, 0, 3, 4\} \\ val &= \{1, -2, -4, 1, -2, 2, 5, 2, 1, -3, 3, 8, 2, 1\} \end{aligned}$$

Sequential AIJ Sparse Matrices

```
MatCreateSeqAIJ(PETSC_COMM_SELF, int m, int n, int nz, int*
nnz, Mat* A)
```

- *m* : number of rows
- *n* : number of columns
- *nz* : can be used to preallocate matrix memory. Constant number of non-zero values by rows. Can be set *nz=0*
- *nnz* : can be used to preallocate matrix memory. *nnz[i]* represents the number of non-zeros in the *i*-th rows. Can be set *nnz=PETSC_NULL*

Parallel AIJ Sparse Matrices

```
MatCreateMPIAIJ(PETSC_COMM_WORLD, int m, int n, int M, int
N, int d_nz, int* d_nnz, int o_nz, int* o_nnz, Mat* A)
```

- *m* : local number of rows, can be set *PETSC_DECIDE*
- *n* : local number of columns, can be set *PETSC_DECIDE*
- *M* : global number of rows, can be set *PETSC_DECIDE*
- *N* : global number of columns, can be set *PETSC_DECIDE*
- *d_nz* : analogous to *nz* for the diagonal portion of the local rows. Can be set *d_nz=0*
- *d_nnz* : analogous to *nnz* for the diagonal portion of the local rows. Can be set *d_nnz=PETSC_NULL*
- *o_nz* : analogous to *nz* for the off-diagonal portion of the local rows. Can be set *o_nz=0*
- *o_nnz* : analogous to *nnz* for the off-diagonal portion of the local rows. Can be set *o_nnz=PETSC_NULL*

Preallocation Memory for Parallel AIJ Sparse Matrices

Example :

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

The "diagonal" sub-matrix on the first process is given by :

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

While the "off-diagonal" sub-matrix is given by :

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

Preallocation Memory for Parallel AIJ Sparse Matrices

Processor 1 : $d\_nz = 2$ or alternatively $d\_nnz = \{2, 2, 2\}$ and $o\_nz = 2$ or alternatively $o\_nnz = \{2, 2, 2\}$.
Processor 2 : $d\_nz = 3$ (maximum of non-zero) or alternatively $d\_nnz = \{3, 3, 2\}$ and $o\_nz = 2$ or alternatively $o\_nnz = \{2, 1, 1\}$.
Processor 3 : $d\_nz = 1$ or alternatively $d\_nnz = \{1, 1\}$ and $o\_nz = 4$ or alternatively $o\_nnz = \{4, 4\}$.

Preallocation Memory for AIJ Sparse Matrices

Remarks

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

Dense Matrix

Sequential :

```
MatCreateSeqDense(PETSC_COMM_SELF, int m, int n,
PetscScalar* data, Mat* M)
```

Parallel :

```
MatCreateMPIDense(PETSC_COMM_WORLD, int m, int n, int M,
int N, PetscScalar* data, Mat* A)
```

- *m* : number global of rows, can be replace by *PETSC_DECIDE*
- *n* : number global of colums, can be replace by *PETSC_DECIDE*
- *M* : number local of rows, can be replace by *PETSC_DECIDE*
- *N* : number local of columns, can be replace by *PETSC_DECIDE*
- *data* : optional argument, indicate location of data for matrix storage, can be replace by *PETSC_NULL*

Matrix operations

Matrix-Vector product :

```
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 parallel matrix-vector operations.

Matrix operations

other matrix operations :

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

Matrix operations

Print a matrix

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

- *viewer* : use in general *PETSC_STDOUT_VIEWER_WORLD* or *PETSC_STDOUT_VIEWER_SELF* . There is additional viewers like : *PETSC_STDOUT_DRAW_WORLD* which draws non-zero structure of the matrix in X-default window

Destroy a matrix : frees space taken by a matrix

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

Exercice : création d'une matrice, affichage, destruction, ...

- Create a vector with value : $u[i] = (i + 1) * 10$
- Create the identity matrix
- Scale the matrix with a double value
- Multiply the matrix and the vector

Other matrix types

Some matrix types available in PETSc :

- *MATSEQBAIJ*
- *MATMPIBAIJ*
- *MATSEQSBAIJ*
- *MATMPISBAIJ*
- *MATSHELL*
- ....

Set type of the matrix :

