Cineca
TRAINING
High Performance
Computing 2017

Domain specific libraries for PDEs

Simone Bnà – simone.bna@cineca.it
SuperComputing Applications and Innovation Department

CINECA

Outline

- Introduction to Sparse Matrix algebra
- The PETSc toolkit
- Sparse Matrix Computation with PETSc
- Profiling and preliminary tests on KNL

Introduction to Sparse matrix algebra

Definition of a Sparse Matrix and a Dense Matrix

- A **sparse matrix** is a matrix in which the number of non-zeroes entries is O(n) (The average number of non-zeroes entries in each row is bounded independently from n)

$$\begin{pmatrix} 1.0 & 0 & 5.0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3.0 & 0 & 0 & 0 & 0 & 11.0 & 0 \\ 0 & 0 & 0 & 0 & 9.0 & 0 & 0 & 0 \\ 0 & 0 & 6.0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 7.0 & 0 & 0 & 0 & 0 \\ 2.0 & 0 & 0 & 0 & 0 & 10.0 & 0 & 0 \\ 0 & 0 & 0 & 8.0 & 0 & 0 & 0 & 0 \\ 0 & 4.0 & 0 & 0 & 0 & 0 & 0 & 12.0 \end{pmatrix}$$

- A **dense matrix** is a non-sparse matrix (The number of non-zeroes elements is $O(n^2)$)

$$\begin{pmatrix} 1.0 & 3.4 & 5.0 & 7.5 & 2.3 & 0 & 2.1 & 8.5 \\ 6.5 & 3.5 & 0 & 5.4 & 1.0 & 1 & 0 & 2.1 \\ 0 & 2.8 & 5.7 & 9.2 & 1.1 & 3 & 0 & 2.4 \\ 3.4 & 5.4 & 0 & 4.3 & 3.4 & 2.1 & 1.1 & 4.3 \\ 8.6 & 5.8 & 2.1 & 2.2 & 3.1 & 5.5 & 3.4 & 2.3 \\ 5.4 & 6.7 & 9.8 & 2.1 & 3.4 & 4.3 & 2.1 & 3.5 \\ 4.3 & 3.4 & 1.2 & 5.4 & 0.2 & 3.2 & 0.8 & 1.2 \\ 3.2 & 0 & 1.3 & 4.5 & 0.7 & 9.8 & 0.3 & 1.2 \end{pmatrix}$$

Sparsity and Density

- The **sparsity** of a matrix is defined as the number of zero-valued elements divided by the total number of elements (m x n for an m x n matrix)
- The **density** of a matrix is defined as the complementary of the sparsity: density = 1 – sparsity
- For Sparse matrices the **sparsity** is ≈ 1 and the **density** is << 1

Example:

m = 8 nnzeros = 12

n = 8 nzeros = m*n – nnzeros

sparsity = 64 – 12 / 64 = 0.8125

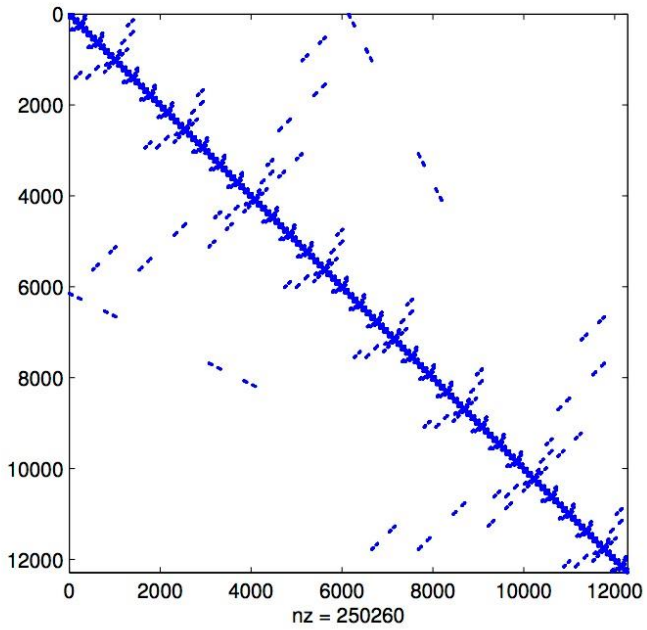
density = 1 – 0.8125 = 0.1875

$$\begin{pmatrix} 1.0 & 0 & 5.0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3.0 & 0 & 0 & 0 & 0 & 11.0 & 0 \\ 0 & 0 & 0 & 0 & 9.0 & 0 & 0 & 0 \\ 0 & 0 & 6.0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 7.0 & 0 & 0 & 0 & 0 \\ 2.0 & 0 & 0 & 0 & 0 & 10.0 & 0 & 0 \\ 0 & 0 & 0 & 8.0 & 0 & 0 & 0 & 0 \\ 0 & 4.0 & 0 & 0 & 0 & 0 & 0 & 12.0 \end{pmatrix}$$

