Towards Algorithmic and Software Composability for Implicit Multiphysics with High Throughput

Jed Brown[1], Matt Knepley[2], Dave May[3], Barry Smith[1]

[1]Mathematics and Computer Science Division, Argonne National Laboratory
[2]Computation Institute, University of Chicago
[3]ETH Zürich

ICES 2012-02-23

Outline

Composable Solvers

Throughput

Multiphysics problems

Examples

- Saddle-point problems (e.g. incompressibility, contact)
- Stiff waves (e.g. low-Mach combustion)
- Mixed type (e.g. radiation hydrodynamics, ALE free-surface flows)
- Multi-domain problems (e.g. fluid-structure interaction)
- Full space PDE-constrained optimization

Software/algorithmic considerations

- Separate groups develop different "physics" components
- Do not know a priori which methods will have good algorithmic properties
- Achieving high throughput is more complicated
- Multiple time and/or spatial scales
  - Splitting methods are delicate, often not in asymptotic regime
  - Strongest nonlinearities usually non-stiff: prefer explicit for TVD limiters/shocks

The Great Solver Schism: Monolithic or Split?

Monolithic

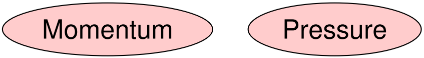
- Direct solvers
- Coupled Schwarz
- Coupled Neumann-Neumann (need unassembled matrices)
- Coupled multigrid
- X Need to understand local spectral and compatibility properties of the coupled system

Split

- Physics-split Schwarz (based on relaxation)
- Physics-split Schur (based on factorization)
  - approximate commutators SIMPLE, PCD, LSC
  - segregated smoothers
  - Augmented Lagrangian
  - "parabolization" for stiff waves
- X Need to understand global coupling strengths

- Preferred data structures depend on which method is used.
- Interplay with geometric multigrid.

Multi-physics coupling in PETSc

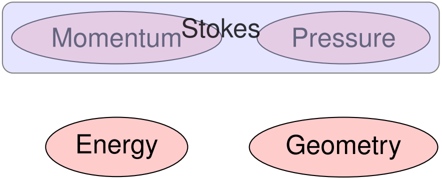


- package each “physics” independently
- solve single-physics and coupled problems
- semi-implicit and fully implicit
- reuse residual and Jacobian evaluation unmodified
- direct solvers, fieldsplit inside multigrid, multigrid inside fieldsplit without recompilation
- use the best possible matrix format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

Multi-physics coupling in PETSc

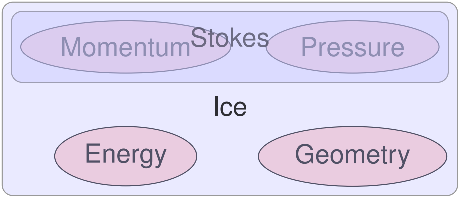
- package each "physics" independently
- solve single-physics and coupled problems
- semi-implicit and fully implicit
- reuse residual and Jacobian evaluation unmodified
- direct solvers, fieldsplit inside multigrid, multigrid inside fieldsplit without recompilation
- use the best possible matrix format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

Multi-physics coupling in PETSc



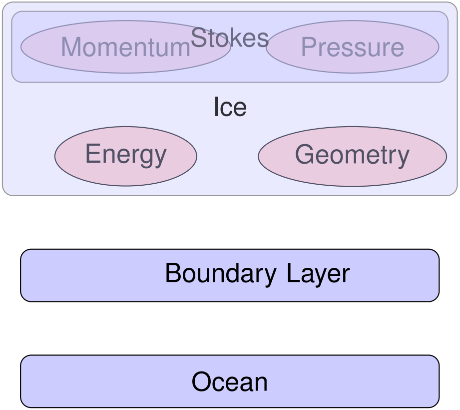
- package each “physics” independently
- solve single-physics and coupled problems
- semi-implicit and fully implicit
- reuse residual and Jacobian evaluation unmodified
- direct solvers, fieldsplit inside multigrid, multigrid inside fieldsplit without recompilation
- use the best possible matrix format 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, fieldsplit inside multigrid, multigrid inside 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, fieldsplit inside multigrid, multigrid inside 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

Splitting for Multiphysics

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

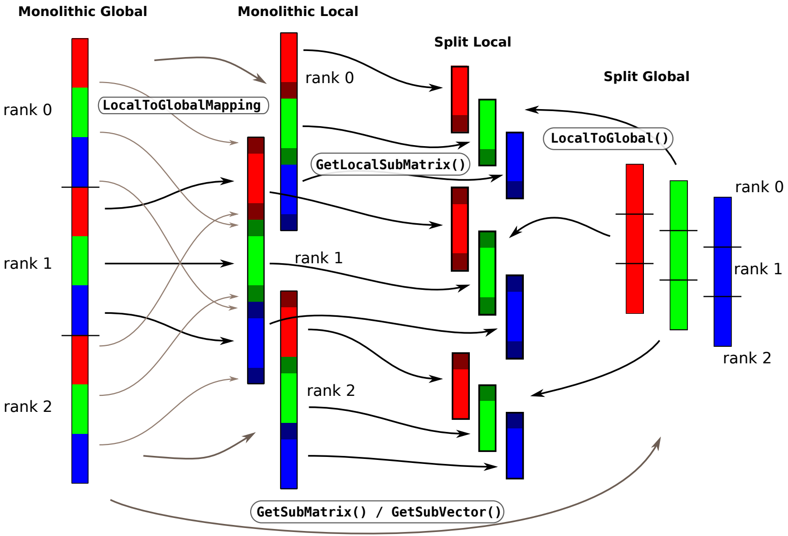
- Relaxation: `-pc_fieldsplit_type [additive,multiplicative,symmetric_multiplicative]`

