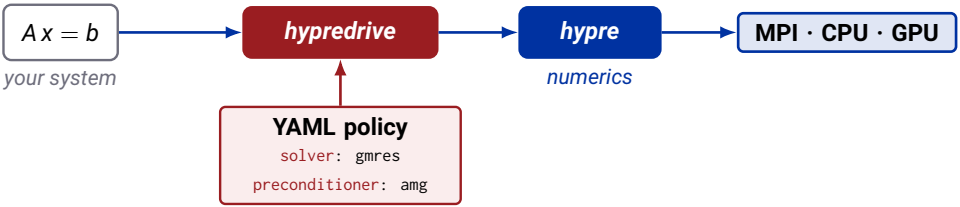
Iterative linear solvers and preconditioners with *hypre*

. . . and how to drive them: solver policy as runtime configuration with *hypredrive*

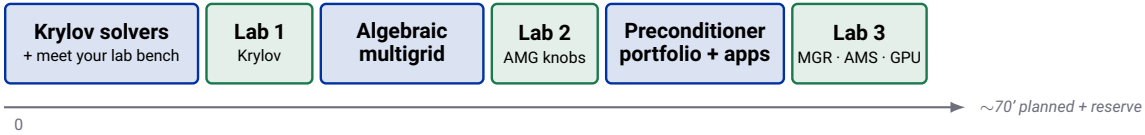


Victor A. P. Magri · Center for Applied Scientific Computing, LLNL · ATPESC 2026

The algorithms live in *hypre*; *hypredrive* is a higher-level interface that moves the solver *policy* out of application code.

LLNL-CODE-861860

How we'll spend the next 80 minutes



hypre: the numerics

Parallel preconditioners & Krylov solvers: Boomer-AMG, MGR, Schwarz, ILU, FSAI, AMS/ADS, PFMG… MPI + CPU/GPU.

hypredrive: flexible runtime control for _hypre_

A YAML-configured CLI + library around *hypre*; every experiment in the labs is one config edit, no recompiling.

Three lecture blocks, three hands-on blocks; laptops out: you will run a representative subset of the solver/preconditioner choices we discuss.

One session, two tools: *hypre* provides the solvers; *hypredrive* makes trying them a one-line edit.

Sparse iterative solvers: why multigrid exists

Part 1

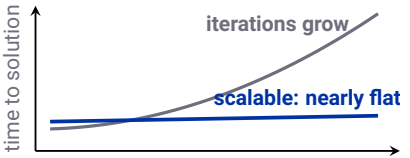
Solving $Ax = b$ at scale: the O(N) ambition

Sparse direct

Robust, but **fill-in** grows memory and work super-linearly; out of reach at 10^8–10^9 unknowns.

Krylov alone

$O(\mathrm{nnz})$ per iteration; but the **iteration count grows** with problem size.



Weak scaling (schematic): same work per rank; real HPC overheads cause the slight rise.

Multigrid-preconditioned Krylov: the takeaway

$O(N)$ **work per solve** (given a bounded hierarchy), iterations $\approx$ **mesh-independent** on suitable problems; *earned* by exploiting problem structure. The rest of the session makes that real.

hypre: scalable preconditioners, 25+ years in production

LLNL CASC · since 1998
Apache-2.0 / MIT · C / Fortran
github.com/hypre-space/hypre

MIT · JOSS 2024 · YAML policies
github.com/hypre-space/hypredrive
CLI + embeddable library

parallel-in-time via MGRIT
wrap an existing time stepper
github.com/XBraid/xbraid

What *hypre* is

High-performance preconditioners and Krylov solvers for large sparse linear systems, with **multigrid** at the center. Designed for *scalable linear solves* inside scientific applications deployed on large-scale supercomputers (HPC).

Where it runs · who uses it

Runs with MPI; GPUs via CUDA, HIP, and SYCL. Demonstrated at **100.7B unknowns on Tuolumne** (MPI + HIP). Used in subsurface, CFD, mechanics, fusion and more; directly or through PETSc, MFEM, deal.II, AMReX, etc...


Rob Falgout


Rui Peng Li


Victor P. Magri


Wayne Mitchell

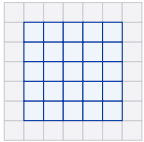

Daniel Osei-Kuffuor

today's core team (LLNL CASC), with **50+ developers** have contributed over the years photos: computing.llnl.gov

hypre = the multigrid-centric linear solver library that you can easily plug into your application code.

Conceptual interfaces define matrix storage and algorithms

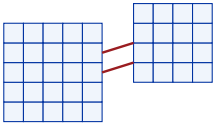
Struct (structured)



owned box ghost layer

- grid = union of boxes; bulk halo exchange
- one contiguous array per stencil entry → tight loops
- setup "already knows its neighbors"; GPU-ideal layout

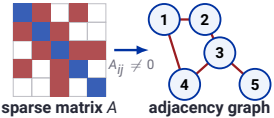
SStruct (semi-structured)



structured *parts* + graph at the seams

- block-structured, AMR, overset meshes
- stencil speed where structure exists; graph entries only where it doesn't
- ParCSR view via `SetObjectType` + `GetObject`

IJ (linear-algebraic)



sparse matrix *A* adjacency graph

- totally general: any graph, any mesh
- distributed CSR-based data structure
- indirect addressing + integer traffic in every kernel
- setup must *discover* neighbors-of-neighbors (*RAP*); comm-heavy

Native: SMG · PFMG

Native: Split · SysPFMG · SSAMG
Alternate: ParCSR view

ParCSR: BoomerAMG · AMS · ADS · MGR
Schwarz · ILU · FSAI

Krylov: PCG · (F)GMRES · BiCGSTAB on every lane. ***hypredrive*** **uses IJ → ParCSR.**

Declare structure when you have it: the interface sets storage, setup cost, and available solvers.

Krylov methods: four workhorses, one anatomy

PCG	needs SPD A *and* SPD precond. · 3-term recurrence: 1 matvec, 2 dots/iter; cheapest
GMRES(k)	general A · most robust · stores k vectors: restart trades memory vs. convergence
BiCGSTAB	general A · short recurrence ($\approx 2\times$ CG work/iter) · can be erratic
FGMRES	tolerates a *variable* preconditioner

One iteration, three parallel costs

mat-vec neighbor comm	**dot products** global reductions
precond apply the real work	**axpy updates** no comm (bandwidth)

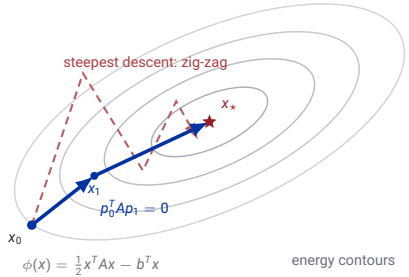
Preconditioner: cheap $B \approx A^{-1}$ for fewer Krylov iterations.

Rule of thumb. SPD $\Rightarrow$ PCG. Nonsymmetric $\Rightarrow$ GMRES first, BiCGSTAB if memory-bound. Variable preconditioner $\Rightarrow$ FGMRES. And check *which* residual is reported (preconditioned vs. true) before comparing tolerances.

Krylov methods are the accelerator; all exposed through one *hypredrive* keyword: `solver:`.

CG: one new direction without undoing the old ones

SPD linear solve = minimize a convex quadratic



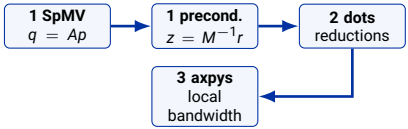
In two dimensions CG reaches $x_\star$ in at most two steps in exact arithmetic; steepest descent can keep correcting the same elongated directions.

The optimality statement

$$x_k = \underset{x \in x_0 + \mathcal{K}_k}{\operatorname{argmin}} \|x - x_\star\|_A$$
$$\mathcal{K}_k = \operatorname{span}\{r_0, Ar_0, \ldots, A^{k-1}r_0\}$$

CG chooses the smallest A-norm error in the growing space. A-conjugate directions preserve earlier minimizations; exact arithmetic finishes in at most n steps.

PCG: short recurrence, fixed memory



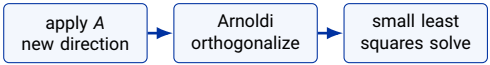
PCG buys optimal-in-energy progress with fixed memory; but only when both A and the preconditioner are SPD.

GMRES: grow a search space, then decide when to restart

Shown without preconditioning for clarity.

$$r_0 = b - Ax_0, \qquad \mathcal{K}_k(A, r_0) = \mathrm{span}\{r_0, Ar_0, \ldots, A^{k-1}r_0\}$$

$$x_k = \underset{x \in x_0 + \mathcal{K}_k}{\mathrm{argmin}} \; \|b - Ax\|_2$$



Within one unrestarted cycle, the search space only grows, so the residual norm cannot increase in exact arithmetic.

