A Hands-on Introduction to MPI Python Programming

Sung Bae, Ph.D

New Zealand eScience Infrastructure

1 INTRODUCTION: PYTHON IS SLOW

1.1.1 Example: Computing the value of π=3.14159…

For

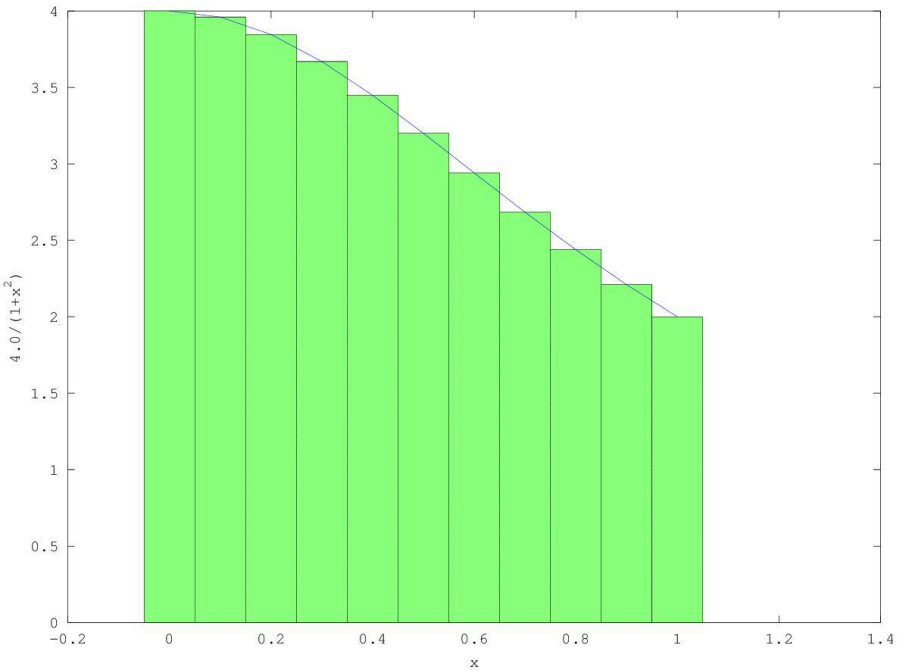
$$F(x) = \frac{4.0}{(1+x^2)}$$

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

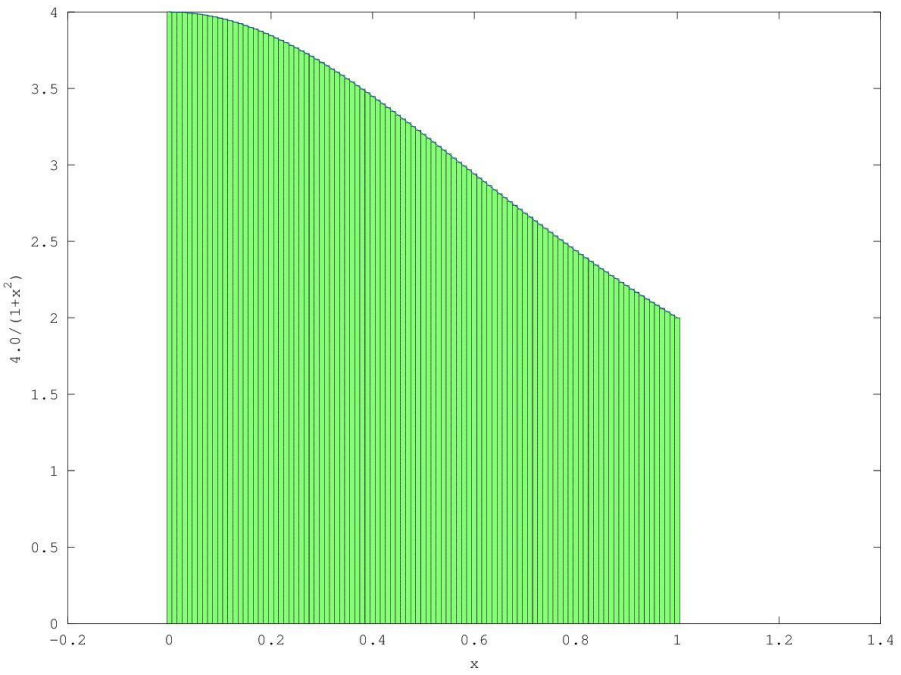
$$\int_0^1 F(x)dx = \pi$$

This can be approximated by

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$



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?

2 FASTER PYTHON CODE

2.1 SPEED-UP OPTIONS

2.2 EXAMPLE : NUMBA

2.2.1 Profiling

- Find what is slowing you down
    - Line-by-line profiling is often useful 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 "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

2.2.2 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?

3 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.

3.1 PARALLEL PROGRAMMING IN PYTHON

3.1.1 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.

3.1.2 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], and as a result, there is no OpenMP package for Python.[2]

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

3.1.3 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.

