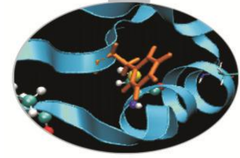
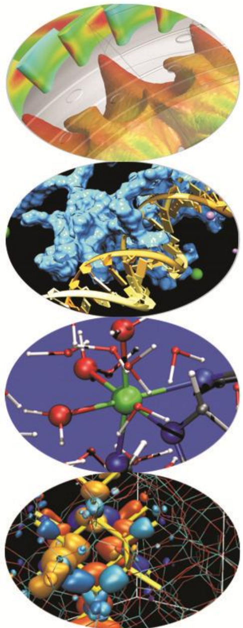
CINECA SCAI
SuperComputing Applications and Innovation



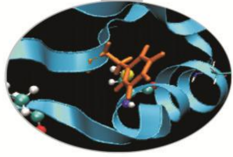


Sparse Matrix Computation with PETSc

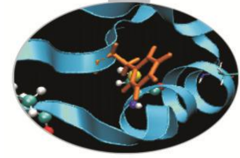
Portable, Extensible Toolkit for Scientific Computation

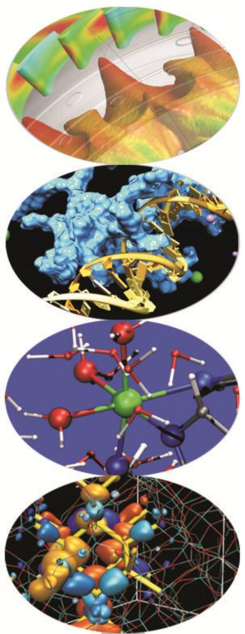
Simone Bnà – s.bn@cineca.it
SuperComputing Applications and Innovation Department

Outline

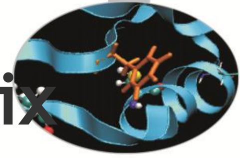


- Introduction to Sparse Matrices
- Sparse Matrix computation with PETSc
- Case studies: Engineering Applications and Domain Decomposition in HPC





Introduction to Sparse matrices



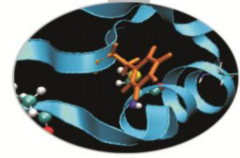
Definition of a Sparse Matrix and a Dense Matrix

- 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:

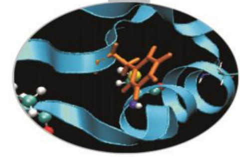
m = 8 nnzeros = 12

n = 8 nzeros = m*n – nnzeros

sparsity = 64 – 12 / 64 = 0.8125

density = 1 – 0.8125 = 0.1875

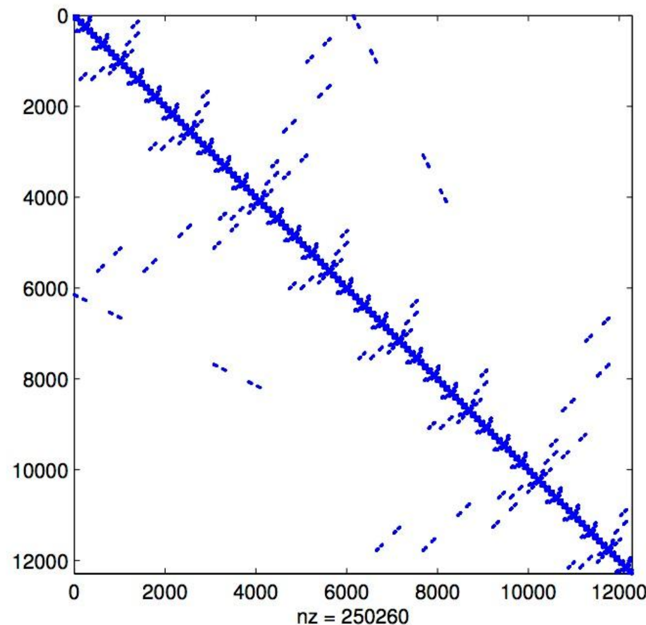
$$\begin{pmatrix} 1.0 & 0 & 5.0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3.0 & 0 & 0 & 0 & 0 & 11.0 & 0 \\ 0 & 0 & 0 & 0 & 9.0 & 0 & 0 & 0 \\ 0 & 0 & 6.0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 7.0 & 0 & 0 & 0 & 0 \\ 2.0 & 0 & 0 & 0 & 0 & 10.0 & 0 & 0 \\ 0 & 0 & 0 & 8.0 & 0 & 0 & 0 & 0 \\ 0 & 4.0 & 0 & 0 & 0 & 0 & 0 &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

