Building Robust Scientific Codes

Matthew Knepley

Computation Institute
University of Chicago

Scientific Computing in the Americas:
The Challenge of Massive Parallelism
Valparaiso, Chile, January 2011
Outline

1. Tools and Infrastructure
   - Version Control
   - Configuration and Build
   - PETSc
   - numpy
   - sympy
   - petsc4py
   - PyCUDA
   - FEniCS

2. GPU Computing

3. Linear Algebra and Solvers

4. First Lab: Assembling a Code
What I Need From You

- Tell me if you do not understand
- Tell me if an example does not work
- Suggest better wording or figures
- Followup problems at petsc-maint@mcs.anl.gov
Ask Questions!!!

- Helps **me** understand what you are missing
- Helps **you** clarify misunderstandings
- Helps **others** with the same question
How We Can Help at the Tutorial

- Point out relevant documentation
- Quickly answer questions
- Help install
- Guide design of large scale codes
- Answer email at petsc-maint@mcs.anl.gov
How We Can Help at the Tutorial

- Point out relevant documentation
- Quickly answer questions
- Help install
- Guide design of large scale codes
- Answer email at petsc-maint@mcs.anl.gov
Point out relevant documentation
Quickly answer questions
Help install
Guide design of large scale codes
Answer email at petsc-maint@mcs.anl.gov
How We Can Help at the Tutorial

- Point out relevant documentation
- Quickly answer questions
- Help install
- **Guide design of large scale codes**
- Answer email at petsc-maint@mcs.anl.gov
New Model for Scientific Software

Simplifying Parallelization of Scientific Codes by a Function-Centric Approach in Python

Jon K. Nilsen, Xing Cai, Bjorn Hoyland, and Hans Petter Langtangen

- **Python** at the application level
- **numpy** for data structures
- **petsc4py** for linear algebra and solvers
- **PyCUDA** for integration (physics) and assembly
New Model for Scientific Software

Tools and Infrastructure

Application

sympy

symbolics

FFC/SyFi

eqn. definition

data structures

integrated/assembly

petsc4py

solvers

numpy

PyCUDA

PETSc

CUDA
OpenCL

M. Knepley ()

Robust

PASI '11 7 / 224
What is Missing from this Scheme?

- Unstructured graph traversal
  - Iteration over cells in FEM
    - Use a copy via numpy, use a kernel via Queue
  - (Transitive) Closure of a vertex
    - Use a visitor and copy via numpy
  - Depth First Search
    - Hell if I know

- Logic in computation
  - Limiters in FV methods
    - Can sometimes use tricks for branchless logic
  - Flux Corrected Transport for shock capturing
    - Maybe use WENO schemes which can be branchless

- Boundary conditions
  - Restrict branching to PETSc C numbering and assembly calls

Audience???
What is Missing from this Scheme?

- Unstructured graph traversal
  - Iteration over cells in FEM
    - Use a copy via numpy, use a kernel via Queue
  - (Transitive) Closure of a vertex
    - Use a visitor and copy via numpy
- Depth First Search
  - Hell if I know
- Logic in computation
  - Limiters in FV methods
    - Can sometimes use tricks for branchless logic
  - Flux Corrected Transport for shock capturing
    - Maybe use WENO schemes which can be branchless
- Boundary conditions
  - Restrict branching to PETSc C numbering and assembly calls

Audience???
What is Missing from this Scheme?

- Unstructured graph traversal
  - Iteration over cells in FEM
    - Use a copy via numpy, use a kernel via Queue
  - (Transitive) Closure of a vertex
    - Use a visitor and copy via numpy
- Depth First Search
  - Hell if I know
- Logic in computation
  - Limiters in FV methods
    - Can sometimes use tricks for branchless logic
  - Flux Corrected Transport for shock capturing
    - Maybe use WENO schemes which can be branchless
- Boundary conditions
  - Restrict branching to PETSc C numbering and assembly calls

Audience???
What is Missing from this Scheme?

- Unstructured graph traversal
  - Iteration over cells in FEM
    - Use a copy via numpy, use a kernel via Queue
  - (Transitive) Closure of a vertex
    - Use a visitor and copy via numpy
  - Depth First Search
    - Hell if I know

- Logic in computation
  - Limiters in FV methods
    - Can sometimes use tricks for branchless logic
  - Flux Corrected Transport for shock capturing
    - Maybe use WENO schemes which can be branchless

- Boundary conditions
  - Restrict branching to PETSc C numbering and assembly calls

- Audience???
What is Missing from this Scheme?

- Unstructured graph traversal
  - Iteration over cells in FEM
    - Use a copy via numpy, use a kernel via Queue
  - (Transitive) Closure of a vertex
    - Use a visitor and copy via numpy
  - Depth First Search
    - Hell if I know

- Logic in computation
  - Limiters in FV methods
    - Can sometimes use tricks for branchless logic
  - Flux Corrected Transport for shock capturing
    - Maybe use WENO schemes which can be branchless

- Boundary conditions
  - Restrict branching to PETSc C numbering and assembly calls

- Audience???
What is Missing from this Scheme?

- Unstructured graph traversal
  - Iteration over cells in FEM
    - Use a copy via numpy, use a kernel via Queue
  - (Transitive) Closure of a vertex
    - Use a visitor and copy via numpy
  - Depth First Search
    - Hell if I know

- Logic in computation
  - Limiters in FV methods
    - Can sometimes use tricks for branchless logic
  - Flux Corrected Transport for shock capturing
    - Maybe use WENO schemes which can be branchless

- Boundary conditions
  - Restrict branching to PETSc C numbering and assembly calls

Audience???
What is Missing from this Scheme?

- Unstructured graph traversal
  - Iteration over cells in FEM
    - Use a copy via numpy, use a kernel via Queue
  - (Transitive) Closure of a vertex
    - Use a visitor and copy via numpy
  - Depth First Search
    - *Hell if I know*

- Logic in computation
  - Limiters in FV methods
    - Can sometimes use tricks for branchless logic
  - Flux Corrected Transport for shock capturing
    - Maybe use WENO schemes which can be branchless

- Boundary conditions
  - Restrict branching to PETSc C numbering and assembly calls

*Audience???
What is Missing from this Scheme?

- Unstructured graph traversal
  - Iteration over cells in FEM
    - Use a copy via numpy, use a kernel via Queue
  - (Transitive) Closure of a vertex
    - Use a visitor and copy via numpy
  - Depth First Search
    - Hell if I know

- Logic in computation
  - Limiters in FV methods
    - Can sometimes use tricks for branchless logic
  - Flux Corrected Transport for shock capturing
    - Maybe use WENO schemes which can be branchless
  - Boundary conditions
    - Restrict branching to PETSc C numbering and assembly calls

- Audience???
Tools and Infrastructure

Version Control

Configuration and Build

PETSc

numpy

sympy

petsc4py

PyCUDA

FEniCS
Location and Retrieval
“Where’s the Tarball”

- **Version Control**
  - Mercurial, Git, Subversion

- **Hosting**
  - BitBucket, GitHub, Launchpad

- **Community involvement**
  - arXiv
Distributed VC

- CVS/SVN manage a single repository
  - Versioned data
  - Local copy for modification and checkin

- Mercurial manages many repositories
  - Identified by URLs
  - No one *Master*

- Repositories communicate by *ChangeSets*
  - Use *push* and *pull* to move changesets
  - Can move arbitrary changes with *patch queues*
Project Workflow

Figure: Single Repository
Project Workflow

Figure: Master Repository with User Clones
Figure: Project with Release and Bugfix Repositories
Tools and Infrastructure

1. Version Control
2. Configuration and Build
   - PETSc
   - numpy
   - sympy
   - petsc4py
   - PyCUDA
   - FEniCS
Configuration and Build

“It won’t run on my iPhone”

- Portability
  - PETSc BuildSystem, autoconf

- Dependencies
  - Does this work with UnsupportedGradStudentAMG?

- Configurable build
  - Build must integrate with the configuration system
  - SCons, Jam, CMake
BuildSystem

Provides tools for Configuration and Build

- Dependency tracking and analysis
- Package management and hierarchy
- Library of standard tests
- Standard build rules
- Automatic package build and integration

http://petsc.cs.iit.edu/petsc/BuildSystem
BuildSystem.config.base configures a specific functionality

- Entry points:
  - setupHelp()
  - setupDependencies()
  - configure()

- Built-in capabilities:
  - Preprocessing, compilation, linking, running
  - Manages languages
  - Checks for executables

- Output types:
  - Define, typedef, or prototype
  - Make macro or rule
  - Substitution (old-style)
BuildSystem.config.framework manages the configure run

- Manages configure modules
  - Dependencies with DAG, `require()`
  - Options table
  - Initialization, run, cleanup

- Outputs
  - Configure headers and log
  - Make variable and rules
  - Pickled configure tree
A module can declare a dependency using:

```python
fw = self.framework
self.mpi = fw.require('config.packages.MPI', self)
```

so that MPI is configured before `self`. Information is retrieved during `configure()`:

```python
if self.mpi.found:
    include.extend(self.mpi.include)
    libs.extend(self.mpi.lib)
```
A module can declare a dependency using:

```python
fw = self.framework
self.mpi = fw.require('config.packages.MPI', self)
```

so that MPI is configured before `self`. Information is retrieved during `configure()`:

```python
if self.mpi.found:
    include.extend(self.mpi.include)
    libs.extend(self.mpi.lib)
```
A build system can acquire the information using:

```python
class ConfigReader(script.Script):
    def __init__(self):
        import RDict
        argDB = RDict.RDict(None, None, 0, 0)
        argDB.saveFilename = os.path.join('path','RDict.db')
        argDB.load()
        script.Script.__init__(self, argDB = argDB)
        return

    def getMPIModule(self):
        self.setup()
        fw = self.loadConfigure()
        mpi = fw.require('config.packages.MPI', None)
        return mpi
```
GNU Make automates a package build

- Has a single predicate, older-than
- Executes shell code for actions
- PETSc has support for
  - configuration integration
  - automatic compilation
- Alternatives
  - SCons
  - Jam
  - CMake
Simple replacement for GNU make

- Excellent configure integration
- User-defined predicates
- Dependency analysis and tracking
- Python actions
- Support for test execution
Testing

“They are identical in the eyeball norm”

- Unit tests
  - cppUnit

- Regression tests
  - buildbot

- Benchmarks
  - Cigma
Outline

1 Tools and Infrastructure
   • Version Control
   • Configuration and Build
   • PETSc
     • numpy
     • sympy
     • petsc4py
     • PyCUDA
     • FEniCS
PETSc was developed as a Platform for Experimentation

We want to experiment with different

- Models
- Discretizations
- Solvers
- Algorithms
  - which blur these boundaries
The Role of PETSc

Developing parallel, nontrivial PDE solvers that deliver high performance is still difficult and requires months (or even years) of concentrated effort.

*PETSc is a toolkit that can ease these difficulties and reduce the development time, but it is not a black-box PDE solver, nor a silver bullet.*

— Barry Smith
What is PETSc?

A freely available and supported research code

- Download from http://www.mcs.anl.gov/petsc
- Free for everyone, including industrial users
- Hyperlinked manual, examples, and manual pages for all routines
- Hundreds of tutorial-style examples
- Support via email: petsc-maint@mcs.anl.gov
- Usable from C, C++, Fortran 77/90, and Python
What is PETSc?

