The Portable Extensible Toolkit for Scientific computing

Day 2: Performance, scalability, and hard problems

Jed Brown

CSCS 2010-05-11

Outline

1. Algorithms (continued from yesterday)
   - Hydrostatic Ice
   - Driven cavity
2. Application Integration
3. Performance and Scalability
   - Memory hierarchy
   - Profiling
4. Hard problems
   - Splitting

Requests

- Tell me if you do not understand
- Tell me if an example does not work
- Suggest better wording, figures, organization
- Follow up:
  - Configuration issues, private: `petsc-maint@mcs.anl.gov`
  - Public questions: `petsc-users@mcs.anl.gov`

Hydrostatic equations for ice sheet flow

- Valid in the limit $w_x \ll u_z$, independent of basal friction
- Eliminate p and w by incompressibility:

  3D elliptic system for $u = (u, v)$

$$-\nabla \cdot \left[\eta \begin{pmatrix} 4u_x + 2v_y & u_y + v_x & u_z \\ u_y + v_x & 2u_x + 4v_y & v_z \end{pmatrix} \right] + \rho g \nabla s = 0$$

$$\eta(\gamma) = \frac{B}{2}(\varepsilon^2 + \gamma)^{\frac{1-\mathfrak{n}}{2\mathfrak{n}}}, \qquad \mathfrak{n} \approx 3$$

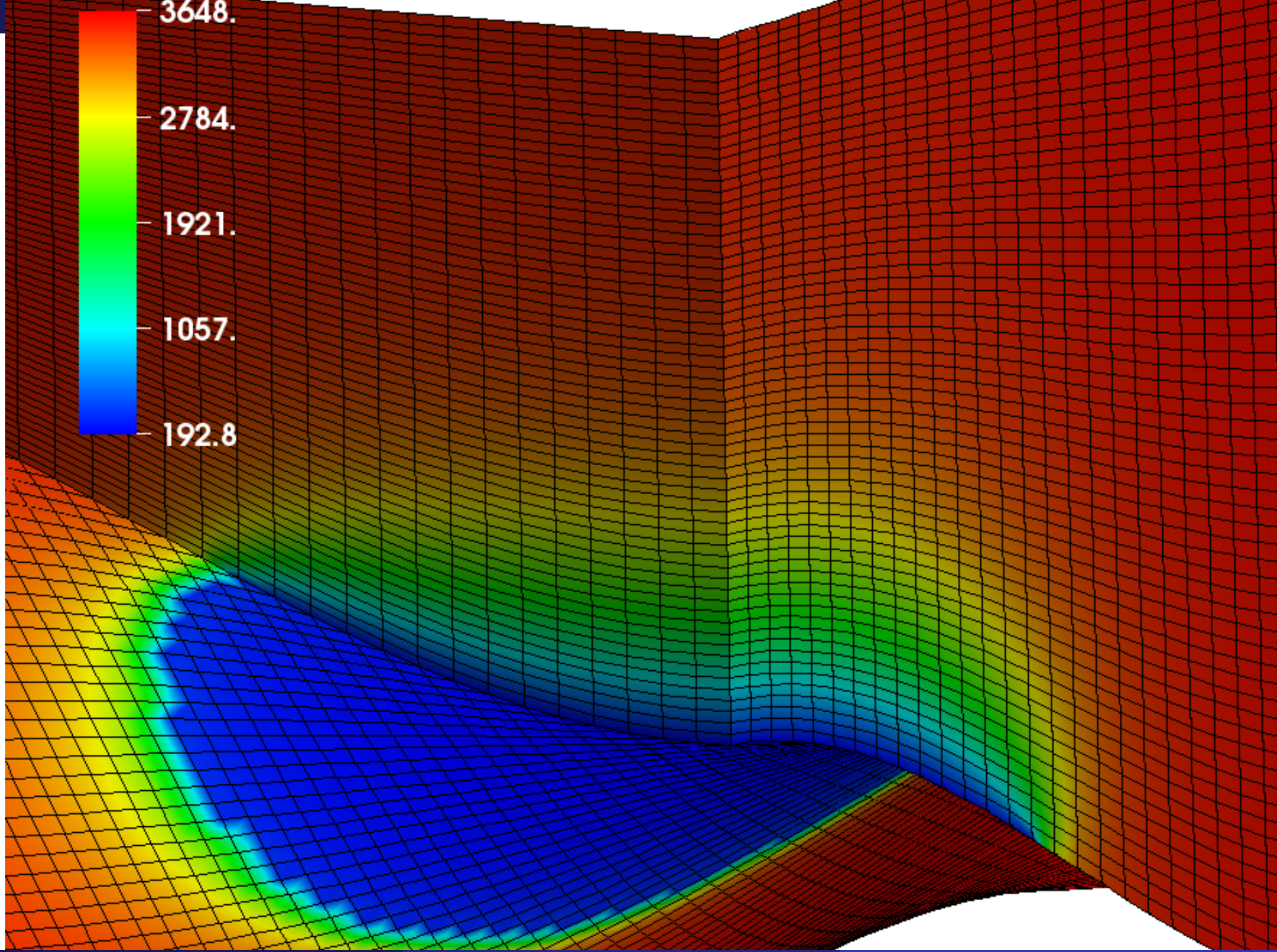
$$\gamma = u_x^2 + v_y^2 + u_x v_y + \frac{1}{4}(u_y + v_x)^2 + \frac{1}{4}u_z^2 + \frac{1}{4}v_z^2$$

  and slip boundary $\sigma \cdot n = \beta^2 u$ where

$$\beta^2(\gamma_b) = \beta_0^2(\varepsilon_b^2 + \gamma_b)^{\frac{\mathfrak{m}-1}{2}}, \qquad 0 < \mathfrak{m} \leq 1$$

