Sparse systems with PETSc

TMA4280—Introduction to Supercomputing

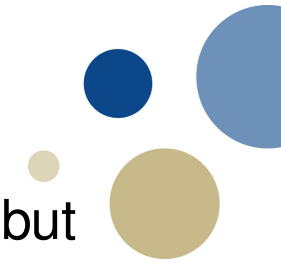
NTNU, IMF

April 13. 2018

Linear algebra packages

- During the course BLAS and LAPACK were discussed for dense linear algebra.
- Most engineering applications involve solving sparse linear systems.
- For instance Partial Differential Equation discretizations involve:
  - discrete differential operators in the form of stencils (finite differences, finite volums)
  - discrete spaces with compactly supported basis functions (finite elements)
- This locality of the discretization translates into the structure of the matrix.
- Sparse linear algebra packages are available:
  - MUMPS: MUltifrontal Massively Parallel sparse direct Solver,
  - UMFPACK: Unsymmetric MultiFrontal method,
  - SuperLU: Sparse, direct solver (comes in threaded and distributed variants but no hybrid)
  - HYPRE: Sparse, iterative solvers and preconditioners

Linear algebra packages



Some linear algebra packages provide more then linear solvers, but contain other tools that are helpful in scientific computing as well.

1. **PETSc**: Portable Extensible Toolkit for Scientific computing.

   https://www.mcs.anl.gov/petsc/

2. **Trilinos**: Object-oriented software framework for the solution of large-scale, complex multi-physics engineering and scientific problems (can be used trough PETSc).

   https://trilinos.org/

Such frameworks offer generic interfaces which can be also used for external packages: it is interesting to implement them in your code to be able to use a common interface.

PETSc

- PETSc (PET-see) is an open-source scientific computing library written in C.
- Written in C with an object-oriented design.
- Interfaces for Fortran, Python, Java and Matlab available.
- Great support for distributed programming: MPI, and GPUs through CUDA or OpenCL, as well as hybrid MPI-GPU parallelism.

- Parallel vectors with synchronization of ghost entries.
- Parallel matrices with different storage foramts and distributed implementations.
- Linear solvers: direct and preconditioned iterative methods for sparse matrices (https://www.mcs.anl.gov/petsc/documentation/linearsolvertable.html).
- Parallel Newton-based nonlinear solvers.
- Parallel timestepping (ODE) solvers.
- Distributed arrays for finite difference methods.

PETSc conventions

— PETSc is a large library, so it needs conventions to keep it organized.

— Methods and symbols have prefixes which relate to their category.

`Vec`	Vectors
`Mat`	Matrices
`KSP`	Krylov solvers (CG, Bi-CGStab, GMRES, . . .)
`PC`	Preconditioners (Jacobi, SOR, ILU, . . .)

— Extensive documentation:

http://www.mcs.anl.gov/petsc/petsc-current/docs/manualpages/singleindex.html

PETSc Vectors

— A vector is stored in an opaque data structure called a `Vec`:

```
// Look at petscvec.h
typedef struct _p_Vec*          Vec;
```

— Vectors can have different types, such as

seq	Sequential
mpi	Distributed
pthread	Threaded
cusp	CUDA (Nividia GPU) format through CUSP

— The type and its implementation is hidden from the user: it is an implementation detail.

— There is *no* direct data access: similar to MPI access to data through functions.

— This helps to abstract away implementation details for different types.

PETSc Vectors: first steps

Declaration of a vector: a pointer to the actual implemented type

```
// PETSc Vec pointer
Vec x;
// Local size
PetscInt n;
```

Creation of a sequential vector:

```
VecCreate(PETSC_COMM_SELF, &x);
VecSetSizes(x, PETSC_DECIDE, n);
VecSetFromOptions(x);
```

Creation of a MPI vector, the local size only is specified:

```
VecCreateMPI(PETSC_COMM_WORLD, n, PETSC_DETERMINE,
             &x);
```

PETSc Vectors: first steps

Any operation is performed through functions:

```
// Set all entries to zero
PetscScalar a = 0.0;
VecSet(x, a);

// Get global size
PetscInt N;
VecGetSize(x, &N);
```

PETSc provides function to display the content:

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

PETSc object must be deallocated to reclaim the memory:

```
VecDestroy(&x);
```

PETSc Vectors: some predefined operations

```
PetscScalar a; Vec x; Vec y; Vec w;

VecScale(x, a);

VecAXPY(x, a, y);

VecDot(x, y, &a);

VecPointwiseMult(w, x, y);

VecMin(x, PETSC_NULL, &a);
VecMax(x, PETSC_NULL, &a);

VecNorm(x, NORM_2, &a);
```

PETSc Vectors: setting values

Two types of operations (example of encapsulation):

1. Insertion: only last value is retained

```
void PETScVector::set(const real* block, int m,
                      int* rows)
{
  dolfin_assert(x);
  VecSetValues(x, m, rows, block, INSERT_VALUES);
}
```

2. Addition: all values are summed

```
void PETScVector::add(const real* block, int m,
                      int* rows)
{
  dolfin_assert(x);
  VecSetValues(x, m, rows, block, ADD_VALUES);
}
```

PETSc Vectors: ownership

Data ownership is the crucial aspect of distributed computing:

- a vector on a process *owns* a contiguous list of entries stored in the local memory.
- the local size is the number of entries owned
- the offset is the index of the first entry
- the range is: $[offset, offset + localsize[$
- PETSc provides a function to get the range:

```
int low, high;
VecGetOwnershipRange(x, &low, &high);
```

PETSc Vectors: ghost entries

- If the problem is solved in parallel on a mesh, each process is responsible for a partition.
- Example of a domain for the simulation of a Turbulent Jet in a cylinder.
- Degrees of freedom on the inter-process (interior) boundary are shared between processes.
- Only one owner, ghosted on other processes.