[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

[2] Recent development combined OpenMP with Cython and demonstrated how to use OpenMP from Python
http://archive.euroscipy.org/talk/6857

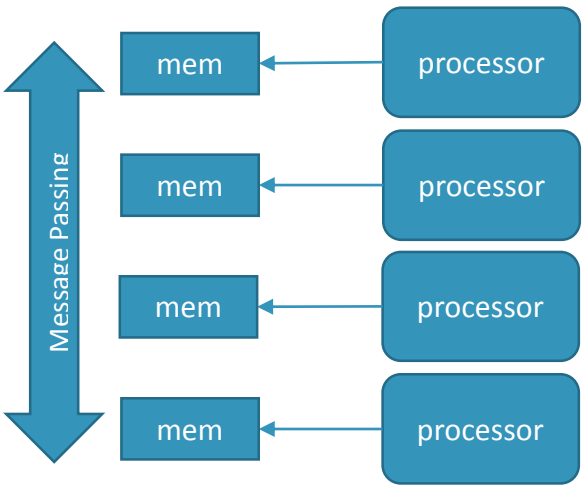


Figure 1. Distributed Memory

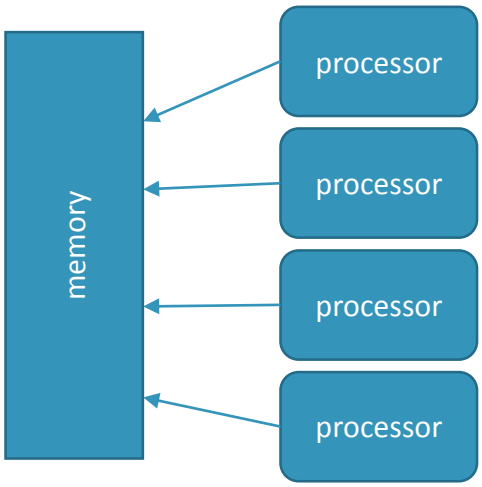


Figure 2. Shared Memory

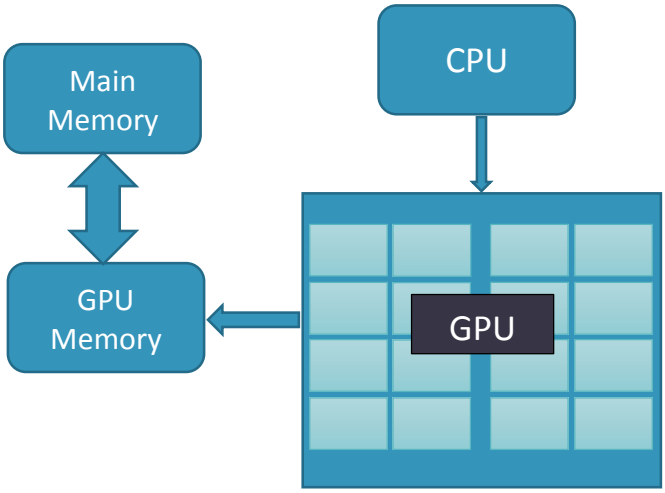


Figure 3. GPU

3.2 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
```

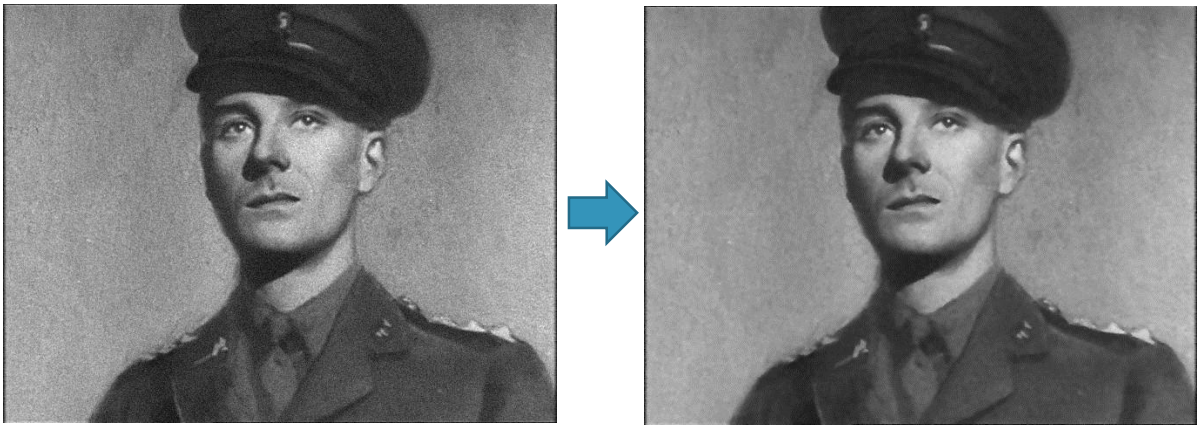
EXERCISE 1. *EMBARRASSINGLY* PARALLEL PHOTO PROCESSING

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)
```

