A Hands-on Introduction to Performance Python through Cython and mpi4py

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

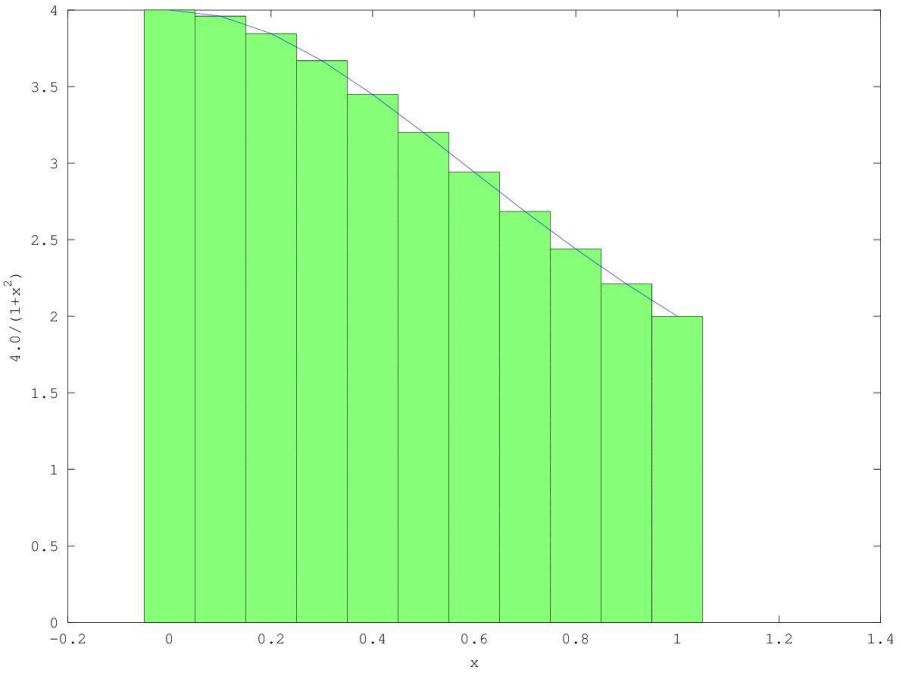
$$F(x) = \frac{4.0}{(1+x^2)}$$

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

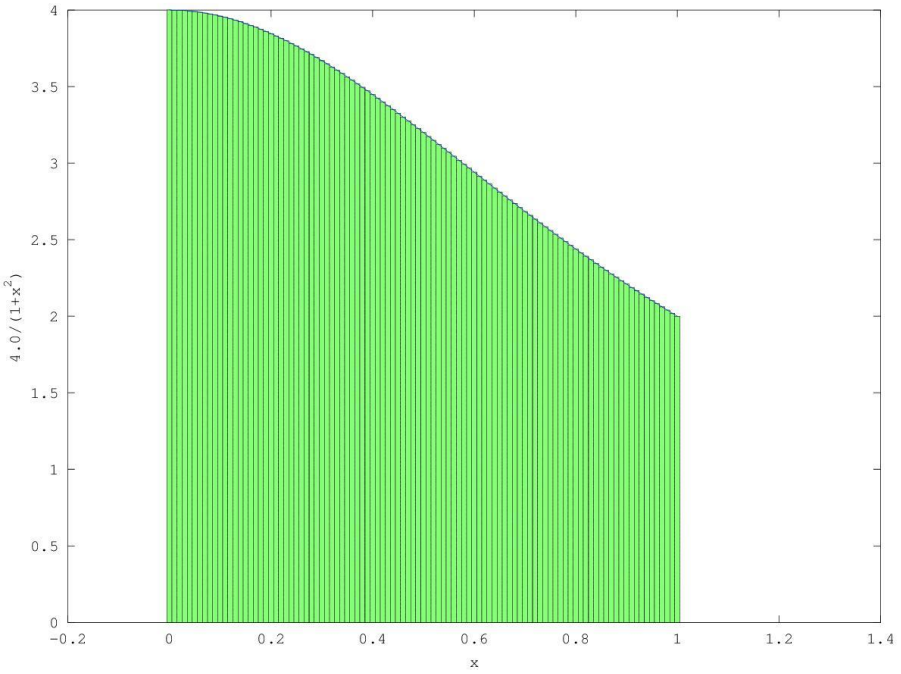
$$\int_0^1 F(x)dx = \pi$$

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 SPEED-UP OPTIONS

2.1 SPEED-UP THROUGH LITTLE EFFORT

- www.pypy.org
- Just-In-Time Python compiler
- Claims 5.5x faster
- Almost free speed-up : Just run with “pypy” instead of “python”
- Compatibility issue with some Python libraries
- No NumPy yet
- x86, ARM support (PowerPC in development)