- Portable to any parallel system supporting MPI, including:
  - Tightly coupled systems
    - Cray XT5, BG/P, NVIDIA Tesla, Earth Simulator, Sun Blade
  - Loosely coupled systems, such as networks of workstations
    - IBM, Mac, Sun, PCs running Linux or Windows

PETSc History

- Begun September 1991
- Over 60,000 downloads since 1995 (version 2)
- Currently 400 per month

PETSc Funding and Support

- Department of Energy
  - SciDAC, MICS Program, INL Reactor Program
- National Science Foundation
  - CIG, CISE, Multidisciplinary Challenge Program
Timeline

- PETSc 1 release
- PETSc 2 release
- +Barry
- +Bill
- +Lois
- +Satish
- +Matt
- +Hong
- +Kris
- -Lois
- -Bill
- +Victor
- +Kris
- +Matt
- +Hong
What Can We Handle?

- PETSc has run implicit problems with over **500 billion** unknowns
  - UNIC on BG/P and XT5
  - PFLOTRAN for flow in porous media

- PETSc has run on over **224,000** cores efficiently
  - UNIC on the IBM BG/P Intrepid at ANL
  - PFLOTRAN on the Cray XT5 Jaguar at ORNL

- PETSc applications have run at **22 Teraflops**
  - Kaushik on XT5
  - LANL PFLOTRAN code
What Can We Handle?

- PETSc has run implicit problems with over 500 billion unknowns
  - UNIC on BG/P and XT5
  - PFLOTRAN for flow in porous media

- PETSc has run on over 224,000 cores efficiently
  - UNIC on the IBM BG/P Intrepid at ANL
  - PFLOTRAN on the Cray XT5 Jaguar at ORNL

- PETSc applications have run at 22 Teraflops
  - Kaushik on XT5
  - LANL PFLOTRAN code
What Can We Handle?

- PETSc has run implicit problems with over **500 billion** unknowns
  - UNIC on BG/P and XT5
  - PFLOTRAN for flow in porous media

- PETSc has run on over **224,000** cores efficiently
  - UNIC on the IBM BG/P Intrepid at ANL
  - PFLOTRAN on the Cray XT5 Jaguar at ORNL

- PETSc applications have run at **22 Teraflops**
  - Kaushik on XT5
  - LANL PFLOTRAN code
1 Tools and Infrastructure

- Version Control
- Configuration and Build
- PETSc
- numpy
  - sympy
  - petsc4py
  - PyCUDA
  - FEniCS
**numpy** is ideal for building Python data structures

- Supports multidimensional arrays
- Easily interfaces with C/C++ and Fortran
- High performance BLAS/LAPACK and functional operations
- Python 2 and 3 compatible
- Used by petsc4py to talk to PETSc
1. Tools and Infrastructure
   - Version Control
   - Configuration and Build
   - PETSc
   - numpy
   - sympy
   - petsc4py
   - PyCUDA
   - FEniCS
sympy is useful for symbolic manipulation

- Interacts with numpy
- Derivatives and integrals
- Series expansions
- Equation simplification
- Small and open source
1 Tools and Infrastructure

- Version Control
- Configuration and Build
- PETSc
- numpy
- sympy
- petsc4py
- PyCUDA
- FEniCS
**petsc4py** provides Python bindings for PETSc

- Provides **ALL** PETSc functionality in a Pythonic way
  - Logging using the Python `with` statement

- Can use Python callback functions
  - `SNESSetFunction()`, `SNESSetJacobian()`

- Manages all memory (creation/destruction)

- Visualization with `matplotlib`
Old style
- Configure PETSc using `-download-petsc4py`
- Downloaded to `externalpackages/petsc4py-version`
- Demo code is there as well
- Installed into PETSc lib directory
- Add `$PETSC_DIR/$PETSC_ARCH/lib` to `PYTHONPATH`

New style
- `pip install --install-options=-user petsc4py`
- Uses `$PETSC_DIR` and `$PETSC_ARCH`
- Installed into `$HOME/.local`
- No additions to `PYTHONPATH`
**petsc4py Examples**

- `externalpackages/petsc4py-1.1/demo/bratu2d/bratu2d.py`
  - Solves Bratu equation (SNES `ex5`) in 2D
  - Visualizes solution with `matplotlib`

- `src/ts/examples/tutorials/ex8.py`
  - Solves a 1D ODE for a diffusive process
  - Visualize solution using `-vec_view_draw`
  - Control timesteps with `-ts_max_steps`
Tools and Infrastructure

- Version Control
- Configuration and Build
- PETSc
- numpy
- sympy
- petsc4py
- PyCUDA
- FEniCS
Python packages by Andreas Klöckner for embedded GPU programming

- Handles unimportant details automatically
  - CUDA compile and caching of objects
  - Device initialization
  - Loading modules onto card

- Excellent Documentation & Tutorial

- Excellent platform for Metaprogramming
  - Only way to get portable performance
  - Road to FLAME-type reasoning about algorithms
void kernel(float *output) {
    float G[dim*dim] = {
        .join(['3.0']*dim*dim)
    };
    float K[dim*dim] = {
        .join(['3.0']*dim*dim)
    };
    float product = 0.0;
    const int Ooffset = gridIdx * numThreads;

    // Contract G and K
    for n in range(numLocalElements):
        for alpha in range(dim):
            for beta in range(dim):
                gIdx = (n*dim + alpha)*dim + beta
                kIdx = alpha*dim + beta
                product += G[gIdx] * K[kIdx];
        endfor
    endfor
    output[Ooffset+idx] = product;
    return;
}
We render code template into strings using a dictionary of inputs.

```python
args = {'dim': self.dim,
        'numLocalElements': 1,
        'numThreads': self.threadBlockSize}

kernelTemplate = self.getKernelTemplate()
gpuCode = kernelTemplate.render(isGPU = True, **args)
cpuCode = kernelTemplate.render(isGPU = False, **args)
```
__global__ void kernel( float *output) {
    const int gridIdx = blockIdx.x + blockIdx.y*gridDim.x;
    const int idx = threadIdx.x + threadIdx.y*1; // This is (i,j)
    float G[9] = {3.0,3.0,3.0,3.0,3.0,3.0,3.0,3.0,3.0};
    float K[9] = {3.0,3.0,3.0,3.0,3.0,3.0,3.0,3.0,3.0};
    float product = 0.0;
    const int Ooffset = gridIdx*1;

    // Contract G and K
    product += G[0] * K[0];
    product += G[1] * K[1];
    product += G[4] * K[4];
    product += G[5] * K[5];
    product += G[7] * K[7];
    product += G[8] * K[8];
    output[Ooffset+idx] = product;
    return;
}
void kernel(int numInvocations, float *output) {
    for(int gridIdx = 0; gridIdx < numInvocations; ++gridIdx) {
        for(int i = 0; i < 1; ++i) {
            for(int j = 0; j < 1; ++j) {
                const int idx = i + j*1; // This is (i,j)
                float G[9] = {3.0,3.0,3.0,3.0,3.0,3.0,3.0,3.0,3.0};
                float K[9] = {3.0,3.0,3.0,3.0,3.0,3.0,3.0,3.0,3.0};
                float product = 0.0;
                const int Ooffset = gridIdx*1;

                // Contract G and K
                product += G[0] * K[0];
                product += G[1] * K[1];
                product += G[4] * K[4];
                product += G[5] * K[5];
                product += G[7] * K[7];
                product += G[8] * K[8];
                output[Ooffset+idx] = product;
            }
        }
    }
    return;
}
Creating a Module

CPU:

# Output kernel and C support code
self.outputKernelC(cpuCode)
self.writeMakefile()
out, err, status = self.executeShellCommand('make')

GPU:

from pycuda.compiler import SourceModule
mod = SourceModule(gpuCode)
self.kernel = mod.get_function('kernel')
self.kernelReport(self.kernel, 'kernel')
import pycuda.driver as cuda
import pycuda.autoinit

blockDim = (self.dim, self.dim, 1)
start = cuda.Event()
end = cuda.Event()
grid = self.calculateGrid(N, numLocalElements)
start.record()
for i in range(iters):
    self.kernel(cuda.Out(output),
                block = blockDim, grid = grid)
end.record()
end.synchronize()
gpuTimes.append(start.time_till(end)*1e-3/iters)
Outline

1. Tools and Infrastructure
   - Version Control
   - Configuration and Build
   - PETSc
   - numpy
   - sympy
   - petsc4py
   - PyCUDA
   - FEniCS
FIAT

Finite Element Integrator And Tabulator by Rob Kirby

http://www.fenics.org/fiat

FIAT understands

- Reference element shapes (line, triangle, tetrahedron)
- Quadrature rules
- Polynomial spaces
- Functionals over polynomials (dual spaces)
- Derivatives

User can build arbitrary elements specifying the Ciarlet triple \((K, P, P')\)

FIAT is part of the FEniCS project, as is the PETSc Sieve module
FFC is a compiler for variational forms by Anders Logg.

Here is a mixed-form Poisson equation:

\[ a((\tau, w), (\sigma, u)) = L((\tau, w)) \quad \forall (\tau, w) \in V \]

where

\[ a((\tau, w), (\sigma, u)) = \int_{\Omega} \tau \sigma - \nabla \cdot \tau u + w \nabla \cdot u \, dx \]

\[ L((\tau, w)) = \int_{\Omega} wf \, dx \]
shape = "triangle"

BDM1 = FiniteElement("Brezzi-Douglas-Marini", shape, 1)
DG0 = FiniteElement("Discontinuous Lagrange", shape, 0)

element = BDM1 + DG0
(tau, w) = TestFunctions(element)
(sigma, u) = TrialFunctions(element)

a = (dot(tau, sigma) - div(tau)*u + w*div(sigma))*dx

f = Function(DG0)
L = w*f*dx
Here is a discontinuous Galerkin formulation of the Poisson equation:

\[ a(v, u) = L(v) \quad \forall v \in V \]

where

\[
\begin{align*}
    a(v, u) &= \int_{\Omega} \nabla u \cdot \nabla v \, dx \\
    &\quad + \sum_S \int_S -[\nabla v \cdot [u]]_n - [v]]_n \cdot \nabla u - (\alpha / h) vu \, dS \\
    &\quad + \int_{\partial \Omega} -\nabla v \cdot [u]_n - [v]_n \cdot \nabla u - (\gamma / h) vu \, ds \\
    L(v) &= \int_{\Omega} vf \, dx
\end{align*}
\]
DG1 = FiniteElement("Discontinuous Lagrange", shape, 1)
v = TestFunctions(DG1)
u = TrialFunctions(DG1)
f = Function(DG1)
g = Function(DG1)
n = FacetNormal("triangle")
h = MeshSize("triangle")
a = dot(grad(v), grad(u)) * dx 
   - dot(avg(grad(v)), jump(u, n)) * dS 
   - dot(jump(v, n), avg(grad(u))) * dS 
   + alpha/h*dot(jump(v, n) + jump(u, n)) * dS 
   - dot(grad(v), jump(u, n)) * ds 
   - dot(jump(v, n), grad(u)) * ds 
   + gamma/h*v*u*ds 
L = v*f*dx + v*g*ds
Big Picture

- **Usability** is paramount
  - Need community buy-in
  - Need complete workflow

- Leverage **existing systems**
  - Adoption is much easier with the familiar
  - arXiv, package managers
Outline

1. Tools and Infrastructure

2. GPU Computing
   - FEM-GPU
   - PETSc-GPU

3. Linear Algebra and Solvers

4. First Lab: Assembling a Code

5. Second Lab: Debugging and Performance Benchmarking
Collaborators

- **Dr. Andy Terrel** (FEniCS)
  - Dept. of Computer Science, University of Texas
  - Texas Advanced Computing Center, University of Texas

- **Prof. Andreas Klöckner** (PyCUDA)
  - Courant Institute of Mathematical Sciences, New York University

- **Dr. Brad Aagaard** (PyLith)
  - United States Geological Survey, Menlo Park, CA

- **Dr. Charles Williams** (PyLith)
  - GNS Science, Wellington, NZ
2 GPU Computing

- FEM-GPU
- PETSc-GPU
What’s Good Here?