Ghosted entries are managed with: `VecCreateGhost`

First step is to get the ownership range:

```
int local_size, size, low, high;
VecGetSize(x, &size);
VecGetLocalSize(x, &local_size);
VecGetOwnershipRange(x, &low, &high);
```

PETSc Vectors: ghost entries

Then build a list of indices `ghost_indices` present on the process but outside the range.

Ghost entries can then be specified:

```
VecCreateGhost(MPI::DOLFIN_COMM, local_size, size,
               num_ghost_indices, &ghost_indices[0], &
```

Do not forget to synchronize the ghosts!

```
VecGhostUpdateBegin(x, INSERT_VALUES, SCATTER_FORWARD)
VecGhostUpdateEnd(x, INSERT_VALUES, SCATTER_FORWARD);
```

PETSc Matrices

— A matrix is stored in an opaque data structure called a `Mat`:

```
// Look at petscmat.h
typedef struct _p_Mat*                Mat;
```

— Matrices can also have different types depending on structure,

MATDENSE	"dense"	dense matrices
MATAIJ	"aij"	sparse matrices
MATBAIJ	"baij"	block sparse matrices
MATSBAIJ	"sbaij"	symmetric block sparse matrices

— as well as specialized implementations (sequential, MPI, CUDA):

- seqdense: Sequential (normal) dense
- seqaij: Sequential (normal) sparse
- mpiaij: Distributed sparse
- aijcusp: CUDA (Nvidia GPU) sparse

PETSc Matrices

Example for a sequential matrix:

```
Mat A;
PetscInt M;
PetscInt N;

// Distributed with 50 non-zero entries per-row
MatCreateSeqAIJ(PETSC_COMM_SELF, M, N, 50, PETSC_NULL
                &A);
```

Example for an MPI matrix:

```
// Distributed with 120 non-zero entries per-row
// in DIAGONAL and OFF-DIAGONAL
MatCreateAIJ(PETSC_COMM_WORLD, PETSC_DECIDE,
             PETSC_DECIDE, M, N, 120, PETSC_NULL,
             120, PETSC_NULL, &A);
```

For distributed matrices, DIAGONAL and OFF-DIAGONAL block are the natural extension of vector range to two dimensions.

Declare the sparsity pattern

- Why is this important?
- PETSc uses a popular sparse matrix data structure called *compressed row storage* (CRS).
- Each nonzero element in the matrix is stored along with its column index in a single, one-dimensional array.
- Another array designates the start of each row.

Compressed row storage: example

$$\boldsymbol{A} = \begin{pmatrix} 1 & 0 & 0 & 2 \\ 0 & 0 & 4 & 0 \\ 0 & 5 & 0 & 6 \\ 7 & 0 & 0 & 0 \end{pmatrix}$$

```
vals = [1, 2, 4, 5, 6, 7]
cols = [0, 3, 2, 1, 3, 0]
rowptrs = [0, 2, 3, 5]
```

Compressed row storage

- The problem with CRS is that inserting a new nonzero element somewhere in the matrix may involve a lot of data shifting.
- Other sparse structures exist that are optimized for insertion but not for matrix-vector operations.
- Various workarounds exist, such as assembling the final CSR structure after all elements have been set, or (in this case) pre-declare the sparsity pattern before inserting values.
- It is necessary to know how many nonzero elements are on each row. For MPI applications, it is also good to know how many nonzero elements are off or on the diagonal.

PETSc Matrices: specifying the sparsity pattern

To avoid overallocation of memory the structured should be initialized exactly for non-zero entries.

Creation with specification of sparsity pattern:

```
// Specify non-zero entries for each row on DIAGONAL
MatCreateAIJ(MPI::DOLFIN_COMM,
    M, N, PETSC_DETERMINE, PETSC_DETERMINE,
    PETSC_DETERMINE, (PetscInt*) d_nzrow,
    PETSC_DETERMINE, (PetscInt*) o_nzrow, &A);
```

This is a two-dimensional extension of the vector range + ghosts:

1. DIAGONAL: the block owned by the current process
2. OFF-DIAGONAL: entries on columns outside the column range

Poisson solver in PETSc

- As is common with C libraries like this, much of our code will be initialization.
- PETSc has tools for finite difference methods, but we will avoid them.
- We focus here only on the setup of the vector, the matrix and the solution of the linear system.