$$\gamma_b = \frac{1}{2}(u^2 + v^2)$$

- Q_1 FEM: `src/snes/examples/tutorials/ex48.c`


3648.
2784.
1921.
1057.
192.8

Some Multigrid Options

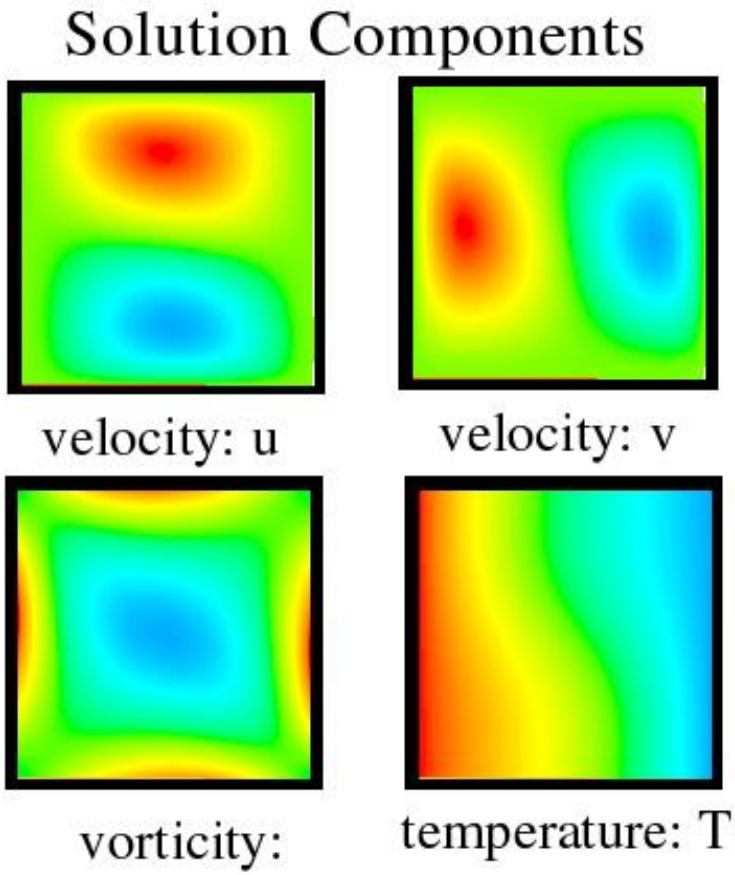
- `-dmmg_grid_sequencce`: [FALSE]
  Solve nonlinear problems on coarse grids to get initial guess
- `-pc_mg_galerkin`: [FALSE]
  Use Galerkin process to compute coarser operators
- `-pc_mg_type`: [FULL]
  (choose one of) MULTIPLICATIVE ADDITIVE FULL KASKADE
- `-mg_coarse_{ksp,pc}_*`
  control the coarse-level solver
- `-mg_levels_{ksp,pc}_*`
  control the smoothers on levels
- `-mg_levels_3_{ksp,pc}_*`
  control the smoother on specific level
- These also work with ML's algebraic multigrid.

What is this doing?

- ```
  mpiexec -n 4 ./ex48 -M 16 -P 2 -da_refine_hierarchy_x 1,8,8
  -da_refine_hierarchy_y 2,1,1 -da_refine_hierarchy_z 2,1,1
  -dmmg_grid_sequence 1 -dmmg_view -log_summary
  -ksp_converged_reason -ksp_gmres_modifiedgramschmidt
  -ksp_monitor -ksp_rtol 1e-2
  -pc_mg_type multiplicative
  -mg_coarse_pc_type lu -mg_levels_0_pc_type lu
  -mg_coarse_pc_factor_mat_solver_package mumps
  -mg_levels_0_pc_factor_mat_solver_package mumps
  -mg_levels_1_sub_pc_type cholesky
  -snes_converged_reason -snes_monitor -snes_stol 1e-12
  -thi_L 80e3 -thi_alpha 0.05 -thi_friction_m 0.3
  -thi_hom x -thi_nlevels 4
  ```
- What happens if you remove `-dmmg_grid_sequence`?
- What about solving with block Jacobi, ASM, or algebraic multigrid?
  ```

SNES Example

Driven Cavity



- Velocity-vorticity formulation
- Flow driven by lid and/or bouyancy
- Logically regular grid
  - Parallelized with `DA`
- Finite difference discretization
- Authored by David Keyes

SNES Example

Driven Cavity Application Context

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

typedef struct {
      /* physical parameters */
   PassiveReal lidvelocity,prandtl,grashof;
      /* color plots of the solution */
   PetscTruth  draw_contours;
} AppCtx;
```

SNES Example

Driven Cavity Residual Evaluation

```
DrivenCavityFunction(SNES snes, Vec X, Vec F, void *ptr) {
  AppCtx         *user = (AppCtx *) ptr;
  /* local starting and ending grid points */
  PetscInt       istart, iend, jstart, jend;
  PetscScalar    *f;                 /* local vector data */
  PetscReal      grashof = user->grashof;
  PetscReal      prandtl = user->prandtl;
  PetscErrorCode ierr;

  /* Code to communicate nonlocal ghost point data */
  VecGetArray(F, &f);
  /* Code to compute local function components */
  VecRestoreArray(F, &f);
  return 0;
}
```

SNES Example

Better Driven Cavity Residual Evaluation

```
PetscErrorCode DrivenCavityFuncLocal(DALocalInfo *info,
                    Field **x,Field **f,void *ctx) {
  /* Handle boundaries */
  /* Compute over the interior points */
  for(j = info->ys; j < info->ys+info->ym; j++) {
    for(i = info->xs; i < info->xs+info->xm; i++) {
      /* convective coefficients for upwinding */
      /* U velocity */
      u          = x[j][i].u;
      uxx        = (2.0*u - x[j][i-1].u - x[j][i+1].u)*hydhx;
      uyy        = (2.0*u - x[j-1][i].u - x[j+1][i].u)*hxdhy;
      upw        = 0.5*(x[j+1][i].omega-x[j-1][i].omega)*hx
      f[j][i].u  = uxx + uyy - upw;
      /* V velocity, Omega, Temperature */
}}}
```

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

Running the driven cavity

- `./ex19 -lidvelocity 100 -grashof 1e2 -da_grid_x 16 -da_grid_y 16 -snes_monitor -dmmg_view -nlevels 3`
- `./ex19 -lidvelocity 100 -grashof 1e4 -da_grid_x 16 -da_grid_y 16 -snes_monitor -dmmg_view -nlevels 3`
- `./ex19 -lidvelocity 100 -grashof 1e5 -da_grid_x 16 -da_grid_y 16 -snes_monitor -dmmg_view -nlevels 3`
- Uh oh, we have convergence problems
- Run with `-snes_monitor_convergence`
- Does `-dmmg_grid_sequence` help?