Low Order FEM on GPUs

- Analytic Flexibility
- Computational Flexibility
- Efficiency

http://www.bitbucket.org/aterrel/flamefem
Low Order FEM on GPUs

- Analytic Flexibility
- Computational Flexibility
- Efficiency

http://www.bitbucket.org/aterrel/flamefem
What’s Good Here?

Low Order FEM on GPUs

- Analytic Flexibility
- Computational Flexibility
- Efficiency

http://www.bitbucket.org/aterrel/flamefem
What’s Good Here?

Low Order FEM on GPUs

- Analytic Flexibility
- Computational Flexibility
- Efficiency

http://www.bitbucket.org/aterrel/flamefem
Analytic Flexibility
Laplacian

\[ \int_T \nabla \phi_i(x) \cdot \nabla \phi_j(x) \, dx \]  

```
element = FiniteElement('Lagrange', tetrahedron, 1)
v = TestFunction(element)
u = TrialFunction(element)
a = inner(grad(v), grad(u)) * dx
```
Analytic Flexibility
Laplacian

\[
\int_T \nabla \phi_i(x) \cdot \nabla \phi_j(x) \, dx
\]  

(1)

element = FiniteElement('Lagrange', tetrahedron, 1)
v = TestFunction(element)
u = TrialFunction(element)
a = inner(grad(v), grad(u))\, dx
\[
\frac{1}{4} \int_T \left( \nabla \phi_i(x) + \nabla^T \phi_i(x) \right) : \left( \nabla \phi_j(x) + \nabla^T \phi_j(x) \right) \, dx
\] (2)

element = VectorElement('Lagrange', tetrahedron, 1)
v = TestFunction(element)
u = TrialFunction(element)
a = inner(sym(grad(v)), sym(grad(u))) \times dx
\[
\frac{1}{4} \int_T \left( \nabla \phi_i(x) + \nabla^T \phi_i(x) \right) : \left( \nabla \phi_j(x) + \nabla \phi_j(x) \right) \, dx
\]  

\text{element} = \text{VectorElement('Lagrange', tetrahedron, 1)} \\
\text{v} = \text{TestFunction(element)} \\
\text{u} = \text{TrialFunction(element)} \\
\text{a} = \text{inner}\left(\text{sym}(\text{grad}(\text{v})), \text{sym}(\text{grad}(\text{u}))\right) \star dx
\[ \frac{1}{4} \int_T \left( \nabla \phi_i(x) + \nabla^T \phi_i(x) \right) : C : \left( \nabla \phi_j(x) + \nabla \phi_j(x) \right) \, dx \quad (3) \]

```python
element = VectorElement('Lagrange', tetrahedron, 1)
cElement = TensorElement('Lagrange', tetrahedron, 1,
(dim, dim, dim, dim, dim))
v = TestFunction(element)
u = TrialFunction(element)
C = Coefficient(cElement)
i, j, k, l = indices(4)
a = sym(grad(v))[i,j]*C[i,j,k,l]*sym(grad(u))[k,l]*dx
```

Currently broken in FEniCS release
Analytic Flexibility
Full Elasticity

\[
\frac{1}{4} \int_{\mathcal{T}} \left( \nabla \phi_i(\mathbf{x}) + \nabla^T \phi_i(\mathbf{x}) \right) : \mathbf{C} : \left( \nabla \phi_j(\mathbf{x}) + \nabla^T \phi_j(\mathbf{x}) \right) d\mathbf{x} \quad (3)
\]

element = VectorElement('Lagrange', tetrahedron, 1)
cElement = TensorElement('Lagrange', tetrahedron, 1,
        (dim, dim, dim, dim, dim))
v = TestFunction(element)
u = TrialFunction(element)
C = Coefficient(cElement)
i, j, k, l = indices(4)
a = sym(grad(v))[i,j]*C[i,j,k,l]*sym(grad(u))[k,l]*dx

Currently broken in FEniCS release
\[
\frac{1}{4} \int_{\mathcal{T}} \left( \nabla \vec{\phi}_i(x) + \nabla^T \vec{\phi}_i(x) \right) : C : \left( \nabla \vec{\phi}_j(x) + \nabla^T \vec{\phi}_j(x) \right) \, dx
\] (3)

```python
element = VectorElement('Lagrange', tetrahedron, 1)
cElement = TensorElement('Lagrange', tetrahedron, 1, (dim, dim, dim, dim, dim))
v = TestFunction(element)
u = TrialFunction(element)
C = Coefficient(cElement)
i, j, k, l = indices(4)
a = sym(grad(v))[[i, j]] * C[[i, j, k, l]] * sym(grad(u))[k, l] * dx
```

Currently **broken** in FEniCS release
Element integrals are decomposed into **analytic** and **geometric** parts:

\[
\int_{\mathcal{T}} \nabla \phi_i(\mathbf{x}) \cdot \nabla \phi_j(\mathbf{x}) \, d\mathbf{x} = \int_{\mathcal{T}} \frac{\partial \phi_i(\mathbf{x})}{\partial x_\alpha} \frac{\partial \phi_j(\mathbf{x})}{\partial x_\alpha} \, d\mathbf{x}
\]

(4)

\[
= \int_{\mathcal{T}} \frac{\partial \phi_i(\xi)}{\partial x_\alpha} \frac{\partial \phi_j(\xi)}{\partial x_\alpha} \, d\xi \quad \text{(5)}
\]

\[
= \int_{\mathcal{T}_{\text{ref}}} \frac{\partial \xi_\beta}{\partial x_\alpha} \frac{\partial \phi_i(\xi)}{\partial \xi_\gamma} \frac{\partial \phi_j(\xi)}{\partial \xi_\gamma} |J| \, d\xi \quad \text{(6)}
\]

\[
= \left| \frac{\partial \xi_\beta}{\partial x_\alpha} \frac{\partial \xi_\gamma}{\partial x_\alpha} \right| \int_{\mathcal{T}_{\text{ref}}} \frac{\partial \phi_i(\xi)}{\partial \xi_\beta} \frac{\partial \phi_j(\xi)}{\partial \xi_\gamma} \, d\xi \quad \text{(7)}
\]

\[
= G_{\beta\gamma}(\mathcal{T}) K_{ij}^{\beta\gamma} \quad \text{(8)}
\]

Coefficients are also put into the geometric part.
Additional fields give rise to **multilinear** forms.

\[
\int_T \phi_i(x) \cdot (\phi_k(x) \nabla \phi_j(x)) \, dA \\
= \int_T \phi^\beta_i(x) \left( \phi^\alpha_k(x) \frac{\partial \phi_j^\beta(x)}{\partial x^\alpha} \right) \, dA \\
= \int_{T_{\text{ref}}} \phi^\beta_i(\xi) \phi^\alpha_k(\xi) \frac{\partial \xi^\gamma}{\partial x^\alpha} \frac{\partial \phi_j^\beta(\xi)}{\partial \xi^\gamma} |J| \, dA \\
= \frac{\partial \xi^\gamma}{\partial x^\alpha} |J| \int_{T_{\text{ref}}} \phi^\beta_i(\xi) \phi^\alpha_k(\xi) \frac{\partial \phi_j^\beta(\xi)}{\partial \xi^\gamma} \, dA \\
= G^{\alpha\gamma}(T) K_{\alpha\gamma}^{ijk}
\]

The index calculus is fully developed by Kirby and Logg in *A Compiler for Variational Forms*.
Form Decomposition

Isoparametric Jacobians also give rise to **multilinear** forms

\[
\int_{\mathcal{T}} \nabla \phi_i(\mathbf{x}) \cdot \nabla \phi_j(\mathbf{x}) dA = \int_{\mathcal{T}} \frac{\partial \phi_i(\mathbf{x})}{\partial x_\alpha} \frac{\partial \phi_j(\mathbf{x})}{\partial x_\alpha} dA
\]

\[
= \int_{\mathcal{T}_{\text{ref}}} \frac{\partial \xi_{\beta}}{\partial x_\alpha} \frac{\partial \phi_i(\xi)}{\partial \xi_{\beta}} \frac{\partial \xi_{\gamma}}{\partial x_\alpha} \frac{\partial \phi_j(\xi)}{\partial \xi_{\gamma}} |J| dA
\]

\[
= |J| \int_{\mathcal{T}_{\text{ref}}} \phi_k J^\beta_\alpha \frac{\partial \phi_i(\xi)}{\partial \xi_{\beta}} \phi_l J^\gamma_\alpha \frac{\partial \phi_j(\xi)}{\partial \xi_{\gamma}} dA
\]

\[
= J^\beta_\alpha J^\gamma_\alpha |J| \int_{\mathcal{T}_{\text{ref}}} \phi_k \frac{\partial \phi_i(\xi)}{\partial \xi_{\beta}} \phi_l \frac{\partial \phi_j(\xi)}{\partial \xi_{\gamma}} dA
\]

\[
= G^{\beta\gamma}_{kl}(\mathcal{T}) K^{ijkl}_{\beta\gamma}
\]

A different space could also be used for Jacobians
from ffc.analysis import analyze_forms
from ffc.compiler import compute_ir

parameters = ffc.default_parameters()
parameters['representation'] = 'tensor'
analysis = analyze_forms([a,L], {}, parameters)
ir = compute_ir(analysis, parameters)

a_K = ir[2][0]['AK'][0][0]
a_G = ir[2][0]['AK'][0][1]

K = a_K.A0.astype(numpy.float32)
G = a_G
We *generate* different computations on the fly, and can change:

- Element Batch Size
- Number of Concurrent Elements
- Loop unrolling
- Interleaving stores with computation
Computational Flexibility
Basic Contraction

Figure: Tensor Contraction $G^{\beta\gamma}(T)K_{\beta\gamma}^{ij}$
Figure: Tensor Contraction $G^{\beta\gamma}(T)K_{\beta\gamma}^{ij}$
Computational Flexibility

Basic Contraction

Figure: Tensor Contraction $G^{\beta\gamma}(T)K_{\beta\gamma}^{ij}$
Figure: Tensor Contraction $G^{\beta \gamma}(T)K_{\beta \gamma}^{ij}$
Computational Flexibility
Element Batch Size

Figure: Tensor Contraction $G_{\beta\gamma}(T)K^{ij}_{\beta\gamma}$
Figure: Tensor Contraction $G^{\beta\gamma}(T)K_{ij}^{\gamma\beta}$
Computational Flexibility
Element Batch Size

Figure: Tensor Contraction $G^{\beta\gamma}(T)K_{\beta\gamma}^{ij}$
Figure: Tensor Contraction $G^{\beta\gamma}(T)K_{\beta\gamma}^{ij}$
Computational Flexibility
Concurrent Elements

Figure: Tensor Contraction

\[ K \]

\[ G_0^0 \]
\[ G_1^0 \]
\[ G_2^0 \]
\[ G_3^0 \]

\[ G_0^1 \]
\[ G_1^1 \]
\[ G_2^1 \]
\[ G_3^1 \]

thread 0
thread 5
thread 15
thread 16
thread 21
thread 31

M. Knepley ()
Robust
PASI '11 68 / 224
Computational Flexibility
Concurrent Elements

Figure: Tensor Contraction

\[ K \]

\[ G_0^0 \]

\[ G_0^0 \]

\[ G_0^1 \]

\[ G_0^2 \]

\[ G_0^3 \]

\[ G_1^1 \]

\[ G_1^1 \]

\[ G_2^1 \]

\[ G_3^1 \]

thread 0

thread 5

thread 15

thread 16

thread 21

thread 31

\[ \beta \gamma (T) K_{ij} \beta \gamma \]
Computational Flexibility
Concurrent Elements

Figure: Tensor Contraction

$G_0^0 \quad G_0^1 \quad G_1^0 \quad G_1^1 \quad G_2^0 \quad G_2^1 \quad G_3^0 \quad G_3^1$

