

Advanced Python for Scientific Computing

Michael Milligan

milligan@msi.umn.edu

[http://www.msi.umn.edu/...](http://www.msi.umn.edu/)

To get the most out of this...

- Basic knowledge of Python
- Working Python install (feel free to use ours!)
 - Enthought Python Distribution / Canopy provides scientific and math libraries pre-installed

```
% module load python-epd  
% python
```

- MSI login + SSH or NX
- “**ssh -X itasca.msi.umn.edu**” to follow along

Why Python for Scientific Computing?

- **Rapid development**
 - Easy, readable syntax
 - Versatile tools for experimentation/learning
 - Comprehensive libraries
- **Powerful Features**
 - Process data at near “native code” speeds
 - Excellent visualization packages
 - Comprehensive libraries

When you leave today, you should be able to...

- Program interactively with **ipython**
- understand the basics of **numpy** and **scipy**
- **Efficiently compute** with large arrays of data
- Load and save data to/from **files on disk**
- Use **matplotlib** to plot data
- Take advantage of supercomputing resources with **parallel computing**
- **Know where to turn** for more help with these topics

Details

- We are describing Enthought Python Distribution.
 - Essentially: Pre-assembled compilation of Python 2.7 + numpy, scipy, other useful libraries
 - Free for academic use, a basic version is free for non-commercial use
 - Your computers, departments, etc may have a different version of Python installed. Everything we will see today is open source.
- In MSI labs: **module load python-epd**

Workshop Conventions

- UNIX shell commands are indicated with the **percent** sign.
- IPython interpreter commands have In/Out labels
- Neither sign indicates python code that should be entered into a text file.

```
% module load python-epd  
% ipython
```

```
In [1]: numpy.arange(5)  
Out[1]: array([0, 1, 2, 3, 4])
```

```
x = 5  
y = 6  
print x + y
```

IPython: Interactive Python

- Powerful environment for interactive work
- Run as “ipython” from any terminal
- --pylab option auto-loads **numpy**, sets up graphics for plotting
- Inspect any object with “?” or help()

```
% module load python-epd  
% ipython --pylab
```

```
In [2]: x = [1,2,3]  
  
In [3]: x?  
Type:      list  
String Form:[1, 2, 3]  
Length:    3  
Docstring:  
list() -> new empty list  
list(iterable) -> new list  
initialized from iterable's  
items
```

IPython: Interactive Python

- Build up a workspace of objects and functions
- Full history access through Out[], %recall, up/down arrow keys
- %load, %edit, or %run external files
- Lots more, type %magic

```
In [5]: def df(a,b):  
.....:     return a - b  
.....:  
  
In [6]: df(5,2)  
Out[6]: 3  
  
In [7]: Out[6] * 2  
Out[7]: 6
```

NumPy and SciPy

- **NumPy provides:**
 - the basic array and matrix data types
 - Efficient implementations of low-level math operations
 - A large library of high-level math functions built from efficient primitives
- **SciPy provides:**
 - A home for a wide variety of open-source mathematical and scientific algorithms
 - Modules for optimization, signal processing, linear algebra, statistics, interpolation, and more

NumPy arrays

- Array data type with vectorized operations(similar to Matlab or IDL)
- Supports same operations as Python list type
- ...except every element is of same data type
- ...so they can be stored in memory packed like C arrays

```
In [9]: a = arange(5)
In [10]: a
Out[10]: array([0, 1, 2, 3, 4])
In [11]: b = ones(5)
In [12]: b
Out[12]: array([ 1.,  1.,  1.,
 1.,  1.])
In [13]: a - b * 3
Out[13]: array([-3., -2., -1.,
 0.,  1.])
In [14]: sorted(a - b * 3)
Out[14]: [-3.0, -2.0, -1.0,
 0.0, 1.0]
In [15]: b.dtype
Out[15]: dtype('float64')
```

NumPy arrays are fast

`addition.py`

```
import numpy

def py_add(a, b):
    c = []
    for i in xrange(0,len(a)):
        c.append(1.324 * a[i] -
                  12.99 * b[i] + 1)
    return c

def np_add(a, b):
    return 1.324 * a - 12.99 * b + 1
```