Sparsity pattern

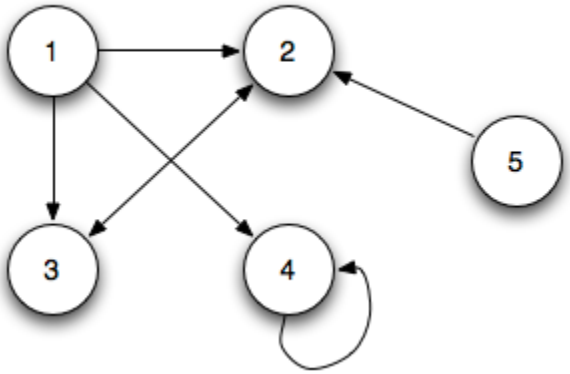
- The distribution of non-zero elements of a sparse matrix can be described by the **sparsity pattern**, which is defined as the set of entries of the matrix different from zero. In symbols:

$$\{ (i, j) : Aij \neq 0 \}$$



Sparsity pattern

- The sparsity pattern can be represented also as a **Graph**, where nodes *i* and *j* are connected by an edge if and only if $A_{ij} \neq 0$
- In a Sparse Matrix the **degree of a vertex** in the graph is <<relatively low>>
- Conceptually, sparsity corresponds to a system loosely coupled



	1	2	3	4	5
1	0	1	1	1	0
2	0	0	1	0	0
3	0	1	0	0	0
4	0	0	0	1	0
5	0	1	0	0	0

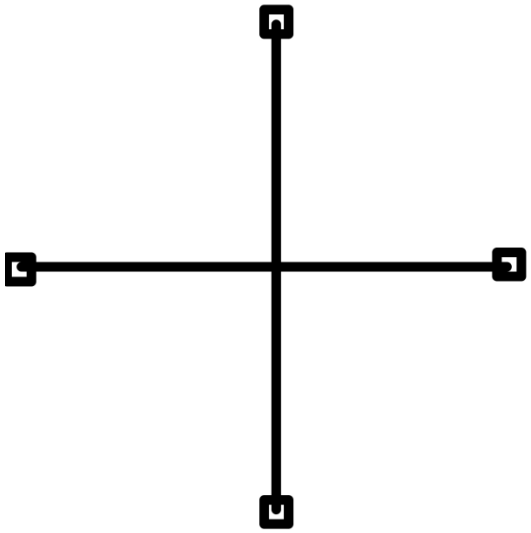
Jacobian of a PDE

- Matrices are used to store the Jacobian of a PDE.
- The following discretizations generates a sparse matrix
  - Finite difference
  - Finite volume
  - Finite element method (FEM)
- Different discretization can lead to a Dense linear matrix:
  - Spectral element method (SEM)

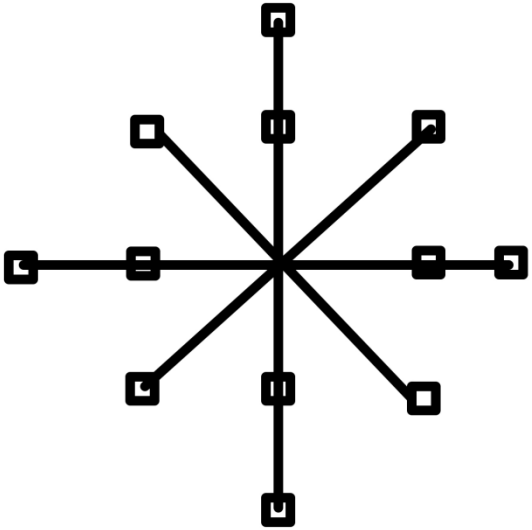
Sparsity pattern in Finite Difference

- The sparsity pattern in finite difference depends on the topology of the adopted computational grid (e.g. cartesian grid), the indexing of the nodes and the type of stencil

