PETSc : The Portable Extension Toolkit for Scientific Computing

Jérémy Foulon

Institut du Calcul et de la Simulation - Université Pierre et Marie Curie - Paris

22 janvier 2013

Librairie MPI : **M**essage **P**assage **I**nterface

- une librairie portable, standardisée, efficace de communication/échange parallèle entre processeurs
- conçue en 1993
- utilisable en langage C, C++ et Fortran
- un système d'exécution de job parallèle

Deux modèles d'exécution :

1. Single Program Multiple Data : le même programme est exécuté sur des données différentes
2. Multiple Program Multiple Data : différents programmes sont exécutés sur des données différentes

Échanges de messages :

- un processus *i* envoie des données *source* vers un processeur *j*
- le processeur *j* reçoit des données du processus *i* et les met dans une *cible*

PETSc : The Portable Extension Toolkit for Scientific Computing

Il s'agit d'une librairie fournissant des outils pour la résolution d'équations aux dérivées partielles sur des "supercomputers". C'est une librairie libre développée à Argone National Laboratory par S. Balay, J. Brown, K. Buschelman, V. Eijkhout, W. Gropp, D. Kaushik, M. Knepley, L. Curfman McInnes, B. Smith et H . Zhang.
Utilisable en C/C++, Fotran 77/90, et python,

Historique

- le développement a débuté en septembre 1991
- plus de 8 500 téléchargements depuis 1995, en moyenne 250 par mois

Financement

- Department of Energy,
  SciDAC (http://www.scidac.gov), MICS Program
- National Science Foundation,
  CIG, CISE, multidisciplinary Challenge Program

Quelques chiffres

Publications utilisant PETSc : 630

- Nano-simulations 51
- Biology – Medical 49
- Fusion 18
- Geosciences 67
- Environmental – Subsurface Flow 35
- CFD 74
- Wave propagation and the Helmholtz equation 13
- Optimization 11
- Fast Algorithms 3
- Other Application Areas 113
- Software Packages that use or interface to PETSc 51
- Software Engineering 47
- Algorithm design and analysis 78
- Books 20

Des logiciels utilisant PETSc

- PETSc-FEM : A General Purpose, Parallel, Multi-Physics FEM Program (http://www.cimec.org.ar/twiki/bin/view/Cimec/PETScFEM
- The FEniCS Project (http://fenicsproject.org)
- SLEPc : the Scalable Library For EigenValue Problem Computations (http://www.grycap.upv.es/slepc/
- FELiScE : a finite element library dedicated to life sciences and engineering problems (https://gforge.inria.fr/projects/felisce/
- FEEL++ : a C++ library for partial differential equation solves using generalized Galerkin methods i.e. fem, hp/fem, spectral methods. (http://www.feelpp.org)

Télécharger PETSc

- version téléchargeable ici : http://www.mcs.anl.gov/petsc/
- utilisable en C/C++, Fotran 77/90, et python
- libre pour tout le monde, incluant les industriels
- documentation/aide :
  - manuel : http://www.mcs.anl.gov/petsc/snapshots/petsc-dev/docs/manual.pdf
  - manuels utilisateurs en ligne pour chaque routine : http://www.mcs.anl.gov/petsc/documentation/
  - une centaine de tutoriels/exemples
  - FAQ : http://www.mcs.anl.gov/petsc/documentation/faq.html
  - support en ligne : petsc-maint@mcs.anl.gov

Dernière version de PETSc est : 3.3 parue en juin 2012.