In [26]: `from addition import *`

In [27]: `a = arange(1e6)`

In [28]: `b = random.randn(1e6)`

In [29]: `len(a)`

Out[29]: `1000000`

In [30]: `%timeit py_add(a,b)`
1 loops, best of 3: 2.97 s per loop

In [31]: `%timeit np_add(a,b)`
100 loops, best of 3: 10 ms per loop

Here we are comparing a “pure Python” loop to the equivalent in numpy

Multidimensional arrays

- NumPy arrays are rectangles in arbitrarily many dimensions
- + - * / operate element-by-element for same-shape arrays

```
In [4]: M = arange(2*4*3).reshape([2,4,3])
```

```
In [5]: M
```

```
Out[5]:
```

```
array([[[ 0,  1,  2],  
       [ 3,  4,  5],  
       [ 6,  7,  8],  
       [ 9, 10, 11]],  
  
      [[12, 13, 14],  
       [15, 16, 17],  
       [18, 19, 20],  
       [21, 22, 23]]])
```

```
In [6]: M[0,2,2]
```

```
Out[6]: 8
```

Array slicing

- Index notation gives access to any “slice” of an array
- Array slices can be assigned – this changes the original array
- $X = M[1,:,:].copy()$ would avoid changing M

```
In [20]: M[1,:,:]
Out[20]:
array([[12, 13, 14],
       [15, 16, 17],
       [18, 19, 20],
       [21, 22, 23]])

In [21]: x = M[1,:,:]

In [22]: x -= 10

In [23]: M
Out[23]:
array([[[ 0,  1,  2],
        [ 3,  4,  5],
        [ 6,  7,  8],
        [ 9, 10, 11]],

       [[ 2,  3,  4],
        [ 5,  6,  7],
        [ 8,  9, 10],
        [11, 12, 13]]])
```

Other common methods

- Numpy arrays have many useful built-in methods

```
In [40]: M = arange(1,10)
```

```
In [41]: M.min()  
Out[41]: 1
```

```
In [42]: M.mean()  
Out[42]: 5.0
```

```
In [43]: M.max()  
Out[43]: 9
```

```
In [44]: M.sum()  
Out[44]: 45
```

```
In [45]: M.prod()  
Out[45]: 362880
```

```
In [46]: M.ptp()  
Out[46]: 8
```

```
In [47]: M.cumsum()  
Out[47]: array([ 1,  3,  6, 10, 15, 21, 28, 36, 45])
```

Other common methods

- ...and the numpy module provides more

```
In [50]: numpy.floor(M)
Out[50]: array([ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,
9.])

In [51]: numpy.diff(M)
Out[51]: array([1, 1, 1, 1, 1, 1, 1, 1])

In [52]: numpy.dot(M,M)
Out[52]: 285

In [54]: numpy.sin(M)
Out[54]:
array([ 0.84147098,  0.90929743,  0.14112001,
-0.7568025 , -0.95892427,
-0.2794155 ,  0.6569866 ,  0.98935825,  0.41211849])

In [56]: numpy.log(M)
Out[56]:
array([ 0.          ,  0.69314718,  1.09861229,
1.38629436,  1.60943791,
1.79175947,  1.94591015,  2.07944154,  2.19722458])
```

Conditions and tests

- Vectorized logical operators + indexing functions
- Output of index functions can be used to slice arrays

```
In [57]: C = arange(10)
```

```
In [58]: C
```

```
Out[58]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [59]: C == 5
```

```
Out[59]: array([False, False, False, False, False,
   True, False, False, False], dtype=bool)
```

```
In [60]: numpy.where(C == 5)
```

```
Out[60]: (array([5]),)
```

```
In [67]: numpy.where((C > 3) & (C < 7))
```

```
Out[67]: (array([4, 5, 6]),)
```

```
In [72]: C[numpy.where((C > 3) & (C < 7))]
```

```
Out[72]: array([4, 5, 6])
```