Try and see if it works and suffices your needs.

- http://numba.pydata.org/index.html
- just-in-time specializing compiler which compiles annotated Python and NumPy code to LLVM (through decorators)
- Very new and promising - Two lines of code to get significant speed-ups
- Somewhat difficult to install
- x86 only

2.2 ORTHODOX OPTIMIZATION

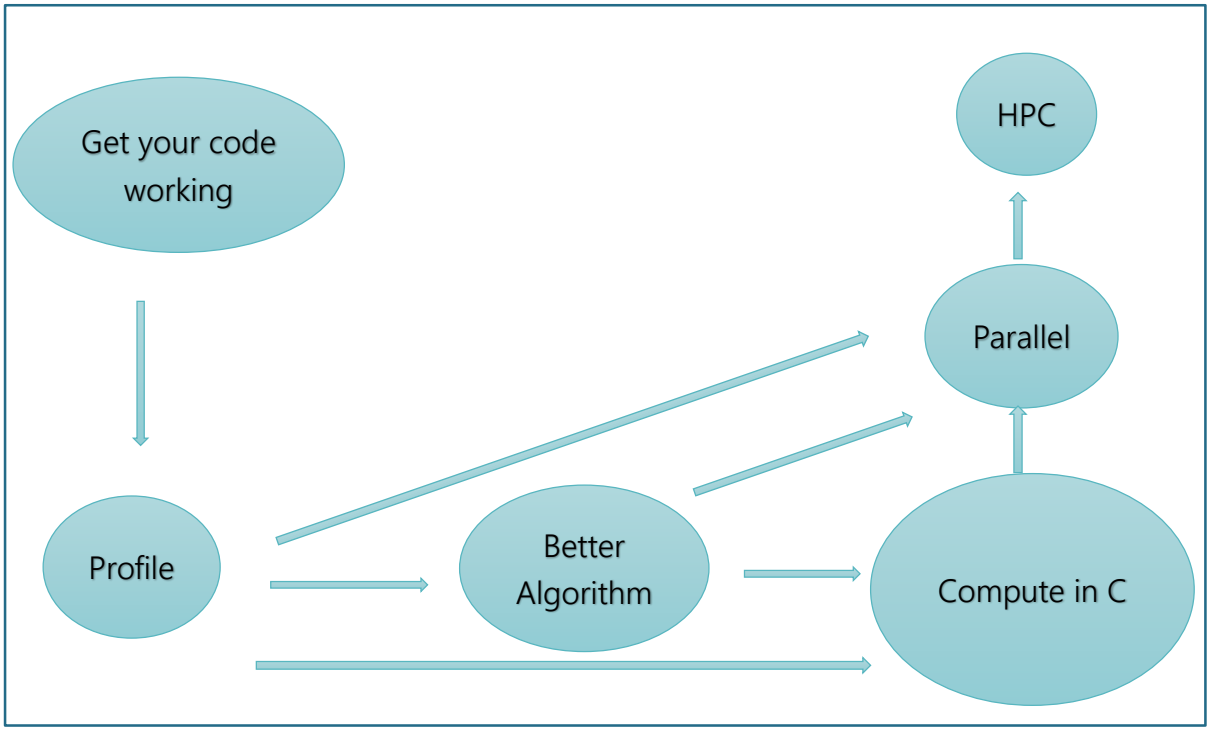


Figure 1 Steps to optimize Python programs

3 SPEED-UP EXAMPLE

3.1 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

3.2 BETTER ALGORITHM

Algorithm optimization sometimes rewards significant performance boost that exceeds the returns from other speed-up options. However, it is beyond the scope of this tutorial.