$K$

thread 0
thread 5
thread 15
thread 16
thread 21
thread 31
Computational Flexibility
Concurrent Elements

Figure: Tensor Contraction

$$G^0_0 \quad G^0_1 \quad G^0_2 \quad G^0_3$$

$$G^1_0 \quad G^1_1 \quad G^1_2 \quad G^1_3$$

$$K$$

Threads:
- Thread 0
- Thread 5
- Thread 15
- Thread 31
/* G K contraction: unroll = full */
E[0] += G[0] * K[0];
E[0] += G[1] * K[1];
E[0] += G[7] * K[7];
E[0] += G[8] * K[8];
/* G K contraction: unroll = none */
for(int b = 0; b < 1; ++b) {
    const int n = b*1;
    for(int alpha = 0; alpha < 3; ++alpha) {
        for(int beta = 0; beta < 3; ++beta) {
            E[b] += G[n*9+alpha*3+beta] * K[alpha*3+beta];
        }
    }
}
/* G K contraction: unroll = none */

for (int b = 0; b < 4; ++b) {
    const int n = b*1;
    for (int alpha = 0; alpha < 3; ++alpha) {
        for (int beta = 0; beta < 3; ++beta) {
            E[b] += G[n*9+alpha*3+beta] * K[alpha*3+beta];
        }
    }
}

/* Store contraction results */

elemMat[Eoffset+idx+0] = E[0];
elemMat[Eoffset+idx+16] = E[1];
elemMat[Eoffset+idx+32] = E[2];
elemMat[Eoffset+idx+48] = E[3];
n = 0;
for (int alpha = 0; alpha < 3; ++alpha) {
    for (int beta = 0; beta < 3; ++beta) {
        E += G[n*9+alpha*3+beta] * K[alpha*3+beta];
    }
}
/* Store contraction result */
elemMat[Eoffset+idx+0] = E;

n = 1; E = 0.0; /* contract */
elemMat[Eoffset+idx+16] = E;

n = 2; E = 0.0; /* contract */
elemMat[Eoffset+idx+32] = E;

n = 3; E = 0.0; /* contract */
elemMat[Eoffset+idx+48] = E;
```c
void kernel(float *output) {
    float G[dim*dim] = {"3.0"*(dim*dim)};
    float K[dim*dim] = {"3.0"*(dim*dim)};
    float product = 0.0;
    const int Ooffset = gridIdx*numThreads;

    // Contract G and K
    for n in range(numLocalElements):
        for alpha in range(dim):
            for beta in range(dim):
                gIdx = (n*dim + alpha)*dim + beta
                kIdx = alpha*dim + beta
                product += G[gIdx] * K[kIdx];

    output[Ooffset+idx] = product;
}
```
We render code template into strings using a dictionary of inputs.

```python
args = {'dim': self.dim,
        'numLocalElements': 1,
        'numThreads': self.threadBlockSize}

kernelTemplate = self.getKernelTemplate()
gpuCode = kernelTemplate.render(isGPU = True, **args)
cpuCode = kernelTemplate.render(isGPU = False, **args)
```
__global__ void kernel(float *output) {
    const int gridIdx = blockIdx.x + blockIdx.y * blockDim.x; // This is (i,j)
    const int idx = threadIdx.x + threadIdx.y * 1;
    float G[9] = {3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0};
    float K[9] = {3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0};
    float product = 0.0;
    const int Ooffset = gridIdx * 1;

    // Contract G and K
    product += G[0] * K[0];
    product += G[1] * K[1];
    product += G[4] * K[4];
    product += G[5] * K[5];
    product += G[7] * K[7];
    product += G[8] * K[8];
    output[Ooffset + idx] = product;
    return;
}
void kernel(int numInvocations, float *output) {
    for(int gridIdx = 0; gridIdx < numInvocations; ++gridIdx) {
        for(int i = 0; i < 1; ++i) {
            for(int j = 0; j < 1; ++j) {
                const int idx = i + j*1; // This is (i,j)
                float G[9] = {3.0,3.0,3.0,3.0,3.0,3.0,3.0,3.0,3.0};
                float K[9] = {3.0,3.0,3.0,3.0,3.0,3.0,3.0,3.0,3.0};
                float product = 0.0;
                const int Ooffset = gridIdx*1;

                // Contract G and K
                product += G[0] * K[0];
                product += G[1] * K[1];
                product += G[4] * K[4];
                product += G[5] * K[5];
                product += G[7] * K[7];
                product += G[8] * K[8];
                output[Ooffset+idx] = product;
            }
        }
    }
    return;
}
Creating a Module

CPU:

```python
# Output kernel and C support code
self.outputKernelC(cpuCode)
self.writeMakefile()
out, err, status = self.executeShellCommand('make')
```

GPU:

```python
from pycuda.compiler import SourceModule

mod = SourceModule(gpuCode)
self.kernel = mod.get_function('kernel')
self.kernelReport(self.kernel, 'kernel')
```
Executing a Module

```python
import pycuda.driver as cuda
import pycuda.autoinit

blockDim = (self.dim, self.dim, 1)
start = cuda.Event()
end = cuda.Event()
grid = self.calculateGrid(N, numLocalElements)
start.record()
for i in range(iters):
    self.kernel(cuda.Out(output),
                block=blockDim, grid=grid)
end.record()
end.synchronize()
gpuTimes.append(start.time_till(end)*1e-3/iters)
```
Element matrix $K$ is now made up of small tensors

Contract all tensor elements with each the geometry tensor $G(T)$

\[
\begin{array}{cccccccc}
3 & 0 & 0 & -1 & 1 & 1 & -4 & -4 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-1 & 0 & 3 & 1 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 3 & 3 & -4 & 0 & 0 \\
1 & 0 & 1 & 3 & 3 & -4 & 0 & 0 \\
-4 & 0 & 0 & -4 & -4 & 8 & 4 & 0 \\
-4 & 0 & 0 & 0 & 0 & 4 & 8 & 0 \\
0 & 0 & 0 & 0 & -4 & 8 & 4 & 0 \\
4 & 0 & 0 & 0 & 0 & -4 & 8 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 8 & 4 \\
0 & 0 & 0 & 0 & 0 & 0 & 4 & 8 \\
\end{array}
\]
For $N$ elements, map blocks of $N_L$ elements to each Thread Block (TB)

- Launch grid must be $g_x \times g_y = \frac{N}{N_L}$
- TB grid will depend on the specific algorithm
- Output is size $N_{basis} \times N_{basis} \times N_L$

We can split a TB to work on multiple, $N_B$, elements at a time

- Note that each TB always gets $N_L$ elements, so $N_B$ must divide $N_L$
Mapping $G^{\alpha\beta}K^{ij}_{\alpha\beta}$ to the GPU

Kernel Arguments

```c
__global__
void integrateJacobian(float *elemMat,
                       float *geometry,
                       float *analytic)
```

- **geometry**: Array of $G$ tensors for each element
- **analytic**: $K$ tensor
- **elemMat**: Array of $E = G : K$ tensors for each element
We can interleave stores with computation, or wait until the end

- Waiting could improve coalescing of writes
- Interleaving could allow overlap of writes with computation

Also need to

- Coalesce accesses between global and local/shared memory
  (use `moveArray()`)
- Limit use of shared and local memory
Superior GPU memory bandwidth is due to both

bus width and clock speed.

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bus Width (bits)</td>
<td>64</td>
<td>512</td>
</tr>
<tr>
<td>Bus Clock Speed (MHz)</td>
<td>400</td>
<td>1600</td>
</tr>
<tr>
<td>Memory Bandwidth (GB/s)</td>
<td>3</td>
<td>102</td>
</tr>
<tr>
<td>Latency (cycles)</td>
<td>240</td>
<td>600</td>
</tr>
</tbody>
</table>

Tesla always accesses blocks of 64 or 128 bytes
Choose strategies to minimize reductions

- Only reductions occur in summation for contractions
  - Similar to the reduction in a quadrature loop

- **Strategy #1**: Each thread uses all of $K$

- **Strategy #2**: Do each contraction in a separate thread
Strategy #1
TB Division

Each thread computes an entire element matrix, so that

$$\text{blockDim} = \left(\frac{N_L}{N_B}, 1, 1\right)$$

We will see that there is little opportunity to overlap computation and memory access.
Strategy #1
Analytic Part

Read $K$ into shared memory (need to synchronize before access)

```c
__shared__ float K[${dim*dim*numBasisFuncs*numBasisFuncs}];

${fm.moveArray('K', 'analytic',
                   dim*dim*numBasisFuncs*numBasisFuncs, '', numThreads)}
__syncthreads();
```
Strategy #1
Geometry

- Each thread handles $N_B$ elements
- Read $G$ into local memory (not coalesced)
- Interleaving means writing after each thread does a single element matrix calculation

```python
float G[${dim}*{dim}*{numBlockElements}];

if (interleaved) {
    const int Goffset = (gridIdx*${numLocalElements} + idx)*{dim*dim};
    for n in range(numBlockElements):
        ${fm.moveArray('G', 'geometry', dim*dim, 'Goffset',
                       blockNumber = n*numLocalElements/numBlockElements,
                       localBlockNumber = n, isCoalesced = False)}
    endfor
} else {
    const int Goffset = (gridIdx*${numLocalElements/numBlockElements} + idx)*{dim*dim*{numBlockElements}};
    ${fm.moveArray('G', 'geometry', dim*dim*{numBlockElements}, 'Goffset',
                   isCoalesced = False)}
}
```
We write element matrices out contiguously by TB

```c
const int matSize = numBasisFuncs*numBasisFuncs;
const int Eoffset = gridIdx*matSize*numLocalElements;

if (interleaved) {
    const int elemOff = idx*matSize;
    __shared__ float E[matSize*numLocalElements/numBlockElements];
} else {
    const int elemOff = idx*matSize*numBlockElements;
    __shared__ float E[matSize*numLocalElements];
}
```
matSize = numBasisFuncs*numBasisFuncs

if interleaveStores:
    for b in range(numBlockElements):
        # Do 1 contraction for each thread
        __syncthreads();
        fm.moveArray('E', 'elemMat',
                     matSize*numLocalElements/numBlockElements,
                     'Eoffset', numThreads, blockNumber = n, isLoad = 0)

else:
    # Do numBlockElements contractions for each thread
    __syncthreads();
    fm.moveArray('E', 'elemMat',
                 matSize*numLocalElements,
                 'Eoffset', numThreads, isLoad = 0)
Each thread computes a single element of an element matrix, so that

$$\text{blockDim} = (N_{\text{basis}}, N_{\text{basis}}, N_B)$$

This allows us to overlap computation of another element in the TB with writes for the first.
Strategy #2

Analytic Part

- Assign an \((i,j)\) block of \(K\) to local memory
- \(N_B\) threads will simultaneously calculate a contraction

```cpp
const int Kidx = threadIdx.x + threadIdx.y * numBasisFuncs; // This is (i, j)
const int idx = Kidx + threadIdx.z * numBasisFuncs*numBasisFuncs;
const int Koffset = Kidx*dim*dim;
float K[dim*dim];

// for alpha in range(dim):
// for beta in range(dim):
<# kIdx = alpha*dim + beta #>
K[kIdx] = analytic[Koffset+kIdx];
#endfor
#endfor
```
Strategy #2
Geometry

- Store $N_L$ G tensors into shared memory
- Interleaving means writing after each thread does a single element calculation

```c
const int Goffset = gridIdx * dim*dim*numLocalElements;
__shared__ float G[dim*dim*numLocalElements];

$\text{fm.moveArray(}’G’, ’geometry’, \text{dim*dim*numLocalElements, ’Goffset’, numThreads)}$
__syncthreads();
```
Strategy #2