Conditions and tests

- Vectorized logical operators + indexing functions
- Output of index functions can be used to slice arrays

```
In [73]: x = linspace(0,2*pi,100)
```

```
In [74]: y = sin(x)
```

```
In [75]: x[where(y > 0.9)]
```

```
Out[75]:
```

```
array([ 1.14239733,  1.20586385,  1.26933037,
       1.33279688,  1.3962634 ,  1.45972992,  1.52319644,  1.58666296,
       1.65012947,  1.71359599,  1.77706251,  1.84052903,  1.90399555,
       1.96746207])
```

More useful numpy modules

- **numpy.fft** – FFTs, forward/inverse, 1-D and N-D
- **numpy.random** – generate random numbers, many distributions to choose from
- **numpy.matrix** – special arrays that obey matrix math
- **numpy.polynomial** – module for representing and manipulating arbitrary polynomials

Plotting made easy

- **Matplotlib** provides high-quality 2-D (and some 3-D) plotting
 - Display in window or output to PDF, SVG, PNG, etc
 - Implemented as modular object-oriented system
- **Pylab** provides a Matlab-ish interactive interface to Matplotlib
 - Access with ipython –pylab
 - Defaults to popping up plots in a separate window

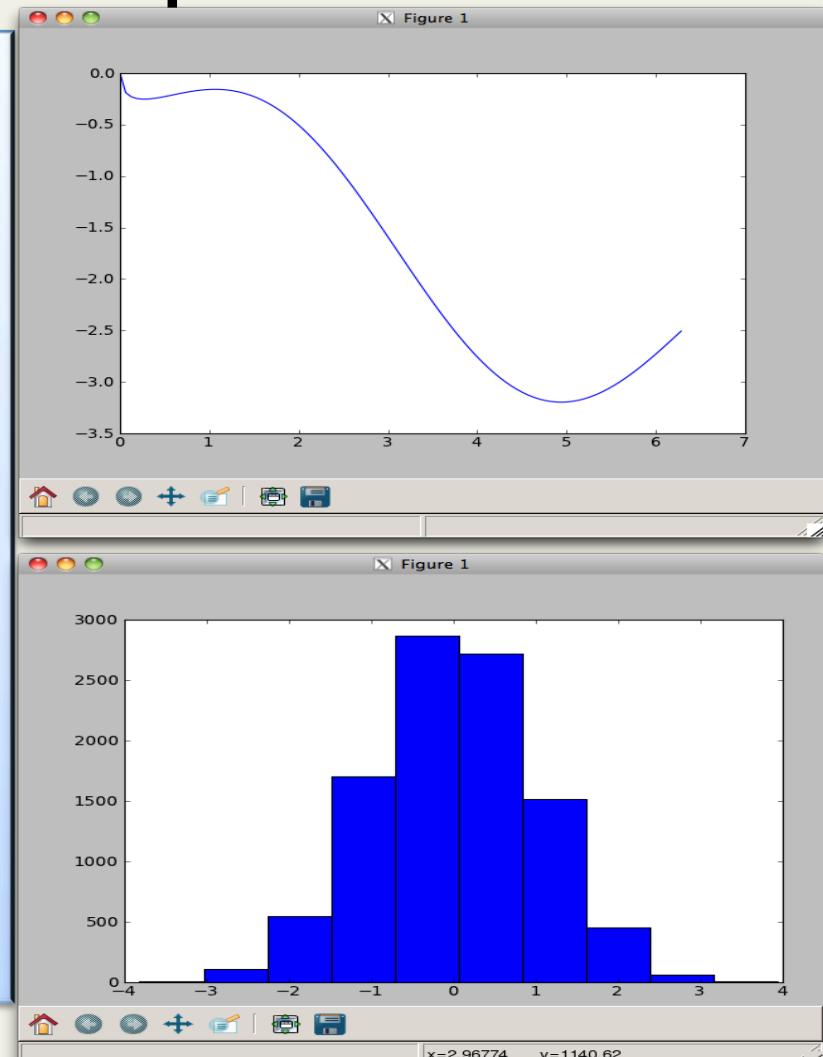
Some basic examples...

```
% ipython --pylab

In [1]: x = numpy.linspace(0,2*pi,100)

In [2]: y = sin(x)

In [3]: plot(x, sin(x) - sqrt(x))
Out[3]: [
```



Some advanced examples...

