October 29-31, 2024

# ALCF Hands-on HPC Workshop

# An Introduction to PETSc/TAO

Presented in two parts:
I. Solving numerical optimization with the Toolkit for Advanced Optimization (TAO)
II. Solving nonlinear equations with the Scalable Nonlinear Equation Solvers (SNES)

Hansol Suh, Argonne National Laboratory
October 31, 2024

# How to set it up

See ALCF Workshop git, petsc directory

For local machines, make sure to update `PETSC_DIR` in makefile



Argonne NATIONAL LABORATORY

# Goals and Agenda

- Introduce several key concepts and common patterns in PETSc by examples
  - 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 uses GPU accelerators
  - How to use the built-in logging framework to understand and tune performance
- *Toolkit for Advanced Optimization* (TAO):
  - Rosenbrock function example
- *Scalable Nonlinear Equation Solvers* (SNES):
  - Steady-State Nonisothermal Lid-Driven Cavity example

# 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).
  - Discord - link on the website

# PETSc - the Portable, Extensible Toolkit for Scientific computation

**Application Codes** **Higher-Level Libraries and Frameworks**

**TS** Time Steppers
- Pseudo-Transient, Runge-Kutta, IMEX, SSP, ...
- Local and Global Error Estimators
- Adaptive Timestepping
- Event Handling
- Sensitivity via Adjoints
- ...

**TAO** Optimization Solvers
- PDE-Constrained
- Adjoint-Based
- Derivative-Free
- Levenberg-Marquardt
- Newton's Method
- Interior Point Methods
- ...

**SLEPc** Eigensolvers

**SNES** Nonlinear Solvers
- Newton Linesearch
- Newton Trust Region
- BFGS (Quasi-Newton)
- Nonlinear Gauss-Seidel
- Successive Substitution
- Nonlinear CG
- Active Set VI
- ...

**KSP** Linear Solvers
- CG, GMRES, BiCGStab, FGMRES, ...
- Pipelined Krylov Methods
- Hierarchical Krylov Methods
- ...

**DM** Domain Management
- DMDA Regular Grids
- DMPlex Unstructured Meshes
- DMForest Forest-of-octrees AMR
- DMStag Staggered Grids
- DMNetwork Networks
- DMSwarm Particles

**PC** Preconditioners
- ILU/ICC
- Additive Schwarz
- Fieldsplit (Block Preconditioners)
- PCMG (Geometric Multigrid)
- GAMG (Algebraic Multigrid)
- ...

**Vec** Vectors

**IS** Index Sets

**Mat** Linear Operators
- AIJ (Compressed Sparse Row)
- SAIJ (Symmetric)
- BAIJ (Blocked)
- Dense
- GPU Matrices
- ...

**PetscSF** Parallel Communication

**Communication and Computational Kernels**

MPI | BLAS/LAPACK | Kokkos | CUDA | . . .

# What is optimization?

$$\underset{p}{\text{minimize}} \quad f(p)$$

- Optimization variables $p \in \mathbb{R}^n$
  - e.g.: boundary conditions, parameters, geometry
- Objective function $f : \mathbb{R}^n \to \mathbb{R}$
  - e.g.: lift, drag, total energy, energy norm, etc

# What is optimization?

$$\underset{p}{\text{minimize}} \quad f(p)$$

- Simplification: $f(p)$ is minimized where $\nabla_p f(p) = 0$
- **Gradient-free**: Heuristic search through $p$ space
  - Easy to use, no sensitivity analysis required
- **Gradient-based**: Find search direction based on $\nabla_p f$
  - Converges to local minima with significantly fewer function evaluations than gradient-free methods



Argonne NATIONAL LABORATORY

# Optimization - Outline

- Introduction to Gradient-Based Optimization
  - Sequential Quadratic Programming
  - PDE-Constrained Optimization
- Introduction to TAO
  - Sample main program
  - User/problem callback function
- Examples: Rosenbrock equation
  - 2D Unconstrained
  - Multidimensional unconstrained
  - 2D with general constraints