Output

- We write element matrices out contiguously by TB
- If interleaving stores, only need a single product
- Otherwise, need $N_L/N_B$, one per element processed by a thread

```c
const int matSize = numBasisFuncs*numBasisFuncs;
const int Eoffset = gridIdx*matSize*numLocalElements;

if (interleaved) {
    float product = 0.0;
    const int elemOff = idx*matSize;
} else {
    float product[numLocalElements/numBlockElements];
    const int elemOff = idx*matSize*numBlockElements;
}
```
if interleaveStores:
    for n in range(numLocalElements/numBlockElements):
        # Do 1 contraction for each thread
        __syncthreads()
        # Do coalesced write of element matrix
        elemMat[Eoffset+idx + n*numThreads] = product
else:
    # Do numLocalElements/numBlockElements contractions
    # save results in product[]
    for n in range(numLocalElements/numBlockElements):
        elemMat[Eoffset+idx + n*numThreads] = product[n]
Results
GTX 285

CPU vs. GPU Flop Rate for 3D $P_1$ Lagrange

Element Blocksize = 32
RESULTS

GTX 285

CPU vs. GPU Flop Rate for 3D $P_1$ Lagrange

Element Blocksize = 32
Interleaving stores
Results
GTX 285, 2 Simultaneous Elements

CPU vs. GPU Flop Rate for 3D $P_1$ Lagrange

Element Blocksize = 32
Results

GTX 285, 2 Simultaneous Elements

CPU vs. GPU Flop Rate for 3D $P_1$ Lagrange

Element Blocksize = 32
Interleaving stores
Results
GTX 285

CPU vs. GPU Flop Rate for 3D $P_1$ Lagrange

Element Blocksize = 64
Results

GTX 285

CPU vs. GPU Flop Rate for 3D $P_1$ Lagrange

Element Blocksize = 64
Interleaving stores
Results

GTX 285, 2 Simultaneous Elements

CPU vs. GPU Flop Rate for 3D $P_1$ Lagrange

Element Blocksize = 64
Results
GTX 285, 2 Simultaneous Elements

CPU vs. GPU Flop Rate for 3D $P_1$ Lagrange

Element Blocksize = 64
Interleaving stores
Results

GTX 285

CPU vs. GPU Flop Rate for 3D $P_1$ Lagrange

Element Blocksize = 128
Results
GTX 285

CPU vs. GPU Flop Rate for 3D $P_1$ Lagrange

Element Blocksize = 128
Interleaving stores
Results
GTX 285, 2 Simultaneous Elements

CPU vs. GPU Flop Rate for 3D $P_1$ Lagrange

Element Blocksize = 128
Results
GTX 285, 2 Simultaneous Elements

CPU vs. GPU Flop Rate for 3D $P_1$ Lagrange

Element Blocksize = 128
Interleaving stores
Results
GTX 285, 2 Simultaneous Elements

CPU vs. GPU Flop Rate for 3D $P_1$ Lagrange

Element Blocksize = 256
Results
GTX 285, 2 Simultaneous Elements

CPU vs. GPU Flop Rate for 3D $P_1$ Lagrange

Element Blocksize = 256
Interleaving stores
Results
GTX 285, 4 Simultaneous Elements

CPU vs. GPU Flop Rate for 3D $P_1$ Lagrange

Element Blocksize = 256
Results
GTX 285, 4 Simultaneous Elements

CPU vs. GPU Flop Rate for 3D $P_1$ Lagrange

Element Blocksize = 256
Interleaving stores
Performance

Influence of Element Batch Sizes

CPU vs. GPU Flop Rate for 3D $P_1$ Lagrange Laplacian

Interleave Stores = 1
Loop Unrolling = none
Performance
Influence of Element Batch Sizes

CPU vs. GPU Flop Rate for 2D $P_1$ Lagrange Elasticity

Interleave Stores = 1
Loop Unrolling = full
Performance
Influence of Code Structure

CPU vs. GPU Flop Rate for 3D $P_1$ Lagrange Laplacian

Element Blocksize = 128
Concurrent Elem = 2

M. Knepley ()
Robust
PASI '11 101 / 224
Performance
Influence of Code Structure

CPU vs. GPU Flop Rate for 2D $P_1$ Lagrange Elasticity

Element Blocksize = 256
Concurrent Elem = 2
Price-Performance Comparison of CPU and GPU 3D $P_1$ Laplacian Integration

<table>
<thead>
<tr>
<th>Model</th>
<th>Price ($)</th>
<th>GF/s</th>
<th>MF/s$</th>
</tr>
</thead>
<tbody>
<tr>
<td>GTX285</td>
<td>390</td>
<td>90</td>
<td>231</td>
</tr>
<tr>
<td>Core 2 Duo</td>
<td>300</td>
<td>2</td>
<td>6.6</td>
</tr>
</tbody>
</table>
Price-Performance Comparison of CPU and GPU 3D $P_1$ Laplacian Integration

<table>
<thead>
<tr>
<th>Model</th>
<th>Price ($)</th>
<th>GF/s</th>
<th>MF/s$</th>
</tr>
</thead>
<tbody>
<tr>
<td>GTX285</td>
<td>390</td>
<td>90</td>
<td>231</td>
</tr>
<tr>
<td>Core 2 Duo</td>
<td>300</td>
<td>12*</td>
<td>40</td>
</tr>
</tbody>
</table>

* Jed Brown Optimization Engine
2 GPU Computing
   • FEM-GPU
   • PETSc-GPU
**Thrust** is a CUDA library of parallel algorithms

- Interface similar to C++ Standard Template Library
- Containers *(vector, list, map)* on both host and device
- Algorithms: *sort, reduce, scan*
- Freely available, and part of PETSc configure *(--with-thrust-dir)*
**Cusp** is a CUDA library for sparse linear algebra and graph computations

- Builds on data structures in Thrust
- Provides sparse matrices in several formats (CSR, Hybrid)
- Includes some preliminary preconditioners (Jacobi, SA-AMG)
- Freely available, and part of PETSc configure (`-with-cusp-dir`)
Strategy: Define a new Vec implementation

- Uses Thrust for data storage and operations on GPU
- Supports full PETSc Vec interface
- Inherits PETSc scalar type
- Can be activated at runtime, `--vec_type cuda`
- PETSc provides memory coherence mechanism
Memory Coherence

PETSc Objects now hold a coherence flag

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PETSC_CUDA_UNALLOCATED</td>
<td>No allocation on the GPU</td>
</tr>
<tr>
<td>PETSC_CUDA_GPU</td>
<td>Values on GPU are current</td>
</tr>
<tr>
<td>PETSC_CUDA_CPU</td>
<td>Values on CPU are current</td>
</tr>
<tr>
<td>PETSC_CUDA_BOTH</td>
<td>Values on both are current</td>
</tr>
</tbody>
</table>

**Table**: Flags used to indicate the memory state of a PETSc CUDA Vec object.
Also define new **Mat** implementations

- Uses Cusp for data storage and operations on GPU
- Supports full PETSc **Mat** interface, some ops on CPU
- Can be activated at runtime, \(-\text{mat\_type aijcuda}\)
- Notice that parallel matvec necessitates off-GPU data transfer
Solvers

Solvers come for Free

- All linear algebra types work with solvers
- Entire solve can take place on the GPU
  - Only communicate scalars back to CPU
- GPU communication cost could be amortized over several solves
- Preconditioners are a problem
  - Cusp has a promising AMG
PETSc only needs

```sh
# Turn on CUDA
--with-cuda

# Specify the CUDA compiler
--with-cudac='nvcc -m64'

# Indicate the location of packages
# --download-* will also work
--with-thrust-dir=/PETSc3/multicore/thrust
--with-cusp-dir=/PETSc3/multicore/cusp

# Can also use double precision
--with-precision=single
```
Example
Driven Cavity Velocity-Vorticity with Multigrid

```
ex19  -da_vec_type seqcuda
    -da_mat_type aijcuda -mat_no_inode
    -da_grid_x 100 -da_grid_y 100
    -pc_type none -dmmg_nlevels 1
   -preload off -cuda_synchronize
   -log_summary

# Setup types
# Set grid size
# Setup solver
# Setup run
```
Outline

1. Tools and Infrastructure
2. GPU Computing
3. Linear Algebra and Solvers
   - Vector Algebra
   - Matrix Algebra
   - Algebraic Solvers
   - SNES
   - DA
   - PCFieldSplit
4. First Lab: Assembling a Code
5. Second Lab: Debugging and Performance Benchmarking
3 Linear Algebra and Solvers

- Vector Algebra
  - Matrix Algebra
  - Algebraic Solvers
  - SNES
  - DA
  - PCFieldSplit
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
A PETSc Vec

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

- Two phase assembly allows overlap of communication and computation
  - VecAssemblyBegin(Vec v)
  - VecAssemblyEnd(Vec v)
VecGetSize(x, &N);
MPI_Comm_rank(PETSC_COMM_WORLD, &rank);
if (rank == 0) {
    val = 0.0;
    for (i = 0; i < N; ++i) {
        VecSetValues(x, 1, &i, &val, INSERT_VALUES);
        val += 10.0;
    }
}

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

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>VecAXPY(Vec y, PetscScalar a, Vec x)</td>
<td>( y = y + a \times x )</td>
</tr>
<tr>
<td>VecAYPX(Vec y, PetscScalar a, Vec x)</td>
<td>( y = x + a \times y )</td>
</tr>
<tr>
<td>VecWAYPX(Vec w, PetscScalar a, Vec x, Vec y)</td>
<td>( w = y + a \times x )</td>
</tr>
<tr>
<td>VecScale(Vec x, PetscScalar a)</td>
<td>( x = a \times x )</td>
</tr>
<tr>
<td>VecCopy(Vec y, Vec x)</td>
<td>( y = x )</td>
</tr>
<tr>
<td>VecPointwiseMult(Vec w, Vec x, Vec y)</td>
<td>( w_i = x_i \times y_i )</td>
</tr>
<tr>
<td>VecMax(Vec x, PetscInt *idx, PetscScalar *r)</td>
<td>( r = \max r_i )</td>
</tr>
<tr>
<td>VecShift(Vec x, PetscScalar r)</td>
<td>( x_i = x_i + r )</td>
</tr>
<tr>
<td>VecAbs(Vec x)</td>
<td>( x_i =</td>
</tr>
<tr>
<td>VecNorm(Vec x, NormType type, PetscReal *r)</td>
<td>( r = |x| )</td>
</tr>
</tbody>
</table>

M. Knepley ()  
Robust  
PASI '11 120 / 224
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
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);
```c
#include "finclude/petsc.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