- These examples are from the **matplotlib.org** examples section...

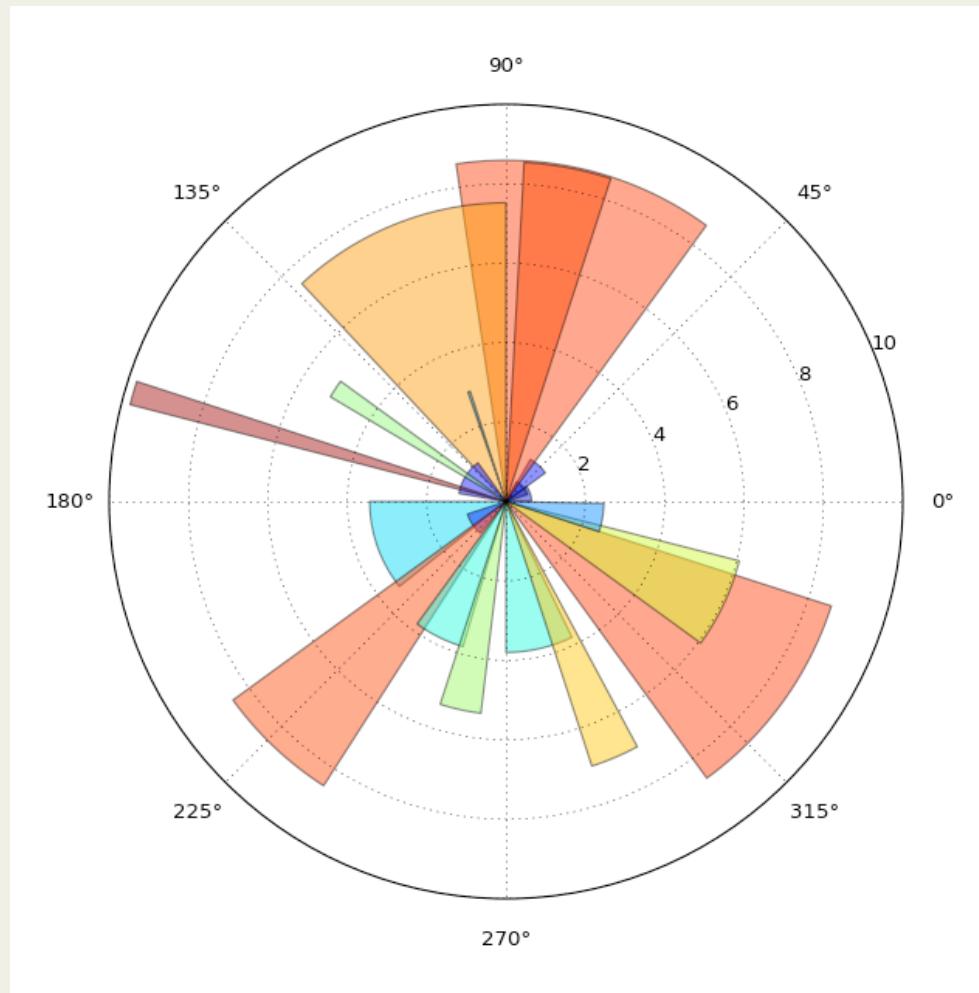
Some advanced examples...

```
import numpy as np
import matplotlib.cm as cm
from matplotlib.pyplot import figure, show, rc

# force square figure and square axes looks better for polar, IMO
fig = figure(figsize=(8,8))
ax = fig.add_axes([0.1, 0.1, 0.8, 0.8], polar=True)

N = 20
theta = np.arange(0.0, 2*np.pi, 2*np.pi/N)
radii = 10*np.random.rand(N)
width = np.pi/4*np.random.rand(N)
bars = ax.bar(theta, radii, width=width, bottom=0.0)
for r,bar in zip(radii, bars):
    bar.set_facecolor( cm.jet(r/10.))
    bar.set_alpha(0.5)
show()
```

Some advanced examples...