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

  - 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



Work in Split Local space, matrix data structures reside in any space.

Multiphysics Assembly Code: Residuals

```
FormFunction_Coupled(SNES snes,Vec X,Vec F,void *ctx) {
  struct UserCtx *user = ctx;
  // ...
  SNESGetDM(snes,&pack);
  DMCompositeGetEntries(pack,&dau,&dak);
  DMDAGetLocalInfo(dau,&infou);
  DMDAGetLocalInfo(dak,&infok);
  DMCompositeScatter(pack,X,Uloc,Kloc);
  DMDAVecGetArray(dau,Uloc,&u);
  DMDAVecGetArray(dak,Kloc,&k);
  DMCompositeGetAccess(pack,F,&Fu,&Fk);
  DMDAVecGetArray(dau,Fu,&fu);
  DMDAVecGetArray(dak,Fk,&fk);
  FormFunctionLocal_U(user,&infou,u,k,fu); // u residual with k given
  FormFunctionLocal_K(user,&infok,u,k,fk); // k residual with u given
  DMDAVecRestoreArray(dau,Fu,&fu);
  // More restores
```

Multiphysics Assembly Code: Jacobians

```
FormJacobian_Coupled(SNES snes,Vec X,Mat *J,Mat *B,...) {
  // Access components as for residuals
  MatGetLocalSubMatrix(*B,is[0],is[0],&Buu);
  MatGetLocalSubMatrix(*B,is[0],is[1],&Buk);
  MatGetLocalSubMatrix(*B,is[1],is[0],&Bku);
  MatGetLocalSubMatrix(*B,is[1],is[1],&Bkk);
  FormJacobianLocal_U(user,&infou,u,k,Buu);          // single physics
  FormJacobianLocal_UK(user,&infou,&infok,u,k,Buk); // coupling
  FormJacobianLocal_KU(user,&infou,&infok,u,k,Bku); // coupling
  FormJacobianLocal_K(user,&infok,u,k,Bkk);          // single physics
  MatRestoreLocalSubMatrix(*B,is[0],is[0],&Buu);
  // More restores
```

- Assembly code is independent of matrix format
- Single-physics code is used unmodified for coupled problem
- No-copy fieldsplit:

  `-pack_dm_mat_type nest -pc_type fieldsplit`
- Coupled direct solve:

  `-pack_dm_mat_type aij -pc_type lu -pc_factor_mat_solver_package mumps`

`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

Monolithic nonlinear solvers

Coupled nonlinear multigrid accelerated by NGMRES with multi-stage smoothers

```
-lidvelocity 200 -grashof 1e4
-snes_grid_sequence 5 -snes_monitor -snes_view
-snes_type ngmres
-npc_snes_type fas
-npc_snes_max_it 1
-npc_fas_coarse_snes_type ls
-npc_fas_coarse_ksp_type preonly
-npc_fas_snes_type ms
-npc_fas_snes_ms_type vltp61
-npc_fas_snes_max_it 1
-npc_fas_ksp_type preonly
-npc_fas_pc_type pbjacobi
-npc_fas_snes_max_it 1
```

- Uses only residuals and point-block diagonal
- High arithmetic intensity and parallelism

Nonlinear solvers in PETSc SNES

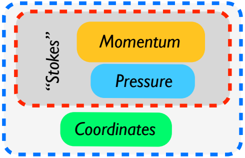
- **LS, TR** Newton-type with line search and trust region
- **NRichardson** Nonlinear Richardson, usually preconditioned
- **VIRS, VIRSAUG, and VISS** reduced space and semi-smooth methods for variational inequalities
- **QN** Quasi-Newton methods like BFGS
- **NGMRES** Nonlinear GMRES
- **NCG** Nonlinear Conjugate Gradients
- **SORQN** SOR quasi-Newton
- **GS** Nonlinear Gauss-Seidel sweeps
- **FAS** Full approximation scheme (nonlinear multigrid)
- **MS** Multi-stage smoothers, often used with FAS for hyperbolic problems
- **Shell** Your method, often used as a (nonlinear) preconditioner

Stokes + Implicit Free Surface

$$\left[\eta D_{ij}(\boldsymbol{u})\right]_{,j} - p_{,i} = f_i$$

$$u_{k,k} = 0$$

$$\hat{x}_i = \hat{x}_i^{t-\Delta t} + \Delta t \, u_i(\hat{x}_i)$$



COORDINATE RESIDUALS

$$F_x := -u_i + \frac{\hat{x}_i}{\Delta t} - \frac{\hat{x}_i^{t-\Delta t}}{\Delta t}$$

