Tightly coupled solvers with loosely coupled software

Modular linear algebra for multi-physics

Jed Brown

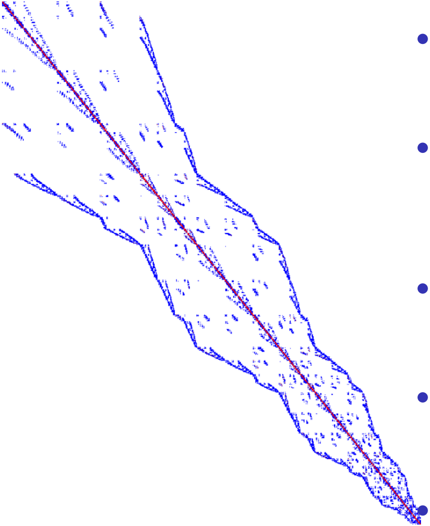
Laboratory of Hydrology, Hydraulics, and Glaciology
ETH Zürich

KAUST 2011-03-27

Outline

Outline

Bottlenecks of (Jacobian-free) Newton-Krylov



- Matrix assembly
  - integration/fluxes: FPU
  - insertion: memory/branching
- Preconditioner setup
  - coarse level operators
  - overlapping subdomains
  - (incomplete) factorization
- Preconditioner application
  - triangular solves/relaxation: memory
  - coarse levels: network latency
- Matrix multiplication
  - Sparse storage: memory
  - Matrix-free: FPU
- Globalization

Hardware capabilities

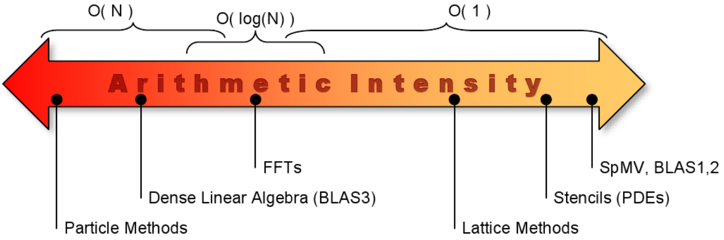
Floating point unit

Recent Intel: each core can issue

- 1 packed add (latency 3)
- 1 packed mult (latency 5)
- One can include an aligned read
- Out of Order execution
- Peak: 10 Gflop/s (`double`)

Memory

- $\sim$ 250 cycle latency
- 5.3 GB/s bandwidth
- 1 `double` load / 3.7 cycles
- Pay by the cache line (32/64 B)
- L2 cache: $\sim$ 10 cycle latency



Memory Bandwidth

- Stream Triad benchmark (GB/s): $w \leftarrow \alpha x + y$

Threads per Node	Cray XT5		BlueGene/P	
	Total	Per Core	Total	Per Core
1	8448	8448	2266	2266
2	10112	5056	4529	2264
4	10715	2679	8903	2226
6	10482	1747	-	-

- Sparse matrix-vector product: 6 bytes per flop

Machine	Peak MFlop/s per core	Bandwidth (GB/s)		Ideal MFlop/s
		Required	Measured	
Blue Gene/P	3,400	20.4	2.2	367
XT5	10,400	62.4	1.7	292

Sparse Mat-Vec performance model

Compressed Sparse Row format (AIJ)

For $m \times n$ matrix with N nonzeros

- ai row starts, length $m+1$
- aj column indices, length N, range $[0, n-1)$
- aa nonzero entries, length N, scalar values

$$y \leftarrow y + Ax$$

```
for (i=0; i<m; i++)
  for (j=ai[i]; j<ai[i+1]; j++)
    y[i] += aa[j] * x[aj[j]];
```

- One add and one multiply per inner loop
- Scalar `aa[j]` and integer `aj[j]` only used once
- Must load `aj[j]` to read from `x`, may not reuse cache well