Star stencil

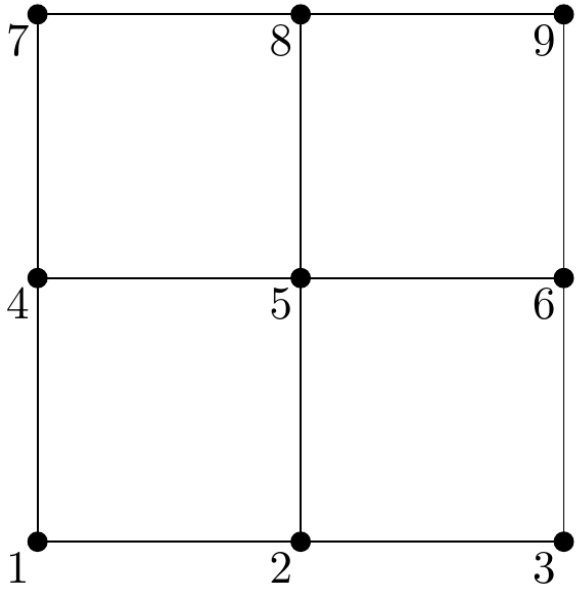


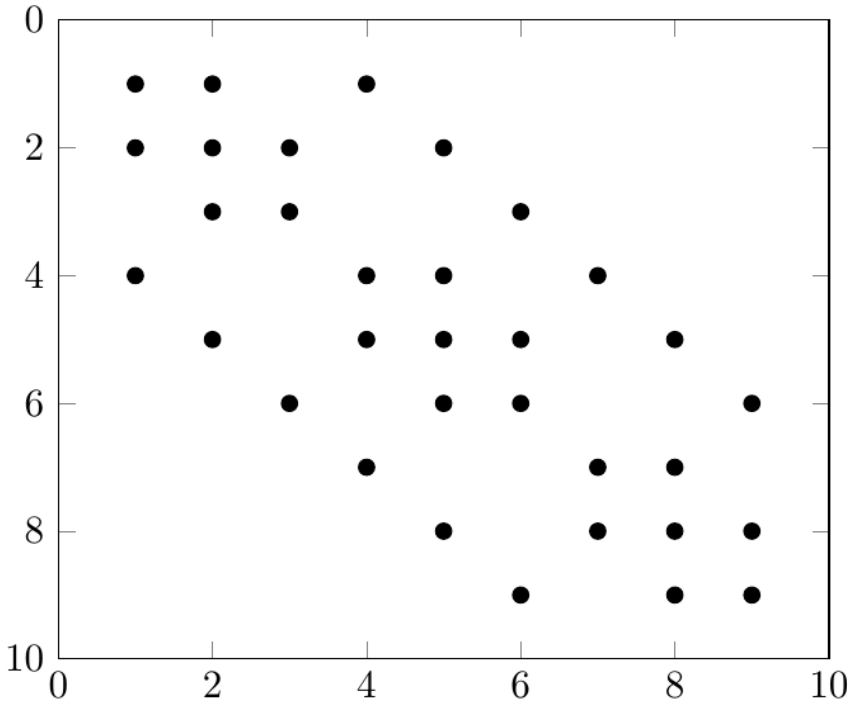
Box stencil



Sparsity pattern in Finite Difference

- The sparsity pattern in finite difference depends on the topology of the adopted computational grid (e.g. cartesian grid), the indexing of the nodes and the type of stencil



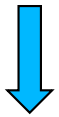


Sparsity pattern in Finite Element

- The sparsity pattern depends on the topology of the adopted computational grid (e.g. unstructured grid), the kind of the finite element (e.g. Taylor-Hood, Crouzeix-Raviart, Raviart-Thomas, Mini-Element,…) and on the indexing of the nodes.
- In Finite-Element discretizations, the sparsity of the matrix is a direct consequence of the small-support property of the finite element basis
- Finite Volume can be seen as a special case of Finite Element

Don’t reinvent the wheel!

- The use of storage techniques for sparse matrices is fundamental, in particular for large-scale problems
- Standard dense-matrix structures and algorithms are slow and ineffcient when applied to large sparse matrices
- There are some available tools to work with Sparse matrices that uses specialised algorithms and data structures to take advantage of the sparse structure of the matrix



- **The PETSc toolkit** (http://www.mcs.anl.gov/petsc/)
- The TRILINOS project (https://trilinos.org/)

