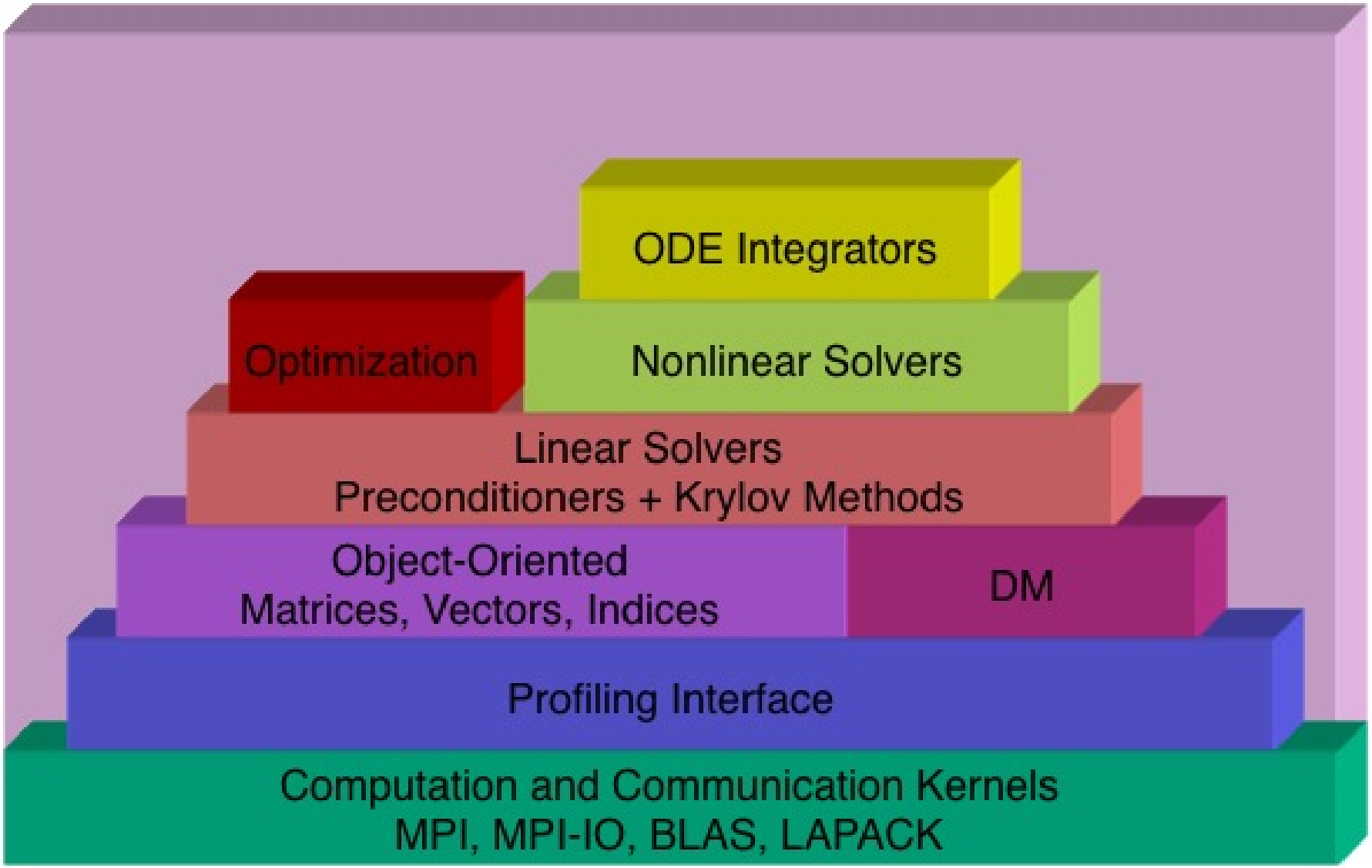
PETSc Tutorial

NCSA/UIUC

August 19th, 2015

Barry Smith

PETSc Structure



Valgrind

Valgrind is a debugging framework

- **Memcheck**: Check for memory overwrite and illegal use
- **Callgrind**: Generate call graphs
- **Cachegrind**: Monitor cache usage
- **Helgrind**: Check for race conditions
- **Massif**: Monitor memory usage

Memcheck will catch

- Illegal reads and writes to memory
- Uninitialized values
- Illegal frees
- Overlapping copies
- Memory leaks

Let's try a simple experiment

```
# Memcheck is the default tool
valgrind --trace-children=yes --suppressions=bin/simple.supp \
  ./bin/ex5 -use_coords
# Try it for multiple processes
valgrind --trace-children=yes --suppressions=bin/simple.supp \
  $PETSC_DIR/$PETSC_ARCH/bin/mpiexec -n 2 ./bin/ex5 -use_coords
```

We get an error!

```
==13697== Invalid read of size 8
==13697==    at 0x100005263: MyInitialGuess(AppCtx*, _p_Vec*) (myStuff.c:45)
==13697==    by 0x100004447: main (ex5.c:202)
==13697==  Address 0x103dc6fa0 is 0 bytes after a block of size 48 alloc'd
==13697==    at 0x10001ED75: malloc (vg_replace_malloc.c:236)
==13697==    by 0x1005CABC4: PetscMallocAlign(unsigned long, int, char const*, char const*, char c
==13697==    by 0x1009CC07D: VecGetArray2d(_p_Vec*, int, int, int, int, double***) (rvector.c:1739
==13697==    by 0x10030D980: DMDAVecGetArray(_p_DM*, _p_Vec*, void*) (dagetarray.c:72)
==13697==    by 0x100005102: MyInitialGuess(AppCtx*, _p_Vec*) (myStuff.c:38)
==13697==    by 0x100004447: main (ex5.c:202)
==13697==
==13697== Invalid read of size 8
==13697==    at 0x100005273: MyInitialGuess(AppCtx*, _p_Vec*) (myStuff.c:45)
==13697==    by 0x100004447: main (ex5.c:202)
==13697==  Address 0x18 is not stack'd, malloc'd or (recently) free'd
==13697==
==13698== Use of uninitialised value of size 8
==13698==    at 0x10000529D: MyInitialGuess(AppCtx*, _p_Vec*) (myStuff.c:45)
==13698==    by 0x100004447: main (ex5.c:202)
==13698==
==13698== Invalid read of size 8
==13698==    at 0x10000529D: MyInitialGuess(AppCtx*, _p_Vec*) (myStuff.c:45)
==13698==    by 0x100004447: main (ex5.c:202)
==13698==  Address 0x6f5c300000018 is not stack'd, malloc'd or (recently) free'd
```

Valgrind
Massif

```
# Memcheck is the default tool
valgrind --tool=massif --trace-children=yes \
  --massif-out-file=vecfem.massif \
  ./vecfem --sizes=[100,100] -ksp_rtol 1.0e-9
# Turn on stack profiling
valgrind --tool=massif --trace-children=yes \
  --massif-out-file=vecfem.massif \
  ./vecfem --stacks=yes --sizes=[100,100] -ksp_rtol 1.0e-9
# Visualize output
ms_print --threshold=10.0 vecfem.massif
```

Correctness Debugging

PETSc provides

- Automatic generation of tracebacks
- Detection of memory corruption and leaks
- Optional user-defined error handlers

Correctness Debugging

- What does `./configure --with-debugging=1` (default) do?
  - Keeps debugging symbols (of course)
  - Maintains a stack so that errors produce a full stack trace (even SEGV)
  - Does lots of integrity checking of user input
  - Places sentinels around allocated memory to detect memory errors
  - Allocates related memory chunks separately (to help find memory bugs)
  - Keeps track of and reports unused options
  - Keeps track of and reports allocated memory that is not freed

    `-malloc_dump`

Interacting with the Debugger

- Launch the debugger
  - `-start_in_debugger [gdb,dbx,noxterm]`
  - `-on_error_attach_debugger [gdb,dbx,noxterm]`
- Attach the debugger only to some parallel processes
  - `-debugger_nodes 0,1`
- Set the display (often necessary on a cluster)
  - `-display khan.mcs.anl.gov:0.0`

Interacting with the Debugger

```
  $ ./ex6 -start_in_debugger noxterm,lldb
[0]PETSC ERROR: PETSC: Attaching lldb to ./ex6 of pid 7432
Process 7432 stopped
    frame 0: 0x00007fff8d94b48a libsystem_kernel.dylib`__se
libsystem_kernel.dylib`__semwait_signal:
->  0x7fff8d94b48a <+10>: jae    0x7fff8d94b494
    0x7fff8d94b48c <+12>: movq   %rax, %rdi
    0x7fff8d94b48f <+15>: jmp    0x7fff8d946c78
    0x7fff8d94b494 <+20>: retq
(lldb) c
Process 7432 resuming
(lldb)
Process 7432 stopped
    frame 0: 0x0000000102ecbb80 ex6`main(argc=3, args=0x000
   71      ierr = PetscBinaryRead(fd,avec,sz,PETSC_SCALAR);C
-> 72      avec[10000000] = 23;
   73      ierr = VecRestoreArray(vec,&avec);CHKERRQ(ierr);
(lldb)
```

Time integration in PETSc

- ODE forms supported

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

- User provides:
  - `FormRHSFunction(ts,`t, x, F`,void *ctx);`
  - `FormIFunction(ts,`$t, x, \dot{x}, G$`,void *ctx);`
  - `FormIJacobian(ts,`$t, x, \dot{x}, \alpha, J, J_p$`,void *ctx);`

Motivation for IMEX time integration

- Explicit methods are easy and accurate, but must resolve all time scales
  - reactions, acoustics, incompressibility
- Implicit methods are robust
  - mathematically good for stiff systems
  - harder to implement, need efficient solvers
- Implicit-explicit methods are fragile and complicated
  - Severe order reduction
  - Still need implicit solvers, similar complexity to implicit
  -

Motivation for IMEX time integration

- Explicit methods are easy and accurate, but must resolve all time scales
  - reactions, acoustics, incompressibility
- Implicit methods are robust
  - mathematically good for stiff systems
  - harder to implement, need efficient solvers
- Implicit-explicit methods are fragile and complicated
  - Severe order reduction
  - Still need implicit solvers, similar complexity to implicit
  - Why bother?

Motivation for IMEX time integration

- Explicit methods are easy and accurate, but must resolve all time scales
  - reactions, acoustics, incompressibility
- Implicit methods are robust
  - mathematically good for stiff systems
  - harder to implement, need efficient solvers
- Implicit-explicit methods are fragile and complicated
  - Severe order reduction
  - Still need implicit solvers, similar complexity to implicit
  - Very expensive non-stiff residual evaluation
  - Non-stiff components are non-smooth.
    - TVD limiters for monotone transport
    - Phase change

IMEX time integration in PETSc

- Can have L-stable DIRK for stiff part G, SSP explicit part, etc.
- Orders 2 through 5, embedded error estimates
- Dense output, hot starts for Newton
- More accurate methods if G is linear, also Rosenbrock-W
- Can use preconditioner from classical “semi-implicit” methods
- FAS nonlinear solves supported
- Extensible adaptive controllers, can change order within a family
- Easy to register new methods: `TSARKIMEXRegister()`
- Single step interface so user can have own time loop
- Same interface for Extrapolation IMEX

Some TS methods

TSSSPRK104 10-stage, fourth order, low-storage, optimal explicit SSP Runge-Kutta $c_{\text{eff}} = 0.6$ (Ketcheson 2008)

TSARKIMEX2E second order, one explicit and two implicit stages, L-stable, optimal (Constantinescu)

TSARKIMEX3 (and 4 and 5), L-stable (Kennedy and Carpenter, 2003)

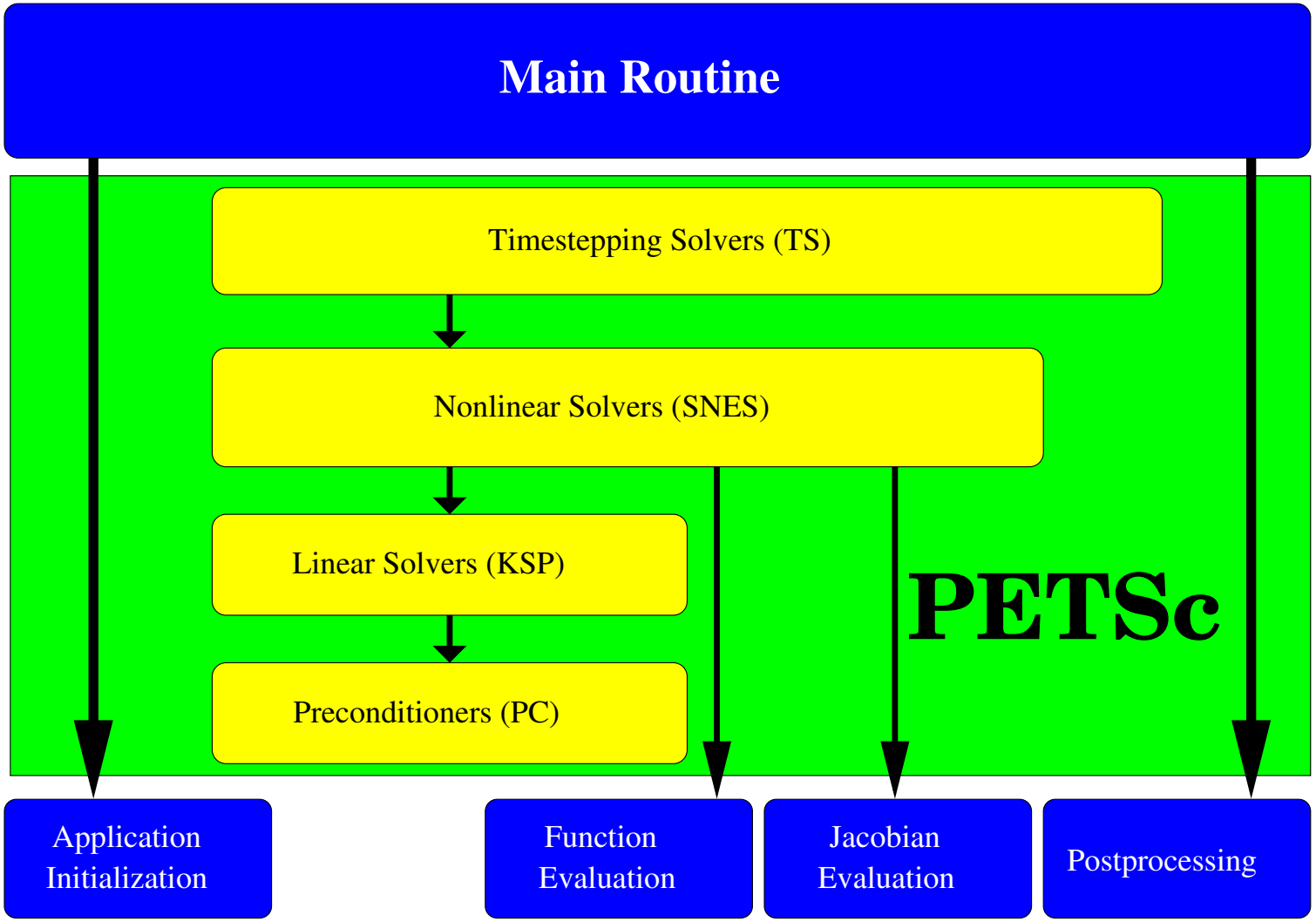
TSROSWRA3PW three stage, third order, for index-1 PDAE, A-stable, $R(\infty) = 0.73$, second order strongly A-stable embedded method (Rang and Angermann, 2005)

TSROSWRA34PW2 four stage, third order, L-stable, for index 1 PDAE, second order strongly A-stable embedded method (Rang and Angermann, 2005)

TSROSWLLSSP3P4S2C four stage, third order, L-stable implicit, SSP explicit, L-stable embedded method (Constantinescu)

TS Examples

- 1D nonlinear hyperbolic conservation laws
  - `src/ts/examples/tutorials/ex9.c`
  - `./ex9 -da_grid_x 100 -initial 1 -physics shallow -limit minmod -ts_ssp_type rks2 -ts_ssp_nstages 8 -ts_monitor_draw_solution`
- Stiff linear advection-reaction test problem
  - `src/ts/examples/tutorials/ex22.c`
  - `./ex22 -da_grid_x 200 -ts_monitor_draw_solution -ts_type rosw -ts_rosw_type ra34pw2 -ts_adapt_monitor`
- 1D Brusselator (reaction-diffusion)
  - `src/ts/examples/tutorials/ex25.c`
  - `./ex25 -da_grid_x 40 -ts_monitor_draw_solution -ts_type rosw -ts_rosw_type 2p -ts_adapt_monitor`



- IGA used to evaluate nonlinear residuals
- PETSc `DA` used to manage parallelism.
- Adaptive time integration using method of lines.
  - Generalized α method from PETSc `TS`.

The PETSc Programming Model

- Goals
  - Portable, runs everywhere
  - High performance
  - Scalable parallelism
- Approach
  - Distributed memory (“shared-nothing”)
  - No special compiler
  - Access to data on remote machines through MPI
  - Hide within objects the details of the communication
  - User orchestrates communication at a higher abstract level

Library Design

Numerical libraries should interact at a higher level than MPI

- MPI coordinates data movement and synchronization for data parallel applications
- Numerical libraries should coordinate access to a given data structure
  - MPI can handle data parallelism and something else (runtime engine) handle task parallelism (van de Geijn, Strout, Demmel)
  - Algorithm should be data structure neutral, but its main operation is still to structure access

Collectivity

- MPI communicators (`MPI_Comm`) specify collectivity
  - Processes involved in a computation
- Constructors are collective over a communicator
  - `VecCreate(MPI_Comm comm, Vec *x)`
  - Use `PETSC_COMM_WORLD` for all processes and `PETSC_COMM_SELF` for one
- Some operations are collective, while others are not
  - collective: `VecNorm()`
  - not collective: `VecGetLocalSize()`
- Sequences of collective calls must be in the same order on each process

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

A PETSc Vec

- Supports all vector space operations
  - `VecDot(), VecNorm(), VecScale()`
- Has a direct interface to the values
  - `VecGetArray(), VecGetArrayF90()`
- Has unusual operations
  - `VecSqrtAbs(), VecStrideGather()`
- Communicates automatically during assembly
- Has customizable communication (**VecScatter**)

Object-Oriented Design

- Design based on operations you perform,
  - rather than the data in the object
- Example: A vector is
  - not a 1d array of numbers
  - an object allowing addition and scalar multiplication
- The efficient use of the computer is an added difficulty
  - which often leads to code generation

Vector Algebra

What are PETSc vectors?

