Parallel Solver Essentials for Computational Scientists

Part 2: Distributed Memory Systems

Karl Rupp

me@karlrupp.net
https://karlrupp.net/
@karlrupp

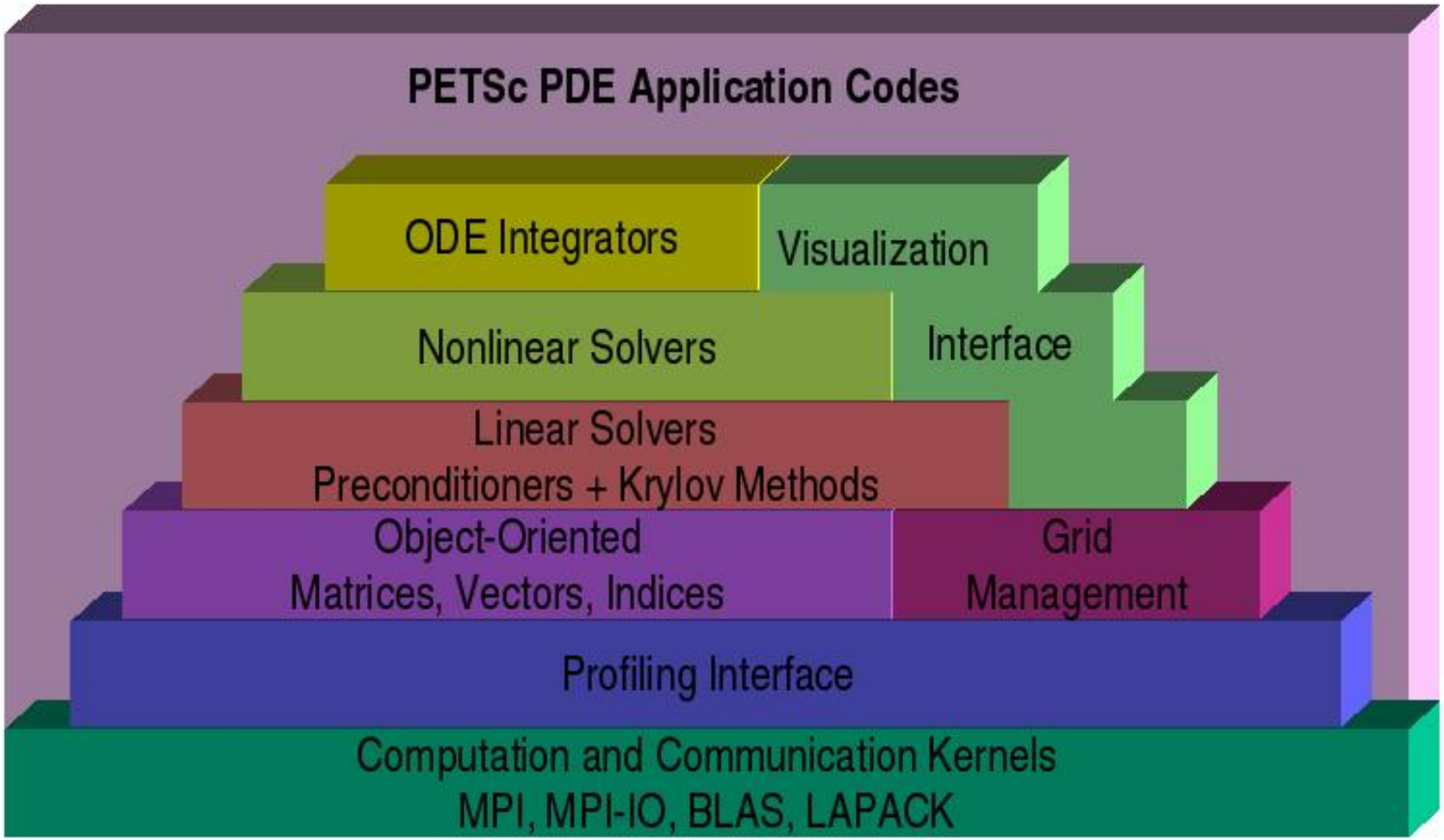
now:
Freelance Scientist

formerly and soon again:
Institute for Microelectronics, TU Wien

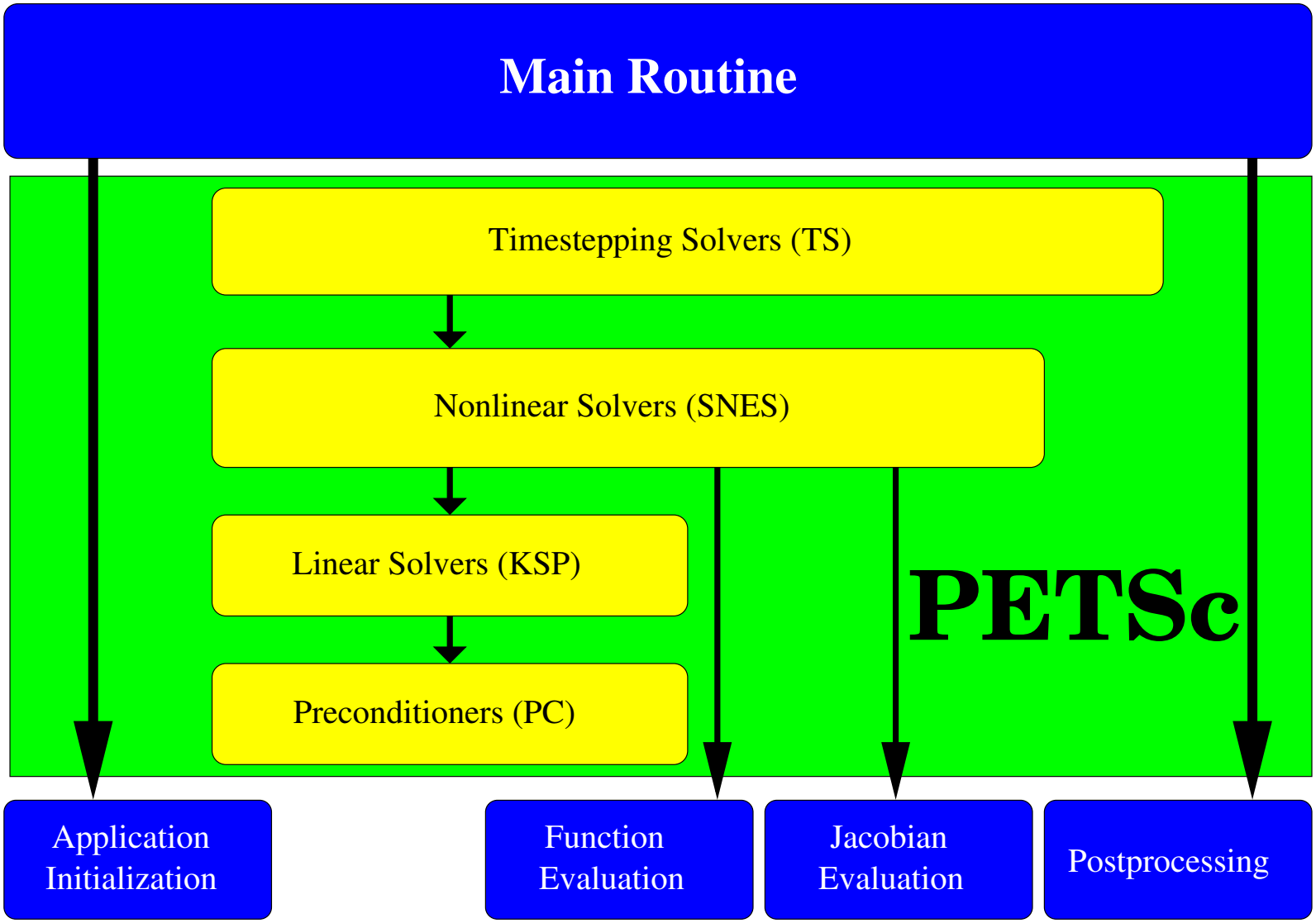
Kazan Federal University
October 19, 2016

PETSc Pyramid

PETSc Structure



Flow Control for a PETSc Application



Vectors and Matrices

The Role of PETSc

You want to think about how you decompose your data structures, how you think about them globally. [...]