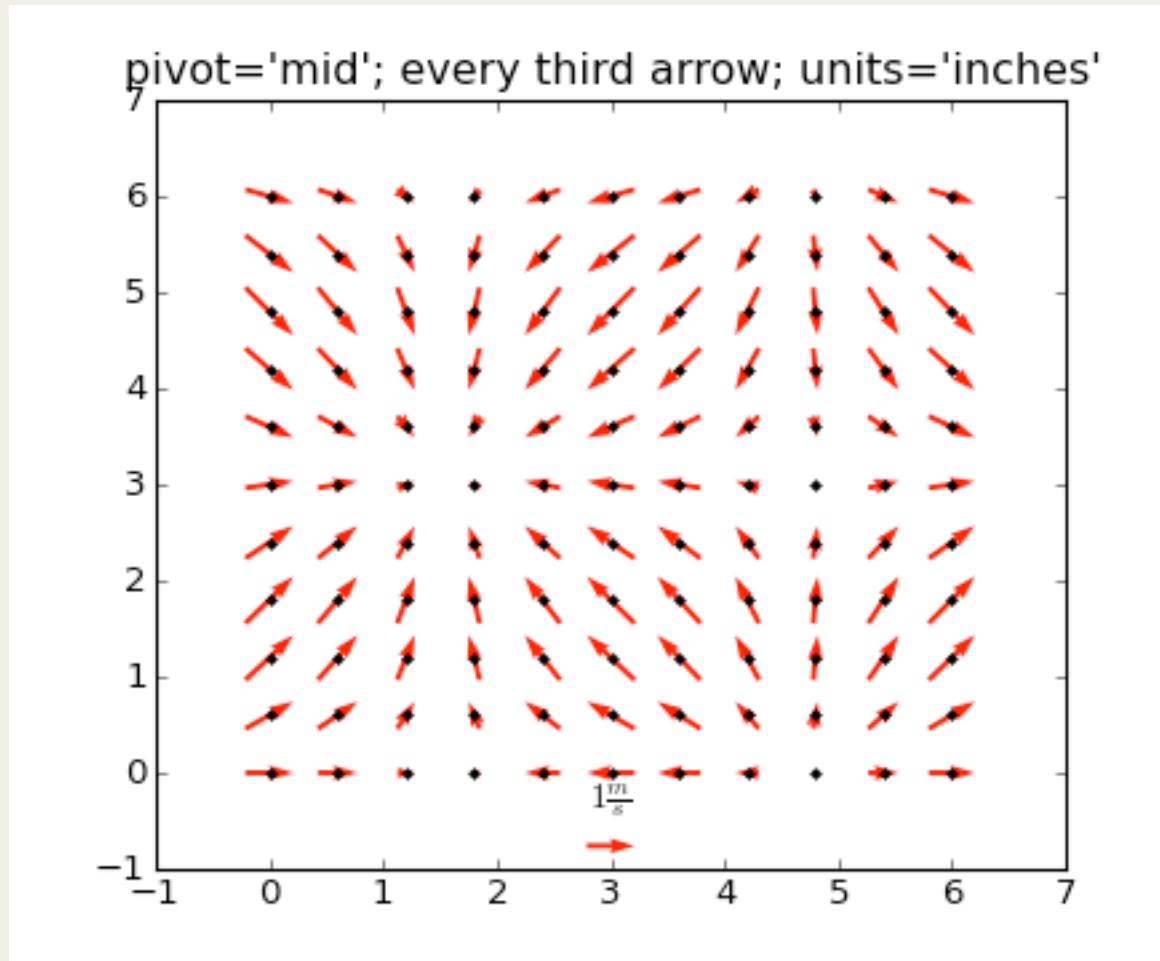


Some advanced examples...

```
import numpy as np
from pylab import *
from numpy import ma
X,Y = meshgrid( arange(0,2*pi,.2),arange(0,2*pi,.2) )
U = cos(X)
V = sin(Y)

figure()
Q = quiver( X[::3, ::3], Y[::3, ::3], U[::3, ::3], V[::3, ::3],
             pivot='mid', color='r', units='inches' )
qk = quiverkey(Q, 0.5, 0.03, 1, r'$1 \frac{m}{s}$',
                fontproperties={'weight': 'bold'})
plot( X[::3, ::3], Y[::3, ::3], 'k.' )
axis([-1, 7, -1, 7])
title("pivot='mid'; every third arrow; units='inches'")
```

Some advanced examples...



