An Introduction to PETSc/TAO by Example

Presented in two parts:
I. Solving nonlinear equations with the Scalable Nonlinear Equation Solvers (SNES)
II. A survey of applications of PETSc/TAO to different types of "physics"

New User Tutorial at the PETSC Annual Meeting

Richard Tran Mills, Argonne National Laboratory
Matthew Knepley, University at Buffalo
June 5, 2023

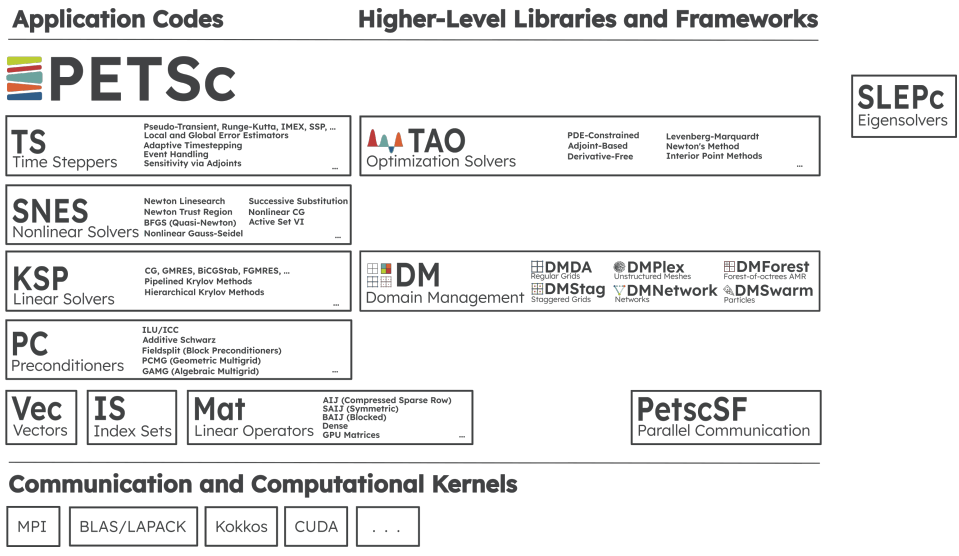
Goals and Agenda

- Introduce several key concepts and common patterns in PETSc by examples and demonstrations using the *Scalable Nonlinear Equation Solvers* (SNES) component:
  - Using PETSc runtime options to query what is happening, set problem parameters, specify solver options, and build (and experiment with!) sophisticated composite solvers on the command line
  - The user callback paradigm common to many PETSc components
  - How PETSc `DM` objects help manage mesh-like objects and their interactions with algebraic solvers
  - How PETSc uses GPU accelerators
  - How to use the built-in logging framework to understand and tune performance
- Survey various examples of using PETSc to solve problems with different kinds of "physics"

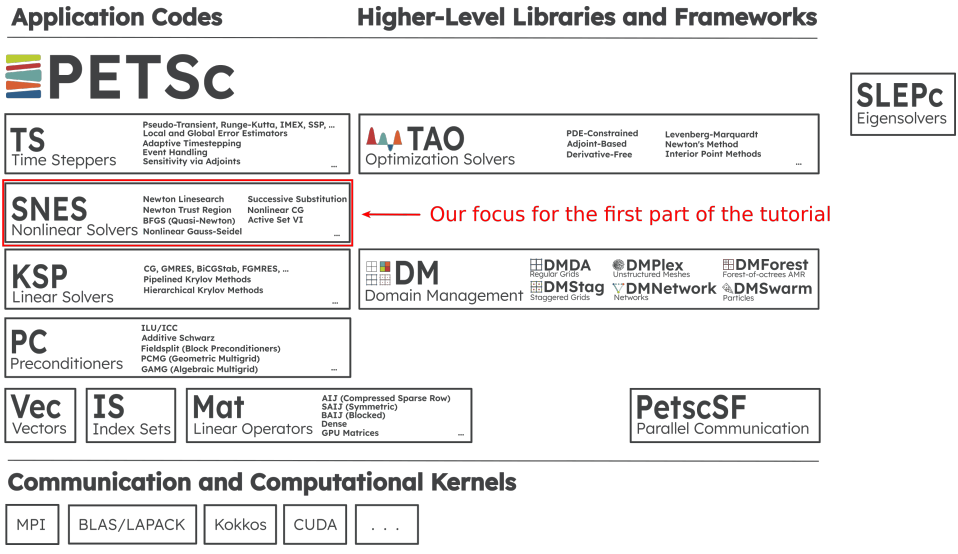
Getting Help (with installation, examples, anything else)

- Due to time constraints, we will not cover PETSc installation.
  - Find tutorials and other information on this at `https://petsc.org/release/install/`.
- For help via email:
  - Email `petsc-maint@mcs.anl.gov` for help from PETSc maintainers (messages to list are not public).
  - Email `petsc-users@mcs.anl.gov` for help from the broad PETSc community (many experienced users may have encountered your issues).
- For help at the meeting:
  - Get the attention of one of the many PETSc maintainers and experienced users here — we want to interact with you!
  - Try the `#petsc-meeting-2023` Slack channel.

PETSc - the Portable, Extensible Toolkit for Scientific computation



PETSc - the Portable, Extensible Toolkit for Scientific computation



Iterative Solvers for Nonlinear Systems

Systems of nonlinear equations

$$F(x) = b \quad \text{where} \quad F : \mathbb{R}^N \to \mathbb{R}^N \tag{1}$$

arise in countless settings in computational science.
Direct methods for general nonlinear systems do not exist. Iterative methods are required!

Nonlinear Richardson (simple) iteration:

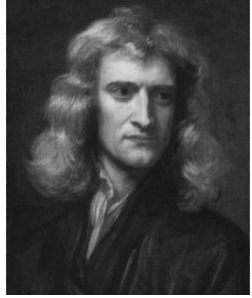
$$x_{k+1} = x_k + \lambda(b - F(x_k)) \tag{2}$$

This has linear convergence at best: $\|e_{k+1}\| \leq C\|e_k\|$

Nonlinear Krylov methods

Nonlinear CG - Mimic CG to force each new search direction to be orthogonal to previous directions.
Nonlinear GMRES (Anderson mixing) - minimize $\|F(x_{k+1} - b)\|$ by using x_{k+1} as a linear combination of previous solutions and solving a linear least squares problem.
These have superlinear convergence at best: $\|e_{k+1}\| \leq C\|e_k\|^{\alpha \geq 1}$

Newton's Method: The workhorse of nonlinear solvers



- Standard form of a nonlinear system

$$F(u) = 0$$

- Iteration

$$\begin{aligned} \text{Solve:} \quad & J(u)w = -F(u) \\ \text{Update:} \quad & u^+ \leftarrow u + w \end{aligned}$$

  Where the Jacobian $J(u) = F'(u) = \frac{\partial F(u)}{\partial u}$.

- Quadratically convergent near a root: $\left|u^{n+1} - u^*\right| \in \mathcal{O}\left(\left|u^n - u^*\right|^2\right)$

Example (Nonlinear Poisson)

$$\begin{aligned} F(u) = 0 \quad &\sim \quad -\nabla\cdot\left[(1+u^2)\nabla u\right] - f = 0 \\ J(u)w \quad &\sim \quad -\nabla\cdot\left[(1+u^2)\nabla w + 2uw\nabla u\right] \end{aligned}$$

Inexact Newton; Newton-Krylov

In practice, incurring the expense of an exact solve for the Newton step is often not desirable:

Inexact Newton methods find an approximate Newton direction Δx_k that satisfies

$$\|F'(x_k)\Delta x_k + F(x_k)\| \leq \eta \|F(x_k)\| \tag{3}$$

for a forcing term $\eta \in [0, 1)$ (static or chosen adaptively via Eisenstat-Walker method, `-snes_ksp_ew`).

Newton-Krylov methods, which use Krylov subspace projection methods as the "inner", linear iterative solver, are a robust and widely-used variant.

PETSc provides a wide range of Krylov methods and linear preconditioners that can be accessed via runtime options (`-ksp_type <ksp_method> -pc_type <pc_method>`).

Globalization Strategies

Newton has quadratic convergence *only when the iterate is sufficiently close to the solution*.
Far from the solution, the computed Newton step is often too large in magnitude.

In practice, some globalization strategy is often needed to expand the domain of convergence.
PETSc offers several options; most common (and default) is backtracking line search.

Backtracking line search

- Replaces the full Newton step s with some scalar multiple: $x_{k+1} = x_k + \lambda_k s, \quad \lambda > 0$
- Introduce merit function $\phi(x) = \frac{1}{2}\|F(x)\|_2^2$, (approximately) find $\min_{\lambda>0} \phi(x_k + \lambda s)$
- Accurate minimization not worth the expense; simply ensure sufficent decrease:

$$\phi(x_k + \lambda s) \leq \phi(x_k) + \alpha \lambda \, s^\top \nabla \phi(x_k) \tag{4}$$

  (α is a user-tunable parameter; defaults to `1e-4`)
- Builds polynomial model for $\phi(x_k + \lambda s)$ (default is cubic; change via `-snes_linesearch_order <n>`).