If you were building a house, you'd start with a set of blueprints that give you a picture of what the whole house looks like. You wouldn't start with a bunch of tiles and say. "Well I'll put this tile down on the ground, and then I'll find a tile to go next to it."

But all too many people try to build their parallel programs by creating the smallest possible tiles and then trying to have the structure of their code emerge from the chaos of all these little pieces. You have to have an organizing principle if you're going to survive making your code parallel.

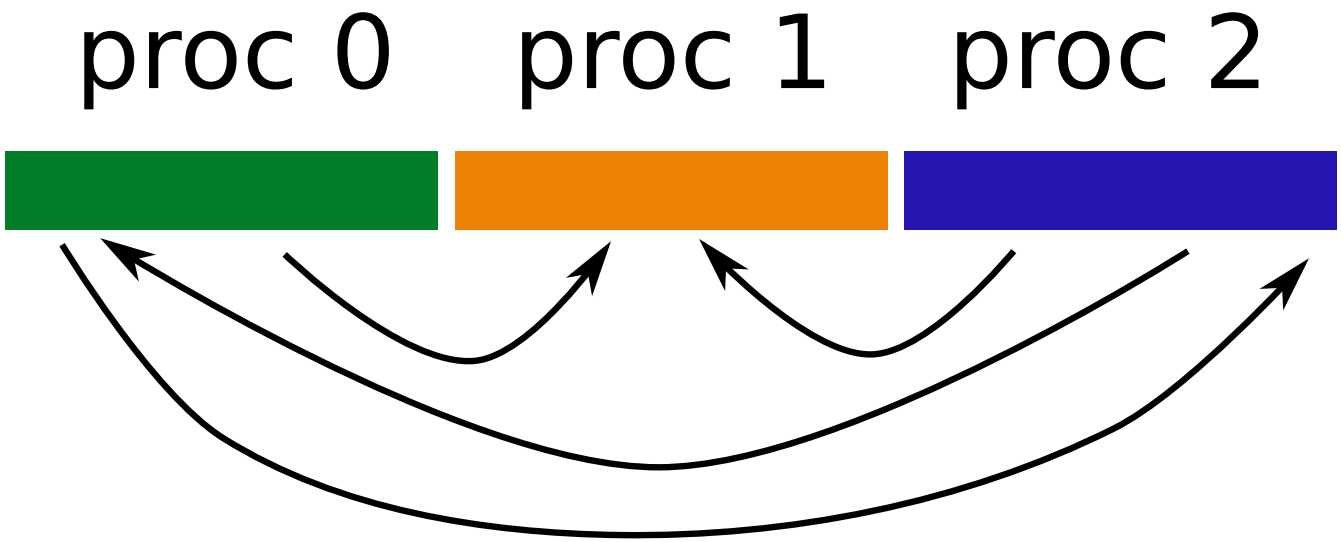
— Bill Gropp
— http://www.rce-cast.com/Podcast/rce-28-mpich2.html