[We use a full Lagrangian update of our mesh, with no remeshing]

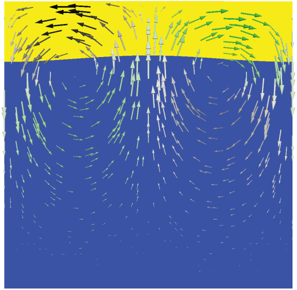
JACOBIAN

$$\mathcal{J}_{si} = \begin{bmatrix} A + \delta_{\hat{x}} A & B + \delta_{\hat{x}} B & J_{ac} \\ B^T + \delta_{\hat{x}} B^T & 0 & J_{bc} \\ -I & 0 & \frac{I}{\Delta t} \end{bmatrix}$$

Reuse stokes operators and saddle point preconditioners

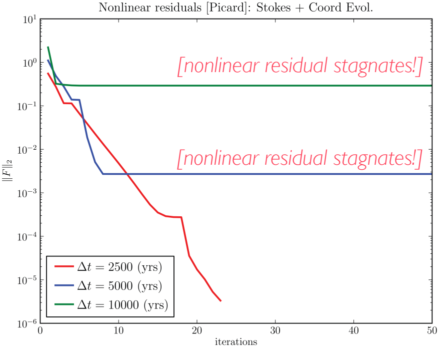
NESTED PRECONDITIONER

$$\mathcal{P}_{si} = \begin{bmatrix} [\ \mathcal{P}_s^l\ ] & \\ [\ I\ ] & [\ -\frac{I}{\Delta t}\ ] \end{bmatrix} \qquad \mathcal{P}_s^l = \begin{bmatrix} A & 0 \\ B^T & -S \end{bmatrix}$$

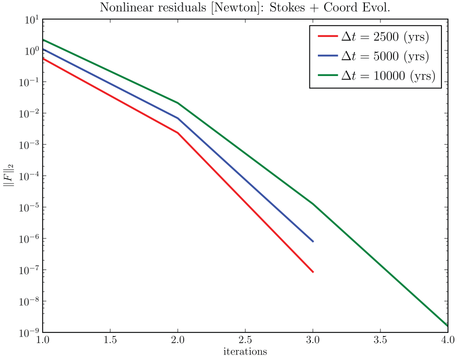


"Drunken seaman", Rayleigh Taylor instability test case from Kaus et al., 2010. Dense, viscous material (yellow) overlying less dense, less viscous material (blue).

Stokes + Implicit Free Surface

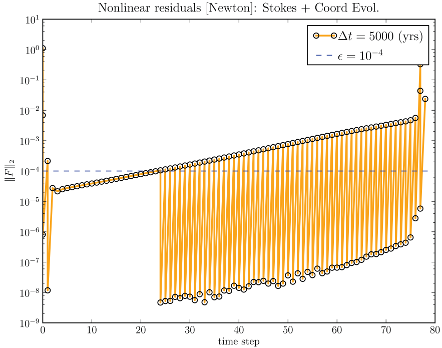


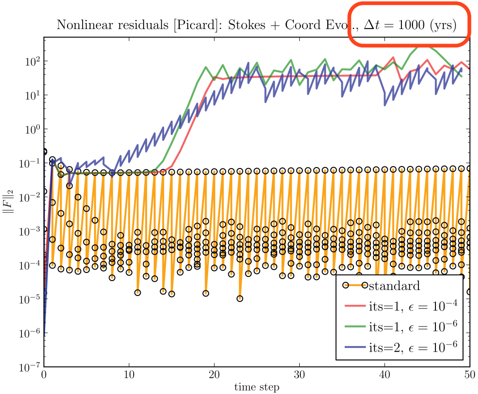
* Picard fails to converge for large time steps sizes.



* Newton is robust for a wide range of time step sizes.

Stokes + Implicit Free Surface





* The nonlinear residual *ALWAYS* increases from one step to the next.
* A nonlinear solve is required to control the error.

* An accurate nonlinear solve on the first time step, combined with 1 or 2 nonlinear iterations on subsequent steps still results in severe errors. *This is true even when dt is small.*

Conservative (non-Boussinesq) two-phase ice flow

Find momentum density ρu, pressure p, and total energy density E:

$$(\rho u)_t + \nabla\cdot(\rho u \otimes u - \eta D u_i + p1) - \rho g = 0$$
$$\rho_t + \nabla\cdot \rho u = 0$$
$$E_t + \nabla\cdot\big((E+p)u - k_T \nabla T - k_\omega \nabla \omega\big) - \eta D u_i : D u_i - \rho u \cdot g = 0$$

- Solve for density ρ, ice velocity u_i, temperature T, and melt fraction ω using constitutive relations.
  - Simplified constitutive relations can be solved explicitly.
  - Temperature, moisture, and strain-rate dependent rheology η.
  - High order FEM, typically Q_3 momentum & energy
- DAEs solved implicitly after semidiscretizing in space.
- Preconditioning using nested fieldsplit