- Fundamental objects representing field solutions, right-hand sides, etc.
- Each process locally owns a subvector of contiguous global data

How do I create vectors?

- `VecCreate(MPI_Comm, Vec *)`
- `VecSetSizes(Vec, int n, int N)`
- `VecSetType(Vec, VecType typeName)`
- `VecSetFromOptions(Vec)`
  - Can set the type at runtime

Vector Algebra

A PETSc Vec

- Has a direct interface to the values
- Supports all vector space operations
  - `VecDot()`, `VecNorm()`, `VecScale()`
- Has unusual operations, e.g. `VecSqrt()`, `VecWhichBetween()`
- Communicates automatically during assembly
- Has customizable communication (scatters)

Parallel Assembly
Vectors and Matrices

- Processes may set an arbitrary entry
  - Must use proper interface
- Entries need not be generated locally
  - Local meaning the process on which they are stored
- PETSc automatically moves data if necessary
  - Happens during the assembly phase

Vector Assembly

- A three step process
  - Each process sets or adds values
  - Begin communication to send values to the correct process
  - Complete the communication
- `VecSetValues(Vec v, int n, int rows[], PetscScalar values[], mode)`
  - `mode` is either INSERT_VALUES or ADD_VALUES
- Two phase assembly allows overlap of communication and computation
  - `VecAssemblyBegin(Vec v)`
  - `VecAssemblyEnd(Vec v)`

One Way to Set the Elements of a Vector

```
VecGetSize(x, &N);
MPI_Comm_rank(PETSC_COMM_WORLD, &rank);
if (rank == 0) {
  for(i = 0, val = 0.0; i < N; i++, val += 10.0) {
    VecSetValues(x, 1, &i, &val, INSERT_VALUES);
  }
}
/* These routines ensure that the data is distributed
to the other processes */
VecAssemblyBegin(x);
VecAssemblyEnd(x);
```

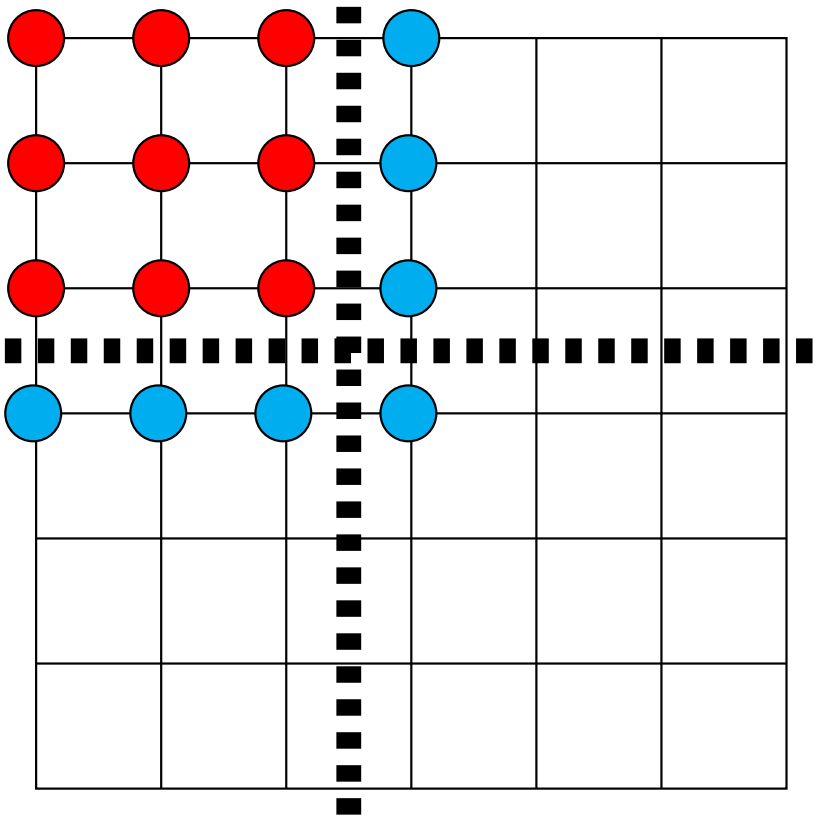
A Better Way to Set the Elements of a Vector

```
VecGetOwnershipRange(x, &low, &high);
for(i = low,val = low*10.0; i < high; i++,val += 10.0)
{
  VecSetValues(x, 1, &i, &val, INSERT_VALUES);
}
/* These routines ensure that the data is distributed
to the other processes */
VecAssemblyBegin(x);
VecAssemblyEnd(x);
```

Ghost Values

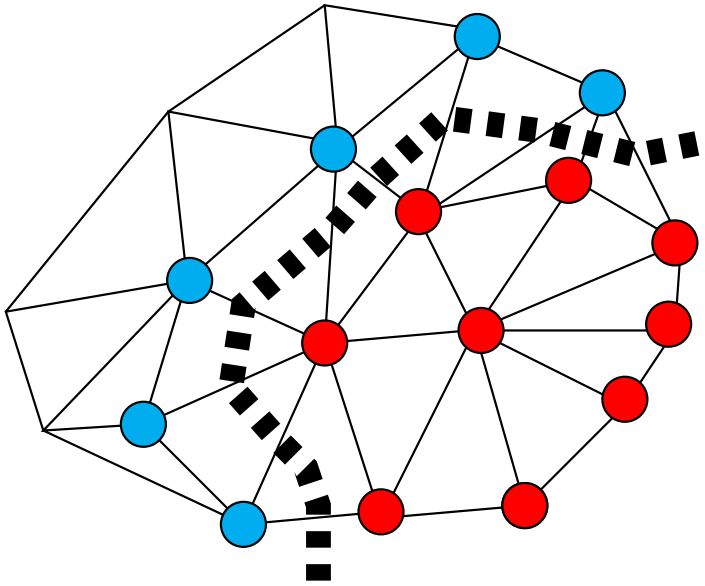
To evaluate a local function $f(x)$, each process requires

- its local portion of the vector x
- its ghost values, bordering portions of x owned by neighboring processes



Local Node

Ghost Node



Working With Local Vectors

It is sometimes more efficient to directly access local storage of a `Vec`.

- PETSc allows you to access the local storage with
  - `VecGetArray(Vec, double *[])`
- You must return the array to PETSc when you finish
  - `VecRestoreArray(Vec, double *[])`
- Allows PETSc to handle data structure conversions
  - Commonly, these routines are inexpensive and do not involve a copy

VecGetArray in C

```
Vec v;
PetscScalar *array;
PetscInt n, i;
PetscErrorCode ierr;

VecGetArray(v, &array);
VecGetLocalSize(v, &n);
PetscSynchronizedPrintf(PETSC_COMM_WORLD,
 "First element of local array is %f\n", array[0]);
PetscSynchronizedFlush(PETSC_COMM_WORLD);
for(i = 0; i < n; i++) {
  array[i] += (PetscScalar) rank;
}
VecRestoreArray(v, &array);
```

VecGetArray in F77

```
#include "finclude/petsc.h"
#include "finclude/petscvec.h"
Vec v;
PetscScalar array(1)
PetscOffset offset
PetscInt n, i
PetscErrorCode ierr

call VecGetArray(v, array, offset, ierr)
call VecGetLocalSize(v, n, ierr)
do i=1,n
  array(i+offset) = array(i+offset) + rank
end do
call VecRestoreArray(v, array, offset, ierr)
```

VecGetArray in F90

```
#include "finclude/petsc.h"
#include "finclude/petscvec.h"
#include "finclude/petscvec.h90"
Vec v;
PetscScalar pointer :: array(:)
PetscInt n, i
PetscErrorCode ierr

call VecGetArrayF90(v, array, ierr)
call VecGetLocalSize(v, n, ierr)
do i=1,n
  array(i) = array(i) + rank
end do
call VecRestoreArrayF90(v, array, ierr)
```

Selected Vector Operations

Function Name	Operation
VecAXPY(Vec y, PetscScalar a, Vec x)	$y = y + a * x$
VecAYPX(Vec y, PetscScalar a, Vec x)	$y = x + a * y$
VecWAYPX(Vec w, PetscScalar a, Vec x, Vec y)	$w = y + a * x$
VecScale(Vec x, PetscScalar a)	$x = a * x$
VecCopy(Vec y, Vec x)	$y = x$
VecPointwiseMult(Vec w, Vec x, Vec y)	$w_i = x_i * y_i$
VecMax(Vec x, PetscInt *idx, PetscScalar *r)	$r = \max r_i$
VecShift(Vec x, PetscScalar r)	$x_i = x_i + r$
VecAbs(Vec x)	$x_i = \|x_i\|$
VecNorm(Vec x, NormType type, PetscReal *r)	$r = \|\|x\|\|$

What is a DM?

- Interface for linear algebra to talk to grids
- Defines (topological part of) a finite-dimensional function space
  - Get an element from this space: `DMCreateGlobalVector()`
- Provides parallel layout
- Refinement and coarsening
  - `DMRefine()`, `DMCoarsen()`
- Ghost value coherence
  - `DMGlobalToLocalBegin()`
- Matrix preallocation:
  - `DMCreateMatrix()` (formerly `DMGetMatrix()`)

Topology Abstractions

- `DMDA`
  - Abstracts Cartesian grids in 1, 2, or 3 dimension
  - Supports stencils, communication, reordering
  - Nice for simple finite differences
- `DMPLEX`
  - Abstracts general topology in any dimension
  - Also supports partitioning, distribution, and global orders
  - Allows aribtrary element shapes and discretizations
- `DMCOMPOSITE`
  - Composition of two or more DMs
- `DMNetwork` - for discrete networks like power grids and circuits
- `DMMoab` - interface to the MOAB unstructured mesh library

DM Vectors

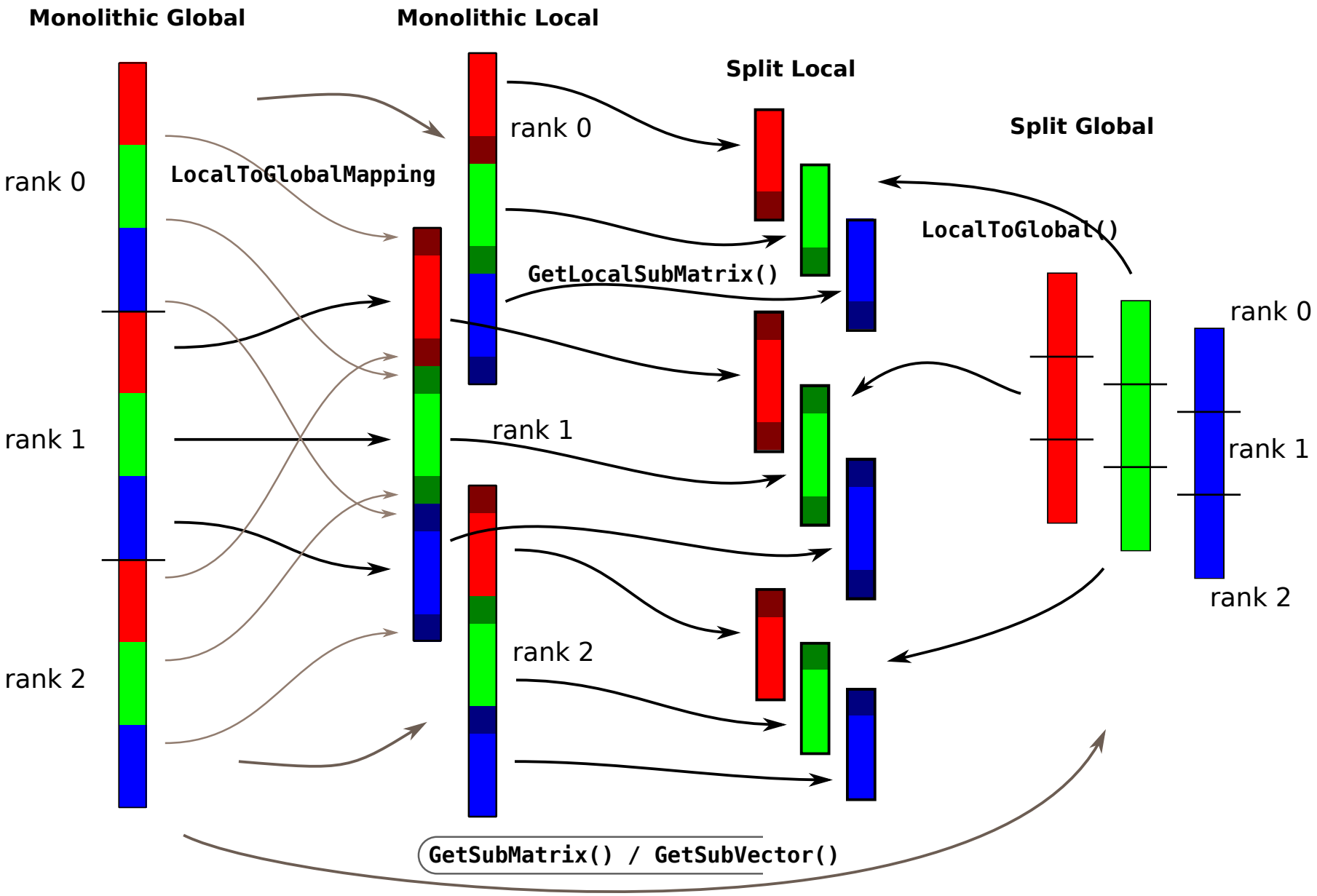
- The DM object contains only layout (topology) information
  - All field data is contained in PETSc **Vecs**
- Global vectors are parallel
  - Each process stores a unique local portion
  - `DMCreateGlobalVector(DM da, Vec *gvec)`
- Local vectors are sequential (and usually temporary)
  - Each process stores its local portion plus ghost values
  - `DMCreateLocalVector(DM da, Vec *lvec)`
  - includes ghost values!

Updating Ghosts

Two-step process enables overlapping computation and communication

- `DMGlobalToLocalBegin(dm, gvec, mode, lvec)`
  - `gvec` provides the data
  - `mode` is either `INSERT_VALUES` or `ADD_VALUES`
  - `lvec` holds the local and ghost values
- `DMGlobalToLocalEnd(dm, gvec, mode, lvec)`
  - Finishes the communication

The process can be reversed with `DMLocalToGlobalBegin()` and `DMLocalToGlobalEnd()`.



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

What is a DMDA?

`DMDA` is a topology interface handling parallel data layout on structured grids

- Handles local and global indices
  - `DMDAGetGlobalIndices()` and `DMDAGetAO()`
- Provides local and global vectors
  - `DMGetGlobalVector()` and `DMGetLocalVector()`
- Handles ghost values coherence
  - `DMGetGlobalToLocal()` and `DMGetLocalToGlobal()`

DMDA Global vs. Local Numbering

- **Global**: Each vertex has a unique id belongs on a unique process
- **Local**: Numbering includes vertices from neighboring processes
  - These are called ghost vertices

Proc 2			Proc 3	
X	X	X	X	X
X	X	X	X	X
12	13	14	15	X
8	9	10	11	X
4	5	6	7	X
0	1	2	3	X
Proc 0			Proc 1	

Local numbering

Proc 2			Proc 3	
21	22	23	28	29
18	19	20	26	27
15	16	17	24	25
6	7	8	13	14
3	4	5	11	12
0	1	2	9	10
Proc 0			Proc 1	

Global numbering

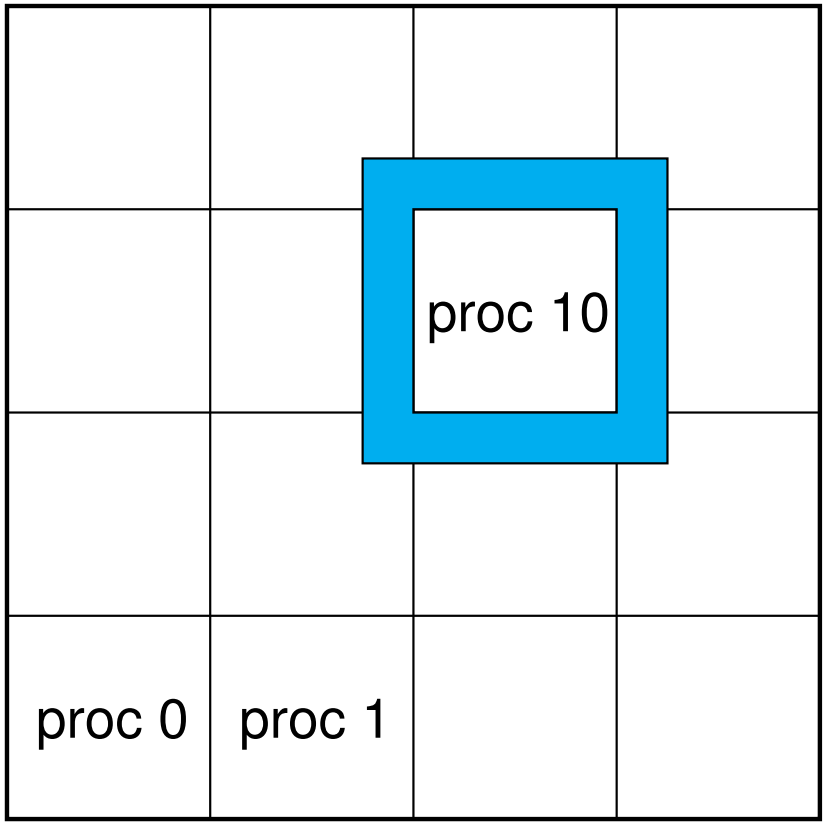
Creating a DADM

```
DMDACreate2d(comm, bdX, bdY, type, M, N, m, n, dof, s, lm[], ln[], DMDA *d
```