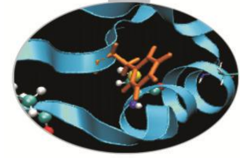


Sparsity pattern

- 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:

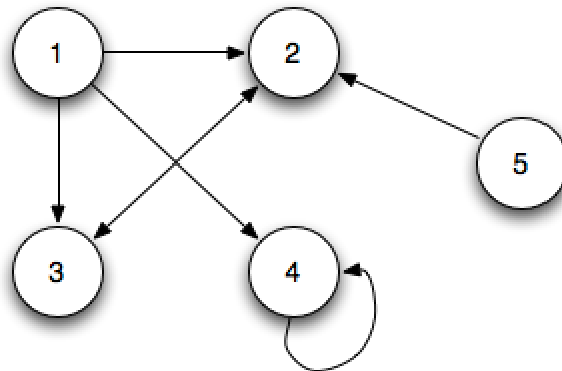
$$\{\,(i,j)\colon A_{ij} \neq 0\}$$



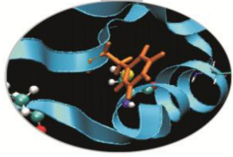


Sparsity pattern

- 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

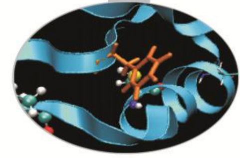


	1	2	3	4	5
1	0	1	1	1	0
2	0	0	1	0	0
3	0	1	0	0	0
4	0	0	0	1	0
5	0	1	0	0	0



Jacobian of a PDE

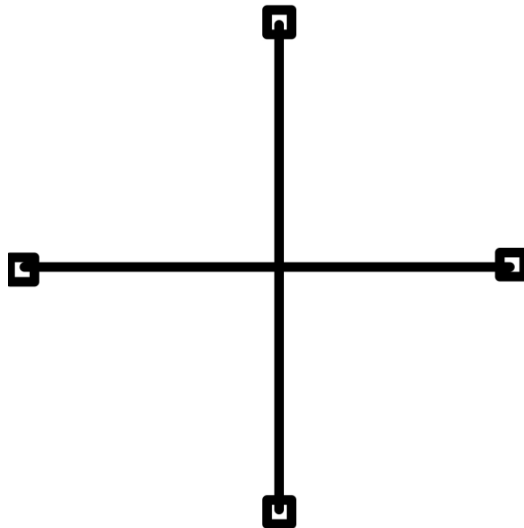
- Matrices are used to store the Jacobian of a PDE.
- The following discretizations generates a sparse matrix
  - Finite difference
  - Finite volume
  - Finite element method (FEM)
- Different discretization can lead to a Dense linear matrix:
  - Spectral element method (SEM)
  - Fast fourier transform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



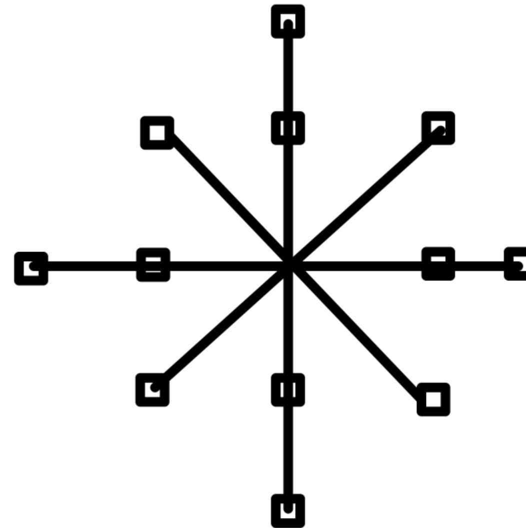
Sparsity pattern in Finite Difference

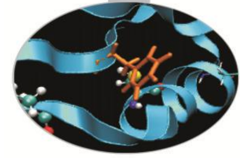
- The sparsity pattern in finite difference depends on the topology of the adopted computational grid (e.g. cartesian grid), the indexing of the nodes and the type of stencil

