PETSc4Py

Python in HPC

TACC Training, Oct. 15, 2012

Presenters:
Andy R. Terrel, PhD
Texas Advanced Computing Center
University of Texas at Austin

Yaakoub El Khamra
Texas Advanced Computing Center
University of Texas at Austin

0.1 Interacting with the Tutorial Slides

This tutorial is an interactive worksheet designed to encourage you to try out the lessons during the demonstration. If you are looking at the pdf version, we encourage you to download the updated version (see previous slide) and try the interactive version.

To run the interactive version, you need a good Python environment including:

- IPython version >= 13.0
- Numpy version >= 1.5
- Scipy
- Matplotlib

Move to the directory containing the tarball and execute:

```
$ ipython notebook --pylab=inline
```

We heartily endorse the Anaconda distribution and the Free Enthought Python Distribution.

0.2 Presentation mode

The slide show mode is only supported by an IPython development branch version. To get it I recommend cloning from the official branch, adding Matthias Carreau's remote, fetching and using his branch slideshow_extension2. Here are the commands:

```
git clone git://github.com/ipython/ipython.git # Official clone
cd ipython
git remote add carreau git://github.com/Carreau/ipython.git # Matthias' branch
git fetch carreau # Fetch the branches
git checkout carreau/slideshow_extension2 # Checkout the slideshow extension
python setup.py develop # Install the development version
ipython notebook # Check out the slideshows.
```

0.3 Acknowledgements

- petsc4py examples developed by Lisandro Dalcin
- petsc4py petsc4py are Python bindings for PETSc, the Portable, Extensible Toolkit for Scientific Computation.

1 What is PETSc

“PETSc. . . is a suite of data structures and routines for the scalable (parallel) solution of scientific applications modeled by partial differential equations. It supports MPI, shared memory pthreads, and NVIDIA GPUs, as well as hybrid MPI-shared memory pthreads or MPI-GPU parallelism”

1.1 PETSc's Role

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

1.2 What can you use it for?

- Scientific Computations: parallel linear algebra, in particular linear and nonlinear solvers
- Toolkit: Contains high level solvers, but also the low level tools to roll your own.
- Portable: Available on many platforms, basically anything that has MPI

Why use it? It's big, powerful, well supported.

1.3 A bit more about PETSc

What problems can it tackle:

- Serial and Parallel
- Linear and nonlinear
- Finite difference and finite element
- Structured and unstructured

What is in PETSc:

- Linear system solvers (sparse/dense, iterative/direct)
- Nonlinear system solvers
- Tools for distributed matrices
- Support for profiling, debugging, graphical output

And much much more

1.4 Where to learn more about PETSc

- PETSc Tutorial at TACC
- Regular tutorials by PETSc developers
- The PETSc website
- The PETSc documentation
- PETSc source code examples in the tarball

1.5 PETSc4Py

- Python bindings for PETSc, the Portable Extensible Toolkit for Scientic Computation
- Implemented with Cython
- A good friend of petsc4py is:
- mpi4py: Python bindings for MPI, the Message Passing Interface
- Other two projects depend on petsc4py:
- slepc4py: Python bindings for SLEPc, the Scalable Library for Eigenvalue Problem Computations
- tao4py: Python bindings for TAO, the Toolkit for Advanced Optimization

1.6 Python for control and logic C for local computation

- Decouple organization of storage from mathematical operations
- Vectors are not arrays
- Lots of small arrays
- get/setValues() methods
- Views into larger arrays
- Dense, local computation is cache/bandwidth efficient

1.7 PETSc4Py Interface

- Using PETSc4Py is very similar to using MPI4Py
- Provides ALL PETSc functionality in a Pythonic way
- Manages all memory (creation/destruction)
- Visualization with matplotlib