Specifying the SNES Problem: the User Callback Function Paradigm in PETSc

The PETSc SNES (Scalable Nonlinear Equation Solvers) interface is based upon callback functions

- `FormFunction()`, set by `SNESSetFunction()`
- `FormJacobian()`, set by `SNESSetJacobian()`

When PETSc needs to evaluate the nonlinear residual $F(x)$,

- Solver calls the **user's** function
- User function gets application state through the `ctx` variable
  - PETSc *never* sees application data

This user callback paradigm is followed by many other PETSc components (as we'll see later), and does not change even when doing things like using GPU accelerators.

SNES Function

The user provided function which calculates the nonlinear residual has signature

```
PetscErrorCode (*func)(SNES snes,Vec x,Vec r,void *ctx)
```

`x`: The current solution

`r`: The residual

`ctx`: The user context passed to `SNESSetFunction()`

- Use this to pass application information, e.g. physical constants

SNES Jacobian

The user provided function that calculates the Jacobian has signature

```
PetscErrorCode (*func)(SNES snes,Vec x,Mat J,
                       Mat Jpre,void *ctx)
```

`x`: The current solution

`J`: The Jacobian

`Jpre`: The Jacobian preconditioning matrix (possibly J itself)

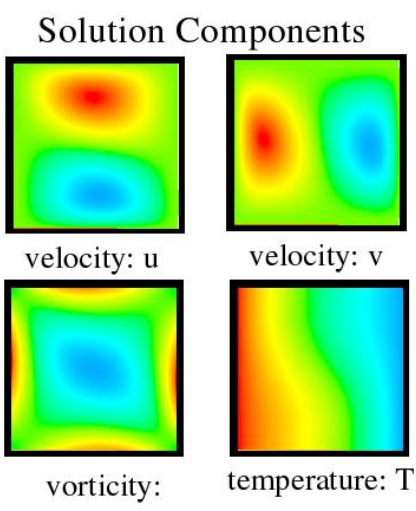
`ctx`: The user context passed to `SNESSetFunction()`

- Use this to pass application information, e.g. physical constants

Alternatively, you can use

- a builtin sparse finite difference approximation ("coloring")

SNES Example ex19, Steady-State Nonisothermal Lid-Driven Cavity



$$-\Delta U - \partial_y \Omega = 0$$
$$-\Delta V + \partial_x \Omega = 0$$
$$-\Delta \Omega + \nabla \cdot ([U\Omega, V\Omega]) - \mathrm{Gr}\ \partial_x T = 0$$
$$-\Delta T + \mathrm{Pr}\ \nabla \cdot ([UT, VT]) = 0$$

- 2D square domain with a moving lid; nonisothermal (temperature T)
- Flow driven by lid motion and buoyancy effects
- Velocity(U, V)-vorticity(Ω) formulation
- Finite difference discretization
- Logically regular grid
  - Parallelized with `DMDA`
- Analytical Jacobian not provided; Calculated by finite-differences (using coloring)
- Contributed by David Keyes
- Run as part of `make check`

`src/snes/tutorials/ex19.c`

SNES Example

Driven Cavity Application Context

```
/*
   User-defined data structures
*/

/* Collocated at each node */
typedef struct {
  PetscScalar u,v,omega,temp;
} Field;

typedef struct {
   PetscReal lidvelocity,prandtl,grashof;  /* physical parameters */
   PetscBool  draw_contours;               /* flag - 1 indicates drawing countours */
} AppCtx;
```

SNES Function with local evaluation

```
PetscErrorCode FormFunctionLocal(DMDALocalInfo *info,Field **x,Field **f,void *ptr)
{
  ...
  xints = info->xs; xinte = info->xs+info->xm; yints = info->ys; yinte = info->ys+info->ym;
  /* Handle boundaries ... */
  /* Compute over the interior points */

  for (j=yints; j<yinte; j++) {
    for (i=xints; i<xinte; i++) {
      /* convective coefficients for upwinding ... */
      /* U velocity */
      u          = x[j][i].u;
      uxx        = (2.0*u - x[j][i-1].u - x[j][i+1].u)*hydhx;
      uyy        = (2.0*u - x[j-1][i].u - x[j+1][i].u)*hxdhy;
      f[j][i].u  = uxx + uyy - .5*(x[j+1][i].omega-x[j-1][i].omega)*hx;
      /* V velocity, Omega ... */
      /* Temperature */
      u             = x[j][i].temp;
      uxx           = (2.0*u - x[j][i-1].temp - x[j][i+1].temp)*hydhx;
      uyy           = (2.0*u - x[j-1][i].temp - x[j+1][i].temp)*hxdhy;
      f[j][i].temp =  uxx + uyy + prandtl
        * (  (vxp*(u - x[j][i-1].temp) + vxm*(x[j][i+1].temp - u)) * hy
           + (vyp*(u - x[j-1][i].temp) + vym*(x[j+1][i].temp - u)) * hx);
}}}
```

$PETSC_DIR/src/snes/tutorials/ex19.c

Hands-on: Running the driven cavity

Run SNES ex19 with a single MPI rank (see full instructions for all hands-on exercises here):

```
./ex19 -snes_monitor -snes_converged_reason -da_grid_x 16 -da_grid_y 16 -da_refine 2 -lidvelocity 100
     -grashof 1e2
```

Hands-on: Running the driven cavity

Run SNES ex19 with a single MPI rank (see full instructions for all hands-on exercises here):

```
./ex19 -snes_monitor -snes_converged_reason -da_grid_x 16 -da_grid_y 16 -da_refine 2 -lidvelocity 100
    -grashof 1e2
```

Explanation of the above command-line options:

- `-snes_monitor`: Show progress of the SNES solver
- `-snes_converged_reason`: Print reason for SNES convergence or divergence
- `-da_grid_x` 16: Set initial grid points in x direction to 16
- `-da_grid_y` 16: Set initial grid poings in y direction to 16
- `-da_refine` 2: Refine the initial grid 2 times before creation
- `-lidvelocity` 100: Set dimensionless velocity of lid to 100
- `-grashof` 1e2: Set Grashof number to 1e2

An element of the PETSc design philosophy is extensive runtime customizability;
Use `-help` to enumerate and explain the various command-line options available.

Hands-on: Running the driven cavity

Run SNES ex19 with a single MPI rank (see full instructions for all hands-on exercises here):

```
./ex19 -snes_monitor -snes_converged_reason -da_grid_x 16 -da_grid_y 16 -da_refine 2 -lidvelocity 100
     -grashof 1e2
```

```
lid velocity = 100., prandtl # = 1., grashof # = 100.
  0 SNES Function norm 7.681163231938e+02
  1 SNES Function norm 6.582880149343e+02
  2 SNES Function norm 5.294044874550e+02
  3 SNES Function norm 3.775102116141e+02
  4 SNES Function norm 3.047226778615e+02
  5 SNES Function norm 2.599983722908e+00
  6 SNES Function norm 9.427314747057e-03
  7 SNES Function norm 5.212213461756e-08
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 7
Number of SNES iterations = 7
```

What is the SNES solver actually doing? Add `-snes_view` to see

```
SNES Object: 1 MPI processes
  type: newtonls
  maximum iterations=50, maximum function evaluations=10000
  tolerances: relative=1e-08, absolute=1e-50, solution=1e-08
  total number of linear solver iterations=835
  total number of function evaluations=11
  norm schedule ALWAYS
  Jacobian is built using colored finite differences on a DMDA
  SNESLineSearch Object: 1 MPI processes
    type: bt
      interpolation: cubic
      alpha=1.000000e-04
    maxstep=1.000000e+08, minlambda=1.000000e-12
    tolerances: relative=1.000000e-08, absolute=1.000000e-15, lambda=1.000000e-08
    maximum iterations=40
  KSP Object: 1 MPI processes
    type: gmres
      restart=30, using Classical (unmodified) Gram-Schmidt Orthogonalization with no iterative refinement
      happy breakdown tolerance 1e-30
    maximum iterations=10000, initial guess is zero
    tolerances:  relative=1e-05, absolute=1e-50, divergence=10000.
    left preconditioning
    using PRECONDITIONED norm type for convergence test
  PC Object: 1 MPI processes
    type: ilu
      out-of-place factorization
      0 levels of fill
      tolerance for zero pivot 2.22045e-14
      matrix ordering: natural
      factor fill ratio given 1., needed 1.
        Factored matrix follows:
          Mat Object: 1 MPI processes
            type: seqaij
            rows=14884, cols=14884, bs=4
            package used to perform factorization: petsc
            total: nonzeros=293776, allocated nonzeros=293776
                                [continued...]
```

Managing PETSc options

PETSc offers a very large number of runtime options.
All can be set via command line, but can also be set from input files and shell environment variables.

To facilitate readability, we'll put the command-line arguments common to the remaining hands-on exercises in `PETSC_OPTIONS`.

```
export PETSC_OPTIONS="-snes_monitor -snes_converged_reason -lidvelocity 100 -da_grid_x
    16 -da_grid_y 16 -ksp_converged_reason -log_view :log.txt"
```

We've added `-ksp_converged_reason` to see how and when linear solver halts.