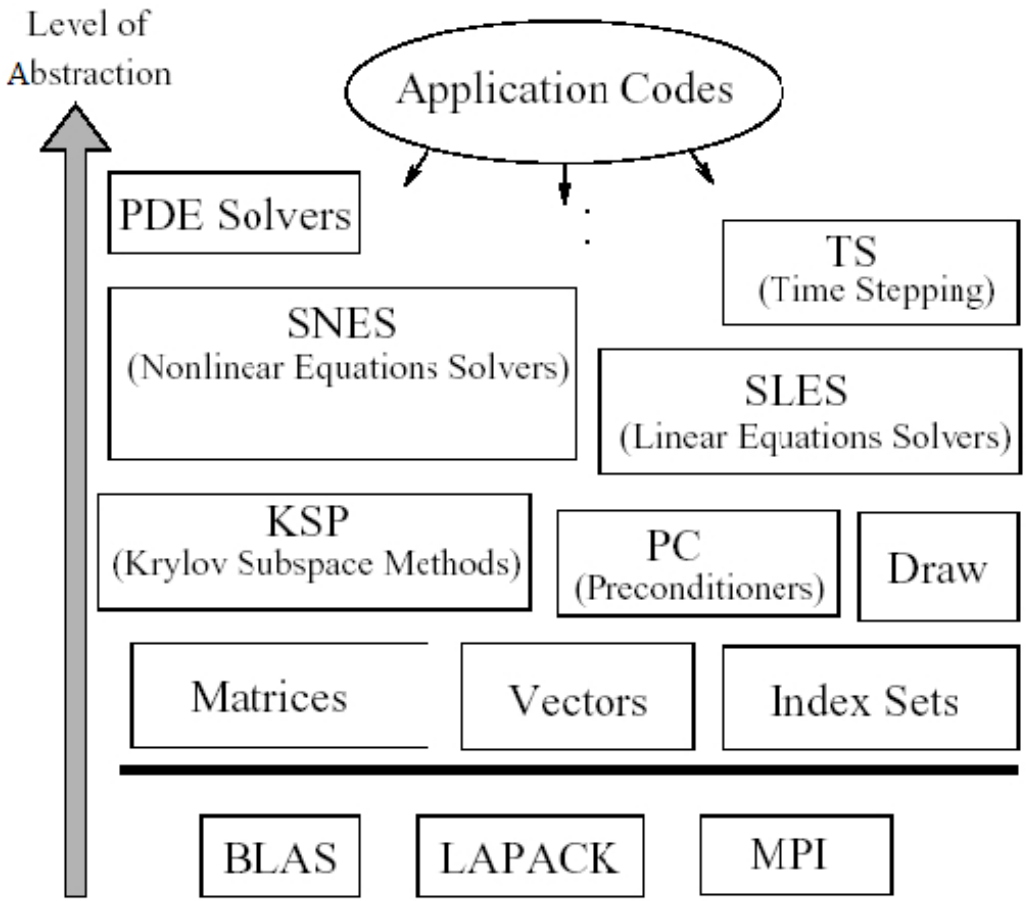
Organisation de la librairie

PETSc s'appuie sur l'implémentation du parallélisme avec MPI et sur les libraires d'algèbres linéaires : **BLAS** et **LAPACK** comme un assemblage de modules/librairies, similaire à des classes C++ et manipulant des familles d'objets. Il existe 8 modules :

- index sets (**IS**) : gestionnaire de numérotation
- vectors (**Vec**) : gestion des vecteurs
- matrices (**Mat**) : gestions des matrices
- DM object (**DM**) : gestionnaire des communications entre les structures algébriques (Vec & Mat) et les structures de données maillages
- Krylov subspace methods (**KSP**) : solveurs
- Preconditioners (**PC**) : préconditionneurs
- Nonlinear Solver (**SNES**)
- Timesteppers (**TS**) : gestion des pas de temps

Organisation de la librairie (2)

FIGURE : PETSc organisation



Les librairies externes interfacées

- FFTW : Fastest Fourier Transform in the West (http://www.fftw.org)
- Hypre : the LNLL preconditioner library (https://computation.llnl.gov/casc/linear_solvers/sls_hypre.html)
- MATLAB
- MUMPS : MUltifrontal Massively Parallel sparse direct Solver (http://graal.ens-lyon.fr/MUMPS/)
- ParMeTiS : parallel graph partitionner (http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview)
- PaStiX : a parallel LU and Cholesky solver package (http://pastix.gforge.inria.fr/files/README-txt.html)
- SuperLU & SuperLU_Dist : (http://crd-legacy.lbl.gov/~xiaoye/SuperLU/)
- UMFPACK : sparse direct solvers (http://www.cise.ufl.edu/research/sparse/umfpack/)
- ...

Configuration de PETSc 3.3 pour Ubuntu 12.04

Pré-requis

Installation de packages numériques et de développement suivant à l'aide de la commande : *sudo apt-get install*

- build-essential
- gfortran
- mpi-default-bin
- mpi-default-dev
- libx11-dev
- libboost-dev
- libblas-dev
- liblapack-dev
- cmake

Configuration de PETSc 3.3 pour Ubuntu 12.04 (2)

Pour obtenir une liste des options d'installation, il faut utiliser la commande suivante : *configure -h* dans le répertoire de PETSc .
Il est recommandé d'utiliser le téléchargement et l'installation automatique des librairies complémentaires ainsi vous vous assurez de leur bonne compatibilité.
On appelle *INSTALL_DIR* le répertoire d'installation du PETSc et *ARCHIVE_DIR* le répertoire contenant l'archive **petsc-3.3-p5**, puis on exécute l'ensemble des commandes suivantes (installation de base "enrichie" de solveurs supplémentaires) :

```
cd $INSTALL_DIR
tar xvzf $ARCHIVE_DIR/petsc-3.3-p5 -C .
cd petsc-3.3-p5
export PETSC_DIR=$INSTALL_DIR/petsc-3.3-p5
```

Configuration de PETSc 3.3 pour Ubuntu 12.04 (3)

- Compilation en mode "debug"

```
./configure \
        --with-debugging=yes \
        --with-dynamic-loading \
        --with-shared-libraries \
        --download-metis=1 \
        --download-parmetis=1 \
        --download-superlu_dist=1 \
        --download-scalapack=1 \
        --download-blacs=1 \
        --download-mumps=1 \
        --with-mpi-dir=/usr/lib/openmpi

make all
make test
```

Configuration de PETSc 3.3 pour Ubuntu 12.04 (4)