Optimizing Sparse Mat-Vec

- Order unknows so that vector reuses cache (Reverse Cuthill-McKee)
  - Optimal: $\dfrac{(2\text{ flops})(\text{bandwidth})}{\texttt{sizeof(Scalar)}+\texttt{sizeof(Int)}}$
  - Usually improves strength of ILU and SOR
- Coalesce indices for adjacent rows with same nonzero pattern (Inodes)
  - Optimal: $\dfrac{(2\text{ flops})(\text{bandwidth})}{\texttt{sizeof(Scalar)}+\texttt{sizeof(Int)}/i}$
  - Can do block SOR (much stronger than scalar SOR)
  - Default in PETSc, turn off with `-mat_no_inode`
  - Requires ordering unknowns so that fields are interlaced, this is (much) better for memory use anyway
- Use explicit blocking, hold one index per block (BAIJ format)
  - Optimal: $\dfrac{(2\text{ flops})(\text{bandwidth})}{\texttt{sizeof(Scalar)}+\texttt{sizeof(Int)}/b^2}$
  - Block SOR and factorization
  - Symbolic factorization works with blocks (much cheaper)
  - Very regular memory access, unrolled dense kernels
  - Faster insertion: `MatSetValuesBlocked()`

Performance of blocked matrix formats

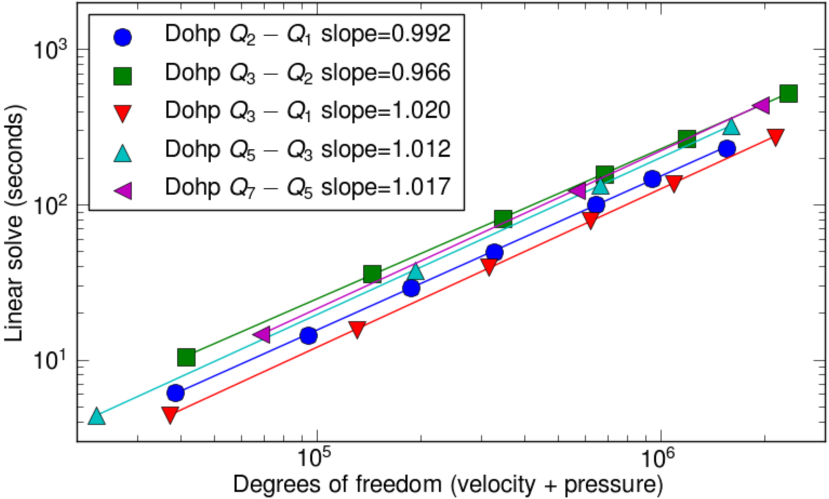
Kernel \ Format	Core 2, 1 process			Opteron, 4 processes		
	AIJ	BAIJ	SBAIJ	AIJ	BAIJ	SBAIJ
`MatMult`	812	985	1507	2226	2918	3119
`MatSolve`	718	957	955	1573	2869	2858

Throughput (Mflop/s) for different matrix formats on Core 2 Duo (P8700) and Opteron 2356 (two sockets). `MatSolve` is a forward- and back-solve with incomplete Cholesky factors. The AIJ format is using “inodes” which unrolls across consecutive rows with identical nonzero pattern (pairs in this case).

Optimizing unassembled Mat-Vec

- High order spatial discretizations do more work per node
  - Dense tensor product kernel (like small BLAS3)
  - Cubic (Q_3) elements in 3D can achieve $> 70\%$ of peak FPU (compare to $< 6\%$ for assembled operators on multicore)
  - Can store Jacobian information at quadrature points (usually pays off for Q_2 and higher in 3D)
  - Spectral, WENO, DG, FD
  - Often still need an assembled operator for preconditioning
- Boundary element methods
  - Dense kernels
  - Fast Multipole Method (FMM)

Power-law Stokes Scaling



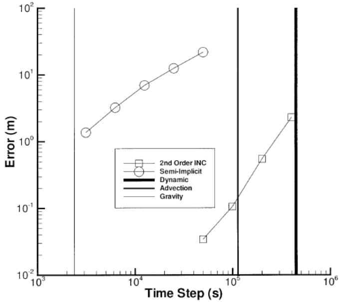
Only assemble Q_1 matrices, ML+PETSc smoothers for elliptic pieces

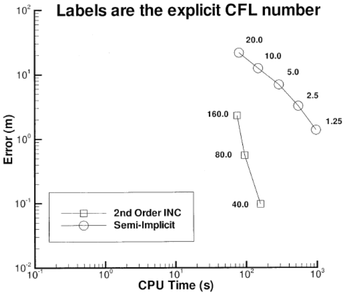
(fairly easy geometry and coefficients, Brown 2010)