We've also added `-log_view` to write the PETSc performance logging info to a file.
We don't have time to explain the performance logs; find the overall wall-clock time via

```
grep Time\ \(sec\): log.txt
```

Hands-on: Exact vs. Inexact Newton

PETSc defaults to inexact Newton. To run exact (and check the execution time), do

```
./ex19 -da_refine 2 -grashof 1e2 -pc_type lu
grep Time\ \(sec\): log.txt
```

Now run inexact Newton and vary the liner solve tolerance (`-ksp_rtol`).

```
./ex19 -da_refine 2 -grashof 1e2 -ksp_rtol 1e-8
./ex19 -da_refine 2 -grashof 1e2 -ksp_rtol 1e-5
./ex19 -da_refine 2 -grashof 1e2 -ksp_rtol 1e-3
./ex19 -da_refine 2 -grashof 1e2 -ksp_rtol 1e-2
./ex19 -da_refine 2 -grashof 1e2 -ksp_rtol 1e-1
```

What happens to the SNES iteration count? When does it diverge?
What yields the shortest execution time?

Hands-on: Scaling up grid size and running in parallel

What happens to iteration counts (and execution time) as we scale up the grid size?
For this exercise, run in parallel because experiments may take too long otherwise.
We also use BiCGStab (`-ksp_type bcgs`) because the default GMRES(30) fails for some cases.

Hands-on: Scaling up grid size and running in parallel

What happens to iteration counts (and execution time) as we scale up the grid size?
For this exercise, run in parallel because experiments may take too long otherwise.
We also use BiCGStab (`-ksp_type bcgs`) because the default GMRES(30) fails for some cases.

Run with default preconditioner. What happens to iteration counts and execution time?

```
mpiexec -n 12 ./ex19 -ksp_type bcgs -grashof 1e2 -da_refine 2
mpiexec -n 12 ./ex19 -ksp_type bcgs -grashof 1e2 -da_refine 3
mpiexec -n 12 ./ex19 -ksp_type bcgs -grashof 1e2 -da_refine 4
```

Hands-on: Scaling up grid size and running in parallel

What happens to iteration counts (and execution time) as we scale up the grid size?
For this exercise, run in parallel because experiments may take too long otherwise.
We also use BiCGStab (`-ksp_type bcgs`) because the default GMRES(30) fails for some cases.

Run with default preconditioner. What happens to iteration counts and execution time?

```
mpiexec -n 12 ./ex19 -ksp_type bcgs -grashof 1e2 -da_refine 2
mpiexec -n 12 ./ex19 -ksp_type bcgs -grashof 1e2 -da_refine 3
mpiexec -n 12 ./ex19 -ksp_type bcgs -grashof 1e2 -da_refine 4
```

```
$ mpiexec -n 12 ./ex19 -ksp_type bcgs -grashof 1e2 -da_refine 4
lid velocity = 100., prandtl # = 1., grashof # = 100.
  0 SNES Function norm 1.545962539057e+03
  Linear solve converged due to CONVERGED_RTOL iterations 172
  1 SNES Function norm 9.780980584978e+02
  Linear solve converged due to CONVERGED_RTOL iterations 128
  2 SNES Function norm 6.620854219003e+02
  Linear solve converged due to CONVERGED_RTOL iterations 600
  3 SNES Function norm 3.219025282761e+00
  Linear solve converged due to CONVERGED_RTOL iterations 470
  4 SNES Function norm 9.280944447516e-03
  Linear solve converged due to CONVERGED_RTOL iterations 467
  5 SNES Function norm 1.354460792476e-07
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 5
```

Hands-on: Scaling up grid size and running in parallel

What happens to iteration counts (and execution time) as we scale up the grid size?
For this exercise, run in parallel because experiments may take too long otherwise.
We also use BiCGStab (`-ksp_type bcgs`) because the default GMRES(30) fails for some cases.

Let's try geometric multigrid (defaults to V-cycle) by adding `-pc_type mg`

```
mpiexec -n 12 ./ex19 -ksp_type bcgs -grashof 1e2 -pc_type mg -da_refine 2
mpiexec -n 12 ./ex19 -ksp_type bcgs -grashof 1e2 -pc_type mg -da_refine 3
mpiexec -n 12 ./ex19 -ksp_type bcgs -grashof 1e2 -pc_type mg -da_refine 4
```

Hands-on: Scaling up grid size and running in parallel

What happens to iteration counts (and execution time) as we scale up the grid size?
For this exercise, run in parallel because experiments may take too long otherwise.
We also use BiCGStab (`-ksp_type bcgs`) because the default GMRES(30) fails for some cases.

Let's try geometric multigrid (defaults to V-cycle) by adding `-pc_type mg`

```
mpiexec -n 12 ./ex19 -ksp_type bcgs -grashof 1e2 -pc_type mg -da_refine 2
mpiexec -n 12 ./ex19 -ksp_type bcgs -grashof 1e2 -pc_type mg -da_refine 3
mpiexec -n 12 ./ex19 -ksp_type bcgs -grashof 1e2 -pc_type mg -da_refine 4
```

```
mpiexec -n 12 ./ex19 -ksp_type bcgs -grashof 1e2 -pc_type mg -da_refine 4
lid velocity = 100., prandtl # = 1., grashof # = 100.
  0 SNES Function norm 1.545962539057e+03
  Linear solve converged due to CONVERGED_RTOL iterations 6
  1 SNES Function norm 9.778196290981e+02
  Linear solve converged due to CONVERGED_RTOL iterations 6
  2 SNES Function norm 6.609659458090e+02
  Linear solve converged due to CONVERGED_RTOL iterations 7
  3 SNES Function norm 2.791922927549e+00
  Linear solve converged due to CONVERGED_RTOL iterations 6
  4 SNES Function norm 4.973591997243e-03
  Linear solve converged due to CONVERGED_RTOL iterations 6
  5 SNES Function norm 3.241555827567e-05
  Linear solve converged due to CONVERGED_RTOL iterations 9
  6 SNES Function norm 9.883136583477e-10
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 6
```

Hands-on: Increasing strength of the nonlinearity

What happens as we increase the nonlinearity by raising the Grashof number?

```
./ex19 -da_refine 2 -grashof 1e2
./ex19 -da_refine 2 -grashof 1e3
./ex19 -da_refine 2 -grashof 1e4
./ex19 -da_refine 2 -grashof 1.3e4
```

Hands-on: Increasing strength of the nonlinearity

What happens as we increase the nonlinearity by raising the Grashof number?

```
./ex19 -da_refine 2 -grashof 1e2
./ex19 -da_refine 2 -grashof 1e3
./ex19 -da_refine 2 -grashof 1e4
./ex19 -da_refine 2 -grashof 1.3e4
```

```
./ex19 -da_refine 2 -grashof 1.3e4
lid velocity = 100., prandtl # = 1., grashof # = 13000.
  0 SNES Function norm 7.971152173639e+02
  Linear solve did not converge due to DIVERGED_ITS iterations 10000
Nonlinear solve did not converge due to DIVERGED_LINEAR_SOLVE iterations 0
```

Oops! Failure in the linear solver? What if we use a stronger preconditioner?

Hands-on: Increasing strength of the nonlinearity

What happens as we increase the nonlinearity by raising the Grashof number?

```
./ex19 -da_refine 2 -grashof 1e2
./ex19 -da_refine 2 -grashof 1e3
./ex19 -da_refine 2 -grashof 1e4
./ex19 -da_refine 2 -grashof 1.3e4
./ex19 -da_refine 2 -grashof 1.3e4 -pc_type mg
```

Hands-on: Increasing strength of the nonlinearity

What happens as we increase the nonlinearity by raising the Grashof number?

```
./ex19 -da_refine 2 -grashof 1e2
./ex19 -da_refine 2 -grashof 1e3
./ex19 -da_refine 2 -grashof 1e4
./ex19 -da_refine 2 -grashof 1.3e4
./ex19 -da_refine 2 -grashof 1.3e4 -pc_type mg
```

```
./ex19 -da_refine 2 -grashof 1.3e4 -pc_type mg
lid velocity = 100., prandtl # = 1., grashof # = 13000.
  ...
  4 SNES Function norm 3.209967262833e+02
  Linear solve converged due to CONVERGED_RTOL iterations 9
  5 SNES Function norm 2.121900163587e+02
  Linear solve converged due to CONVERGED_RTOL iterations 9
  6 SNES Function norm 1.139162432910e+01
  Linear solve converged due to CONVERGED_RTOL iterations 8
  7 SNES Function norm 4.048269317796e-01
  Linear solve converged due to CONVERGED_RTOL iterations 8
  8 SNES Function norm 3.264993685206e-04
  Linear solve converged due to CONVERGED_RTOL iterations 8
  9 SNES Function norm 1.154893029612e-08
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 9
```

Hands-on: Increasing strength of the nonlinearity

What happens as we increase the nonlinearity by raising the Grashof number?

```
./ex19 -da_refine 2 -grashof 1e2
./ex19 -da_refine 2 -grashof 1e3
./ex19 -da_refine 2 -grashof 1e4
./ex19 -da_refine 2 -grashof 1.3e4
./ex19 -da_refine 2 -grashof 1.3e4 -pc_type mg
./ex19 -da_refine 2 -grashof 1.3373e4 -pc_type mg
```