# Sequential Quadratic Programming

**for** $k = 0, 1, 2, \ldots$ **do**

$$\min_d \quad f_k + d^T g_k + 0.5 d^T H_k d$$

$$\min_\alpha \quad \alpha = f(p_k + \alpha d)$$

$$p_{k+1} \leftarrow p_k + \alpha d$$

**end for**

- Solution at $k^{\text{th}}$ iteration, $p_k$
- Gradient $g_k = \nabla_p(p_k)$
- Hessian $H_k = \nabla^2_{pp} f(p_k)$
- Search direction $d \in \mathbb{R}^n$
- Step length $\alpha$

- Replace original problem with a sequence of quadratic subproblems
  - Solution given by $d = -H_k^{-1} g_k$
- Line search maintains consistency between local quadratic model and global nonlinear function (globalization)
  - Avoids undesirable stationary points

# Sequential Quadratic Programming

**for** $k = 0, 1, 2, \ldots$ **do**

$$\min_d \quad f_k + d^T g_k + 0.5 d^T H_k d$$
$$\min_\alpha \quad \alpha = f(p_k + \alpha d)$$
$$p_{k+1} \leftarrow p_k + \alpha d$$

**end for**

- Solution at $k^{\text{th}}$ iteration, $p_k$
- Gradient $g_k = \nabla_p(p_k)$
- Hessian $H_k = \nabla^2_{pp} f(p_k)$
- Search direction $d \in \mathbb{R}^n$
- Step length $\alpha$

- Different approximations to search direction yields different algorithms
  - **Newton's method**: $d = -H_k^{-1} g_k$, no approximation
  - **Quasi-Newton**: $d = -B_k g_k$ with $B_k \approx H_k^{-1}$ based on secant condition
  - **Conjugate Gradient**: $d_k = -g_k + \beta_k d_{k-1}$ with $\beta$ defining different CG updates
  - **Gradient Descent**: $d = -g_k$, replace Hessian with identity

# PDE-Constrained Optimization

$$\underset{p,u}{\text{minimize}} \quad f(p)$$

$$\text{subject to} \quad R(p,u) = 0$$

**Full-Space Formulation**

- State variables $u \in \mathbb{R}^m$
- State equations $R : \mathbb{R}^{n+m} \to \mathbb{R}^m$

$$\underset{u}{\text{minimize}} \quad f(p, u(p))$$

**Reduced-Space Formulation**

- State variables implicit functions of parameters

- Reduced-space formulations enables use of conventional unconstrained optimization algorithms to solve PDE-constrained problems
- Each reduced-step function evalution requires a full PDE solution
- See ATPESC 2019 for more details slides, video



Argonne NATIONAL LABORATORY

# Toolkit for Advanced Optimization

- General-purpose continuous optimization toolbox for large-scale problems
  - Parallel (via PETSc data structures)
  - Gradient-based
  - Bound-constrained
  - PDE-Constrained problems w/ reducd-space formulation


Argonne NATIONAL LABORATORY

# TAO: The Basics

```
AptCtx   user;
Tao      tao;
Vec      sol;

PetscInitialize(&argc, &argv, (char *)0, help);
VecCreateMPI(PETSC_COMM_WORLD, user.n, user.N, &sol);
VecSet(sol, 0.0);

TaoCreate(PETSC_COMM_WORLD, &tao);
TaoSetType(tao, TAOBQNLS); /* bound-constrained quasi-Newton */
TaoSetSolution(tao, sol);
TaoSetObjective(tao, FormFunction, &user);
TaoSetGradient(tao,  FormGradient, &user);
TaoSetFromOptions(tao);    TaoSolve(tao);

VecDestroy(&sol);  TaoDestroy(&tao);  PetscFinalize();
```

# TAO: User Function

```
typedef struct { /* User-created context */
} AppCtx;

PetscErrorCode FormFunction (Tao tao, Vec p, PetscReal *fcn, void
    *ptr) {
  AppCtx *user = (AppCtx*) ptr;
  /* Evaluate f(p), store it in fcn */
  return 0;
}
PetscErrorCode FormGradient(Tao tao, Vec p, Vec g, void *ptr) {
  AppCtx *user = (AppCtx*) ptr;
  /* Evaluate gradient, and store it at g */
}
```