PETSc Vectors

Parallel Vector Layout


```
VecCreate(PETSC_COMM_WORLD, &x);
VecSetSizes(x, PETSC_DECIDE, N);
VecSetFromOptions(x);
```

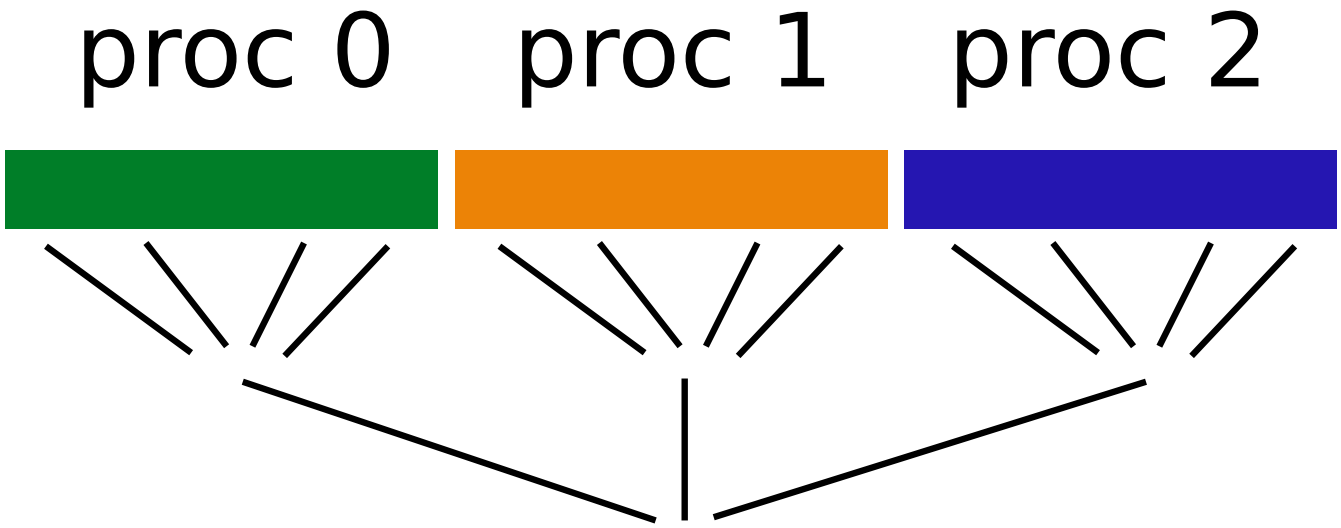
Vector Gather and Scatter




```
// y[iy[i]] = x[ix[i]]
VecScatterCreate(...);
VecScatterBegin(...);
VecScatterEnd(...);
```

PETSc Vectors

Vector Reductions




```
VecNorm(...);
VecDot(...);
VecMax(...);
...
```

PETSc Vectors

Local (Sequential) Operations

Executed by an arbitrary subset of MPI ranks

Usually involve `VecGetArray()/VecRestoreArray()`

Collective Operations

Must be executed by all processes in the MPI communicator

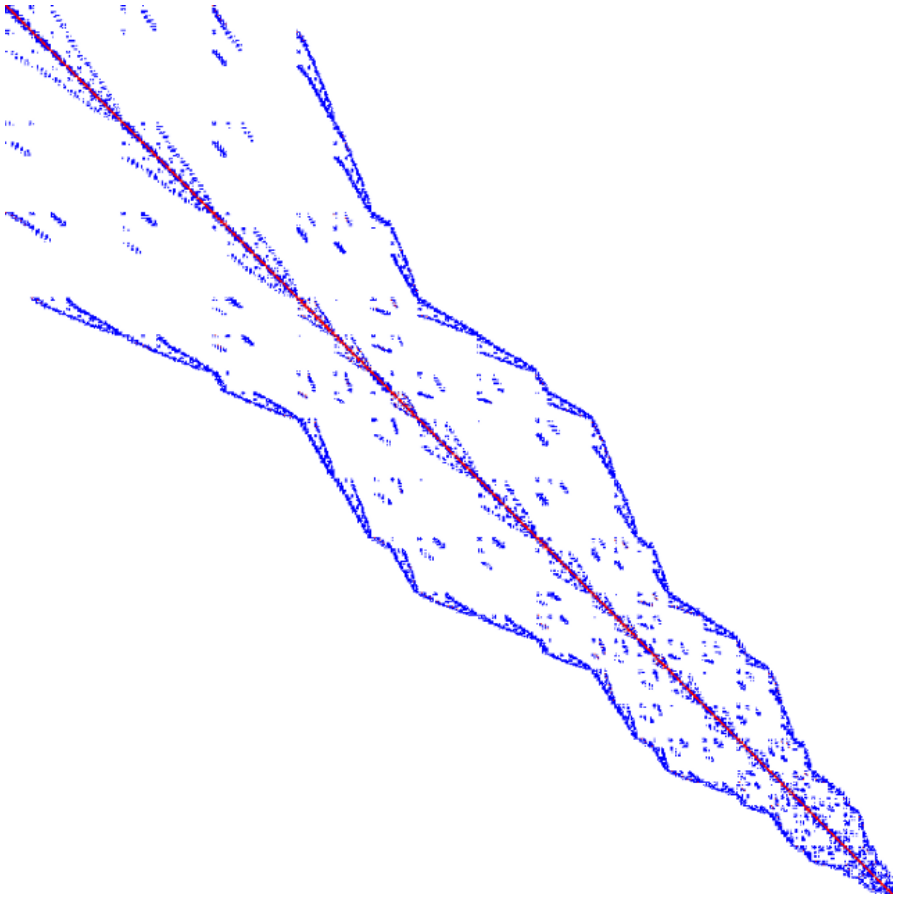
Involve MPI operations (scatter, gather, reduce, etc.)