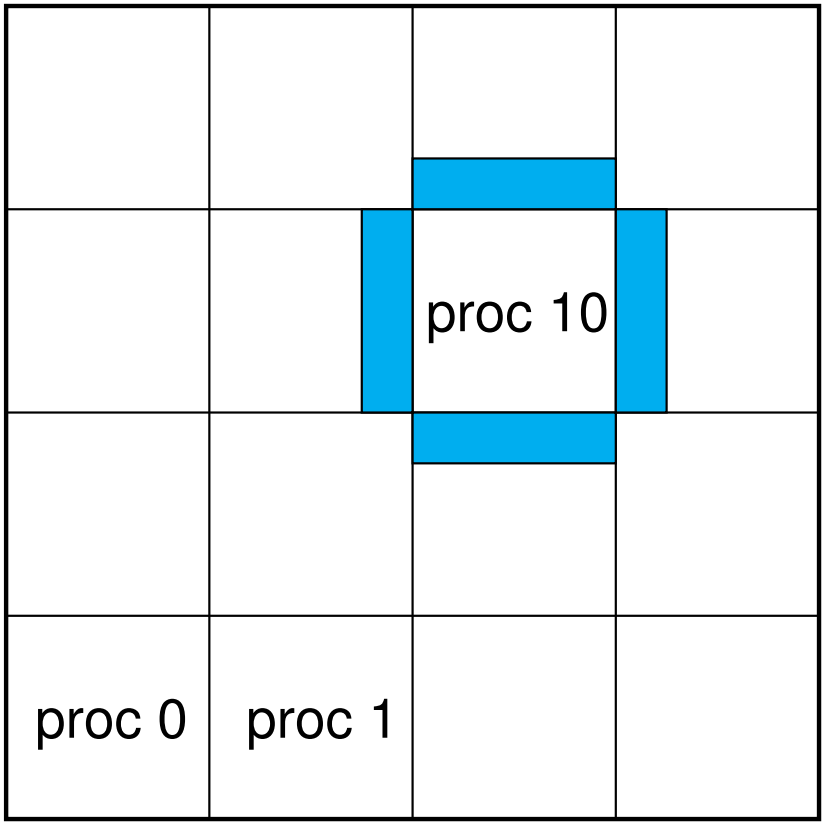
- `bd`: Specifies boundary behavior
  - `DMDA_BOUNDARY_NONE`, `DMDA_BOUNDARY_GHOSTED`, or `DMDA_BOUNDARY_PERIODIC`
- `type`: Specifies stencil
  - `DA_STENCIL_BOX` or `DA_STENCIL_STAR`
- `M/N`: Number of grid points in x/y-direction
- `m/n`: Number of processes in x/y-direction
- `dof`: Degrees of freedom per node
- `s`: The stencil width
- `lm/n`: Alternative array of local sizes
  - Use `PETSC_NULL` for the default

DMDA Stencils

Both the box stencil and star stencil are available.



Box Stencil



Star Stencil

Matrices

Definition (Matrix)

A matrix is a linear transformation between finite dimensional vector spaces.

Definition (Forming a matrix)

Forming or assembling a matrix means defining it's action in terms of entries (usually stored in a sparse format).

Matrices

Definition (Matrix)

A matrix is a linear transformation between finite dimensional vector spaces.

Definition (Forming a matrix)

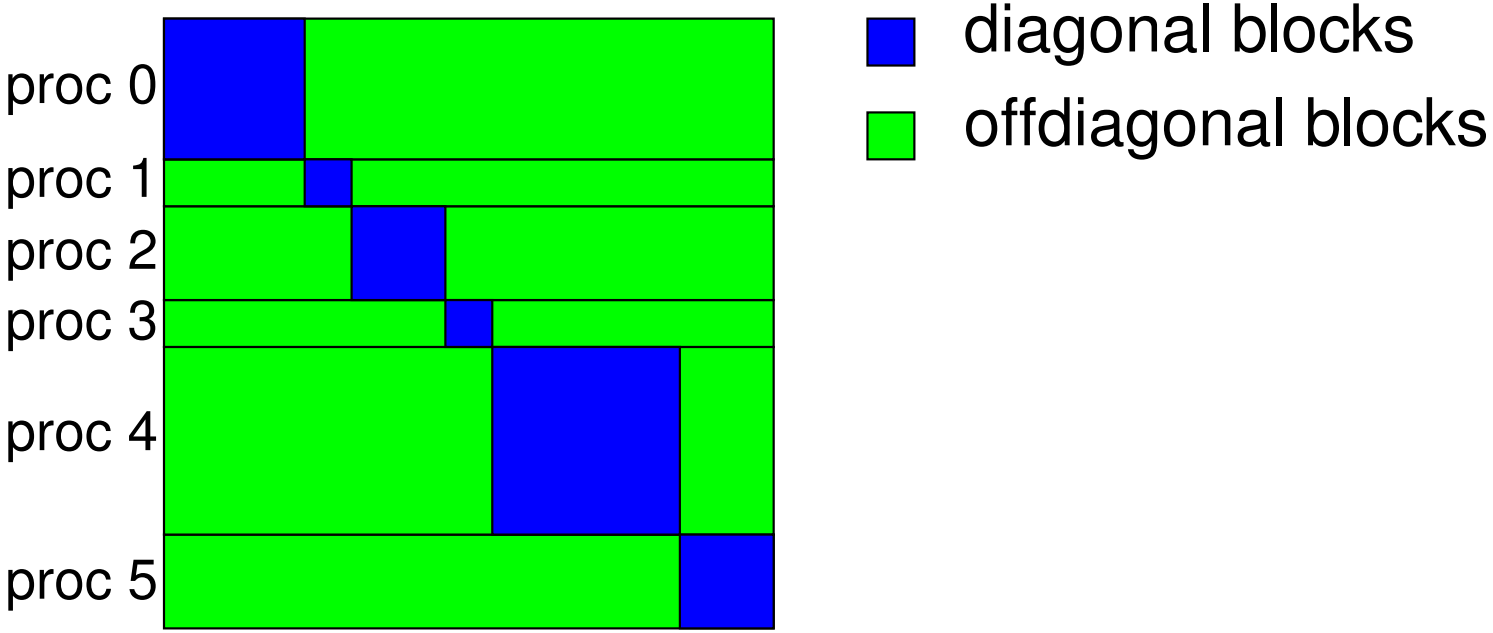
Forming or assembling a matrix means defining it's action in terms of entries (usually stored in a sparse format).

How do I create matrices?

- `MatCreate(MPI_Comm, Mat *)`
- `MatSetSizes(Mat, int m, int n, int M, int N)`
- `MatSetType(Mat, MatType typeName)`
- `MatSetFromOptions(Mat)`
  - Can set the type at runtime
- `MatMPIBAIJSetPreallocation(Mat,...)`
  - important for assembly performance, more tomorrow
- `MatSetBlockSize(Mat, int bs)`
  - for vector problems
- `MatSetValues(Mat,...)`
  - **MUST** be used, but does automatic communication
  - `MatSetValuesLocal(),MatSetValuesStencil()`
  - `MatSetValuesBlocked()`

Matrix Storage Layout

- 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

Matrix Assembly

- A three step process
  - Each process sets or adds values
  - Begin communication to send values to the correct process
  - Complete the communication
- `MatSetValues(Mat A, m, rows[], n, cols[], values[], mode)`
  - `mode` is either INSERT_VALUES or ADD_VALUES
  - Logically dense block of values
- Two phase assembly allows overlap of communication and computation
  - `MatAssemblyBegin(Mat m, type)`
  - `MatAssemblyEnd(Mat m, type)`
  - `type` is either MAT_FLUSH_ASSEMBLY or MAT_FINAL_ASSEMBLY
- For vector problems

```
MatSetValuesBlocked(Mat A, m, rows[],
        n, cols[], values[], mode)
```

- The same assembly code can build matrices of different format

Matrix Assembly

- A three step process
  - Each process sets or adds values
  - Begin communication to send values to the correct process
  - Complete the communication
- `MatSetValues(Mat A, m, rows[], n, cols[], values[], mode)`
  - `mode` is either INSERT_VALUES or ADD_VALUES
  - Logically dense block of values
- Two phase assembly allows overlap of communication and computation
  - `MatAssemblyBegin(Mat m, type)`
  - `MatAssemblyEnd(Mat m, type)`
  - `type` is either MAT_FLUSH_ASSEMBLY or MAT_FINAL_ASSEMBLY
- For vector problems

```
MatSetValuesBlocked(Mat A, m, rows[],
        n, cols[], values[], mode)
```

- The same assembly code can build matrices of different format

One Way to Set the Elements of a Matrix

Simple 3-point stencil for 1D Laplacian

```
v[0] = -1.0; v[1] = 2.0; v[2] = -1.0;
if (rank == 0) {
  for(row = 0;  row < N; row++) {
    cols[0] = row-1; cols[1] = row; cols[2] = row+1;
    if (row == 0) {
      MatSetValues(A,1,&row,2,&cols[1],&v[1],INSERT_VALUES)
    } else if (row == N-1) {
      MatSetValues(A,1,&row,2,cols,v,INSERT_VALUES);
    } else {
      MatSetValues(A,1,&row,3,cols,v,INSERT_VALUES);
    }
  }
}
MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);
MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);
```

A Better Way to Set the Elements of a Matrix

Simple 3-point stencil for 1D Laplacian

```
v[0] = -1.0; v[1] = 2.0; v[2] = -1.0;
for(row = start;  row < end; row++) {
  cols[0] = row-1; cols[1] = row; cols[2] = row+1;
  if (row == 0) {
    MatSetValues(A,1,&row,2,&cols[1],&v[1],INSERT_VALUES);
  } else if (row == N-1) {
    MatSetValues(A,1,&row,2,cols,v,INSERT_VALUES);
  } else {
    MatSetValues(A,1,&row,3,cols,v,INSERT_VALUES);
  }
}
MatAssemblyBegin(A, MAT_FINAL_ASSEMBLY);
MatAssemblyEnd(A, MAT_FINAL_ASSEMBLY);
```

Matrix Memory Preallocation

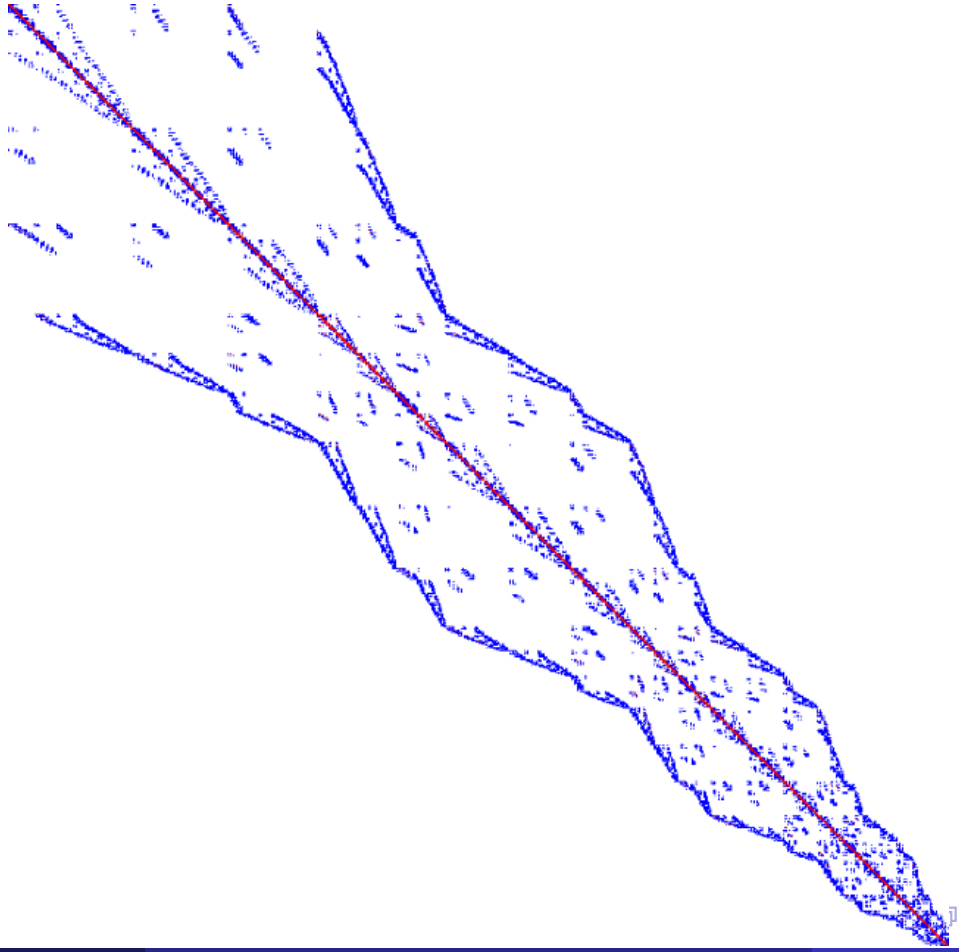
- PETSc sparse matrices are dynamic data structures
  - can add additional nonzeros freely
- Dynamically adding many nonzeros
  - requires additional memory allocations
  - requires copies
  - can kill performance
- Memory preallocation provides
  - the freedom of dynamic data structures
  - good performance
- 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

`MatSeqAIJSetPreallocation(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 Matrices

```
MatMPIAIJSetPreallocation(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 options
  `-mat_new_nonzero_location_err`
  `-mat_new_nonzero_allocation_err`
- 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!
```

Matrix Polymorphism

The PETSc `Mat` has a single user interface,

- Matrix assembly
  - `MatSetValues()`
- Matrix-vector multiplication
  - `MatMult()`
- Matrix viewing
  - `MatView()`

but multiple underlying implementations.

- AIJ, Block AIJ, Symmetric Block AIJ,
- Dense, Elemental
- Matrix-Free
- etc.

A matrix is defined by its interface, not by its data structure.

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

Performance of blocked matrix formats

Kernel \ Format	Core 2, 1 process			Opteron, 4 processes		
	AIJ	BAIJ	SBAIJ	AIJ	BAIJ	SBAIJ
`MatMult`	812	985	1507	2226	2918	3119
`MatSolve`	718	957	955	1573	2869	2858

Throughput (Mflop/s) for different matrix formats on Core 2 Duo (P8700) and Opteron 2356 (two sockets). `MatSolve` is a forward- and back-solve with incomplete Cholesky factors. The AIJ format is using "inodes" which unrolls across consecutive rows with identical nonzero pattern (pairs in this case).

Objects

```
Mat A;
PetscInt m,n,M,N;
MatCreate(comm,&A);
MatSetSizes(A,m,n,M,N);        /* or PETSC_DECIDE */
MatSetOptionsPrefix(A,"foo_");
MatSetFromOptions(A);
/* Use A */
MatView(A,PETSC_VIEWER_DRAW_WORLD);
MatDestroy(A);
```

- `Mat` is an opaque object (pointer to incomplete type)
  - Assignment, comparison, etc, are cheap
- What's up with this "Options" stuff?
  - Allows the type to be determined at runtime: `-foo_mat_type sbaij`
  - Inversion of Control similar to "service locator", related to "dependency injection"
  - Other options (performance and semantics) can be changed at

Matrices, redux

What are PETSc matrices?

- Linear operators on finite dimensional vector spaces.
- Fundamental objects for storing stiffness matrices and Jacobians
- Each process locally owns a contiguous set of rows
- Supports many data types
  - AIJ, Block AIJ, Symmetric AIJ, Block Diagonal, etc.
- Supports structures for many packages
  - MUMPS, Spooles, SuperLU, UMFPack, DSCPack

Matrices, redux

What are PETSc matrices?

- Linear operators on finite dimensional vector spaces.
- Fundamental objects for storing stiffness matrices and Jacobians
- Each process locally owns a contiguous set of rows
- Supports many data types
  - AIJ, Block AIJ, Symmetric AIJ, Block Diagonal, etc.
- Supports structures for many packages
  - MUMPS, Spooles, SuperLU, UMFPack, DSCPack

Why Are PETSc Matrices That Way?