```
MatSetType(Mat mat, const MatType matype)
```

Exercice 1

Soit l'équation

$$-\frac{\partial^2 u}{\partial x^2}(x,t) = f(x,t) \quad \forall x \in [0, L]$$

Pour résoudre numériquement le problème de Poisson par la méthode des différences finies, on discrétise en espace. Soit N est un entier positif, nous posons $h = \frac{1}{N+1}$ et $x_i = ih$ avec $i = 0, 1, 2, ...N + 1$. Soit u_i une approximation de $u(x)$ au point $x = x_i$. Nous noterons $u_i \simeq u(x_i), i = 1, 2, ..., N$. D'où :

$$\frac{1}{h^2}(-u_{i-1} + 2u_i - u_{i+1}) \quad = \quad f_i \quad i = 1, .., N \tag{1}$$

Exercice 1 (suite)

Soit A la matrice $N \times N$ tridiagonale définie par :

$$A = \frac{1}{h^2} \begin{bmatrix} 2 & -1 & 0 & 0 & \dots & 0 \\ -1 & 2 & -1 & 0 & \dots & 0 \\ 0 & -1 & 2 & -1 & \dots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & \dots & -1 & 2 & -1 \\ 0 & 0 & \dots & 0 & -1 & 2 \end{bmatrix}$$

On résout le système

$$Au = f$$

.

1. Assemble matrix from a finite difference method on Poisson equation
2. Use the preallocation method to improve efficiency on assembling. Increase the size of the discretization and use the command time to compare efficiency.

Exercice 2

Implement gradient conjugate method to solve a system Ax = b :

initialisation;

$r_0 := b - Ax_0, p_0 := r_0$;

while *(j <it_max && non_convergence)* **do**

$\alpha_j := (r_j, r_j)/Ap_j, p_j)$;

$x_{j+1} := x_j + \alpha_j p_j$;

$r_{j+1} := r_j - \alpha_j A p_j$;

$\beta_j := (r_{j+1}, r_{j+1})/(r_j, r_j)$;

$p_{j+1} := r_{j+1} + \beta_j p_j$;

end

Exercice 3

1. Solve the problem :

$$-\Delta u = f \text{ on } \Omega = [0, \Pi]$$

avec $f(x) = -sin(x)$. We define $u = 0$ on $x = 0$ et $x = \Pi$.

2. Print the solution with *gnuplot* : store the your solution and an exact solution in file.
File description for *gnuplot* :

$$\begin{matrix} x_0 & u_0 & sol_0 \\ x_1 & u_1 & sol_1 \\ \vdots & \vdots & \vdots \\ x_N & u_N & sol_N \end{matrix}$$

command to plot with gnuplot : *plot "filename.txt" u 1 :2 w l, "filename.txt" u 1 :3 wl*

Get information about the matrix

Returns the numbers of rows and columns in a matrix

```
MatGetSize(Mat mat,PetscInt *m,PetscInt* n)
```

or

Returns the number of rows and columns in a matrix stored locally

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

- m : number global or local of rows
- n : number global of local columns

Remarks : not collective function.

Get information about the matrix

Returns the range of matrix rows owned by this processor

```
MatGetOwnershipRange(Mat mat,PetscInt *m,PetscInt* n)
```

- m : global index of the first local row
- n : global index + 1 of the last local row

Returns the range of matrix rows owned by each process

```
MatGetOwnershipRanges(Mat mat,const PetscInt **ranges)
```

- *ranges* : returns the range of matrix rows owned by each process

Get information about the matrix

```
MatGetInfo(Mat mat,MatInfoType flag,MatInfo *info)
```

- *flag* : flag indicating the type of parameters to be returned (*MAT_LOCAL* - local matrix, *MAT_GLOBAL_MAX* - maximum over all processors, *MAT_GLOBAL_SUM* - sum over all processors)
- *info* : matrix information context