1.8 PETSc4Py Basic Operations

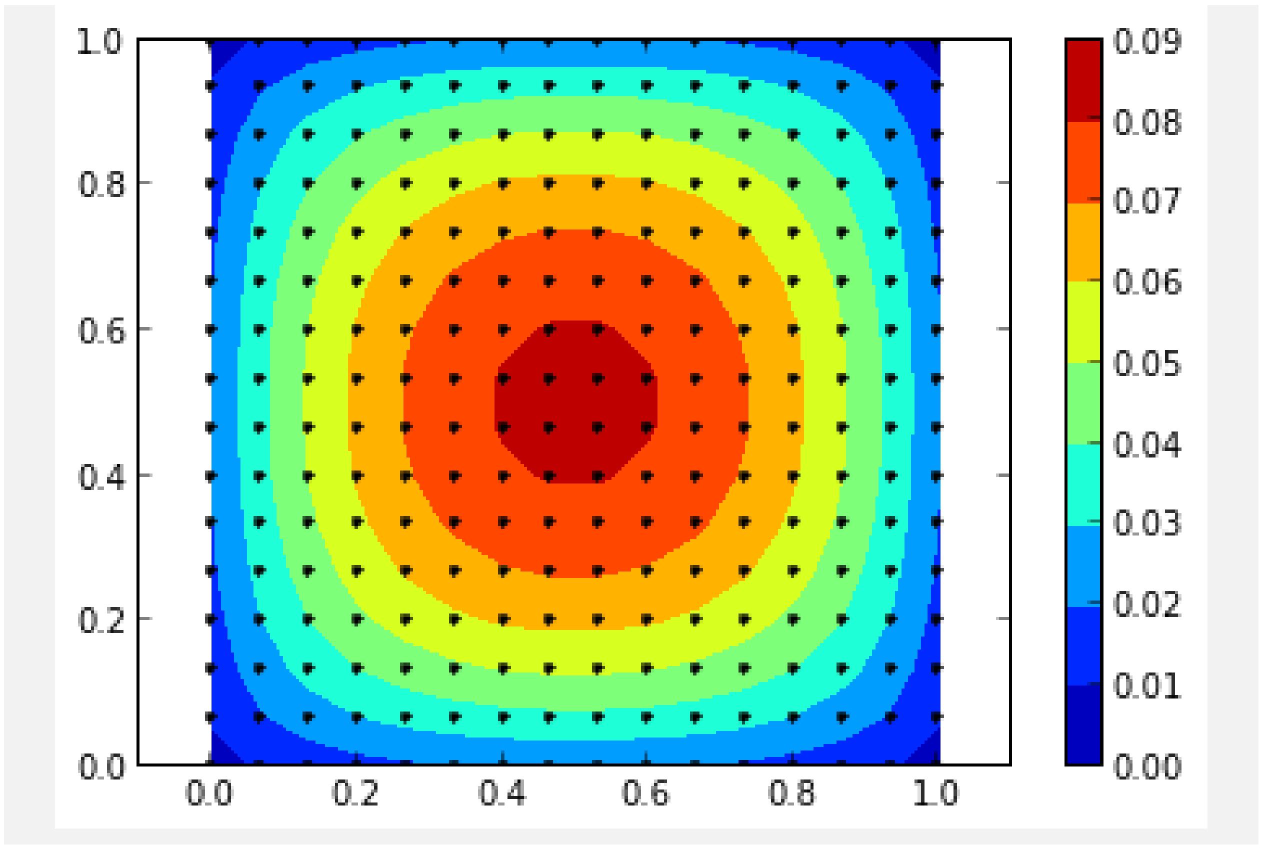
- Create a sparse matrix, set its size and type:
- A = PETSc.Mat()
- A.create(PETSc.COMM_WORLD)
- A.setSizes([mn, mn])
- A.setType('mpiaij')
- Create a linear solver and solve:
- ksp = PETSc.KSP()
- ksp.create(PETSc.COMM_WORLD)
- ksp.setOperators(A)
- ksp.setFromOptions()
- ksp.solve(b, x)