```c
#include "finclude/petsc.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)
```
```python
with v as a:
    for i in range(len(a)):
        a[i] = 5.0*i
```
3 Linear Algebra and Solvers
- Vector Algebra
- Matrix Algebra
- Algebraic Solvers
- SNES
- DA
- PCFieldSplit
What are PETSc matrices?

- 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
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

- MatSeqAIJPreallocation(Mat, PetscInt nz, const PetscInt nnz[])
- MatMPIAIJPreallocation(Mat, dnz, dnz[], onz, onz[])
- MatSetValues(Mat, m, rows[], n, cols[], values[], InsertMode)

  **MUST** be used, but does automatic communication
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
- Matrix-Free
- etc.

A matrix is defined by its interface, not by its data structure.
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 m, 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`
One Way to Set the Elements of a Matrix

Simple 3-point stencil for 1D Laplacian

\[ v[0] = -1.0; \quad v[1] = 2.0; \quad v[2] = -1.0; \]

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

\[ v[0] = -1.0; \ v[1] = 2.0; \ v[2] = -1.0; \]

MatGetOwnershipRange(A,&start,&end);

\textbf{for}
\begin{align*}
\text{row} = \text{start; } &\text{ row < end; row++ } \{ \\
\text{cols}[0] = \text{row-1}; \ &\text{cols}[1] = \text{row}; \ &\text{cols}[2] = \text{row+1}; \\
\textbf{if} \ (\text{row} == 0) \{ \\
\ &\text{MatSetValues}(A,1,\text{&row},2,\text{&cols}[1],\&v[1],\text{INSERT\_VALUES}); \\
\} \ \textbf{else if} \ (\text{row} == N-1) \{ \\
\ &\text{MatSetValues}(A,1,\text{&row},2,\text{cols},v,\text{INSERT\_VALUES}); \\
\} \ \textbf{else} \ { \\
\ &\text{MatSetValues}(A,1,\text{&row},3,\text{cols},v,\text{INSERT\_VALUES}); \\
\}
\}
\]

MatAssemblyBegin(A, MAT\_FINAL\_ASSEMBLY);

MatAssemblyEnd(A, MAT\_FINAL\_ASSEMBLY);
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)
Linear Algebra and Solvers

- Vector Algebra
- Matrix Algebra
- Algebraic Solvers
  - SNES
  - DA
  - PCFieldSplit
Solver Types

- **Explicit**: Field variables are updated using local neighbor information
- **Semi-implicit**: Some subsets of variables are updated with global solves, Others with direct local updates
- **Implicit**: Most or all variables are updated in a single global solve
Using PETSc linear algebra, just add:
- KSPSetOperators(KSP ksp, Mat A, Mat M, MatStructure flag)
- KSPSolve(KSP ksp, Vec b, Vec x)

Can access subobjects
- KSPGetPC(KSP ksp, PC *pc)

Preconditioners must obey PETSc interface
- Basically just the KSP interface

Can change solver dynamically from the command line
- -ksp_type bicgstab
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`
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
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)
Outline

3 Linear Algebra and Solvers
- Vector Algebra
- Matrix Algebra
- Algebraic Solvers
- **SNES**
  - DA
  - PCFieldSplit
Flow Control for a PETSc Application

Main Routine

Nonlinear Solvers (SNES)

Linear Solvers (KSP)

Preconditioners (PC)

PETSc

Application Initialization

Function Evaluation

Jacobian Evaluation

Postprocessing
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
Topology Abstractions

- **DA**
  - Abstracts Cartesian grids in any dimension
  - Supports stencils, communication, reordering
  - Nice for simple finite differences

- **Mesh**
  - Abstracts general topology in any dimension
  - Also supports partitioning, distribution, and global orders
  - Allows arbitrary element shapes and discretizations
Assembly Abstractions

- **DM**
  - Abstracts the logic of multilevel (multiphysics) methods
  - Manages allocation and assembly of local and global structures
  - Interfaces to DMMG solver

- **Section**
  - Abstracts functions over a topology
  - Manages allocation and assembly of local and global structures
  - Will merge with DM somehow
SNES Function

The user provided function which calculates the nonlinear residual has signature

\[
\text{PetscErrorCode} \ (\ast \text{func})(\text{SNES} \ \text{snes}, \text{Vec} \ x, \text{Vec} \ r, \text{void} \ *\text{ctx})
\]

- \text{x}: The current solution
- \text{r}: The residual
- \text{ctx}: The user context passed to \text{SNESSetFunction()}
  - Use this to pass application information, e.g. physical constants
**SNES Jacobian**

The user provided function which calculates the Jacobian has signature

\[ (\ast \text{func})(\text{SNES } \text{snes}, \text{Vec } \text{x}, \text{Mat } \ast \text{J}, \text{Mat } \ast \text{M}, \text{MatStructure } \ast \text{flag}, \text{void } \ast \text{ctx}) \]

- **x**: The current solution
- **J**: The Jacobian
- **M**: The Jacobian preconditioning matrix (possibly J itself)
- **ctx**: The user context passed to `SNESSetJacobian()`
  - Use this to pass application information, e.g. physical constants
  - Possible `MatStructure` values are:
    - `SAME_NONZERO_PATTERN`
    - `DIFFERENT_NONZERO_PATTERN`

Alternatively, you can use
- a builtin sparse finite difference approximation
- automatic differentiation (ADIC/ADIFOR)
SNES Variants

- Line search strategies
- Trust region approaches
- Pseudo-transient continuation
- Matrix-free variants
PETSc can compute and explicitly store a Jacobian via 1st-order FD

- **Dense**
  - Activated by `-snes_fd`
  - Computed by `SNESDefaultComputeJacobian()`

- **Sparse via colorings**
  - 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 Example
Driven Cavity

Solution Components

velocity: u
velocity: v
vorticity:
temperature: T

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

$PETCS_DIR/src/snes/examples/tutorials/ex19.c
typedef struct {
    /*----- basic application data -----*/
    double lid_velocity;
    double prandtl, grashof;
    int mx, my;
    int mc;
    PetscTruth draw_contours;
    /*----- parallel data -----*/
    MPI_Comm comm;
    DA da;
    /* Local ghosted solution and residual */
    Vec localX, localF;
} AppCtx;
Residual(SNES snes, Vec X, Vec F, void *ptr) {
    AppCtx *user = (AppCtx *) ptr;

    /* local starting and ending grid points */
    int istart, iend, jstart, jend;
    PetscScalar *f; /* local vector data */
    PetscReal grashof = user->grashof;
    PetscReal prandtl = user->prandtl;
    PetscErrorCode ierr;

    /* Code to communicate nonlocal ghost point data */
    VecGetArray(F, &f);

    /* Code to compute local function components */
    VecRestoreArray(F, &f);
    return 0;
}
Better Driven Cavity Residual Evaluation

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

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

M. Knepley ()
Outline

3 Linear Algebra and Solvers
- Vector Algebra
- Matrix Algebra
- Algebraic Solvers
- SNES
- DA
- PCFieldSplit
**DA** is a topology interface handling parallel data layout on structured grids

- Handles local and global indices
  - `DAGetGlobalIndices()` and `DAGetAO()`
- Provides local and global vectors
  - `DAGetGlobalVector()` and `DAGetLocalVector()`
- Handles ghost values coherence
  - `DAGetGlobalToLocal()` and `DAGetLocalToGlobal()`
The DA interface is based upon local callback functions

- \texttt{FormFunctionLocal()}, \textbf{set by} \texttt{DASetLocalFunction()}
- \texttt{FormJacobianLocal()}, \textbf{set by} \texttt{DASetLocalJacobian()}

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

- Each process evaluates the local residual
- PETSc assembles the global residual automatically
  - \textbf{Uses} \texttt{DALocalToGlobal()} method
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
### DA Global Numberings

<table>
<thead>
<tr>
<th>Proc 2</th>
<th>Proc 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>25 26 27</td>
<td>28 29</td>
</tr>
<tr>
<td>20 21 22</td>
<td>23 24</td>
</tr>
<tr>
<td>15 16 17</td>
<td>18 19</td>
</tr>
<tr>
<td>10 11 12</td>
<td>13 14</td>
</tr>
<tr>
<td>5  6  7</td>
<td>8  9</td>
</tr>
<tr>
<td>0  1  2</td>
<td>3  4</td>
</tr>
</tbody>
</table>

#### Natural numbering

<table>
<thead>
<tr>
<th>Proc 0</th>
<th>Proc 1</th>
</tr>
</thead>
</table>

### PETSc numbering

<table>
<thead>
<tr>
<th>Proc 2</th>
<th>Proc 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>21 22 23</td>
<td>28 29</td>
</tr>
<tr>
<td>18 19 20</td>
<td>26 27</td>
</tr>
<tr>
<td>15 16 17</td>
<td>24 25</td>
</tr>
<tr>
<td>6  7  8</td>
<td>13 14</td>
</tr>
<tr>
<td>3  4  5</td>
<td>11 12</td>
</tr>
<tr>
<td>0  1  2</td>
<td>9  10</td>
</tr>
</tbody>
</table>

| Proc 0 | Proc 1 |

M. Knepley () Robust PASI ’11 157 / 224
**Global**: Each vertex has a unique id belongs on a unique process

**Local**: Numbering includes vertices from neighboring processes

- These are called *ghost* vertices

### Local numbering

<table>
<thead>
<tr>
<th></th>
<th>Proc 0</th>
<th>Proc 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

### Global numbering

<table>
<thead>
<tr>
<th></th>
<th>Proc 0</th>
<th>Proc 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>
DA Local Function

The user provided function which calculates the nonlinear residual in 2D has signature

\[(*lfunc)(DALocalInfo *info, PetscScalar **x, PetscScalar **r, void *ctx)\]

**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 DA function is activated by calling

`SNESSetFunction(snes, r, SNESDADAFormFunction, ctx)`
Bratu Residual Evaluation

\[ \Delta u + \lambda e^u = 0 \]

```c
ResLocal(DA_Local_Info *info, Field **x, Field **f, void *ctx) {
    /* Not Shown: Handle boundaries */
    /* Compute over the interior points */
    for (j = info->ys; j < info->xs + info->ym; j++) {
        for (i = info->xs; i < info->ys + info->xm; i++) {
            u = x[j][i];
            u_xx = (2.0 * u - x[j][i-1] - x[j][i+1]) * hydhx;
            u_yy = (2.0 * u - x[j-1][i] - x[j+1][i]) * hxdhy;
            f[j][i] = u_xx + u_yy - hx*hy*lambda*exp(u);
        }
    }
}
```

$PETCS\_DIR/src/snes/examples/tutorials/ex5.c

M. Knepley ()  Robust  PASI '11  160 / 224
The user provided function which calculates the Jacobian in 2D has signature

\[ (*lfunc)(DALocalInfo *info, PetscScalar **x, Mat J, void *ctx) \]

- **info:** All layout and numbering information
- **x:** The current solution
- **J:** The Jacobian
- **ctx:** The user context passed to \texttt{DASetLocalJacobian}()

The local DA function is activated by calling

\[ \text{SNESSetJacobian(snes, } J, J, \text{ SNESDADComputeJacobian, ctx)} \]
Bratu Jacobian Evaluation

JacLocal(DALocalInfo *info, PetscScalar **x, Mat jac, void *ctx) {
    for (j = info->ys; j < info->ys + info->ym; j++) {
        for (i = info->xs; i < info->xs + info->xm; i++) {
            row.j = j; row.i = i;
            if (i == 0 || j == 0 || i == mx-1 || j == my-1) {
                v[0] = 1.0;
                MatSetValuesStencil(jac, 1, &row, 1, &row, v, INSERT_VALUES);
            } else {
                v[0] = -(hx/hy); col[0].j = j-1; col[0].i = i;
                v[1] = -(hy/hx); col[1].j = j; col[1].i = i-1;
                v[2] = 2.0*(hy/hx+hx/hy)
                    - hx*hy*lambda*PetscExpScalar(x[j][i]);
                v[3] = -(hy/hx); col[3].j = j; col[3].i = i+1;
                v[4] = -(hx/hy); col[4].j = j+1; col[4].i = i;
                MatSetValuesStencil(jac, 1, &row, 5, col, v, INSERT_VALUES);
            }
        }
    }
}}

$PETCS_DIR/src/snes/examples/tutorials/ex5.c
A DA contains topology, geometry, and an implicit Q1 discretization.

It is used as a template to create

- Vectors (functions)
- Matrices (linear operators)
DA Vectors

The DA 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
  - `DACreateGlobalVector(DA da, Vec *gvec)`

Local vectors are sequential (and usually temporary)
  - Each process stores its local portion plus ghost values
  - `DACreateLocalVector(DA da, Vec *lvec)`
  - includes ghost values!
Two-step process enables overlapping computation and communication

- DAGlobalToLocalBegin(da, gvec, mode, lvec)
  - gvec provides the data
  - mode is either INSERT_VALUES or ADD_VALUES
  - lvec holds the local and ghost values

- DAGlobalToLocalEnd(da, gvec, mode, lvec)
  - Finishes the communication

The process can be reversed with DALocalToGlobal().
Both the **box** stencil and **star** stencil are available.
Setting Values on Regular Grids

PETSc provides