Running the driven cavity

- `./ex19 -lidvelocity 100 -grashof 1e2 -da_grid_x 16 -da_grid_y 16 -snes_monitor -dmmg_view -nlevels 3`
- `./ex19 -lidvelocity 100 -grashof 1e4 -da_grid_x 16 -da_grid_y 16 -snes_monitor -dmmg_view -nlevels 3`
- `./ex19 -lidvelocity 100 -grashof 1e5 -da_grid_x 16 -da_grid_y 16 -snes_monitor -dmmg_view -nlevels 3`
- Uh oh, we have convergence problems
- Run with `-snes_monitor_convergence`
- Does `-dmmg_grid_sequence` help?

Why isn't SNES converging?

- The Jacobian is wrong (maybe only in parallel)
  - Check with `-snes_type test` and `-snes_mf_operator -pc_type lu`
- The linear system is not solved accurately enough
  - Check with `-pc_type lu`
  - Check `-ksp_monitor_true_residual`, try right preconditioning
- The Jacobian is singular with inconsistent right side
  - Use `MatNullSpace` to inform the `KSP` of a known null space
  - Use a different Krylov method or preconditioner
- The nonlinearity is just really strong
  - Run with `-info` or `-snes_ls_monitor` (petsc-dev) to see line search
  - Try using trust region instead of line search `-snes_type tr`
  - Try grid sequencing if possible
  - Use a continuation

Globalizing the lid-driven cavity

Pseudotransient continuation continuation (Ψtc)

- Do linearly implicit backward-Euler steps, driven by steady-state residual
- Clever way to adjust step sizes to retain quadratic convergence it terminal phase

- Implemented in `src/snes/examples/tutorials/ex27.c`
- `$ make runex27`
- Make the method linearly implicit: `-snes_max_it 1`
  - Compare required number of linear iterations
- Try increasing `-lidvelocity`, `-grashof`, and problem size
- Coffey, Kelley, and Keyes, *Pseudotransient continuation and differential algebraic equations*, SIAM J. Sci. Comp, 2003.

Application Integration

- Be willing to experiment with algorithms
  - No optimality without interplay between physics and algorithmics
- Adopt flexible, extensible programming
  - Algorithms and data structures not hardwired
- Be willing to play with the real code
  - Toy models are rarely helpful
- If possible, profile before integration
  - Automatic in PETSc

Incorporating PETSc into existing codes

- PETSc does not seize `main()`, does not control output
- Propogates errors from underlying packages, flexible error handling
- Nothing special about `MPI_COMM_WORLD`
- Can wrap existing data structures/algorithms
  - `MatShell`, `PCShell`, full implementations
  - `VecCreateMPIWithArray()`
  - `MatCreateSeqAIJWithArrays()`
  - Use an existing semi-implicit solver as a preconditioner
  - Usually worthwhile to use native PETSc data structures unless you have a good reason not to
- Uniform interfaces across languages: C, C++, Fortran 77/90, Python
- Do not have to use high level interfaces
  - but PETSc can offer more if you do, like MFFD and SNES Test

Integration Stages

- Version Control
  - It is impossible to overemphasize
- Initialization
  - Linking to PETSc
- Profiling
  - Profile before changing
  - Also incorporate command line processing
- Linear Algebra
  - First PETSc data structures
- Solvers
  - Very easy after linear algebra is integrated

Initialization

- Call `PetscInitialize()`
  - Setup static data and services
  - Setup MPI if it is not already
  - Can set `PETSC_COMM_WORLD` to use your communicator (can always use subcommunicators for each object)
- Call `PetscFinalize()`
  - Calculates logging summary
  - Can check for leaks/unused options
  - Shutdown and release resources
- Can only initialize PETSc once

Matrix Memory Preallocation

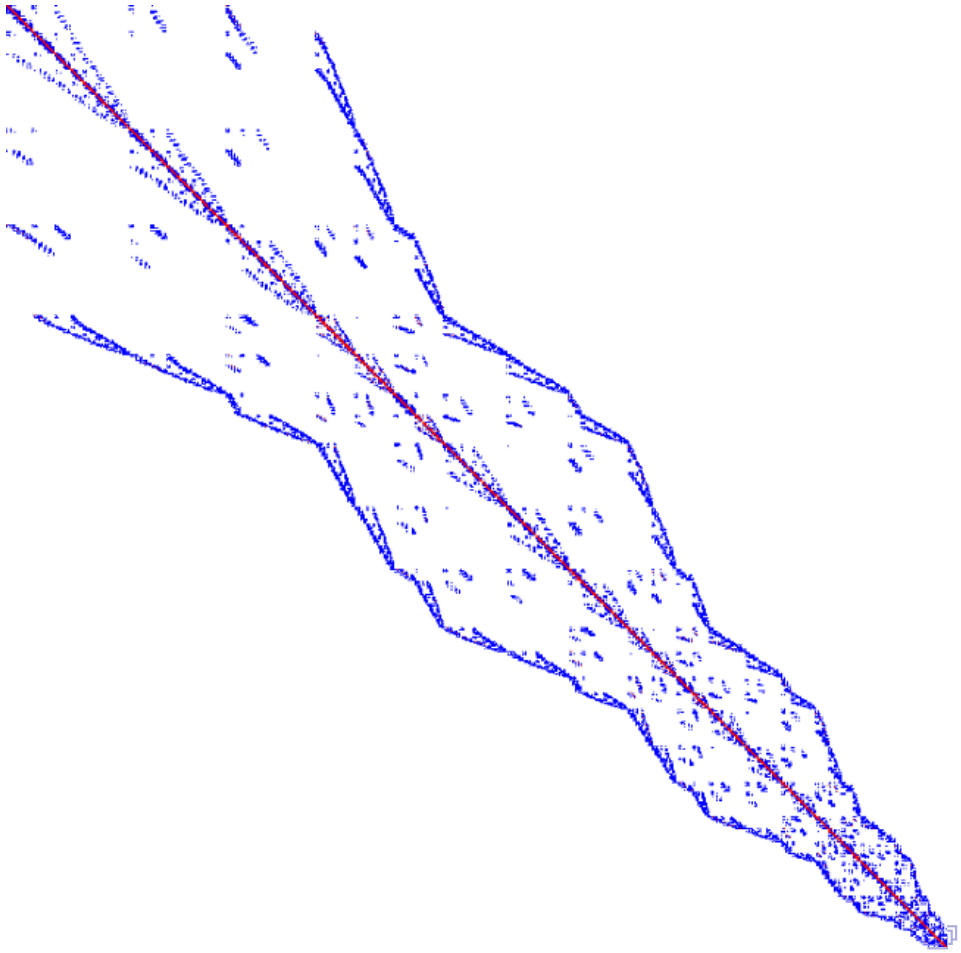
- PETSc sparse matrices are dynamic data structures
  - can add additional nonzeros freely
- Dynamically adding many nonzeros
  - requires additional memory allocations
  - requires copies
  - can kill performance
- Memory preallocation provides
  - the freedom of dynamic data structures
  - good performance
- Easiest solution is to replicate the assembly code
  - Remove computation, but preserve the indexing code
  - Store set of columns for each row
- Call preallocation routines for all datatypes
  - `MatSeqAIJSetPreallocation()`
  - `MatMPIBAIJSetPreallocation()`
  - Only the relevant data will be used