3.3 NUMPY

- Add-on modules to Python for mathematical/numerical routines
- Written in C and pre-compiled = fast

EXAMPLE

```
>>> from numpy import *
>>> i=arange(10)
>>> i
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> i+0.5
array([ 0.5,  1.5,  2.5,  3.5,  4.5,  5.5,  6.5,  7.5,  8.5,  9.5])
>>> x=(i+0.5)*10
>>> x
array([  5.,  15.,  25.,  35.,  45.,  55.,  65.,  75.,  85.,  95.])
>>> x.sum()
500.0
>>> x*x
array([   25.,   225.,   625.,  1225.,  2025.,  3025.,  4225.,  5625., 7225.,  9025.])
```

HANDS ON

1. The original Python code is on the left. Use the NumPy techniques above and complete the code on the right (Go to "pi_example" and modify "numpy_pi.py")

2. Execute the revised code and compare the performance of two versions

```
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)
```

```
import time
from numpy import *
def Pi(num_steps):
        start = time.time()
        step = 1.0/num_steps

        ??????

        (3 lines)

        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)
```

3.4 C EXTENSIONS

Rewriting in C will probably almost always result in some level of performance boost. But rewriting is an expensive job. Alternatively, you can leave the most Python code alone, and convert only *slow* routines into C. You know what routine is slow thanks to profiling. Typically you can follow steps below.

1. Isolate the bottleneck
2. Create a "C extension"
3. Import the C extension into the Python code

4. Execute the routine in C extension from the Python code.

Many Python modules we use (possibly unknowingly) are C extensions (eg. NumPy).

How do we know if a function is in a C extension?

EXAMPLE

```
>>> import Bakuriu
>>> import numpy
>>> Bakuriu.is_implemented_in_c(numpy.arange)
True
>>> import pi
>>> Bakuriu.is_implemented_in_c(pi.Pi)
False
```

Bakuriu.py[1] does a number of checks and determines whether a function is implemented in C or not.

As shown in the example above, numpy.arange function is already a C function, whereas pi.Pi is not.

If a function is slow and it is not implemented in C, creating a C extension is very likely to result in a significant speed-up. On the other hand, if a function is slow and it is already implemented in C, there is no point in pursuing the C extension option. In such a case, better algorithm or parallelism are the best options for speed-up.

3.4.1 Making C Extensions

- Hard way – Official Python way (not recommended)
  http://docs.python.org/2/extending/extending.html

- Easy way
  - SWIG (http://www.swig.org/Doc1.3/Python.html)
  - Pyrex (or Cython)
    (http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/)
    (http://www.cython.org/)

3.4.2 Cython

- Forked from Pyrex Invented by Greg Ewing of Uni. Canterbury
  http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/

- Very similar to Python syntax

- Makes writing C extensions for Python easy

[1] http://stackoverflow.com/questions/16746546/determine-whether-a-python-function-is-already-implemented-in-c-extension/

3.4.2.1 Cython Example

HANDS ON

Go to “pi_example/cython_pi”

STEP 1. SELECT THE AREA OF CODE TO CYTHON-IZE.

```
#pi.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. CREATE A .PYX FILE

```
#cython_pi.pyx
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
```

STEP 3. MODIFY THE ORIGINAL .PY FILE

```
#pi.py
import time
import cython_pi as cpi
def Pi(num_steps ):
   start = time.time()
   sum = cpi.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)
```

STEP 4. CREATE A SETUP.PY

(Already available in pi_example/cython_pi)

```
#setup.py
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext
ext_modules = [Extension("cython_pi", ["cython_pi.pyx"])]
setup(
    name = 'Cython Pi approximation app',
    cmdclass={'build_ext':build_ext},
    ext_modules = ext_modules
)
```

STEP 5. BUILD THE MODULE (.SO)