Rule of thumb. Fixed preconditioner: GMRES. Changing preconditioner: **FGMRES**.

What the basis costs

Store k full vectors. Orthogonalization touches the growing basis and needs dot products: bandwidth work plus **global reductions**.

What GMRES(k) throws away

Restart after k directions: retain x_k, discard the basis, rebuild. Work stays bounded, but useful directions vanish; a small k can stall. **Lab 1:** $k = 5$ took 446 iterations vs. 121 at the default.

Restart length is a three-way trade: memory, global communication, and convergence.

Preconditioning is where the scalability lives

Precondition with $B \approx A^{-1}$, cheap to build and apply:

$B = I$	Richardson; rarely converges alone (model-problem intuition)
$B = D^{-1}$	Jacobi; helps for strongly diagonally dominant problems
$M = \tilde{L}\tilde{U} \approx A$	ILU: apply $B = M^{-1}$ by triangular solves
$B = G^T G \approx A^{-1}$	FSAI (for SPD A): approximate inverse, SPD by construction
B = V-cycle	one AMG cycle; $\approx$ mesh-independent iterations

3-D Poisson, PCG, same tolerance

	fsai	amg
40^3 = 64k rows	52	**7**
80^3 = 512k rows	92	**7**

iterations to 10^{-8} (fixed FSAI sparsity); you will reproduce this in Labs 1–2. PCG needs an SPD preconditioner; FSAI qualifies; pair generic ILU with GMRES.

Three tests for a good preconditioner

Spectrally effective — clusters the spectrum of BA.

Cheap end to end — low setup and application cost.

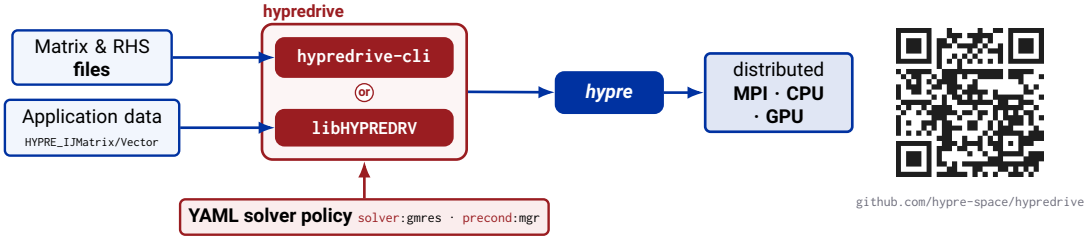
Parallel by design — concurrent, low-sync kernels.

Fixed-sparsity preconditioners lose ground as N grows; multigrid is how iterations stay $\approx$ flat.

hypredrive: solver policy as runtime configuration

Your lab bench

hypredrive provides flexible runtime control of *hypre* solvers



Swap strategies one config line

Reuse setup amortize build

Capture statistics setup·solve·iters

Solvers PCG · (F)GMRES · AMG · ILU · FSAI · MGR · Schwarz · AMS · ADS

Solver strategy becomes shareable configuration, not application code.

hypredrive: The same solve from six language bindings

C

```c
HYPREDRV_Create(comm, &hd);
HYPREDRV_InputArgsParse(1, yaml, hd);
HYPREDRV_LinearSolverSetup(hd);
HYPREDRV_LinearSolverApply(hd);
```

C++

```cpp
hypredrive::driver hd(comm);
hd.parse_yaml(yaml);
hd.set_matrix(A);
hd.solve();
```

Fortran

```fortran
call HYPREDRV_Create(comm, hd)
call HYPREDRV_InputArgsParse(hd, yaml)
call HYPREDRV_LinearSolverSetup(hd)
call HYPREDRV_LinearSolverApply(hd)
```

Python

```python
import hypredrive as hd
res = hd.solve(A, b,
    options={"solver": {"pcg": {}}})
x = res.x
```

MATLAB / Octave

```matlab
opts.solver = 'pcg';
opts.preconditioner = 'amg';
[x, info] = ...
   hypredrive_solve(A, b, opts);
```

Julia

```julia
using HypreDrive
x, info = hypredrive_solve(A, b)
# options=Dict(...) to choose
```

Build note: Only the C interface is enabled by default; enable the other language bindings at build time. See **interfaces** for more info.

All six bindings drive the same YAML-configured *hypre* solve.

How a *hypre* solve is composed: where the policy lives

Direct *hypre*: compose · set · setup · solve

```
HYPRE_BoomerAMGCreate(&pc);
HYPRE_BoomerAMGSetCoarsenType(pc, 10);
HYPRE_BoomerAMGSetStrongThreshold(pc, 0.5);
HYPRE_BoomerAMGSetCycleRelaxType(pc, 18, 1); // down
HYPRE_BoomerAMGSetCycleRelaxType(pc, 18, 2); // up
HYPRE_ParCSRPCGCreate(comm, &solver);
HYPRE_PCGSetTol(solver, 1e-8);
HYPRE_ParCSRPCGSetPrecond(solver, ..., pc);
HYPRE_ParCSRPCGSetup(solver, A, b, x);
HYPRE_ParCSRPCGSolve(solver, A, b, x);
/* each knob: exposed by the app itself */
```

hypredrive: the same knobs as data

```
solver:
  pcg:
    relative_tol: 1.0e-8
preconditioner:
  amg:
    coarsening:
      type: HMIS
      strong_th: 0.5
    relaxation:
      down_type: l1-jacobi
      up_type: l1-jacobi
```

The native lifecycle: create, set **policy**, attach, setup, solve; *hypredrive* standardizes schema, validation, lifecycle, reporting.

Direct *hypre* = full low-level control; *hypredrive* standardizes the *repeated reconfiguration* layer.

Krylov methods in practice (live build + exercises, ∼10 min)

Hands-on 1

Lab setup: one tree, two reusable Polaris builds

```
# (Required) Unpack tarball shared via Slack
$ tar xzvf atpesc26-track_4-hypre.tar.gz && cd atpesc26-track_4-hypre
# (Optional) Full checkout: configure on login (network)
$ bash scripts/build.sh --config
# (Optional) Compile/install in allocation (takes about 5 min)
$ bash scripts/build.sh
# (Required) For the sake of time, use the pre-compiled binaries
$ export BASE_DIR=/eagle/ATPESC2026/EXAMPLES/track-5-numerical/krylov_amg_hypre
# (Required): run experiments on a compute node
$ qsub -I -l select=1 -l filesystems=home:eagle -l walltime=1:00:00 -q ATPESC -A ATPESC2026
```

CPU product · 32-core AMD EPYC Milan

Binaries: `$BASE_DIR/drivers/polaris-cpu/`

MPI + host execution · accelerator backends off

Launch: 32 MPI ranks/node

GPU product · 4 NVIDIA A100s

Binaries: `$BASE_DIR/drivers/polaris-gpu/`

MPI + CUDA · `sm_80` device execution

Launch: 4 MPI ranks/node · one rank/A100

datasets
`ps3d10pt7` · `compflow6k`
`poromech2k`

complete targets
all mini-apps + data
HYPRE `ij` · `struct` · `sstruct`

one policy language
same `solver:` + `preconditioner:` YAML

Configure/fetch on the login node; build and run experiments inside the compute allocation.

1 Your first solve and the statistics table

Check solver options via `hypredrive-cli` helper

```
$ ./build-polaris-cpu/hypredrive-cli -h

Usage: hypredrive-cli [options] <input.yml>

List of valid sections for input.yml:

Valid keys:
general         <section> Global configuration settings
linear_system   <section> Linear system settings
solver          <section> Linear solver settings
preconditioner  <section> Preconditioner settings

Nested topics:
  general  linear_system  solver  preconditioner

Examples (use ":" for nested options):
$ ./build-polaris-cpu/hypredrive-cli -help \
  solver:gmres
$ ./build-polaris-cpu/hypredrive-cli -help \
  preconditioner:mgr:level:f_relaxation
```

Solve a 7-point Laplacian $(-\Delta u = f)$ discretization on a 30^3 cube

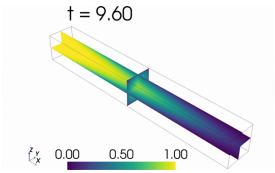
```
$ mpirun -n 1 ./build-polaris-cpu/laplacian -ns 1 -n 30 30 30 -i lab/pcg-jacobi.yml

STATISTICS SUMMARY:
+-------+-----------+-----------+-----------+------------+------------+--------+
|       |  LS build |     setup |     solve |    initial |   relative |        |
| Entry | times [s] | times [s] | times [s] |  res. norm |  res. norm |  iters |
+-------+-----------+-----------+-----------+------------+------------+--------+
|     0 |     0.008 |     0.000 |     0.035 |   3.00e+01 |   9.82e-09 |    100 |
+-------+-----------+-----------+-----------+------------+------------+--------+
```

`lab/pcg-jacobi.yml`: PCG to 10^{-8}, preconditioned by Jacobi.