- Compilation en mode "release"

```
./configure \
--with-debugging=no \
--with-dynamic-loading \
--with-shared-libraries \
--download-metis=1 \
--download-parmetis=1 \
--download-superlu_dist=1 \
--download-scalapack=1 \
--download-blacs=1 \
--download-mumps=1 \
--with-mpi-dir=/usr/lib/openmpi
```

```
make all
make test
```

Exercice 1 : installation de PETSc

1. Ouvrir un terminal et se rendre dans le répertoire : *cd /usr/local/petsc-3.3-p5*
2. Lancer l'exécution du "configure" avec les options précédentes
3. Compiler avec les deux commandes successives : *make install*
4. Vérification de la bonne installation de la librairie avec la commande *make test*

Compilation d'un programme avec *makefile*

Définir les variables d'environnement *PETSC_DIR* et *PETSC_ARCH* dans le *makefile* ou dans le fichier *.bashrc* .
L'ensemble des *header* se trouvent dans le répertoire :
$(PETSC_DIR)/include.
L'ensemble des librairies se trouvent dans le répertoire :
$(PETSC_DIR)/$(PETSC_ARCH)/lib .
La librairie PETSc utilisant MPI , il est également nécessaire de définir la variable d'environnement *MPI_DIR* .
Dans notre cas, on aura :

```
PETSC_DIR=/usr/local/petsc-3.3-p5
PETSC_ARCH=/arch-linux2-c-debug
MPI_DIR=/usr/lib/openmpi
```

exemple : *Makefile*

Initialisation d'un programme : *PetscInitialize*

Syntaxe :

```
PetscErrorCode  PetscInitialize(int *argc,
                                char ***args,
                                const char file[],
                                const char help[])
```

Il est nécessaire d'inclure pour y accéder : *#include "petscsys.h"* Cette fonction permet l'initialisation de la librairie PETSc ainsi qu'MPI .
Les communicateurs *PETSC_COMM_WORLD* équivalent à *MPI_COMM_WORLD* (communicateur global entre les processeurs) et *PETSC_COMM_SELF* équivalent à *MPI_COMM_SELF* (communicateur associé à un processeur seul).

Finalisation d'un programme : *PetscFinalize*

Comme en MPI il est nécessaire de "terminer" les processus en cours d'exécution par l'intermédiaire de la fonction "finalize" .
Syntaxe :

```
PetscErrorCode  PetscFinalize(void)
```

La structure d'un programme PETSc

```
static char help[] = "Program \n";
int main(int argc,char **argv)
{
      PetscInitialize(&argc,&argv,(char *)0,help);
      ..... // corps du programme
      PetscFinalize();
      return 0;
}
```

L'affichage avec PETSc

La fonction C d'affichage standard *printf(..)* est redéfinie sous la forme :

```
PetscErrorCode  PetscPrintf(MPI_Comm comm,
                            const char format[],...)
```

Le changement principal venant de l'ajout du paramètre **comm**.
L'affichage est effectué par le 1er processeur du groupe.
De la même manière, la fonction *fprintf(..)* est redéfinie par la fonction :

```
PetscErrorCode  PetscFPrintf(MPI_Comm comm,FILE* fd,
                             const char format[],...)
```

Le fonctionnement est similaire à la fonction *PetscPrintf*.

Hello World avec PETSc

```
//First program with Petsc : Hello World in parallel
#include <petscsys.h>

static char help[] = "Hello \n";
int main(int argc,char **argv)
{
        PetscInitialize(&argc,&argv,(char *)0,help);
        PetscPrintf(PETSC_COMM_SELF,"Hello world !\n");
        PetscFinalize();
        return 0;
}
```

Exercice 2 : affichage

1. Écrire un programme dans lequel chaque processeur indique son id pour le communicateur global (PETSC_COMM_WORLD).

Remarque : toutes les fonctionnalités C et MPI sont utilisables

Exercice 3 : parallélisation de l'algorithme de calcul de π

π peut-être calculé par intégration :

$$\pi = \int_0^1 f(x)dx \text{ avec } f(x) = \frac{4}{a + x^2}$$

Une approximation de cette relation est :

$$\pi = h * \sum_{i=1}^{n} f(x_{i-1/2}) \text{ avec } h = \frac{1}{n} \text{ et } x_{i-1/2} = \frac{i - 1/2}{n}$$

code séquentielle : *pi_seq.c*

1. paralléliser à l'aide de PETSc .