```
$ python setup.py build_ext --inplace
running build_ext
cythoning cython_pi.pyx to cython_pi.c
building 'cython_pi' extension
gcc -pthread -fno-strict-aliasing -O2 -DNDEBUG -g -fwrapv -O3 -Wall -
Wstrict-prototypes -fPIC -
I/usr/local/pkg/python/version/include/python2.7 -c cython_pi.c -o
build/temp.linux-ppc64-2.7/cython_pi.o
gcc -pthread -shared -Wl,--as-needed build/temp.linux-ppc64-
2.7/cython_pi.o -L. -lpython2.7 -o
/hpc/home/bfcsc10/workshop/pi_example/cython_pi/cython_pi.so
```

This step first generates cython_pi.c, an auto-translated C program, and builds a library cython_pi.so. This library can be imported into a Python program.

DISCUSS

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

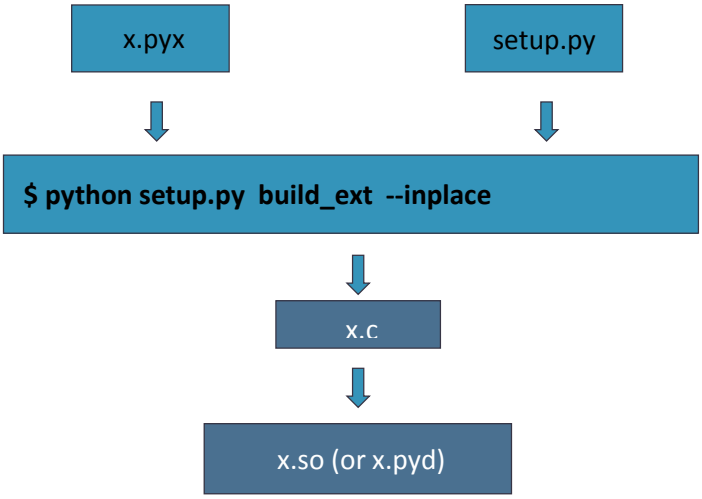


Figure 2 Cython workflow

STEP 6. STATIC TYPING

#cython_pi.pyx
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
	# cython_pi.pyx
	def loop(**int** num_steps):
	cdef int i
	cdef double sum, step, x
	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

When variables are statically typed, the auto-translated C code gets significantly optimized.

STEP 7. REPEAT STEP 5

DISCUSS

1. Execute the revised version of pi.py and compare its performance. Is it adequately improved? In comparison to the C program,