Sparse Matrices

The important data type when solving PDEs

Two main phases:

- Filling with entries (assembly)
- Application of its action (e.g. SpMV)



Matrix Memory Preallocation

PETSc sparse matrices are dynamic data structures

- can add additional nonzeros freely

Dynamically adding many nonzeros

- requires additional memory allocations
- requires copies
- can kill performance

Memory preallocation provides

- the freedom of dynamic data structures
- good performance

Easiest solution is to replicate the assembly code

- Remove computation, but preserve the indexing code
- Store set of columns for each row

Call preallocation routines for all datatypes

- `MatSeqAIJSetPreallocation()`
- `MatMPIBAIJSetPreallocation()`
- Only the relevant data will be used

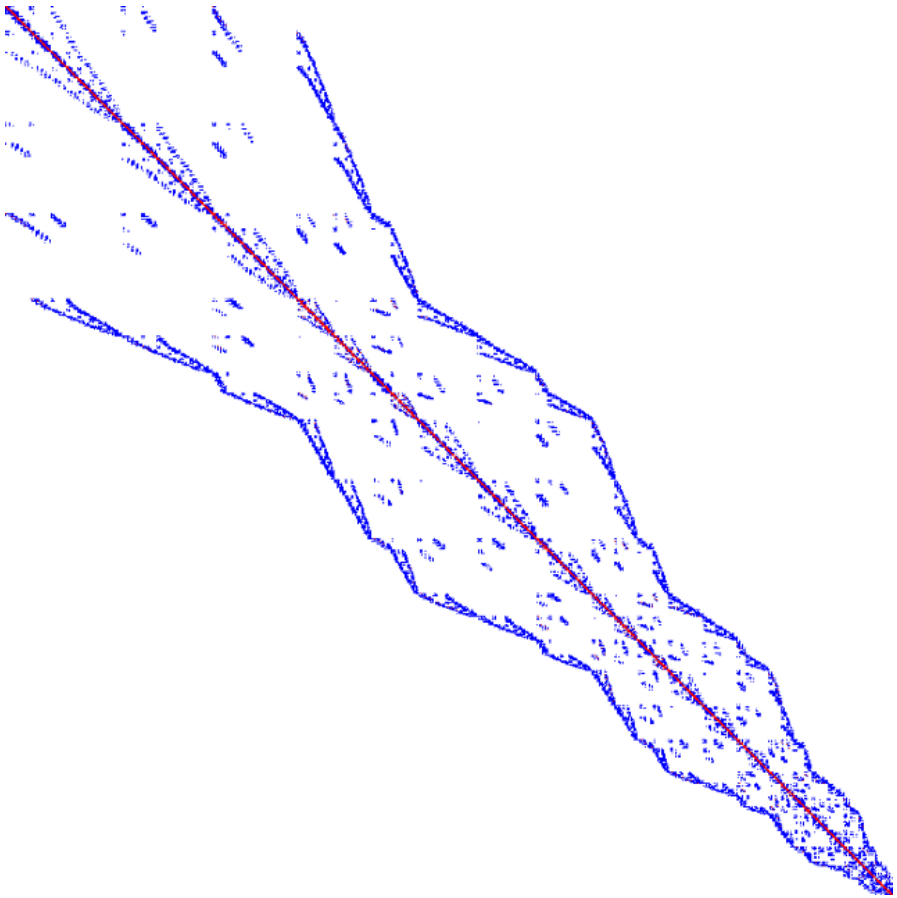
PETSc Application Integration

Sequential Sparse Matrices

```
MatSeqAIJSetPreallocation(Mat A, int nz, int nnz[])
```