Les types de bases

- *PetscInt* : représente les entiers (par défaut 32 bits mais 64 bits avec l'option de configuration *–with-64-bit-indices*)
- *PetscScalar* : représente les réels "double précision"
- *PetscBool* : variable logique pouvant prendre les valeurs *PETSC_TRUE* ou *PETSC_FALSE*
- *PetscMPIInt* : identique à *PetscInt* sauf en 64-bits où ils sont en 32-bits

Quelques mots-clés

- *PETSC_NULL* : évite de définir un tableau ou un pointeur "inutile" et "null". La fonction utilise un paramètre par défaut dans ce cas.
- *PETSC_IGNORE* : similaire à *PETSC_NULL*, on ignore le paramètre de la fonction
- *PETSC_DEFAULT* : la fonction utilise un paramètre par défaut
- *PETSC_DECIDE* : passage d'un paramètre entier ou flottant par défaut
- *PETSC_DETERMINE* : PETSc évalue la valeur "optimale" à choisir

La programmation avec PETSc

Objectifs

- portabilité : exécution possible sur "toutes" les architectures et "reproductibilité" des résultats
- haute performance : HPC
- scalabilité

Approche

- mémoire distribué : "ne rien partager"
- pas de compilateur spéciaux
- masquer les communications entre les objets
- l'utilisateur organise les communications à un plus haut niveau d'abstraction

Travail collectif

Définition de communicateurs MPI collectif

- un processus est associé à un calcul

Les constructeurs sont collectifs dans un communicateur

- *VecCreate(MPI_Comm comm, Vec* v)*
- on utilize *PETSC_COMM_WORLD* : pour tous les processeurs et *PETSC_COMM_SELF* pour un seul

Des opérations collectives et d'autres pas :

- *VecNorm* : collectif
- *VecGetLocalSize()* : non collectif

L'interface de base d'un objet

Chaque objet PETSc possède au minimum l'interface suivante :

- *Create()* : création de l'objet
- *Get/SetName()* : nommer l'objet
- *Get/SetType()* : choisir le type de donnée (ex : parallèle ou séquentielle)
- *Get/SetOptionPrefix()* : fixer les options
- *SetFromOptions()* : paramètre l'objet suivant les options passées en ligne de commande
- *SetUp()* : prépare les structures données internes à utiliser
- *View()* : affiche l'objet
- *Destroy()* : détruit l'objet

Tous les objets supportent l'option *-help*.

Petsc Vectors

Qu'est ce qu'un vecteur PETSc ?

- un objet pour stocker des champs solutions, second-membre, etc.
- chaque processeur stocke son propre sous-vecteur de données contigues

Comment créer un vecteur ?

- *VecCreate(MPI_Comm, Vec *)*
- *VecSetSizes(Vec, int n, int N)*
- *VecSetType(Vec, VecType typeName)*
- *VecSetFromOptions(Vec)*

Petsc Vectors (2)

Un vecteur PETSc :

- a une interface directe avec les valeurs
- supporte toutes les opérations de l'espace des vecteurs : *VecDot*, *VecNorm*, *VecScale*
- supporte des opérations non-usuelles : *VecSqrtAbs()* (remplacer chaque composantes par sa racine carrée de son module)
- gère les communications automatiquement pendant son assemblage
- possède des méthodes de communications adaptées (scatters)

Création d'un vecteur

```
Vec x;
PetscInt N;
PetscErrorCode ierr;

ierr = PetscInitialize(&argc, &argv, PETSC_NULL, PETSC_NULL);
CHKERRQ(ierr);
ierr = VecCreate(PETSC_COMM_WORLD, &x);CHKERRQ(ierr);
ierr = VecSetSizes(x, PETSC_DECIDE, N);CHKERRQ(ierr);
ierr = VecSetType(x, "mpi");CHKERRQ(ierr);
ierr = VecSetFromOptions(x);CHKERRQ(ierr);
ierr = VecDestroy(&x);CHKERRQ(ierr);
ierr = PetscFinalize();CHKERRQ(ierr);
```

VecSetSizes

PetscErrorCode VecSetSizes(Vec v, PetscInt n, PetscInt N)

- *n* : dimension locale
- *N* : dimension globale

VecSetType

PetscErrorCode VecSetType(Vec vec, const VecType method)

- *method* : nom du type de vecteur

Les différents types de vecteurs :

1. *VECMPI* = "mpi" : vecteur parallel basique
2. *VECSEQ* = "seq" : vecteur sequentiel basique
3. *VECSHARED* = "shared" : vecteur parallèle utilisant de la mémoire partagée

Affichage d'un vecteur

PetscErrorCode VecView(Vec vec,PetscViewer viewer) avec l'un des "viewer" (le contexte d'affichage) suivants :

- *PETSC_VIEWER_STDOUT_SELF* : sortie standard
- *PETSC_VIEWER_STDOUT_WORLD* : sortie synchronisée, toutes les données sont envoyés au processeur 0 qui affiche le vecteur

Exercice 4 : création de vecteurs

1. Construire 2 vecteurs de taille 10, l'un séquentiel, l'autre parallèle.
2. Afficher les deux vecteurs.
3. Utiliser la fonction *VecGetSize* (voir document html) pour afficher la dimension des vecteurs.
4. Executer le code en séquentiel et en parallèle avec 2 processeurs

Remarque : on aurait pu utiliser les fonctions suivantes :

- *PetscErrorCode VecCreateSeq(MPI_Comm comm,PetscInt n,Vec *v)* : avec le communicateur *PETSC_COMM_SELF*
- *PetscErrorCode VecCreateMPI(MPI_Comm comm,PetscInt n,PetscInt N,Vec *v)* : avec le communicateur *PETSC_COMM_WORLD*