Sequential Sparse Matrices

`MatSeqAIJPreallocation(Mat A, int nz, int nnz[])`

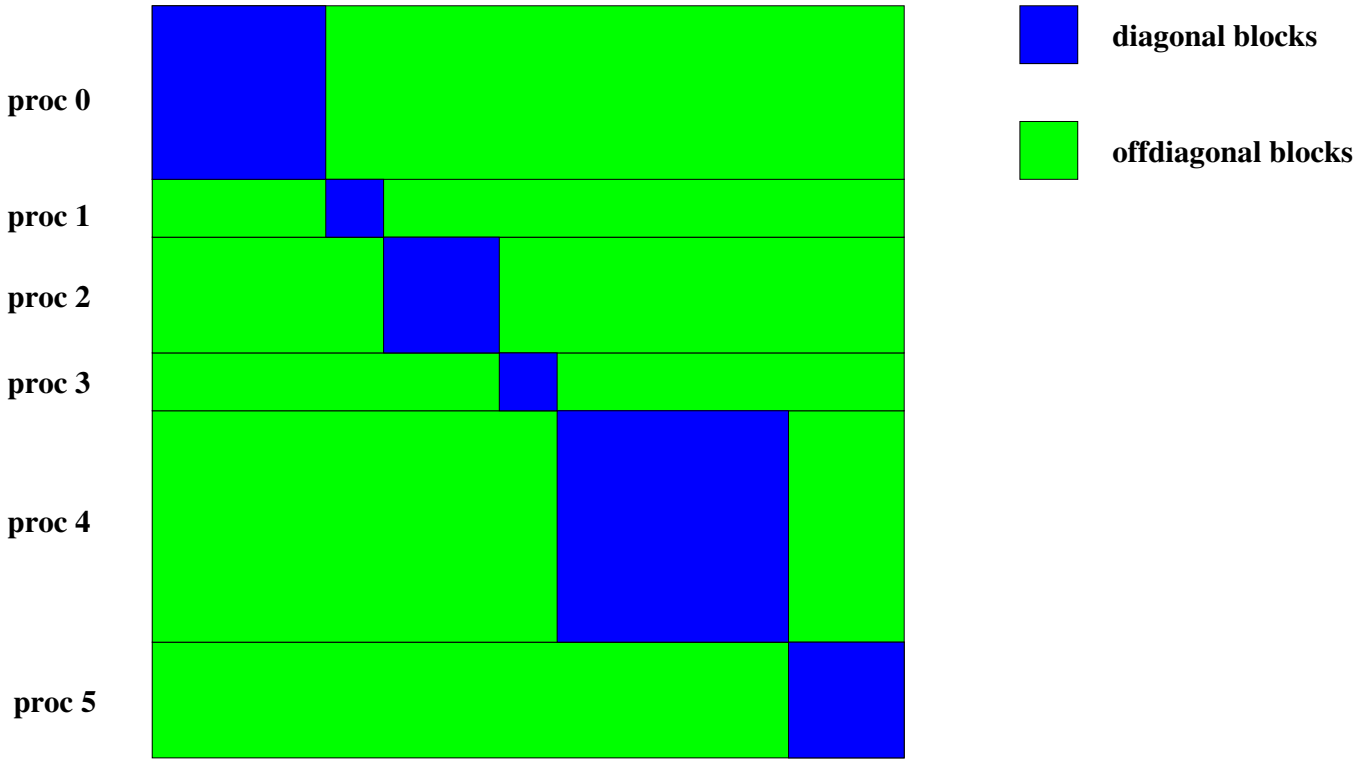
nz: expected number of nonzeros in any row

nnz(i): expected number of nonzeros in row i



Parallel Sparse Matrix

- Each process locally owns a submatrix of contiguous global rows
- Each submatrix consists of diagonal and off-diagonal parts



- `MatGetOwnershipRange(Mat A,int *start,int *end)`

`start`: first locally owned row of global matrix

`end-1`: last locally owned row of global matrix

Parallel Sparse Matrices

```
MatMPIAIJPreallocation(Mat A, int dnz, int dnnz[],
        int onz, int onnz[])
```

dnz: expected number of nonzeros in any row in the diagonal block

dnnz(i): expected number of nonzeros in row i in the diagonal block

onz: expected number of nonzeros in any row in the offdiagonal portion

onnz(i): expected number of nonzeros in row i in the offdiagonal portion

Verifying Preallocation

- Use runtime option `-info`
- Output:

  `[proc #] Matrix size: %d X %d; storage space: %d unneeded, %d used`

  `[proc #] Number of mallocs during MatSetValues( ) is %d`

```
[merlin] mpirun ex2 -log_info
[0]MatAssemblyEnd_SeqAIJ:Matrix size: 56 X 56; storage space:
[0]   310 unneeded, 250 used
[0]MatAssemblyEnd_SeqAIJ:Number of mallocs during MatSetValues() is 0
[0]MatAssemblyEnd_SeqAIJ:Most nonzeros in any row is 5
[0]Mat_AIJ_CheckInode: Found 56 nodes out of 56 rows. Not using Inode routine
[0]Mat_AIJ_CheckInode: Found 56 nodes out of 56 rows. Not using Inode routine
Norm of error 0.000156044 iterations 6
[0]PetscFinalize:PETSc successfully ended!
```

Block and symmetric formats

- BAIJ
  - Like AIJ, but uses static block size
  - Preallocation is like AIJ, but just one index per block
- SBAIJ
  - Only stores upper triangular part
  - Preallocation needs number of nonzeros in upper triangular parts of on- and off-diagonal blocks
- `MatSetValuesBlocked()`
  - Better performance with blocked formats
  - Also works with scalar formats, if `MatSetBlockSize()` was called
  - Variants `MatSetValuesBlockedLocal()`, `MatSetValuesBlockedStencil()`
  - Change matrix format at runtime, don't need to touch assembly code