Try it. Append `-a --solver:pcg:max_iter 50`; stopping above 10^{-8} confirms nonconvergence.

PCG + Jacobi is a simple valid baseline; the table records its cost and convergence.

2 Convection–diffusion: when CG's contract is broken



Flow evolution with time.

$$\partial c/\partial t + \boldsymbol{u}\cdot\nabla c - \kappa\Delta c = 0$$

```
$ CD="mpirun -n 1 ./build-polaris-cpu/convdif -n 64 16 16 \
      -k 1e-4 -w 2 -nt 10 -v 1"
$ $CD -i lab/convdif-gmres-amg.yml
$ $CD -i lab/convdif-pcg-amg.yml
```

Upwinding breaks symmetry: $a_W = -(D + C)$ but $a_E = -D$, hence $A \neq A^T$. At $\mathrm{Pe}_h \approx 625$, convection dominates.

Recorded; same operator, same AMG, one CPU rank

outer solver	total iters	per step	outcome
GMRES(30)	**84**	8–9	all 10 converged
PCG	854	up to 100	8/10 hit the cap

Observe. CG requires an **SPD** operator. Here PCG stalls; GMRES is valid and residual-minimizing.

Solver choice begins with matrix class: SPD ⇒ PCG; nonsymmetric ⇒ GMRES or BiCGSTAB.

3 Grow the problem and hit the Krylov wall

```
$ LAP="./build-polaris-cpu/laplacian -ns 1"
$ mpirun -n 1 $LAP -n 40 40 40 -i lab/pcg-fsai.yml -v 1
$ mpirun -n 1 $LAP -n 40 40 40 -i lab/gmres-fsai.yml -v 1
$ mpirun -n 1 $LAP -n 40 40 40 -i lab/bicgstab-fsai.yml -v 1
$ mpirun -n 1 $LAP -n 80 80 80 -i lab/pcg-fsai.yml -v 1
$ mpirun -n 1 $LAP -n 80 80 80 -i lab/gmres-fsai.yml -v 1
$ mpirun -n 1 $LAP -n 80 80 80 -i lab/bicgstab-fsai.yml -v 1
$ mpirun -n 1 $LAP -n 80 80 80 -i lab/gmres-fsai-k5.yml -v 1
```

Seven runs use four configs that differ only in the `solver:` block (`krylov_dim: 5` for the restart study); FSAI preconditioner throughout.

iterations to 10^{-8} (FSAI precond.)

	40^3	80^3
PCG	52	92
GMRES(30)	59	121
BiCGSTAB	38	66
GMRES(5)	n/a	446

Observe. Iterations grow: PCG and GMRES nearly double, BiCGSTAB +74%, and GMRES(5) is catastrophic. Part 2 fixes the preconditioner.

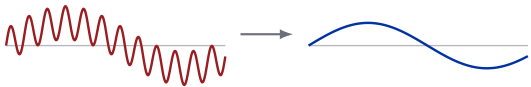
Krylov methods paired with single-level preconditioners cannot keep iterations flat as the mesh grows.

Algebraic multigrid: flat iterations by design

Part 2

The multigrid idea: every error scale is cheap somewhere

1 · Smoothing kills the oscillatory error

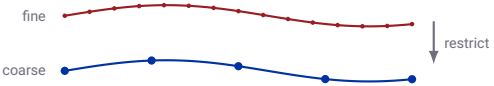


error after 1 Jacobi sweep

after a few sweeps: **smooth**

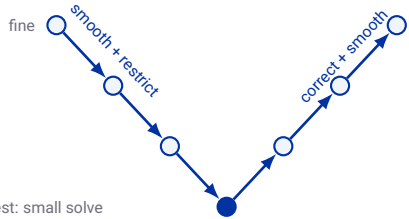
2 · Smooth error lives happily on a coarser grid

Represent the same smooth mode with fewer points; solve there *cheaply*, interpolate back, and recurse.



Key: smooth error on the fine grid appears more oscillatory—higher frequency in grid units—on the coarse grid.

The V-cycle



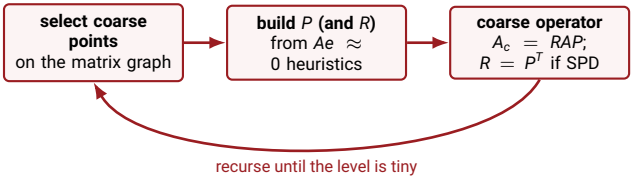
Coarse grids cost pennies: with ideal 3-D full coarsening each level is $\sim 8\times$ smaller, so one V-cycle $= O(N)$ work. The coarsest solve may be direct or approximate. In parallel, the V-cycle is the robust default.

Other cycle families: W-, F-, and K-cycles trade additional coarse-grid work and communication for a stronger correction.

Smoothers kill oscillatory error, coarse grids the smooth part; recursively: that is $O(N)$.

AMG needs no grid: the matrix graph *is* the grid

SETUP phase: build the hierarchy (from A alone)



SOLVE phase: run V-cycles

smooth → restrict → … → coarse solve → interpolate → smooth, using only matvec-like kernels.

Black-box to *use*…

You hand BoomerAMG a matrix through the IJ interface; no mesh, no geometry.

…not black-box to *tune*

Coarsening, interpolation, and smoothing choices decide both convergence *and* cost; the defaults are chosen with care, and those three knob families are the next four slides.

AMG builds its “grids” from the matrix graph; setup is a real cost, paid before iteration one.

The AMG landscape: two complementary families

Classical (C/F) AMG

Coarse points are a **subset** of the fine points, chosen on the strong-connection graph; interpolation built from strong neighbors.

Reference implementation: ***hypre* BoomerAMG**, today's workhorse.

Aggregation AMG

Group fine points into aggregates; each aggregate becomes one coarse point (tentative prolongator + smoothing).

Trilinos **ML / MueLu**, PETSc **GAMG**.

Mature, actively developed libraries that interoperate: PETSc, MFEM, deal.II, AMReX can all call *hypre*.

Benchmark on *your* operator; the family matters less than the fit to your problem.

Coarsening: keep the points that matter

Why strong edges? Relaxation leaves algebraically smooth error.

$$Ae \approx 0 \quad \Longrightarrow \quad e_i \approx -\frac{1}{a_{ii}} \sum_{j \neq i} a_{ij} e_j$$

The influential neighbors must be represented in the coarse space.

Classical scalar *M*-matrix test: j strongly influences i when

$$-a_{ij} \geq \theta \max_{k \neq i} (-a_{ik})$$

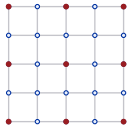
Directed influence: $j \to i$ need not imply $i \to j$. Strong means important for smooth-error interpolation, not merely large $|a_{ij}|$.

Threshold. Larger θ filters edges and changes C/F, *P*, *RAP*, communication, and C_{op}; not automatically better.

Starting points. 0.25 (2-D), 0.5–0.6 (3-D), ∼0.9 (elasticity); tune with interpolation.

Structure. Anisotropy preserves dominant directions; systems need `num_functions`, nodal coarsening, or MGR.

C/F splitting on the filtered strong graph



• C (coarse) ○ F (fine)

Coverage: retain enough C-points to represent every strong F-point dependency—directly or through strong F-neighbors—without keeping redundant nearby points. *P* reconstructs F-error from C-values.

Classical (Ruge–Stüben, Falgout): strong coverage, higher complexity.
PMIS/HMIS: leaner parallel grids; pair with long-range interpolation. Defaults are build-dependent: HMIS on CPU builds, PMIS on GPU builds.

Coarsen the graph seen by smooth error: cover every strong F-dependency with as few C-points as possible.

Interpolation: the knob between accuracy and sparsity

P rebuilds fine-level error from coarse points ($Ae \approx 0$ heuristic, strong neighbors):

classical — `standard`: natural partner of Ruge–Stüben / Falgout C/F splittings

long-range — `extended+i` (distance-2): partner of lean PMIS/HMIS grids

truncation — `trunc_factor` / `max_nnz_row` cap P's width → sparser *RAP*

aggressive — multipass / 2-stage P on aggressive levels

In general, we look for

$$P \approx \begin{bmatrix} -A_{FF}^{-1} A_{FC} \\ I \end{bmatrix}$$

interpolation setting in *hypredrive*

```
preconditioner:
  amg:
    interpolation:
      prolongation_type: extended+i
```

Rule of thumb. Pair RS/Falgout + `standard`; PMIS/HMIS + `extended+i`. Tune P's width: richer interpolation improves convergence; truncation lowers *RAP* density until convergence suffers.

Coarsening picks *where*; P decides *how well* and drives the next slide's C_{op}.