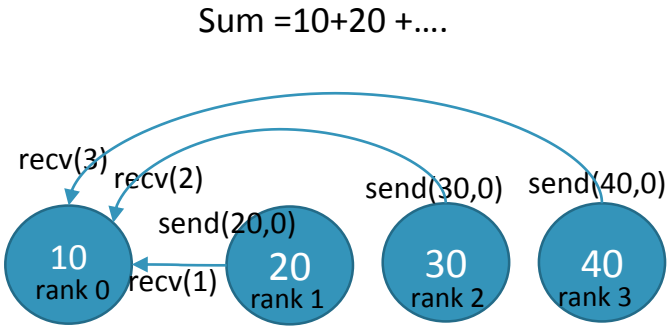


Figure 4. Computing Sum at Rank 0: Values received from Rank 1,2 and 3

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
```

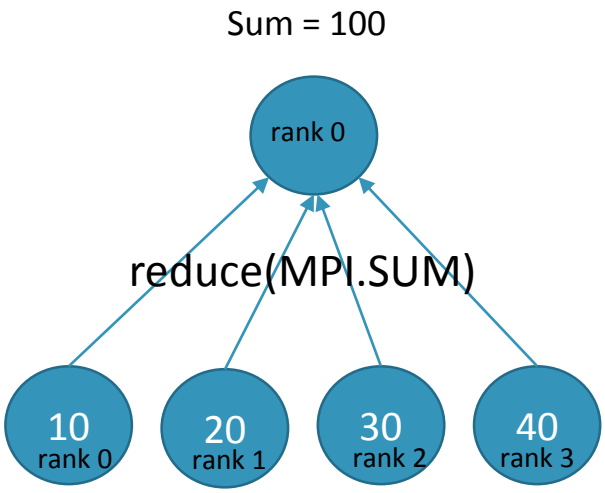


Figure 5. Computing Sum at Rank 0: All values collected and computed by "reduce"

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] 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)
```

[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

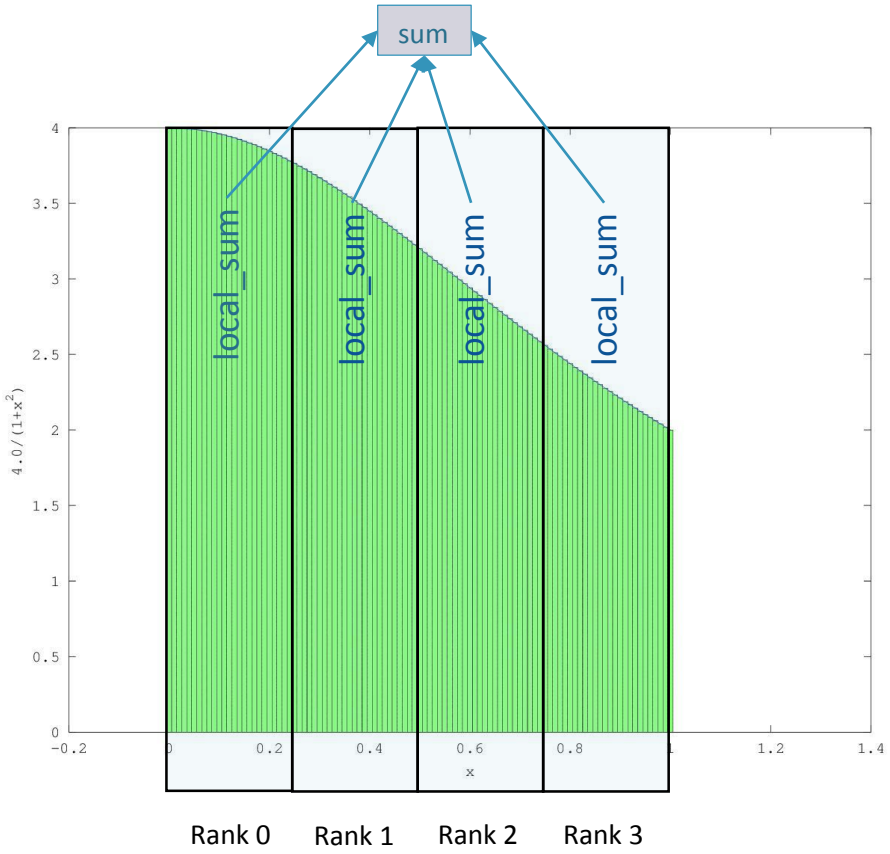


Figure 7 Computing total sum from local_sum's computed by processes

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, ?????? , ??????)
      end = time.time()
##(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 = ??????

      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?

4 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*) 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]. If you identify that matrix multiplication is the bottleneck of the program, replacing BLAS library can give you a simple solution for parallel computing.

5 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/.

[4] http://stackoverflow.com/questions/5260068/multithreaded-blas-in-python-numpy

[8] SciPy.org, “Tentative NumPy Tutorial,” [Online]. Available: http://wiki.scipy.org/Tentative_NumPy_Tutorial.