The PETSc toolkit

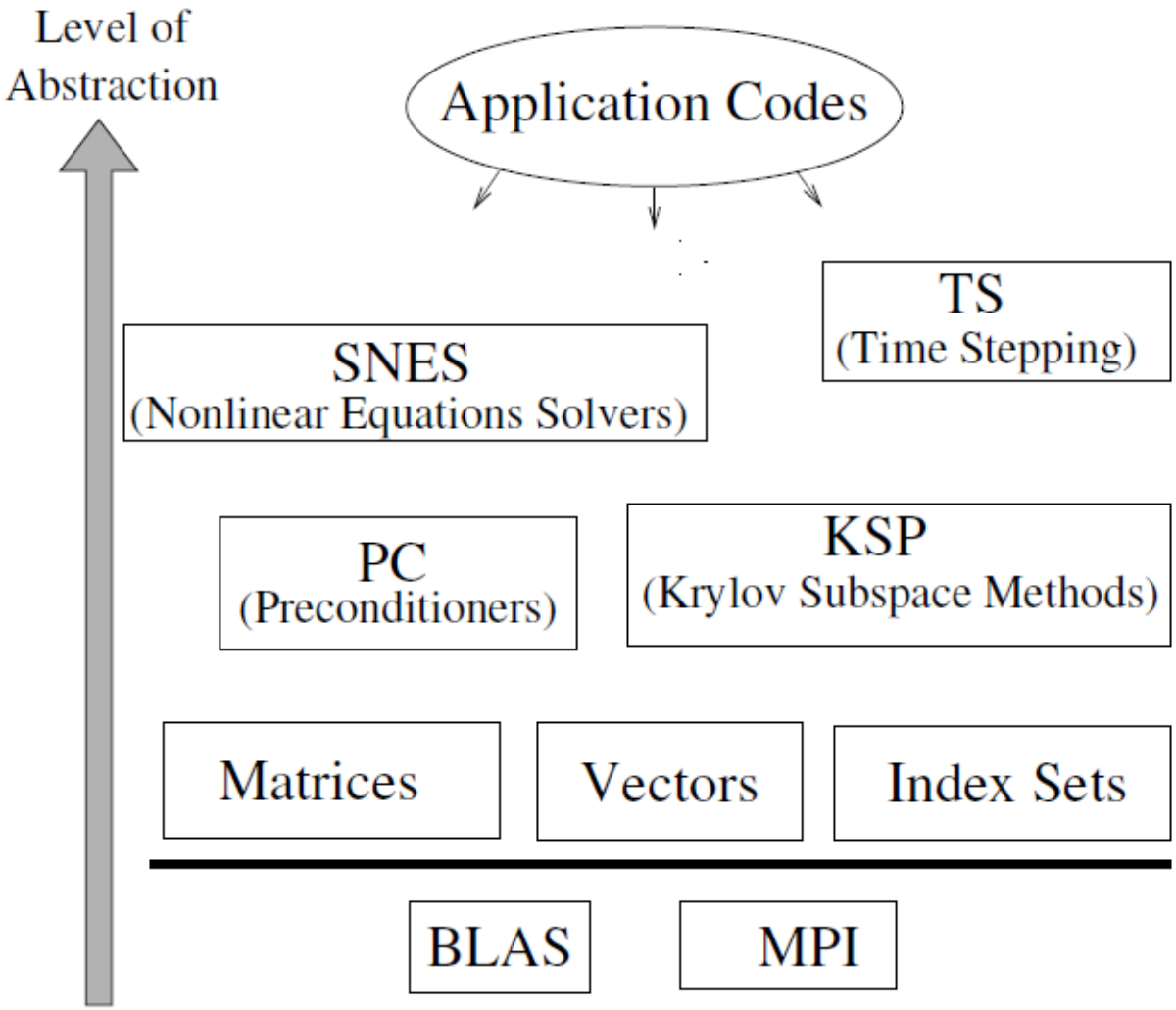
PETSc in a nutshell

PETSc – Portable, Extensible Toolkit for Scientific Computation

Is a suite of data structures and routines for the scalable (parallel) solution of scientific applications mainly modelled by partial differential equations.

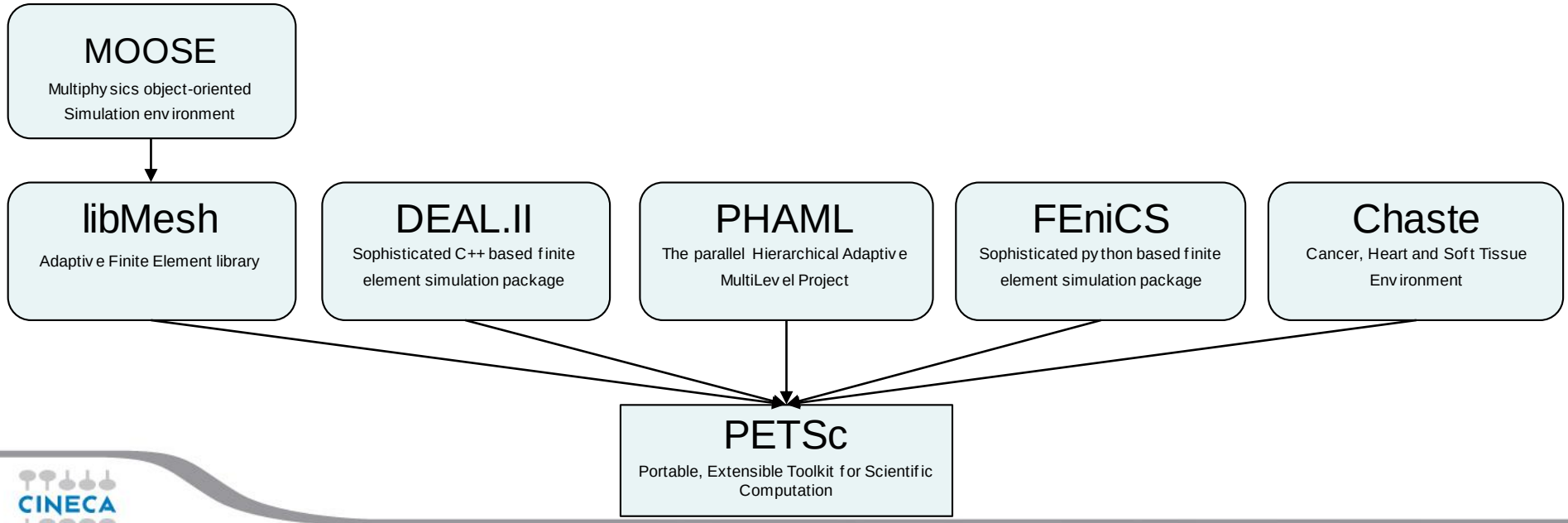
- Tools for distributed vectors and matrices
- Linear system solvers (sparse/dense, iterative/direct)
- Non linear system solvers
- Serial and parallel computation
- Support for Finite Difference and Finite Elements PDE discretizations
- Structured and Unstructured topologies
- Support for debugging, profiling and graphical output

PETSc class hierarchy



Frameworks built on top of Petsc

- PETSc is a toolkit, not a framework
- PETSc is PDE oriented, but not specific to any kind of PDE
- Alternatives:
  - FEM packages: MOOSE, libMesh, DEAL.II, FEniCS
  - Solvers for classes of problems: CHASTE



PETSc numerical components

Parallel Numerical Components of PETSc

Nonlinear Solvers		
Newton–based Methods		Other
Line Search	Trust Region	

Time Steppers			
General Linear	IMEX	Pseudo–Time Stepping	Runge–Kutt

Krylov Subspace Methods							
GMRES	CG	CGS	Bi–CG–Stab	TFQMR	Richardson	Chebychev	Other

Preconditioners						
Additive Schwarz	Block Jacobi	Jacobi	ILU	ICC	LU (sequential only)	Other

Matrices				
Compressed Sparse Row (AIJ)	Block Compressed Sparse Row (BAIJ)	Symmetric Block Compressed Row (SBAIJ)	Dense	Other

Vectors

Index Sets			
Indices	Block Indices	Stride	Other

External Packages

- Dense linear algebra: Scalapack, Plapack
- Sparse direct linear solvers: Mumps, SuperLU, SuperLU_dist
- Grid partitioning software: Metis, ParMetis, Jostle, Chaco, Party
- ODE solvers: PVODE
- Eigenvalue solvers (including SVD): SLEPc
- Optimization: TAO