What you can do

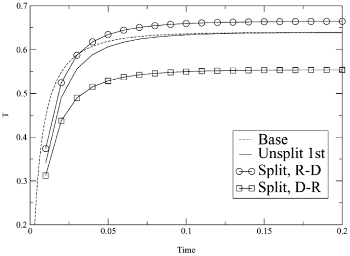
- Speak at the most specific language possible
  - 3D structural analysis: symmetric block size 3
  - 3D compressible flow: nonsymmetric block size 5
- Order unknowns for cache reuse (low-bandwidth like RCM is good)
- Dual order
  - Assemble a low-order discretization
  - Provide matrix-free high-order operator
    (FD, ADI, caching at quadrature points)
  - More robust with SOR and ILU due to *h*-ellipticity
  - Sometimes Picard linearization has a more compact stencil

Outline





Shallow water traveling vortex with Coriolis. Moussau et al, 2002.



Linear reaction-diffusion, split method converges to the wrong steady state . Knoll et al, 2003.

- CFL too restrictive for explicit
  - But hyperbolic systems do not weak scale if you care about phase
- Naive semi-implicit has poor accuracy, stability, robustness
- Good IMEX exists, but still need to treat stiff part implicitly

Coupled approach to multiphysics

- Smooth all components together
  - Block SOR is the most popular
  - Vanka smoothers for indefinite problems
  - Block ILU often more robust
- Scaling between fields is critical
- Indefiniteness
  - Make smoothers and interpolants respect inf-sup condition
  - Difficult to handle anisotropy
  - Can use Schur field-split to define a smoother
- Transport
  - Define smoother in terms of first-order upwind discretization (h-ellipticity)
  - Evaluate residuals using high-order discretization
  - Use Schur field-split to “parabolize” at the top level or to define smoother on levels
- Open research area, hard to write modular software

Anisotropy, Heterogeneity

- Anisotropy
  - Semi-coarsening
  - Line smoothers
  - Order unknowns so that incomplete factorization “includes” a line smoother
- Heterogeneity
  - Make coarse grids align
  - Strong smoothers
  - Energy-minimizing interpolants
- Mostly possible with generic software components

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]`

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

  - Gauss-Seidel inspired, works when fields are loosely coupled
- Factorization: `-pc_fieldsplit_type schur`

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

  - robust (exact factorization), can often drop lower block
  - how to precondition S which is usually dense?
    - interpret as differential operators, use approximate commutators

Physics-based preconditioners (semi-implicit method)

Shallow water with stiff gravity wave

h is hydrostatic pressure, u is velocity, $\sqrt{gh}$ is fast wave speed

$$h_t - (uh)_x = 0$$
$$(uh)_t + (u^2 h + \frac{1}{2} g h^2)_x = 0$$

Semi-implicit method

Suppress spatial discretization, discretize in time, implicitly for the terms contributing to the gravity wave

$$\frac{h^{n+1} - h^n}{\Delta t} + (uh)_x^{n+1} = 0$$
$$\frac{(uh)^{n+1} - (uh)^n}{\Delta t} + (u^2 h)_x^n + g(h^n h^{n+1})_x = 0$$

Rearrange, eliminating $(uh)^{n+1}$

$$\frac{h^{n+1} - h^n}{\Delta t} - \Delta t (g h^n h_x^{n+1})_x = -S_x^n$$

Delta form

- Preconditioner should work like the Newton step

$$-F(x) \mapsto \delta x$$

- Recast semi-implicit method in delta form

$$\frac{\delta h}{\Delta t} + (\delta uh)_x = -F_0, \quad \frac{\delta uh}{\Delta t} + gh^n(\delta h)_x = -F_1, \quad \hat{J} \begin{pmatrix} \frac{1}{\Delta t} & \mathrm{div} \\ gh^n \nabla & \frac{1}{\Delta t} \end{pmatrix}$$

- Eliminate δuh

$$\frac{\delta h}{\Delta t} - \Delta t(gh^n(\delta h)_x)_x = -F_0 + (\Delta t F_1)_x, \quad S \sim \frac{1}{\Delta t} - g\Delta t \,\mathrm{div}\, h^n \nabla$$