2. (Advanced) Cython code .pyx compiles with or without static typing. Ie. Not all variables need to be typed. Then which variable made the biggest speed improvement when static-typed?

4 PARALLEL PROGRAMMING

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

4.1 PARALLEL PROGRAMMING IN PYTHON

4.1.1 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.

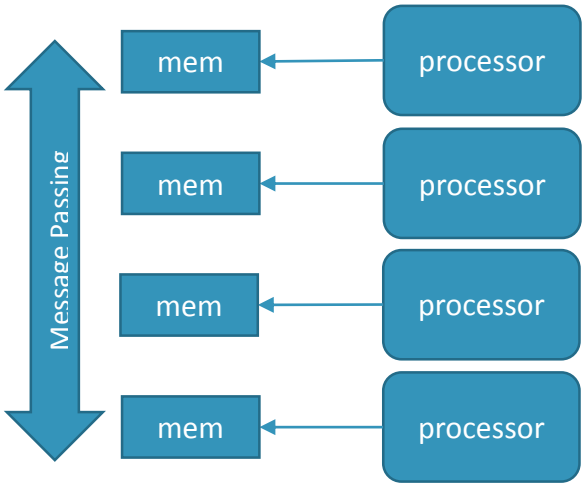


Figure 3 Distributed Memory

4.1.2 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[2], and as a result, there is no OpenMP package for Python.[3]

[2] 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

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

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

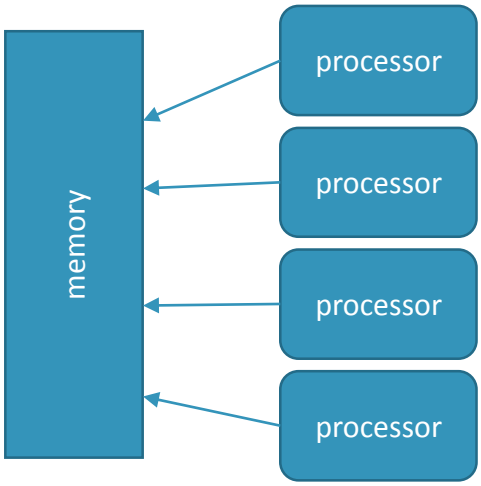


Figure 4 Shared Memory

4.1.3 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.

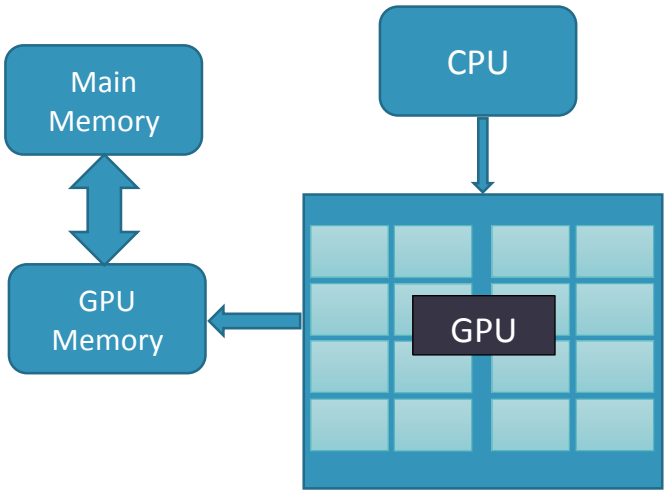


Figure 5 GPGPU

4.2 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

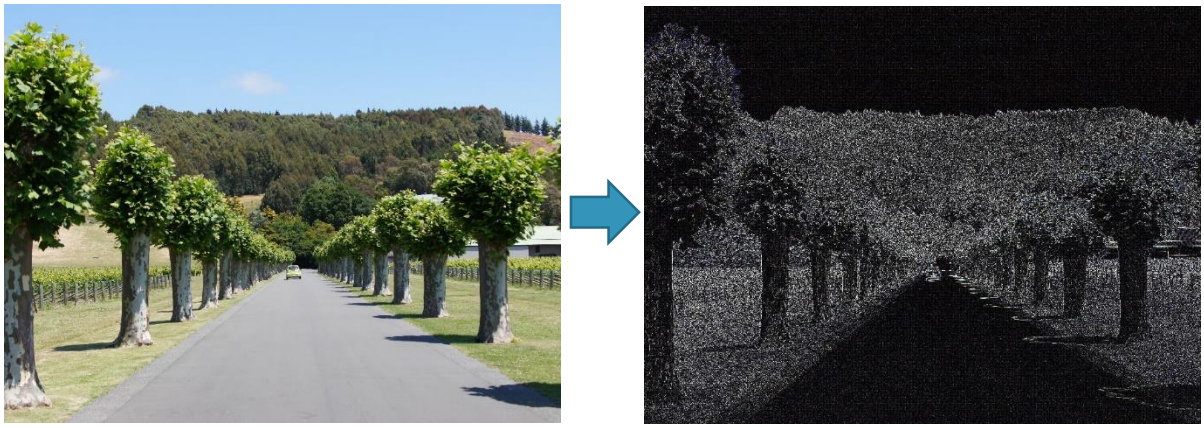
```
hello world from process 0/4
hello world from process 1/4
hello world from process 3/4
hello world from process 2/4
```

EXERCISE 1. PARALLEL PHOTO PROCESSING

The following program photoBatchProcess.py applies an image filter to the list of photos.