Operator complexity: the price of a rich hierarchy

The coarse product *RAP densifies* levels ($R = P^T$ in the SPD Galerkin case); memory and matvec cost follow

$$C_{\text{op}} = \frac{\sum_\ell \text{nnz}(A_\ell)}{\text{nnz}(A_0)}$$

Rule of thumb. $C_{\text{op}} < 2.5$ is a starting heuristic, not a pass/fail test. Levers: `aggressive: num_levels`, interpolation truncation (previous slide), stronger θ.

80^3 **Poisson, PCG+AMG: aggressive coarsening**

agg. levels	C_{op}	iters	setup	total
0 (default)	3.25	7	873 ms	1210 ms
1	**1.37**	12	548 ms	**864 ms**
2	1.22	18	442 ms	885 ms

1 MPI rank, Polaris AMD EPYC Milan; you will reproduce this in Lab 2.

One aggressive level: 2.3× leaner operator, cheaper setup, more iterations (7 → 12), and the **best total time**.

Complexity is the other half of convergence; the best *total time* often accepts extra iterations.

Smoothers: cheap, parallel, and GPU-aware

hybrid (l1-)GS	CPU-build default: GS *inside* a rank, Jacobi across; true GS is sequential
l1-Jacobi	GPU-build default; convergent smoother for SPD problems, fully parallel
Chebyshev	matvecs only; strong *and* GPU friendly
ILU / FSAI	heavyweight smoothing for tough problems

relaxation choices in *hypredrive*

```
preconditioner:
  amg:
    relaxation:
      down_type: chebyshev
      up_type: chebyshev
```

80^3 Poisson: Chebyshev matches the default's 7 iterations at higher cost per sweep; its value is being *matvec-shaped* (GPUs). Total time decides.

$$x^{(k+1)} = x^{(k)} + M^{-1}\left(b - Ax^{(k)}\right), \qquad e^{(k+1)} = \left(I - M^{-1}A\right)e^{(k)}$$

Rule of thumb. One V-cycle per Krylov iteration is a robust default; BoomerAMG also runs standalone. With PCG, keep the cycle *symmetric*.

Smoothing is a convergence–cost–hardware triangle: on GPUs, prefer matvec-shaped smoothers.

One A100 versus all 32 CPU cores on a Polaris node

Same work

Same global 7-point Poisson operator, PCG+AMG policy, tolerance, and one Polaris node (32 CPU cores vs 1 A100 GPU).

Warm lifecycle

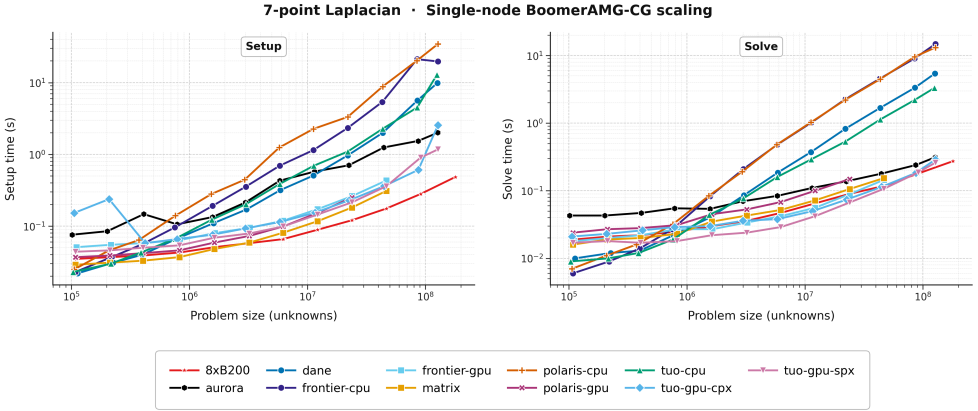
Median of solves 2–5; setup + apply only. Matrix assembly and first-use warm-up are excluded.

```
# 192^3 · 32 CPU ranks
$ mpirun -n 32 ./build-polaris-cpu/laplacian \
  -n 192 192 192 -ns 5 -P 4 4 2 \
  -i lab/pcg-amg.yml -v 1

# 192^3 · 1 A100
$ mpirun -n 1 ./build-polaris-gpu/laplacian \
  -n 192 192 192 -ns 5 -P 1 1 1 \
  -i lab/pcg-amg-gpu.yml -v 1
```

problem	resource	setup	solve	setup+solve	iters
		milliseconds			
128^3	32 CPU ranks	303	142	445	10
128^3	**1 A100**	**58**	**41**	**99**	14
192^3	32 CPU ranks	1004	711	1715	11
192^3	**1 A100**	**165**	**120**	**285**	15

One A100 wins this solver phase by 4.5× at 128^3 and 6.0× at 192^3, despite extra iterations.

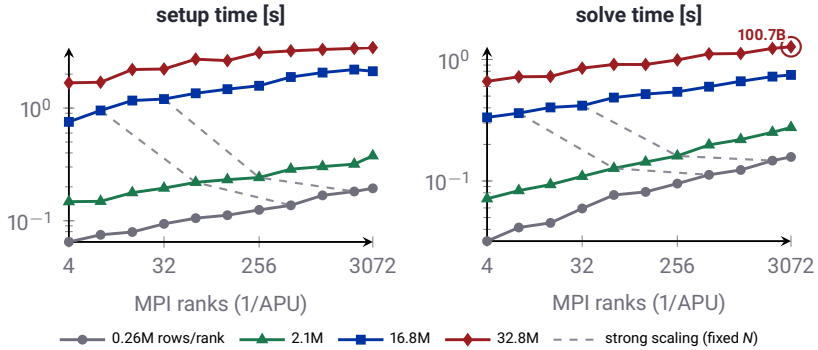
One full node: where accelerators pull away



One full node per system; log–log axes. **Methodology, machine layouts, and reproducibility script.**

Small problems are overhead-dominated; as *N* grows, accelerator nodes generally pull away in both AMG setup and CG solve.

. . . and measured across nodes: *hypre* on Tuolumne



BoomerAMG-PCG, 27-pt 3D Poisson · MPI + HIP, one rank per MI300A APU (4/node)
Tuolumne (El Capitan-class, LLNL): 1 → 768 nodes = 75% of the machine

At 768 compute nodes

100.7B unknowns
(3072 ranks × 32.8M)

768× the ranks:
setup ×2.1 · solve ×1.9

iterations 22 → 33

parallel efficiency:
setup 49% · solve 52%

Distributed weak scaling, measured: 768× the ranks, ∼2× the time; the flat line holds at machine scale.

Driving BoomerAMG (∼10 min + optional)

Hands-on 2

4 AMG holds the line as the problem grows

lab/pcg-amg.yml: the policy block

```yaml
solver:
  pcg:
    relative_tol: 1.0e-8
preconditioner: amg
```

iterations to 10^{-8}

rows	fsai	amg
40^3 = 64k	52	**7**
80^3 = 512k	92	**7**
128^3 = 2.1M	143	**7**

```
$ LAP=./build-polaris-cpu/laplacian
$ for n in 40 80 128; do
  mpirun -n 1 $LAP -ns 1 -n $n $n $n \
    -i lab/pcg-amg.yml -v 1
done
```

```
# after defining LAP at left: 80^3 on 8 ranks
$ mpirun -n 8 $LAP -ns 1 \
  -n 80 80 80 -P 2 2 2 \
  -i lab/pcg-amg.yml -v 1
```

The multigrid promise, on your machine: iterations ≈ constant while N grows 33×.

5 Read the hierarchy, then coarsen aggressively

reuse lab/pcg-amg.yml

```
solver:
  pcg:
    relative_tol: 1.0e-8
preconditioner: amg
```

`-a` overlays only the controlled knob and comes last. One policy defines the entire sweep.

80^3 **Poisson · PCG+AMG**

num_levels	C_{op}	iters	total
0	3.25	7	1210 ms
1	1.37	12	**864 ms**
2	1.22	18	885 ms

`print_level: 1` dumps rows, nnz/row, complexities per level.

```
$ run() { mpirun -n 1 ./build-polaris-cpu/laplacian \
  -n 80 80 80 -ns 1 -i lab/pcg-amg.yml -v 1 "$@"; }
$ for L in 0 1 2; do run -a --preconditioner:amg:print_level 1 \
  --preconditioner:amg:aggressive:num_levels "$L"; done
$ run -a --preconditioner:amg:coarsening:strong_th 0.5
```

Print the hierarchy, watch C_{op}, and optimize *total time*, not iteration count.