PETSc design concepts

Goals

- Portability: available on many platforms, basically anything that has MPI
- Performance
- Scalable parallelism
- Flexibility: easy switch among different implementations

Approach

- Object Oriented Delegation Pattern : many specific implementations of the same object
- Shared interface (overloading):
  MatMult(A,x,y); // y <- A x
  same code for sequential, parallel, dense, sparse
- Command line customization

Drawback

- Nasty details of the implementation hidden

PETSc and Parallelism

- PETSc is layered on top of **MPI**: you do not need to know much MPI when you use PETSc
- All objects in PETSc are defined on a communicator; they can only interact if on the same communicator
- Parallelism through MPI (**Pure MPI programming model**). Limited support for use with the hybrid MPI-thread model.
  - PETSc supports to have individual threads (OpenMP or others) to each manage their own (sequential) PETSc objects (and each thread can interact only with its own objects).
  - No support for threaded code that made Petsc calls (OpenMP, Pthreads) since PETSc is not «thread-safe».
- Transparent: same code works sequential and parallel.

Sparse Matrix computation with PETSc

Vectors

What are PETSc vectors?

- Represent elements of a vector space over a field (e.g. R^n)
- Usually they store field solutions and right-hand sides of PDE
- Vector elements are **PetscScalars** (there are no vectors of integers)
- Each process locally owns a subvector of contiguously numbered global indices

Features

- Vector types: **STANDARD** (**SEQ** on one process and **MPI** on several), **VIENNACL**, **CUSP**…
- Supports all vector space operations
  - `VecDot(),VecNorm(),VecScale(), …`
- Also unusual ops, like e.g. `VecSqrt(),VecReciprocal()`
- Hidden communication of vector values during assembly
- Communications between different parallel vectors

Numerical vector operations

Function Name	Operation
VecAXPY(Vec y,PetscScalar a,Vec x);	$y = y + a * x$
VecAYPX(Vec y,PetscScalar a,Vec x);	$y = x + a * y$
VecWAXPY(Vec w,PetscScalar a,Vec x,Vec y);	$w = a * x + y$
VecAXPBY(Vec y,PetscScalar a,PetscScalar b,Vec x);	$y = a * x + b * y$
VecScale(Vec x, PetscScalar a);	$x = a * x$
VecDot(Vec x, Vec y, PetscScalar *r);	$r = \bar{x}' * y$
VecTDot(Vec x, Vec y, PetscScalar *r);	$r = x' * y$
VecNorm(Vec x,NormType type, PetscReal *r);	$r = \|\|x\|\|_{type}$
VecSum(Vec x, PetscScalar *r);	$r = \sum x_i$
VecCopy(Vec x, Vec y);	$y = x$
VecSwap(Vec x, Vec y);	$y = x$ while $x = y$
VecPointwiseMult(Vec w,Vec x,Vec y);	$w_i = x_i * y_i$
VecPointwiseDivide(Vec w,Vec x,Vec y);	$w_i = x_i / y_i$
VecMDot(Vec x,int n,Vec y[],PetscScalar *r);	$r[i] = \bar{x}' * y[i]$
VecMTDot(Vec x,int n,Vec y[],PetscScalar *r);	$r[i] = x' * y[i]$
VecMAXPY(Vec y,int n, PetscScalar *a, Vec x[]);	$y = y + \sum_i a_i * x[i]$
VecMax(Vec x, int *idx, PetscReal *r);	$r = \max x_i$
VecMin(Vec x, int *idx, PetscReal *r);	$r = \min x_i$
VecAbs(Vec x);	$x_i = \|x_i\|$
VecReciprocal(Vec x);	$x_i = 1/x_i$
VecShift(Vec x,PetscScalar s);	$x_i = s + x_i$
VecSet(Vec x,PetscScalar alpha);	$x_i = \alpha$

Matrices

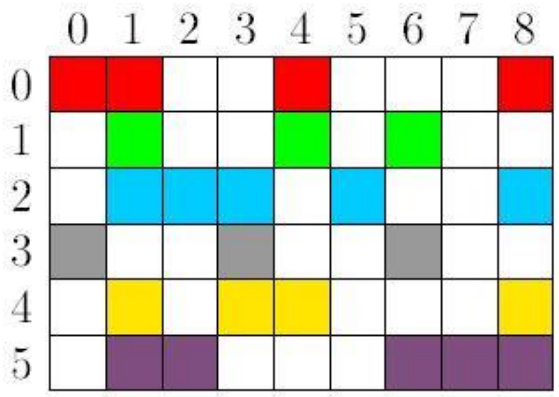
What are PETSc matrices?