Star stencil



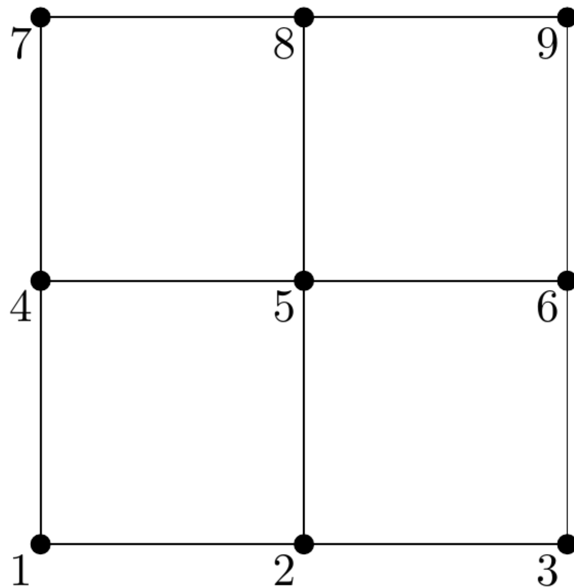
Box stencil

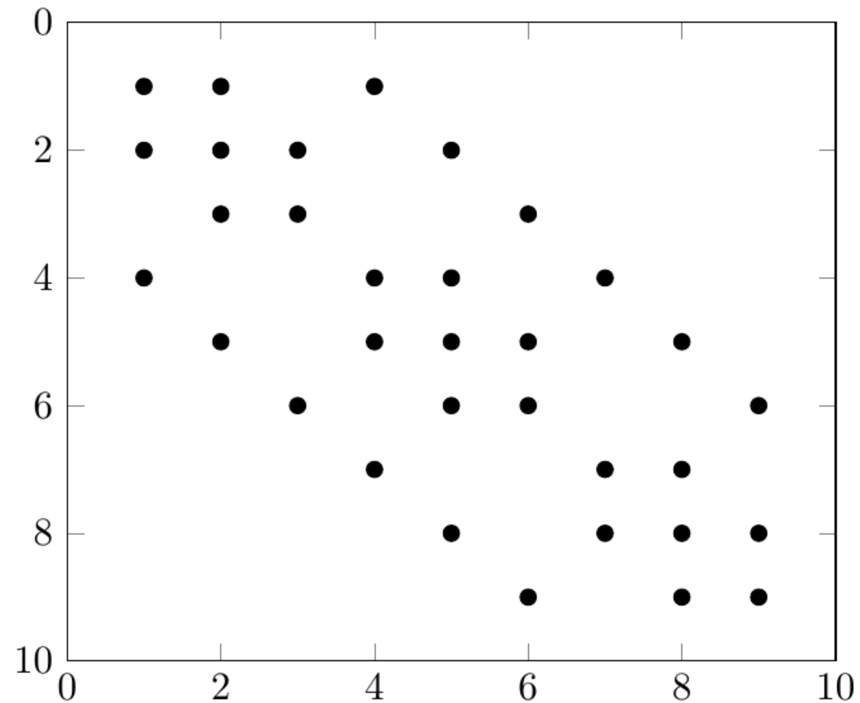


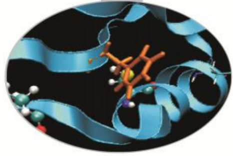


Sparsity pattern in Finite Difference

- 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



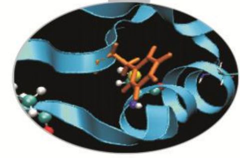




Sparsity pattern in Finite Element

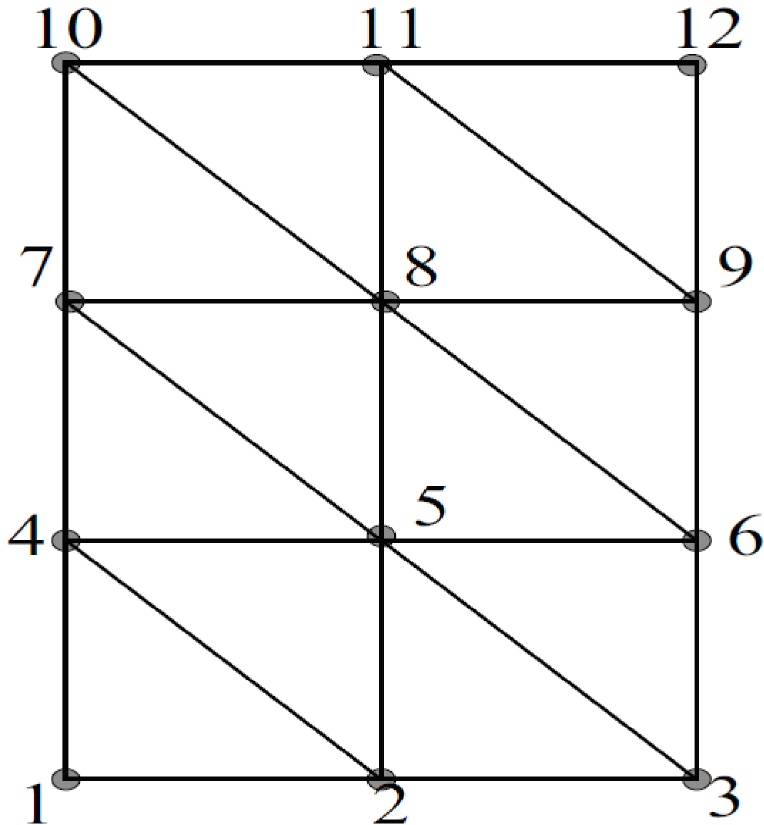