1.9 PETSc4Py Example

2D Poisson problem: -Laplacian(u) = 1, 0 < x,y < 1
with boundary conditions u = 0 for x=0, x=1, y=0, y=1

```
try: range = xrange
except: pass

import sys, petsc4py
petsc4py.init(sys.argv)

from petsc4py import PETSc

class Poisson2D(object):

    def __init__(self, da):
        assert da.getDim() == 2
        self.da = da
        self.localX = da.createLocalVec()

    def formRHS(self, B):
        b = self.da.getVecArray(B)
        mx, my = self.da.getSizes()
        hx, hy = [1.0/m for m in [mx, my]]
        (xs, xe), (ys, ye) = self.da.getRanges()
        for j in range(ys, ye):
            for i in range(xs, xe):
                b[i, j] = 1*hx*hy

    def mult(self, mat, X, Y):
        #
        self.da.globalToLocal(X, self.localX)
        x = self.da.getVecArray(self.localX)
        y = self.da.getVecArray(Y)
        #
        mx, my = self.da.getSizes()
        hx, hy = [1.0/m for m in [mx, my]]
        (xs, xe), (ys, ye) = self.da.getRanges()
        for j in range(ys, ye):
            for i in range(xs, xe):
                u = x[i, j] # center
                u_e = u_w = u_n = u_s = 0
                if i > 0: u_w = x[i-1, j] # west
                if i < mx-1: u_e = x[i+1, j] # east
                if j > 0: u_s = x[i, j-1] # south
                if j < ny-1: u_n = x[i, j+1] # north
                u_xx = (-u_e + 2*u - u_w)*hy/hx
                u_yy = (-u_n + 2*u - u_s)*hx/hy
                y[i, j] = u_xx + u_yy

OptDB = PETSc.Options()
```

```
n = OptDB.getInt('n', 16)
nx = OptDB.getInt('nx', n)
ny = OptDB.getInt('ny', n)

da = PETSc.DA().create([nx, ny], stencil_width=1)
pde = Poisson2D(da)

x = da.createGlobalVec()
b = da.createGlobalVec()
# A = da.createMat('python')
A = PETSc.Mat().createPython(
    [x.getSizes(), b.getSizes()], comm=da.comm)
A.setPythonContext(pde)
A.setUp()

ksp = PETSc.KSP().create()
ksp.setOperators(A)
ksp.setType('cg')
pc = ksp.getPC()
pc.setType('none')
ksp.setFromOptions()

pde.formRHS(b)
ksp.solve(b, x)

u = da.createNaturalVec()
da.globalToNatural(x, u)

from matplotlib import pylab
from numpy import mgrid

X, Y = mgrid[0:1:1j*nx,0:1:1j*ny]
Z = x[...].reshape(nx,ny, order='F')
pylab.figure()
pylab.contourf(X,Y,Z)
pylab.plot(X.ravel(),Y.ravel(),'.k')
pylab.axis('equal')
pylab.colorbar()
pylab.show()
```


0.0
0.2
0.4
0.6
0.8
1.0
0.00
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0.08
0.09

1.10 PETSc4Py and in parallel

```python
# Some preliminaries to connect to the engines

from IPython.parallel import Client
c = Client()
view = c[:]

%load_ext parallelmagic
view.activate()
view.block = True
%autopx
```

```
%autopx enabled
```

```python
try: range = xrange
except: pass

import sys, petsc4py
petsc4py.init(sys.argv)

from petsc4py import PETSc

class Poisson2D(object):

    def __init__(self, da):
        assert da.getDim() == 2
        self.da = da
        self.localX = da.createLocalVec()

    def formRHS(self, B):
        b = self.da.getVecArray(B)
        mx, my = self.da.getSizes()
        hx, hy = [1.0/m for m in [mx, my]]
        (xs, xe), (ys, ye) = self.da.getRanges()
        for j in range(ys, ye):
            for i in range(xs, xe):
                b[i, j] = 1*hx*hy

    def mult(self, mat, X, Y):
        #
        self.da.globalToLocal(X, self.localX)
        x = self.da.getVecArray(self.localX)
        y = self.da.getVecArray(Y)
        #
        mx, my = self.da.getSizes()
        hx, hy = [1.0/m for m in [mx, my]]
        (xs, xe), (ys, ye) = self.da.getRanges()
        for j in range(ys, ye):
            for i in range(xs, xe):
                u = x[i, j] # center
                u_e = u_w = u_n = u_s = 0
                if i > 0: u_w = x[i-1, j] # west
                if i < mx-1: u_e = x[i+1, j] # east
                if j > 0: u_s = x[i, j-1] # south
                if j < ny-1: u_n = x[i, j+1] # north
                u_xx = (-u_e + 2*u - u_w)*hy/hx
                u_yy = (-u_n + 2*u - u_s)*hx/hy
                y[i, j] = u_xx + u_yy

OptDB = PETSc.Options()

# change nx and ny if you want
n = OptDB.getInt('n', 16)
nx = OptDB.getInt('nx', n)
ny = OptDB.getInt('ny', n)

da = PETSc.DA().create([nx, ny], stencil_width=1)
pde = Poisson2D(da)

x = da.createGlobalVec()
b = da.createGlobalVec()
# A = da.createMat('python')
A = PETSc.Mat().createPython(
    [x.getSizes(), b.getSizes()], comm=da.comm)
A.setPythonContext(pde)
A.setUp()

ksp = PETSc.KSP().create()
ksp.setOperators(A)
ksp.setType('cg')
pc = ksp.getPC()
pc.setType('none')
ksp.setFromOptions()

pde.formRHS(b)
ksp.solve(b, x)

u = da.createNaturalVec()
da.globalToNatural(x, u)
x.getSizes()

engine_arrays = u.array
rank=da.comm.rank
```