- No one data structure is appropriate for all problems
  - Blocked and diagonal formats provide significant performance benefits
  - PETSc has many formats and makes it easy to add new data structures
- Assembly is difficult enough without worrying about partitioning
  - PETSc provides parallel assembly routines
  - Achieving high performance still requires making most operations local
  - However, programs can be incrementally developed.
  - `MatPartitioning` and `MatOrdering` can help
- Matrix decomposition in contiguous chunks is simple
  - Makes interoperation with other codes easier
  - For other ordering, PETSc provides “Application Orderings” (AO)

`MatGetLocalSubMatrix()` spaces

- Newton method for $F(x) = 0$ solves

$$J(x)\delta x = -F(x)$$

$$J = \begin{pmatrix} J_{aa} & J_{ab} & J_{ac} \\ J_{ba} & J_{bb} & J_{bc} \\ J_{ca} & J_{cb} & J_{cc} \end{pmatrix}.$$

- Conceptually, there are three spaces in parallel
  - V "monolithic" globally assembled space
  - V_i "split" global space for a single physics i
  - $\overline{V}_i$ Local space (with ghosts) for a single physcs i
  - $\overline{V}$ $\prod_i \overline{V}_i$ Concatenation of all single-physics local spaces
- Different components need different relationships

$V_i \to V$ field-split

$\overline{V} \to V$ coupled Neumann domain decomposition methods

$\overline{V}_i$ natural language for modular residual evaluation and assembly

```
MatGetLocalSubMatrix(Mat A,IS rows,IS cols,Mat *B);
```

- Primarily for assembly
  - `B` is not guaranteed to implement `MatMult`
  - The communicator for `B` is not specified,
    only safe to use non-collective ops (unless you check)
- `IS` represents an index set, includes a block size and communicator
- `MatSetValuesBlockedLocal()` is implemented
- MatNest returns nested submatrix, no-copy
- No-copy for Neumann-Neumann formats (unassembled across procs, e.g. BDDC, FETI-DP)
- Most other matrices return a lightweight proxy `Mat`
  - `COMM_SELF`
  - Values not copied, does not implement `MatMult`
  - Translates indices to the language of the parent matrix
  - Multiple levels of nesting are flattened

`MatGetLocalSubMatrix()` spaces

Spaces

- V Globally assembled space
- V_i Global space for a single physics i
- $\overline{V}_i$ Local space (with ghosts) for a single physcs i
- $\overline{V}$ $\prod_i \overline{V}_i$ Concatenation of all single-physics local spaces

- Multiple physics $x = [x_a, x_b, x_c]$
- I_i Map indices from V_i to V.
- R_i Global physics restriction $R_i : V \to V_i$

$$R_i x = x[I_i] = x_i$$

- $\overline{I}_i$ Map indices from $\overline{V}_i$ to V_i
- $\overline{R}_i$ Extract local single-physics part from global single-physics

$$\overline{R}_i x_i = x_i[\overline{I}_i] = \overline{x}_i$$

- $\tilde{I}_i$ Map indices from $\overline{V}_i$ to $\overline{V}$

`MatGetLocalSubMatrix()` spaces

- Globally assembled coupled matrix in terms of assembled single-physics blocks

$$J = \sum_{ij} R_i^T J_{ij} R_j$$

  - Language of Schwarz and fieldsplit
- Assembled single-physics blocks in terms of local single-physics matrices

$$J_{ij} = \overline{R}_i^T \overline{J}_{ij} \overline{R}_j$$

  - Language of assembly and Neumann/FETI domain decomposition
  - `MatSetValuesLocal()`

Setting Values on Regular Grids

PETSc provides

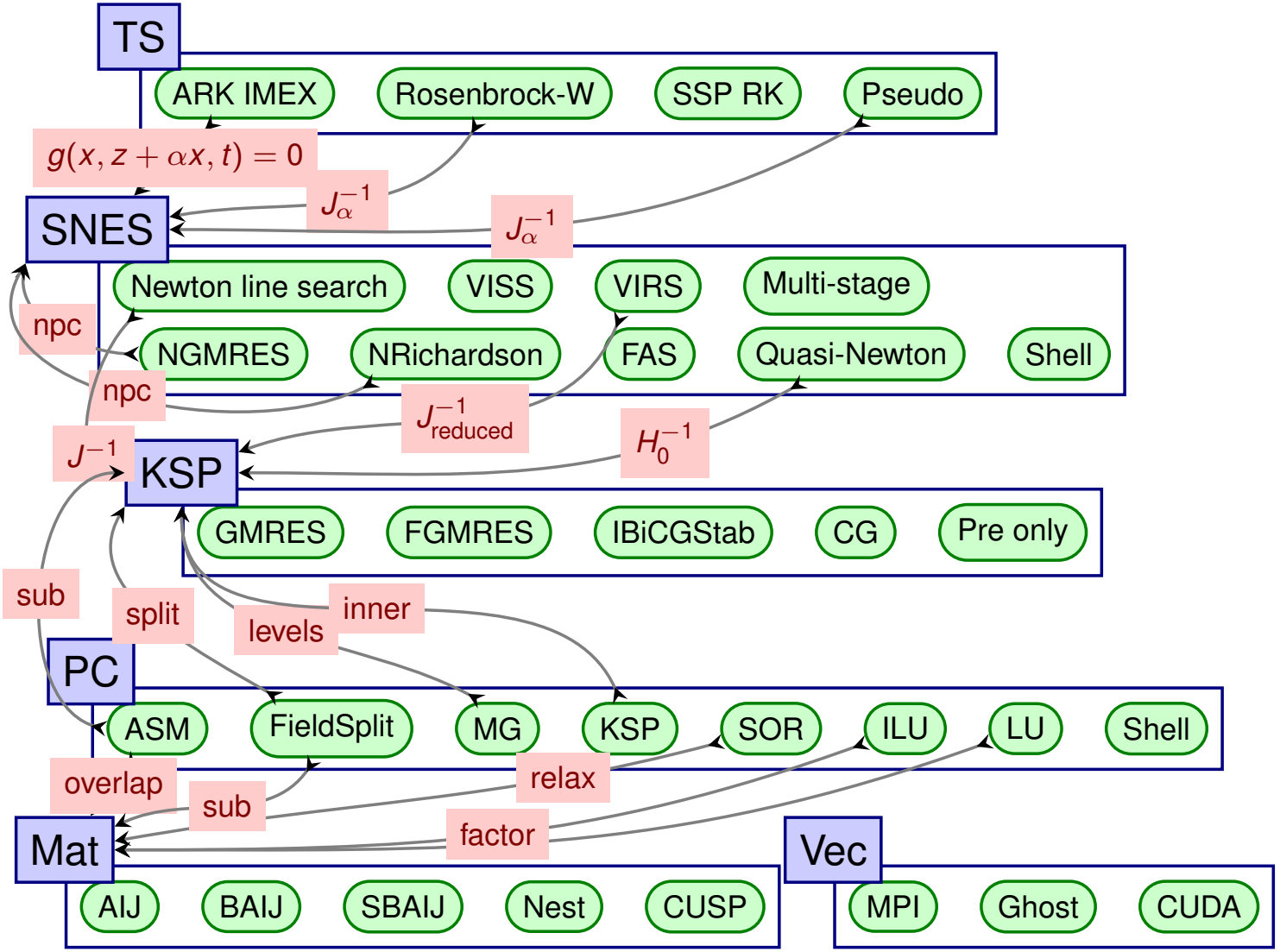
```
MatSetValuesStencil(Mat A, m, MatStencil idxm[], n, MatStencil idxn[],
                    PetscScalar values[], InsertMode mode)
```

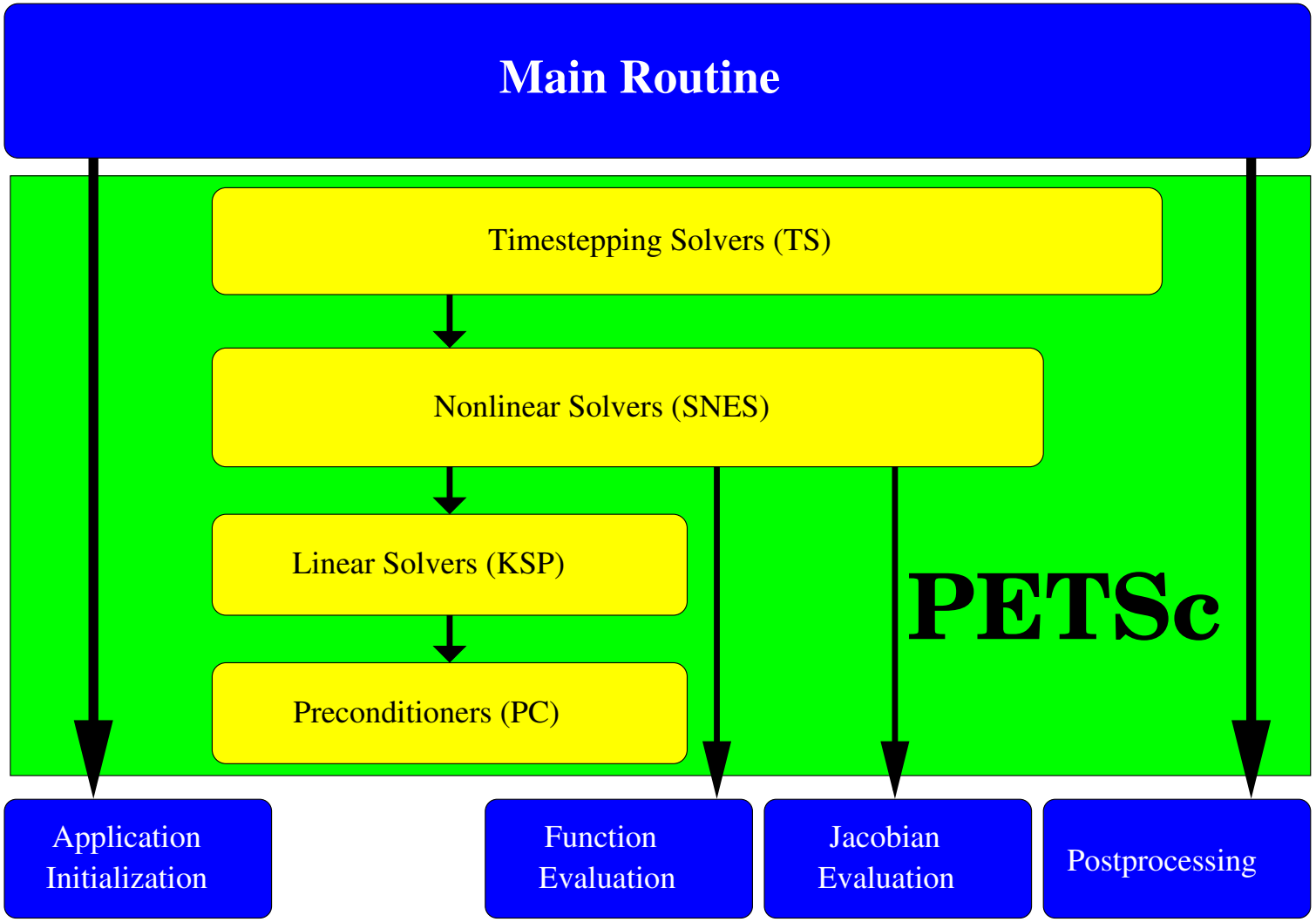
- Each row or column is actually a `MatStencil`
  - This specifies grid coordinates and a component if necessary
  - Can imagine for unstructured grids, they are *vertices*
- The values are the same logically dense block in row/col

DMDA matrices

- DMCreateMatrix(DM da,Mat *A)
- Evaluate only the local portion
  - No nice local array form without copies
- Use `MatSetValuesStencil()` to convert `(i,j,k)` to indices
- `make NP=2 EXTRA_ARGS="-run test -da_grid_x 10 -da_grid_y 10 -mat_view_draw -draw_pause -1" runbratu`
- `make NP=2 EXTRA_ARGS="-run test -dim 3 -da_grid_x 5 -da_grid_y 5 -da_grid_z 5 -mat_view_draw -draw_pause -1" runbratu`

Interactions among composable linear, nonlinear, and timestepping solvers





- IGA used to evaluate nonlinear residuals
- PETSc `DA` used to manage parallelism.
- Adaptive time integration using method of lines.
  - Generalized α method from PETSc `TS`.

Nonlinear solvers in PETSc SNES

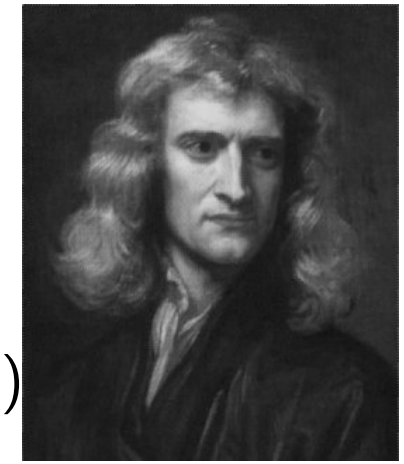
LS, TR	Newton-type with line search and trust region
NRichardson	Nonlinear Richardson, usually preconditioned
VIRS, VIRSAUG, and VISS	reduced space and semi-smooth methods for variational inequalities
QN	Quasi-Newton methods like BFGS
NGMRES	Nonlinear GMRES
NCG	Nonlinear Conjugate Gradients
SORQN	SOR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

Basic Solver Usage

We will illustrate basic solver usage with `SNES`.

- Use `SNESSetFromOptions()` so that everything is set dynamically
  - Use `-snes_type` to set the type or take the default
- Override the tolerances
  - Use `-snes_rtol` and `-snes_atol`
- View the solver to make sure you have the one you expect
  - Use `-snes_view`
- For debugging, monitor the residual decrease
  - Use `-snes_monitor`
  - Use `-ksp_monitor` to see the underlying linear solver

Newton iteration: workhorse of SNES



- Standard form of a nonlinear system

$$F(u) = 0$$

- Iteration

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

- Quadratically convergent near a root:
  $\left|u^{n+1} - u^*\right| \in \mathcal{O}\left(\left|u^n - u^*\right|^2\right)$
- Picard is the same operation with a different $J(u)$

Example (Nonlinear Poisson)

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

SNES Paradigm

The SNES interface is based upon callback functions

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

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

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

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`

SNES Function

The user provided function which calculates the nonlinear residual has signature

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

- `x`: The current solution
- `r`: The residual
- `ctx`: The user context passed to `SNESSetFunction()`
  - Use this to pass application information, e.g. physical constants

SNES Jacobian

The user provided function that calculates the Jacobian has signature

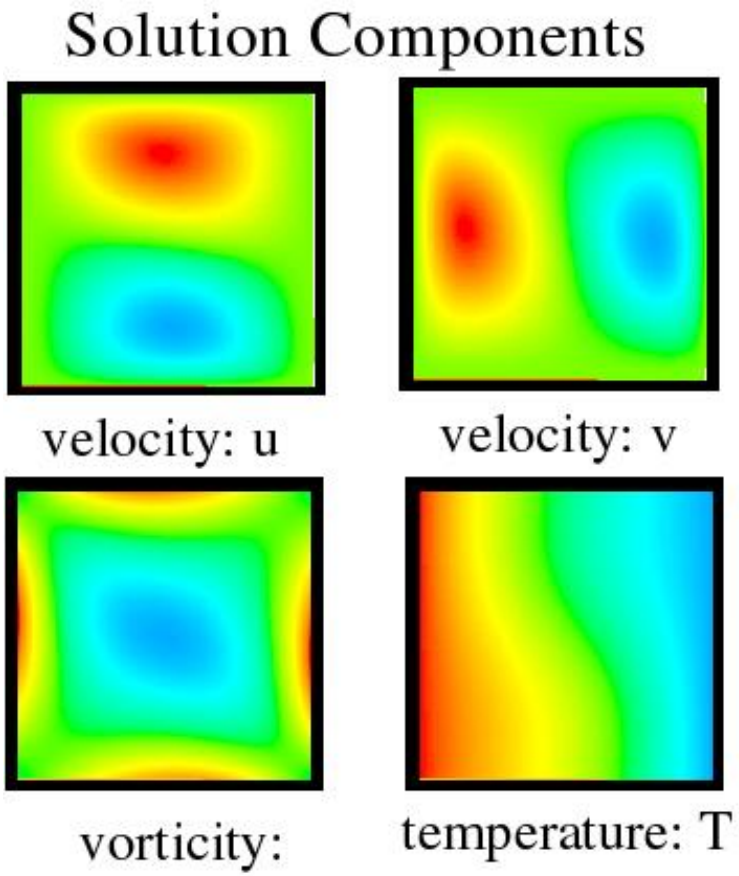
```
PetscErrorCode (*func)(SNES snes,Vec x,Mat J,
                       Mat Jpre,void *ctx)
```

- `x`: The current solution
- `J`: The Jacobian
- `Jpre`: The Jacobian preconditioning matrix (possibly J itself)
- `ctx`: The user context passed to `SNESSetFunction()`
  - Use this to pass application information, e.g. physical constants

Alternatively, you can use

- a builtin sparse finite difference approximation ("coloring")
- automatic differentiation (ADIC/ADIFOR)

SNES Example
Driven Cavity



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

`src/snes/examples/tutorials/ex19.c`

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

```
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 */
  DMDAVecGetArray(da, F, &f);

  /* Loop over local part and assemble into f[idxloc] */
  /* .... */

  DMDAVecRestoreArray(da, F, &f);
  return 0;
}
```

DMDA Local Function

User provided function calculates the nonlinear residual (in 2D)

```
(*lfunc)(DMDALocalInfo *info, PetscScalar **x, PetscScalar **r, void *ctx)
```

`info`: All layout and numbering information

`x`: The current solution

- Notice that it is a multidimensional array

`r`: The residual

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

The local DMDA function is activated by calling

```
SNESSetFunction(snes, r, SNESDAFormFunction, ctx)
```

SNES Example with local evaluation

```
PetscErrorCode DrivenCavityFuncLocal(DMDALocalInfo *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;
      f[j][i].u  = uxx + uyy - .5*(x[j+1][i].omega-x[j-1][i].omega
      /* V velocity, Omega ... */
      /* Temperature */
      u            = x[j][i].temp;
      uxx          = (2.0*u - x[j][i-1].temp - x[j][i+1].temp)*hy
      uyy          = (2.0*u - x[j-1][i].temp - x[j+1][i].temp)*hx
      f[j][i].temp =  uxx + uyy + prandtl
        * (  (vxp*(u - x[j][i-1].temp) + vxm*(x[j][i+1].temp - u))
           + (vyp*(u - x[j-1][i].temp) + vym*(x[j+1][i].temp - u))
```

DMDA Local Jacobian

User provided function calculates the Jacobian (in 2D)

```
(*lfunc)(DMDALocalInfo *info, PetscScalar **x, Mat J, void *ctx)
```

- `info`: All layout and numbering information
- `x`: The current solution
- `J`: The Jacobian
- `ctx`: The user context passed to `DASetLocalJacobian()`

The local DMDA function is activated by calling

```
SNESSetJacobian(snes, J, J, SNESDAComputeJacobian, ctx)
```

Running the driven cavity

- `./ex19 -lidvelocity 100 -grashof 1e2 -da_grid_x 16 -da_grid_y 16 -snes_monitor -snes_view -da_refine 2`
- `./ex19 -lidvelocity 100 -grashof 1e4 -da_grid_x 16 -da_grid_y 16 -snes_monitor -snes_view -da_refine 2`
- `./ex19 -lidvelocity 100 -grashof 1e5 -da_grid_x 16 -da_grid_y 16 -snes_monitor -snes_view -da_refine 2 -pc_type lu`
- Uh oh, we have convergence problems
- Does `-snes_grid_sequence` help?