- The sparsity pattern depends on the topology of the adopted computational grid (e.g. unstructured grid), the kind of the finite element (e.g. Taylor-Hood, Crouzeix-Raviart, Raviart-Thomas, Mini-Element,…) and on the indexing of the nodes.
- In Finite-Element discretizations, the sparsity of the matrix 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

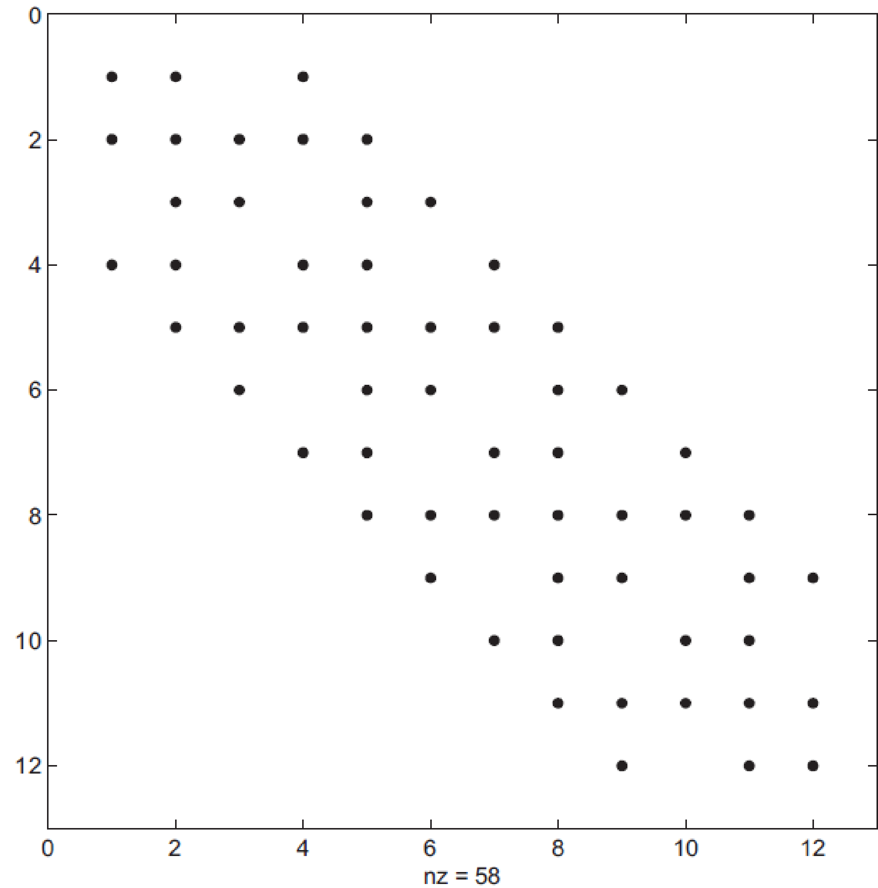
CINECA SCAI
SuperComputing Applications and Innovation