Linear Solvers

Krylov Methods

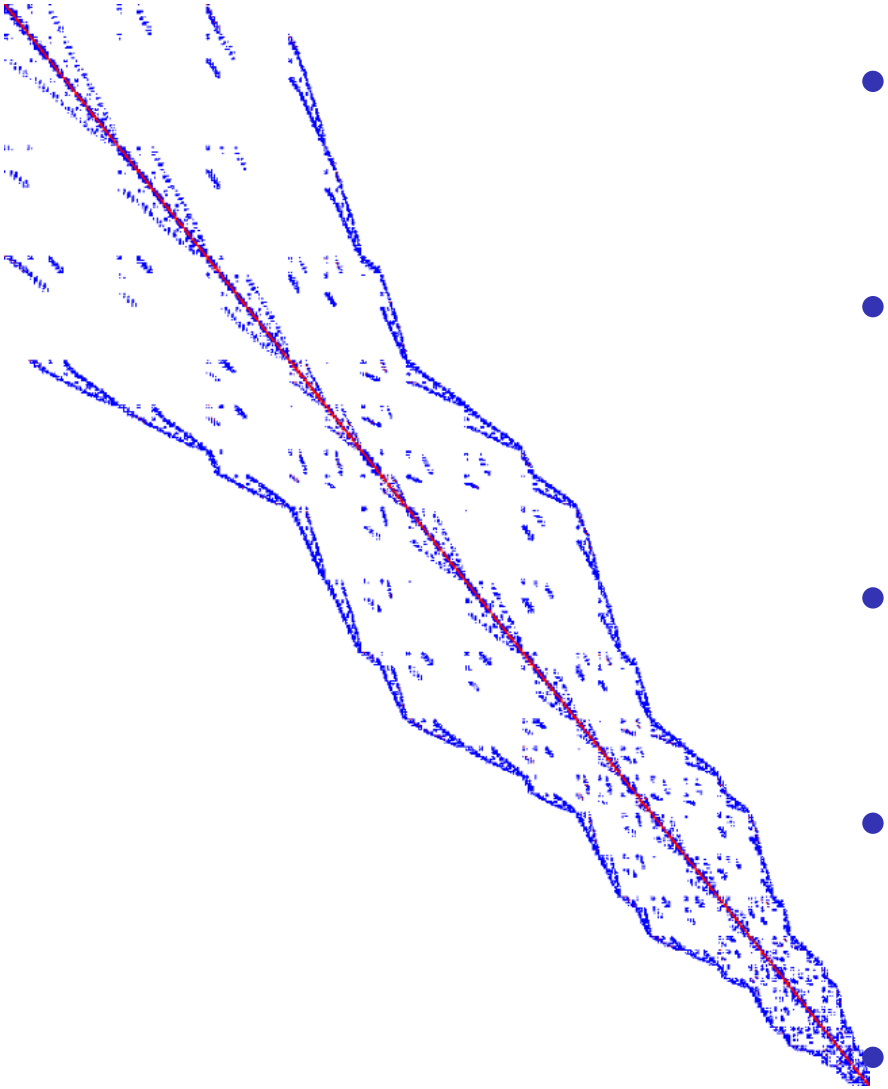
- Using PETSc linear algebra, just add:
  - `KSPSetOperators(KSP ksp, Mat A, Mat M, MatStructure flag)`
  - `KSPSolve(KSP ksp, Vec b, Vec x)`
- Can access subobjects
  - `KSPGetPC(KSP ksp, PC *pc)`
- Preconditioners must obey PETSc interface
  - Basically just the KSP interface
- Can change solver dynamically from the command line, `-ksp_type`

Nonlinear Solvers

Newton and Picard Methods

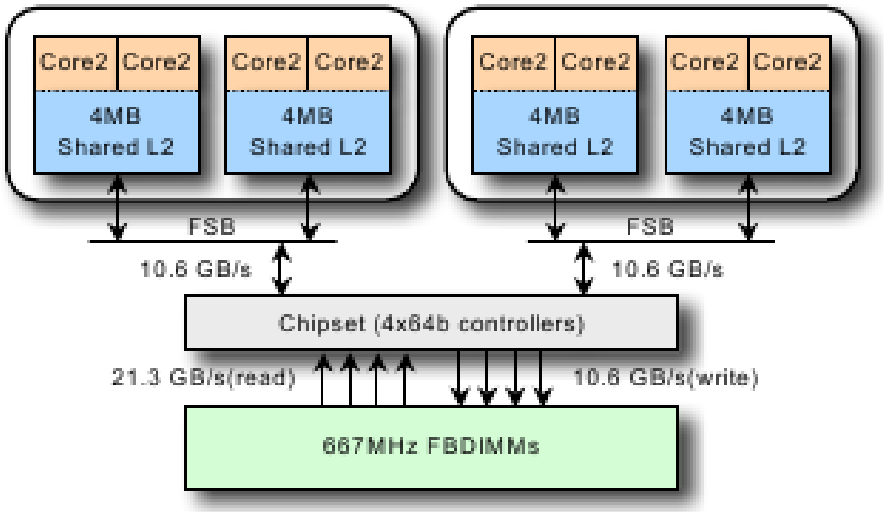
- Using PETSc linear algebra, just add:
  - `SNESSetFunction(SNES snes, Vec r, residualFunc, void *ctx)`
  - `SNESSetJacobian(SNES snes, Mat A, Mat M, jacFunc, void *ctx)`
  - `SNESSolve(SNES snes, Vec b, Vec x)`
- Can access subobjects
  - `SNESGetKSP(SNES snes, KSP *ksp)`
- Can customize subobjects from the cmd line
  - Set the subdomain preconditioner to ILU with `-sub_pc_type ilu`