# TAO: User Function - Example

$$f(p) = \frac{1}{2} p^T A p$$

```
// f(p) = 0.5 * <p ,A p>, gradf(p) = Ap
typedef struct {
  Mat A;
  Vec workvec;
} AppCtx;
```

# TAO: User Function - Example

```
PetscErrorCode FormFunction (Tao tao, Vec p, PetscReal *fcn, void
    *ptr) {
  AppCtx *user = (AppCtx*) ptr;

  MatMult(user->A, p, user->workvec);   /* workvec = A*p        */
  VecDot(user->workvec, p, fcn);        /* fcn = <workvec, p> */
  *fcn *= 0.5;
  return 0;
}
PetscErrorCode FormGradient(Tao tao, Vec p, Vec G, void *ptr) {
  AppCtx *user = (AppCtx*) ptr;

  MatMult(user->A, p, g);  /* g = A*p */
  return 0;
}
```

# TAO: User Function

- **Objective evaluation**
  - Compute $f(p)$ at given $p$
- **Gradient evaluation**
  - Compute $g = \nabla_p f$ at given $p$
- **(ADVANCED) Second-order Methods**
  - Compute $H = \nabla_p^2 f$ at given $p$
  - Use `TaoSetHessian()` interface
- **(ADVANCED) Constraints**
  - Set bound constraints $p_1 \leq p \leq p_u$
  - Define nonlinear constraints $c_e(p) = 0, c_i(p) \geq 0$

# TAO: Bound Constraints

- What if we wanted to restrict the solution variables?

$$\underset{x}{\text{minimize}} \quad f(x)$$
$$\text{subject to} \quad x_l \leq x \leq x_u$$

```
VecSet(XL, PETSC_NINFINITY);
VecSet(XU, 0.0);
TaoSetVariableBounds(tao, XL, XU);
```

- Must use bound-constrained TAO algorithms
  - TAOBNLS: Bound Newton Line-Search
  - TAOBNTR: Bounded Newton Trust Region
  - TAOBQNLS: Bounded quasi-Newton Line-Search
  - TAOBNCG: Bounded Nonlinear Conjugate Gradient

# TAO: General Nonlinear Constraints

- Incorporate all constraint types

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & c_e(x) = 0 \\ & c_i(x) \geq 0 \\ & x_l \leq x \leq x_u \end{aligned}$$

- `TAOALMM` solver:
  - Augmented Lagrangian method
- `TAOPDIPM` solver:
  - Primal-dual Interior Point Method
- Define constraints with user call-backs
  - See manual

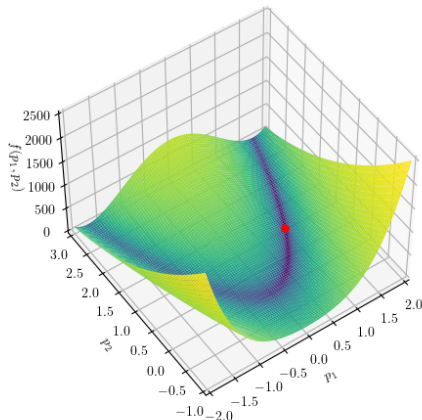


Argonne NATIONAL LABORATORY

# TAO Example: 2-dimensional Rosenbrock

$$\underset{p}{\text{minimize}} \quad f(p) = (1 - p_1)^2 + 100(p_2 - p_1^2)$$

- Global minimum at $p = (1, 1)$
- Also called the "banana function"
- Canonical test problem for optimizat algorithm
- Easy to find the valley, difficult to travese it towards the solution



# TAO Example: Multidimensional Rosenbrock

$$\underset{p}{\text{minimize}} \quad f(p) = \sum_{i=1}^{N-1} (1 - p_i)^2 + 100(p_{i+1} - p_i^2)^2$$