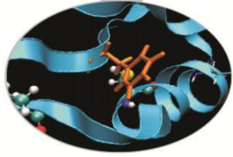


Sparsity pattern in Finite Element

$$A_{ij} = \int \psi_i \, \psi_j$$

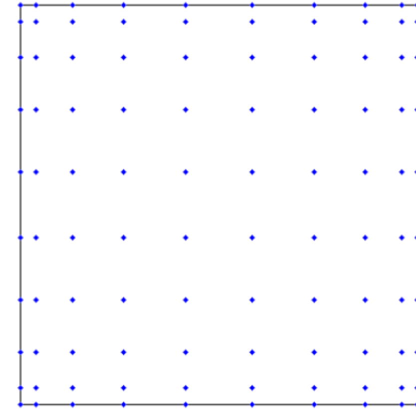




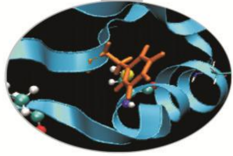


Sparsity pattern in Spectral Element Method

$$A_{ij} = \int \psi_i \psi_j$$

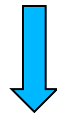


- SEM uses a tensor product space spanned by nodal basis functions associated with Gauss-Lobatto nodes
- In Spectral Element discretizations, the density of the matrix is a direct consequence of the support of the spectral element basis (Orthogonal polynomials: Legendre polynomials, Chebyshev

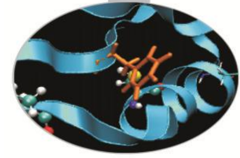


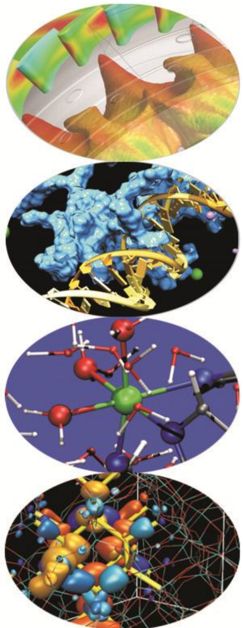
Don’t reinvent the wheel!

- 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



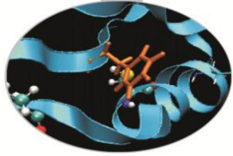
- **The PETSc toolkit** (http://www.mcs.anl.gov/petsc/)
- The TRILINOS project (https://trilinos.org/)





Sparse Matrix computation with PETSc

PETSc in a nutshell

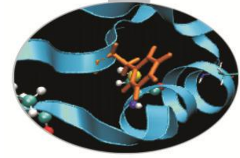


PETSc – Portable, Extensible Toolkit for Scientific Computation

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



Level of Abstraction

Application Codes

TS
(Time Stepping)

SNES
(Nonlinear Equations Solvers)

PC
(Preconditioners)

KSP
(Krylov Subspace Methods)

Matrices

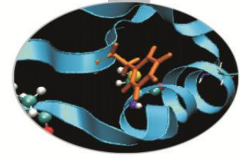
Vectors

Index Sets

BLAS

MPI

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

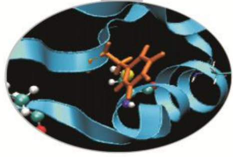
Preconditioners						
Additive Schwarz	Block Jacobi	Jacobi	ILU	ICC	LU (sequential only)	Other

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

Vectors

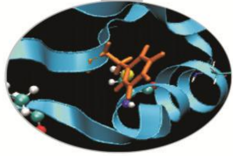
Index Sets			
Indices	Block Indices	Stride	Other

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

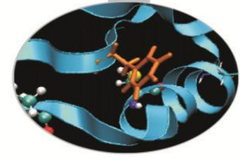
Approach

- 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

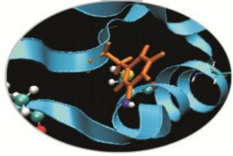
- Nasty details of the implementation hidden

PETSc and Parallelism



- All objects in PETSc are defined on a communicator; they can only interact if on the same communicator
- PETSc is layered on top of **MPI**: you do not need to know much MPI when you use PETSc
- 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.

Matrices



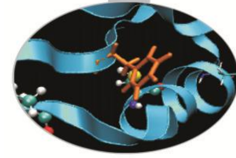
What are PETSc matrices?