Hands-on: Increasing strength of the nonlinearity

What happens as we increase the nonlinearity by raising the Grashof number?

```
./ex19 -da_refine 2 -grashof 1e2
./ex19 -da_refine 2 -grashof 1e3
./ex19 -da_refine 2 -grashof 1e4
./ex19 -da_refine 2 -grashof 1.3e4
./ex19 -da_refine 2 -grashof 1.3e4 -pc_type mg
./ex19 -da_refine 2 -grashof 1.3373e4 -pc_type mg
```

```
lid velocity = 100., prandtl # = 1., grashof # = 13373.
...
 48 SNES Function norm 3.124919801005e+02
  Linear solve converged due to CONVERGED_RTOL iterations 17
 49 SNES Function norm 3.124919800338e+02
  Linear solve converged due to CONVERGED_RTOL iterations 17
 50 SNES Function norm 3.124919799645e+02
Nonlinear solve did not converge due to DIVERGED_MAX_IT iterations 50
```

Hands-on: Increasing strength of the nonlinearity

What happens as we increase the nonlinearity by raising the Grashof number?

```
./ex19 -da_refine 2 -grashof 1e2
./ex19 -da_refine 2 -grashof 1e3
./ex19 -da_refine 2 -grashof 1e4
./ex19 -da_refine 2 -grashof 1.3e4
./ex19 -da_refine 2 -grashof 1.3e4 -pc_type mg
./ex19 -da_refine 2 -grashof 1.3373e4 -pc_type mg
./ex19 -da_refine 2 -grashof 1.3373e4 -pc_type lu
```

Hands-on: Increasing strength of the nonlinearity

What happens as we increase the nonlinearity by raising the Grashof number?

```
./ex19 -da_refine 2 -grashof 1e2
./ex19 -da_refine 2 -grashof 1e3
./ex19 -da_refine 2 -grashof 1e4
./ex19 -da_refine 2 -grashof 1.3e4
./ex19 -da_refine 2 -grashof 1.3e4 -pc_type mg
./ex19 -da_refine 2 -grashof 1.3373e4 -pc_type mg
./ex19 -da_refine 2 -grashof 1.3373e4 -pc_type lu
```

```
./ex19 -da_refine 2 -grashof 1.3373e4 -pc_type lu
...
 48 SNES Function norm 3.193724239842e+02
  Linear solve converged due to CONVERGED_RTOL iterations 1
 49 SNES Function norm 3.193724232621e+02
  Linear solve converged due to CONVERGED_RTOL iterations 1
 50 SNES Function norm 3.193724181714e+02
Nonlinear solve did not converge due to DIVERGED_MAX_IT iterations 50
```

A strong linear solver can't help us here. What now?

Let's try combining Newton's method with one of the other nonlinear solvers we mentioned in the introduction, using PETSc's support for nonlinear composition and preconditioning.

Abstract Nonlinear System and Solver

To discuss nonlinear composition and preconditioning, we introduce some definitions and notation.

Our prototypical nonlinear equation is of the form

$$F(x) = b \tag{5}$$

and we define the residual as

$$r(x) = F(x) - b \tag{6}$$

We use the notation $x_{k+1} = \mathcal{M}(F, x_k, b)$ for the action of a nonlinear solver.

Nonlinear Composition: Additive

Nonlinear composition consists of a sequence of two (or more) methods $\mathcal{M}$ and $\mathcal{N}$, which both provide an approximation solution to $F(x) = b$.

In the linear case, application of a stationary solver by defect correct can be written as

$$x_{k+1} = x_k - P^{-1}(Ax_k - b) \tag{7}$$

where P^{-1} is a linear preconditioner. (Richardson iteration applied to a preconditioned system.)

An additive composition of preconditioners P^{-1} and Q^{-1} with weights α and β may be written as

$$x_{k+1} = x_k - (\alpha P^{-1} + \beta Q^{-1})(Ax_k - b) \tag{8}$$

Analogously, for the nonlinear case, additive composition is

$$x_{k+1} = x_k + \alpha \cdot (\mathcal{M}(F, x_k, b) - x_k) + \beta \cdot (\mathcal{N}(F, x_k, b) - x_k) \tag{9}$$

Nonlinear Composition: Multiplicative

A multiplicative combination of linear preconditioners may be written as

$$\begin{aligned} x_{k+1/2} &= x_k - P^{-1}(Ax_k - b), \\ x_{k+1} &= x_{k+1/2} - Q^{-1}(Ax_{k+1/2} - b), \end{aligned} \tag{10}$$

Analogously, for the nonlinear case

$$x_{k+1} = \mathcal{M}(F, \mathcal{N}(F, x_k, b), b) \tag{11}$$

which simply indicates to update the solution using the current solution and residual with the first solver and then update the solution again using the resulting new solution and new residual with the second solver.

Nonlinear Left Preconditioning

Recall that the stationary iteration for our left-preconditioned linear system is

$$x_{k+1} = x_k - P^{-1}(Ax_k - b) \tag{12}$$

And since $Ax_k - b = r$, for the linear case we can write the action of our solver $\mathcal{N}$ as

$$\mathcal{N}(F, x, b) = x_k - P^{-1}r \tag{13}$$

With slight rearranging, we can express the left-preconditioned residual

$$P^{-1}r = x_k - \mathcal{N}(F, x, b) \tag{14}$$

And generalizing to the nonlinear case, the left preconditioning operation provides a modified residual

$$r_L = x_k - \mathcal{N}(F, x, b) \tag{15}$$

Nonlinear Right Preconditioning

For a right preconditioned linear system $AP^{-1}Px = b$, we solve the systems

$$\begin{aligned} AP^{-1}y &= b \\ x &= P^{-1}y \end{aligned} \tag{16}$$

Analogously, we define the right preconditioning operation in the nonlinear case as

$$\begin{aligned} y &= \mathcal{M}(F(\mathcal{N}(F, \cdot, b)), x_k, b) \\ x &= \mathcal{N}(F, y, b) \end{aligned} \tag{17}$$

(Note: In the linear case the above actually reduces to $A(I - P^{-1}A)y = (I - AP^{-1})b$, but the inner solver is applied before the function evaluation (matrix-vector product in the linear case), so we retain the "right preconditioning" name.)

Nonlinear Composition and Preconditioning

Type	Sym	Statement	Abbreviation
Additive	$+$	$x + \alpha(\mathcal{M}(\mathcal{F}, x, b) - x)$ $+ \beta(\mathcal{N}(\mathcal{F}, x, b) - x)$	$\mathcal{M} + \mathcal{N}$
Multiplicative	$*$	$\mathcal{M}(\mathcal{F}, \mathcal{N}(\mathcal{F}, x, b), b)$	$\mathcal{M} * \mathcal{N}$
Left Prec.	$-_L$	$\mathcal{M}(x - \mathcal{N}(\mathcal{F}, x, b), x, b)$	$\mathcal{M} -_L \mathcal{N}$
Right Prec.	$-_R$	$\mathcal{M}(\mathcal{F}(\mathcal{N}(\mathcal{F}, x, b)), x, b)$	$\mathcal{M} -_R \mathcal{N}$
Inner Lin. Inv.	$\backslash$	$y = J(x)^{-1} r(x) = \mathsf{K}(J(x), y_0, b)$	$\mathcal{N} \backslash \mathsf{K}$

Composing Scalable Nonlinear Algebraic Solvers,
Brune, Knepley, Smith, and Tu, SIAM Review, 2015.

For details on using nonlinear composition and preconditioning, see manual pages for `SNESCOMPOSITE` and `SNESGetNPC()`.

Hands-on: Nonlinear Richardson Preconditioned with Newton

```
./ex19 -da_refine 2 -grashof 1.3373e4 -snes_type nrichardson -npc_snes_type newtonls
    -npc_snes_max_it 4 -npc_pc_type mg
```

Hands-on: Nonlinear Richardson Preconditioned with Newton

```
./ex19 -da_refine 2 -grashof 1.3373e4 -snes_type nrichardson -npc_snes_type newtonls
    -npc_snes_max_it 4 -npc_pc_type mg

lid velocity = 100., prandtl # = 1., grashof # = 13373.
Nonlinear solve did not converge due to DIVERGED_INNER iterations 0
```

Hands-on: Nonlinear Richardson Preconditioned with Newton

```
./ex19 -da_refine 2 -grashof 1.3373e4 -snes_type nrichardson -npc_snes_type newtonls
    -npc_snes_max_it 4 -npc_pc_type mg
./ex19 -da_refine 2 -grashof 1.3373e4 -snes_type nrichardson -npc_snes_type newtonls
    -npc_snes_max_it 4 -npc_pc_type lu
```

Hands-on: Nonlinear Richardson Preconditioned with Newton

```
./ex19 -da_refine 2 -grashof 1.3373e4 -snes_type nrichardson -npc_snes_type newtonls
    -npc_snes_max_it 4 -npc_pc_type mg
./ex19 -da_refine 2 -grashof 1.3373e4 -snes_type nrichardson -npc_snes_type newtonls
    -npc_snes_max_it 4 -npc_pc_type lu

lid velocity = 100., prandtl # = 1., grashof # = 13373.
  0 SNES Function norm 7.987708558131e+02
  1 SNES Function norm 8.467169687854e+02
  2 SNES Function norm 7.300096001529e+02
  3 SNES Function norm 5.587232361127e+02
  4 SNES Function norm 3.071143076019e+03
  5 SNES Function norm 3.347748537471e+02
  6 SNES Function norm 1.383297972324e+01
  7 SNES Function norm 1.209841384629e-02
  8 SNES Function norm 8.660606193428e-09
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 8
```

So nonlinear Richardson preconditioned with Newton has let us go further than Newton alone.