Problem

— Solve

$$\mathrm{A}\boldsymbol{x} = \boldsymbol{b}, \qquad \boldsymbol{b}, \boldsymbol{x} \in \mathbb{R}^N, \qquad \mathrm{A} \in M_N(\mathbb{R})$$

where A can be the system resulting from discretizing a Poisson problem using finite differences.

— We use standard notation for matrices and vectors, i.e.

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,N} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N,1} & a_{N,2} & \cdots & a_{N,N} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_N \end{pmatrix}$$

Vector setup

```
Vec b;

// Use PETSC_COMM_SELF to get a seq vector
VecCreate(PETSC_COMM_WORLD, &b);

// Local and global sizes
VecSetSizes(b, PETSC_DECIDE, m*m);

double hh = 1.0 / n / n;
for (size_t j = 0; j < m*m; j++)
  VecSetValue(b, j, hh, INSERT_VALUES);
```

Matrix setup

```
Mat A;
MatCreate(PETSC_COMM_WORLD, &A);
MatSetType(A, MATSEQAIJ);
MatSetSizes(A, PETSC_DECIDE, PETSC_DECIDE, m*m, m*m);

// Diagonal
for (size_t i = 0; i < m*m, i++)
  MatSetValue(A, i, i, 4.0, INSERT_VALUES);

// L-R coupling
for (size_t i = 0; i < m*m - 1, i++) {
  if (i % m != m-1)
    MatSetValue(A, i, i+1, -1.0, INSERT_VALUES);
  if (i % m)
    MatSetValue(A, i, i-1, -1.0, INSERT_VALUES);
}

// U-D coupling
for (size_t i = m; i < m*m, i++) {
  MatSetValue(A, i, i-m, -1.0, INSERT_VALUES);
  MatSetValue(A, i-m, i, -1.0, INSERT_VALUES);
}
```

Synchronization

The vector and matrix must be *assembled* before we can use them. This might involve communication.

```
VecAssemblyBegin(b);
VecAssemblyEnd(b);
MatAssemblyBegin(A, MAT_FINAL_ASSEMBLY);
MatAssemblyEnd(A, MAT_FINAL_ASSEMBLY);
```

Solver setup

```
KSP sol;
KSPCreate(PETSC_COMM_WORLD, &sol);

KSPSetType(ksp, "cg");
KSPSetTolerances(ksp, 1e-10, 1e-10, 1e6, 10000);
KSPSetOperators(ksp, A, A);

PC pc;
KSPGetPC(ksp, &pc);
PCSetType(pc, "ilu");
PCSetFromOptions(pc);
PCSetUp(pc);

KSPSetFromOptions(ksp);
KSPSetUp(ksp);
```

Solving

— To solve,

```
Vec x;
VecDuplicate(b, &x);
KSPSolve(ksp, b, x);
```

— You will probably find that the solver performs rather poorly, both in serial and parallel.

— Reasons for this include

1. We fill the vector element by element,
2. We fill the matrix element by element,
3. We haven't declared the sparsity pattern of the matrix, and
4. All processes fill all elements.

Fill the whole vector at once

```
PetscInt low, high;
VecGetOwnershipRange(b, &low, &high);

PetscInt indices[high-low];
double vals[high-low];
for (size_t i = 0; i < high - 1; i++) {
  inds[i] = low + i;
  vals[i] = hh;
}
VecSetValues(b, high-low, inds, vals, INSERT_VALUES
free(inds); free(vals);
```

Similar but more involved for the matrix.

Declare the sparsity pattern

```
PetscInt first, last;
MatGetOwnershipRange(A, &first, &last);
PetscInt d_nz = (PetscInt *)
  malloc((last - first) * sizeof(PetscInt));
PetscInt o_nz = (PetscInt *)
  malloc((last - first) * sizeof(PetscInt));

for (size_t i = first; i < last; i++) {
  d_nz[i - first] = 5;
  o_nz[i - first] = 5;
}
MatMPIAIJSetPreallocation(
  A, PETSC_DEFAULT, d_nz, PETSC_DEFAULT, o_nz
);
```

Here we have slightly overallocated for the sake of simplicity.

Fully declare the sparsity pattern

— We can also pin the sparsity pattern completely.

```
PetscInt d_nz = (PetscInt *) malloc(m*m * sizeof(PetscInt));
int total = 0;
// count number of nonzeros per row -> d_nz and total

MatSeqAIJSetPreallocation(A, PETSC_DEFAULT, d_nz);

PetscInt col = (PetscInt *) malloc(total * sizeof(PetscInt));
// compute the actual column indices
MatSeqAIJSetColumnIndices(A, col);
```

— This way the matrix format will never change when adding values, which allows for multi-threaded assembly.

— Unfortunately it doesn't work in hybrid mode. (There is no `MatMPIAIJSetColumnIndices`.)

Fill only your own elements

- We can use `VecGetOwnershipRange` and `MatGetOwnershipRange` to get the global indices that are assigned to our own process.
- Then, each process can set just those indices.
- Note that in PETSc, a process owns whole *rows* of the matrix, not columns as we have been (often) using in this course.