- 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, CUSP (GPU, dev-only) ...
- 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

Matrix AIJ format

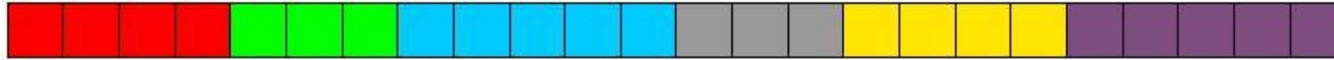


	0	1	2	3	4	5	6	7	8
0									
1									
2									
3									
4									
5									

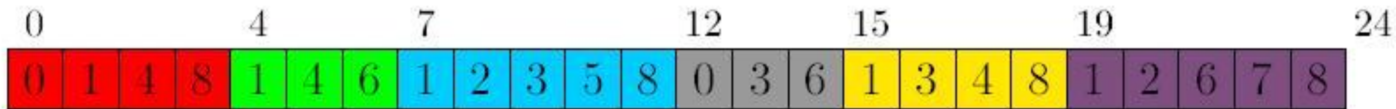
The default matrix representation within PETSc is the general sparse **AIJ format** (Yale sparse matrix or Compressed Sparse Row, CSR)

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

value

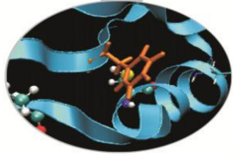


index

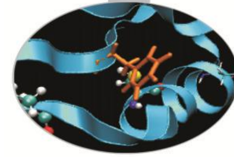


row pointer

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.

$$
\begin{array}{l}
P0 \\ \\ \\ P1 \\ \\ \\ P2 \\ \\
\end{array}
\left(\begin{array}{ccc|ccc|cc}
1 & 2 & 0 & 0 & 3 & 0 & 0 & 4 \\
0 & 5 & 6 & 7 & 0 & 0 & 8 & 0 \\
9 & 0 & 10 & 11 & 0 & 0 & 12 & 0 \\
\hline
13 & 0 & 14 & 15 & 16 & 17 & 0 & 0 \\
0 & 18 & 0 & 19 & 20 & 21 & 0 & 0 \\
0 & 0 & 0 & 22 & 23 & 0 & 24 & 0 \\
\hline
25 & 26 & 27 & 0 & 0 & 28 & 29 & 0 \\
30 & 0 & 0 & 31 & 32 & 33 & 0 & 34
\end{array}\right)
$$

Process 0

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

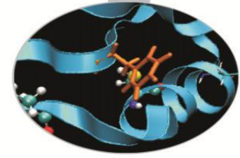
Process 1

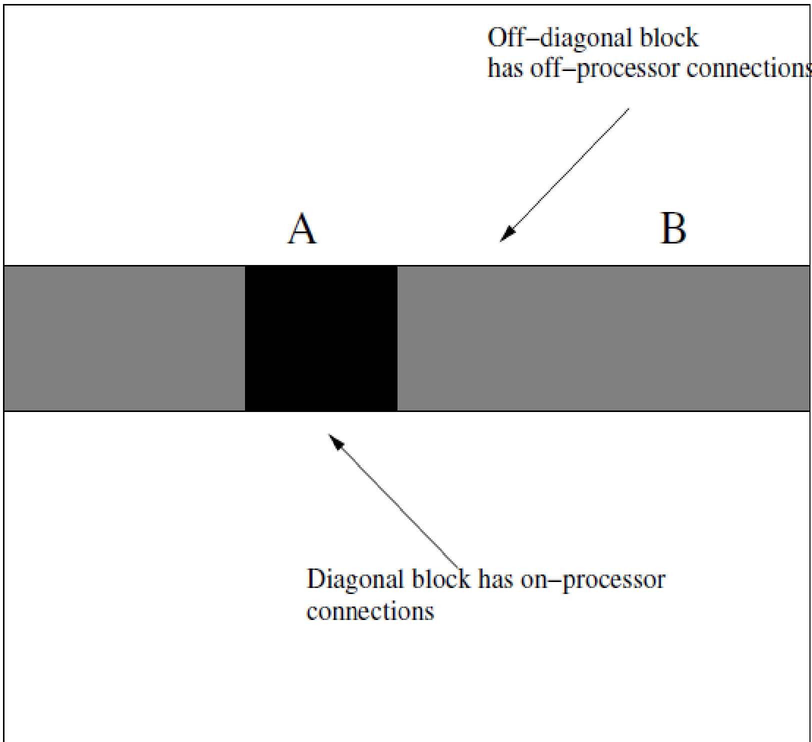
```
dnz=3, onz=2
dnnz[0]=3, onnz[0]=2
dnnz[1]=3, onnz[1]=1
dnnz[2]=2, onnz[2]=1
```

Process 2

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

Preallocation of a parallel sparse matrix





y ← A x_A + B x_B

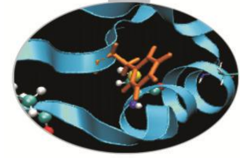
- x_B needs to be communicated
- A x_A can be computed in the meantime

Algorithm

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

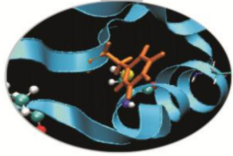
The splitting of the matrix storage into A (diag) and B (off-diag) part, code for the
sequential case can be reused.