Hands-on: Nonlinear Richardson Preconditioned with Newton

```
./ex19 -da_refine 2 -grashof 1.3373e4 -snes_type nrichardson -npc_snes_type newtonls
    -npc_snes_max_it 4 -npc_pc_type mg
./ex19 -da_refine 2 -grashof 1.3373e4 -snes_type nrichardson -npc_snes_type newtonls
    -npc_snes_max_it 4 -npc_pc_type lu
./ex19 -da_refine 2 -grashof 1.4e4 -snes_type nrichardson -npc_snes_type newtonls
    -npc_snes_max_it 4 -npc_pc_type lu
```

Hands-on: Nonlinear Richardson Preconditioned with Newton

```
./ex19 -da_refine 2 -grashof 1.3373e4 -snes_type nrichardson -npc_snes_type newtonls
    -npc_snes_max_it 4 -npc_pc_type mg
./ex19 -da_refine 2 -grashof 1.3373e4 -snes_type nrichardson -npc_snes_type newtonls
    -npc_snes_max_it 4 -npc_pc_type lu
./ex19 -da_refine 2 -grashof 1.4e4 -snes_type nrichardson -npc_snes_type newtonls
    -npc_snes_max_it 4 -npc_pc_type lu


lid velocity = 100., prandtl # = 1., grashof # = 14000.
...
 37 SNES Function norm 5.992348444448e+02
 38 SNES Function norm 5.992348444290e+02
Nonlinear solve did not converge due to DIVERGED_INNER iterations 38
```

We've hit another barrier. What about switching things up?
Let's try preconditioning Newton with nonlinear Richardson.

Hands-on: Newton Preconditioned with Nonlinear Richardson

```
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 1
     -snes_max_it 1000
```

Hands-on: Newton Preconditioned with Nonlinear Richardson

```
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 1
     -snes_max_it 1000


...
352 SNES Function norm 2.145588832260e-02
  Linear solve converged due to CONVERGED_RTOL iterations 7
353 SNES Function norm 1.288292314235e-05
  Linear solve converged due to CONVERGED_RTOL iterations 8
354 SNES Function norm 3.219155715396e-10
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 354
```

Hands-on: Newton Preconditioned with Nonlinear Richardson

```
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 1
     -snes_max_it 1000
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 3
```

Hands-on: Newton Preconditioned with Nonlinear Richardson

```
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 1
     -snes_max_it 1000
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 3


...
 23 SNES Function norm 4.796734188970e+00
  Linear solve converged due to CONVERGED_RTOL iterations 7
 24 SNES Function norm 2.083806106198e-01
  Linear solve converged due to CONVERGED_RTOL iterations 8
 25 SNES Function norm 1.368771861149e-04
  Linear solve converged due to CONVERGED_RTOL iterations 8
 26 SNES Function norm 1.065794992653e-08
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 26
```

Hands-on: Newton Preconditioned with Nonlinear Richardson

```
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 1
    -snes_max_it 1000
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 3
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 4
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 5
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 6
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 7
```

Hands-on: Newton Preconditioned with Nonlinear Richardson

```
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 1
    -snes_max_it 1000
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 3
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 4
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 5
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 6
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 7
```

```
lid velocity = 100., prandtl # = 1., grashof # = 14000.
  0 SNES Function norm 8.016512665033e+02
  Linear solve converged due to CONVERGED_RTOL iterations 11
  1 SNES Function norm 7.961475922316e+03
  Linear solve converged due to CONVERGED_RTOL iterations 10
  2 SNES Function norm 3.238304139699e+03
  Linear solve converged due to CONVERGED_RTOL iterations 10
  3 SNES Function norm 4.425107973263e+02
  Linear solve converged due to CONVERGED_RTOL iterations 9
  4 SNES Function norm 2.010474128858e+02
  Linear solve converged due to CONVERGED_RTOL iterations 8
  5 SNES Function norm 2.936958163548e+01
  Linear solve converged due to CONVERGED_RTOL iterations 8
  6 SNES Function norm 1.183847022611e+00
  Linear solve converged due to CONVERGED_RTOL iterations 8
  7 SNES Function norm 6.662829301594e-03
  Linear solve converged due to CONVERGED_RTOL iterations 7
  8 SNES Function norm 6.170083332176e-07
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 8
```

Hands-on: Newton Preconditioned with Nonlinear Richardson

```
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 1
    -snes_max_it 1000
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 3
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 4
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 5
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 6
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 7
```

Newton preconditioned with nonlinear Richardson can be pushed quite far! Try

```
./ex19 -da_refine 2 -grashof 1e6 -pc_type lu -npc_snes_type nrichardson -npc_snes_max_it 7 -snes_max_it
    1000
```

Hands-on: Newton Preconditioned with Nonlinear Richardson

```
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 1
    -snes_max_it 1000
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 3
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 4
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 5
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 6
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 7
```

Newton preconditioned with nonlinear Richardson can be pushed quite far! Try

```
./ex19 -da_refine 2 -grashof 1e6 -pc_type lu -npc_snes_type nrichardson -npc_snes_max_it 7 -snes_max_it
    1000
```

```
lid velocity = 100., prandtl # = 1., grashof # = 1e+06
...
 69 SNES Function norm 4.241700887134e+00
  Linear solve converged due to CONVERGED_RTOL iterations 1
 70 SNES Function norm 3.238739735055e+00
  Linear solve converged due to CONVERGED_RTOL iterations 1
 71 SNES Function norm 1.781881532852e+00
  Linear solve converged due to CONVERGED_RTOL iterations 1
 72 SNES Function norm 1.677710773493e-05
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 72
```

Hands-on: Newton Preconditioned with Nonlinear Richardson

```
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 1
    -snes_max_it 1000
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 3
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 4
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 5
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 6
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 7
```

Newton preconditioned with nonlinear Richardson can be pushed quite far! Try

```
./ex19 -da_refine 2 -grashof 1e6 -pc_type lu -npc_snes_type nrichardson -npc_snes_max_it 7 -snes_max_it
    1000
```

```
lid velocity = 100., prandtl # = 1., grashof # = 1e+06
...
 69 SNES Function norm 4.241700887134e+00
  Linear solve converged due to CONVERGED_RTOL iterations 1
 70 SNES Function norm 3.238739735055e+00
  Linear solve converged due to CONVERGED_RTOL iterations 1
 71 SNES Function norm 1.781881532852e+00
  Linear solve converged due to CONVERGED_RTOL iterations 1
 72 SNES Function norm 1.677710773493e-05
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 72
```

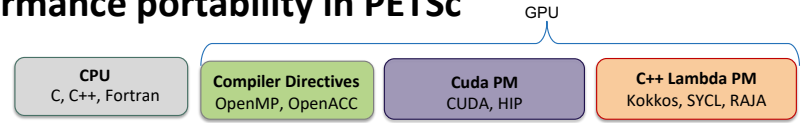
Takeaway: The PETSc philosophy of supporting extensive runtime experimentation and composition enables discovery of effective approaches when the best solver cannot be determined *a priori*.

Bonus / Extra Credit: Running With GPUs

Our discussion of nonlinear solver algorithms and how to use them via PETSc is mostly orthogonal to the topic of how to run PETSc solvers on GPUs.

Since computing on GPUs has become so important, however, in the slides that follow, we take a brief look at GPU support in PETSc and how its nonlinear solvers can be executed on GPUs.

Performance portability in PETSc



Application code

Using PETSc API

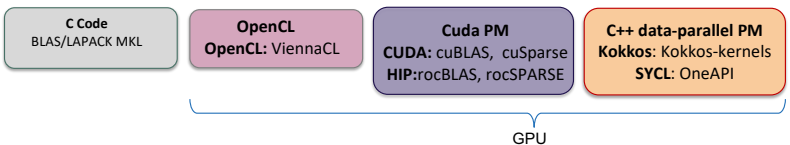
Front-end PETSc vector and matrix arrays are shared with user programming language/model

PETSc computation kernels

CPU: Use compiler options and vendor libraries for performance

GPU: Chosen for either speed of development or highest performance. Use GPU vendor libraries

Back-end PETSc Vector and Matrix implementations



The Common Pattern for PETSc Application Codes

PETSc application codes, regardless of whether they use time integrators, nonlinear solvers, or linear solvers, follow a common pattern:

- Compute application-specific data structures,
- Provide a `Function` computation callback,
- Provide a `Jacobian` (or `Hessian`, etc.) computation callback, and
- Call the PETSc solver, possibly in a loop.

This approach does not change with the use of GPUs.
In particular, the creation of solver, matrix, and vector objects and their manipulation do not change.