```
import os, sys
import Image, ImageFilter,ImageEnhance
import time

photos = ['bridge.jpg','airplane.jpg','swan.jpg','winery.jpg']

def imgProcess(img):

    img=img.filter(ImageFilter.FIND_EDGES)
    img = ImageEnhance.Brightness(img).enhance(5.0)
    return img


def serial() :
    begin=time.time()
    for photo in photos:
        im = Image.open('img/'+photo)
        im1=imgProcess(im)
        im1.save('img/new_'+photo,'JPEG')
    print "Took %f seconds" %(time.time()-begin)


if __name__=='__main__':
    serial()
```

The image filter processes a photo like below.



DISCUSS

How long does it take to process 4 photos?
What options can we use to speed-up? Can we use Cython to speed-up?

HANDS ON

Complete “parallel” function using MPI such that 4 photos can be processed in parallel

```
from mpi4py import MPI
photos = ['bridge.jpg','airplane.jpg','swan.jpg','winery.jpg']
…
def parallel() :
      comm = MPI.COMM_WORLD
      rank = comm.Get_rank()
      size = comm.Get_size() #size must be 4!!!

      begin=time.time()

#      DO SOMETHING HERE!!!
#      complete one line here
#      hint: we want Rank 0 to process 'bridge.jpg', Rank1 to process 'airplaine.jpg' etc.
      ?????? (1 line)
      im = Image.open('img/'+photo)
      im1=imgProcess(im)
      im1.save('img/new_'+photo,'JPEG')
      print "Took %f seconds" %(time.time()-begin)

if __name__=='__main__':
      parallel()
```

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

```
$ mpirun –n 4 python ./ photoBatchProcess.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)
comm.send(val, dest=0)
if rank ==0:
        sum = 0
        for i in range(size):
                sum += comm.recv(source=i)
        print "Rank 0 worked out the total %d" %sum
```

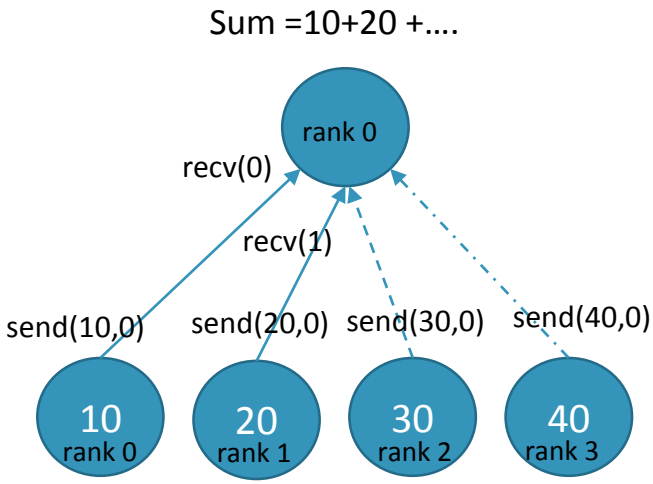


Figure 6 Computing Sum at Rank 0: Received values from Rank 0 and Rank 1

Each process sends a value to Rank 0 – Rank 0 sends 10, Rank 1 sends 20 etc.

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

OUTPUT

```
Rank 0 worked out the total 100
```

Note that Rank 0 “receives” from Rank 0, 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 be a big deal, but this can be very inefficient when we scale up.

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
```

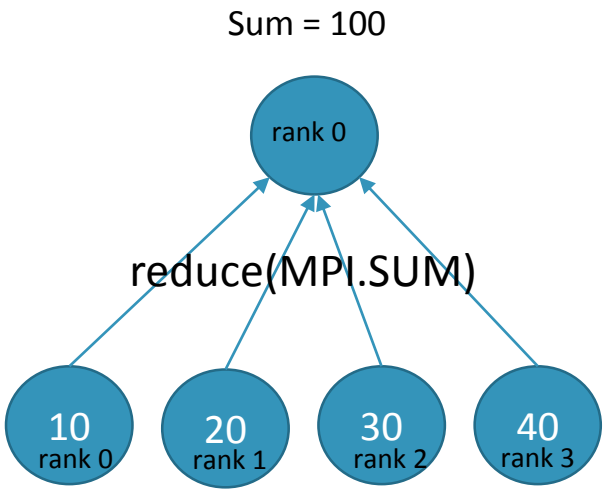


Figure 7 Computing Sum at Rank 0: All values collected and computed by "reduce"

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"[4] operation on all values. As a result, multiple values are *reduced* to one value.

