A Hands-on Introduction to

MPI Python Programming

Sung Bae, Ph.D

New Zealand eScience Infrastructure

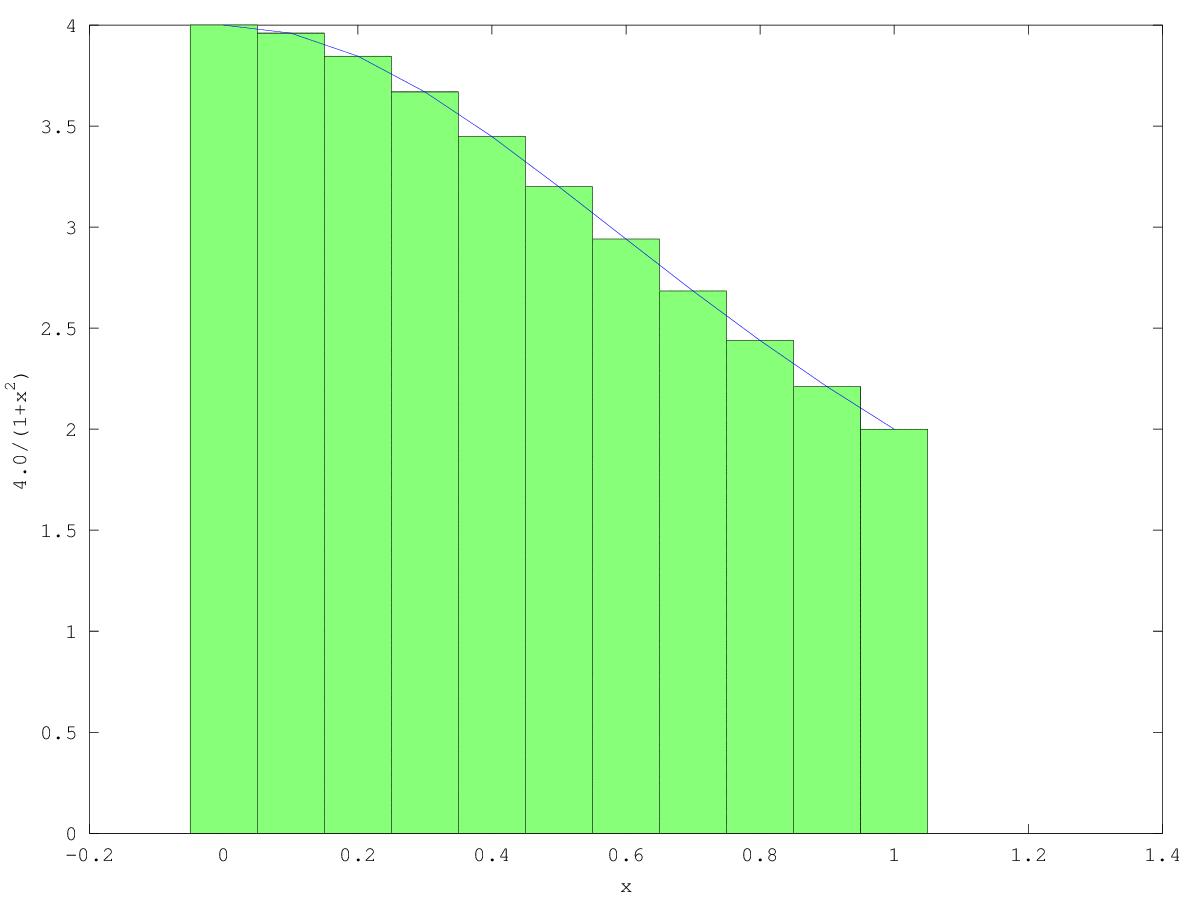
# Introduction: Python is SLOW

### Example: Computing the value of π=3.14159…

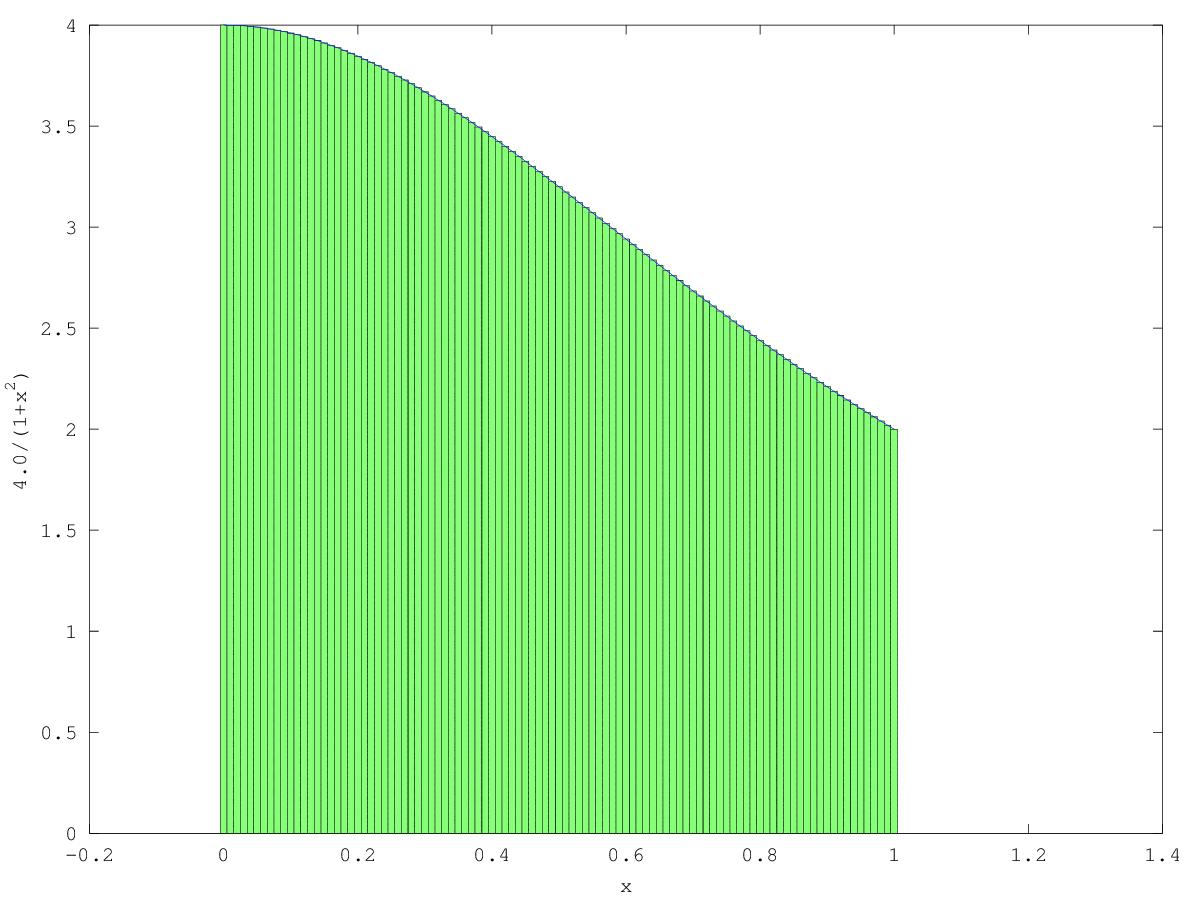
For

it is known that the value of π can be computed by the numerical integration

This can be approximated by



By increasing the number of steps (ie. smaller Δx), the approximation gets more precise.



We can design the following C and Python programs.

**EXAMPLE**

|  |  |
| --- | --- |
| import time  def Pi(num\_steps):  start = time.time()  step = 1.0/num\_steps  sum = 0  for i in xrange(num\_steps):  x= (i+0.5)\*step  sum = sum + 4.0/(1.0+x\*x)  pi = step \* sum  end = time.time()  print "Pi with %d steps is %f in %f secs" %(num\_steps, pi, end-start)  if \_\_name\_\_ == '\_\_main\_\_':  Pi(100000000) | #include <stdio.h>  #include <time.h>  void Pi(int num\_steps) {  double start, end, pi, step, x, sum;  int i;  start = clock();  step = 1.0/(double)num\_steps;  sum = 0;  for (i=0;i<num\_steps;i++) {  x = (i+0.5)\*step;  sum = sum + 4.0/(1.0+x\*x);  }  pi = step \* sum;  end= clock();  printf("Pi with %d steps is %f in %f secs\n", num\_steps, pi,(float)(end-begin)/CLOCKS\_PER\_SEC);  void main() {  Pi(100000000);  } |