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);	$Y = Y + a * I$

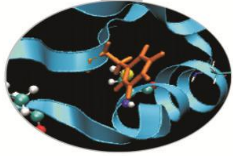
Sparse Matrices and Linear Solvers



- Solve a linear system A x = b using the Gauss Elimination method can be very time-resource consuming

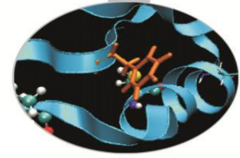
- Alternatives to direct solvers are iterative solvers
- Convergence of the succession is not always guarateed
- Possibly much faster and less memory consuming
- Basic iteration: y <- A x executed once x iteration
- Also needed a good preconditioner: $B \approx A^{-1}$

Iterative solver basics



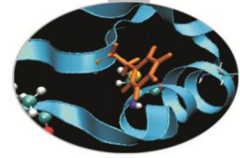
- **KSP** (*Krylov SPace Methods*) objects are used for solving linear systems by means of iterative methods.
- Convergence can be improved by using a suitable **PC** object (preconditoner).
- Almost all iterative methods are implemented.
- Classical iterative methods (not belonging to KSP solvers) are 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



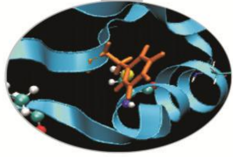
Method	KSPType	Options Database Name
Richardson	KSPRICHARDSON	richardson
Chebyshev	KSPCHEBYSHEV	chebyshev
Conjugate Gradient [12]	KSPCG	cg
BiConjugate Gradient	KSPBICG	bicg
Generalized Minimal Residual [16]	KSPGMRES	gmres
Flexible Generalized Minimal Residual	KSPFGMRES	fgmres
Deflated Generalized Minimal 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

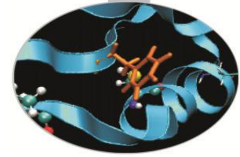
Factorization preconditioner



- Exact factorization: A = LU
- Inexact factorization: $A \approx M = \underline{L}\,\underline{U}$ where $\underline{L}$, $\underline{U}$ 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

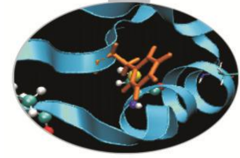
- PCICC: symmetric matrix, PCILU: nonsymmetric matrix

Parallel preconditioners

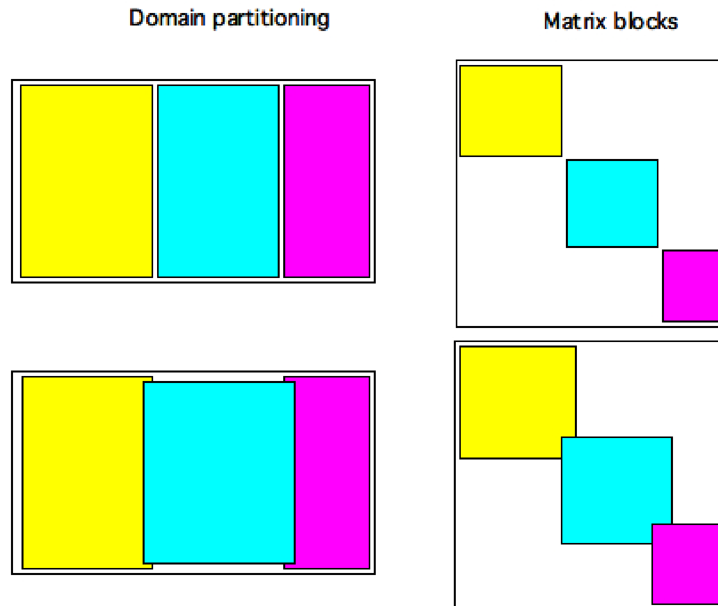


- Factorization preconditioners are sequential
- We can use them in parallel as a subpreconditioner of a parallel preconditioner as Block Jacobi or Additive Schwarz methods
- Each processor has its own block(s) to work with

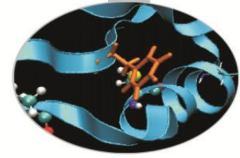
Block Jacobi and Additive Schwarz preconditioners