Bottlenecks of (Jacobian-free) Newton-Krylov



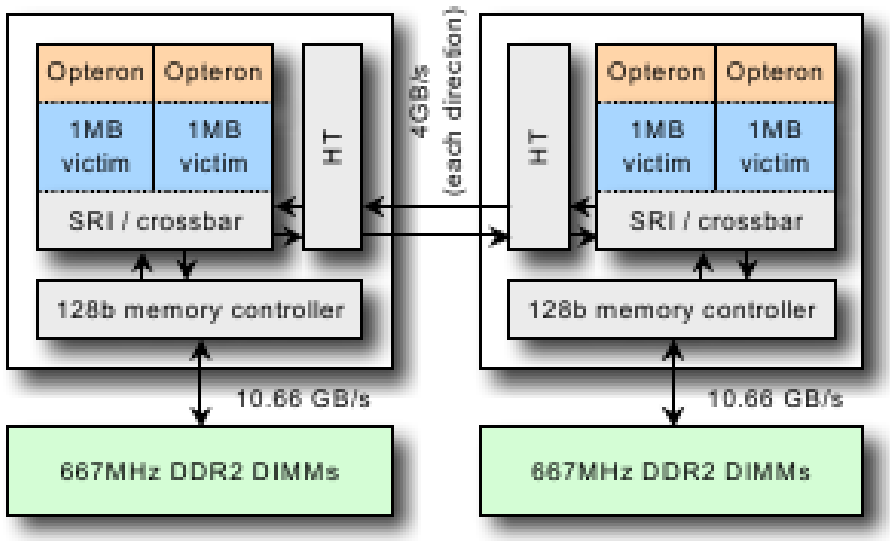
- Matrix assembly
  - integration/fluxes: FPU
  - insertion: memory/branching
- Preconditioner setup
  - coarse level operators
  - overlapping subdomains
  - (incomplete) factorization
- Preconditioner application
  - triangular solves/relaxation: memory
  - coarse levels: network latency
- Matrix multiplication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

Intel Clowertown



- 75 Gflop/s
- 21 GB/s bandwidth
- thread + instruction level parallelism
- vector instructions (SSE)

AMD Opteron



- 17 Gflop/s
- 21 GB/s bandwidth
- thread + instruction level parallelism
- vector instructions (SSE)

Hardware capabilities

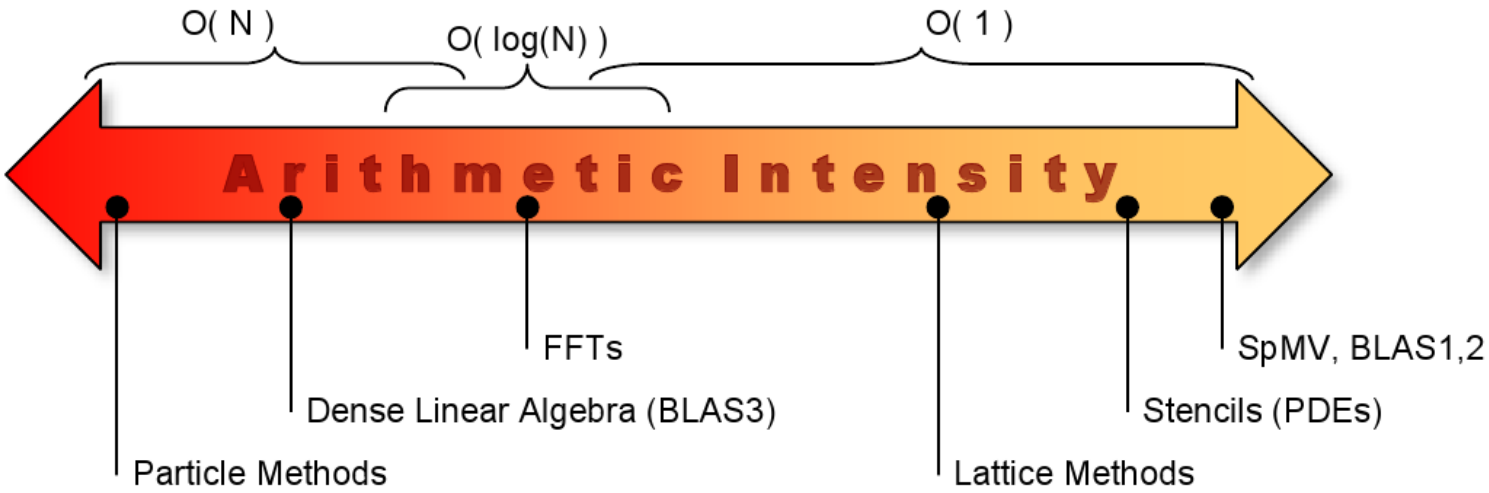
Floating point unit

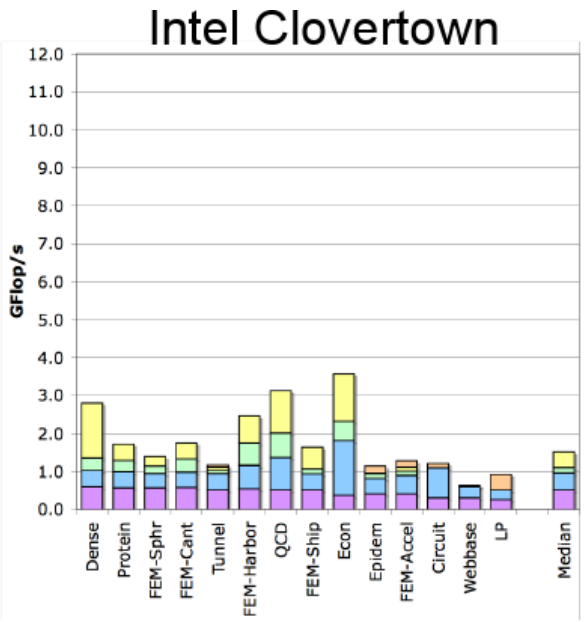
Recent Intel: each core can issue

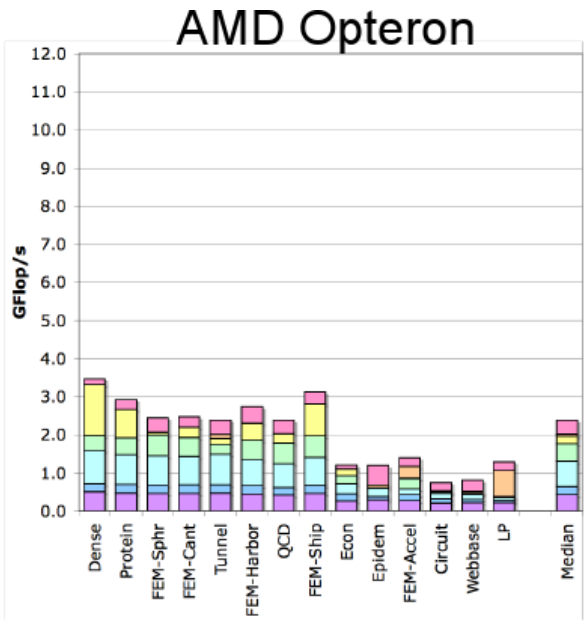
- 1 packed add (latency 3)
- 1 packed mult (latency 5)
- One can include a read
- Out of Order execution
- Peak: 10 Gflop/s (`double`)