```
typedef struct {
  PetscLogDouble block_size; //block size
  //number of nonzeros
  PetscLogDouble nz_allocated,nz_used,nz_unneeded;
  PetscLogDouble memory;   //memory allocated
  PetscLogDouble assemblies; //nb of matrix assemblies called
  PetscLogDouble mallocs; //nb of mallocs during MatSetValues()
  // fill ratio for LU/ILU
  PetscLogDouble fill_ratio_given,fill_ratio_needed;
  // nb of mallocs during factorization
  PetscLogDouble factor_mallocs;
} MatInfo;
```

Get information about the matrix

Remarks :

- to get information about the matrix, we can also use the options : *-info* or *-mat_view_info*.
- to print "skeleton" of your matrix use the viewer *PETSC_VIEWER_DRAW_WORLD* or the option *-mat_view_draw* (with a sleep time : *-draw_pause 5*).

Get data store in the matrix

Gets a block of values from a matrix

```
MatGetValues(Mat mat,PetscInt m,const PetscInt idxm[]
,PetscInt n,const PetscInt idxn[],PetscScalar v[])
```

- *m* : number of rows
- *idxm* : global indices of rows
- *n* : number of columns
- *idxm* : global indices of columns
- *v* : a logically two-dimensional array for storing the values

Remark :

- not collective function, returns only local values

Get data store in the matrix

Gets a row of a matrix

```
MatGetRow(Mat mat,PetscInt row,PetscInt *ncols,const
PetscInt *cols[],const PetscScalar *vals[])
```

- *row* : indice of the row
- *ncols* : if not *NULL*, nb of non-zeros
- *cols* : if not *NULL*, indices column number
- *vals* : if not *NULL*, values

Remark :

- not collective function, return only local row

Get data store in the matrix

Gets the diagonal of a matrix

```
MatGetDiagonal(Mat mat,Vec v)
```

- v : vector with diagonal values

Remark :

- collective function

Get data store in the matrix

Extracts several submatrices from a matrix. If submat points to an array of valid matrices, they may be reused to store the new submatrices

```
MatGetSubMatrices(Mat mat,PetscInt n,const IS irow[],const
IS icol[],MatReuse scall,Mat *submat[])
```

or

Gets a single submatrix on the same number of processors as the original matrix

```
MatGetSubMatrix(Mat mat,IS isrow,IS iscol,MatReuse cll,Mat
*newmat)
```

Remark :

- collective functions

Collective functions to modify the matrix

Zeros all entries of a matrix. For sparse matrices this routine retains the old nonzero structure

```
MatZeroEntries(Mat mat)
```

Zeros all entries (except possibly the main diagonal) of a set of rows of a matrix

```
MatZeroRows(Mat mat,PetscInt numRows,const PetscInt rows[],PetscScalar diag,Vec x,Vec b)
```

- *numRows* : number of rows
- *rows* : indices of rows
- *diag* : values put in diagonal position
- *x* : optional vector of solutions for zeroed rows (other entries in vector are not used)
- *b* : optional vector of right hand side, that will be adjusted by provided solution

Collective functions to modify the matrix

Remarks :

- *MatZeroRowsLocal* : equivalent of *MatZeroRows* with local indices
- *MatZeroRowsColumns* : zeros all entries (except possibly the main diagonal) of a set of rows and columns of a matrix

Collective functions to modify the matrix

Calculates various norms of a matrix

```
MatNorm(Mat mat,NormType type,PetscReal *nrm)
```

- *type* : the type of norm, *NORM_1*, *NORM_FROBENIUS*, *NORM_INFINITY*
- *nm* : the resulting norm

Gets the norms of each column of a sparse or dense matrix

```
MatGetColumnNorms(Mat A,NormType type,PetscReal *norms)
```

- *type* : the type of norm, *NORM_1*, *NORM_2*, *NORM_INFINITY*
- *norms* : an array as large as the total number of columns in the matrix

CSRPreallocation

Allocates memory for a sparse parallel matrix in AIJ format

```
MatMPIAIJSetPreallocationCSR(Mat B,const PetscInt
i[],const PetscInt j[], const PetscScalar v[])
```

- i : the indices into j for the start of each local row
- j : the column indices for each local row
- v : optional values in the matrix

Exercice 4 : Solve Poisson finite element problem

Use and complete the file *main.c* in the repository EF to solve a finite element problem.

1. line 223 : define *start* and *end* integers which define range of the assembly loop. (Use the example with point in line 259).
2. assemble the finite element matrix
3. solve the system with conjugate gradient method develop previously
4. change the method to apply boundary conditions and use *MatZeroRow(...)* method.