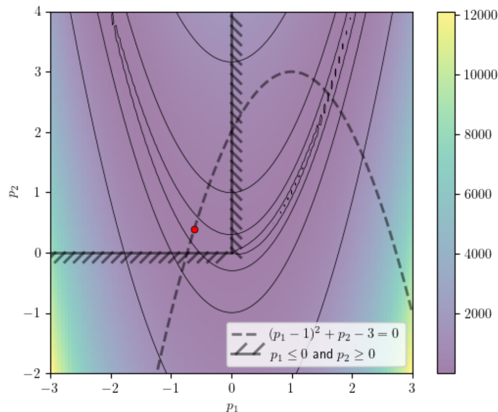
- Global minimum at $p_i = 1, \forall i = 1, 2, ..., N$
- Implementation supports parallel runs and provides analytical gradient and sparse Hessian
- TAO can compute sensitivities via finite-differencing when analytical derivatives are not available
  - Convenient for prototyping or debugging
  - Computationally expensive for large optimization problems or expensive objectives

# TAO Example: 2-dimensional Rosenbrock with Constraints

$$\underset{x}{\text{minimize}} \quad f(x) = \sum_{i=1}^{N-1} (1-p_i)^2 + 100(p_{i+1} - p_i^2)^2$$

$$\text{subject to} \quad (p_1 - 1)^2 + p_2 - 3 = 0$$
$$p_1 \leq 0, p_2 \geq 0$$

- Bound: $f(0,0) = 1$
- Equality: $f(-0.62, 0.38) = 2.62$, $f(1.62, 2.62) = 0.38$
- Combined: $f(-0.62, 0.38) = 2.62$
- Not valid for the multidimensional problem



# TAO: Take away

- PETSc/TAO offers parallel optimization algorithms for large-scale problems.
- Efficient gradients are needed for best results (e.g., algorithmic differentiation), and second-order methods don't always achieve faster/better solutions
- PETSc/TAO can automatically compute gradients via finite differencing
- PETSc/TAO can incorporate bound, equality and inequality constraints into the solution
- Most scientific problems are nonlinear and nonconvex... starting point matters


Argonne NATIONAL LABORATORY

# Iterative Solvers for Nonlinear Systems

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

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: $\|e_{k+1}\| \leq C\|e_k\|^{\alpha \geq 1}$


Argonne NATIONAL LABORATORY

# Newton' Method

- Standard form of a nonlinear system

$$F(u) = 0$$

- Iteration

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

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

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

- In practice, exact Newton step not desirable
- *Inexact Newton methods* find an approximate Newton direction $\Delta x_k$ that satisfies
  - $\|F'(x_k)\Delta x_k + F(x_k)\| \leq \eta \|F(x_k)\|$
- *Newton-Krylov methods*: uses Krylov subspace projection methods
- PETSc provides a wide range of Krylov methods and linear preconditioners
  - (`-ksp_type <ksp_method> -pc_type <pc_method>`)

# Globalization Strategies

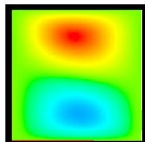
- Newton has quadratic convergence only when the iterate is sufficiently close to the solution.
- Far from the solution, the computed Newton step is often too large in magnitude
- In practice, some globalization strategy is often needed to expand the domain of convergence.
- PETSc offers several options; most common (and default) is backtracking line search.

# SNES: User Function

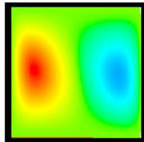
- Similar interface like `TAO`
- `FormFunction()`, set by `SNESSetFunction(SNES snes, Vec x, Vec r, void *ctx)`
- `FornJacobian()`, set by `SNESSetJacobian(SNES snes, Vec x, Mat J, Mat Jpre, void *ctx)`

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

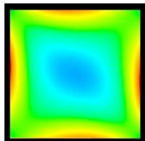
Solution Components



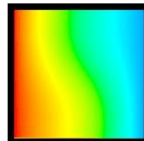
velocity: u



velocity: v



vorticity:



temperature: T

`src/snes/tutorials/ex19.c`

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

- 2D square domain with a moving lid; nonisothermal (temperature $T$)
- Flow driven by lid motion and buoyancy effects
- Velocity$(U, V)$-vorticity$(\Omega)$ formulation
- Finite difference discretization
- Logically regular grid
- Analytical Jacobian not provided; Calculated by finite-differences (using coloring)