```c
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
Creating a DA

```c
DACreate2d(comm, wrap, type, M, N, m, n, dof, s, lm[], ln[], DA *da)
```

- **wrap**: Specifies periodicity
  - DA_NONPERIODIC, DA_XPERIODIC, DA_YPERIODIC, or DA_XYPERIODIC

- **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
Outline

3 Linear Algebra and Solvers
- Vector Algebra
- Matrix Algebra
- Algebraic Solvers
- SNES
- DA
- PCFieldSplit
The **PCFieldSplit** interface

- extracts functions/operators corresponding to each physics
  - *VecScatter* and *MatGetSubmatrices()* for efficiency
- assemble functions/operators over all physics
  - Generalizes *LocalToGlobal()* mapping
- is composable with **ANY** PETSc solver and preconditioner
  - This can be done recursively
MultiPhysics Paradigm

The **PCFieldSplit** interface

- extracts functions/operators corresponding to each physics
  - *VecScatter* and *MatGetSubmatrices()* for efficiency

- assemble functions/operators over all physics
  - Generalizes *LocalToGlobal()* mapping

- is composable with ANY PETSc solver and preconditioner
  - This can be done recursively

FieldSplit provides the **buildings blocks**
for multiphysics preconditioning.
MultiPhysics Paradigm

The **PCFieldSplit** interface

- extracts functions/operators corresponding to each physics
  - **VecScatter** and **MatGetSubmatrices()** for efficiency
- assemble functions/operators over all physics
  - Generalizes **LocalToGlobal()** mapping
- is composable with **ANY** PETSc solver and preconditioner
  - This can be done recursively

Notice that this works in exactly the same manner as

- multiple resolutions (MG, FMM, Wavelets)
- multiple domains (Domain Decomposition)
- multiple dimensions (ADI)
Preconditioning

Several varieties of preconditioners can be supported:
- Block Jacobi or Block Gauss-Siedel
- Schur complement
- Block ILU (approximate coupling and Schur complement)
- Dave May’s implementation of Elman-Wathen type PCs which only require actions of individual operator blocks

Notice also that we may have any combination of
- “canned” PCs (ILU, AMG)
- PCs needing special information (MG, FMM)
- custom PCs (physics-based preconditioning, Born approximation) since we have access to an algebraic interface
Outline

1. Tools and Infrastructure
2. GPU Computing
3. Linear Algebra and Solvers
4. First Lab: Assembling a Code
   - Beginning with Mercurial
   - Setting up Configure and Build
   - Running a PETSc Python example
   - Running a PETSc solver on the GPU
   - Adding a GPU kernel
5. Second Lab: Debugging and Performance Benchmarking
First Lab: Assembling a Code

- Beginning with Mercurial
- Setting up Configure and Build
- Running a PETSc Python example
- Running a PETSc solver on the GPU
- Adding a GPU kernel
Create a clone

hg clone http://petsc.cs.iit.edu/tutorials/PASI

or

from mercurial.dispatch import dispatch

args = ['clone',
        'http://petsc.cs.iit.edu/petsc/tutorials/vecfem']
dispatch(args)
Make a change

# Make sure we are up-to-date
hg pull -u
# Create a file
touch TODO
# Schedule the file for version control
hg add TODO
# Commit to a ChangeSet
hg ci -m "Added TODO list" TODO
# Edit file
echo "Get configure working" >> TODO
# Commit edit
hg ci -m "Added first task"
# Push change to the master repository
hg push
Make a bugfix

# Create new repository
hg clone -r1.1 vecfem vecfem-bugfix
cd vecfem-bugfix

# Fix bug
hg ci -m "Fixed bug"

# Merge in changes from Master
hg pull --rebase

# Push bugfix
hg push
# Create new repository
hg clone -r1.1 vecfem vecfem-bugfix
cd vecfem-bugfix

# Fix bug
hg ci -m "Fixed bug"

# Merge in changes from Master
hg pull
hg merge
hg ci -m "Merge"

# Push bugfix
hg push
First Lab: Assembling a Code

- Beginning with Mercurial
- Setting up Configure and Build
- Running a PETSc Python example
- Running a PETSc solver on the GPU
- Adding a GPU kernel
Basic Configuration

http://petsc.cs.iit.edu/petsc/SimpleConfigure provides basic configure support

1. Choose `projectName`
2. Copy in:
   - `configure.py` to the root
   - `config/Matt/*` to `config/projectName/`
3. Change main call in `configure.py`:
   ```python
   ConfigurationManager(projectName).configure([])
   ```
4. Change project name in `config/projectName/Configure.py`
We can add support for multiple builds using `--arch`

class ProjectArch(object):
    def __init__(self, argDB):
        self.argDB = argDB
    @property
def arch(self):
        return self.argDB['arch']

changing the member variable in `__init__()` to

    self.arch = ProjectArch(self.argDB)

and adding an option to `setupHelp()`

    help.addArgument(self.Project, '-arch=<name>', 
                     nargs.Arg(None, 'debug', 'The name of this build'))
Basic Configuration

Arch

We can add support for multiple builds using `--arch`

```python
class ProjectArch(object):
    def __init__(self, argDB):
        self.argDB = argDB

    @property
    def arch(self):
        return self.argDB['arch']

changing the member variable in `__init__() to

    self.arch = ProjectArch(self.argDB)

and adding an option to `setupHelp()`

    help.addArgument(self.Project, '-arch=<name>',
                     nargs.Arg(None,'debug','The name of this build'))
```
Basic Configuration

Arch

We can add support for multiple builds using `--arch`

class ProjectArch(object):
    def __init__(self, argDB):
        self.argDB = argDB
    @property
    def arch(self):
        return self.argDB['arch']

changing the member variable in `__init__() to

    self.arch = ProjectArch(self.argDB)

and adding an option to `setupHelp()`

    help.addArgument(self.Project, '-arch=<name>',
                     nargs.Arg(None,'debug','The name of this build'))
We can add support for builds using PETSc make rules

CFLAGS = -I.
ARCHFLAGS = -arch x86_64

meshObjs: sieveMesh.o
ARCHFLAGS="$\{ARCHFLAGS\}"
python setupSieve.py build_ext --inplace

ex1: ex1.o

\$\{CLINKER\} -o ex1 ex1.o \$\{PETSC_LIB\}
We need a `setupSieve.py` to build an extension module

```python
import os, numpy
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

ext_modules = [Extension('petscSieve', ['petscSieve.pyx'],
    extra_objects = ['sieveMesh.o'],
    library_dirs = [os.path.join(os.environ['PETSC_DIR'], os.environ['PETSC_ARCH'], 'lib'),
                    '/usr/X11/lib',
                    '/System/Library/Frameworks/Accelerate.framework/Versions/A/Frameworks/vecLib.framework/Versions/A'],
    libraries = ['petsc', 'X11', 'LAPACK', 'BLAS', 'pmpich', 'mpich', 'opa',
                 include_dirs = [numpy.get_include()],
    language = 'c++')]

setup(
    name = 'PETSc Sieve converter',
    cmdclass = {'build_ext': build_ext},
    ext_modules = ext_modules
)
```
First Lab: Assembling a Code

- Beginning with Mercurial
- Setting up Configure and Build
- Running a PETSc Python example
- Running a PETSc solver on the GPU
- Adding a GPU kernel
# Update repository to new code
hg update -r2
# Execute empty run
python vecfem.py
# We can set verbosity level
./vecfem.py --verbose=2
# Python 3 should fail gracefully
python3 vecfem.py
hg update -r5
# Monitor solve and reason for termination
./vecfem.py -ksp_monitor -ksp_converged_reason
# Look at solve configuration and performance data
./vecfem.py -ksp_view -log_summary
# Can put performance data in a module
./vecfem.py -log_summary_python logSummary.py
Add Empty Vec, Mat, and KSP
Structured Mesh

hg update -r7
# Solve system to high tolerance
./vecfem.py -ksp_monitor -ksp_converged_reason -ksp_rtol 1.0e-9
# Output the computed solution
# Notice that the constant mode is removed
./vecfem.py -ksp_monitor -ksp_converged_reason -ksp_rtol 1.0e-9 --verbose
# We can change the size
./vecfem.py -ksp_monitor -ksp_converged_reason -ksp_rtol 1.0e-9 --sizes=
# Output intermediate objects
./vecfem.py -ksp_monitor -ksp_converged_reason -ksp_rtol 1.0e-9 --verbose
First Lab: Assembling a Code

- Beginning with Mercurial
- Setting up Configure and Build
- Running a PETSc Python example
- Running a PETSc solver on the GPU
- Adding a GPU kernel
hg update -r8

# Specify CUDA types for linear algebra
# Solve will automatically happen on GPU
./vecfem.py -ksp_monitor -ksp_converged_reason \
- ksp_rtol 1.0e-9 \
- da_vec_type cuda - da_mat_type aijcuda
First Lab: Assembling a Code

- Beginning with Mercurial
- Setting up Configure and Build
- Running a PETSc Python example
- Running a PETSc solver on the GPU
- Adding a GPU kernel
Updates

- Configure Sieve extension module
- Added Python modules for:
  - Mesh handling (Sieve)
  - Geometry
  - Discretization
  - Integration
  - Computational Modeling
  - Kernel Execution
- Extension module for Sieve has:
  - C wrapper for C++ methods
  - Cython wrapper for C methods
  - Python build script
- Added unstructured mesh support
**GPU Kernel**

- `femIntegration` makes kernel using templating
- PyCUDA creates module and launches
- Cython bridges gap with C support libraries
- numpy is used to transfer data
Outline

1. Tools and Infrastructure
2. GPU Computing
3. Linear Algebra and Solvers
4. First Lab: Assembling a Code

5. Second Lab: Debugging and Performance Benchmarking
   - Debugging
   - Performance Benchmarking
Outline

Second Lab: Debugging and Performance Benchmarking

- Debugging
- Performance Benchmarking
Valgrind

Valgrind is a debugging framework

- **Memcheck**: Check for memory overwrite and illegal use
- **Callgrind**: Generate call graphs
- **Cacheegrind**: 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

```bash
# Get the tutorial repository
hg clone http://petsc.cs.iit.edu/petsc/tutorials/SimpleTutorial
hg update -r2
# 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 const*, void**) (mal.c:37)
==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==  Address 0x18 is not stack'd, malloc'd or (recently) free'd
```

```
==13697==  Use of uninitialised value of size 8
==13697==  at 0x10000529D: MyInitialGuess(AppCtx *, _p_Vec*) (myStuff.c:45)
==13697==  by 0x100004447: main (ex5.c:202)
==13697==  Address 0x6f5c300000018 is not stack'd, malloc'd or (recently) free'd
```

```
==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 0x6f5c300000018 is not stack'd, malloc'd or (recently) free'd
```
We can fix the error by using *ghosted* coordinates

```bash
hg update -r3
make
valgrind --trace-children=yes --suppressions=bin/simple.supp $PETSC_DIR/
```
A *suppressions* file suppresses errors

It can be generated automatically

```
valgrind --trace-children=yes --gen-suppressions=all ./vecfem.py -ksp_rtol
```

and we put them into `vecfem.supp`, so that

```
hg -r9 update
valgrind --trace-children=yes --suppressions=vecfem.supp ./ex2
```

produces no errors.
Valgrind

# 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

- Automatic generation of tracebacks
- Detecting memory corruption and leaks
- Optional user-defined error handlers
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`
Debugging Tips

- Put a breakpoint in PetscError() to catch errors as they occur.
- PETSc tracks memory overwrites at both ends of arrays.
  - The CHKMEMQ macro causes a check of all allocated memory.
  - Track memory overwrites by bracketing them with CHKMEMQ.
- PETSc checks for leaked memory.
  - Use PetscMalloc() and PetscFree() for all allocation.
  - Print unfreed memory on PetscFinalize() with -malloc_dump.
- Simply the best tool today is valgrind.
  - It checks memory access, cache performance, memory usage, etc.
  - http://www.valgrind.org
  - Need -trace-children=yes when running under MPI.
Exercise 7

Use the debugger to find a SEGV
Locate a memory overwrite using CHKMEMQ.