nz: expected number of nonzeros in any row

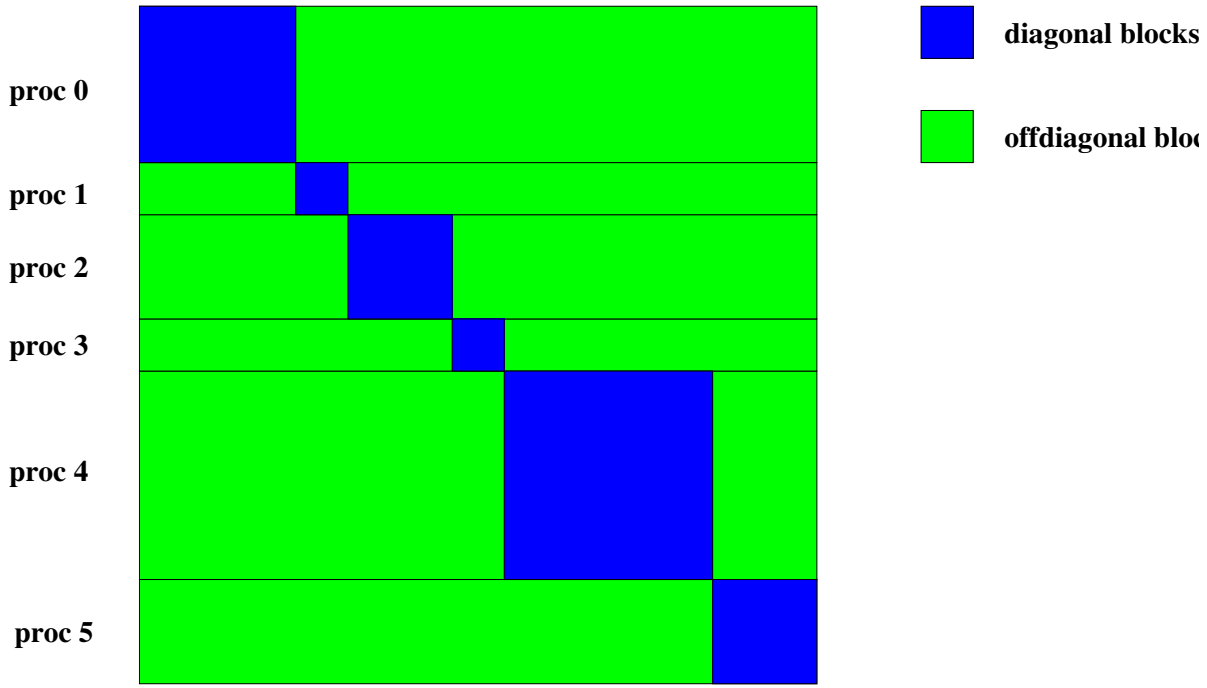
nnz(i): expected number of nonzeros in row i



Parallel Sparse Matrix

Each process locally owns a submatrix of contiguous global rows

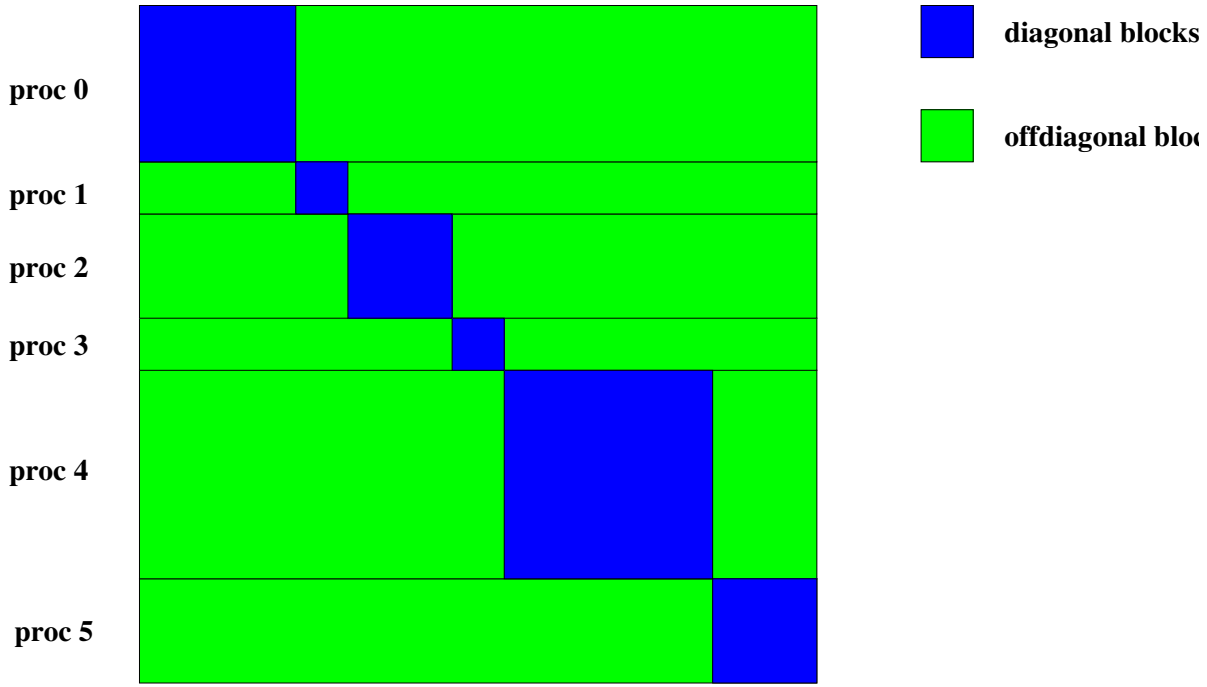
Each submatrix consists of diagonal and off-diagonal parts




```
MatGetOwnershipRange(Mat A,int *start,int *end)
```

`start`: first locally owned row of global matrix
`end-1`: last locally owned row of global matrix

PETSc Application Integration



Proc 2			Proc 3	
25	26	27	28	29
20	21	22	23	24
15	16	17	18	19
10	11	12	13	14
5	6	7	8	9
0	1	2	3	4
Proc 0			Proc 1	

Natural numbering

Proc 2			Proc 3	
21	22	23	28	29
18	19	20	26	27
15	16	17	24	25
6	7	8	13	14
3	4	5	11	12
0	1	2	9	10
Proc 0			Proc 1	