**HANDS ON**

|  |
| --- |
| Go to pi\_example directory   1. Compile pi.c (gcc pi.c –o pi –O3) and execute it (./pi) 2. Run pi.py (python pi.py) |

**DISCUSS**

|  |
| --- |
| Why is Python code slow?  How can we speed it up? |

# Faster Python code

## Speed-up Options



## Example : Numba

### Profiling

* Find what is slowing you down
  + Line-by-line profiling is often useful [http://pythonhosted.org/line\_profiler](http://pythonhosted.org/line_profiler/)
  + Not part of standard python. Needs separate installation (already installed)
* Put @profile above the function that you’re interested in

**EXAMPLE**

|  |
| --- |
| **…**  **@profile**  def Pi(num\_steps):  start = time.time()  step = 1.0/num\_steps  sum = 0  for i in xrange(num\_steps):  x= (i+0.5)\*step  sum = sum + 4.0/(1.0+x\*x)  ….. |

**HANDS ON**

|  |
| --- |
| 1. Go to “profiling” subdirectory. 2. Open pi.py 3. Add @profile to the function Pi 4. This will take some time. Update the last line of pi.py : Pi(100000000) 🡺 Pi(1000000) 5. Run interactive -A uoa00243 -c 1 -e "python kernprof.py -l -v pi.py" |

**OUTPUT**

|  |
| --- |
| Pi with 1000000 steps is 3.14159265358976425020 in 13.541438 secs  Wrote profile results to pi.py.lprof  Timer unit: 1e-06 s  File: pi.py  Function: Pi at line 8  Total time: 6.54915 s  Line # Hits Time Per Hit % Time Line Contents  ==============================================================  8 @profile  9 def Pi(num\_steps):  10 1 5 5.0 0.0 start = time.time()  11 1 4 4.0 0.0 step = 1.0/num\_steps  12  13 1 2 2.0 0.0 sum = 0  14 1000001 1986655 2.0 30.3 for i in range(num\_steps):  15 1000000 2189274 2.2 33.4 x= (i+0.5)\*step  16 1000000 2373071 2.4 36.2 sum = sum + 4.0/(1.0+x\*x)  17  18 1 5 5.0 0.0 pi = step \* sum  19  20 1 6 6.0 0.0 end = time.time()  21 1 128 128.0 0.0 print "Pi with %d steps is %.20f in %f secs" %(num\_steps, pi, end-start) |

**DISCUSS**

|  |
| --- |
| Identify the bottleneck of this program |

### NUMBA

Numba (<http://numba.pydata.org/>) is a just-in-time compiler and produces optimized native code from Python code.

**HANDS ON**

Go to “pi\_example/pi\_numba.py”

Step 1. Separate the bottleneck

|  |
| --- |
| # pi\_numba.py  import time  def Pi(num\_steps ):  start = time.time()  step = 1.0/num\_steps  sum = 0  for i in xrange(num\_steps):  x= (i+0.5)\*step  sum = sum + 4.0/(1.0+x\*x)  pi = step \* sum  end = time.time()  print "Pi with %d steps is %f in %f secs" %(num\_steps, pi, end-start)  if \_\_name\_\_ == '\_\_main\_\_':  Pi(100000000) |

Step 2. Make a function that contains the bottleneck

|  |
| --- |
| # pi\_numba.py  import time  def loop(num\_steps):  step = 1.0/num\_steps  sum = 0  for i in xrange(num\_steps):  x= (i+0.5)\*step  sum = sum + 4.0/(1.0+x\*x)  return sum  def Pi(num\_steps ):  start = time.time()  **sum = loop(num\_steps)**  pi = sum/num\_steps  end = time.time()  print "Pi with %d steps is %f in %f secs" %(num\_steps, pi, end-start)  if \_\_name\_\_ == '\_\_main\_\_':  Pi(100000000) |

Step 3. Import Numba and add a decorator

|  |
| --- |
| # pi\_numba.py  import time  **from numba import jit**  **@jit**  def loop(num\_steps):  step = 1.0/num\_steps  sum = 0  for i in xrange(num\_steps):  x= (i+0.5)\*step  sum = sum + 4.0/(1.0+x\*x)  return sum  def Pi(num\_steps ):  start = time.time()  **sum = loop(num\_steps)**  pi = sum/num\_steps  end = time.time()  print "Pi with %d steps is %f in %f secs" %(num\_steps, pi, end-start)  if \_\_name\_\_ == '\_\_main\_\_':  Pi(100000000) |

**DISCUSS**

|  |
| --- |
| 1. Execute the revised version of pi.py and compare its performance. Is it adequately improved? |

# Parallel Programming

Once all the options in “serial (or sequential) processing” paradigm have been exhausted, and if we still need further speed-up, “parallel processing” is the next step.