EXERCISE 2 PARALLEL COMPUTATION OF PI

Let's revisit pi.py

We have identified the "for" loop was the bottleneck and created a C extension using Cython to speed-up.

```
#pi.py
import time
import cython_pi as cpi
def Pi(num_steps ):
    start = time.time()
    sum = cpi.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)
```

[4] 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

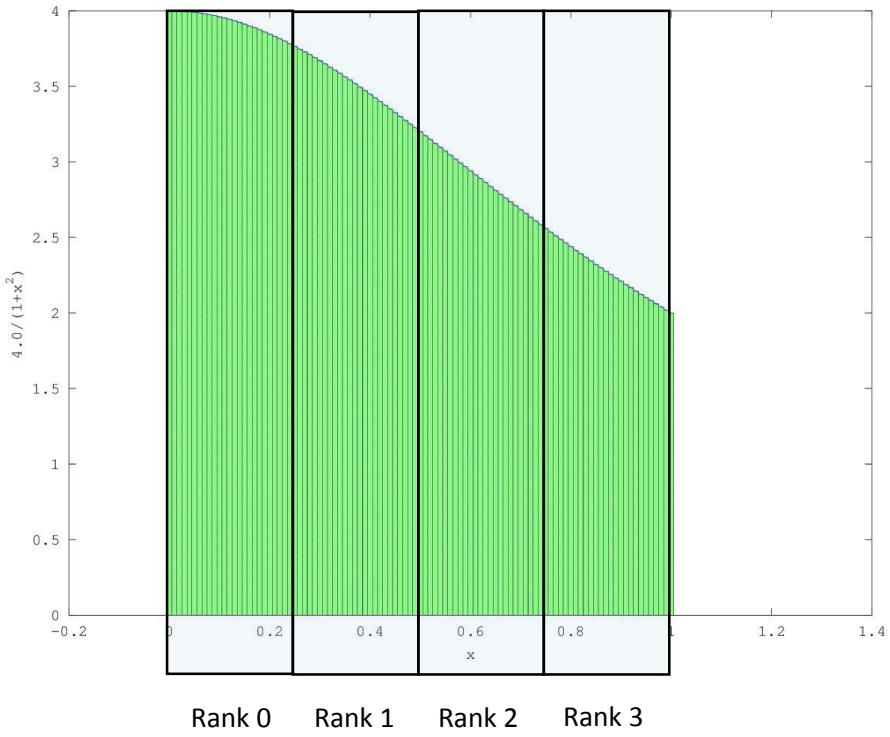


Figure 8 Allocating steps to processes

Here, num_steps=100000000, and cython_pi.loop will process this number "steps".

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 Rank2
- Steps [3*num_steps/4..num_steps] allocated to Rank 3

We need to modify cython_pi.pyx to accommodate this idea.

HANDS ON

STEP 1: MODIFY THE FOLLOWING CODE

```
#cython_pi.pyx
def loop(int num_steps, int begin, int end):
    cdef int i
    cdef double sum, step, x
    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. REBUILD THE MODULE (.SO)

```
$ python setup.py build_ext –inplace
…
/hpc/home/bfcsc10/workshop/pi_example/cython_pi/cython_pi.so
```

We reuse setup.py created in Section 3.4.2.1

STEP 3. PARALLELIZE PI.PY (PART I)

```
import time
from mpi4py import MPI
import cython_pi as cpi
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 = cpi.loop(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 need to be collected and added up to get the total “sum”.