Points to consider when porting an application to GPUs:

- Some data structures reside in GPU memory, either
  - constructed on the CPU and copied to the GPU or
  - constructed directly on the GPU.
- `Function` will call GPU kernels.
- `Jacobian` will call GPU kernels.

Main Application Code for CPU or GPU

Consider excerpt of a typical PETSc main application program for solving a nonlinear set of equations on a structured grid using Newton's method. It creates a solver object `SNES`, a data management object `DM`, a vector of degrees of freedom `Vec`, and a `Mat` to hold the Jacobian. Then, Function and Jacobian evaluation callbacks are passed to the `SNES` object to solve the nonlinear equations.

```
SNESCreate(PETSC_COMM_WORLD,&snes);
DMDACreate1d(PETSC_COMM_WORLD,...,&ctx.da);
DMCreateGlobalVector(ctx.da,&x);
VecDuplicate(x,&r);
DMCreateMatrix(ctx.da,&J);
if (useKokkos) {
 SNESSetFunction(snes,r,KokkosFunction,&ctx);
 SNESSetJacobian(snes,J,J,KokkosJacobian,&ctx);
} else {
 SNESSetFunction(snes,r,Function,&ctx);
 SNESSetJacobian(snes,J,J,Jacobian,&ctx);
}
SNESSolve(snes,NULL,x);
```

Listing 1: Main application code for CPU or GPU

Traditional PETSc Function and Kokkos version

```
DMGetLocalVector(da,&xl);
DMGlobalToLocal(da,x,INSERT_VALUES,xl);
DMDAVecGetArrayRead(da,xl,&X); // only read X[]
DMDAVecGetArrayWrite(da,r,&R); // only write R[]
DMDAVecGetArrayRead(da,f,&F);  // only read F[]
DMDAGetCorners(da,&xs,NULL,NULL,&xm,...);
for (i=xs; i<xs+xm; ++i)
  R[i] = d*(X[i-1]-2*X[i]+X[i+1])+X[i]*X[i]-F[i];
-----------------------------------------------------
DMGetLocalVector(da,&xl);
DMGlobalToLocal(da,x,INSERT_VALUES,xl);
DMDAVecGetKokkosOffsetView(da,xl,&X); // no copy
DMDAVecGetKokkosOffsetView(da,r,&R,overwrite);
DMDAVecGetKokkosOffsetView(da,f,&F);
xs = R.begin(0); xm = R.end(0);
Kokkos::parallel_for(
  Kokkos::RangePolicy<>(xs,xm),KOKKOS_LAMBDA
  (int i) {
    R(i) = d*(X(i-1)-2*X(i)+X(i+1))+X(i)*X(i)-F(i);});
```

Listing 2: Traditional PETSc Function (top) and Kokkos version (bottom). `xl`, `x`, `r`, `f` are PETSc vectors. `X`, `R`, `F` at the top are `double*` or `const double*` like pointers but at the bottom are Kokkos unmanaged `OffsetViews`.

On-Device Matrix Assembly: MatSetValuesCOO() API

```
/* simulation of CPU based finite assembly process with COO */
PetscScalar    *v,*s;
PetscCall(PetscMalloc1(3 * 3 * fe->Ne, &v));
for (PetscInt e=0; e<fe->Ne; e++) {
  s = v + fe->coo[e]; /* point to location in COO of current element stiffness */
  for (PetscInt vi=0; vi<3; vi++) {
    for (PetscInt vj=0; vj<3; vj++) {
      s[3*vi+vj] = vi+2*vj;
    }
  }
}
PetscCall(MatSetValuesCOO(A, v, ADD_VALUES));
```

Listing 3: Matrix assembly on CPU

```
// Simulation of GPU based finite assembly process with COO
Kokkos::View<PetscScalar*,DefaultMemorySpace> v("v",3*3*fe->Ne);
Kokkos::parallel_for(
  "AssembleElementMatrices", fe->Ne, KOKKOS_LAMBDA(PetscInt i) {
    PetscScalar *s = &v(3 * 3 * i);
    for (PetscInt vi = 0; vi < 3; vi++) {
      for (PetscInt vj = 0; vj < 3; vj++) s[vi * 3 + vj] = vi + 2 * vj;
    }
  });

PetscCall(MatSetValuesCOO(A, v.data(), ADD_VALUES));
```

Listing 4: Matrix assembly on GPU using Kokkos

How PETSc Uses GPUs: Back-End

- Provides several new implementations of PETSc's `Vec` (distributed vector) and `Mat` (distributed matrix) classes which allow data storage and manipulation in device (GPU) memory
- Embue all `Vec` (and `Mat`) objects with the ability to track the state of a second "offloaded" copy of the data, and synchronize these two copies of the data (only) when required ("lazy-mirror" model).
- Because higher-level PETSc objects rely on `Vec` and `Mat` operations, execution occurs on GPU when appropriate delegated types for `Vec` and `Mat` are chosen.

Host and Device Data

```
struct _p_Vec {
  ...
  void              *data;        // host buffer
  void              *spptr;       // device buffer
  PetscOffloadMask offloadmask;  // which copies are valid
};
```

Possible Flag States

```
  typedef enum {PETSC_OFFLOAD_UNALLOCATED,
                PETSC_OFFLOAD_GPU,
                PETSC_OFFLOAD_CPU,
                PETSC_OFFLOAD_BOTH} PetscOffloadMask;
```

Using GPU Back-Ends in PETSc

Transparently use GPUs for common matrix and vector operations, via runtime options.
Currently CUDA/cuSPARSE, HIP/hipSPARSE, Kokkos, and ViennaCL are supported.

CUDA/cuSPARSE usage:

- CUDA matrix and vector types:
  `-mat_type aijcusparse -vec_type cuda`
- GPU-enabled preconditioners:
  - GPU-based ILU: `-pc_type ilu -pc_factor_mat_solver_type cusparse`
  - Jacobi: `-pc_type jacobi`

Because PETSc separates high-level control logic from optimized computational kernels, even very complicated hierarchical/multi-level/domain-decomposed/physics-based solvers can run on different architectures by simply choosing the appropriate back-end at runtime; **re-coding is not needed**.

Hands-on: CPU and GPU, detailed performance logging

Run a large version of the driven cavity problem, using multigrid with GPU and SIMD-friendly Chebyshev-Jacobi smoothing (`-mg_levels_pc_type jacobi`), and collect a breakdown by multigrid level (`-pc_mg_log`), both in plain text and flamegraph stack formats.

We get the best CPU-only performance on ThetaGPU using 8 MPI ranks:

```
mpiexec -n 8 ./ex19 -da_refine 9 -pc_type mg -mg_levels_pc_type jacobi -pc_mg_log -log_view
    :log_mg_cpu_n8.txt

mpiexec -n 8 ./ex19 -da_refine 9 -pc_type mg -mg_levels_pc_type jacobi -pc_mg_log -log_view
    :log_mg_cpu_n8.stack:ascii_flamegraph
```

Running on GPU, we get best performance using only one rank
(could probably use more if running NVIDIA MPS, but this is not enabled on ThetaGPU):

```
mpiexec -n 1 ./ex19 -da_refine 9 -pc_type mg -mg_levels_pc_type jacobi -pc_mg_log -dm_vec_type cuda
    -dm_mat_type aijcusparse -log_view_gpu_time -log_view :log_mg_gpu_n1.txt

mpiexec -n 1 ./ex19 -da_refine 9 -pc_type mg -mg_levels_pc_type jacobi -pc_mg_log -dm_vec_type cuda
    -dm_mat_type aijcusparse -log_view_gpu_time -log_view :log_mg_gpu_n1.stack:ascii_flamegraph
```

Reading -log_view

- Overall summary:

```
                         Max       Max/Min     Avg       Total
Time (sec):           4.106e+01     1.000   4.106e+01
Objects:              1.201e+03     1.000   1.201e+03
Flops:                2.577e+10     1.005   2.569e+10  2.055e+11
Flops/sec:            6.276e+08     1.005   6.255e+08  5.004e+09
MPI Msg Count:        1.102e+04     1.543   9.172e+03  7.338e+04
MPI Msg Len (bytes):  4.471e+07     1.655   3.910e+03  2.869e+08
MPI Reductions:       2.806e+03     1.000
```

- Also a summary per stage
- Memory usage per stage (based on when it was allocated)
- Time, messages, reductions, balance, flops per event per stage
- Always send `-log_view` when asking performance questions on mailing list!