6 Smoothers and a nastier operator

(optional, take home if time is short)

```
$ LAP=./build-polaris-cpu/laplacian
$ BASE="mpirun -n 1 $LAP -n 80 80 80 -ns 1 -v 1"
# a) Chebyshev smoothing
$ $BASE -i lab/pcg-amg-cheby.yml
# b) anisotropic diffusion: c_z = 1e-3
$ $BASE -c 1 1 0.001 -i lab/pcg-amg.yml
$ $BASE -c 1 1 0.001 -i lab/pcg-fsai.yml
```

what you should see

run	iters
AMG, default smoother	7
AMG, Chebyshev	6
anisotropic · AMG	**7**
anisotropic · FSAI	96

Observe. Chebyshev matches iterations at higher cost here; its shape (matvecs only) is the GPU story. And on *this* anisotropic operator, strength-of-connection shines: AMG is *unchanged* at 7 while FSAI needs 96.

AMG adapts through strength of connection; that robustness is what setup pays for.

Beyond Poisson: the preconditioner portfolio

Part 3

Match the method to your problem and interface

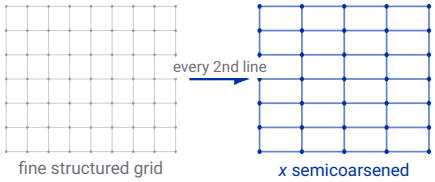
Method	Best match	Defining idea	Interfaces		
			Struct	SStruct	IJ
BoomerAMG	unstructured elliptic	general-purpose algebraic multigrid	×	✓	✓
PFMG	grid + stencil	semicoarsening + point relaxation	✓	✓	×
SMG	grid + hard smoothing	semicoarsening + plane/line relaxation	✓	✓	×
SysPFMG	structured systems	system semicoarsening	×	✓	×
SSAMG	structured parts + seams	preserve structure within each part	×	✓	×
AIR	nonsymmetric transport	local ideal restriction + F-relaxation	×	✓	✓
AMS	$H(\mathrm{curl})$ edge elements	auxiliary nodal spaces	×	✓	✓
ADS	$H(\mathrm{div})$ face elements	curl + gradient auxiliary spaces	×	✓	✓
MGR	coupled multiphysics blocks	physics-informed block reduction	×	×	✓
Schwarz	overlapping local coupling	independent subdomain solves	×	×	✓
ILU / ILUT	general sparse systems	incomplete factorization	×	×	✓
FSAI	SPD + GPU / smoother	approximate inverse; apply = SpMV	×	×	✓

Supported interface means hypre provides the compatible matrix/solver path. SStruct spans native structured methods and selected ParCSR-family methods.

Preserve real structure in the interface; then choose the method that matches the operator's physics.

PFMG: semicoarsening keeps the hierarchy cheap

coarsen one direction at a time



One PFMG V-cycle

1. Point smooth damp oscillatory fine-grid error.

2. Restrict move the residual to retained grid lines.

3. Correct solve on successively coarser grids.

4. Interpolate prolong with the neighboring coarse lines, then post-smooth.

Why the operators stay regular

Two-point interpolation touches the retained lines bracketing a fine point. Coarse stencils remain bounded: at most **9 points in 2-D** and **27 in 3-D**.

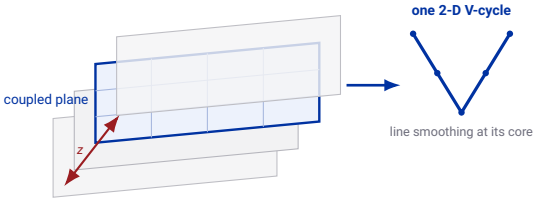
What makes it algebraic

Coefficients guide coarsening and interpolation; geometry enables optional non-Galerkin coarse operators that preserve the stencil. Best fit: scalar diffusion with grid-aligned anisotropy.

PFMG spends known grid geometry to avoid graph discovery and stencil growth.

SMG: plane smoothing buys robustness

in 3-D, relax a whole *xy* plane together



Semicoarsen in *z*; approximate each *xy* plane solve by one 2-D SMG V-cycle.

Why point relaxation can fail

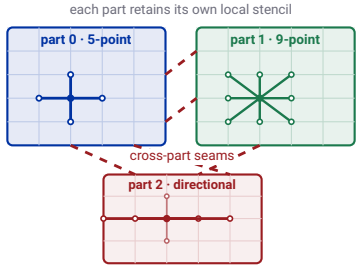
When unknowns are strongly coupled along a plane, the stubborn error is *collective*: changing one point at a time barely reduces it. A plane correction treats that entire coupled block as one relaxation unit.

	PFMG	SMG
hierarchy	semicoarsen	semicoarsen
smoother	points	planes → lines
per-cycle work	lower	higher
robustness	good	stronger

SMG keeps semicoarsening but replaces cheap point smoothing with a nested plane solve.

SSAMG: preserve structure inside parts, model the seams

one operator, two kinds of connectivity



$$A = \underbrace{A_s}_{\text{stencil, within parts}} + \underbrace{A_u}_{\text{graph, across parts}}$$

$$A_c = (P_s^T + P_u^T)\,(A_s + A_u)\,(P_s + P_u)$$

How the hierarchy is built

1. Per part: choose a dominant direction; semicoarsen $2\times$.

2. Transfer: build P_s from A_s; optionally augment with algebraic P_u at seams.

3. Coarsen: RAP carries $A_s + A_u$ at initial levels, then transition to BoomerAMG.

Where it fits and its current scope

Block-structured, overset, and structured-AMR grids. Current: one variable type per part; prefer use as a Krylov preconditioner. `HYPRE_SStructSSAMG*`.

SSAMG pays graph costs at irregular couplings without flattening every structured interior.

Putting it together: pay only for the generality you need

what connectivity can the application declare?

Struct: one grid + stencil

PFMG
point smooth

if it stalls →

SMG
plane smooth

Same semicoarsening family. PFMG: efficient default; SMG spends more to remove plane-coupled error.

SStruct: parts + seams

SSAMG

Structured work in each part; algebraic treatment where the parts couple.

IJ/ParCSR: graph only

BoomerAMG

No exposed grid: discover the hierarchy from the sparse graph.

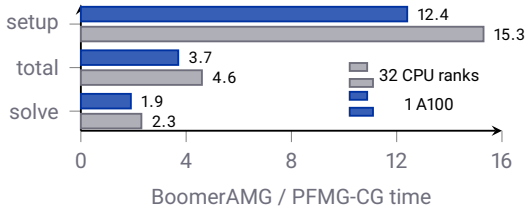
more regular kernels · cheaper setup → more general connectivity

Three decisions: topology chooses the family · coefficients choose the coarsening direction · smoother difficulty separates PFMG from SMG

Declare the most structure that is truly present; then measure whether extra robustness pays.

In general, structure pays most during setup

PFMG-CG speedup over BoomerAMG at $n = 200$



Absolute warm-median time (ms)

	32 CPU ranks		1 A100	
	BoomerAMG	**PFMG-CG**	**BoomerAMG**	**PFMG-CG**
setup	1161.0	75.68	184.0	14.87
solve	816.5	351.51	133.0	69.82
total	1977.5	427.19	317.0	84.69

Setup, solve, and total exclude matrix assembly; 7-point 200^3 Laplace.

Why setup separates

PFMG starts from grid/stencil connectivity and regular BoxLoops. BoomerAMG must discover the strength graph, build *P*, and form *RAP* in ParCSR.

Reproduce: two 32-rank CPU commands

```
# ParCSR / BoomerAMG
mpirun -n 32 ./build-polaris-cpu/laplacian \
  -n 200 200 200 -ns 5 -P 4 4 2 \
  -i lab/pcg-amg.yml -v 1
# Struct / PFMG-CG
for r in {1..5}; do mpirun -n 32 \
  ./build-polaris-cpu/struct -n 50 50 100 \
  -P 4 4 2 -solver 11 -tol 1e-8 -skip 1; done
```

A100: GPU build · 1 rank · $P = 1^3$ · Struct $n = 200^3$ · GPU YAML.

Polaris: setup improves 15.3× on 32 CPU ranks and 12.4× on one A100; the clearest structural win.

The generalists: ILU and FSAI

ILU: incomplete factorization

$M = \tilde{L}\tilde{U} \approx A$, limited fill (`fill_level`, `droptol`); apply = triangular solves; sequential; GPUs can **iterate** them. Generally nonsymmetric ⇒ **pair with GMRES**.

```
preconditioner: { ilu: { fill_level: 1 } }
```

FSAI: factored approximate inverse

Builds $G \approx L^{-1}$, so $B = G^T G \approx A^{-1}$; **SPD by construction**: safe with PCG. Apply = **sparse matvecs** (GPU-friendly; setup support depends on the build).

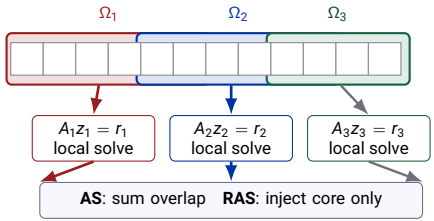
```
preconditioner: { fsai: { max_steps: 5 } }
```

Rule of thumb. With fixed fill/sparsity, iterations grow with N (Lab 1!); use where AMG is overkill, and as **AMG smoothers**.