Memory

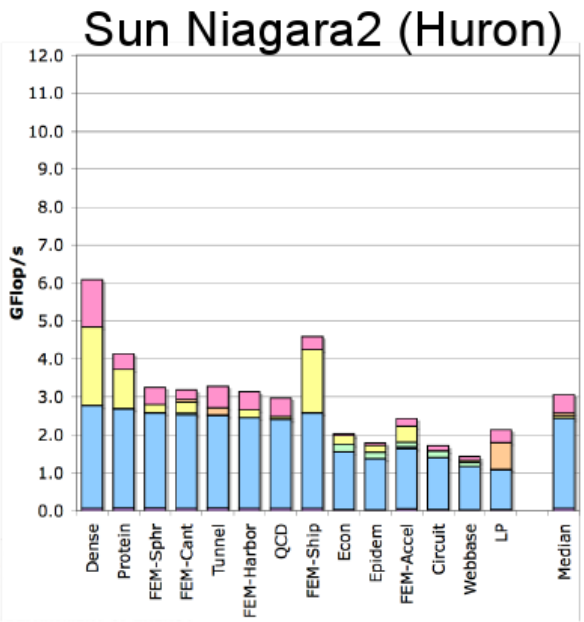
- $\sim$ 250 cycle latency
- 5.3 GB/s bandwidth
- 1 `double` load / 3.7 cycles
- Pay by the cache line (32/64 B)
- L2 cache: $\sim$ 10 cycle latency

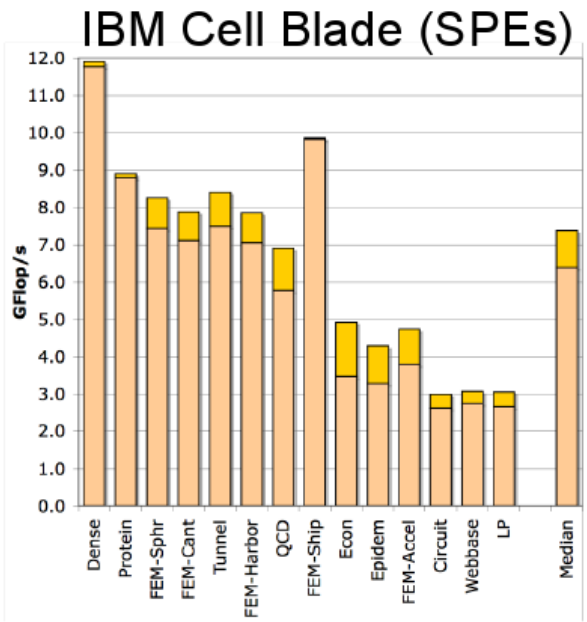


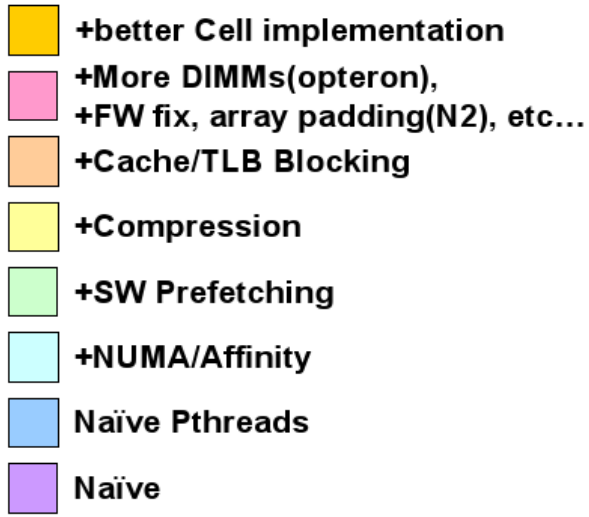




- Model faster cores by commenting out the inner kernel calls, but still performing all DMAs
- Enabled 1x1 BCOO
- ~16% improvement







(Oliker et al. *Multi-core Optimization of Sparse Matrix Vector Multiplication*, 2008)

Sparse Mat-Vec performance model

Compressed Sparse Row format (AIJ)

For $m \times n$ matrix with N nonzeros

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

$y \leftarrow y + Ax$

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

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

Memory Bandwidth

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

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

- Sparse matrix-vector product: 6 bytes per flop

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

Optimizing Sparse Mat-Vec

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

Optimizing unassembled Sparse Mat-Vec

- High order spatial discretizations do more work per node
  - Dense tensor product kernel (like small BLAS3)
  - Cubic (Q_3) elements in 3D can achieve $> 60\%$ of peak FPU (compare to $< 6\%$ for assembled operators on multicore)
  - Can store Jacobian information at quadrature points (usually pays off for Q_2 and higher in 3D)
  - Spectral methods
  - Often still need an assembled operator for preconditioning
- Boundary element methods
  - Dense kernels
  - Fast Multipole Method (FMM)

Profiling

- Use `-log_summary` for a performance profile
  - Event timing
  - Event flops
  - Memory usage
  - MPI messages
- Call `PetscLogStagePush()` and `PetscLogStagePop()`
  - User can add new stages
- Call `PetscLogEventBegin()` and `PetscLogEventEnd()`
  - User can add new events
- Call `PetscLogFlops()` to include your flops

Reading `-log_summary`

-

	Max	Max/Min	Avg	Total
Time (sec):	1.548e+02	1.00122	1.547e+02	
Objects:	1.028e+03	1.00000	1.028e+03	
Flops:	1.519e+10	1.01953	1.505e+10	1.204e+11
Flops/sec:	9.814e+07	1.01829	9.727e+07	7.782e+08
MPI Messages:	8.854e+03	1.00556	8.819e+03	7.055e+04
MPI Message Lengths:	1.936e+08	1.00950	2.185e+04	1.541e+09
MPI Reductions:	2.799e+03	1.00000		