- Roughly represent linear operators that belong to the dual of a vector space over a field (e.g. R^n)
- In most of the PETSc low-level implementations, each process logically owns a submatrix of contiguous rows

Features

- Supports many storage formats
  - AIJ, BAIJ, SBAIJ, DENSE, VIENNACL, CUSP (on GPU) ...
- Data structures for many external packages
  - MUMPS (parallel), SuperLU_dist (parallel), SuperLU, UMFPack
- Hidden communications in parallel matrix assembly
- Matrix operations are defined from a common interface
- Shell matrices via user defined MatMult and other ops

Matrices

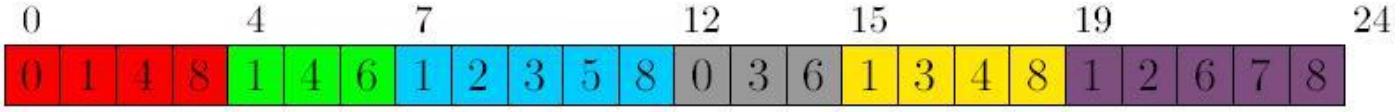


The default matrix representation within PETSc is the general sparse **AIJ format** (Yale sparse matrix or Compressed Sparse Row, CSR)

- The nonzero elements are stored by rows
- Array of corresponding column numbers
- Array of pointers to the beginning of each row

value

index



row pointer

Matrix memory preallocation

- PETSc matrix creation is very flexible: No preset sparsity pattern
- Memory **preallocation** is critical for achieving **good performance** during matrix assembly, as this reduces the number of allocations and copies required during the assembling process. Remember: malloc is very expensive (run your code with -memory_info, -malloc_log)
- Private representations of PETSc sparse matrices are dynamic data structures: **additional nonzeros can be freely added** (if no preallocation has been explicitly provided).
- No preset sparsity pattern, any processor can set any element: potential for lots of malloc calls
- Dynamically adding many nonzeros
  - requires additional memory allocations
  - requires copies

  → **kills performances!**

Preallocation of a parallel sparse matrix

Each process **logically owns** a matrix subset of contiguously numbered global rows. Each subset consists of two sequential matrices corresponding to diagonal and off-diagonal parts.

P0

P1

P2

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

Process 0

```
dnz=2, onz=2
dnnz[0]=2, onnz[0]=2
dnnz[1]=2, onnz[1]=2
dnnz[2]=2, onnz[2]=2
```

Process 1

```
dnz=3, onz=2
dnnz[0]=3, onnz[0]=2
dnnz[1]=3, onnz[1]=1
dnnz[2]=2, onnz[2]=1
```

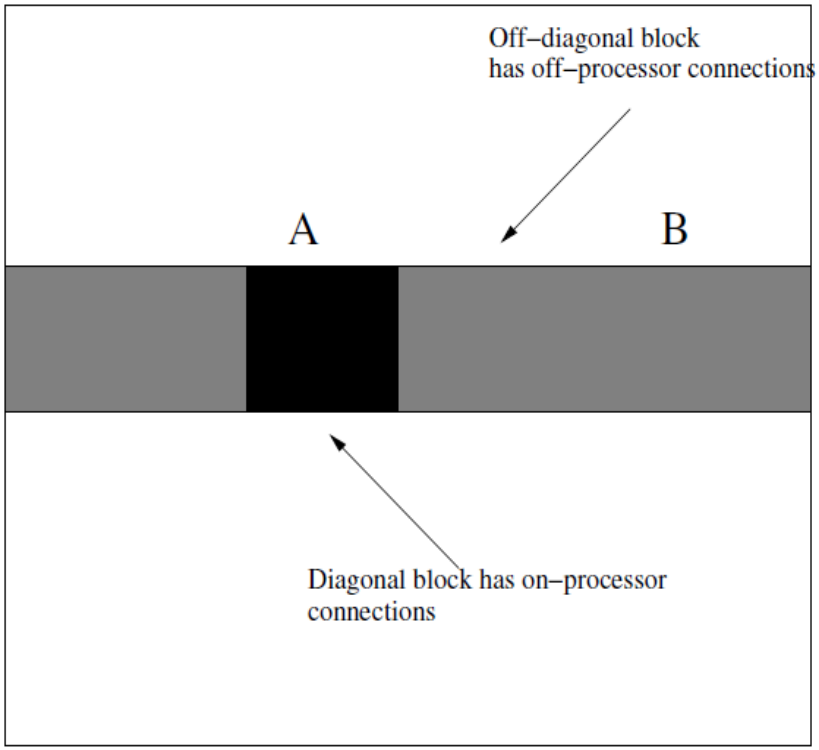
Process 2

```
dnz=1, onz=4
dnnz[0]=1, onnz[0]=4
dnnz[1]=1, onnz[1]=4
```

Numerical Matrix Operations

Function Name	Operation
MatAXPY(Mat Y, PetscScalar a,Mat X,MatStructure);	$Y = Y + a * X$
MatMult(Mat A,Vec x, Vec y);	$y = A * x$
MatMultAdd(Mat A,Vec x, Vec y,Vec z);	$z = y + A * x$
MatMultTranspose(Mat A,Vec x, Vec y);	$y = A^T * x$
MatMultTransposeAdd(Mat A,Vec x, Vec y,Vec z);	$z = y + A^T * x$
MatNorm(Mat A,NormType type, double *r);	$r = \|\|A\|\|_{type}$
MatDiagonalScale(Mat A,Vec l,Vec r);	$A = \text{diag}(l) * A * \text{diag}(r)$
MatScale(Mat A,PetscScalar a);	$A = a * A$
MatConvert(Mat A,MatType type,Mat *B);	$B = A$
MatCopy(Mat A,Mat B,MatStructure);	$B = A$
MatGetDiagonal(Mat A,Vec x);	$x = \text{diag}(A)$
MatTranspose(Mat A,MatReuse,Mat* B);	$B = A^T$
MatZeroEntries(Mat A);	$A = 0$
MatShift(Mat Y,PetscScalar a);	$Y = Y + a * I$

Matrix multiplication (MatMult)



$y \leftarrow A * x_A + B * x_B$

- x_B needs to be communicated
- $A * x_A$ can be computed in the meantime

Algorithm

- Initiate asynchronous sends/receives for x_B
- compute $A * x_A$
- make sure x_B is in
- compute $B * x_B$

Due to the splitting of the matrix storage into A (diag) and B (off-diag) part, code for the sequential case can be reused.