Relative effect of the blocks

$$J = \begin{pmatrix} J_{uu} & J_{up} & J_{uE} \\ J_{pu} & 0 & 0 \\ J_{Eu} & J_{Ep} & J_{EE} \end{pmatrix}.$$

- J_{uu} Viscous/momentum terms, nearly symmetric, variable coefficionts, anisotropy from Newton.
- J_{up} Weak pressure gradient, viscosity dependence on pressure (small), gravitational contribution (pressure-induced density variation). Large, nearly balanced by gravitational forcing.
- J_{uE} Viscous dependence on energy, very nonlinear, not very large.
- J_{pu} Divergence (mass conservation), nearly equal to J_{up}^T.
- J_{Eu} Sensitivity of energy on momentum, mostly advective transport. Large in boundary layers with large thermal/moisture gradients.
- J_{Ep} Thermal/moisture diffusion due to pressure-melting, $u \cdot \nabla$.
- J_{EE} Advection-diffusion for energy, very nonlinear at small regularization. Advection-dominated except in boundary layers and stagnant ice, often balanced in vertical.

How much nesting?

$$P_1 = \begin{pmatrix} J_{uu} & J_{up} & J_{uE} \\ 0 & B_{pp} & 0 \\ 0 & 0 & J_{EE} \end{pmatrix}$$

- B_{pp} is a mass matrix in the pressure space weighted by inverse of kinematic viscosity.
- Elman, Mihajlović, Wathen, JCP 2011 for non-dimensional isoviscous Boussinesq.
- Works well for non-dimensional problems on the cube, not for realistic parameters.

$$P = \begin{bmatrix} \begin{pmatrix} J_{uu} & J_{up} \\ J_{pu} & 0 \end{pmatrix} & \\ \begin{pmatrix} J_{Eu} & J_{Ep} \end{pmatrix} & J_{EE} \end{bmatrix}$$

- Inexact inner solve using upper-triangular with B_{pp} for Schur.
- Another level of nesting.
- GCR tolerant of inexact inner solves.
- Outer converges in 1 or 2 iterations.

- Low-order preconditioning full-accuracy unassembled high order operator.
- Build these on command line with PETSc `PCFieldSplit`.

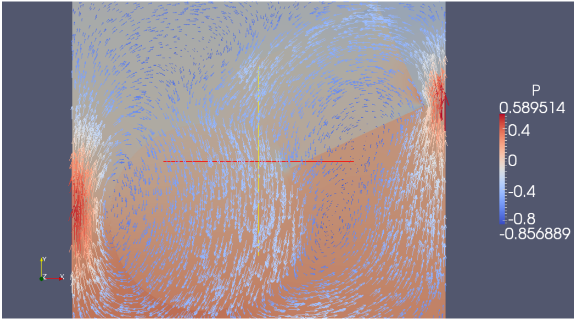
Example 3×3 problem with nested 2×2 split

```
-fieldsplit_s_ksp_type gcr
-fieldsplit_s_ksp_rtol 1e-1
-fieldsplit_s_ksp_monitor_vht
-fieldsplit_s_ksp_monitor_singular_value
-fieldsplit_s_pc_type fieldsplit
-fieldsplit_s_pc_fieldsplit_type schur
-fieldsplit_s_pc_fieldsplit_real_diagonal
-fieldsplit_s_pc_fieldsplit_schur_factorization_type lower
-fieldsplit_s_fieldsplit_u_ksp_type gmres
-fieldsplit_s_fieldsplit_u_ksp_max_it 10
-fieldsplit_s_fieldsplit_u_pc_type asm
-fieldsplit_s_fieldsplit_u_sub_pc_type ilu
-fieldsplit_s_fieldsplit_u_sub_pc_factor_levels 1
-fieldsplit_s_fieldsplit_u_ksp_converged_reason
-fieldsplit_s_fieldsplit_p_ksp_type preonly
-fieldsplit_s_fieldsplit_p_ksp_max_it 1
-fieldsplit_s_fieldsplit_p_pc_type jacobi
-fieldsplit_e_ksp_type gmres
-fieldsplit_e_ksp_converged_reason
-fieldsplit_e_pc_type asm
-fieldsplit_e_sub_pc_type ilu
-fieldsplit_e_sub_pc_factor_levels 2
```

Coupled MG for Stokes, split smoothers

$$J = \begin{pmatrix} A & B^T \\ B & C \end{pmatrix}$$

$$P_{\text{smooth}} = \begin{pmatrix} A_{\text{SOR}} & 0 \\ B & M \end{pmatrix}$$




```
-pc_type mg -pc_mg_levels 5 -pc_mg_galerkin
-mg_levels_pc_type fieldsplit
-mg_levels_pc_fieldsplit_block_size 3
-mg_levels_pc_fieldsplit_0_fields 0,1
-mg_levels_pc_fieldsplit_1_fields 2
-mg_levels_fieldsplit_0_pc_type sor
```