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

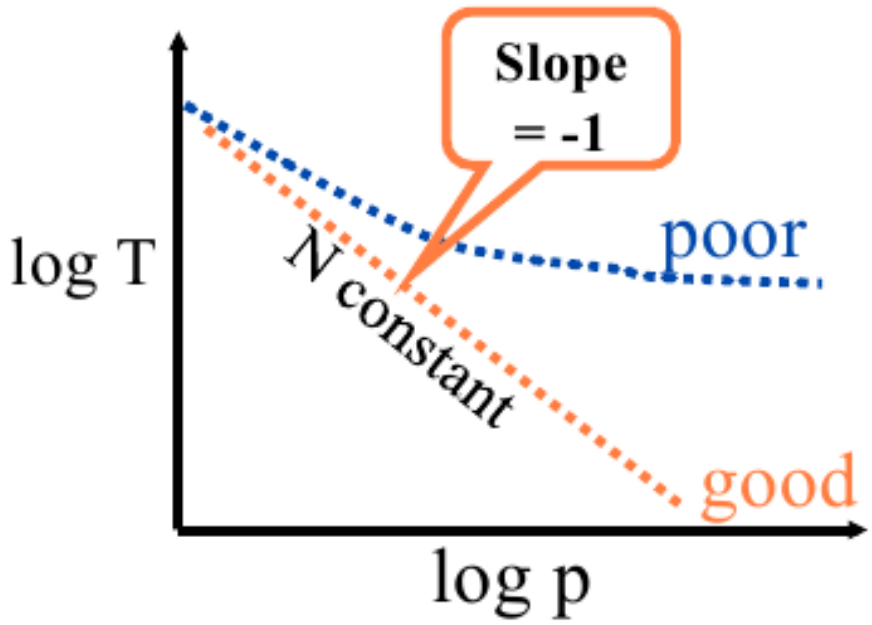
Communication Costs

- Reductions: usually part of Krylov method, latency limited
  - `VecDot`
  - `VecMDot`
  - `VecNorm`
  - `MatAssemblyBegin`
  - Change algorithm (e.g. IBCGS)
- Point-to-point (nearest neighbor), latency or bandwidth
  - `VecScatter`
  - `MatMult`
  - `PCApply`
  - `MatAssembly`
  - `SNESFunctionEval`
  - `SNESJacobianEval`
  - Compute subdomain boundary fluxes redundantly
  - Ghost exchange for all fields at once
  - Better partition

Scalability definitions

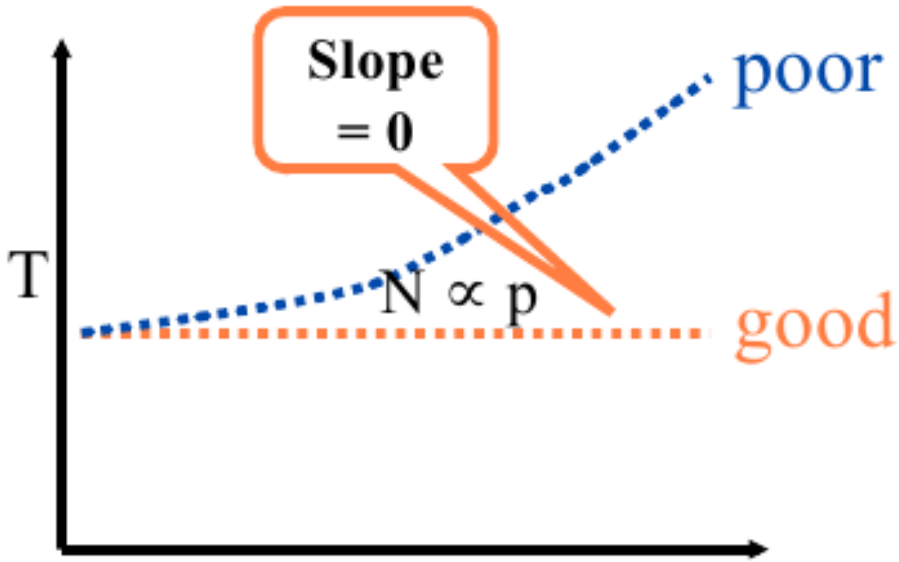
Strong scalability

- Fixed problem size
- execution time T inversely proportional to number of processors p



Weak scalability

- Fixed problem size per processor
- execution time constant as problem size increases



Multigrid advice

- Rapid coarsening is essential for weak scalability
  - Push the algorithm towards “multilevel domain decomposition”
- Energy minimizing interpolants
  - Similar to exotic Schwarz methods, see Dohrmann and Widlund 2008, 2009
  - Closely related to FETI-DP/BDDC coarse spaces
- Interpolation operators must be compatible with physics (e.g. inf-sup conditions)
- Ordering of unknowns can make incomplete factorization behave similar to line smoothers
- Nonlinear multigrid (FAS) is worth trying if pointwise or block residuals are cheap, or globalization is especially challenging
- Monotone multigrid (Kornhuber) for variational inequalities
- Boundary conditions in subdomain problems (“optimized Schwarz”)

Splitting for Multiphysics

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

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

$$\begin{bmatrix} A & \\ C & D \end{bmatrix}$$

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

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

  - robust (exact factorization), can often drop lower block
  - how to precondition S which is usually dense?

Coupled approach to multiphysics

- Smooth all components together
  - Block SOR is the most popular
  - Vanka smoothers for indefinite problems
- Interpolate in a compatible way
- Scaling between fields is critical
- First-order upwind for transport
- Coarse space and subdomain problems should be compatible with inf-sup condition
- Open research area

Anisotropy and Heterogeneity

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

Physics-based preconditioners (semi-implicit method)

Shallow water with stiff gravity wave

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

$$h_t - (uh)_x = 0$$

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

Semi-implicit method

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

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

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

Delta form

- Preconditioner should work like the Newton step

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

- Recast semi-implicit method in delta form

$$\frac{\delta h}{\Delta t} + (\delta uh)_x = -F_0$$

$$\frac{\delta uh}{\Delta t} + gh^n(\delta h)_x = -F_1$$

- Eliminate δuh

$$\frac{\delta h}{\Delta t} - \Delta t(gh^n(\delta h)_x)_x = -F_0 + (\Delta t F_1)_x$$

- Solve for δh, then evaluate

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

- Fully implicit solver
  - Is nonlinearly consistent (no splitting error)
  - Can be high-order in time

References

- Knoll and Keyes, *Jacobian-free Newton-Krylov methods: a survey of approaches and applications*, JCP, 2004.
- Elman et. al., *A taxonomy and comparison of parallel block multi-level preconditioners for the incompressible Navier-Stokes equations*, JCP, 2008.
- Wan, Chan, and Smith, *An energy-minimizing interpolation for robust multigrid methods*, SIAM J. Sci. Comp, 2000.
- Gropp, Kaushik, Keyes, Smith, *Performance Modeling and Tuning of an Unstructured Mesh CFD Application*, Supercomputing, 2000.