Sparse Matrices and Linear Solvers

- Solve a linear system A x = b using the Gauss Elimination method can be very time-resource consuming
- Alternatives to direct solvers are iterative solvers
- Convergence of the succession is not always guaranteed
- Possibly much faster and less memory consuming
- Basic iteration: y <- A x executed once x iteration
- Also needed a good preconditioner: $B \approx A^{-1}$

Iterative solver basics

- **KSP** (*Krylov SPace Methods*) objects are used for solving linear systems by means of iterative methods.
- Convergence can be improved by using a suitable **PC** object (preconditoner).
- Almost all iterative methods are implemented.
- Classical iterative methods (not belonging to KSP solvers) are classified as preconditioners
- Direct solution for parallel square matrices available through external solvers (MUMPS, SuperLU_dist). Petsc provides a built-in LU serial solver.
- Many KSP options can be controlled by command line
- Tolerances, convergence and divergence reason
- Custom monitors and convergence tests

Solver Types

Method	KSPType	Options Database Name
Richardson	KSPRICHARDSON	richardson
Chebyshev	KSPCHEBYSHEV	chebyshev
Conjugate Gradient [12]	KSPCG	cg
BiConjugate Gradient	KSPBICG	bicg
Generalized Minimal Residual [16]	KSPGMRES	gmres
Flexible Generalized Minimal Residual	KSPFGMRES	fgmres
Deflated Generalized Minimal Residual	KSPDGMRES	dgmres
Generalized Conjugate Residual	KSPGCR	gcr
BiCGSTAB [19]	KSPBCGS	bcgs
Conjugate Gradient Squared [18]	KSPCGS	cgs
Transpose-Free Quasi-Minimal Residual (1) [8]	KSPTFQMR	tfqmr
Transpose-Free Quasi-Minimal Residual (2)	KSPTCQMR	tcqmr
Conjugate Residual	KSPCR	cr
Least Squares Method	KSPLSQR	lsqr
Shell for no KSP method	KSPPREONLY	preonly

Preconditioner types

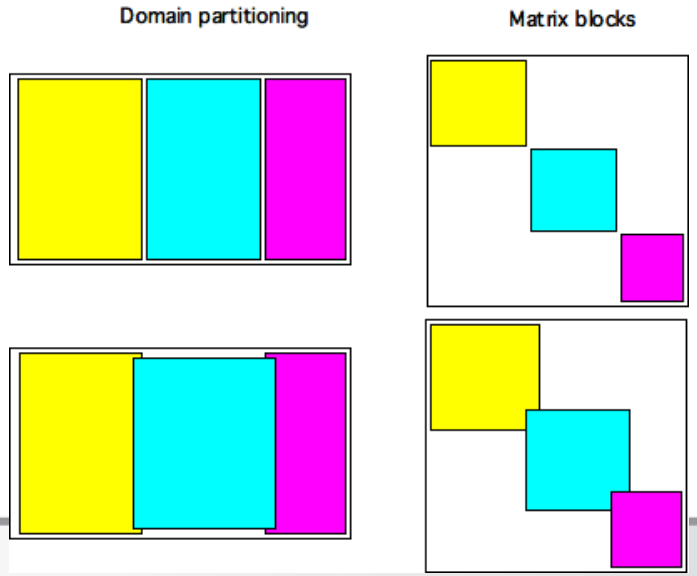
Method	PCType	Options Database Name
Jacobi	PCJACOBI	jacobi
Block Jacobi	PCBJACOBI	bjacobi
SOR (and SSOR)	PCSOR	sor
SOR with Eisenstat trick	PCEISENSTAT	eisenstat
Incomplete Cholesky	PCICC	icc
Incomplete LU	PCILU	ilu
Additive Schwarz	PCASM	asm
Algebraic Multigrid	PCGAMG	gamg
Linear solver	PCKSP	ksp
Combination of preconditioners	PCCOMPOSITE	composite
LU	PCLU	lu
Cholesky	PCCHOLESKY	cholesky
No preconditioning	PCNONE	none
Shell for user-defined PC	PCSHELL	shell

Factorization preconditioner

- Exact factorization: A = LU
- Inexact factorization: A ≈ M = L U where L, U obtained by throwing away the 'fill-in' during the factorization process (sparsity pattern of M is the same as A)
- Application of the preconditioner (that is, solve Mx = y) approx same cost as matrix-vector product y <- A x
- Factorization preconditioners are sequential
- PCICC: symmetric matrix, PCILU: nonsymmetric matrix

Parallel preconditioners

- Factorization preconditioners are sequential
- We can use them in parallel as a subpreconditioner of a parallel preconditioner as Block Jacobi or Additive Schwarz Methods (ASM)
- Each processor has its own block(s) to work with
- Block Jacobi is fully parallel, ASM requires communications between neighbours
- ASM can be more robust than Block Jacobi and have better convergence properties



Profiling and preliminary tests on KNL

Profiling and performance tuning

- Integrated profiling of:
  - time
  - floating-point performance
  - memory usage
  - communication
- User-defined events
- Profiling by stages of an application

`-log_view` - Prints an ASCII version of performance data at program's conclusion. These statistics are comprehensive and concise and require little overhead; thus, `-log_view` is intended as the primary means of monitoring the performance of PETSc codes.