SciPy expands the menu

- [Clustering algorithms \(scipy.cluster\)](#)
- [Integration and ODEs \(scipy.integrate\)](#)
- [Interpolation \(scipy.interpolate\)](#)
- [Input and output \(scipy.io\)](#)
- [Linear algebra \(scipy.linalg\)](#)
- [Multi-dimensional image processing \(scipy.ndimage\)](#)
- [Optimization and root finding \(scipy.optimize\)](#)
- [Signal processing \(scipy.signal\)](#)
- [Sparse matrices \(scipy.sparse\)](#)
- [Spatial algorithms and data structures \(scipy.spatial\)](#)
- [Special functions \(scipy.special\)](#)
- [Statistical functions \(scipy.stats\)](#)
- And then some...

SciPy is also fast

- Most SciPy routines use fast NumPy low-level math operations
- Some SciPy routines use highly optimized external libraries
 - E.g. `scipy.linalg` links to BLAS, LAPACK or MKL behind the scenes

Data on disk

- Chances are you want to load and save data
- **numpy** and **scipy.io** offer a variety of facilities

Data on disk: text files

- **Very common for smaller data sets:**
simple columns of numbers
- `numpy.loadtxt()` – simple interface, good defaults
- `numpy.genfromtxt()` – more complex, handles unusual formatting, comments, missing values, etc

```
data.txt
```

```
0.000 10.000 20.000 30.000 40.000
1.000 11.000 21.000 31.000 41.000
2.000 12.000 22.000 32.000 42.000
3.000 13.000 23.000 33.000 43.000
4.000 14.000 24.000 34.000 44.000
...
```

```
In [21]: A = numpy.loadtxt('data.txt')
```

```
In [22]: A
```

```
Out[22]:
```

```
array([[ 0.,  10.,  20.,  30.,  40.],
       [ 1.,  11.,  21.,  31.,  41.],
       [ 2.,  12.,  22.,  32.,  42.],
       [ 3.,  13.,  23.,  33.,  43.],
       ...]]
```

Data on disk: text files

- Numpy.savetxt() – write to columns of numbers

```
In [25]: numpy.savetxt('data.txt', A, delimiter=",",
                      fmt="%3.3f")
```

```
In [26]: %pycat data.txt
0.000,10.000,20.000,30.000,40.000
1.000,11.000,21.000,31.000,41.000
2.000,12.000,22.000,32.000,42.000
3.000,13.000,23.000,33.000,43.000
4.000,14.000,24.000,34.000,44.000
5.000,15.000,25.000,35.000,45.000
6.000,16.000,26.000,36.000,46.000
7.000,17.000,27.000,37.000,47.000
8.000,18.000,28.000,38.000,48.000
9.000,19.000,29.000,39.000,49.000
```

Data on disk: binary formats

- **Binary data** is much more scalable
 - Smaller files on disk
 - Faster to load and save
 - May be necessary to exchange data with other software
 - Stick to **portable** (machine-independent) formats

Data on disk: binary formats

- **NumPy native format (.npy)**
 - `numpy.load()` and `numpy.save()`
 - Or use `numpy.savez()` to store many arrays in compressed .npz
 - Fast, portable, but mostly only supported by Python
- **scipy.io.matlab** – support for Matlab (.mat)
 - `scipy.io.loadmat()` and `scipy.io.savemat()`
- **scipy.io.idl** – read (no save) IDL .sav files
 - `scipy.io.readsav()`
- Several standard formats supported
 - `scipy.io.netcdf`, `h5py` for HDF5, `pyfits` for FITS, etc

Scaling up with parallelization

- For big jobs you will eventually want to **parallelize** your code
- The Python interpreter has trouble with multithreading – multi-process is usually best
- Approach depends on the problem you need to solve

Parallel processes

- Many jobs need to process lots of data, don't need to communicate amongst themselves
- Sometimes called “embarrassingly parallel”
- **GNU Parallel** -- a simple way to launch jobs
 - Launch one job for every file in a dir, line in a file, etc
 - Can work with PBS on itasca to use many nodes