## Parallel Programming in Python

### Distributed Memory – mpi4Py

Each processor (CPU or core) accesses its own memory and processes a job. If a processor needs to access data resident in the memory owned by another processor, these two processors need to exchange “messages”. Python supports MPI (Message Passing Interface) through mpi4py module.

### Shared Memory - multiprocessing

Processors share the access to the same memory. OpenMP is the most popular in this category, which enables concurrently running multiple threads, with the runtime environment allocating threads to different processors. Python has Global Interpreter Lock (GIL), which prevents multiple native threads from executing Python bytecodes at once[[1]](#footnote-1), and as a result, there is no OpenMP package for Python.[[2]](#footnote-2)

Python’s standard “multiprocessing” module (http://docs.python.org/2/library/multiprocessing.html) may be considered as an alternative option.

### GPGPU – PyCUDA, PyOpenCL

General-purpose computing on graphics processing units (GPGPU) utilizes GPU as an array of parallel processors. Python supports NVidia’s proprietary CUDA and open standard OpenCL. Ideal for applications having large data sets, high parallelism, and minimal dependency between data elements.

|  |  |
| --- | --- |
| processor  processor  processor  processor  mem  mem  mem  mem  Message Passing | processor  processor  processor  processor  memory |
| Figure 1. Distributed Memory | Figure 2. Shared Memory  Figure 1Shared Memory |
| GPU  Memory  CPU  GPU  Main  Memory | |
| Figure 3. GPU | |

## Basics mpi4Py programming

Go to “parallel” subdirectory.

Example 1. MPI Hello World

Write hello\_mpi.py as follows.

|  |
| --- |
| #hello\_mpi.py  from mpi4py import MPI  comm = MPI.COMM\_WORLD  rank = comm.Get\_rank()  size = comm.Get\_size()  print "hello world from process %d/%d“ %(rank,size) |

MPI program is executed by the following command

|  |
| --- |
| $ mpirun –n 4 python ./hello\_mpi.py |

where “–n 4” means the number of parallel processes.

**OUTPUT**

|  |
| --- |
| hello world from process 0/4  hello world from process 1/4  hello world from process 3/4  hello world from process 2/4 |

Exercise 1. *Embarrassingly* Parallel Photo processing

The following program denoise\_serial.py applies a de-noise algorithm to the list of photos.

|  |
| --- |
| import numpy as np  from skimage import data, img\_as\_float  from skimage.filter import denoise\_bilateral  import skimage.io  import os.path  import time  curPath = os.path.abspath(os.path.curdir)  noisyDir = os.path.join(curPath,'noisy')  denoisedDir = os.path.join(curPath,'denoised')  def loop(imgFiles):  for f in imgFiles:  img = img\_as\_float(data.load(os.path.join(noisyDir,f)))  startTime = time.time()  img = denoise\_bilateral(img, sigma\_range=0.1, sigma\_spatial=3),  skimage.io.imsave(os.path.join(denoisedDir,f), img)  print("Took %f seconds for %s" %(time.time() - startTime, f))  def serial():  total\_start\_time = time.time()  imgFiles = os.listdir(noisyDir)  imgFiles=[x for x in imgFiles if '.jpg' in x]  imgFiles.sort()  loop(imgFiles)  print("Total time %f seconds" %(time.time() - total\_start\_time))  if \_\_name\_\_=='\_\_main\_\_':  serial() |

A noisy photo will look less grainy.