PETSc numbering

PETSc Application Integration

Parallel Sparse Matrix

```
MatMPIAIJSetPreallocation(Mat A, int dnz, int dnnz[],
                                 int onz, int onnz[]
```

dnz: expected number of nonzeros in any row in the diagonal block

dnnz(i): expected number of nonzeros in row i in the diagonal block

onz: expected number of nonzeros in any row in the offdiagonal portion

onnz(i): expected number of nonzeros in row i in the offdiagonal portion

Verifying Preallocation

Use runtime options

```
-mat_new_nonzero_location_err
-mat_new_nonzero_allocation_err
```

Use runtime option

```
-info
```

Output:

```
[proc #] Matrix size: %d X %d; storage space: %d unneeded, %d used
[proc #] Number of mallocs during MatSetValues( )  is %d
```

```
[merlin] mpirun ex2 -log_info
[0]MatAssemblyEnd_SeqAIJ:Matrix size: 56 X 56; storage space:
[0]   310 unneeded, 250 used
[0]MatAssemblyEnd_SeqAIJ:Number of mallocs during MatSetValues() is 0
[0]MatAssemblyEnd_SeqAIJ:Most nonzeros in any row is 5
[0]Mat_AIJ_CheckInode: Found 56 nodes out of 56 rows. Not using Inode routines
[0]Mat_AIJ_CheckInode: Found 56 nodes out of 56 rows. Not using Inode routines
Norm of error 0.000156044 iterations 6
[0]PetscFinalize:PETSc successfully ended!
```

Block and Symmetric Formats

BAIJ

- Like AIJ, but uses static block size
- Preallocation is like AIJ, but just one index per block

SBAIJ

- Only stores upper triangular part
- Preallocation needs number of nonzeros in upper triangular parts of on- and off-diagonal blocks

MatSetValuesBlocked()

- Better performance with blocked formats
- Also works with scalar formats, if `MatSetBlockSize()` was called
- Variants `MatSetValuesBlockedLocal()`, `MatSetValuesBlockedStencil()`
- Change matrix format at runtime, don't need to touch assembly code

One Way to Set the Elements of a Matrix

Simple 3-point stencil for 1D Laplacian

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

A Better Way to Set the Elements of a Matrix

A More Efficient Way

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

Advantages

All ranks busy: Scalable!

Amount of code essentially unchanged

Matrices

Definition (Matrix)

A matrix is a linear transformation between finite dimensional vector spaces.

Definition (Forming a matrix)

Forming or assembling a matrix means defining it's action in terms of entries (usually stored in a sparse format).

Important Matrices

1. Sparse (e.g. discretization of a PDE operator)
2. Inverse of *anything* interesting $B = A^{-1}$
3. Jacobian of a nonlinear function $Jy = \lim_{\epsilon \to 0} \frac{F(x+\epsilon y) - F(x)}{\epsilon}$
4. Fourier transform $\mathcal{F}, \mathcal{F}^{-1}$
5. Other fast transforms, e.g. Fast Multipole Method
6. Low rank correction $B = A + uv^T$
7. Schur complement $S = D - CA^{-1}B$
8. Tensor product $A = \sum_e A_x^e \otimes A_y^e \otimes A_z^e$
9. Linearization of a few steps of an explicit integrator

Important Matrices

1. Sparse (e.g. discretization of a PDE operator)
2. Inverse of *anything* interesting $B = A^{-1}$
3. Jacobian of a nonlinear function $Jy = \lim_{\epsilon \to 0} \frac{F(x+\epsilon y) - F(x)}{\epsilon}$
4. Fourier transform $\mathcal{F}, \mathcal{F}^{-1}$
5. Other fast transforms, e.g. Fast Multipole Method
6. Low rank correction $B = A + uv^T$
7. Schur complement $S = D - CA^{-1}B$
8. Tensor product $A = \sum_e A_x^e \otimes A_y^e \otimes A_z^e$
9. Linearization of a few steps of an explicit integrator

These matrices are dense. Never form them.

Important Matrices

1. Sparse (e.g. discretization of a PDE operator)
2. Inverse of *anything* interesting $B = A^{-1}$
3. Jacobian of a nonlinear function $Jy = \lim_{\epsilon \to 0} \frac{F(x+\epsilon y) - F(x)}{\epsilon}$
4. Fourier transform $\mathcal{F}, \mathcal{F}^{-1}$
5. Other fast transforms, e.g. Fast Multipole Method
6. Low rank correction $B = A + uv^T$
7. Schur complement $S = D - CA^{-1}B$
8. Tensor product $A = \sum_e A_x^e \otimes A_y^e \otimes A_z^e$
9. Linearization of a few steps of an explicit integrator

These are not very sparse. Don't form them.

Matrices

Important Matrices

1. Sparse (e.g. discretization of a PDE operator)
2. Inverse of *anything* interesting $B = A^{-1}$
3. Jacobian of a nonlinear function $Jy = \lim_{\epsilon \to 0} \frac{F(x+\epsilon y) - F(x)}{\epsilon}$
4. Fourier transform $\mathcal{F}, \mathcal{F}^{-1}$
5. Other fast transforms, e.g. Fast Multipole Method
6. Low rank correction $B = A + uv^T$
7. Schur complement $S = D - CA^{-1}B$
8. Tensor product $A = \sum_e A_x^e \otimes A_y^e \otimes A_z^e$
9. Linearization of a few steps of an explicit integrator

None of these matrices “have entries”

Iterative Solvers

Matrices

What can we do with a matrix that doesn't have entries?