ILU and FSAI: cheap setup, honest convergence; useful far beyond standalone duty.

Schwarz: overlap local solves to recover global coupling

Standalone ParCSR preconditioner · BoomerAMG smoother · nested MGR
phase: restrict to overlapping subdomains, solve locally, combine corrections.



$$A_i = R_i A R_i^{\mathsf{T}}, \qquad B_{\mathrm{AS}} = \sum_i R_i^{\mathsf{T}} A_i^{-1} R_i$$

RAS replaces the left R_i^{T} with an injection into the nonoverlapping core.

No coarse level: one-level Schwarz alone is not rank-independent (not scalable).

Three choices control the method

Variant	AS/RAS (parallel); hybrid multiplicative (ordered)
Overlap	0/1/2 layers: coupling versus work, memory, communication
Local	ILU(k), ILUT, AMG, or sparse direct

hypredrive: RAS(1)–ILU(0)

```
preconditioner:
  schwarz:
    variant: ras-iluk
    overlap: 1
    iluk_level_of_fill: 0
```

Schwarz trades larger local subdomain solves for fewer global Krylov iterations; overlap is the central knob.

AIR: approximate ideal restriction for nonsymmetric transport

AIR is still AMG, but reduction heuristics build a direction-aware $R \neq P^{\mathsf{T}}$ for nonsymmetric operators.

$$A = \begin{pmatrix} A_{ff} & A_{fc} \\ A_{cf} & A_{cc} \end{pmatrix}, \qquad R_\star = \begin{pmatrix} -A_{cf}A_{ff}^{-1} & I \end{pmatrix}$$

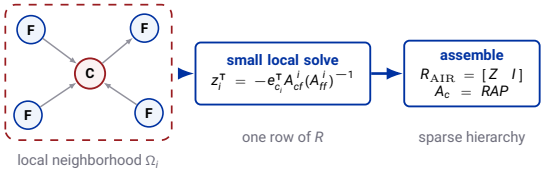
$$P_1 = \begin{pmatrix} W_1 \\ I \end{pmatrix}, \qquad \mathrm{nnz}\big((W_1)_{i,:}\big) = 1, \quad i \in F$$

AIR approximates $R_\star$ locally; one-point P_1 assigns one coarse parent to each F-point.

Where it fits

Target: strongly nonsymmetric, advection-dominated operators, especially upwind and DG discretizations.

Cycle: low-complexity P, AIR restriction, F-relaxation, and an outer GMRES or BiCGSTAB solve.



local neighborhood Ω_i — one row of R — sparse hierarchy

AIR-1 uses one-hop F-neighbors; **AIR-2** also includes F-to-F neighbors.

convdif driver: operator complexity (memory proxy)

3.4 vs. 5.4: ≈ 37% lower hierarchy storage.
Manteuffel et al. report runtime gains for solving DG advection.

AIR's advantage here is lower hierarchy complexity; its strongest applications are directional transport problems.

MGR: multigrid reduction for block systems

MGR principle: coarsen according to physics

Physics defines the split at every level: relax the *F*-fields and retain the *C*-fields.

One-level F/C block form

$$A = \begin{array}{c} \\ F \\ C \end{array} \begin{array}{c} \begin{array}{cc} F & C \end{array} \\ \left(\begin{array}{c|c} A_{FF} & A_{FC} \\ \hline A_{CF} & A_{CC} \end{array}\right) \end{array}$$

F-points relax the variables eliminated on this level.

C-points retain the Schur-complement-like reduced problem.

`f_dofs` + dofmap define the split; recurse until AMG sees the final *C*-problem.

$$E_{\mathrm{MGR}} = \underbrace{(I - P\widehat{S}^{-1}RA)}_{\text{coarse-grid correction}} \underbrace{(I - Q M_F^{-1} Q^{\mathsf{T}} A)}_{F\text{-relaxation}} \underbrace{(I - M_G^{-1} A)}_{G\text{-relaxation}}$$

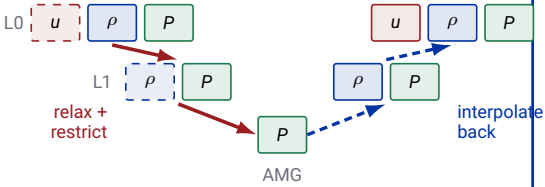
One level combines coarse correction, F-relaxation, and G-relaxation.

Example: poromechanics $u \to \rho \to P$

u: displacement ρ: concentration *P*: pressure

Poromechanics operator

$$A = \begin{pmatrix} A_{uu} & A_{u\rho} & A_{uP} \\ A_{\rho u} & A_{\rho\rho} & A_{\rho P} \\ A_{Pu} & A_{P\rho} & A_{PP} \end{pmatrix}$$



Each level removes one field; AMG receives the terminal *C*-block.

MGR follows the physics: retain selected *C*-variables, relax *F*-variables, and recurse on the reduced block system.

MGR exposes a solver choice at every phase

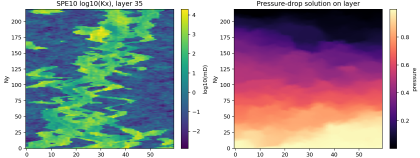
Treat MGR as six decisions; begin with cheap transfers, then strengthen the phase that leaves the dominant error.

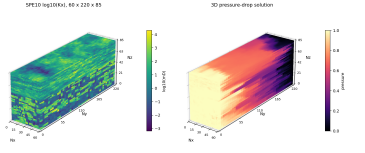
Phase	Start with	Strengthen with	Question answered
G-relaxation	`none` or hybrid GS	block GS; nested AMG/ILU/FSAI/Krylov	Is coupled error left globally?
F-relaxation	`l1-jacobi` or Chebyshev	block/dense solves; nested AMG/MGR/ILU/FSAI	How well is A_{FF}^{-1} applied?
Restriction R	`injection` or Jacobi	approximate inverse; block/CPR/lumping	What residual reaches C?
Interpolation P	`injection` or Jacobi	classical; approximate inverse; block row variants	How is F-error rebuilt?
Coarse operator	Galerkin RAP	non-Galerkin; CPR-like; approximate inverse/ACC	Fidelity or lower fill?
Coarsest solve	AMG	ILU/FSAI/Schwarz; direct; Krylov	What solves the terminal block?

Practical order: strengthen F-relaxation first; improve R/P only when transfer accuracy is limiting; pay for the coarsest solve last. **Exact YAML catalog: Backup E.**

MGR is a composition: spend work in the phase that attacks the error your physics leaves behind.

Solving a porous media flow problem via MGR





SPE10 case 2a: $\log_{10} K_x$ and pressure; 2-D slices (left), 3-D volumes (right)

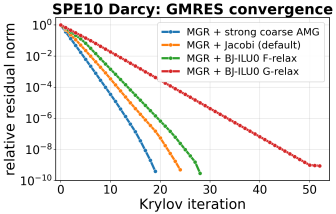
Mixed RT0/P0 Darcy

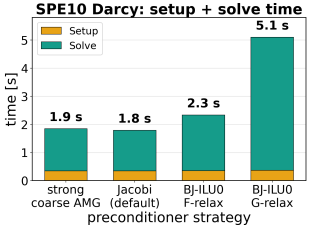
$$K^{-1}\mathbf{u} + \nabla p = 0, \quad \nabla \cdot \mathbf{u} = q$$

$$\begin{pmatrix} M & B^{\mathsf{T}} \\ B & 0 \end{pmatrix} \begin{pmatrix} \mathbf{u} \\ p \end{pmatrix} = \begin{pmatrix} \mathbf{f} \\ \mathbf{g} \end{pmatrix}$$

RT0 face flux
P0 cell

4,525,000 unknowns · 23.5M nnz
3.40M flux (MGR F) + 1.12M pressure (C)
GMRES + two-block **MGR**





Four policies, one YAML each. Only a stronger coarse-pressure AMG cuts iterations; the Jacobi default wins wall time.

Auxiliary spaces: AMS for curl, ADS for div

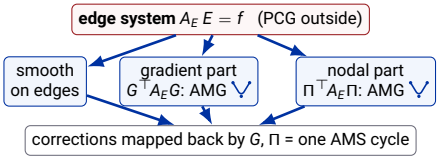
Definite Maxwell, Nédélec edge elements:

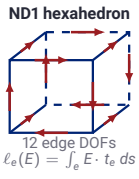
$$\mu^{-1}\nabla\times\nabla\times E+\sigma E=f$$

Curl near-null space = **all gradients** → scalar AMG cannot infer the smooth-error space from A alone.