Phase field models

State variables $u = (u_1, \dots, u_N)^T$ are concentrations of different phases satisfying the inequality and sum constraints

$$u(x,t) \in G = \{v \in \mathbb{R}^d | v_i \geq 0, \sum_{i=1}^{N} v_i = 1\}, \quad \forall (x,t) \in Q.$$

Minimize free energy, reduced space active set method

$$J = \begin{pmatrix} A & 0 & 0 & -I \\ 0 & A & 0 & -I \\ 0 & 0 & A & -I \\ -I & -I & -I & 0 \end{pmatrix}, \qquad P = \begin{pmatrix} A & 0 & 0 & 0 \\ 0 & A & 0 & 0 \\ 0 & 0 & A & 0 \\ -I & -I & -I & S_{\mathrm{LSC}} \end{pmatrix}$$

```
-ksp_type fgmres -pc_type fieldsplit
-pc_fieldsplit_detect_saddle_point
-pc_fieldsplit_type schur
-pc_fieldsplit_schur_precondition self
-fieldsplit_0_ksp_type preonly
-fieldsplit_0_pc_type hypre
-fieldsplit_1_ksp_type fgmres
-fieldsplit_1_pc_type lsc
```

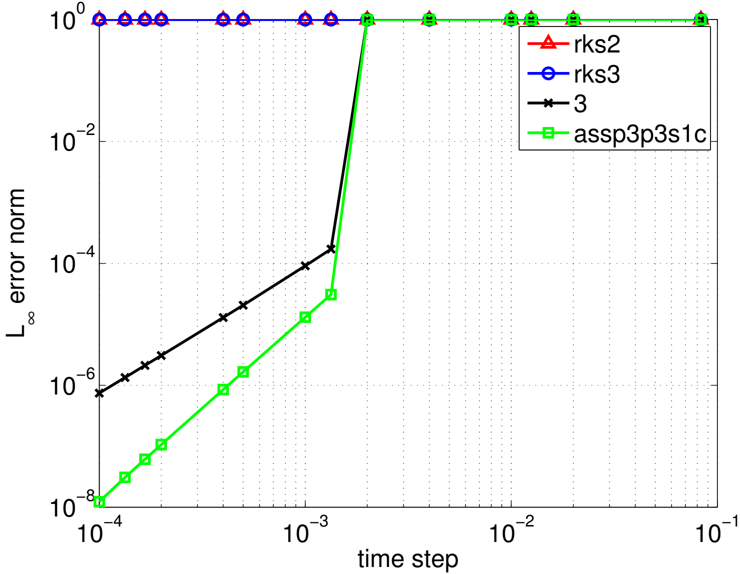
IMEX time integration in PETSc

- Additive Runge-Kutta IMEX methods

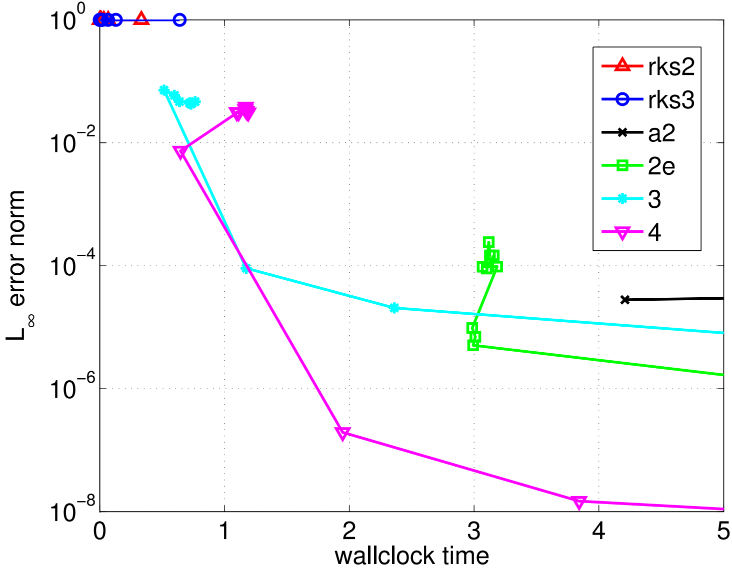
$$G(t,x,\dot{x}) = F(t,x)$$
$$J_\alpha = \alpha G_{\dot{x}} + G_x$$

  - User provides:
    - `FormRHSFunction(ts,`t`,`x`,`F`,void *ctx);`
    - `FormIFunction(ts,`t`,`x`,`$\dot{x}$`,`G`,void *ctx);`
    - `FormIJacobian(ts,`t`,`x`,`$\dot{x}$`,`α`,`J`,`J_p`,mstr,void *ctx);`
  - L-stable DIRK for stiff part G
  - Choice of explicit method, e.g. SSP
  - Orders 2 through 5, embedded error estimates
  - Dense output, hot starts for Newton
  - More accurate methods if G is linear, also Rosenbrock-W
  - Can use preconditioner from classical “semi-implicit” methods
  - Extensible adaptive controllers, can change order within a family
  - Easy to register new methods: `TSARKIMEXRegister()`
- Eliminate many interface quirks
- Single step interface so user can have own time loop

Stiff advection-reaction accuracy



Stiff advection-reaction efficiency



Outlook

- Unintrusive composition of multigrid and block preconditioning
- We can build many preconditioners from the literature *on the command line*
- User code does not depend on matrix format, preconditioning method, nonlinear solution method, time integration method (implicit or IMEX), or size of coupled system (except for driver).