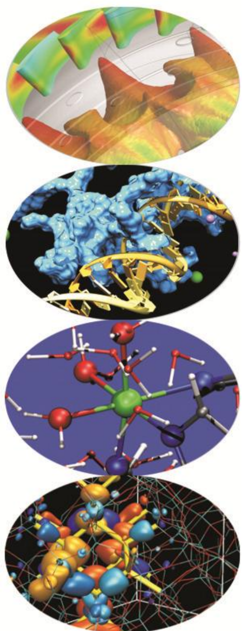
- Both methods are parallel
- BlockJacobi is fully parallel, Schwarz requires communications between neighbours
- Both require sequential local solver
- Schwarz can be more robust than BlockJacobi and have better convergence properties



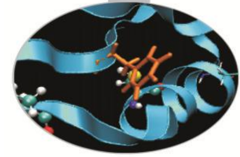
CINECA SCAI
SuperComputing Applications and Innovation



Case Studies: Engineering Applications and Domain Decomposition in HPC



Case study: Engineering Applications



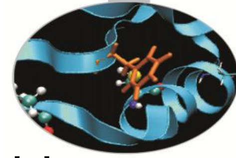
- Typical applications of Sparse Matrices are in all engineering problems where **large linear systems** generated by Finite Difference or Finite Volume discretizations have to be solved:

- CFD (Navier-Stokes equations – parabolic equations)
  - $\frac{\partial u}{\partial t} + (\mathrm{u} \cdot \nabla) + \nabla p - \nu \nabla^2 u + f = 0$

- Heat transfer (Poisson's equation – elliptic equation)
  - $k \, \Delta T = f$

- CSM (Time dependent Elasticity equations – hyperbolic equations)
  - $\frac{\partial^2 u}{\partial t^2} = \nabla \cdot \sigma(u) + f$

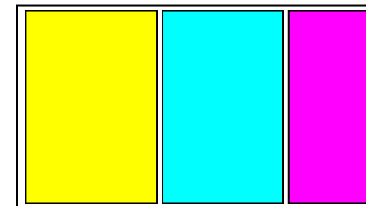
Case study: Domain Decomposition in HPC

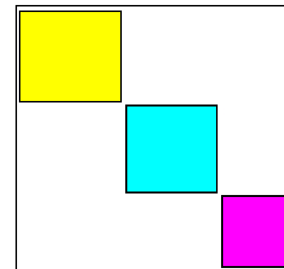


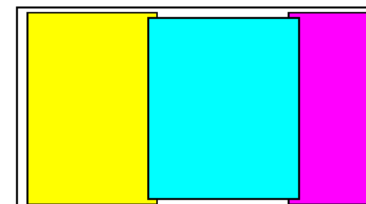
- In the HPC world, the matrices cannot be stored in a single machine due to the limitation of the memory installed in a single node
- One solution is to decompose the domain of the equations in many subdomains (DD: domain decomposition)
- The initial matrix is decomposed in many blocks, each of them can be stored in a different node of the HPC machine
- Many decomposition have been proposed in literature *(for a reference see: A. Valli, A. Quarteroni, Domain Decomposition Methods for Partial Differential Equations)*:
    - Classical Schwartz algorithm (Dirichlet – Dirichlet DD)
    - Block Jacobi preconditioner
    - Balancing Domain Decomposition by Constraints (PCBDDC)

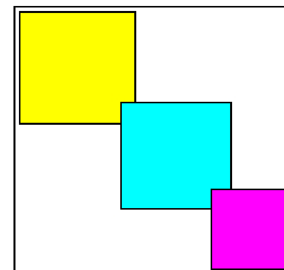
Domain partitioning

Matrix blocks

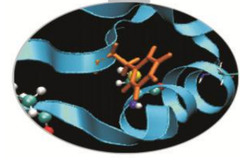


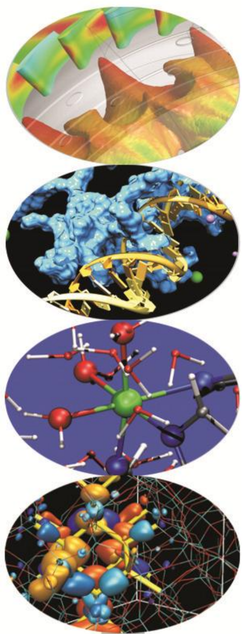






CINECA SCAI
SuperComputing Applications and Innovation





Thank you for the attention