- Solve for δh, then evaluate

$$\delta uh = -\Delta t\left[gh^n(\delta h)_x - F_1\right]$$

- Fully implicit solver
  - Is nonlinearly consistent (no splitting error), can be high-order in time
  - Leverages existing code when a semi-implicit method has been implemented
  - Allows efficient bifurcation analysis, steady-state analysis

Stokes

Weak form of the Newton step

Find (u,p) such that

$$\int_\Omega (Dv)^T \left[\eta 1 + \eta' Dw \otimes Dw\right] Du$$
$$- p\nabla \cdot v - q\nabla \cdot u = -v \cdot F(w) \qquad\qquad \forall (v,q)$$

Matrix

$$\begin{bmatrix} A(w) & B^T \\ B & \end{bmatrix} \begin{pmatrix} u \\ p \end{pmatrix} = -\begin{pmatrix} F_u(w) \\ 0 \end{pmatrix}$$

Block factorization

$$\begin{bmatrix} A & B^T \\ B & \end{bmatrix} = \begin{bmatrix} 1 & \\ BA^{-1} & 1 \end{bmatrix} \begin{bmatrix} A & B^T \\ & S \end{bmatrix} = \begin{bmatrix} A & \\ B & S \end{bmatrix} \begin{bmatrix} 1 & A^{-1}B^T \\ & 1 \end{bmatrix}$$

where the Schur complement is

$$S = -BA^{-1}B^T.$$

Properties of the Schur complement

Block factorization

$$\begin{bmatrix} A & B^T \\ B & \end{bmatrix} = \begin{bmatrix} 1 & \\ BA^{-1} & 1 \end{bmatrix} \begin{bmatrix} A & B^T \\ & S \end{bmatrix} = \begin{bmatrix} A & \\ B & S \end{bmatrix} \begin{bmatrix} 1 & A^{-1}B^T \\ & 1 \end{bmatrix}$$

where

$$S = -BA^{-1}B^T.$$

- S is symmetric negative definite if A is SPD and B has full rank (discrete inf-sup condition)
- S is dense
- We only need to multiply B, B^T with vectors.
- We need preconditioners for A and S.
- Any definite preconditioner can be used for A.
- It's not obvious how to precondition S, more on that later.

Preconitioning the Schur complement

- $S = -BA^{-1}B^T$ is dense so we can't form it, we need S^{-1}.

Physics-based commutator: anisotropic pressure diffusion

$$v^T A(w) u \sim \int (Dv)^T \left[\eta 1 + \eta' Dw \otimes Dw\right] Du$$

- We would like to find an operator A_p such that

$$-S = BA^{-1}B^T \approx BB^T A_p^{-1} =: P_S$$

  so that

$$P_S^{-1} = A_p (BB^T)^{-1}$$

- Note

$$BB^T \sim (-\nabla\cdot)\nabla = -\Delta$$

  corresponds to a Laplacian in the pressure space (multigrid).
- If $\eta', \nabla\eta \ll 1$ then $A_p \sim -\eta\Delta$ so $P_S^{-1} = \eta 1$

Least squares commutator

- Schur complement

$$S = -BA^{-1}B^T$$

Suppose B is square and nonsingular. Then

$$S^{-1} = -B^{-T}AB^{-1}.$$

B is not square, replace B^{-1} with Moore-Penrose pseudoinverse

$$B^{\dagger} = B^T(BB^T)^{-1}, \qquad (B^T)^{\dagger} = (BB^T)^{-1}B.$$

Then

$$P_S^{-1} = -(BB^T)^{-1}BAB^T(BB^T)^{-1}.$$

- Requires 2 Poisson preconditioners for $(BB^T)^{-1}$ per iteration
- Better with scaling, from mass matrices and effective viscosity (Elman et al. 2006, May & Moresi 2008)
- `-pc_type fieldsplit -pc_fieldsplit_type schur -fieldsplit_p_pc_type lsc -fieldsplit_p_lsc_pc_type mg`

Unsteady Navier-Stokes

Strong form

$$J(w)\begin{bmatrix}u\\p\end{bmatrix} \sim \begin{cases} \rho(\alpha u + w\cdot\nabla u + u\cdot\nabla w) - \eta\nabla^2 u + \nabla p &= -F(w) \\ \nabla\cdot u &= 0 \end{cases}$$