Initialisation d'un vecteur

- *PetscErrorCode VecZeroEntries(Vec vec)* : chaque composante du vecteur est mise à zéro
- *PetscErrorCode VecSet(Vec x,PetscScalar alpha)* : toutes les composantes ont pour valeur alpha
- *PetscErrorCode VecSetRandom(Vec x,PetscRandom rctx)* : le vecteur est initialisé de façon aléatoire

exemple : *vecInit.c*

Assemblage en parallèle de vecteur

Un processus en 3 étapes :

1. chaque processeur ajoute ou modifie des composantes
2. début de la communications pour échanger les valeurs entre les processeurs
3. fin de la communication

On utilise la fonction : *PetscErrorCode VecSetValues(Vec x,PetscInt ni,const PetscInt ix[],const PetscScalar y[],InsertMode iora)* avec deux modes *INSERT_VALUES* et *ADD_VALUES*.
L'assemblage se fait en deux phases permettant la superposition de communication et de calcul (en pratique, on ajoute pas deux dírectives entre les deux appels suivants) :

- *VecAssemblyBegin(Vec v)*
- *VecAssemblyEnd(Vec v)*

Une méthode de modifier les valeurs dans un vecteur

```
ierr = VecGetSize(x, &N);CHKERRQ(ierr);
ierr = MPI_Comm_rank(PETSC_COMM_WORLD, &rank);CHKERRQ(ierr);
if (rank == 0) {
for(i = 0, val = 0.0; i < N; i++, val += 10.0) {
    ierr = VecSetValues(x, 1, &i, &val, INSERT VALUES);
    CHKERRQ(ierr);
}
}
/* These two routines ensure that the data is distributed to
    the other processes */
ierr = VecAssemblyBegin(x);CHKERRQ(ierr);
ierr = VecAssemblyEnd(x);CHKERRQ(ierr);
```

Une méthode plus efficace de modifier les valeurs dans un vecteur

```
ierr = VecGetOwnershipRange(x, &low, &high);CHKERRQ(ierr);
for(i = low, val = low*10.0; i < high; i++, val += 10.0) {
    ierr = VecSetValues(x, 1, &i, &val, INSERT_VALUES);
    CHKERRQ(ierr);
}
/* These two routines ensure that the data is distributed to
the other processes */
ierr = VecAssemblyBegin(x);CHKERRQ(ierr);
ierr = VecAssemblyEnd(x);CHKERRQ(ierr);
```

Remarques sur l'assemblage

1. lorsqu'on agit sur une seule valeur on peut utiliser la fonction *PetscErrorCode VecSetValue(Vec v,int row,PetscScalar value, InsertMode mode)*
2. si possible et pour une plus grande efficacité il est préférable d'assembler plusieurs valeurs en un appel de la fonction *VecSetValues*
3. on peu également utiliser les fonctions *VecSetValueLocal* et *VecSetValuesLocal* qui utilisent locaux de vecteurs plutôt que des indices globaux

Une sélection d'opérations sur les vecteurs

- *VecAXPY(Vec y, PetscScalar a, Vec x)* : $y = y + a * x$
- *VecAYPX(Vec y, PetscScalar a, Vec x)* : $y = x + a * y$
- *VecWAYPX(Vec w, PetscScalar a, Vec x, Vec y)* : $w = y + a * x$
- *VecScale(Vec x, PetscScalar a)* : $x = a * x$
- *VecCopy(Vec y, Vec x)* : $y = x$
- *VecPointwiseMult(Vec w, Vec x, Vec y)* : $w_i = x_i * y_i$
- *VecMax(Vec x, PetscInt *idx, PetscScalar *r)* : $r = maxx_i$
- *VecShift(Vec x, PetscScalar r)* : $x_i = x_i + r$
- *VecAbs(Vec x)* : $x_i = |x_i|$
- *VecNorm(Vec x, NormType type, PetscReal *r)* : $r = x$

Remarques :

1. la fonction *VecDuplicate* crée une copie de vecteur du même avec ou sans copie des valeurs
2. les normes : *NORM_1* , *NORM_2*, *NORM_INFINITY* et *NORM_MAX*

Exercice 5 : manipulation de vecteurs

1. créer et initialiser plusieurs vecteurs avec les méthodes suivantes : *VecSetValue*, *VecSetValues*, *VecSetValueLocal* et *VecSetValuesLocal*
2. tester l'une des trois fonctions : *VecAXPY*, *VecAYX*, *VecWAYPX*
3. effectuer le calcul de la norme infini pour le vecteur de votre choix
4. chercher dans la documentation la méthode "produit scalaire" et l'utiliser

Travailler localement avec le sous-vecteur

Il est parfois utile de pouvoir manipuler les données localement pour cela on utilise la fonction :

- *VecGetArray(Vec, double* [])*

à la fin de l'utilisation des données, ne pas oublier d'appeler :

- *VecRestoreArray(Vec, double* [])*

pour éviter des problèmes de fuite mémoire.
exercice 6 : implémenter la méthode *VecGetArray*

Utilisation *VecScatter*

Il est parfois utile de récupérer l'ensemble des données d'un vecteur sur chaque processeur. Pour cela, on utilise les outils *VecScatter* de PETSc comme dans l'exemple suivant où l'on stocke l'intégralité du vecteur *solution* dans le vecteur *vOut* .

```
Vec vOut;
VecScatter vecscat;
VecScatterCreateToAll(solution,&vecscat,&vOut);
VecScatterBegin(vecscat,solution,vOut,INSERT_VALUES,
SCATTER_FORWARD);
VecScatterEnd(vecscat,solution,vOut,INSERT_VALUES,
SCATTER_FORWARD);
VECSCATTERDESTROY(vecscat);
```

PETSc Matrix

Qu'est ce qu'une matrice ?