Running the driven cavity

- ```
  ./ex19 -lidvelocity 100 -grashof 1e2 -da_grid_x 16
  -da_grid_y 16 -snes_monitor -snes_view -da_refine 2
  lid velocity = 100, prandtl # = 1, grashof # = 1000
  0 SNES Function norm 7.682893957872e+02
  1 SNES Function norm 6.574700998832e+02
  2 SNES Function norm 5.285205210713e+02
  3 SNES Function norm 3.770968117421e+02
  4 SNES Function norm 3.030010490879e+02
  5 SNES Function norm 2.655764576535e+00
  6 SNES Function norm 6.208275817215e-03
  7 SNES Function norm 1.191107243692e-07
  Number of SNES iterations = 7
  ```
- `./ex19 -lidvelocity 100 -grashof 1e4 -da_grid_x 16 -da_grid_y 16 -snes_monitor -snes_view -da_refine 2`
- `./ex19 -lidvelocity 100 -grashof 1e5 -da_grid_x 16 -da_grid_y 16 -snes_monitor -snes_view -da_refine 2 -pc_type lu`
- Uh oh, we have convergence problems
- Does `-snes_grid_sequence` help?
  ```

Running the driven cavity

- `./ex19 -lidvelocity 100 -grashof 1e2 -da_grid_x 16 -da_grid_y 16 -snes_monitor -snes_view -da_refine 2`
- `./ex19 -lidvelocity 100 -grashof 1e4 -da_grid_x 16 -da_grid_y 16 -snes_monitor -snes_view -da_refine 2`

```
lid velocity = 100, prandtl # = 1, grashof # = 10000
0 SNES Function norm 7.854040793765e+02
1 SNES Function norm 6.630545177472e+02
2 SNES Function norm 5.195829874590e+02
3 SNES Function norm 3.608696664876e+02
4 SNES Function norm 2.458925075918e+02
5 SNES Function norm 1.811699413098e+00
6 SNES Function norm 4.688284580389e-03
7 SNES Function norm 4.417003604737e-08
Number of SNES iterations = 7
```

- `./ex19 -lidvelocity 100 -grashof 1e5 -da_grid_x 16 -da_grid_y 16 -snes_monitor -snes_view -da_refine 2 -pc_type lu`
- Uh oh, we have convergence problems
- Does `-snes_grid_sequence` help?

Running the driven cavity

- `./ex19 -lidvelocity 100 -grashof 1e2 -da_grid_x 16 -da_grid_y 16 -snes_monitor -snes_view -da_refine 2`
- `./ex19 -lidvelocity 100 -grashof 1e4 -da_grid_x 16 -da_grid_y 16 -snes_monitor -snes_view -da_refine 2`
- `./ex19 -lidvelocity 100 -grashof 1e5 -da_grid_x 16 -da_grid_y 16 -snes_monitor -snes_view -da_refine 2 -pc_type lu`

```
lid velocity = 100, prandtl # = 1, grashof # = 100000
0 SNES Function norm 1.809960438828e+03
1 SNES Function norm 1.678372489097e+03
2 SNES Function norm 1.643759853387e+03
3 SNES Function norm 1.559341161485e+03
4 SNES Function norm 1.557604282019e+03
5 SNES Function norm 1.510711246849e+03
6 SNES Function norm 1.500472491343e+03
7 SNES Function norm 1.498930951680e+03
8 SNES Function norm 1.498440256659e+03
...
```

- Uh oh, we have convergence problems
- Does `-snes_grid_sequence` help?

Running the driven cavity

- `./ex19 -lidvelocity 100 -grashof 1e2 -da_grid_x 16 -da_grid_y 16 -snes_monitor -snes_view -da_refine 2`
- `./ex19 -lidvelocity 100 -grashof 1e4 -da_grid_x 16 -da_grid_y 16 -snes_monitor -snes_view -da_refine 2`
- `./ex19 -lidvelocity 100 -grashof 1e5 -da_grid_x 16 -da_grid_y 16 -snes_monitor -snes_view -da_refine 2 -pc_type lu`
- Uh oh, we have convergence problems
- Does `-snes_grid_sequence` help?

Exercise 5

Run SNES Example 5 using come custom options.

1. `cd $PETSC_DIR/src/snes/examples/tutorials`
2. `make ex5`
3. `mpiexec ./ex5 -snes_monitor -snes_view`
4. `mpiexec ./ex5 -snes_type tr -snes_monitor -snes_view`
5. `mpiexec ./ex5 -ksp_monitor -snes_monitor -snes_view`
6. `mpiexec ./ex5 -pc_type jacobi -ksp_monitor -snes_monitor -snes_view`
7. `mpiexec ./ex5 -ksp_type bicg -ksp_monitor -snes_monitor -snes_view`

Sample output (SNES and KSP)

```
SNES Object: 1 MPI processes
  type: ls
    line search variant: CUBIC
    alpha=1.000000000000e-04, maxstep=1.000000000000e+08, minlambd
    damping factor=1.000000000000e+00
  maximum iterations=50, maximum function evaluations=10000
  tolerances: relative=1e-08, absolute=1e-50, solution=1e-08
  total number of linear solver iterations=5
  total number of function evaluations=6
  KSP Object:   1 MPI processes
    type: gmres
      GMRES: restart=30, using Classical (unmodified) Gram-Schmidt
      GMRES: 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
```

Multiphysics Assembly Code: Residuals

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

Multiphysics Assembly Code: Jacobians

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

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

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

  `-pack_dm_mat_type aij -pc_type lu -pc_factor_mat_solver_package mumps`

Finite Difference Jacobians

PETSc can compute and explicitly store a Jacobian via 1st-order FD

- Dense
  - Activated by `-snes_fd`
  - Computed by `SNESDefaultComputeJacobian()`
- Sparse via colorings
  - Activated by `-snes_fd_color` (default when no Jacobian set and using DM)
  - Coloring is created by `MatFDColoringCreate()`
  - Computed by `SNESDefaultComputeJacobianColor()`

Can also use Matrix-free Newton-Krylov via 1st-order FD

- Activated by `-snes_mf` without preconditioning
- Activated by `-snes_mf_operator` with user-defined preconditioning
  - Uses preconditioning matrix from `SNESSetJacobian()`

SNES Variants

- Line search strategies
- Trust region approaches
- Picard iteration
- Variational inequality approaches

Variational Inequalities

- Supports inequality and box constraints on solution variables.
- Solution methods
  - Semismooth Newton
    - reformulate problem as a non-smooth system, Newton on subdifferential
    - Newton step solves diagonally perturbed systems
  - Active set
    - similar linear algebra to solving PDE
    - solve in reduced space by eliminating constrained variables
    - or enforce constraints by Lagrange multipliers
    - sometimes slower convergence or “bouncing”
- composes with multigrid and field-split
- demonstrated optimality for phase-field problems with millions of degrees of freedom

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 `-snes_linesearch_monitor`
  - Try using trust region instead of line search `-snes_type newtontr`
  - Try grid sequencing if possible
  - Use a continuation

SNES Test

- PETSc can compute a finite difference Jacobian and compare it to yours
- `-snes_type test`
  - Is the difference significant?
- `-snes_type test -snes_test_display`
  - Are the entries in the star stencil correct?
- Find which line has the typo
- `$ git checkout 9-newton-correct`
- Check with `-snes_type test`
- and `-snes_mf_operator -pc_type lu`

Nonlinear solvers in PETSc SNES

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

Taxonomy of implicit solvers

Global linearization: Picard and Newton

- Linear solve "$J(u)w = -F(u)$"
  - (sparse) direct vs. iterative (Krylov) with preconditioning
  - classical relaxation (Jacobi, Gauss-Seidel), incomplete factorization (ILU)
  - domain decomposition and multigrid
- Globalization: "$u_{\text{next}} = u + \alpha w$"
  - Line search, trust region, continuation

Inherently nonlinear methods

- Nonlinear GMRES, Nonlinear CG (can use preconditioning)
- Nonlinear domain decomposition
- Nonlinear multigrid: Full Approximation Scheme (FAS)

Taxonomy of implicit solvers

Global linearization: Picard and Newton

- Linear solve "$J(u)w = -F(u)$"
  - (sparse) direct vs. iterative (Krylov) with preconditioning
  - classical relaxation (Jacobi, Gauss-Seidel), incomplete factorization (ILU)
  - domain decomposition and multigrid
- Globalization: "$u_{\text{next}} = u + \alpha w$"
  - Line search, trust region, continuation

Inherently nonlinear methods

- Nonlinear GMRES, Nonlinear CG (can use preconditioning)
- Nonlinear domain decomposition
- Nonlinear multigrid: Full Approximation Scheme (FAS)

- These methods can be **scalable**.

Taxonomy of implicit solvers

Global linearization: Picard and Newton

- Linear solve "$J(u)w = -F(u)$"
  - (sparse) direct vs. iterative (Krylov) with preconditioning
  - classical relaxation (Jacobi, Gauss-Seidel), incomplete factorization (ILU)
  - domain decomposition and multigrid
- Globalization: "$u_{\text{next}} = u + \alpha w$"
  - Line search, trust region, continuation

Inherently nonlinear methods

- Nonlinear GMRES, Nonlinear CG (can use preconditioning)
- Nonlinear domain decomposition
- Nonlinear multigrid: Full Approximation Scheme (FAS)

- How nonlinear are the scales? How expensive is setup?

Full Approximation Scheme

$$\tilde{u}^h \leftarrow S^h_{\text{pre}} u^h_0 \qquad \text{pre-smooth}$$

$$L^H u^H = I^H_h f^h + \underbrace{L^H \hat{I}^H_h \tilde{u}^h - I^H_h L^h \tilde{u}^h}_{\tau^H_h} \qquad \text{solve coarse problem for } u^H$$

$$u^h \leftarrow S^h_{\text{post}} \left[\tilde{u}^h + I^h_H (u^H - \hat{I}^H_h \tilde{u}^h) \right] \qquad \text{apply correction and post-smooth}$$

- Nonlinearities from spatial discretization fixed locally
- No assembled matrices so better floating point utilization, less memory
- Makes progress on all physical components at once
- FD and DG good, less efficient for continuous finite element methods
- Influence of surface evolution is low rank, no need to visit finest level on each iteration

Nonlinear Multigrid

Most authors just offer an ansatz with nonlinear smoothing

$$x^{new} = S(x^{old}, b) \tag{1}$$

and coarse-grid correction

$$\begin{aligned} F_c(x_c) &= F_c(\tilde{x}_c) + \gamma R(b - F(x^{old})) && (2) \\ x^{new} &= x^{old} + \frac{1}{\gamma} R^T (x_c - \tilde{x}_c) && (3) \end{aligned}$$

where $\tilde{x}$ is an approximate solution.

If F is a linear operator L, the correction reduces to

$$\begin{aligned} L_c(x_c) &= L_c(\tilde{x}_c) + \gamma R(b - L(x^{old})) && (4) \\ L_c(x_c - \tilde{x}_c) &= \gamma R(b - L(x^{old})) && (5) \\ L_c \delta x_c &= \gamma R r && (6) \end{aligned}$$

Nonlinear Multigrid

Most authors just offer an ansatz with nonlinear smoothing

$$x^{new} = S(x^{old}, b) \tag{1}$$

and coarse-grid correction

$$\begin{aligned} F_c(x_c) &= F_c(\tilde{x}_c) + \gamma R(b - F(x^{old})) && (2) \\ x^{new} &= x^{old} + \frac{1}{\gamma} R^T (x_c - \tilde{x}_c) && (3) \end{aligned}$$

where $\tilde{x}$ is an approximate solution.

and the update becomes

$$\begin{aligned} x^{new} &= x^{old} + \frac{1}{\gamma} R^T \delta x_c && (4) \\ x^{new} &= x^{old} + R^T \hat{L}_c^{-1} R r && (5) \end{aligned}$$

Nonlinear Multigrid

It is instructive to look at the alternate derivation of Barry Smith

Begin with the nonlinear generalization $F(u) = 0$, for a correction

$$J_c x_c = R(b - Jx^{old}) \tag{6}$$
$$J_c x_c = -R(F(u) + Jx^{old}) \tag{7}$$

and then using Taylor series

$$F(u^{old}) = F(u) + J(u^{old} - u) + \ldots \tag{8}$$
$$F_c(u_c^{old} + x_c) = F_c(u_c^{old}) + J_c x_c + \ldots \tag{9}$$

we have the correction

$$F_c(u_c^{old} + x_c) - F_c(u_c^{old}) = -RF(u^{old}) \tag{10}$$
$$F_c(u_c^{old} + x_c) = F_c(u_c^{old}) - RF(u^{old}) \tag{11}$$

Nonlinear Multigrid

It is instructive to look at the alternate derivation of Barry Smith

Begin with the nonlinear generalization $F(u) = 0$, for a correction

$$J_c x_c = R(b - Jx^{old}) \tag{6}$$
$$J_c x_c = -R(F(u) + Jx^{old}) \tag{7}$$

and then using Taylor series

$$F(u^{old}) = F(u) + J(u^{old} - u) + \ldots \tag{8}$$
$$F_c(u_c^{old} + x_c) = F_c(u_c^{old}) + J_c x_c + \ldots \tag{9}$$

and the same update

$$x^{new} = x^{old} + R^T x_c \tag{10}$$

Nonlinear multigrid (full approximation scheme)

- V-cycle structure, but use nonlinear relaxation and skip the matrices
- `./ex19 -da_refine 4 -snes_monitor -snes_type nrichardson -npc_fas_levels_snes_type gs -npc_fas_levels_snes_gs_sweeps 3 -npc_snes_type fas -npc_fas_levels_snes_type gs -npc_snes_max_it 1 -npc_snes_fas_smoothup 6 -npc_snes_fas_smoothdown 6 -lidvelocity 100 -grashof 4e4`
- `./ex19 -da_refine 4 -snes_monitor -snes_type ngmres -npc_fas_levels_snes_type gs -npc_fas_levels_snes_gs_sweeps 3 -npc_snes_type fas -npc_fas_levels_snes_type gs -npc_snes_max_it 1 -npc_snes_fas_smoothup 6 -npc_snes_fas_smoothdown 6 -lidvelocity 100 -grashof 4e4`

Nonlinear multigrid (full approximation scheme)

- V-cycle structure, but use nonlinear relaxation and skip the matrices
- `./ex19 -da_refine 4 -snes_monitor -snes_type nrichardson -npc_fas_levels_snes_type gs -npc_fas_levels_snes_gs_sweeps 3 -npc_snes_type fas -npc_fas_levels_snes_type gs -npc_snes_max_it 1 -npc_snes_fas_smoothup 6 -npc_snes_fas_smoothdown 6 -lidvelocity 100 -grashof 4e4`

```
lid velocity = 100, prandtl # = 1, grashof # = 40000
0 SNES Function norm 1.065744184802e+03
1 SNES Function norm 5.213040454436e+02
2 SNES Function norm 6.416412722900e+01
3 SNES Function norm 1.052500804577e+01
4 SNES Function norm 2.520004680363e+00
5 SNES Function norm 1.183548447702e+00
6 SNES Function norm 2.074605179017e-01
7 SNES Function norm 6.782387771395e-02
8 SNES Function norm 1.421602038667e-02
9 SNES Function norm 9.849816743803e-03
10 SNES Function norm 4.168854365044e-03
11 SNES Function norm 4.392925390996e-04
12 SNES Function norm 1.433224993633e-04
13 SNES Function norm 1.074357347213e-04
14 SNES Function norm 6.107933844115e-05
15 SNES Function norm 1.509756087413e-05
16 SNES Function norm 3.478180386598e-06

Number of SNES iterations = 16
```

- `./ex19 -da_refine 4 -snes_monitor -snes_type ngmres -npc_fas_levels_snes_type gs -npc_fas_levels_snes_gs_sweeps 3 -npc_snes_type fas -npc_fas_levels_snes_type gs`

Nonlinear multigrid (full approximation scheme)

- V-cycle structure, but use nonlinear relaxation and skip the matrices
- `./ex19 -da_refine 4 -snes_monitor -snes_type nrichardson -npc_fas_levels_snes_type gs -npc_fas_levels_snes_gs_sweeps 3 -npc_snes_type fas -npc_fas_levels_snes_type gs -npc_snes_max_it 1 -npc_snes_fas_smoothup 6 -npc_snes_fas_smoothdown 6 -lidvelocity 100 -grashof 4e4`
- `./ex19 -da_refine 4 -snes_monitor -snes_type ngmres -npc_fas_levels_snes_type gs -npc_fas_levels_snes_gs_sweeps 3 -npc_snes_type fas -npc_fas_levels_snes_type gs -npc_snes_max_it 1 -npc_snes_fas_smoothup 6 -npc_snes_fas_smoothdown 6 -lidvelocity 100 -grashof 4e4`

```
lid velocity = 100, prandtl # = 1, grashof # = 40000
0 SNES Function norm 1.065744184802e+03
1 SNES Function norm 9.413549877567e+01
2 SNES Function norm 2.117533223215e+01
3 SNES Function norm 5.858983768704e+00
4 SNES Function norm 7.303010571089e-01
5 SNES Function norm 1.585498982242e-01
6 SNES Function norm 2.963278257962e-02
7 SNES Function norm 1.152790487670e-02
8 SNES Function norm 2.092161787185e-03
9 SNES Function norm 3.129419807458e-04
10 SNES Function norm 3.503421154426e-05
11 SNES Function norm 2.898344063176e-06

Number of SNES iterations = 11
```

Monolithic approaches

Parallel direct solver