Matrix form

$$\begin{bmatrix}A(w) & B^T\\ B & \end{bmatrix} = \begin{bmatrix}1 & \\ BA^{-1} & 1\end{bmatrix}\begin{bmatrix}A & B^T\\ & S\end{bmatrix} \qquad S = -BA^{-1}B^T$$

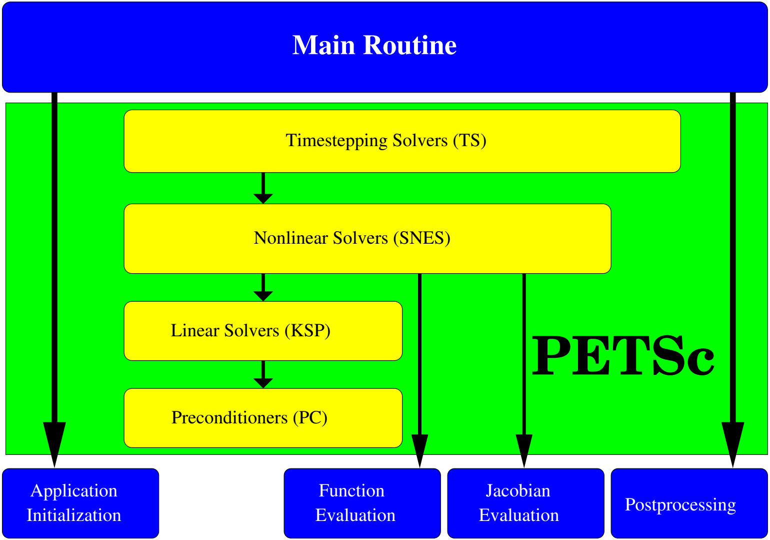
Define $A(w)$ in pressure space

- Want $P_S = (BB^T)A_p^{-1} \approx BA^{-1}B^T, \qquad P_S^{-1} = A_p(BB^T)^{-1}$
- $A_p \sim \rho\Big(\alpha p + w\cdot\nabla p + p\,\mathrm{tr}(\nabla w)\Big) - \eta\nabla^2 p$
- $p\,\mathrm{tr}(\nabla w)$ term is questionable, not needed for Picard
- Almost mesh-independent, weak Reynolds number dependence

(Silvester, Elman, Kay, Wathen. *Efficient preconditioning of the linearized Navier-Stokes equations for*

Outline

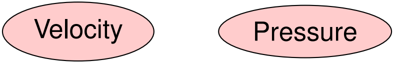
Flow Control for a PETSc Application



Overwhelmed with choices

- If you have a hard problem, no black-box solver will work well
- Everything in PETSc has a plugin architecture
  - Put in the “special sauce” for your problem
  - Your implementations are first-class
- PETSc exposes an algebra of composition at runtime
  - Build a good solver from existing components, at runtime
  - Multigrid, domain decomposition, factorization, relaxation, field-split
  - Choose matrix format that works best with your preconditioner
  - structural blocking, Neumann matrices, monolithic versus nested

Multi-physics coupling in PETSc

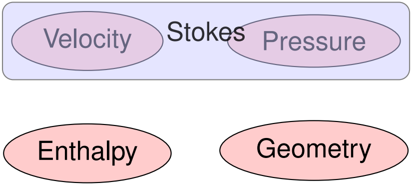


- package each "physics" independently
- solve single-physics and coupled problems
- semi-implicit and fully implicit
- reuse residual and Jacobian evaluation unmodified
- direct solvers and efficient fieldsplit without recompilation
- use the best possible matrix format for each physics (e.g. symmetric block size 3)
- matrix-free anywhere
- multiple levels of nesting

Multi-physics coupling in PETSc

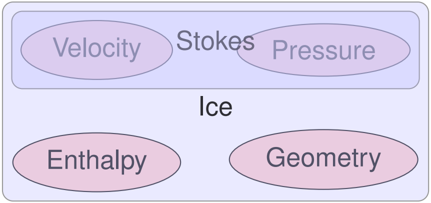
- package each "physics" independently
- solve single-physics and coupled problems
- semi-implicit and fully implicit
- reuse residual and Jacobian evaluation unmodified
- direct solvers and efficient fieldsplit without recompilation
- use the best possible matrix format for each physics (e.g. symmetric block size 3)
- matrix-free anywhere
- multiple levels of nesting