Krylov solvers for $Ax = b$

Krylov subspace: $\{b, Ab, A^2b, A^3b, \dots\}$

Convergence rate depends on the spectral properties of the matrix

For any popular Krylov method $\mathcal{K}$, there is a matrix of size m, such that $\mathcal{K}$ outperforms all other methods by a factor at least $\mathcal{O}(\sqrt{m})$ [Nachtigal et. al., 1992]

Typically...

The action $y \leftarrow Ax$ can be computed in $\mathcal{O}(m)$

Aside from matrix multiply, the n^{th} iteration requires at most $\mathcal{O}(mn)$

GMRES

Brute force minimization of residual in $\{b, Ab, A^2b, \dots\}$

1. Use Arnoldi to orthogonalize the nth subspace, producing

$$AQ_n = Q_{n+1}H_n$$

2. Minimize residual in this space by solving the overdetermined system

$$H_n y_n = e_1^{(n+1)}$$

using QR-decomposition, updated cheaply at each iteration.

Properties

Converges in n steps for all right hand sides if there exists a polynomial of degree n such that $\|p_n(A)\| < \mathit{tol}$ and $p_n(0) = 1$.

Residual is monotonically decreasing, robust in practice

Restarted variants are used to bound memory requirements

Linear Solvers - Krylov Methods

Using PETSc linear algebra, just add:

```
KSPSetOperators(KSP ksp, Mat A, Mat M)
KSPSolve(KSP ksp, Vec b, Vec x)
```

Can access subobjects

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

Preconditioners must obey PETSc interface

- Basically just the KSP interface

Can change solver dynamically from the command line, `-ksp_type`

Linear solvers in PETSc KSP (Excerpt)

- Richardson
- Chebychev
- Conjugate Gradient
- BiConjugate Gradient
- Generalized Minimum Residual Variants
- Transpose-Free Quasi-Minimum Residual
- Least Squares Method
- Conjugate Residual

Preconditioners

Preconditioning

Idea: improve the conditioning of the Krylov operator

Left preconditioning

$$(P^{-1}A)x = P^{-1}b$$

$$\{P^{-1}b, (P^{-1}A)P^{-1}b, (P^{-1}A)^2P^{-1}b, \dots\}$$

Right preconditioning

$$(AP^{-1})Px = b$$

$$\{b, (P^{-1}A)b, (P^{-1}A)^2b, \dots\}$$

The product $P^{-1}A$ or AP^{-1} is *not* formed.

A *preconditioner* $\mathcal{P}$ is a method for constructing a matrix (just a linear function, not assembled!) $P^{-1} = \mathcal{P}(A, A_p)$ using a matrix A and extra information A_p, such that the spectrum of $P^{-1}A$ (or AP^{-1}) is well-behaved.

Preconditioning

Definition (Preconditioner)

A *preconditioner* $\mathcal{P}$ is a method for constructing a matrix $P^{-1} = \mathcal{P}(A, A_p)$ using a matrix A and extra information A_p, such that the spectrum of $P^{-1}A$ (or AP^{-1}) is well-behaved.

- P^{-1} is dense, P is often not available and is not needed
- A is rarely used by $\mathcal{P}$, but $A_p = A$ is common
- A_p is often a sparse matrix, the "preconditioning matrix"
- Matrix-based: Jacobi, Gauss-Seidel, SOR, ILU(k), LU
- Parallel: Block-Jacobi, Schwarz, Multigrid, FETI-DP, BDDC
- Indefinite: Schur-complement, Domain Decomposition, Multigrid

Questions to ask when you see a matrix

1. What do you want to do with it?
   - Multiply with a vector
   - Solve linear systems or eigen-problems
2. How is the conditioning/spectrum?
   - distinct/clustered eigen/singular values?
   - symmetric positive definite ($\sigma(A) \subset \mathbb{R}^+$)?
   - nonsymmetric definite ($\sigma(A) \subset \{z \in \mathbb{C} : \mathrm{Re}[z] > 0\}$)?
   - indefinite?
3. How dense is it?
   - block/banded diagonal?
   - sparse unstructured?
   - denser than we'd like?
4. Is there a better way to compute Ax?
5. Is there a different matrix with similar spectrum, but nicer properties?
6. How can we precondition A?

Questions to ask when you see a matrix

1. What do you want to do with it?
   - Multiply with a vector
   - Solve linear systems or eigen-problems
2. How is the conditioning/spectrum?
   - distinct/clustered eigen/singular values?
   - symmetric positive definite ($\sigma(A) \subset \mathbb{R}^+$)?
   - nonsymmetric definite ($\sigma(A) \subset \{z \in \mathbb{C} : \mathrm{Re}[z] > 0\}$)?
   - indefinite?
3. How dense is it?
   - block/banded diagonal?
   - sparse unstructured?
   - denser than we'd like?
4. Is there a better way to compute Ax?
5. Is there a different matrix with similar spectrum, but nicer properties?
6. How can we precondition A?