AMS · $H(\mathrm{curl})$: edge unknowns; supply discrete gradient G + coordinates.

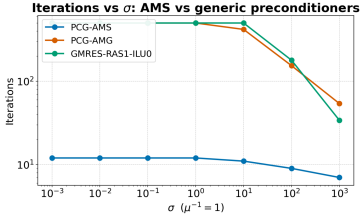
ADS · $H(\mathrm{div})$: face unknowns; supply discrete curl C, gradient G + coordinates.





curl-curl 32^3: 95k edge DOFs, PCG

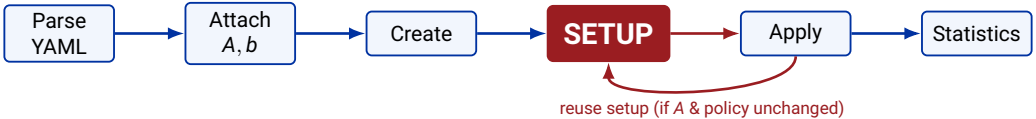
preconditioner	iters	solve
BoomerAMG	316	3.80 s
AMS	**9**	**0.56 s**



σ-sweep (*hypredrive* docs): AMS flat, AMG/ILU in the hundreds; wrong *tool*, not wrong tuning; run both in Lab 3 (Ex 9)

AMS/ADS consume your discretization's de Rham operators and restore multigrid performance.

From preconditioner choice to setup reuse



embedded solve lifecycle (C API)

```
HYPREDRV_LinearSystemSetMatrix(h, A);
HYPREDRV_LinearSystemSetRHS(h, b);
HYPREDRV_LinearSolverCreate(h);
HYPREDRV_LinearSolverSetup(h);  // costly
HYPREDRV_LinearSolverApply(h);  // cheap
```

reuse is a policy: `preconditioner.reuse`

```
preconditioner:
  amg: {}
  reuse:
    enabled: yes
    frequency: 2  # rebuild every 3rd
```

Exact reuse: same *A*, new *b*'s; always valid. **Lagged setup**: *A* drifts (time stepping, Newton); keep the old setup until a *rebuild rule* fires.

...or `reuse: adaptive`; rebuild only when recent convergence degrades.

Reuse is a *decision*, not a default; enable it, schedule it, or let `adaptive` decide.

Block systems, Maxwell, and GPUs ($\sim$7 min + optional)

Hands-on 3

7 A real block system: MGR on GEOS flow

GEOS Jacobian: nonsymmetric coupled blocks

$$J\,\delta x = -R, \qquad \delta x = \begin{pmatrix}\delta p & \delta\rho_1 & \delta\rho_2\end{pmatrix}^T$$

$$J = \begin{pmatrix} J_{pp} & J_{p1} & J_{p2} \\ J_{1p} & J_{11} & J_{12} \\ J_{2p} & J_{21} & J_{22} \end{pmatrix}$$

Fields: label 0 = pressure; labels 1–2 = global component densities.
Recorded matrix: 1,875 cells → 5,625 rows; 77,475 nonzeros.

- **0** eliminate ρ_2; retain $\{p, \rho_1\}$
- **1** eliminate ρ_1; retain $\{p\}$
- **coarse** AMG on the pressure reduced system

GEOS: github.com/GEOS-DEV/GEOS

examples/ex3.yml (excerpt): MGR-GMRES

```
preconditioner:
  mgr:
    level:
      0: {f_dofs: [2]} # eliminate rho_2
      1: {f_dofs: [1]} # eliminate rho_1
    coarsest_level: amg
```

```
$ CLI=./build-polaris-cpu/hypredrive-cli
# ex3: 1 rank, 8 iters; ex4: 4 ranks
# compare iterations, residual, setup + solve
$ mpirun -n 1 $CLI examples/ex3.yml
$ mpirun -n 4 $CLI examples/ex4.yml
```

MGR uses the dofmap to reduce densities first, exposing the pressure problem that AMG solves well.

8 Modular configs & a sequence of solves

(optional, take home if time is short)

examples/ex5.yml: compose with include

```yaml
solver:
  include: ex5-gmres.yml
preconditioner:
  include: ex5-mgr.yml
```

```
$ CLI=./build-polaris-cpu/hypredrive-cli
$ mpirun -n 1 $CLI examples/ex5.yml
# include: reuse shared solver / precond files

$ mpirun -n 1 $CLI examples/ex7.yml
# a SEQUENCE of systems (poromech2k);
# MGR-FGMRES; one stats row per system
```

Try it. In `ex7`, each row is one system; watch iterations drift as the matrix evolves. Then `examples/ex7-mgr-frelax-reuse.yml`: explicit reuse of the *nested F-relaxation AMG hierarchy* (the rest of MGR is rebuilt); compare the setup column; these systems are tiny, so the knob matters at scale.

Compose configurations with `include`; reuse across a sequence is explicit and component-wise.

9 Match both: AMS for Maxwell, AMG on one A100

```
# a) edge-element curl-curl: AMG vs AMS
$ MAX="mpirun -n 1 ./build-polaris-cpu/maxwell -ns 1"
$ $MAX -n 32 32 32 -i lab/amg-host.yml -v 1
$ $MAX -n 32 32 32 -i lab/ams-host.yml -v 1
# b) GPU-ready Laplacian (CUDA 12.9 build, one A100)
$ GPU=./build-polaris-gpu/laplacian
$ CUDA_VISIBLE_DEVICES=0 mpirun -n 1 $GPU \
    -ns 5 -n 128 128 128 -i lab/pcg-amg-gpu.yml -v 1
```

what you should see

run	iters	result
Maxwell · AMG	316	grinds
Maxwell · AMS	**9**	converges
128^3 · 32 CPU ranks	10	0.45 s
128^3 · 1 A100	14	**0.10 s**

CPU/A100 time = median setup+solve from warm rows 1–4.

Observe. These are two distinct choices: metadata selects AMS for the Maxwell operator; `exec_policy: device` retargets AMG for a GPU-ready Laplacian. Your application's assembly and data movement remain separate concerns.

Choose the algorithm for the operator, then the executable and policy for the hardware.

Stretch: reproduce the SPE10 result from Part 3

```
# shared tree: darcy is prebuilt; fetch SPE10 on a login node if absent
$ test -s data/spe10_case2a/spe_perm.dat || \
    scripts/download_spe10_case2a.sh data/spe10_case2a
# inside the Polaris compute allocation:
$ mpirun -n 16 ./build-polaris-cpu/darcy -n 60 220 85 -P 1 4 4 \
    --K-file data/spe10_case2a/spe_perm.dat --K-file-grid 60 220 85 \
    --K-file-k-order top-down -g y -i examples/src/C_darcy/mgr_amg_strong.yml
```

Try it. Four MGR policies on one **4.5M-unknown** mixed RT0/P0 Darcy operator; the Part-3 comparison. Only strengthening the coarse-pressure `coarsest_level` AMG cuts iterations: edit `mgr_jacobi.yml` and watch the GMRES count drop.

From a 1000-row Poisson to a 4.5M-unknown reservoir; the same configuration-driven workflow.

Concluding remarks

1 **Scalability lives in the preconditioner**; Krylov handles the iteration; multigrid makes it near-$O(N)$, and the flat weak-scaling line is measured, not promised.

2 **Data structures flex to the problem**; declare what you have (Struct / SStruct / IJ): structured layouts keep setup cheap and GPU-fast; ParCSR covers the fully general case.

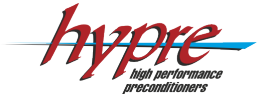
3 **Match the method to the operator**; BoomerAMG for elliptic; PFMG/SMG/SSAMG with structure; AMS/ADS for curl/div; MGR for blocks; Schwarz/ILU/FSAI as generalists. The app supplies the metadata.

4 **Make solver strategy data, not code**; YAML policies made every experiment a one-line edit; reuse and host↔device are decisions in the same file.

5 **Bring your own problem**; `IJ` files from application code (`hypredrive-cli`), `libHYPRE` or `libHYPREDRV` from C/C++/Fortran/Python/MATLAB/Octave/Julia.

Scalability lives in the preconditioner; pick it for your operator; *hypredrive* allows fast prototyping.

Acknowledgments & contact



Get in touch

Repo: github.com/hypre-space/hypre
github.com/hypre-space/hypredrive

Issues: github.com/hypre-space/hypre/issues
github.com/hypre-space/hypredrive/issues

Docs: hypre.readthedocs.io
hypredrive.readthedocs.io

Examples: at-a-glance gallery (*hypredrive* docs)

Presenter: Victor A. P. Magri
paludettomag1@llnl.gov

Funding

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC. LLNL-PRES-2021722.