In development

- Distributive relaxation, Vanka smoothers
- Algebraic coarsening of “dual” variables
- Improving operator-dependent semi-geometric multigrid
- More automatic spectral analysis and smoother optimization
- Better interaction with IMEX time integration
    - Additive Runge-Kutta, Rosenbrock-W, linear multistep
    - Composability with FAS
    - Parallel-in-time approaches

Outline

The Roadmap

Hardware trends

- More cores (keep hearing $\mathcal{O}(1000)$ per node)
- Long vector registers (already 32 bytes for AVX and BG/Q)
- Must use SMT to hide memory latency
- Must use SMT for floating point performance (GPU, BG/Q)
- Large penalty for non-contiguous memory access

“Free flops”, but how can we use them?

- High order methods good: better accuracy per storage
- High order methods bad: work unit gets larger
- GPU threads have very little memory, must keep work unit small
- Need library composability, keep user contribution embarrassingly parallel

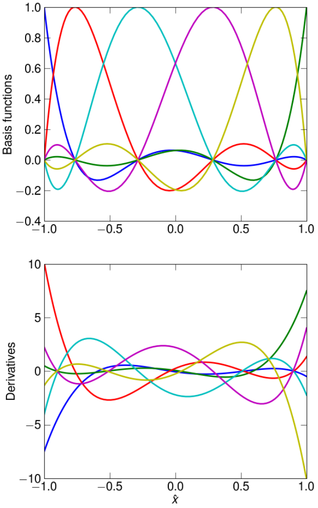
How to program this beast?

- Decouple physics from discretization
  - Expose small, embarrassingly parallel operations to user
  - Library schedules user threads for reuse between kernels
  - User provides physics in kernels run at each quadrature point
  - Continuous weak form: find $u \in \mathcal{V}_D$

$$v^T F(u) \sim \int_\Omega v \cdot f_0(u, \nabla u) + \nabla v : f_1(u, \nabla u) = 0, \qquad \forall v \in \mathcal{V}_0$$

  - Similar form at faces, but may involve Riemann solve
- Library manages reductions
  - Interpolation and differentiation on elements
  - Interaction with neighbors (limiting, edge stabilization)
  - Exploit tensor product structure to keep working set small
  - Assembly into solution/residual vector (sum over elements)

Nodal hp-version finite element methods



1D reference element

- Lagrange interpolants on Legendre-Gauss-Lobatto points
- Quadrature $\hat{R}$, weights $\hat{W}$
- Evaluation: $\hat{B}, \hat{D}$

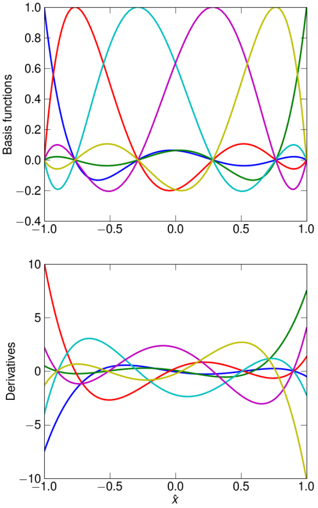
3D reference element

$$\hat{W} = \hat{W} \otimes \hat{W} \otimes \hat{W}$$
$$\hat{B} = \hat{B} \otimes \hat{B} \otimes \hat{B}$$

$$\hat{D}_0 = \hat{D} \otimes \hat{B} \otimes \hat{B}$$
$$\hat{D}_1 = \hat{B} \otimes \hat{D} \otimes \hat{B}$$
$$\hat{D}_2 = \hat{B} \otimes \hat{B} \otimes \hat{D}$$

These tensor product operations are very efficient, 70% of peak flop/s

Nodal *hp*-version finite element methods



1D reference element

- Lagrange interpolants on Legendre-Gauss-Lobatto points
- Quadrature $\hat{R}$, weights $\hat{W}$
- Evaluation: $\hat{B}, \hat{D}$

3D reference element

$$\hat{W} = \hat{W} \otimes \hat{W} \otimes \hat{W}$$
$$\hat{B} = \hat{B} \otimes \hat{B} \otimes \hat{B}$$

$$\hat{D}_0 = \hat{D} \otimes \hat{B} \otimes \hat{B}$$
$$\hat{D}_1 = \hat{B} \otimes \hat{D} \otimes \hat{B}$$
$$\hat{D}_2 = \hat{B} \otimes \hat{B} \otimes \hat{D}$$

These tensor product operations are very efficient, 70% of peak flop/s

Operations on physical elements

Mapping to physical space

$$x^e : \hat{K} \to K^e, \quad J^e_{ij} = \partial x^e_i / \partial \hat{x}_j, \quad (J^e)^{-1} = \partial \hat{x} / \partial x^e$$