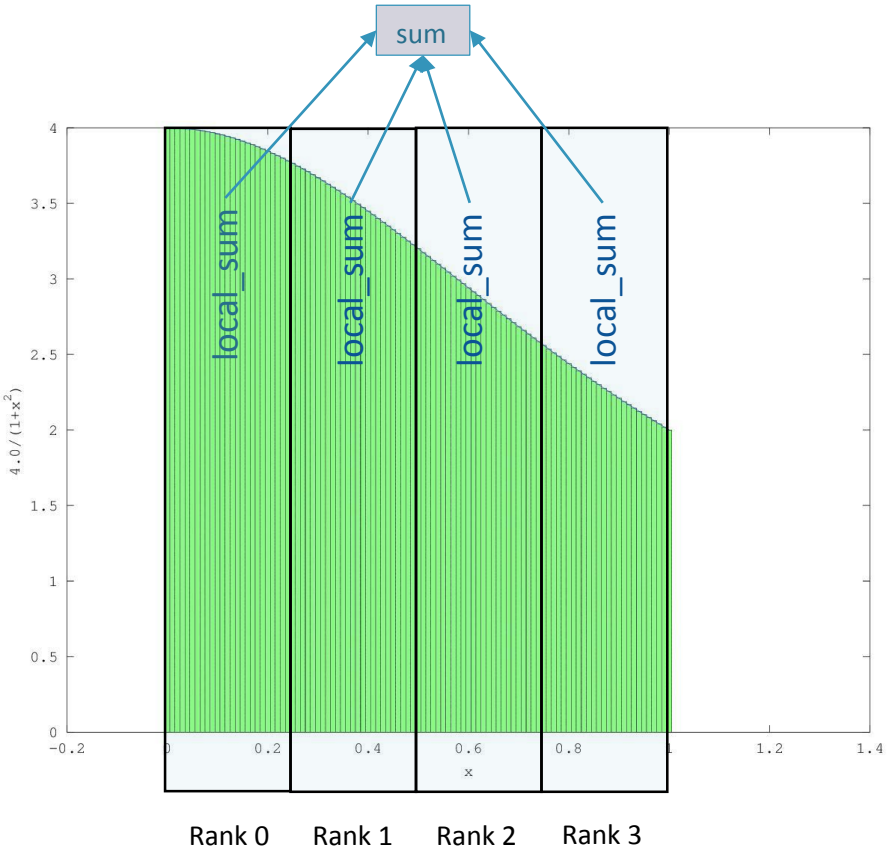


Figure 9 Computing total sum from local_sum's computed by processes

STEP 4. PARALLELIZE PI.PY (PART II)

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 competed 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

Comparing with the serial version of this program (in Cython), do you observe improvement in speed?

4.3 HIGH PERFORMANCE COMPUTING

New Zealand eScience Infrastructure (NeSI) operates HPC facilities that enable you to employ hundreds or thousands of processors to run your MPI Python programs.

5 ADVANCED TOPICS

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

In Cython, static typing can be applied to functions, and functions in C libraries can be called from .pyx code. See [1] for examples. More authoritative information can be found in [2].

MPI is very powerful and complex framework. For more information, see [3] for more advanced tutorial and examples. Also see [4], [5] for references.

While lightly covered in this tutorial, NumPy is one of the most important Python module for scientific programming. A very nice tutorial is available online [6]. NumPy can be used in conjunction with Cython. See [1] 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[5]. If you

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

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

6 REFERENCES

[1] 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.

[2] M. Perry, "A quick Cython introduction," 19 April 2008. [Online]. Available: http://blog.perrygeo.net/2008/04/19/a-quick-cython-introduction/.

[3] J. Bejarano, "A Python Introduction to Parallel Programming with MPI¶," 2012. [Online]. Available: http://jeremybejarano.zzl.org/MPIwithPython/.

[4] L. Dalcin, "MPI for Python v1.3 documentation," 20 Jan 2012. [Online]. Available: http://mpi4py.scipy.org/docs/usrman/index.html.

[5] Open MPI, "Open MPI v1.6.4 documentation," 21 February 2013. [Online]. Available: http://www.open-mpi.org/doc/v1.6/.

[6] SciPy.org, "Tentative NumPy Tutorial," [Online]. Available: http://wiki.scipy.org/Tentative_NumPy_Tutorial.