PETSc Profiling

```
Event                Count      Time (sec)     Flop                             --- Global ---  --- Stage ----  Total
                   Max Ratio  Max     Ratio   Max  Ratio  Mess   AvgLen  Reduct  %T %F %M %L %R  %T %F %M %L %R Mflop/
------------------------------------------------------------------------------------------------------------------------

--- Event Stage 0: Main Stage

BuildTwoSided          86 1.0 2.4533e-02 2.4 0.00e+00 0.0 5.0e+02 4.0e+00 8.6e+01  0  0  1  0  3   0  0  2  0  3     0
BuildTwoSidedF         49 1.0 2.0815e-02 2.5 0.00e+00 0.0 0.0e+00 0.0e+00 4.9e+01  0  0  0  0  2   0  0  0  0  2     0
MatMult               349 1.0 6.0068e+00 1.0 4.25e+09 1.0 6.8e+03 6.5e+03 1.8e+01 14 16  9 16  1  35 41 33 43  1  5642
MatMultTranspose        9 1.0 1.0729e-02 1.0 7.11e+06 1.0 3.5e+02 6.4e+02 9.0e+00  0  0  0  0  0   0  0  2  0  0  5287
MatLUFactorSym          1 1.0 4.8302e-05 1.2 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0     0
MatLUFactorNum          3 1.0 7.8808e-05 1.2 2.73e+04 1.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0  2769
MatCopy                 2 1.0 9.6890e-06 1.3 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0     0
MatConvert              1 1.0 3.8523e-05 1.4 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0     0
MatAssemblyBegin       50 1.0 2.1382e-02 2.4 0.00e+00 0.0 0.0e+00 0.0e+00 4.9e+01  0  0  0  0  2   0  0  0  0  2     0
MatAssemblyEnd         50 1.0 1.1048e-01 1.0 0.00e+00 0.0 0.0e+00 0.0e+00 7.6e+01  0  0  0  0  3   1  0  0  0  3     0
MatGetRowIJ             1 1.0 4.6280e-06 1.5 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0     0
MatCreateSubMats        3 1.0 2.6247e-04 1.1 0.00e+00 0.0 0.0e+00 0.0e+00 6.0e+00  0  0  0  0  0   0  0  0  0  0     0
MatGetOrdering          1 1.0 3.4175e-05 1.2 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0     0
MatFDColorCreate       10 1.0 9.8993e-03 1.0 0.00e+00 0.0 0.0e+00 0.0e+00 2.0e+01  0  0  0  0  1   0  0  0  0  1     0
MatFDColorSetUp        10 1.0 1.5871e+00 1.0 0.00e+00 0.0 0.0e+00 0.0e+00 8.0e+02  4  0  0  0 29   9  0  0  0 29     0
MatFDColorApply        30 1.0 2.8531e+00 1.0 2.29e+09 1.0 1.3e+04 3.8e+03 8.7e+01  7  9 18 17  3  17 22 62 49  3  6390
MatFDColorFunc        630 1.0 9.7597e-01 1.1 2.09e+09 1.0 1.3e+04 3.8e+03 9.0e+00  2  8 18 17  0   6 20 62 49  0 17090
MatRedundantMat         3 1.0 4.1426e-04 1.1 0.00e+00 0.0 0.0e+00 0.0e+00 6.0e+00  0  0  0  0  0   0  0  0  0  0     0
MatMPIConcateSeq        3 1.0 1.0587e-04 1.4 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0     0
KSPSetUp               36 1.0 4.7154e+00 1.0 3.21e+09 1.0 5.8e+03 4.2e+03 1.0e+03 11 12  8  8 37  28 31 28 23 37  5420
KSPSolve                3 1.0 3.0258e+01 1.0 2.00e+10 1.0 5.4e+04 3.8e+03 1.4e+02 74 78 73 71  5 177192258199  5  5269
KSPGMRESOrthog        319 1.0 3.9607e+00 1.1 3.10e+09 1.0 0.0e+00 0.0e+00 3.2e+02  9 12  0  0 11  22 30  0  0 11  6243
SNESSolve               1 1.0 4.0081e+01 1.0 2.58e+10 1.0 7.3e+04 3.9e+03 2.8e+03 98100100100 98 235248354280 99  5127
SNESSetUp               1 1.0 7.4249e-01 1.0 0.00e+00 0.0 0.0e+00 0.0e+00 9.0e+00  2  0  0  0  0   4  0  0  0  0     0
SNESFunctionEval      634 1.0 5.7119e-01 1.1 2.19e+09 1.0 1.2e+02 1.4e+04 1.0e+00  1  9  0  1  0   3 21  1  2  0 30591
SNESJacobianEval       30 1.0 4.4082e+00 1.0 2.29e+09 1.0 1.3e+04 3.8e+03 9.2e+02 11  9 18 17 33  26 22 63 49 33  4136
SNESLineSearch          3 1.0 3.1303e-01 1.0 2.52e+08 1.0 1.2e+02 2.0e+04 1.2e+01  1  1  0  1  0   2  2  1  2  0  6428
DMCoarsen               9 1.0 4.5833e-03 1.0 0.00e+00 0.0 0.0e+00 0.0e+00 1.1e+02  0  0  0  0  4   0  0  0  0  4     0
DMCreateInterp          9 1.0 7.5446e-02 1.0 7.11e+06 1.0 3.5e+02 6.4e+02 2.0e+02  0  0  0  0  7   0  0  2  0  7   752
DMCreateInject          9 1.0 1.0279e-02 1.0 0.00e+00 0.0 2.2e+02 2.9e+04 1.4e+02  0  0  0  2  5   0  0  1  6  5     0
DMCreateMat            10 1.0 9.7872e-01 1.0 0.00e+00 0.0 0.0e+00 0.0e+00 7.0e+01  2  0  0  0  2   6  0  0  0  3     0
[...]
```

PETSc Profiling: Multigrid timings breakdown

```
--- Event Stage 1: MG Apply

BuildTwoSided          2 1.0 8.1144e-05 1.7 0.00e+00 0.0 5.6e+01 4.0e+00 2.0e+00  0  0  0  0  0   0  0  0  0100     0
MatMult             1872 1.0 1.7854e+01 1.0 1.28e+10 1.0 3.7e+04 4.4e+03 0.0e+00 43 50 51 57  0  74 83 71 89  0  5724
MatMultAdd           468 1.0 5.8300e-01 1.1 3.69e+08 1.0 6.1e+03 1.7e+03 0.0e+00  1  1  8  4  0   2  2 12  6  0  5057
MatMultTranspose     468 1.0 7.5539e-01 1.4 3.70e+08 1.0 6.1e+03 1.7e+03 0.0e+00  1  1  8  4  0   3  2 12  6  0  3905
MatSolve              52 1.0 4.5501e-04 1.2 1.96e+05 1.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0  3452
MatResidual          468 1.0 4.7088e+00 1.1 3.29e+09 1.0 9.4e+03 4.4e+03 0.0e+00 11 13 13 14  0  19 21 18 22  0  5565
KSPSolve             988 1.0 1.8321e+01 1.0 1.13e+10 1.0 3.1e+04 4.0e+03 2.0e+00 44 44 42 43  0  75 74 59 67100  4935
SFSetUp                2 1.0 1.0429e-04 1.4 0.00e+00 0.0 1.1e+02 1.8e+01 2.0e+00  0  0  0  0  0   0  0  0  0100     0
SFPack              2912 1.0 1.1586e-02 1.4 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0     0
SFUnpack            2912 1.0 1.4369e-03 3.1 2.44e+05292.8 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0   897
VecCopy             1404 1.0 6.1932e-01 1.2 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  1  0  0  0  0   2  0  0  0  0     0
VecSet              1560 1.0 2.3179e-01 1.4 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   1  0  0  0  0     0
VecAYPX             2808 1.0 1.6977e+00 1.3 6.58e+08 1.0 0.0e+00 0.0e+00 0.0e+00  4  3  0  0  0   7  4  0  0  0  3089
VecAXPBYCZ           936 1.0 9.1505e-01 1.1 8.23e+08 1.0 0.0e+00 0.0e+00 0.0e+00  2  3  0  0  0   4  5  0  0  0  7164
VecPointwiseMult    1872 1.0 1.8484e+00 1.1 3.29e+08 1.0 0.0e+00 0.0e+00 0.0e+00  4  1  0  0  0   7  2  0  0  0  1419
VecScatterBegin     2912 1.0 4.0778e-02 1.4 0.00e+00 0.0 5.3e+04 3.5e+03 2.0e+00  0  0 72 64  0   0  0100100100     0
VecScatterEnd       2912 1.0 1.4528e+00 1.3 2.44e+05292.8 0.0e+00 0.0e+00 0.0e+00  3  0  0  0  0   5  0  0  0  0     1
PCApply             1924 1.0 1.8517e+00 1.1 3.29e+08 1.0 3.0e+03 6.2e+01 2.0e+00  4  1  4  0  0   8  2  6  0100  1417
MGSmooth Level 0      52 1.0 1.8742e-03 1.1 1.96e+05 1.0 3.0e+03 6.2e+01 2.0e+00  0  0  4  0  0   0  0  6  0100   838
MGSmooth Level 1     104 1.0 1.6707e-03 1.2 2.20e+05 3.3 3.1e+03 9.0e+01 0.0e+00  0  0  4  0  0   0  0  6  0  0   758
MGResid Level 1       52 1.0 4.5318e-04 1.6 6.32e+04 3.5 1.0e+03 9.0e+01 0.0e+00  0  0  1  0  0   0  0  2  0  0   797
MGInterp Level 1     104 1.0 1.1601e-03 1.2 1.54e+04 3.7 1.4e+03 4.7e+01 0.0e+00  0  0  2  0  0   0  0  3  0  0    75
MGSmooth Level 2     104 1.0 2.3559e-03 1.4 7.49e+05 1.6 3.1e+03 1.7e+02 0.0e+00  0  0  4  0  0   0  0  6  0  0  1949
MGResid Level 2       52 1.0 8.1213e-04 2.3 2.15e+05 1.6 1.0e+03 1.7e+02 0.0e+00  0  0  1  0  0   0  0  2  0  0  1625
MGInterp Level 2     104 1.0 9.9738e-04 1.2 5.03e+04 1.7 1.4e+03 7.6e+01 0.0e+00  0  0  2  0  0   0  0  3  0  0   308
MGSmooth Level 3     104 1.0 5.0022e-03 1.9 2.51e+06 1.3 3.1e+03 3.2e+02 0.0e+00  0  0  4  0  0   0  0  6  1  0  3487
MGResid Level 3       52 1.0 2.3317e-03 4.2 7.24e+05 1.3 1.0e+03 3.2e+02 0.0e+00  0  0  1  0  0   0  0  2  0  0  2159
MGInterp Level 3     104 1.0 1.1775e-03 1.2 1.61e+05 1.2 1.4e+03 1.4e+02 0.0e+00  0  0  2  0  0   0  0  3  0  0   977
MGSmooth Level 4     104 1.0 1.8681e-02 2.3 9.14e+06 1.1 3.1e+03 6.3e+02 0.0e+00  0  0  4  1  0   0  0  6  1  0  3637
MGResid Level 4       52 1.0 6.5990e-03 3.8 2.64e+06 1.1 1.0e+03 6.3e+02 0.0e+00  0  0  1  0  0   0  0  2  0  0  2978
MGInterp Level 4     104 1.0 1.8010e-03 1.3 5.89e+05 1.1 1.4e+03 2.5e+02 0.0e+00  0  0  2  0  0   0  0  3  0  0  2474
MGSmooth Level 5     104 1.0 5.9776e-02 1.6 3.48e+07 1.1 3.1e+03 1.2e+03 0.0e+00  0  0  4  1  0   0  0  6  2  0  4486
MGResid Level 5       52 1.0 2.1561e-02 2.6 1.01e+07 1.1 1.0e+03 1.2e+03 0.0e+00  0  0  1  0  0   0  0  2  1  0  3601
MGInterp Level 5     104 1.0 5.2430e-03 1.8 2.26e+06 1.0 1.4e+03 4.9e+02 0.0e+00  0  0  2  0  0   0  0  3  0  0  3344
[...]
```