Element operations in physical space

$$B^e = \hat{B} \qquad\qquad W^e = \hat{W} \Lambda(|J^e(r)|)$$

$$D^e_i = \Lambda\left(\frac{\partial \hat{x}_0}{\partial x_i}\right)\hat{D}_0 + \Lambda\left(\frac{\partial \hat{x}_1}{\partial x_i}\right)\hat{D}_1 + \Lambda\left(\frac{\partial \hat{x}_2}{\partial x_i}\right)\hat{D}_2$$

$$(D^e_i)^T = \hat{D}^T_0 \Lambda\left(\frac{\partial \hat{x}_0}{\partial x_i}\right) + \hat{D}^T_1 \Lambda\left(\frac{\partial \hat{x}_1}{\partial x_i}\right) + \hat{D}^T_2 \Lambda\left(\frac{\partial \hat{x}_2}{\partial x_i}\right)$$

Global problem is defined by assembly

$$F(u) = \sum_e \mathcal{E}^T_e \left[(B^e)^T W^e \Lambda(f_0(u^e, \nabla u^e)) + \sum_{i=0}^{d} (D^e_i)^T W^e \Lambda(f_{1,i}(u^e, \nabla u^e)) \right] = 0$$

where $u^e = B^e \mathcal{E}^e u$ and $\nabla u^e = \{D^e_i \mathcal{E}^e u\}_{i=0}^2$

Representation of Jacobians, Automation

- For unassembled representations, decomposition, and assembly
- Continuous weak form: find u

$$v^T F(u) \sim \int_\Omega v \cdot f_0(u, \nabla u) + \nabla v : f_1(u, \nabla u) = 0, \qquad \forall v \in \mathcal{V}_0$$

- Weak form of the Jacobian $J(u)$: find w

$$v^T J(u) w \sim \int_\Omega \begin{bmatrix} v^T & \nabla v^T \end{bmatrix} \begin{bmatrix} f_{0,0} & f_{0,1} \\ f_{1,0} & f_{1,1} \end{bmatrix} \begin{bmatrix} w \\ \nabla w \end{bmatrix}$$

$$[f_{i,j}] = \begin{bmatrix} \frac{\partial f_0}{\partial u} & \frac{\partial f_0}{\partial \nabla u} \\ \frac{\partial f_1}{\partial u} & \frac{\partial f_1}{\partial \nabla u} \end{bmatrix} (u, \nabla u)$$

- Terms in $[f_{i,j}]$ easy to compute symbolically, AD more scalable.
- Nonlinear terms f_0, f_1 usually have the most expensive nonlinearities in the computation of scalar material parameters
  - Equations of state, effective viscosity, “star” region in Riemann solve
  - Compute gradient with reverse-mode, store at quadrature points.
  - Perturb scalars, then use forward-mode to complete the Jacobian.
  - Flip for action of the adjoint.

CPU traversal code

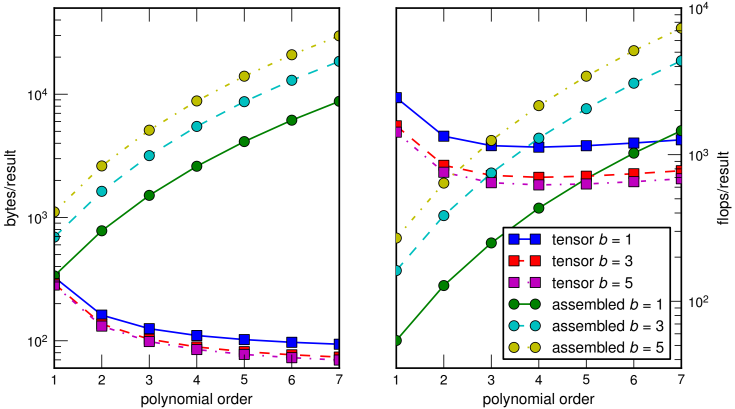
- CPU traversal computes coefficients of test functions, `https://github.com/jedbrown/dohp/`

```
while (IteratorHasPatch(iter)) {
  IteratorGetPatchApplied(iter,&Q,&jw,
      &x,&dx,NULL,NULL,
      &u,&du,&u_,&du_, &p,&dp,&p_,NULL, &e,&de,&e_,&de_);
  IteratorGetStash(iter,NULL,&stash);
  for (dInt i=0; i<Q; i++) {
    PointwiseFunction(context,x[i],dx[i],jw[i],
        u[i],du[i],p[i],dp[i],e[i],de[i],
        &stash[i], u_[i],du_[i],p_[i],e_[i],de_[i]);
  }
  IteratorCommitPatchApplied(iter,INSERT_VALUES, NULL,NULL,
                             u_,du_, p_,NULL, e_,de_);
  IteratorNextPatch(iter);
}
```