This work was supported by the U.S. Department of Energy Office of Science, Office of Advanced Scientific Computing Research (ASCR), Scientific Discovery through Advanced Computing (SciDAC) program, the FASTMath Institute, and by the Exascale Computing Project, a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

Bring your own linear system or keep exploring on Polaris

Extra exercises

Extra exercise 1: SuiteSparse, ILU vs AMG vs FSAI

Janna/StocF-1465

Porous-media flow on an unstructured tetrahedral grid; real, symmetric, SPD.

1,465,137 rows · **21,005,389** nonzeros

`randsol`: deterministic x_{ref}, then $b = Ax_{\text{ref}}$

```
$ scripts/fetch_suitesparse_matrix.sh \
    Janna/StocF-1465
$ CLI=./build-polaris-cpu/hypredrive-cli
$ for p in ilu amg fsai; do
  mpirun -n 1 $CLI lab/extra/$p.yml -q
done
```

Recorded: defaults, one CPU rank

precond.	solver	iters	setup	solve	S+S
				seconds	
ILU	GMRES(50)	69	**0.43**	5.59	6.03
AMG	PCG	**10**	2.23	**2.09**	**4.32**
FSAI	PCG	328	9.32	27.26	36.58

Rule of thumb. The matrix is SPD, so AMG/FSAI use PCG. Generic ILU is not guaranteed SPD, so it uses GMRES. All three reached 10^{-8} relative residual.

Fetch a real operator, use a valid Krylov pairing, then compare setup + solve; defaults are only a baseline.

Extra exercise 2: The full Krylov + AMG lesson

Three guided tracks

Krylov restart, per-iteration cost, invalid CG

AMG 1/8/32-rank growth, complexity, aggressive coarsening

Hardware PFMG, anisotropy, 32 Milan ranks vs one A100

Rule of thumb. Stress serial tuning at 32 ranks · compare 7/27-point strength graphs · test grid-aligned anisotropy · benchmark warm phases.

```
# included beside lab/ in the attendee bundle
$ less lesson/lesson.md

# login node: network-facing configure only
$ bash scripts/build.sh --config
# allocation: compile, then follow one guided track
$ bash scripts/build.sh
$ less lesson/expected/README.md
```

Try it. Start with one track. Run its numbered commands from `lesson.md`; compare with the reviewed expected outcomes.

Suggested time: Krylov+AMG core 45–60 min · hardware/structure 30–45 min · stress tests optional.

Go deeper on the same ideas; scale them, stress them, and measure them reproducibly on Polaris.

Technical backup slides

Backup

Backup A: Build and installation

```
$ git clone https://github.com/hypre-space/hypredrive
$ cd hypredrive
$ cmake -DHYPREDRV_ENABLE_DATA=ON \
        -DHYPREDRV_ENABLE_EXAMPLES=ON \
        -DHYPREDRV_BUILD_DSUPERLU=ON \
        -DHYPRE_BUILD_TESTS=ON \
        -B build
$ cmake --build build -j -t all data ij struct sstruct
$ cmake --install build --prefix install-local
```

Notes

hypre master is fetched automatically if not found.

Reuse an install: `-DHYPRE_ROOT=<path>`.

`-t check` runs the built-in smoke test.

GPU builds: reconfigure that build with one backend; architecture is auto-detected:

NVIDIA GPUs

```
$ cmake \
  -DHYPREDRV_ENABLE_CUDA=ON \
  -DHYPRE_BUILD_UMPIRE=ON -B build
$ cmake --build build -j
```

AMD GPUs

```
$ cmake \
  -DHYPREDRV_ENABLE_HIP=ON \
  -DHYPRE_BUILD_UMPIRE=ON -B build
$ cmake --build build -j
```

Intel GPUs

```
$ cmake \
  -DHYPREDRV_ENABLE_SYCL=ON \
  -DHYPRE_BUILD_UMPIRE=ON \
  -DCMAKE_CXX_COMPILER=icpx -B build
$ cmake --build build -j
```

Backup B: The configuration is the numerical experiment

baseline

```
solver: pcg
preconditioner: amg
```

controlled variant

```
solver:
  gmres:
    max_iter: 100
    relative_tol: 1.0e-8
preconditioner: ilu
```

command-line override

```
BIN=./build-polaris-cpu
$BIN/hypredrive-cli ex.yml -a
--solver:gmres:max_iter 200

# sweep without editing files
```

versionable | **sweepable** | **shareable**

Strategy changes through configuration, not recompilation; versioned, swept, and shared.

Backup C: One edit, a different solver strategy

A · baseline

```
solver: pcg
preconditioner: amg
```

```
STATISTICS SUMMARY
setup [ms]   2.125
solve [ms]   0.522
iters            6
rel. res. 4.98e-08
```

B · swap precond

```
solver: pcg
preconditioner: fsai
```

```
STATISTICS SUMMARY
setup [ms]   4.358
solve [ms]   0.390
iters           11
rel. res. 2.91e-07
```

C · swap solver

```
solver: gmres
preconditioner: amg
```

```
STATISTICS SUMMARY
setup [ms]   2.100
solve [ms]   0.594
iters            6
rel. res. 4.84e-08
```

`same matrix · same RHS · same build · same hardware`

archived recording: hypredrive `b59401c` · hypre `2fe7960` · 1 rank · Polaris Milan · `ex1.yml` (1000 rows) · `lab/out/C-ex1*` · workflow demo

The application and matrix never change; one YAML line swaps the preconditioner (B) or the solver (C).

Backup D: GEOS mechanics on Frontier

Measured weak scaling from one full node

20.7M → 81.3B DOFs | **8 → 32,768 GPUs** | **1 → 4,096 nodes**

Average time per Newton iteration		
	8 GPUs	**32,768 GPUs**
matrix creation	0.4 s	0.7 s
hypre setup	0.9 s	2.2 s
hypre solve	0.7 s	3.1 s
GEOS work	1.8 s	2.4 s
total	**3.8 s**	**8.3 s**

Data: Settgast et al., JOSS 9(102), 6973 (2024), Fig. 2(a), https://doi.org/10.21105/joss.06973.

Full-node to machine scale

8 GPUs = 1 node

4 MI250X packages × 2 GCDs/package

32,768 GPUs

4,096 nodes; ∼43% of Frontier

81.3B DOFs

∼3,900× more DOFs at ∼2.2× the time per Newton iteration.

Backup E: Complete MGR option catalog

Exact *hypredrive* names corresponding to the readable phase menu in the main talk.

Phase	YAML key	Available choices	Design role
G-relaxation	`g_relaxation`	**block:** `blk-jacobi`, `blk-gs`, `mixed-gs` **hybrid/GS:** `h-fgs`, `h-bgs`, `ch-gs`, `h-ssor`, `2stg-fgs`, `2stg-bgs`, `l1-hfgs`, `l1-hbgs`, `l1-hsgs` **solver:** `amg`, `ilu`, `fsai`, `euclid`, `schwarz`, `spdirect` **Krylov:** `pcg`, `gmres`, `fgmres`, `bicgstab`; or `none`	Globally damp coupled error before reduction.
F-relaxation	`f_relaxation`	**point:** `jacobi`, `single`, `l1-jacobi`, `chebyshev` **block/dense:** `v(1,0)`, `ge`, `ge-piv`, `ge-inv` **solver:** `amg`, `mgr`, `ilu`, `fsai`, `schwarz`, `spdirect` **Krylov:** `pcg`, `gmres`, `fgmres`, `bicgstab`; or `none`	Approximate A_{FF}^{-1}; often the strongest physics lever.
Restriction R	`restriction_type`	`injection`, `jacobi`, `approx-inv`, `blk-jacobi`, `cpr-like`, `columped`	Transfer residual to the retained C-space.
Interpolation P	`prolongation_type`	`injection`, `l1-jacobi`, `jacobi`, `classical-mod`, `approx-inv`, `blk-jacobi`, `blk-rowlump`, `blk-rowsum`, `blk-absrowsum`	Reconstruct eliminated error; accuracy versus fill.
Coarse operator	`coarse_level_type`	`rap`, `galerkin`, `non-galerkin`, `cpr-like-diag`, `cpr-like-bdiag`, `approx-inv`, `acc`	Choose fidelity and sparsity of $\widehat{S}$.
Coarsest solve	`coarsest_level`	`def`, `amg`, `ilu`, `fsai`, `schwarz`, `spdirect`, `pcg`, `gmres`, `fgmres`, `bicgstab`	Solve the terminal reduced problem.

Orthogonal controls: `cycle`: `v`, `w`, `v(1,0)`, `v(0,1)`, `v(1,1)`, `w(1,0)`, `w(0,1)`, `w(1,1)`, `1, 2` · `num_sweeps` · `reuse`. `schwarz`/`spdirect`: *build-dependent*.

Use this as a reference catalog; tune a phase for a diagnosed error, not as a Cartesian-product search.