- un objet pour stocker des matrices de rigidité, de masse, etc
- chaque processeur stocke localement un ensemble de ligne contigu de la matrice
- plusieurs types de données disponibles : *AIJ* , *Block AIJ*, *Symmetric AIJ*, *Block Diagonal*, etc
- plusieurs structures afin de pouvoir utiliser de nombreux packages : *MUMPS*, *Spooles*, *SuperLU*, *UMFPack*, *DSCPack*, ...

Comment créer une matrice ?

- *MatCreate(MPI Comm, Mat *)*
- *MatSetSizes(Mat, int m, int n, int M, int N)*
- *MatSetType(Mat, MatType typeName)*
- *MatSetFromOptions(Mat)*
- *MatSetValues(Mat,...)*

La définition d'une matrice PETSc

- *MatCreate(MPI Comm, Mat*)*
- *MatSetSizes(Mat, int m, int n, int M, int N)*
  - *m* : nombre de lignes locales ou PETSC_DECIDE
  - *n* : nombre de colonnes locales ou PETSC_DECIDE
  - *M* : nombre de lignes globales ou PETSC_DECIDE
  - *N* : nombre de colonnes globales ou PETSC_DECIDE
- *MatSetType(Mat, MatType typeName)*
  Les principaux types de matrice :
  - *MATAIJ* = *MATMPIAIJ* en parallèle et *MATSEQAIJ* en sequentiel
  - *MATBAIJ* = *MATMPIBAIJ* en parallèle et *MATSEQBAIJ* en sequentiel
  - *MATDENSE* = *MATMPIDENSE* en parallèle et *MATSEQDENSE* en sequentiel :
- *MatSetFromOptions(Mat)*

Assemblage/Remplissage de la matrice

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

- *m*, *idxm* : nombre et indices de lignes
- *n*, *idxn* : nombre et indices de colonnes
- *addv* : *INSERT_VALUES* ou *ADD_VALUES*

On finalise l'assemblage de la matrice avec l'appel des deux fonctions suivantes :

- MatAssemblyBegin(Mat mat,MatAssemblyType type)
- MatAssemblyEnd(Mat mat,MatAssemblyType type)
  - *type* : en général *MAT_FINAL_ASSEMBLY* sauf si on change/switch de mode "insertion" alors *MAT_FLUSH_ASSEMBLY*

Un exemple d'assemblage

```
values[0] = -1.0; values[1] = 2.0; values[2] = -1.0;
   if (rank == 0) { /* Only one process creates matrix */
      for(row = 0; row < N; row++) {
         cols[0] = row - 1; cols[1] = row; cols[2] = row+1;
         if (row == 0) {
            ierr = MatSetValues(A, 1, &row, 2, &cols[1],
             &values[1], INSERT_VALUES);CHKERRQ(ierr);
         } else if (row == N-1) {
            ierr = MatSetValues(A, 1, &row, 2, &cols[0],
             &values[0], INSERT_VALUES);CHKERRQ(ierr);
         } else {
            ierr = MatSetValues(A, 1, &row, 3, &cols[0],
             &values[0], INSERT_VALUES);CHKERRQ(ierr);
         }
      }
   }
```

Affichage d'une matrice

MatView(Mat mat,PetscViewer viewer)

- *PETSC_VIEWER_STDOUT_SELF* : sortie standard (default)
- *PETSC_VIEWER_STDOUT_WORLD* : sortie synchronisée, seul le 1er processeur affiche
- *PETSC_VIEWER_DRAW_WORLD* : affiche la structure "creuse" de la matrice

Matrice creuse parallèle

Remarques :

- chaque processeur stocke en mémoire un ensemble de lignes de la matrice
- chaque sous-matrice est constitué d'une partie diagonale et d'une partie non-diagonale

La fonction *MatGetOwnershipRange(Mat A, int *start, int *end)* permet d'identifier les lignes associées à chaque processeur.