(Image obtained from The Alfred Hitchcock Wiki (www.hitchcockwiki.com) *– Secret Agent* (1936)

**DISCUSS**

|  |
| --- |
| How long does it take to process 100 photos?  Can we use Numba to speed-up? |

**HANDS ON**

Complete “parallel” function using MPI such that 100 photos can be processed in parallel

|  |
| --- |
| import numpy as np  from skimage import data, img\_as\_float  from skimage.filter import denoise\_tv\_chambolle, denoise\_bilateral,denoise\_tv\_bregman  import skimage.io  import os.path  import time  from mpi4py import MPI  from numba import jit  curPath = os.path.abspath(os.path.curdir)  noisyDir = os.path.join(curPath,'noisy')  denoisedDir = os.path.join(curPath,'denoised')  @jit  def loop(imgFiles,rank):  for f in imgFiles:  img = img\_as\_float(data.load(os.path.join(noisyDir,f)))  startTime = time.time()  img = denoise\_bilateral(img, sigma\_range=0.1, sigma\_spatial=3),  skimage.io.imsave(os.path.join(denoisedDir,f), img)  print ("Process %d: Took %f seconds for %s" %(rank, time.time() - startTime, f))  def parallel():  **comm = MPI.COMM\_WORLD**  **rank = comm.Get\_rank()**  **size = comm.Get\_size()**    totalStartTime = time.time()  imgFiles = os.listdir(noisyDir)  imgFiles=[x for x in imgFiles if '.jpg' in x]  imgFiles.sort()  **numFiles = len(imgFiles)/size** #number of files this process will handle  **imgFiles = imgFiles[rank\*numFiles:(rank+1)\*numFiles]** #list of files this process will handle  loop(imgFiles,rank)  print "Total time %f seconds" %(time.time() - totalStartTime)  if \_\_name\_\_=='\_\_main\_\_':  parallel() |

Don’t forget to run it with “mpirun” command.

|  |
| --- |
| $ mpirun –n 4 python denoise\_parallel.py |

Example 2 Point-to-Point Communication

The following example shows the basic point-to-point communication, send and recv.

|  |
| --- |
| #hello\_p2p.py  from mpi4py import MPI  comm = MPI.COMM\_WORLD  rank = comm.Get\_rank()  size = comm.Get\_size()  if rank == 0:  for i in range(1, size):  sendMsg = “Hello, Rank %d“ %i  **comm.send(sendMsg, dest=i)**  else:  **recvMsg = comm.recv(source=0)**  print recvMsg |

Execute this program by the following command

|  |
| --- |
| $ mpirun –n 4 python ./hello\_p2p.py |

This will launch 4 parallel processes, rank 0…rank 3, and produce output similar to:

**OUTPUT**

|  |
| --- |
| Hello, Rank 1  Hello, Rank 2  Hello, Rank 3 |

Example 3. Collective Communication – Broadcast

|  |
| --- |
| #hello\_bcast.py  from mpi4py import MPI  comm = MPI.COMM\_WORLD  rank = comm.Get\_rank()  size = comm.Get\_size()  if rank == 0:  **comm.bcast("Hello from Rank 0", root=0)**  else:  **msg=comm.bcast(root=0)**  print "Rank %d received: %s" %(rank, msg) |

Execute this program by the following command

|  |
| --- |
| $ mpirun –n 4 python ./hello\_bcast.py |

This will launch 4 parallel processes, rank 0…rank 3, and produce output similar to:

**OUTPUT**

|  |
| --- |
| Rank 2 received: Hello from Rank 0  Rank 1 received: Hello from Rank 0  Rank 3 received: Hello from Rank 0 |

Example 4. P2P vs Collective – Reduce

Consider the following example code.

|  |
| --- |
| #sum\_p2p.py  from mpi4py import MPI  comm = MPI.COMM\_WORLD  rank=comm.Get\_rank()  size=comm.Get\_size()  val = (rank+1)\*10  print "Rank %d has value %d" %(rank, val)  if rank == 0:  sum = val  for i in range(1,size):  sum += comm.recv(source=i)  print "Rank 0 worked out the total %d" %sum  else:  comm.send(val, dest=0) |
|  |
| send(20,0)  recv(1)  send(30,0)  send(40,0)  Sum =10+20 +….  rank 0  rank 1  rank 2  rank 3  recv(2)  recv(3) |
| Figure 4. Computing Sum at Rank 0: Values received from Rank 1,2 and 3 |

Each process sends a value to Rank 0 – Rank 1 sends 20 etc. Rank 0 doesn’t need to send to itself.