Log view: Overview

	Max	Max/Min	Avg	Total
Time (sec):	9.163e+00	1.00040	9.162e+00	
Objects:	2.700e+01	1.00000	2.700e+01	
Flops:	6.364e+09	1.01926	6.286e+09	1.697e+11
Flops/sec:	6.945e+08	1.01926	6.861e+08	1.852e+10
MPI Messages:	8.537e+03	3.71335	3.845e+03	1.038e+05
MPI Message Lengths:	4.705e+07	2.07804	7.962e+03	8.266e+08
MPI Reductions:	9.950e+02	1.00000		

KSPSetUp	1	1.0	1.3081e-02	4572.2	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	0	0
KSPSolve	1	1.0	8.5259e+00	1.0	6.36e+09	1.0	1.0e+05	6.3e+03	9.8e+02	93	100	99	78	98	100	100	100	100	100	100	19907
PCSetUp	1	1.0	1.6221e-01	1.2	1.70e+08	1.0	0.0e+00	0.0e+00	0.0e+00	2	3	0	0	0	2	3	0	0	0	0	27991
PCSetUpOnBlocks	1	1.0	1.6222e-01	1.2	1.70e+08	1.0	0.0e+00	0.0e+00	0.0e+00	2	3	0	0	0	2	3	0	0	0	0	27989
PCApply	326	1.0	4.0862e+00	1.1	2.87e+09	1.0	0.0e+00	0.0e+00	0.0e+00	43	46	0	0	0	46	46	0	0	0	0	18990
MatMult	325	1.0	4.0308e+00	1.1	3.09e+09	1.0	1.0e+05	6.3e+03	0.0e+00	41	48	99	78	0	44	48	100	100	0	0	20186
MatSolve	326	1.0	4.0722e+00	1.1	2.87e+09	1.0	0.0e+00	0.0e+00	0.0e+00	43	46	0	0	0	46	46	0	0	0	0	19055
MatLUFactorNum	1	1.0	1.1603e-01	1.2	1.70e+08	1.0	0.0e+00	0.0e+00	0.0e+00	1	3	0	0	0	1	3	0	0	0	0	39132
MatILUFactorSym	1	1.0	4.3884e-02	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	0
MatGetRowIJ	1	1.0	1.0634e-02	2623.6	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	0
MatGetOrdering	1	1.0	1.1935e-02	11.3	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	0
VecTDot	650	1.0	5.2052e-01	7.5	7.68e+07	1.0	0.0e+00	0.0e+00	6.5e+02	3	1	0	0	65	4	1	0	0	67	3982	
VecNorm	326	1.0	4.0363e-01	10.7	3.85e+07	1.0	0.0e+00	0.0e+00	3.3e+02	2	1	0	0	33	3	1	0	0	33	2575	
VecCopy	2	1.0	3.4809e-04	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	0
VecSet	327	1.0	1.0189e-02	1.2	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	0
VecAXPY	650	1.0	8.4167e-02	1.3	7.68e+07	1.0	0.0e+00	0.0e+00	0.0e+00	1	1	0	0	0	1	1	0	0	0	24625	
VecAYPX	324	1.0	4.2747e-02	1.2	3.83e+07	1.0	0.0e+00	0.0e+00	0.0e+00	0	1	0	0	0	0	1	0	0	0	24168	
VecScatterBegin	325	1.0	4.2347e-02	2.9	0.00e+00	0.0	1.0e+05	6.3e+03	0.0e+00	0	0	99	78	0	0	0	100	100	0	0	
VecScatterEnd	325	1.0	4.2485e-02	2.6	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	0

Petsc benchmark: ex56 (3D linear elasticity)

- 3D, tri-linear quadrilateral (Q1), displacement finite element formulation of linear elasticity. E=1.0, nu=0.25.
- Unit box domain with Dirichlet boundary condition on the y=0 side only.
- Load of 1.0 in x + 2y direction on all nodes (not a true uniform load).
- np = number of processes; npe^{1/3} must be integer
- ne = number of elements in the x,y,z direction; (ne+1)%(npe^{1/3}) must equal zero
- Default solver: GMRES + BLOCK_JACOBI + ILU(0)

Petsc benchmark: ex56 (3D linear elasticity)

	Command	Time
Broadwell (ne=80, np=27)	mpirun -np 27 ./ex56 -ne 80 -log_view	14.2 s
KNL (ne=80, np=27) + DRAM	mpirun -np 64 numactl --membind=0,1 ./ex56 -ne 79 -log_view	38.61 s
KNL (ne=80, np=27) + MCDRAM=FLAT + NUMA=SNC2	mpirun -np 27 numactl --membind=2,3 ./ex56 -ne 80 -log_view	12.12 s
KNL (ne=79, np=64) + MCDRAM=FLAT + NUMA=SNC2	mpirun -np 64 numactl --membind=2,3 ./ex56 -ne 79 -log_view	10.90 s
KNL (ne=80, np=27) + MCDRAM=CACHE	mpirun -np 27 ./ex56 -ne 80 -log_view	14.12 s
KNL (ne=80, np=64) + MCDRAM=CACHE	mpirun -np 64 ./ex56 -ne 79 -log_view	12.50 s

Thank you for the attention