- *start* : premier indice appartenant au processeur dans la numérotation globale
- *end* : dernier indice appartenant au processeur dans la numérotation globale

exemple : *matCreate.c*

Un exemple d'assemblage "optimisé"

```
values[0] = -1.0; values[1] = 2.0; values[2] = -1.0;
for(row = start; row < end; row++) {
   cols[0] = row-1; cols[1] = row; cols[2] = row+1;
   if (row == 0) {
      ierr = MatSetValues(A, 1, &row, 2, &cols[1], &values[1],
      INSERT_VALUES);CHKERRQ(ierr);
   } else if (row == N-1) {
    ierr = MatSetValues(A, 1, &row, 2, cols, values,
    INSERT_VALUES);CHKERRQ(ierr);
   } else {
     ierr = MatSetValues(A, 1, &row, 3, cols, values,
     INSERT_VALUES);CHKERRQ(ierr);
   }
}
ierr = MatAssemblyBegin(A, MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
ierr = MatAssemblyEnd(A, MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
```

1ère méthode de pré-allocation mémoire de la matrice

- *MatSeqAIJSetPreallocation(Mat A,PetscInt nz,const PetscInt nnz[])*
  - *nz* : nombre de non-zeros par ligne (identique pour chaque ligne)
  - *nnz* : tableau contenant le nombre de non-zeros par ligne (peut-être différent pour chaque ligne) or *PETSC_NULL*

  Si on fournit *nez*, *nz* n'est pas pris en compte.
- *MatMPIAIJSetPreallocation(Mat A,PetscInt d_nz,const PetscInt d_nnz[],PetscInt o_nz,const PetscInt o_nnz[])*
  - *d_nz* : nombre de non-zeros par ligne dans la portion diagonale
  - *d_nnz* : tableau contenant le nombre de non-zeros par ligne dans la portion diagonale or *PETSC_NULL*
  - *o_nz* : nombre de non-zeros par ligne pour la partie non-diagonale
  - *o_nnz* : tableau contenant le nombre de non-zeros par ligne dans la portion non diagonale or *PETSC_NULL*

exemple : voir la documentation html des fonctions.

2ème méthode de pré-allocation mémoire de la matrice

- *MatSeqAIJSetPreallocationCSR(Mat B,const PetscInt i[],const PetscInt j[],const PetscScalar v[])*
- *MatMPIAIJSetPreallocationCSR(Mat B,const PetscInt i[],const PetscInt j[], const PetscScalar v[])*
  - i : les indices dans j où chaque ligne commence (commence à 0)
  - j : les indices de colonnes non nuls pour chaque (commençant à 0)
  - v : valeurs optionnelles à insérer dans la matrice

Remarques :

- ces méthodes la pré-allocation de la mémoire associée à la matrice à partir de son squelette pré-calculé et stocké au format CSR (Compress Sparse Row)
- l'utilisation de ces fonctions peut rendre l'assemblage d'une matrice plus de 50 fois plus rapide

Quelques opérations classiques sur les matrices

- *MatMult(Mat mat,Vec x,Vec y)* : produit matrice-vecteur, y = A*x
- *MatMultAdd(Mat mat,Vec v1,Vec v2,Vec v3)* : v3 = v2 + A*v1
- *MatCopy(Mat A,Mat B,MatStructure str)* : copie de matrice
  - *str* : *SAME_NONZERO_PATTERN* ou *DIFFERENT_NONZERO_PATTERN*
- *MatAXPY(Mat Y,PetscScalar a,Mat X,MatStructure str)* : Y = a*X + Y
- *MatAYPX(Mat Y,PetscScalar a,Mat X,MatStructure str)* : Y = a*Y + X
- *MatEqual(Mat A,Mat B,PetscBool *flg)* : compare deux matrices
- *MatIsSymmetric(Mat A,PetscReal tol,PetscBool *flg)* : test la symétrie de la matrice selon la valeur de tolérance *tol*
- *MatNorm(Mat mat,NormType type,PetscReal *nrm)* : calcul de la norme de la matrice
- *MatScale(Mat mat,PetscScalar a)* : "scale" tous les coefficients de la matrice
- *MatShift(Mat Y,PetscScalar a)* : Y = Y + a*I (I : identité)

Exercice 7 :

1. Créer une matrice parallèle identité, A
2. Créer un vecteur parallèle aléatoire, x
3. Calculer la norme infini du vecteur x
4. Effectuer la produit matrice-vecteur, $A*x$

Exercice 8 :

1. Construire le squelette de la matrice "différences finies" de l'équation de Poisson 1D pour une matrice parallèle de taille N (stocké au format CSR).
2. Effectuer l'assemblage de la matrice avec et sans la préallocation mémoire de type CSR pour une grande matrice. Remarque on une différence ?

Exercice 9 :

1. Implémenter l'algorithme du Gradient Conjugué à l'aide de la librairie PETSc

KSP : linear equations solvers

L'objet **KSP** est le coeur de PETSc parce qu'il permet l'utilisation d'un grand de package de résolution de système linéaire, parallèle et séquentiel, direct ou itératif. KSP est prévu pour résoudre des systèmes non singuliers de la forme :

$$Ax = b$$

provenant de l'étude des EDPs.
Dans la résolution de ces problèmes, l'utilisation de préconditionneurs est souvent indispensable. S'ajoute au module **KSP** un module **PC** donnant accès à un grand nombre de préconditionneurs.