Rank 0 collects all values and computes the sum, and produces an output like

**OUTPUT**

|  |
| --- |
| Rank 0 worked out the total 100 |

Note that Rank 0 “receives” from Rank 1, Rank2 and Rank 3 **in sequence**. Each process starts to “send” as soon as the process gets executed, but the “send” only completes when the corresponding “recv” is called by Rank 0.

Having this “sequential” routine in parallel code is not ideal. With only 4 processes, this may not sound like a big deal, but this can be very inefficient when we have, say, 1000 processes. Sending values sequentially defeats the purpose of parallel programming.

Now, consider the following code.

|  |
| --- |
| from mpi4py import MPI  comm = MPI.COMM\_WORLD  rank = comm.Get\_rank()  size = comm.Get\_size()  val = (rank+1)\*10  print "Rank %d has value %d" %(rank, val)  **sum = comm.reduce(val, op=MPI.SUM, root=0)**  if rank==0:  print "Rank 0 worked out the total %d" %sum |
| reduce(MPI.SUM)  Sum = 100  rank 0  rank 0  rank 1  rank 2  rank 3 |
| Figure 5. Computing Sum at Rank 0: All values collected and computed by "reduce" |

This program produces the same result, but uses a collective call “reduce”. This function causes the value in “val” in every process to be sent to the root process (Rank 0 in this case), and applies “SUM”[[3]](#footnote-3) operation on all values. As a result, multiple values are *reduced* to one value.

Exercise 2 Parallel computation of pi

Let’s revisit pi\_numba.py

We have identified the “for” loop was the bottleneck and used NUMBA to make it fast

|  |
| --- |
| #pi\_numba.py  import time  **from numba import jit**  **@jit**  def loop(num\_steps):  step = 1.0/num\_steps  sum = 0  for i in xrange(num\_steps):  x= (i+0.5)\*step  sum = sum + 4.0/(1.0+x\*x)  return sum  def Pi(num\_steps ):  start = time.time()  **sum = loop(num\_steps)**  pi = step \* sum  end = time.time()  print "Pi with %d steps is %f in %f secs" %(num\_steps, pi, end-start)  if \_\_name\_\_ == '\_\_main\_\_':  Pi(100000000) |

|  |
| --- |
| Rank 0  Rank 1  Rank 2  Rank 3  local\_sum  local\_sum  local\_sum  local\_sum  sum |
| Figure 7 Computing total sum from local\_sum's computed by processes |

Here, *num\_steps*=100000000, and the function loop will run *num\_steps* iterations.

Suppose we wish to parallelize this with 4 processes. We will allocate “*num\_steps*/4” steps to each process, such that

* Steps [0..num\_steps/4] allocated to Rank 0
* Steps [num\_steps/4..2\*num\_steps/4] allocated to Rank 1
* Steps [2\*num\_steps/4..3\*num\_steps/4] allocated to Rank 2
* Steps [3\*num\_steps/4..num\_steps] allocated to Rank 3

We need to modify pi\_numba.py to accommodate this idea.

**HANDS ON**

Step 1: Modify function loop() to specify begin and end steps

|  |
| --- |
| @jit  def loop(num\_steps, **begin, end**):  step = 1.0/num\_steps  sum = 0  for i in xrange(**begin, end**):  x= (i+0.5)\*step  sum = sum + 4.0/(1.0+x\*x)  return sum |

Step 2. Add MPI

|  |
| --- |
| **from mpi4py import MPI**  **…**  def Pi(num\_steps):  **comm = MPI.COMM\_WORLD**  **rank = comm.Get\_rank()**  **size = comm.Get\_size()**  … |

Step 4. Decompose the problem

|  |
| --- |
| def Pi(num\_steps):  comm = MPI.COMM\_WORLD  rank = comm.Get\_rank()  size = comm.Get\_size()  start = time.time()  **num\_steps2 = num\_steps/size**  **local\_sum = (num\_steps, rank\*num\_steps2, (rank+1)\*num\_steps2)**  ??????  ??????  ##(to be continued) |

The modified code above makes each process compute “local\_sum” from the allocated steps.

These “local\_sum”s from processes will need to be collected and added up to get the total “sum”.