Argonne NATIONAL LABORATORY

# SNES Example

```c
/*
   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;
```

# 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
```

# 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: Exact vs. Inexact Newton

(On Polaris):

- `-ksp_rtol 1e-8` : 2.157e+00
- `-ksp_rtol 1e-5` : 1.724e+00
- `-ksp_rtol 1e-3` : 1.638e+00
- `-ksp_rtol 1e-2` : 1.632e+00
- `-ksp_rtol 1e-1` : DIVERGED_LINE_SEARCH

# 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
```

- `-da_refine 2`: 2.485e+00
- `-da_refine 3`: 2.514e+00
- `-da_refine 4`: 4.994e+00

# 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.

```
$ 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
```


Argonne NATIONAL LABORATORY

# 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.

- `-da_refine 2` : 2.457e+00
- `-da_refine 3` : 2.598e+00
- `-da_refine 4` : 2.946e+00


Argonne NATIONAL LABORATORY

# 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.

```
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
```



Argonne NATIONAL LABORATORY

# 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

$$F(x) = b \tag{3}$$

and we define the residual as

$$r(x) = F(x) - b \tag{4}$$

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

$$x_{k+1} = x_k - P^{-1}(Ax_k - b) \tag{5}$$

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 $\alpha$ and $\beta$ may be written as

$$x_{k+1} = x_k - (\alpha P^{-1} + \beta Q^{-1})(Ax_k - b) \tag{6}$$

Analogously, for the nonlinear case, additive composition is

$$x_{k+1} = x_k + \alpha \cdot (\mathcal{M}(F, x_k, b) - x_k) + \beta \cdot (\mathcal{N}(F, x_k, b) - x_k) \tag{7}$$

# Nonlinear Composition: Multiplicative

A multiplicative combination of linear preconditioners may be written as

$$\begin{aligned} x_{k+1/2} &= x_k - P^{-1}(Ax_k - b), \\ x_{k+1} &= x_{k+1/2} - Q^{-1}(Ax_{k+1/2} - b), \end{aligned} \tag{8}$$

Analogously, for the nonlinear case

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

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

$$x_{k+1} = x_k - P^{-1}(Ax_k - b) \tag{10}$$

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

$$\mathcal{N}(F, x, b) = x_k - P^{-1}r \tag{11}$$

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

$$P^{-1}r = x_k - \mathcal{N}(F, x, b) \tag{12}$$

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

$$r_L = x_k - \mathcal{N}(F, x, b) \tag{13}$$

# Nonlinear Right Preconditioning

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

$$\begin{aligned} AP^{-1}y &= b \\ x &= P^{-1}y \end{aligned} \tag{14}$$

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

$$\begin{aligned} y &= \mathcal{M}(F(\mathcal{N}(F, \cdot, b)), x_k, b) \\ x &= \mathcal{N}(F, y, b) \end{aligned} \tag{15}$$

(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()`.


Argonne NATIONAL LABORATORY

# 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
```


Argonne NATIONAL LABORATORY

# 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
```



Argonne NATIONAL LABORATORY

# 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
...
 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
```

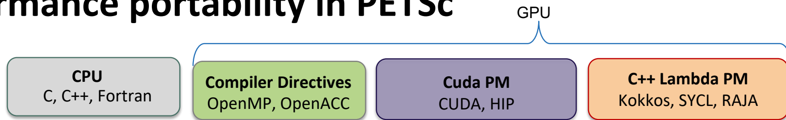
```
lid velocity = 100., prandtl # = 1., grashof # = 1e+06
...
 70 SNES Function norm 3.238739735055e+00
  Linear solve converged due to CONVERGED_RTOL iterations 1
 71 SNES Function norm 1.781881532852e+00
  Linear solve converged due to CONVERGED_RTOL iterations 1
 72 SNES Function norm 1.677710773493e-05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

# GPU

How can you run PETSc with GPU?