GNU Parallel example

```
% ssh itasca
% qsub -I -l walltime=00:05:00,pmem=1000mb,nodes=10:ppn=8
qsub: waiting for job 616263.node1081.localdomain to start
qsub: job 616263.node1081.localdomain ready

% module load parallel
% module load python-epd
% seq 80 | parallel -j 8 --sshloginfile $PBS_NODEFILE --env PATH --env
PYTHONPATH 'echo Job {} on $(hostname)'
Job 1 on node0109
Job 2 on node0109
Job 3 on node0109
Job 4 on node0109
Job 5 on node0109
Job 6 on node0109
Job 7 on node0109
Job 8 on node0109
Job 9 on node0080
Job 10 on node0080
Job 11 on node0080
Job 12 on node0080
... Etc . . .
```

GNU Parallel example

- -j should match ppn (unless you know what you’re doing) – this is processes per node
- Will run one job per line of input on stdin or in argfile – max of nodes * ppn running at once
- See “man parallel” for more features

Alternate ways to run parallel:

```
% parallel -a <argfile> -j 8 --sshloginfile $PBS_NODEFILE --env PATH --env PYTHONPATH 'echo Job {} on $(hostname)'

# Load any needed modules from a shell script:
% parallel -a <argfile> -j 8 --sshloginfile $PBS_NODEFILE "myscript.sh {}"
```

MPI for Python

- MPI “Message Passing Interface” enables parallel processes to communicate efficiently
- Commonly one process will be “controller” and manage worker processes
- Inherent support for scatter-gather operations
- MPI is well-supported on our clusters
- mpi4py interfaces to MPI from inside Python
- Caveat for MPI gurus: numpy does not have distributed arrays yet, complicates some algorithms

Example with mpi4py

- Simple “Hello world” script

```
Helloworld-mpi.py
```

```
#!/usr/bin/env python
"""
Parallel Hello World
"""

from mpi4py import MPI
import sys

size = MPI.COMM_WORLD.Get_size()
rank = MPI.COMM_WORLD.Get_rank()
name = MPI.Get_processor_name()

sys.stdout.write(
    "Hello, World! I am process %d of %d on %s.\n"
    % (rank, size, name))
```

Example with mpi4py

- Simple “Hello world” script

```
% ssh itasca
% qsub -I -l walltime=00:05:00,pmem=1000mb, nodes=10:ppn=8
qsub: waiting for job 616294.node1081.localdomain to start
qsub: job 616294.node1081.localdomain ready

% module load python-epd ompi/gnu
% mpiexec -np 80 python helloworld-mpi.py
Hello, World! I am process 3 of 80 on node0079.
Hello, World! I am process 52 of 80 on node0126.
Hello, World! I am process 6 of 80 on node0079.
Hello, World! I am process 25 of 80 on node0099.
Hello, World! I am process 14 of 80 on node0080.
Hello, World! I am process 64 of 80 on node0128.
. . . Etc . . .
```

More with mpi4py

- Possible to pass numpy arrays like buffers

```
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

# pass explicit MPI datatypes
if rank == 0:
    data = numpy.arange(1000, dtype='i')
    comm.Send([data, MPI.INT], dest=1, tag=77)
elif rank == 1:
    data = numpy.empty(1000, dtype='i')
    comm.Recv([data, MPI.INT], source=0, tag=77)
```

Too much to cover...

- **Ipython notebook** – connect to ipython with a browser for a Mathematica-like notebook interface
- **PyCUDA** and **PyOpenCL** – GPU computing
- **SymPy** – Mathematica-style symbolic math
- **Databases** are easy to connect to Python; or use advanced big data toolkit like **Pandas**

Community and Documentation

- Actively developed and supported
- Excellent documentation
- www.python.org/doc
- Scipy.org
- wiki.scipy.org
- Ipython.org
- matplotlib.org
- Mpi4py.scipy.org

Next Step

- Hands-on
- You can also run the examples on your laptop's Python distribution
- Enthought is installed in all labs and on supercomputers at MSI
- Full academic version of Enthought Canopy is in testing on Itasca now
- Questions!