Step 4. Collect results

In Example 3, two techniques that compute sum of values were demonstrated.

Complete the remaining of the function “Pi” such that local\_sum’s from processes are collected and the total “sum” is computed at Rank 0.

You may choose either approach – “send/recv” or “reduce”, it is advisable to use “reduce”. It is simpler, more efficient and it scales better.

|  |
| --- |
| ##(continued)  ??????  **sum = comm.reduce(local\_sum, root=0)**  end = time.time()  if rank == 0:  pi = sum / num\_steps  print "Pi with %d steps is %.20f in %f secs" %(num\_steps, pi, end-start) |

Step 5. Execute the program

|  |
| --- |
| $ mpirun –n 4 python ./pi.py |

**DISCUSS**

|  |
| --- |
| Try –n 2 and –n 4. How does it scale? |

# Advanced Topics

This tutorial presented some basic techniques that can boost the speed of Python programs.

Numba is a very simple Just-in-time compiler to boost the speed of a Python program. See [1] for more examples. Numba produces a native code automatically, but you can use Cython for more control. See [2]and [3]for more information on Cython.Some performance comparison was made and the difference appears to be very little [4].

MPI is very powerful and complex framework. For more information, see [5] for more advanced tutorial and examples. Also see [6], [7] for references.

While not covered in this tutorial, NumPy is one of the most important Python modules for scientific programming. A very nice tutorial is available online [8].

NumPy can be used in conjunction with Cython. See [2] for more info. NumPy depends on BLAS ([Basic Linear Algebra Subprograms](http://www.netlib.org/lapack/)) library, and if BLAS is built with multithreading support, it will automatically utilize multi-core CPU and do parallel computing for certain linear algebra calculations such as matrix multiplication[[4]](#footnote-4). If you identify that matrix multiplication is the bottleneck of the program, replacing BLAS library can give you a simple solution for parallel computing.

# References

|  |  |
| --- | --- |
| [1] | “Numba Examples,” [Online]. Available: http://numba.pydata.org/numba-doc/dev/examples.html. |
| [2] | S. Behnel, R. Bradshaw, W. Stein, G. Furnish, D. Seljebotn, G. Ewing and G. Gellner, “Cython Tutorial Release 0.15pre,” November 2012. [Online]. Available: http://115.127.33.6/software/Python/Cython/cython.pdf. |
| [3] | M. Perry, “A quick Cython introduction,” 19 April 2008. [Online]. Available: http://blog.perrygeo.net/2008/04/19/a-quick-cython-introduction/. |
| [4] | J. V. d. Plas, “Pythonic Perambulations,” 15 6 2013. [Online]. Available: http://jakevdp.github.io/blog/2013/06/15/numba-vs-cython-take-2/. [Accessed 25 4 2014]. |
| [5] | J. Bejarano, “A Python Introduction to Parallel Programming with MPI¶,” 2012. [Online]. Available: http://jeremybejarano.zzl.org/MPIwithPython/. |
| [6] | L. Dalcin, “MPI for Python v1.3 documentation,” 20 Jan 2012. [Online]. Available: http://mpi4py.scipy.org/docs/usrman/index.html. |
| [7] | Open MPI, “Open MPI v1.6.4 documentation,” 21 February 2013. [Online]. Available: http://www.open-mpi.org/doc/v1.6/. |
| [8] | SciPy.org, “Tentative NumPy Tutorial,” [Online]. Available: http://wiki.scipy.org/Tentative\_NumPy\_Tutorial. |

1. This statement is only true for CPython, which is the default, most-widely used implementation of Python. Other implementations like IronPython, Jython and IPython do not have GIL. <http://wiki.python.org/moin/GlobalInterpreterLock> [↑](#footnote-ref-1)
2. Recent development combined OpenMP with Cython and demonstrated how to use OpenMP from Python

   <http://archive.euroscipy.org/talk/6857> [↑](#footnote-ref-2)
3. Other available operations are MAX, MIN, PRODUCT, Logical AND, Logical OR etc.

   <http://www.open-mpi.org/doc/v1.4/man3/MPI_Reduce.3.php> [↑](#footnote-ref-3)
4. http://stackoverflow.com/questions/5260068/multithreaded-blas-in-python-numpy [↑](#footnote-ref-4)