Création d'un objet **KSP**

syntaxe : *KSPCreate(MPI_Comm comm, KSP *ksp)*
comm est le communicator associé à *ksp* qui forme le "contexte du solveur". Avant de résoudre, l'utilisateur doit relier la matrice au solveur en utilisant :
KSPSetOperators(KSP ksp, Mat Amat, Mat Pmat, MatStructure flag)

- *Amat* : correspond à la matrice du système à résoudre
- *Pmat* : est la "matrice de préconditionnement" (matrice à partir de laquelle on construit le preconditionneur). Elle est souvent identique à celle définissant le système linéaire, en tant que paramètre de la fonction.
- *flat* permet d'éliminer le travail inutile lorsque l'on répète la résolution d'un système linéaire de même taille et utilisant les même options de préconditionnement.

Création d'un objet **KSP** (2)

flat peut prendre les valeurs suivantes :

- *SAME_NONZERO_PATTERN* : la matrice de préconditionnement garde la même structure non zéro pendant les résolutions successives
- *DIFFERENT_NONZERO_PATTERN* : la matrice de préconditionnement ne conserve pas la même structure non zéro pendant les résolutions successives
- *SAME_PRECONDITIONER* : la matrice de préconditionnement est identique à celle de la résolution précédente

si la structure de la matrice n'est pas connue, on utilise *DIFFERENT_NONZERO_PATTERN*. On finalize la création de *ksp* avec : *KSPSetFromOptions(KSP ksp) ;*.

Résolution de $Ax = b$

On peut ensuite faire appel à la fonction : *KSPSolve(KSP ksp, Vec b, Vec x)*

- b : le second membre
- x : la solution

Le nombre d'itération est disponible à l'aide de la fonction

- *KSPGetIterationNumber(KSP ksp, int *its)*

Une fois la résolution terminée, on peut détruire l'objet **KSP** :

- *KSPDestroy(ksp)*

Remarque : le solver par défaut dans PETSc est **GMRES** préconditionné en séquentiel par **ILU(0)** et en parallèle par la **méthode Jacobi par blocks**.

Choix des options solveurs et/ou préconditionneur

Afin que l'utilisateur puisse utiliser les options de son choix, il est nécessaire d'appeler la fonction suivante :

- *KSPGetPC(KSP ksp, PC *pc)*

L'utilisateur peut ainsi appeler les routines associées à chaque objet *ksp* et *pc*.

Les options de **KSP**

À l'aide de la fonction *KSPSetType(KSP ksp, KSPType method)*

- *method* pouvant prendre les valeurs suivantes : *KSPRICHARDSON*, *KSPCHEBYCHEV*, *KSPCG*, *KSPGMRES*, *KSPTCQMR*, *KSPBCGS*, *KSPCGS*,...,*KSPPREONLY*

dont certains peuvent être paramètres à l'aide de fonction particulière :

- *KSPRichardsonSetScale(KSP ksp, double damping_factor*
- *KSPChebychevSetEigenvalues(KSP ksp, double emax, double emin)*
- *KSPGMRESSetRestart(KSP ksp, int max_steps)*
- ....

Les options de **PC**

On fixe le choix du préconditionneur à l'aide de la fonction *PCSetType(PC pc,const PCType type)*

- *type* pouvant prendre les valeurs suivantes :*PCNONE*, *PCJACOBI*, *PCSOR*, *PCLU*, *PCILU*, ...

Chaque préconditionneur est ensuite paramétrable par l'intermédiaire des fonctions associées à l'objet *PC* telle que :

- *PCFactorSetFill(PC pc,PetscReal fill)*

Tests de convergence

Le test de convergence par défaut, *KSPDefaultConverged()* est basé sur la norme l_2 du résidu. La convergence est contrôlé par la diminution de la norme relative du résidu vers la norme du RHS, *rtol*, par la taille absolue de la norme du résidu, *atol*, et l'accroissement du résidu, *dtol* (p. 75 du manuel utilisateur).
On peut paramétrer ces paramètres grâce à la fonction :

- *KSPSetTolerances(KSP ksp, double rtol, double atol, double dtol, int maxits)*

Les valeurs par défauts sont : *rtol*=10^{-5}, *atol*=10^{-50} et *dtol*=10^5 and *maxits*=10^5 .
Il est possible de surveiller/"monitorer" la convergence avec la fonction :

- *KSPMonitorSet(...)*

Exercice 10 :

1. Étudier l'exemple dans le répertoire */usr/lib//petsc-3.3-p5/src/ksp/ksp/examples/tutorials/ex12.c.html*.
2. Compiler et exécuter le programme.
3. Ajouter une fonction d'affichage *KSPView*
4. Modifier le pré-conditionneur PC pour utiliser le préconditionner "additive schwartz"
5. Modifier le solver pour utiliser le solver itératif "GMRES" avec *restant 50*
6. Modifier le solver pour utiliser le solver itératif "gradient conjugué"
7. Modifier le solver pour utiliser le solver direct "MUMPS"

Exercice 11 :

1. Résoudre un problème de Poisson 2D par une méthode éléments finis sur un maillage triangulaire sur le domaine $[0;1]x[0;1]$