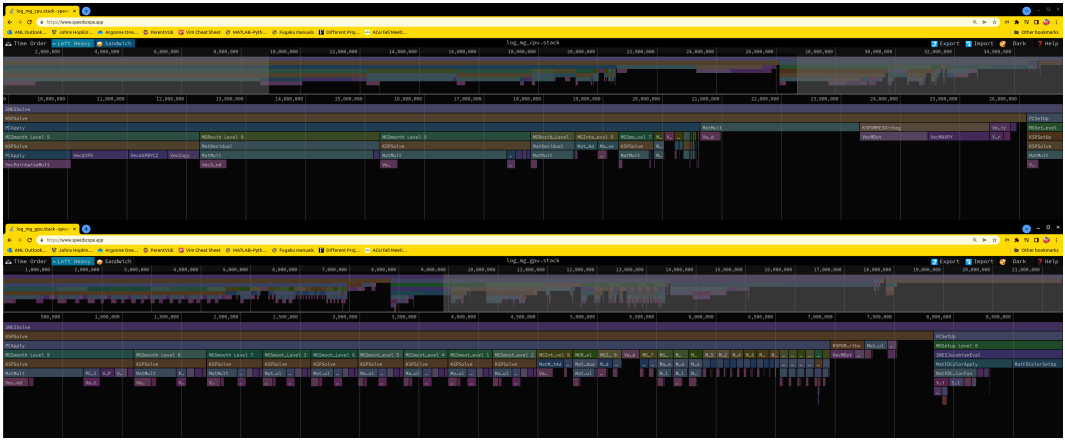
Additional Performance Logging Features: GPU

GPU logging

- When running with GPUs, get additional columns recording GPU flops and data transfers
- Use `-log_view_gpu_time` to get GPU logging for all events (incurs some expense)

```
------------------ ... ----------------------------------------
Event              ...  GPU    - CpuToGpu -   - GpuToCpu - GPU
                   ... Mflop/s Count   Size   Count   Size  %F
------------------ ... ----------------------------------------
5
...
KSPSetUp           ... 82806    112 1.00e+03   35 6.30e+01 100
KSPSolve           ... 119068    65 1.10e-01   65 1.15e-01 100
KSPGMRESOrthog     ... 135451     0 0.00e+00    0 0.00e+00 100
10 SNESSolve       ... 111924  1241 2.75e+03  631 9.30e+02  91
SNESSetUp          ...      0     0 0.00e+00    0 0.00e+00  0
SNESFunctionEval   ...      0     7 3.79e+01    4 3.80e+01  0
SNESJacobianEval   ...      0  1022 1.59e+03  527 8.29e+02  0
SNESLineSearch     ... 18807     6 5.68e+01    3 2.85e+01  71
15 ...
```

SNES ex19 CPU vs. GPU flame graph comparison



Download the *.stack files to your local machine and then use `https://www.speedscope.app` to generate flame graphs, which will let you examine (interactively) the hierarchy of PETSc events.

One thing to note is the relative distribution of time in `MGSmooth` steps (part of `PCApply`). See how the steps on the coarse levels all take roughly the same time on the GPU? This points to the high kernel launch latency. What other noteworthy differences can you find?

Further Things to Try with GPUs

Speedup (or slowdown!) for GPU vs. CPU

Total time in `SNESSolve` tells us the time required to solve our entire problem. Compare these for the CPU and GPU cases to get the overall speedup, but what parts sped up in the GPU case? Which parts actually slowed down? (The slowdowns are mostly due to the fact that the nonlinear function and Jacobian routines in SNES ex19 do not run on the GPU—verify this by looking at the `GPU %F` column in the text version of the log—and we had to use fewer ranks in this case. See SNES tutorial ex55 in the main development branch of PETSc for an example where these run on the GPU.)

If you'd like to try another GPU-back end, you can try PETSc's Kokkos/Kokkos Kernels one. Run with `-dm_mat_type aijkokkos -dm_vec_type kokkos`.

Experimenting with different multigrid cycle types

Our SNES ex19 runs defaulted to using multigrid V-cycles. Try running with W-cycles instead by using the option `-pc_mg_cycle_type w`. Unlike V-cycles, W-cycles visit coarse levels many more times than fine ones. What does this do to the time spent in multigrid smoothers for the GPU case vs. the CPU-only one? Should one or the other of these be favored when using GPUs?

Part II:

Survey of Applications of PETSc/TAO to Different Types of "Physics"

src/snes/tutorials/ex17.c

Linear elasticity

$$\int_{\Omega} \nabla \psi^c \cdot \mathbf{f}_1^c$$

$$\int_{\Omega} \nabla \psi^{c} \cdot \left(\mu \left(\nabla \mathbf{u} + \nabla \mathbf{u}^{T} \right) + \lambda \nabla \cdot \mathbf{u} \mathit{I} \right)$$

$$\int_\Omega \partial_d \psi^c \left(\mu \left(\partial_d u^c + \partial_c u^d \right) + \lambda \partial_e u^e \delta_{cd} \right)$$

Look at the code: `f1_elas_u`

$$\int_\Omega \nabla \psi^c \cdot \mathbf{g}_3^{c,c'} \cdot \nabla \psi^{c'}$$

$$\int_{\Omega} \partial_d \psi^c \cdot \mathbf{g3}^{c,c'}_{d,d'} \cdot \partial_{d'} \psi^{c'}$$

Look at the code: `g3_elas_uu`

Axial compression

$$\mathbf{n} \cdot \sigma = \mathbf{t}$$

develops uniform strain

Look at the code:

`f0_elas_axial_disp_bd_u` `axial_disp_u`

We need to construct

- mesh
- discretization
- equations
- boundary conditions
- parameters
- solver

Tests

- `3d_q2_trig_elas`
- `2d_q1_axial_elas`
- `2d_q3_axial_elas`
- `ge_q1_0`
- `ge_q1_gmg`
- `ge_q1_gdsw`

`src/snes/tutorials/ex77.c`

Large deformation elasticity

$$\frac{\mu}{2}\left(\mathrm{Tr}(C)-3\right)+Jp+\frac{\kappa}{2}\left(J-1\right)$$

Tests

- `2_par`

src/ts/tutorials/ex45.c

Heat equation

Tests

- `2d_p1_sconv_2`
- `2d_p1_tconv_2`
- `egads_sphere`

src/ts/tutorials/ex53.c

Biot poroelasticity

Tests

- `2d_terzaghi_sconv`
- `2d_terzaghi_tconv`

`src/dm/impls/swarm/tests/ex5.c`

Central orbits

We need to construct

- swarm
- equations
- initial conditions
- solver

Tests

- `bsi_2d_multiple_2`

src/dm/impls/swarm/tests/ex9.c

Landau damping

Tests

- `pp_poisson_bsi_2d_4`