```
-dm_mat_type aij -pc_type lu -pc_factor_mat_solver_package m
```

Coupled nonlinear multigrid accelerated by NGMRES with multi-stage smoothers

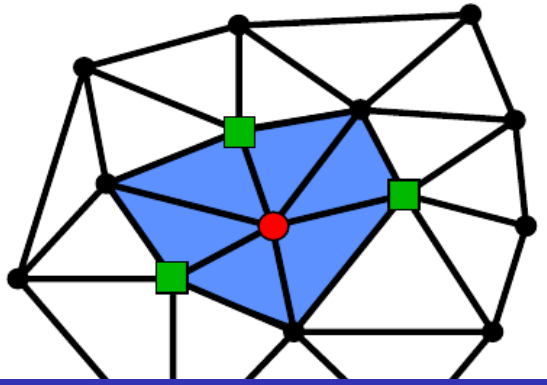
```
-lidvelocity 200 -grashof 1e4
-snes_grid_sequence 5 -snes_monitor -snes_view
-snes_type ngmres
-npc_snes_type fas
-npc_snes_max_it 1
-npc_fas_coarse_snes_type ls
-npc_fas_coarse_ksp_type preonly
-npc_fas_snes_type ms
-npc_fas_snes_max_it 1
-npc_fas_ksp_type preonly
-npc_fas_pc_type pbjacobi
-npc_fas_snes_ms_type m62
-npc_fas_snes_max_it 1
```

Nonlinear and matrix-free smoothing

- matrix-based smoothers require global linearization
- nonlinearity often more efficiently resolved locally
- nonlinear additive or multiplicative Schwarz
- nonlinear/matrix-free is good if

$$C = \frac{(\text{cost to evaluate residual at one “point”}) \cdot N}{(\text{cost of global residual})} \sim 1$$

  - finite difference: $C < 2$
  - finite volume: $C \sim 2$, depends on reconstruction
  - finite element: $C \sim$ number of vertices per cell
- larger block smoothers help reduce C
- additive correction (Jacobi/Chebyshev/multi-stage)
  - global evaluation, as good as $C = 1$
  - but, need to assemble corrector/scaling
  - need spectral estimates or wave speeds



Conclusions

Newton-Multigrid provides

- Good nonlinear solves
- Simple interface for software libraries
- Low computational efficiency

Multigrid-FAS provides

- Good nonlinear solves
- Lower memory bandwidth and storage
- Potentially high computational efficiency
- Requires formation on small systems “on the fly”

Overwhelmed with choices

- If you have a hard problem, no black-box solver will work well
- Everything in PETSc has a plugin architecture
  - Put in the “special sauce” for your problem
  - Your implementations are first-class
- PETSc exposes an algebra of composition at runtime
  - Build a good solver from existing components, at runtime
  - Multigrid, domain decomposition, factorization, relaxation, field-split
  - Choose matrix format that works best with your preconditioner
  - structural blocking, Neumann matrices, monolithic versus nested

Questions to ask when you see a matrix

1. What do you want to do with it?
   - Multiply with a vector
   - Solve linear systems or eigen-problems
2. How is the conditioning/spectrum?
   - distinct/clustered eigen/singular values?
   - symmetric positive definite ($\sigma(A) \subset \mathbb{R}^+$)?
   - nonsymmetric definite ($\sigma(A) \subset \{z \in \mathbb{C} : \Re[z] > 0\}$)?
   - indefinite?
3. How dense is it?
   - block/banded diagonal?
   - sparse unstructured?
   - denser than we'd like?
4. Is there a better way to compute Ax?
5. Is there a different matrix with similar spectrum, but nicer properties?
6. How can we precondition A?

Questions to ask when you see a matrix

1. What do you want to do with it?
   - Multiply with a vector
   - Solve linear systems or eigen-problems
2. How is the conditioning/spectrum?
   - distinct/clustered eigen/singular values?
   - symmetric positive definite ($\sigma(A) \subset \mathbb{R}^+$)?
   - nonsymmetric definite ($\sigma(A) \subset \{z \in \mathbb{C} : \Re[z] > 0\}$)?
   - indefinite?
3. How dense is it?
   - block/banded diagonal?
   - sparse unstructured?
   - denser than we'd like?
4. Is there a better way to compute Ax?
5. Is there a different matrix with similar spectrum, but nicer properties?
6. How can we precondition A?

Preconditioning

Definition (Preconditioner)

A preconditioner $\mathcal{P}$ is a method for constructing a matrix $P^{-1} = \mathcal{P}(A, A_p)$ using a matrix A and extra information A_p, such that the spectrum of $P^{-1}A$ (or AP^{-1}) is well-behaved.

- P^{-1} is dense, P is often not available and is not needed
- A is rarely used by $\mathcal{P}$, but $A_p = A$ is common
- A_p is often a sparse matrix, the "preconditioning matrix"
- Matrix-based: Jacobi, Gauss-Seidel, SOR, ILU(k), LU
- Parallel: Block-Jacobi, Schwarz, Multigrid, FETI-DP, BDDC
- Indefinite: Schur-complement, Domain Decomposition, Multigrid

Preconditioning

Idea: improve the conditioning of the Krylov operator

- Left preconditioning

$$(P^{-1}A)x = P^{-1}b$$

$$\{P^{-1}b, (P^{-1}A)P^{-1}b, (P^{-1}A)^2P^{-1}b, \dots\}$$

- Right preconditioning

$$(AP^{-1})Px = b$$

$$\{b, (P^{-1}A)b, (P^{-1}A)^2b, \dots\}$$

- The product $P^{-1}A$ or AP^{-1} is <u>not</u> formed.

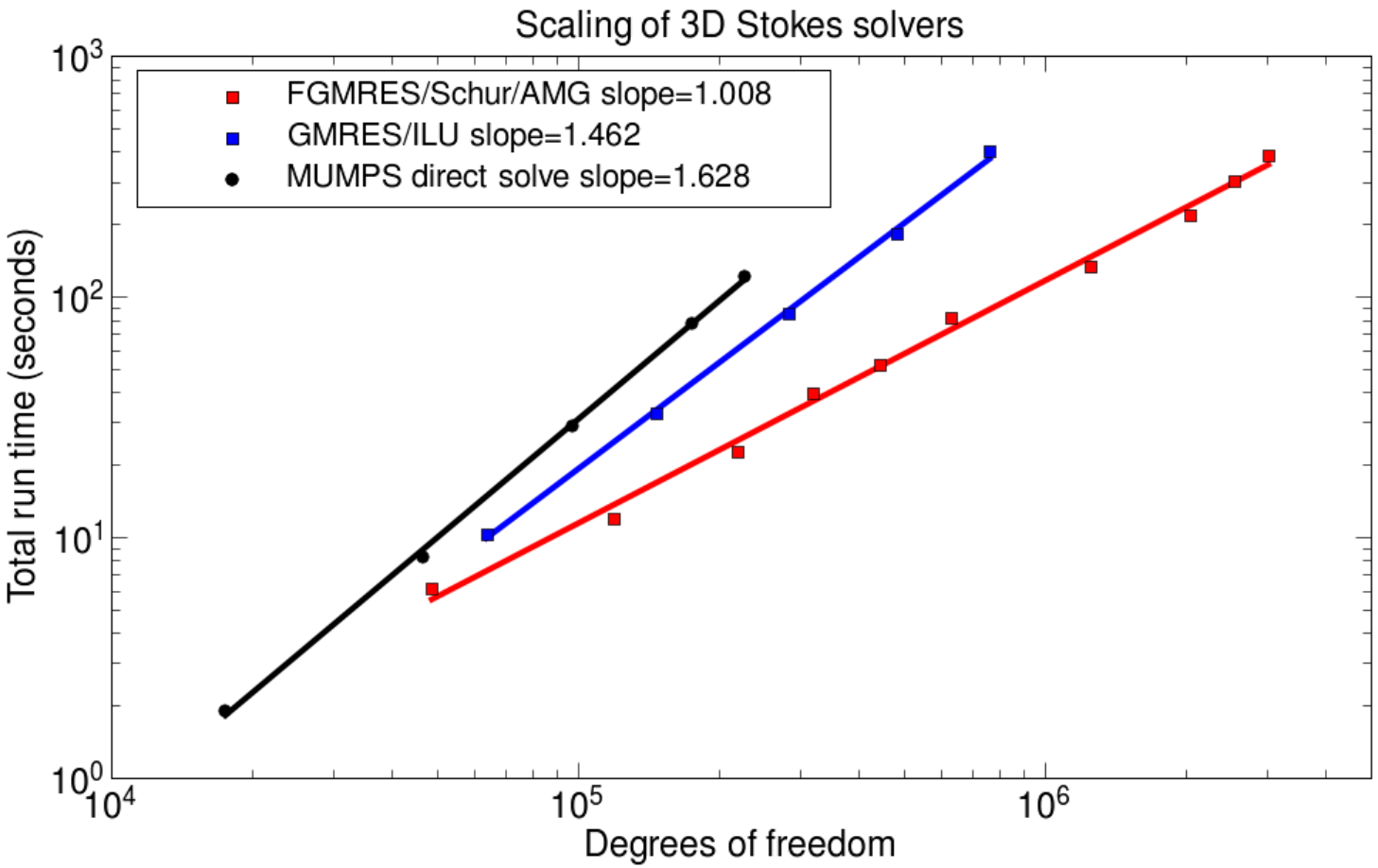
Definition (Preconditioner)

A <u>preconditioner</u> $\mathcal{P}$ is a method for constructing a matrix (just a linear function, not assembled!) $P^{-1} = \mathcal{P}(A, A_p)$ using a matrix A and extra information A_p, such that the spectrum of $P^{-1}A$ (or AP^{-1}) is well behaved.

Linear Solvers

- Use a direct method (small problem size)
- Precondition with Schur Complement method
- Use multigrid approach

What about direct linear solvers?



- By all means, start with a direct solver
- Direct solvers are robust, but not scalable
- **2D**: $\mathcal{O}(n^{1.5})$ flops, $\mathcal{O}(n \log n)$ memory.
- **3D**: $\mathcal{O}(n^2)$ flops, $\mathcal{O}(n^{4/3})$ memory

3rd Party Solvers in PETSc

Complete table of solvers

1. Sequential LU
   - ILUDT (SPARSEKIT2, Yousef Saad, U of MN)
   - EUCLID & PILUT (Hypre, David Hysom, LLNL)
   - ESSL (IBM)
   - SuperLU (Jim Demmel and Sherry Li, LBNL)
   - Matlab
   - UMFPACK (Tim Davis, U. of Florida)
   - LUSOL (MINOS, Michael Saunders, Stanford)
2. Parallel LU
   - MUMPS (Patrick Amestoy, IRIT)
   - SPOOLES (Cleve Ashcroft, Boeing)
   - SuperLU_Dist (Jim Demmel and Sherry Li, LBNL)
3. Parallel Cholesky
   - DSCPACK (Padma Raghavan, Penn. State)
4. XYTlib - parallel direct solver (Paul Fischer and Henry Tufo, ANL)

3rd Party Preconditioners in PETSc

Complete table of solvers

1. Parallel ICC
   - BlockSolve95 (Mark Jones and Paul Plassman, ANL)
2. Parallel ILU
   - BlockSolve95 (Mark Jones and Paul Plassman, ANL)
3. Parallel Sparse Approximate Inverse
   - Parasails (Hypre, Edmund Chow, LLNL)
   - SPAI 3.0 (Marcus Grote and Barnard, NYU)
4. Sequential Algebraic Multigrid
   - RAMG (John Ruge and Klaus Steuben, GMD)
   - SAMG (Klaus Steuben, GMD)
5. Parallel Algebraic Multigrid
   - Prometheus (Mark Adams, PPPL)
   - BoomerAMG (Hypre, LLNL)
   - ML (Trilinos, Ray Tuminaro and Jonathan Hu, SNL)

The Great Solver Schism: Monolithic or Split?

Monolithic

- Direct solvers
- Coupled Schwarz
- Coupled Neumann-Neumann (need unassembled matrices)
- Coupled multigrid

X Need to understand local spectral and compatibility properties of the coupled system

Split

- Physics-split Schwarz (based on relaxation)
- Physics-split Schur (based on factorization)
  - approximate commutators SIMPLE, PCD, LSC
  - segregated smoothers
  - Augmented Lagrangian
  - "parabolization" for stiff waves

X Need to understand global coupling strengths

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

Outlook on Solver Composition

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

In development

- Distributive relaxation, Vanka smoothers
- Algebraic coarsening of “dual” variables
- Improving operator-dependent semi-geometric multigrid
- More automatic spectral analysis and smoother optimization
- Automated support for mixing analysis into levels

Stokes example

The common block preconditioners for Stokes require only options:

The Stokes System

```
-pc_type fieldsplit
-pc_fieldsplit_type
```

$$\begin{pmatrix} A & B \\ B^T & 0 \end{pmatrix}$$

```
-fieldsplit_0_ksp_type preonly
```

Stokes example

The common block preconditioners for Stokes require only options:

```
-pc_type fieldsplit
-pc_fieldsplit_type additive

-fieldsplit_0_pc_type ml
-fieldsplit_0_ksp_type preonly

-fieldsplit_1_pc_type jacobi
-fieldsplit_1_ksp_type preonly
```

$$\text{PC} \begin{pmatrix} \hat{A} & 0 \\ 0 & I \end{pmatrix}$$

Cohouet and Chabard, Some fast 3D finite element solvers for the generalized Stokes problem, 1988.

Stokes example

The common block preconditioners for Stokes require only options:

```
-pc_type fieldsplit
-pc_fieldsplit_type
multiplicative

-fieldsplit_0_pc_type hypre
-fieldsplit_0_ksp_type preonly

-fieldsplit_1_pc_type jacobi
-fieldsplit_1_ksp_type preonly
```

$$\text{PC} \begin{pmatrix} \hat{A} & B \\ 0 & I \end{pmatrix}$$

Elman, Multigrid and Krylov subspace methods for the discrete Stokes equations, 1994.

Stokes example

The common block preconditioners for Stokes require only options:

```
-pc_type fieldsplit
-pc_fieldsplit_type schur

-fieldsplit_0_pc_type gamg
-fieldsplit_0_ksp_type preonly

-fieldsplit_1_pc_type none
-fieldsplit_1_ksp_type minres

-pc_fieldsplit_schur_factorization_type diag
```

$$\text{PC} \begin{pmatrix} \hat{A} & 0 \\ 0 & -\hat{S} \end{pmatrix}$$

May and Moresi, Preconditioned iterative methods for Stokes flow problems arising in computational geodynamics, 2008.

Olshanskii, Peters, and Reusken, Uniform preconditioners for a parameter dependent saddle point problem with application to generalized Stokes interface equations, 2006.

Stokes example

The common block preconditioners for Stokes require only options:

```
-pc_type fieldsplit
-pc_fieldsplit_type schur

-fieldsplit_0_pc_type gamg
-fieldsplit_0_ksp_type preonly

-fieldsplit_1_pc_type none
-fieldsplit_1_ksp_type minres

-pc_fieldsplit_schur_factorization_type lower
```

$$\text{PC} \begin{pmatrix} \hat{A} & 0 \\ B^T & \hat{S} \end{pmatrix}$$

May and Moresi, Preconditioned iterative methods for Stokes flow problems arising in computational geodynamics, 2008.

Stokes example

The common block preconditioners for Stokes require only options:

```
-pc_type fieldsplit
-pc_fieldsplit_type schur

-fieldsplit_0_pc_type gamg
-fieldsplit_0_ksp_type preonly

-fieldsplit_1_pc_type none
-fieldsplit_1_ksp_type minres

-pc_fieldsplit_schur_factorization_type upper
```

$$\text{PC} \begin{pmatrix} \hat{A} & B \\ 0 & \hat{S} \end{pmatrix}$$

May and Moresi, Preconditioned iterative methods for Stokes flow problems arising in computational geodynamics, 2008.

Stokes example

The common block preconditioners for Stokes require only options:

```
-pc_type fieldsplit
-pc_fieldsplit_type schur

-fieldsplit_0_pc_type gamg
-fieldsplit_0_ksp_type preonly

-fieldsplit_1_pc_type lsc
-fieldsplit_1_ksp_type minres

-pc_fieldsplit_schur_factorization_type upper
```

$$\text{PC} \begin{pmatrix} \hat{A} & B \\ 0 & \hat{S}_{\text{LSC}} \end{pmatrix}$$

May and Moresi, Preconditioned iterative methods for Stokes flow problems arising in computational geodynamics, 2008.

Kay, Loghin and Wathen, A Preconditioner for the Steady-State N-S Equations, 2002.
Elman, Howle, Shadid, Shuttleworth, and Tuminaro, Block preconditioners based on approximate commutators, 2006.

Stokes example

The common block preconditioners for Stokes require only options:

```
-pc_type fieldsplit
-pc_fieldsplit_type schur
-pc_fieldsplit_schur_factorization_type full
```

$$\text{PC}$$

$$\begin{pmatrix} I & 0 \\ B^T A^{-1} & I \end{pmatrix} \begin{pmatrix} \hat{A} & 0 \\ 0 & \hat{S} \end{pmatrix} \begin{pmatrix} I & A^{-1}B \\ 0 & I \end{pmatrix}$$

Stokes example

All block preconditioners can be *embedded* in MG using only options:

```
-pc_type mg -pc_mg_levels 5 -pc_mg_galerkin
-mg_levels_pc_type fieldsplit
-mg_levels_pc_fieldsplit_type
```

System on each Coarse Level

$$R \begin{pmatrix} A & B \\ B^T & 0 \end{pmatrix} P$$

Stokes example

All block preconditioners can be *embedded* in MG using only options:

```
-pc_type mg -pc_mg_levels 5 -pc_mg_galerkin
-mg_levels_pc_type fieldsplit
-mg_levels_pc_fieldsplit_type additive

-mg_levels_fieldsplit_0_pc_type sor
-mg_levels_fieldsplit_0_ksp_type preonly

-mg_levels_fieldsplit_1_pc_type jacobi
-mg_levels_fieldsplit_1_ksp_type preonly
```

Smoother PC

$$\begin{pmatrix} \hat{A} & 0 \\ 0 & I \end{pmatrix}$$

Stokes example

All block preconditioners can be *embedded* in MG using only options:

```
-pc_type mg -pc_mg_levels 5 -pc_mg_galerkin
-mg_levels_pc_type fieldsplit
-mg_levels_pc_fieldsplit_type
multiplicative

-mg_levels_fieldsplit_0_pc_type sor
-mg_levels_fieldsplit_0_ksp_type preonly

-mg_levels_fieldsplit_1_pc_type jacobi
-mg_levels_fieldsplit_1_ksp_type preonly
```

Smoother PC

$$\begin{pmatrix} \hat{A} & B \\ 0 & I \end{pmatrix}$$

Stokes example

All block preconditioners can be *embedded* in MG using only options:

```
-pc_type mg -pc_mg_levels 5 -pc_mg_galerkin
-mg_levels_pc_type fieldsplit
-mg_levels_pc_fieldsplit_type schur

-mg_levels_fieldsplit_0_pc_type sor
-mg_levels_fieldsplit_0_ksp_type preonly

-mg_levels_fieldsplit_1_pc_type none
-mg_levels_fieldsplit_1_ksp_type minres
```

Smoother PC

$$\begin{pmatrix} \hat{A} & 0 \\ 0 & -\hat{S} \end{pmatrix}$$

```
-mg_levels_pc_fieldsplit_schur_factorization_type diag
```

Stokes example

All block preconditioners can be *embedded* in MG using only options:

```
-pc_type mg -pc_mg_levels 5 -pc_mg_galerkin
-mg_levels_pc_type fieldsplit
-mg_levels_pc_fieldsplit_type schur

-mg_levels_fieldsplit_0_pc_type sor
-mg_levels_fieldsplit_0_ksp_type preonly

-mg_levels_fieldsplit_1_pc_type none
-mg_levels_fieldsplit_1_ksp_type minres
```

Smoother PC

$$\begin{pmatrix} \hat{A} & 0 \\ B^T & \hat{S} \end{pmatrix}$$

```
-mg_levels_pc_fieldsplit_schur_factorization_type lower
```

Stokes example

All block preconditioners can be *embedded* in MG using only options:

```
-pc_type mg -pc_mg_levels 5 -pc_mg_galerkin
-mg_levels_pc_type fieldsplit
-mg_levels_pc_fieldsplit_type schur

-mg_levels_fieldsplit_0_pc_type sor
-mg_levels_fieldsplit_0_ksp_type preonly

-mg_levels_fieldsplit_1_pc_type none
-mg_levels_fieldsplit_1_ksp_type minres
```

Smoother PC

$$\begin{pmatrix} \hat{A} & B \\ 0 & \hat{S} \end{pmatrix}$$

```
-mg_levels_pc_fieldsplit_schur_factorization_type upper
```

Stokes example

All block preconditioners can be *embedded* in MG using only options:

```
-pc_type mg -pc_mg_levels 5 -pc_mg_galerkin
-mg_levels_pc_type fieldsplit
-mg_levels_pc_fieldsplit_type schur

-mg_levels_fieldsplit_0_pc_type sor
-mg_levels_fieldsplit_0_ksp_type preonly

-mg_levels_fieldsplit_1_pc_type lsc
-mg_levels_fieldsplit_1_ksp_type minres
```

Smoother PC

$$\begin{pmatrix} \hat{A} & B \\ 0 & \hat{S}_{\mathrm{LSC}} \end{pmatrix}$$

```
-mg_levels_pc_fieldsplit_schur_factorization_type upper
```

Programming with Options

ex55: Allen-Cahn problem in 2D

Smoother: Flexible GMRES (2 iterates) with a Schur complement PC

```
-mg_levels_ksp_type fgmres -mg_levels_pc_fieldsplit_detect_saddle_point
-mg_levels_ksp_max_it 2 -mg_levels_pc_type fieldsplit
-mg_levels_pc_fieldsplit_type schur
-mg_levels_pc_fieldsplit_factorization_type full
-mg_levels_pc_fieldsplit_schur_precondition diag
```

Schur complement solver: GMRES (5 iterates) with no preconditioner

```
-mg_levels_fieldsplit_1_ksp_type gmres
-mg_levels_fieldsplit_1_pc_type none -mg_levels_fieldsplit_ksp_max_it 5
```

Shur complement action: Use only the lower diagonal part of A00

```
-mg_levels_fieldsplit_0_ksp_type preonly
-mg_levels_fieldsplit_0_pc_type sor
-mg_levels_fieldsplit_0_pc_sor_forward
```

Programming with Options

ex55: Allen-Cahn problem in 2D

Smoother: Flexible GMRES (2 iterates) with a Schur complement PC

```
-mg_levels_ksp_type fgmres -mg_levels_pc_fieldsplit_detect_saddle_point
-mg_levels_ksp_max_it 2 -mg_levels_pc_type fieldsplit
-mg_levels_pc_fieldsplit_type schur
-mg_levels_pc_fieldsplit_factorization_type full
-mg_levels_pc_fieldsplit_schur_precondition diag
```

Schur complement solver: GMRES (5 iterates) with no preconditioner

```
-mg_levels_fieldsplit_1_ksp_type gmres
-mg_levels_fieldsplit_1_pc_type none -mg_levels_fieldsplit_ksp_max_it 5
```

Shur complement action: Use only the lower diagonal part of A00

```
-mg_levels_fieldsplit_0_ksp_type preonly
-mg_levels_fieldsplit_0_pc_type sor
-mg_levels_fieldsplit_0_pc_sor_forward
```

Programming with Options

ex55: Allen-Cahn problem in 2D

Smoother: Flexible GMRES (2 iterates) with a Schur complement PC

```
-mg_levels_ksp_type fgmres -mg_levels_pc_fieldsplit_detect_saddle_point
-mg_levels_ksp_max_it 2 -mg_levels_pc_type fieldsplit
-mg_levels_pc_fieldsplit_type schur
-mg_levels_pc_fieldsplit_factorization_type full
-mg_levels_pc_fieldsplit_schur_precondition diag
```

Schur complement solver: GMRES (5 iterates) with no preconditioner

```
-mg_levels_fieldsplit_1_ksp_type gmres
-mg_levels_fieldsplit_1_pc_type none -mg_levels_fieldsplit_ksp_max_it 5
```

Shur complement action: Use only the lower diagonal part of A00

```
-mg_levels_fieldsplit_0_ksp_type preonly
-mg_levels_fieldsplit_0_pc_type sor
-mg_levels_fieldsplit_0_pc_sor_forward
```

Programming with Options

ex55: Allen-Cahn problem in 2D

Smoother: Flexible GMRES (2 iterates) with a Schur complement PC

```
-mg_levels_ksp_type fgmres -mg_levels_pc_fieldsplit_detect_saddle_point
-mg_levels_ksp_max_it 2 -mg_levels_pc_type fieldsplit
-mg_levels_pc_fieldsplit_type schur
-mg_levels_pc_fieldsplit_factorization_type full
-mg_levels_pc_fieldsplit_schur_precondition diag
```

Schur complement solver: GMRES (5 iterates) with no preconditioner

```
-mg_levels_fieldsplit_1_ksp_type gmres
-mg_levels_fieldsplit_1_pc_type none -mg_levels_fieldsplit_ksp_max_it 5
```

Shur complement action: Use only the lower diagonal part of A00

```
-mg_levels_fieldsplit_0_ksp_type preonly
-mg_levels_fieldsplit_0_pc_type sor
-mg_levels_fieldsplit_0_pc_sor_forward
```

Relative effect of the blocks

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

J_{uu} Viscous/momentum terms, nearly symmetric, variable coefficionts, anisotropy from Newton.

J_{up} Weak pressure gradient, viscosity dependence on pressure (small), gravitational contribution (pressure-induced density variation). Large, nearly balanced by gravitational forcing.

J_{uE} Viscous dependence on energy, very nonlinear, not very large.

J_{pu} Divergence (mass conservation), nearly equal to J_{up}^T.

J_{Eu} Sensitivity of energy on momentum, mostly advective transport. Large in boundary layers with large thermal/moisture gradients.

J_{Ep} Thermal/moisture diffusion due to pressure-melting, $\boldsymbol{u} \cdot \nabla$.

J_{EE} Advection-diffusion for energy, very nonlinear at small regularization. Advection-dominated except in boundary layers

How much nesting?

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

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

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

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

- Low-order preconditioning full-accuracy unassembled high order operator.

Why do we need multilevel solvers?

- Elliptic problems are globally coupled
- Without a coarse level, number of iterations proportional to inverse mesh size
- High-volume local communication is an inefficient way to communicate long-range information, bad for parallel models
- Most important with 3D flow features and/or slippery beds
- Nested/split multilevel methods
  - Decompose problem into simpler sub-problems, use multilevel methods on each
  - Good reuse of existing software
  - More synchronization due to nesting, more suitable after linearization
- Monolithic/coupled multilevel methods
  - Better convergence and lower synchronization, but harder to get right
  - Internal nonlinearities resolved locally
  - More discretization-specific, less software reuse

Multigrid

Multigrid is <u>optimal</u> in that is does $\mathcal{O}(N)$ work for $||r|| < \epsilon$

- Brandt, Briggs, Chan & Smith
- Constant work per level
  - Sufficiently strong solver
  - Need a constant factor decrease in the residual
- Constant factor decrease in dof
  - Log number of levels

Multilevel Solvers are a Way of Life

- ingredients that discretizations can provide
  - identify “fields”
  - topological coarsening, possibly for fields
  - near-null space information
  - “natural” subdomains
  - subdomain integration, face integration
  - element or subdomain assembly/matrix-free smoothing
- solver composition
  - most splitting methods accessible from command line
  - energy optimization for tentative coarse basis functions
  - algebraic form of distributive relaxation
  - generic assembly for large systems and components
  - working on flexibile “library-assisted” nonlinear multigrid
  - adding support for interactive eigenanalysis

Linear Multigrid

Smoothing (typically Gauss-Seidel)

$$x^{new} = S(x^{old}, b) \tag{11}$$

Coarse-grid Correction

$$\begin{aligned} J_c \delta x_c &= R(b - Jx^{old}) && (12) \\ x^{new} &= x^{old} + R^T \delta x_c && (13) \end{aligned}$$

Multigrid

Hierarchy: Interpolation and restriction operators

$$\mathcal{I}^{\uparrow} : X_{\text{coarse}} \to X_{\text{fine}} \qquad \mathcal{I}^{\downarrow} : X_{\text{fine}} \to X_{\text{coarse}}$$

- Geometric: define problem on multiple levels, use grid to compute hierarchy
- Algebraic: define problem only on finest level, use matrix structure to build hierarchy

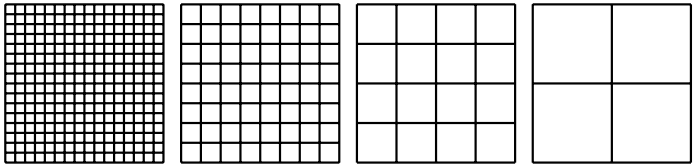
Galerkin approximation

Assemble this matrix: $A_{\text{coarse}} = \mathcal{I}^{\downarrow} A_{\text{fine}} \mathcal{I}^{\uparrow}$

Application of multigrid preconditioner (V-cycle)

- Apply pre-smoother on fine level (any preconditioner)
- Restrict residual to coarse level with $\mathcal{I}^{\downarrow}$
- Solve on coarse level $A_{\text{coarse}} x = r$
- Interpolate result back to fine level with $\mathcal{I}^{\uparrow}$
- Apply post-smoother on fine level (any preconditioner)

Multigrid Preliminaries



Multigrid is an $O(n)$ method for solving algebraic problems by defining a hierarchy of scale. A multigrid method is constructed from:

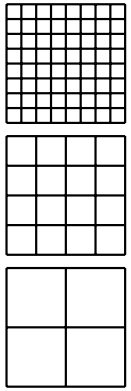
1. a series of discretizations
   - coarser approximations of the original problem
   - constructed algebraically or geometrically
2. intergrid transfer operators
   - residual restriction I_h^H (fine to coarse)
   - state restriction $\hat{I}_h^H$ (fine to coarse)
   - partial state interpolation I_H^h (coarse to fine, 'prolongation')
   - state reconstruction $\mathbb{I}_H^h$ (coarse to fine)
3. Smoothers (S)
   - correct the high frequency error components
   - Richardson, Jacobi, Gauss-Seidel, etc.
   - Gauss-Seidel-Newton or optimization methods

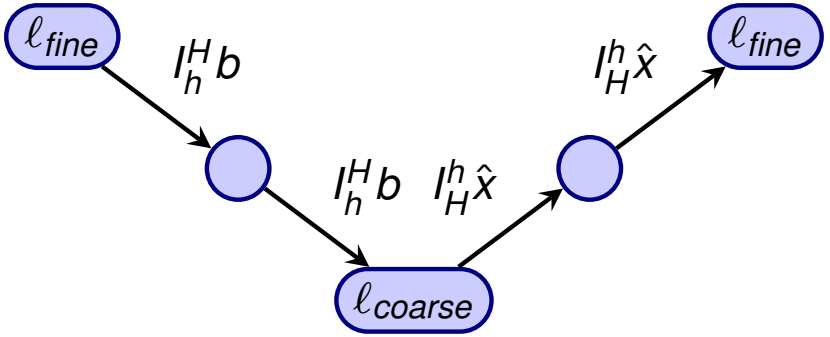
Rediscretized Multigrid using `DM`

- `DM` manages problem data beyond purely algebraic objects
  - structured, redundant, and (less mature) unstructured implementations in PETSc
  - third-party implementations
- `DMCoarsen(dmfine,coarse_comm,&coarsedm)` to create "geometric" coarse level
  - Also `DMRefine()` for grid sequencing and convenience
  - `DMCoarsenHookAdd()` for external clients to move resolution-dependent data for rediscretization and FAS
- `DMCreateInterpolation(dmcoarse,dmfine,&Interp,&Rscale)`
  - Usually uses geometric information, can be operator-dependent
  - Can be improved subsequently, e.g. using energy-minimization from AMG
- Resolution-dependent solver-specific callbacks use attribute caching on `DM`.
  - Managed by solvers, not visible to users unless they need exotic things (e.g. custom homogenization, reduced models)

Multigrid

- **Multigrid** methods uses coarse correction for large-scale error

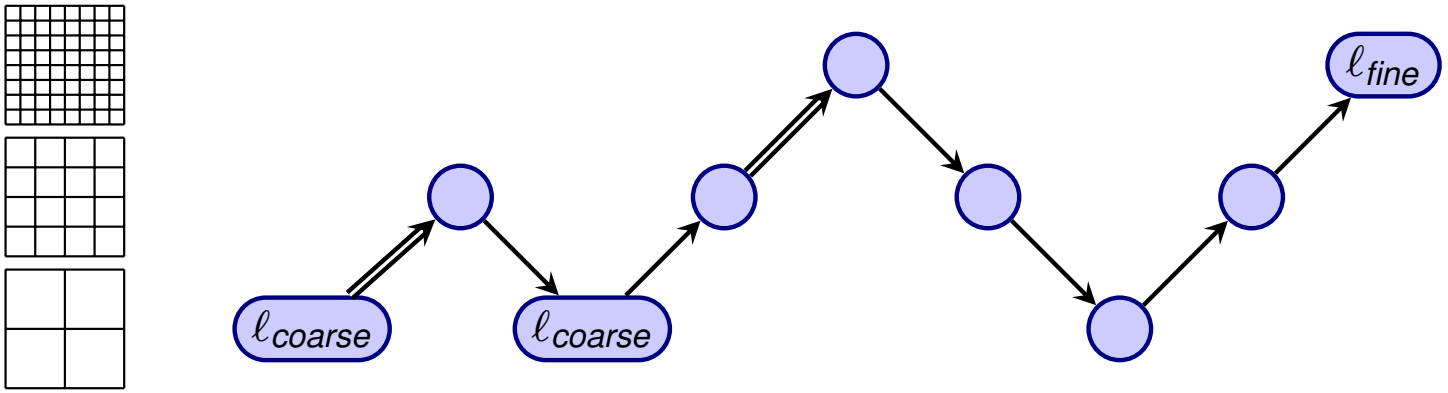




Algorithm $MG(A, b)$ for the solution of $A\vec{x} = b$:

$\vec{x} = S^m(\vec{x}, b)$	pre-smooth
$b^H = I_h^H(\vec{r} - A\vec{x})$	restrict residual
$\hat{x}^H = MG(I_h^H A I_H^h, b^H)$	recurse
$\vec{x} = \vec{x} + I_H^h \hat{x}^H$	prolong correction
$\vec{x} = \vec{x} + S^n(\vec{x}, b)$	post-smooth

Full Multigrid(FMG)



- start wich coarse grid
- $\vec{x}$ is prolonged using $\mathbb{I}_H^h$ on first visit to each finer level
- truncation error within one cycle
- about five work units for many problems
- highly efficient solution method

Some Multigrid Options

- `-snes_grid_sequence`: [0]
  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.

Coupled Multigrids

- Geometric multigrid with isotropic coarsening, ASM(1)/Cholesky and ASM(0)/ICC(0) on levels
  `-mg_levels_pc_type bjacobi -mg_levels_sub_pc_type icc -mg_levels_1_pc_type asm -mg_levels_1_sub_pc_type cholesky`
- ...with Galerkin coarse operators
  `-pc_mg_galerkin`
- ...with ML's aggregates
  `-pc_type ml -mg_levels_pc_type asm`
- Geometric multigrid with aggressive semi-coarsening, ASM(1)/Cholesky and ASM(0)/ICC(0) on levels
  `-da_refine_hierarchy_x 1,1,8,8 -da_refine_hierarchy_y 2,2,1,1 -da_refine_hierarachy_z 2,2,1,1`
- Simulate 1024 cores, interactively, on my laptop
  `-mg_levels_pc_asm_blocks 1024`

Everything is better as a smoother (sometimes)

Block preconditioners work alright, but. . .

- nested iteration requires more dot products
- more iterations: coarse levels don't "see" each other
- finer grained kernels: lower arithmetic intensity, even more limited by memory bandwidth

Coupled multigrid

- need compatible coarsening
  - can do algebraically (Adams 2004) but would need to assemble
- stability issues for lowest order $Q_1 - P_0^{\text{disc}}$
  - Rannacher-Turek looks great, but no discrete Korn's inequality
- coupled "Vanka" smoothers difficult to implement with high performance, especially for FEM
- block preconditioners as smoothers reuse software better
- one level by reducing order for the coarse space, more levels need non-nested geometric MG or go all-algebraic and pay for matrix assembly and setup

Multigrid convergence properties

- Textbook: $P^{-1}A$ is spectrally equivalent to identity
  - Constant number of iterations to converge up to discretization error
- Most theory applies to SPD systems
  - variable coefficients (e.g. discontinuous): low energy interpolants
  - mesh- and/or physics-induced anisotropy: semi-coarsening/line smoothers
  - complex geometry: difficult to have meaningful coarse levels
- Deeper algorithmic difficulties
  - nonsymmetric (e.g. advection, shallow water, Euler)
  - indefinite (e.g. incompressible flow, Helmholtz)
- Performance considerations
  - Aggressive coarsening is critical in parallel
  - Most theory uses SOR smoothers, ILU often more robust
  - Coarsest level usually solved semi-redundantly with direct solver
- Multilevel Schwarz is essentially the same with different language
  - assume strong smoothers, emphasize aggressive coarsening

Algebraic Multigrid Tuning

- Smoothed Aggregation (GAMG, ML)
  - Graph/strength of connection – MatSetBlockSize()
  - Threshold (`-pc_gamg_threshold`)
  - Aggregate (MIS, HEM)
  - Tentative prolongation – MatSetNearNullSpace()
  - Eigenvalue estimate
  - Chebyshev smoothing bounds
- BoomerAMG (Hypre)
  - Strong threshold (`-pc_hypre_boomeramg_strong_threshold`)
  - Aggressive coarsening options

Coupled approach to multiphysics

- Smooth all components together
  - Block SOR is the most popular
  - Block ILU sometimes more robust (e.g. transport/anisotropy)
  - Vanka field-split smoothers or for saddle-point problems
  - Distributive relaxation
- Scaling between fields is critical
- Indefiniteness
  - Make smoothers and interpolants respect inf-sup condition
  - Difficult to handle anisotropy
  - Exotic interpolants for Helmholtz
- Transport
  - Define smoother in terms of first-order upwind discretization (h-ellipticity)
  - Evaluate residuals using high-order discretization
  - Use Schur field-split: “parabolize” at top level or for smoother on levels
- Multigrid inside field-split or field-split inside multigrid
- Open research area, hard to write modular software

Programming with Options

ex55: Allen-Cahn problem in 2D

- constant mobility
- triangular elements

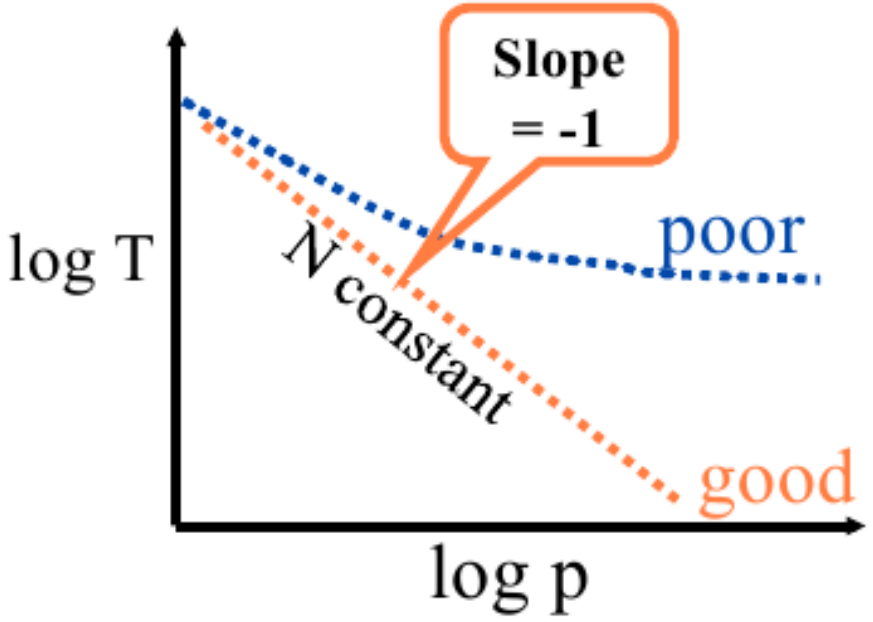
Geometric multigrid method for saddle point variational inequalities:

```
./ex55 -ksp_type fgmres -pc_type mg -mg_levels_ksp_type fgmres
-mg_levels_pc_type fieldsplit -mg_levels_pc_fieldsplit_detect_saddle_point
-mg_levels_pc_fieldsplit_type schur -da_grid_x 65 -da_grid_y 65
-mg_levels_pc_fieldsplit_factorization_type full
-mg_levels_pc_fieldsplit_schur_precondition user
-mg_levels_fieldsplit_1_ksp_type gmres -mg_coarse_ksp_type preonly
-mg_levels_fieldsplit_1_pc_type none   -mg_coarse_pc_type svd
-mg_levels_fieldsplit_0_ksp_type preonly
-mg_levels_fieldsplit_0_pc_type sor    -pc_mg_levels 5
-mg_levels_fieldsplit_0_pc_sor_forward -pc_mg_galerkin
-snes_vi_monitor -ksp_monitor_true_residual -snes_atol 1.e-11
-mg_levels_ksp_monitor -mg_levels_fieldsplit_ksp_monitor
-mg_levels_ksp_max_it 2 -mg_levels_fieldsplit_ksp_max_it 5
```

Scalability definitions

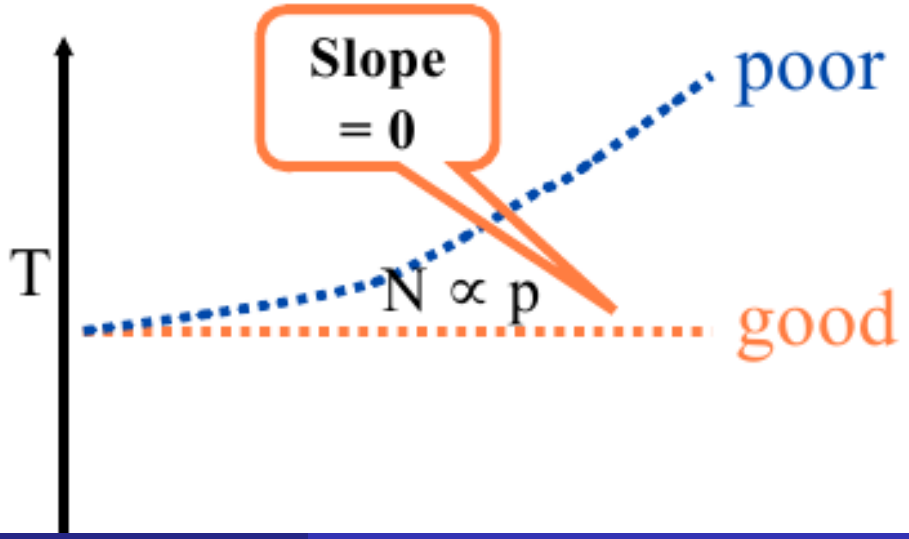
Strong scalability

- 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



Scalability Warning

The easiest way to make software scalable is to make it sequentially inefficient. (Gropp 1999)

- We really want efficient software
- Need a performance model
  - memory bandwidth and latency
  - algorithmically critical operations (e.g. dot products, scatters)
  - floating point unit
- Scalability shows marginal benefit of adding more cores, nothing more
- Constants hidden in the choice of algorithm
- Constants hidden in implementation

Limits of "scalability"?

- **Transient simulation does not weak scale.**
  - Fixed turn-around needed: policy, manufacturing/supply-chain, active control, real-time guidance (field work, surgery, etc.)
  - d-dimensional problem, increase resolution by $2\times$.
  - Data increases by 2^d, but we need $2\times$ more time steps (hyperbolic).
  - With perfect scaling, we use 2^{d+1} more cores.
  - Local data changes by $2^d/2^{d+1} = \frac{1}{2}$
- More applications feeling this
  - Asymptotics are relentless
  - New analysis requires more solves in sequence
    - From forward simulation to optimization with uncertainty ...
  - New physics and higher fidelity observation requires more calibration/validation
- Other applications are safe for now
  - Steady-state solves with scalable methods
  - Transient with a small number of time steps
  - Maximize resolution/problem size – memory-constrained

Evaluating methods

- Performance of methods will depend on grid resolution and model parameters (regime and heterogeneity).
- A method is:
  - scalable (also "optimal") if its performance is independent of resolution and parallelism
  - robust if its performance is (nearly) independent of model parameters
  - efficient if it solves the problem in a small multiple of the cost to evaluate the residual[1]
- Linear problems typically arise from linearizing a nonlinear problem. This step is not necessary, but it is convenient for reusing software and for debugging.

[1] We'll settle for "as fast as the best known method".

Evaluating methods

- Performance of methods will depend on grid resolution and model parameters (regime and heterogeneity).
- A method is:
  - scalable (also "optimal") if its performance is independent of resolution and parallelism
  - robust if its performance is (nearly) independent of model parameters
  - efficient if it solves the problem in a small multiple of the cost to evaluate the residual[1]
- Linear problems typically arise from linearizing a nonlinear problem. This step is not necessary, but it is convenient for reusing software and for debugging.

[1] We'll settle for "as fast as the best known method".

Importance of Computational Modeling

Without a model,
performance measurements are meaningless!

Before a code is written, we should have a model of

- computation
- memory usage
- communication
- bandwidth
- achievable concurrency

This allows us to

- verify the implementation
- predict scaling behavior

Complexity Analysis

The key performance indicator, which we will call the *balance factor* β, is the ratio of flops executed to bytes transfered.

- We will designate the unit $\frac{\text{flop}}{\text{byte}}$ as the *Keyes*
- Using the peak flop rate r_{peak}, we can get the required bandwidth B_{req} for an algorithm

$$B_{\text{req}} = \frac{r_{\text{peak}}}{\beta} \tag{14}$$

- Using the peak bandwidth B_{peak}, we can get the maximum flop rate $r_{\max}$ for an algorithm

$$r_{\max} = \beta B_{\text{peak}} \tag{15}$$

STREAM Benchmark

Simple benchmark program measuring sustainable memory bandwidth

- Protoypical operation is Triad (WAXPY): $\mathbf{w} = \mathbf{y} + \alpha\mathbf{x}$
- Measures the memory bandwidth bottleneck (much below peak)
- Datasets outstrip cache

Machine	Peak (MF/s)	Triad (MB/s)	MF/MW	Eq. MF/s
Matt's Laptop	1700	1122.4	12.1	93.5 (5.5%)
Intel Core2 Quad	38400	5312.0	57.8	442.7 (1.2%)
Tesla 1060C	984000	102000.0*	77.2	8500.0 (0.8%)

Table: Bandwidth limited machine performance

http://www.cs.virginia.edu/stream/

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

Analysis of Sparse Matvec (SpMV)

Assumptions

- No cache misses
- No waits on memory references

Notation

m Number of matrix rows
nz Number of nonzero matrix elements
V Number of vectors to multiply

We can look at bandwidth needed for peak performance

$$\left(8 + \frac{2}{V}\right)\frac{m}{nz} + \frac{6}{V}\ \mathrm{byte/flop} \tag{16}$$

or achieveable performance given a bandwith BW

$$\frac{Vnz}{(8V+2)m + 6nz} BW\ \mathrm{Mflop/s} \tag{17}$$

Towards Realistic Performance Bounds for Implicit CFD Codes, Gropp, Kaushik, Keyes, and Smith.

Performance Caveats

- The peak flop rate r_{peak} on modern CPUs is attained through the usage of a SIMD multiply-accumulate instruction on special 128-bit registers.
- SIMD MAC operates in the form of 4 simultaneous operations (2 adds and 2 multiplies):

$$c_1 = c_1 + a_1 * b_1 \tag{18}$$

$$c_2 = c_2 + a_2 * b_2 \tag{19}$$

  You will miss peak by the corresponding number of operations you are missing. In the worst case, you are reduced to 25% efficiency if your algorithm performs naive summation or products.
- Memory alignment is also crucial when using SSE, the instructions used to load and store from the 128-bit registers throw very costly alignment exceptions when the data is not stored in memory on 16 byte (128 bit) boundaries.

Profiling basics

- Get the math right
  - Choose an algorithm that gives robust iteration counts and really converges
- Look at where the time is spent
  - Run with `-log_summary` and look at events
  - `VecNorm`, `VecDot` measures latency
  - `MatMult` measures neighbor exchange and memory bandwidth
  - `PCSetUp` factorization, aggregation, matrix-matrix products, ...
  - `PCApply` V-cycles, triangular solves, ...
  - `KSPSolve` linear solve
  - `SNESFunctionEval` residual evaluation (user code)
  - `SNESJacobianEval` matrix assembly (user code)

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

Performance Debugging

- PETSc has integrated profiling
  - Option `-log_summary` prints a report on `PetscFinalize()`
- PETSc allows user-defined events
  - Events report time, calls, flops, communication, etc.
  - Memory usage is tracked by object
- Profiling is separated into stages
  - Event statistics are aggregated by stage

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

Reading -log_summary

```
Event                Count      Time (sec)     Flops                             --- Global ---  -
                   Max Ratio  Max     Ratio   Max  Ratio  Mess   Avg len Reduct  %T %F %M %L %R  %
--------------------------------------------------------------------------------------------------
--- Event Stage 1: Full solve
VecDot                43 1.0 4.8879e-02 8.3 1.77e+06 1.0 0.0e+00 0.0e+00 4.3e+01  0  0  0  0  0
VecMDot             1747 1.0 1.3021e+00 4.6 8.16e+07 1.0 0.0e+00 0.0e+00 1.7e+03  0  1  0  0 14
VecNorm             3972 1.0 1.5460e+00 2.5 8.48e+07 1.0 0.0e+00 0.0e+00 4.0e+03  0  1  0  0 31
VecScale            3261 1.0 1.6703e-01 1.0 3.38e+07 1.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0
VecScatterBegin     4503 1.0 4.0440e-01 1.0 0.00e+00 0.0 6.1e+07 2.0e+03 0.0e+00  0  0 50 26  0
VecScatterEnd       4503 1.0 2.8207e+00 6.4 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0
MatMult             3001 1.0 3.2634e+01 1.1 3.68e+09 1.1 4.9e+07 2.3e+03 0.0e+00 11 22 40 24  0  2
MatMultAdd           604 1.0 6.0195e-01 1.0 5.66e+07 1.0 3.7e+06 1.3e+02 0.0e+00  0  0  3  0  0
MatMultTranspose     676 1.0 1.3220e+00 1.6 6.50e+07 1.0 4.2e+06 1.4e+02 0.0e+00  0  0  3  0  0
MatSolve            3020 1.0 2.5957e+01 1.0 3.25e+09 1.0 0.0e+00 0.0e+00 0.0e+00  9 21  0  0  0  1
MatCholFctrSym         3 1.0 2.8324e-04 1.0 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0
MatCholFctrNum        69 1.0 5.7241e+00 1.0 6.75e+08 1.0 0.0e+00 0.0e+00 0.0e+00  2  4  0  0  0
MatAssemblyBegin     119 1.0 2.8250e+00 1.5 0.00e+00 0.0 2.1e+06 5.4e+04 3.1e+02  1  0  2 24  2
MatAssemblyEnd       119 1.0 1.9689e+00 1.4 0.00e+00 0.0 2.8e+05 1.3e+03 6.8e+01  1  0  0  0  1
SNESSolve              4 1.0 1.4302e+02 1.0 8.11e+09 1.0 6.3e+07 3.8e+03 6.3e+03 51 50 52 50 50  9
SNESLineSearch        43 1.0 1.5116e+01 1.0 1.05e+08 1.1 2.4e+06 3.6e+03 1.8e+02  5  1  2  2  1  1
SNESFunctionEval      55 1.0 1.4930e+01 1.0 0.00e+00 0.0 1.8e+06 3.3e+03 8.0e+00  5  0  1  1  0  1
SNESJacobianEval      43 1.0 3.7077e+01 1.0 7.77e+06 1.0 4.3e+06 2.6e+04 3.0e+02 13  0  4 24  2  2
KSPGMRESOrthog      1747 1.0 1.5737e+00 2.9 1.63e+08 1.0 0.0e+00 0.0e+00 1.7e+03  1  1  0  0 14
KSPSetup             224 1.0 2.1040e-02 1.0 0.00e+00 0.0 0.0e+00 0.0e+00 3.0e+01  0  0  0  0  0
KSPSolve              43 1.0 8.9988e+01 1.0 7.99e+09 1.0 5.6e+07 2.0e+03 5.8e+03 32 49 46 24 46  6
PCSetUp              112 1.0 1.7354e+01 1.0 6.75e+08 1.0 0.0e+00 0.0e+00 8.7e+01  6  4  0  0  1  1
PCSetUpOnBlocks     1208 1.0 5.8182e+00 1.0 6.75e+08 1.0 0.0e+00 0.0e+00 8.7e+01  2  4  0  0  1
PCApply              276 1.0 7.1497e+01 1.0 7.14e+09 1.0 5.2e+07 1.8e+03 5.1e+03 25 44 42 20 41  4
```

Adding A Logging Class

```
static int CLASS_ID;

PetscLogClassRegister(&CLASS_ID, "name");
```

- Class ID identifies a class uniquely
- Must initialize before creating any objects of this type

Adding A Logging Event
C

```c
static int USER_EVENT;

PetscLogEventRegister(&USER_EVENT, "name", CLS_ID);
PetscLogEventBegin(USER_EVENT,0,0,0,0);

/* Code to Monitor */

PetscLogFlops(user_event_flops);
PetscLogEventEnd(USER_EVENT,0,0,0,0);
```

Adding A Logging Event
Python

```
with PETSc.logEvent('Reconstruction') as recEvent:
  # All operations are timed in recEvent
  reconstruct(sol)
  # Flops are logged to recEvent
  PETSc.Log.logFlops(user_event_flops)
```

Adding A Logging Stage
C

```
int stageNum;

PetscLogStageRegister(&stageNum, "name");
PetscLogStagePush(stageNum);

/* Code to Monitor */

PetscLogStagePop();
```