Multi-physics coupling in PETSc



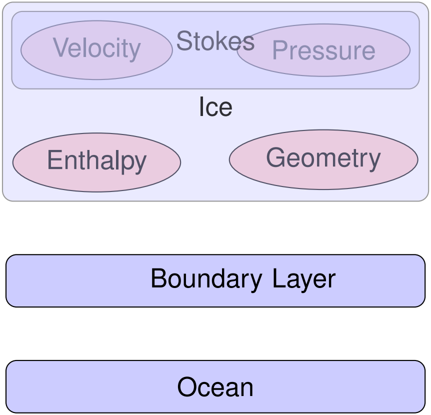
- package each "physics" independently
- solve single-physics and coupled problems
- semi-implicit and fully implicit
- reuse residual and Jacobian evaluation unmodified
- direct solvers and efficient fieldsplit without recompilation
- use the best possible matrix format for each physics (e.g. symmetric block size 3)
- matrix-free anywhere
- multiple levels of nesting

Multi-physics coupling in PETSc



- package each "physics" independently
- solve single-physics and coupled problems
- semi-implicit and fully implicit
- reuse residual and Jacobian evaluation unmodified
- direct solvers and efficient fieldsplit without recompilation
- use the best possible matrix format for each physics (e.g. symmetric block size 3)
- matrix-free anywhere
- multiple levels of nesting

Multi-physics coupling in PETSc



- package each "physics" independently
- solve single-physics and coupled problems
- semi-implicit and fully implicit
- reuse residual and Jacobian evaluation unmodified
- direct solvers and efficient fieldsplit without recompilation
- use the best possible matrix format for each physics (e.g. symmetric block size 3)
- matrix-free anywhere
- multiple levels of nesting

MatNest: a matrix format for field-split

$$\begin{bmatrix} A_{II} & A_{I\Gamma} & & & & \\ & \alpha M_{\Gamma\Gamma} & & -N_{\Gamma\Gamma} & & \\ G_{II} & G_{\Gamma I} & B_{II} & B_{I\Gamma} & C_I^T & D_I \\ G_{I\Gamma} & G_{\Gamma\Gamma} & B_{\Gamma I} & B_{\Gamma\Gamma} & C_\Gamma^T & D_\Gamma \\ G_{Ip} & G_{\Gamma p} & C_I & C_\Gamma & & \\ \alpha E_I & \alpha E_\Gamma & F_I & F_\Gamma & & \alpha M_\Theta + J \end{bmatrix} \begin{bmatrix} x_I \\ x_\Gamma \\ u_I \\ u_\Gamma \\ p \\ \Theta \end{bmatrix}$$

- pseudo-elasticity for mesh motion
- $(\dot{x} - u) \cdot n =$ accumulution
- “just” geometry
- Stokes problem
- temperature dependence of rheology
- ALE and strain heating in heat transport
- thermal advection-diffusion

- Blocks stored separately
  no-copy access
- `MatGetSubMatrix` API
  looks same as “normal” matrices
- Nesting can be recursive
- Implements standard linear algebra operations

`MatGetLocalSubMatrix(Mat A,IS rows,IS cols,Mat *B);`

- Primarily for assembly
  - `B` is not guaranteed to implement `MatMult`
  - The communicator for `B` is not specified, only safe to use non-collective ops (unless you check)
- `IS` represents an index set, includes a block size and communicator
- `MatSetValuesBlockedLocal()` is implemented
- MatNest returns nested submatrix, no-copy
- No-copy for Neumann-Neumann formats (unassembled across procs, e.g. BDDC, FETI-DP)
- Most other matrices return a lightweight proxy `Mat`
  - `COMM_SELF`
  - Values not copied, does not implement `MatMult`
  - Translates indices to the language of the parent matrix
  - Multiple levels of nesting are flattened

Wrap-up

- Software modularity while retaining access to good solvers
  - Reuse single-physics modules
  - Unintrusive “special sauce” (once you figure it out)
- Choose the matrix format at runtime, best for your preconditioner
  - monolithic, nested, Neumann
  - scalar or block, symmetric
- Break into pieces that are “understood”, keep some block structure for high throughput