# Performance portability in PETSc




## Application code

Using PETSc API

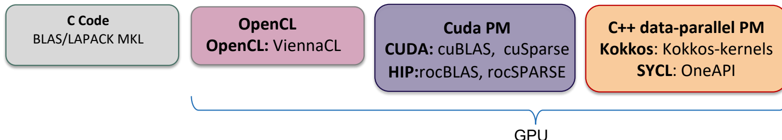
Front-end PETSc vector and matrix arrays are shared with user programming language/model

## 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



# PETSc with GPU: Basic Idea, User Side

```
TaoSetObjective(tao, CPUObj, &ctx) -> TaoSetObjective(tao, GPUobj, &ctx)
TaoSetGradient(tao, GPUgrad, &ctx) -> TaoSetGradient(tao, GPUgrad, &ctx)

SNESSetFunction(snes, r, CPUFunc, &ctx) -> SNESSetFunction(snes, r, GPUFunc, &ctx)
SNESSetJacobian(snes, r, CPUJc, &ctx) -> SNESSetJacobian(snes, r, GPUJc, &ctx)
```

# PETSc with GPU: Basic Idea, User Side

```
TaoSetObjective(tao, CPUObj, &ctx) -> TaoSetObjective(tao, GPUobj, &ctx)
TaoSetGradient(tao, GPUgrad, &ctx) -> TaoSetGradient(tao, GPUgrad, &ctx)

SNESSetFunction(snes, r, CPUFunc, &ctx) -> SNESSetFunction(snes, r, GPUFunc, &ctx)
SNESSetJacobian(snes, r, CPUJc, &ctx) -> SNESSetJacobian(snes, r, GPUJc, &ctx)
```

- Need to be careful with memory location, efficient kernel launches, etc
- see (here) for more details

# 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 1: 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`.

# 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**.

# PETSc with GPU: Basic Idea, Back-end Side

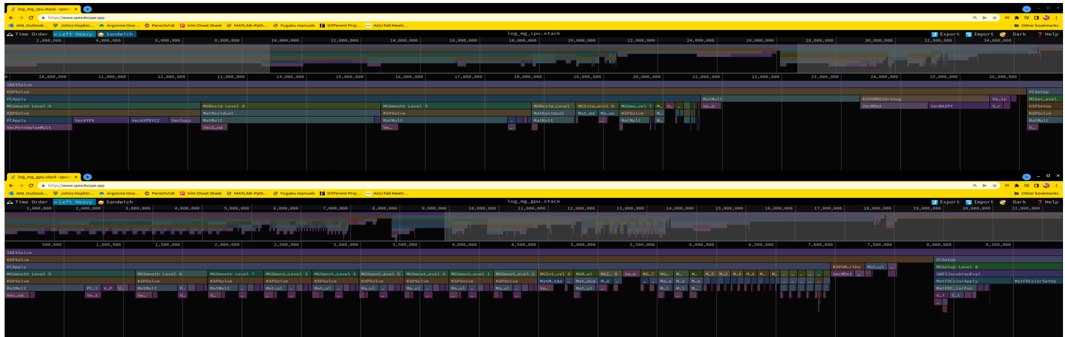
Run a large version of the driven cavity problem, using multigrid with GPU and SIMD-friendly Chebyshev-Jacobi smoothing (-mg_levels_pc_type jacobi), and collect a breakdown by multigrid level (-pc_mg_log), both in plain text and flamegraph stack formats.

## GPU Example, with performance logging

```
mpiexec -n 1 ./ex19 -da_refine 4 -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 4 -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
```

# 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?

# Summary

- TAO can solve various optimization problems
  - Gradient, AD, Hessian, bounds, etc...
- SNES can solve nonlinear system
  - Composition of solvers matter.
  - What preconditioner? Solver type? Inner iteration? Tolerance?
- GPU
  - Change matrix and vector memtype
  - Think about memory location, kernel launches...
- Profiling
  - log view, Flamegraph

Thanks!