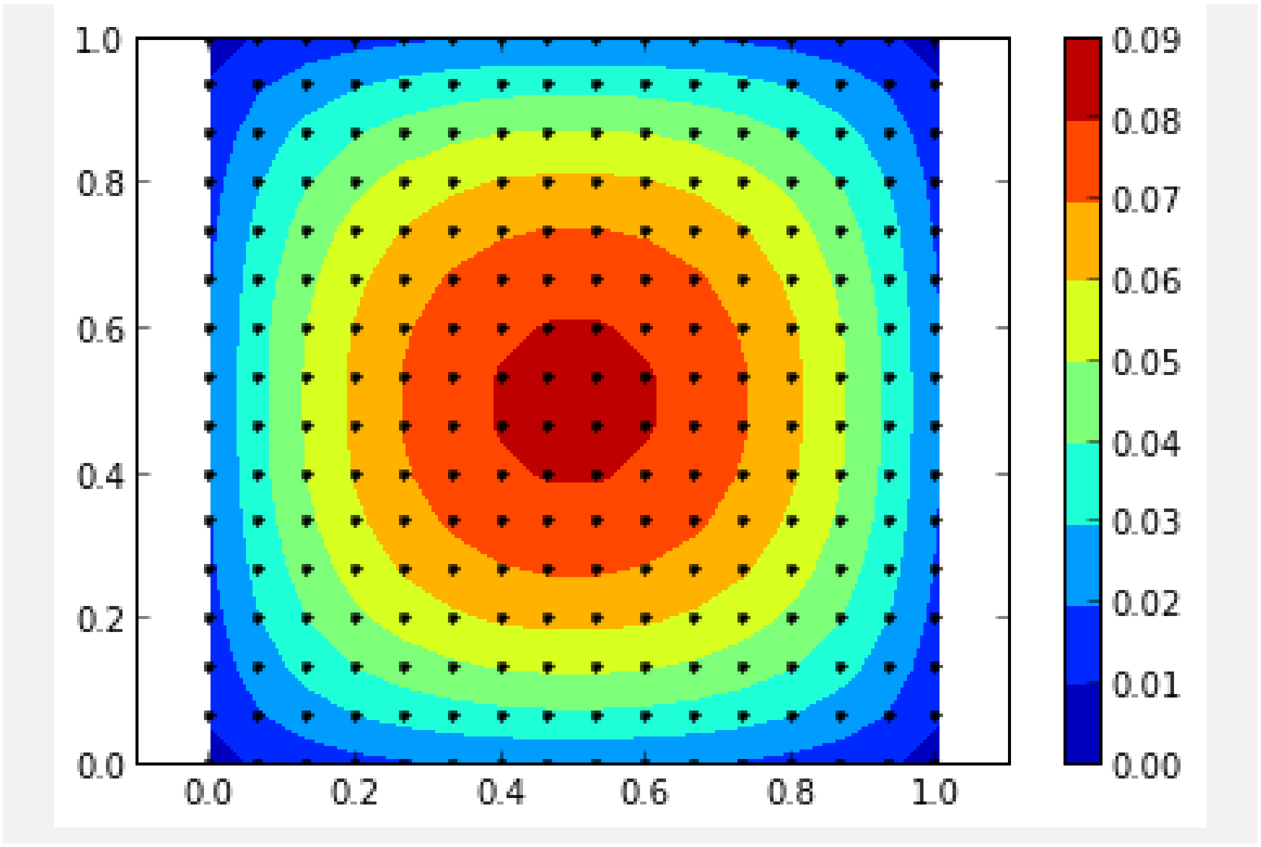
```
# now disable the parallel run to get the d
%autopx
```

```
%autopx disabled
```

```
# view the engine arrays and ranks
arrays = view['engine_arrays']
ranks = view['rank']
# Sort the arrays based on rank
arrays_and_ranks = zip(ranks, arrays)
arrays_and_ranks.sort()
sorted_ranks,sorted_arrays = zip(*arrays_and_ranks)
```

```
# Create a list to use with concatenate
list_arrays=list(sorted_arrays)
# Concatenate the ordered list of arrays into one flat array
u_flat = concatenate(list_arrays, axis=0)
```

```
# Get the global sizes nx and ny
nx=view['nx'][0]
ny=view['ny'][0]

# create a mesh grid
X, Y = mgrid[0:1:1j*nx,0:1:1j*ny]
# reshape the solution, orderering is Fortran based
u = u_flat[...].reshape(nx,ny, order='F')

# Let's plot
pylab.figure()
pylab.contourf(X,Y,u)
pylab.plot(X.ravel(),Y.ravel(),'.k')
pylab.axis('equal')
pylab.colorbar()
pylab.show()
```


1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
0.09
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0.00