Relaxation

Split into lower, diagonal, upper parts: $A = L + D + U$

Jacobi

Cheapest preconditioner: $P^{-1} = D^{-1}$

Successive over-relaxation (SOR)

$$\left(L + \frac{1}{\omega}D\right) x_{n+1} = \left[\left(\frac{1}{\omega} - 1\right) D - U\right] x_n + \omega b$$

$P^{-1} = k$ iterations starting with $x_0 = 0$

Implemented as a sweep

$\omega = 1$ corresponds to Gauss-Seidel

Very effective at removing high-frequency components of residual

Factorization

Two phases

symbolic factorization: find where fill occurs, only uses sparsity pattern
numeric factorization: compute factors

LU decomposition

Ultimate preconditioner
Expensive, for $m \times m$ sparse matrix with bandwidth b, traditionally requires $\mathcal{O}(mb^2)$ time and $\mathcal{O}(mb)$ space.

- Bandwidth scales as $m^{\frac{d-1}{d}}$ in d-dimensions
- Optimal in 2D: $\mathcal{O}(m \cdot \log m)$ space, $\mathcal{O}(m^{3/2})$ time
- Optimal in 3D: $\mathcal{O}(m^{4/3})$ space, $\mathcal{O}(m^2)$ time

Symbolic factorization is problematic in parallel

Incomplete LU

Allow a limited number of levels of fill: ILU(k)
Only allow fill for entries that exceed threshold: ILUT
Usually poor scaling in parallel
No guarantees

1-level Domain decomposition

Domain size L, subdomain size H, element size h

Overlapping/Schwarz

Solve Dirichlet problems on overlapping subdomains

No overlap: *its* $\in \mathcal{O}\left(\frac{L}{\sqrt{Hh}}\right)$

Overlap δ: *its* $\in \left(\frac{L}{\sqrt{H\delta}}\right)$

Neumann-Neumann

Solve Neumann problems on non-overlapping subdomains

its $\in \mathcal{O}\left(\frac{L}{H}(1+\log\frac{H}{h})\right)$

Tricky null space issues (floating subdomains)

Need subdomain matrices, net globally assembled matrix.

Multilevel variants knock off the leading $\frac{L}{H}$

Both overlapping and nonoverlapping with this bound

Multigrid

Hierarchy: Interpolation and restriction operators

$$\mathcal{I}^{\uparrow} : X_{\text{coarse}} \to X_{\text{fine}} \qquad \mathcal{I}^{\downarrow} : X_{\text{fine}} \to X_{\text{coarse}}$$

Geometric: define problem on multiple levels, use grid to compute hierarchy
Algebraic: define problem only on finest level, use matrix structure to build hierarchy

Galerkin approximation

Assemble this matrix: $A_{\text{coarse}} = \mathcal{I}^{\downarrow} A_{\text{fine}} \mathcal{I}^{\uparrow}$

Application of multigrid preconditioner (V-cycle)

Apply pre-smoother on fine level (any preconditioner)
Restrict residual to coarse level with $\mathcal{I}^{\downarrow}$
Solve on coarse level $A_{\text{coarse}} x = r$
Interpolate result back to fine level with $\mathcal{I}^{\uparrow}$
Apply post-smoother on fine level (any preconditioner)

Multigrid convergence properties

Textbook: $P^{-1}A$ is spectrally equivalent to identity
- Constant number of iterations to converge up to discretization error

Most theory applies to SPD systems
- variable coefficients (e.g. discontinuous): low energy interpolants
- mesh- and/or physics-induced anisotropy: semi-coarsening/line smoothers
- complex geometry: difficult to have meaningful coarse levels

Deeper algorithmic difficulties
- nonsymmetric (e.g. advection, shallow water, Euler)
- indefinite (e.g. incompressible flow, Helmholtz)

Performance considerations
- Aggressive coarsening is critical in parallel
- Most theory uses SOR smoothers, ILU often more robust
- Coarsest level usually solved semi-redundantly with direct solver

Multilevel Schwarz is essentially the same with different language
- assume strong smoothers, emphasize aggressive coarsening

Splitting for Multiphysics

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} f \\ g \end{bmatrix}$$

Relaxation: `-pc_fieldsplit_type [additive,multiplicative,symmetric_multiplicative]`

$$\begin{bmatrix} A & \\ & D \end{bmatrix}^{-1} \qquad \begin{bmatrix} A & \\ C & D \end{bmatrix}^{-1} \qquad \begin{bmatrix} A & \\ & \mathbf{1} \end{bmatrix}^{-1} \left(\mathbf{1} - \begin{bmatrix} A & B \\ & \mathbf{1} \end{bmatrix} \begin{bmatrix} A & \\ C & D \end{bmatrix}^{-1} \right)$$

Gauss-Seidel inspired, works when fields are loosely coupled

Factorization: `-pc_fieldsplit_type schur`

$$\begin{bmatrix} A & B \\ & S \end{bmatrix}^{-1} \begin{bmatrix} \mathbf{1} & \\ CA^{-1} & \mathbf{1} \end{bmatrix}^{-1}, \qquad S = D - CA^{-1}B$$

robust (exact factorization), can often drop lower block
how to precondition S which is usually dense?

interpret as differential operators, use approximate commutators

End of Lecture

Distributed Systems

- Mind data partitioning
- Asymptotics of solvers important
- Reuse existing packages!

End of Lecture

Distributed Systems

- Mind data partitioning
- Asymptotics of solvers important
- Reuse existing packages!

General Recommendations

- Consult literature for preconditioners
- Performance modeling is very handy
- Your time is more valuable than computing time!