- GPU version calls `PointwiseFunction()` directly.
- Unassembled Jacobian application reuses `stash`

```
PointwiseJacobian(context,&stash[i],dx[i],jw[i],
                  u[i],du[i],p[i],dp[i],e[i],de[i],
                  u_[i],du_[i],p_[i],e_[i],de_[i]);
```

Performance of assembled versus unassembled



- High order Jacobian stored unassembled using coefficients at quadrature points, can use local AD
- Choose approximation order at run-time, independent for each field
- Precondition high order using assembled lowest order method
- Implementation $> 70\%$ of FPU peak, SpMV bandwidth wall $< 4\%$

Hardware Arithmetic Intensity

Operation	Arithmetic Intensity (flops/B)
Sparse matrix-vector product	1/6
Dense matrix-vector product	1/4
Unassembled matrix-vector product	≈ 8
High-order residual evaluation	> 5

Processor	BW (GB/s)	Peak (GF/s)	Balanced AI (F/B)
Sandy Bridge 6-core	21*	150	7.2
Magny Cours 16-core	42*	281	6.7
Blue Gene/Q node	43	205	4.8
Tesla M2050	144	515	3.6

Finer grained parallelism for GPU FEM

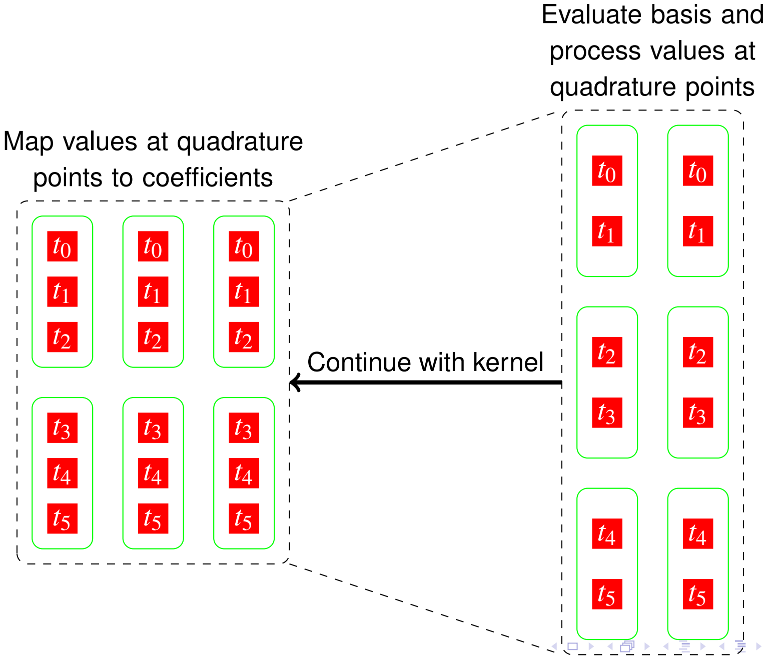
- One element per thread uses too much local memory.
- Would like to use *about* one quadrature point per thread.
- Tensor product requires several synchronizations

$$\begin{aligned}\tilde{u} &= (A \otimes B \otimes C)u \\ &= (A \otimes I \otimes I)(I \otimes B \otimes I)(I \otimes I \otimes C)u\end{aligned}$$

- Accumulation easy if only one thread accumulates into a location.
- Threads within a warp are implicitly synchronized, no need for `__syncthreads()`.
- Synchronization scope depends on approx order

Element	# quad pts	32T warps/element	TB size
Q_1	8	1/4	any
Q_2	27	1 (5T unused)	any
Q_3	64	2	64
Q_4	125	4 (3T unused)	128

Finer grained parallelism for GPUs, low order



On preconditioning and multigrid

- Currently using assembled matrices for preconditioning
- Want matrix-free preconditioners for high hardware utilization
- Geometric h- and p-multigrid, could be FAS
- Smoothers build/solve with small dense matrices
  - "point" matrices: can use single threads
  - "element" matrices: need to cooperate within thread blocks
  - I want a dense linear algebra library to be called collectively within a thread block
- Multiplicative (Gauss-Seidel) is algorithmically nice
- Spectral analysis for polynomial/multi-stage smoothers
- Coarser levels better to do on CPU
  - Potential for additive correction to run concurrently

Outlook

- Sparse matrix assembly (for preconditioning)
  - > 100 GF/s for lowest order Stokes (Matt Knepley)
  - common “pointwise” physics code with CPU implementation
  - Dohp CPU version faster than libMesh and Deal.II for Q_1
  - Q_1 assembly embedded in higher order is 8% slower than hand-rolled
- Matrix-free tensor-product versions reliably get about 70% of peak flops
- Finer grained parallelism in GPU tensor product kernels
- Can’t wait for OpenCL to implement indirect function calls
- Symbolic differentiation too slow, tired of hand-differentiation
  - I want source-transformation AD with indirect function calls
- Find correct amount of reuse between face and cell integration
- Riemann solves harder to vectorize
- Hide dispatch to pointwise kernels inside library
  - Easy, but scary. Library/framework becomes **F**ramework.
  - Interoperbility of user-rolled, library-provided, and generated traversal code.