- Get the example
  - `hg clone -r1 http://petsc.cs.iit.edu/petsc/SimpleTutorial`

- Build the example `make`

- Run it and watch the fireworks
  - `mpiexec -n 2 ./bin/ex5 -use_coords`

- Run it under the debugger and correct the error
  - `mpiexec -n 2 ./bin/ex5 -use_coords -start_in_debugger -display :0.0`
  - `hg update -r2`

- Build it and run again smoothly
Second Lab: Debugging and Performance Benchmarking

Outline

- Debugging
- Performance Benchmarking
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
Using Stages and Events

- **Use** `PetscLogStageRegister()` to create a new stage
  - Stages are identified by an integer handle
- **Use** `PetscLogStagePush/Pop()` to manage stages
  - Stages may be nested, but will not aggregate in a nested fashion
- **Use** `PetscLogEventRegister()` to create a new stage
  - Events also have an associated class
- **Use** `PetscLogEventBegin/End()` to manage events
  - Events may also be nested and will aggregate in a nested fashion
  - **Can use** `PetscLogFlops()` to log user flops
Adding A Logging Stage

```c
int stageNum;

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

/* Code to Monitor */

PetscLogStagePop();
```
Adding A Logging Stage
Python

```python
with PETSc.LogStage('Fluid Stage') as fluidStage:
    # All operations will be aggregated in fluidStage
fluid.solve()
```
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);
```
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)
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()
  - MatMPIAIJSetPreallocation()
  - Only the relevant data will be used
Matrix Memory Preallocation
Sequential Sparse Matrices

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

nz: expected number of nonzeros in any row

nnz(i): expected number of nonzeros in row i
Matrix Memory Preallocation

ParallelSparseMatrix

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

- **dnz**: expected number of nonzeros in any row in the diagonal block
- **nnz(i)**: expected number of nonzeros in row i in the diagonal block
- **onz**: expected number of nonzeros in any row in the offdiagonal portion
- **nnz(i)**: expected number of nonzeros in row i in the offdiagonal portion
Matrix Memory Preallocation

Verifying Preallocation

- Use runtime option `-info`
- Output:

  ```
  [proc #] Matrix size: %d X %d; storage space:
  %d unneeded, %d used
  [proc #] Number of mallocs during MatSetValues() is %d
  ```

```
[merlin] mpirun ex2 -log_info
[0]MatAssemblyEnd_SeqAIJ:Matrix size: 56 X 56; storage space:
[0] 310 unneeded, 250 used
[0]MatAssemblyEnd_SeqAIJ: Number of mallocs during MatSetValues() is 0
[0]MatAssemblyEnd_SeqAIJ: Most nonzeros in any row is 5
[0]Mat_AIJ_CheckInode: Found 56 nodes out of 56 rows. Not using Inode routine
[0]Mat_AIJ_CheckInode: Found 56 nodes out of 56 rows. Not using Inode routine
Norm of error 0.000156044 iterations 6
[0]PetscFinalize:PETSc successfully ended!
```
Exercise 8

Return to Exercise 7 and add more profiling.

- Update to the next revision
  - `hg update -r3`
- Build, run, and look at the profiling report
  - `make ex5`
  - `./bin/ex5 -use_coords -log_summary`
- Add a new stage for setup
- Add a new event for `FormInitialGuess()` and log the flops
- Build it again and look at the profiling report
Scalability is not Efficiency

Scalability is easy

Efficiency is hard
Scalability is not Efficiency

Scalability is easy

Efficiency is hard
Scalability is not Efficiency

Scalability is easy

Efficiency is hard
**Scalability**

**Def:** Computation, Communication, and Memory are in $O(N)$

- Can also demand $O(P)$
- Watch out for hidden constants
  - $6N$ and $6000N$ are both scalable
PDEs are scalable

- Computations are local
  - abstract data types, e.g. Mat

- Communication is nearest neighbor
  - parallel data structures, e.g. DA

- Referenced arbitrary unknowns
  - GlobalToLocalMapping
  - DA, Mesh, VecScatter
PDEs are scalable

- Computations are local
  - abstract data types, e.g. Mat
- Communication is nearest neighbor
  - parallel data structures, e.g. DA
- Referenced arbitrary unknowns
  - GlobalToLocalMapping
  - DA, Mesh, VecScatter
PDEs are scalable

- Computations are local
  - abstract data types, e.g. Mat
- Communication is nearest neighbor
  - parallel data structures, e.g. DA
- Referenced arbitrary unknowns
  - GlobalToLocalMapping
  - DA, Mesh, VecScatter
PDEs are scalable unless you screw something up

- Prescribed data structures
  - abstract data types, e.g. Mat
- Fully replicated data structures
  - parallel data structures, e.g. DA
- Referenced arbitrary unknowns
  - GlobalToLocalMapping
  - DA, Mesh, VecScatter
PDEs are scalable unless you screw something up

Mistakes:

- Prescribed data structures
  - abstract data types, e.g. Mat
- Fully replicated data structures
  - parallel data structures, e.g. DA
- Referenced arbitrary unknowns
  - GlobalToLocalMapping
  - DA, Mesh, VecScatter
PDEs are scalable unless you screw something up

Mistakes:

- Prescribed data structures
  - abstract data types, e.g. Mat
- Fully replicated data structures
  - parallel data structures, e.g. DA
- Referenced arbitrary unknowns
  - GlobalToLocalMapping
  - DA, Mesh, VecScatter
PDEs are scalable unless you screw something up

Mistakes:
- Prescribed data structures
  - abstract data types, e.g. Mat
- Fully replicated data structures
  - parallel data structures, e.g. DA
- Referenced arbitrary unknowns
  - GlobalToLocalMapping
  - DA, Mesh, VecScatter
PDEs are scalable unless you screw something up

Mistakes:
- Prescribed data structures
  - abstract data types, e.g. Mat
- Fully replicated data structures
  - parallel data structures, e.g. DA
- Referenced arbitrary unknowns
  - GlobalToLocalMapping
  - DA, Mesh, VecScatter
Integral equations can be scalable

- But, they couple all unknowns
- Need special algorithms
  - Fast Fourier Transform
  - Fast Multipole Method
  - Fast Wavelet Transform
Integral Equations

Integral equations can be scalable

- But, they couple all unknowns

- Need special algorithms
  - Fast Fourier Transform
  - Fast Multipole Method
  - Fast Wavelet Transform
Integral Equations

Integral equations can be scalable

- But, they couple all unknowns

- Need special algorithms
  - Fast Fourier Transform
  - Fast Multipole Method
  - Fast Wavelet Transform
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
The key performance indicator, which we will call the *balance factor* $\beta$, is the ratio of *flops* executed to *bytes* transferred.

- We will designate the unit $\frac{\text{flop}}{\text{byte}}$ as the *Keyes*.
- Using the peak flop rate $r_{\text{peak}}$, we can get the required bandwidth $B_{\text{req}}$ for an algorithm
  \[ B_{\text{req}} = \frac{r_{\text{peak}}}{\beta} \]  
  \[ (20) \]
- Using the peak bandwidth $B_{\text{peak}}$, we can get the maximum flop rate $r_{\text{max}}$ for an algorithm
  \[ r_{\text{max}} = \beta B_{\text{peak}} \]  
  \[ (21) \]
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):

  $$c_1 = c_1 + a_1 \ast b_1$$
  $$c_2 = c_2 + a_2 \ast b_2$$

  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.
Analysis of BLAS \texttt{axpy()}\

\[
\vec{y} \leftarrow \alpha \vec{x} + \vec{y}
\]

For vectors of length $N$ and $b$-byte numbers, we have

- **Computation**
  - $2N$ flops

- **Memory Access**
  - $(3N + 1)b$ bytes

Thus, our balance factor $\beta = \frac{2N}{(3N+1)b} \approx \frac{2}{3b}$ Keyes
Analysis of BLAS \texttt{axpy()} \\

\[
\vec{y} \leftarrow \alpha \vec{x} + \vec{y}
\]

For Matt’s Laptop,

- \( r_{\text{peak}} = 1700 \text{MF/s} \)
  - implies that

- \( B_{\text{req}} = 2550b \text{ MB/s} \)
  - Much greater than \( B_{\text{peak}} \)

- \( B_{\text{peak}} = 1122 \text{MB/s} \)
  - implies that

- \( r_{\text{max}} = \frac{748}{b} \text{ MF/s} \)
  - 5.5\% of \( r_{\text{peak}} \)
STREAM Benchmark

Simple benchmark program measuring **sustainable** memory bandwidth

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

<table>
<thead>
<tr>
<th>Machine</th>
<th>Peak (MF/s)</th>
<th>Triad (MB/s)</th>
<th>MF/MW</th>
<th>Eq. MF/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matt’s Laptop</td>
<td>1700</td>
<td>1122.4</td>
<td>12.1</td>
<td>93.5 (5.5%)</td>
</tr>
<tr>
<td>Intel Core2 Quad</td>
<td>38400</td>
<td>5312.0</td>
<td>57.8</td>
<td>442.7 (1.2%)</td>
</tr>
<tr>
<td>Tesla 1060C</td>
<td>984000</td>
<td>102000.0*</td>
<td>77.2</td>
<td>85000.0 (0.8%)</td>
</tr>
</tbody>
</table>

**Table:** Bandwidth limited machine performance

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

$$\left(8 + \frac{2}{V}\right) \frac{m}{nz} + \frac{6}{V} \text{ byte/flop} \quad (24)$$

or achievable performance given a bandwidth $BW$

$$\frac{Vnz}{(8V + 2)m + 6nz} BW \text{ Mflop/s} \quad (25)$$

Towards Realistic Performance Bounds for Implicit CFD Codes, Gropp, Kaushik, Keyes, and Smith.
For a single matvec with 3D FD Poisson, Matt’s laptop can achieve at most

\[
\frac{1}{(8 + 2)^{\frac{1}{7}} + 6} \text{ bytes/flop}(1122.4 \text{ MB/s}) = 151 \text{ MFlops/s, (26)}
\]

which is a dismal 8.8% of peak.

Can improve performance by

- Blocking
- Multiple vectors

but operation issue limitations take over.
Improving Serial Performance

For a single matvec with 3D FD Poisson, Matt’s laptop can achieve at most

\[
\frac{1}{(8 + 2)^{\frac{1}{7}} + 6} \text{ bytes/flop}(1122.4 \text{ MB/s}) = 151 \text{ MFlops/s}, \quad (26)
\]

which is a dismal 8.8% of peak.

Better approaches:
- Unassembled operator application (Spectral elements, FMM)
  - \(N\) data, \(N^2\) computation
- Nonlinear evaluation (Picard, FAS, Exact Polynomial Solvers)
  - \(N\) data, \(N^k\) computation
We must balance storage, bandwidth, and cycles

- **Assembled Operator Action**
  - Trades cycles and storage for bandwidth in application

- **Unassembled Operator Action**
  - Trades bandwidth and storage for cycles in application
  - For high orders, storage is impossible
  - Can make use of FErari decomposition to save calculation
  - Could storage element matrices to save cycles

- **Partial assembly gives even finer control over tradeoffs**
  - Also allows introduction of parallel costs (load balance, ...)

M. Knepley ()

Robust

PASI '11