

UNIVERSIDAD ICESI
DEPARTAMENTO DE TECNOLOGÍAS DE INFORMACIÓN Y COMUNICACIONES
CURSO: SISTEMAS DISTRIBUIDOS
DOCENTE: DANIEL BARRAGAN C.
COMPLEMENTO TALLER 3 PARTE B

1. Objetivos

Capacitar al estudiante en el desarrollo de programas para ambientes distribuidos empleando MPI y el lenguaje de programación Python

2. Introducción

Python

Es un lenguaje de programación interpretado, multiparadigma (soporta programación orientada a objetos, programación imperativa y programación funcional), de tipado dinámico y multiplataforma. Posee una licencia de código abierto compatible con la licencia GNU.

El intérprete de Python estándar incluye un modo interactivo en el cual se escriben las instrucciones en una consola, las expresiones pueden ser introducidas una a una, pudiendo verse el resultado de su evaluación inmediatamente, lo que da la posibilidad de probar porciones de código en el modo interactivo antes de integrarlo como parte de un programa.

Python fue diseñado para ser leído con facilidad. Una de sus características es el uso de palabras donde otros lenguajes utilizarían símbolos. El contenido de los bloques de código (bucles, funciones, clases, etc.) es delimitado mediante espacios o tabuladores, conocidos como indentación, antes de cada línea de órdenes pertenecientes al bloque

Para leer el Zen de Python en una consola python escriba el comando ***import this***.

<http://www.python.org/>

MPI4PY

MPI4PY es una librería desarrollada para el lenguaje Python, la cual provee una interfaz para el desarrollo de aplicaciones usando el estándar de MPI. MPI4PY permite a un programa desarrollado en Python emplear múltiples núcleos.

MPI4PY provee soporte para las siguientes operaciones que describe el estándar MPI: comunicación punto a punto de objetos (sends, receives), comunicación colectiva de objetos (broadcast, scatters, gathers).

<http://mpi4py.scipy.org/>

<https://bitbucket.org/mpi4py/mpi4py/downloads>

3. Software Necesario

Para realizar las actividades de la siguiente guía se requiere la modificación de la receta para el aprovisionamiento automático de MPI del Taller 3 complemento A (actividad 3 ó actividad 4) teniendo en cuenta el aprovisionamiento de mpi4py por medio de los siguientes comandos en todos los nodos:

```
cd /home/mpiu
apt-get install python python-dev python-numpy -y
wget https://bitbucket.org/mpi4py/mpi4py/downloads/mpi4py-1.3.1.tar.gz
tar xzf mpi4py-1.3.1.tar.gz
cd mpi4py-1.3.1
python setup.py build
sudo python setup.py install
```

Cree un archivo de nombre **helloworld.py** y guárdelo en el directorio **/mirror**
nano /mirror/ helloworld.py

helloworld.py

```
#helloworld.py
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
print "hello world from process ", rank
```

Para ejecutarlo emplee el siguiente comando:

```
mpiexec -n 2 -f machinefile python /mirror/helloworld.py
```

Nota: Recuerde que el archivo **machinefile** debe tener los hostname de los nodos y la cantidad de hilos a emplear por nodo

4. Ejemplos Python-MPI

A continuación se presentan algunos ejemplos de Python en MPI.

separateCodes.py

```
#separateCodes.py
from mpi4py import MPI
rank = MPI.COMM_WORLD.Get_rank()

a = 6.0
b = 3.0
if rank == 0:
    print a + b
if rank == 1:
    print a * b
if rank == 2:
    print max(a,b)
```

```
mpiexec -n 2 -f machinefile python /mirror/separateCodes.py
```

passRandomDraw.py

```
#passRandomDraw.py
import numpy
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

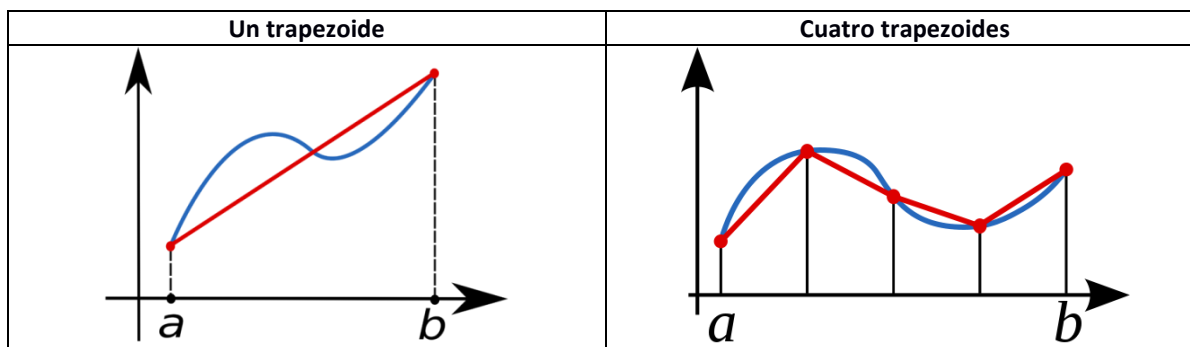
randNum = numpy.zeros(1)

if rank == 1:
    randNum = numpy.random.random_sample(1)
    print "Process", rank, "drew the number", randNum[0]
    comm.Send(randNum, dest=0)

if rank == 0:
    print "Process", rank, "before receiving has the number",
    randNum[0]
    comm.Recv(randNum, source=1)
    print "Process", rank, "received the number", randNum[0]
```

mpiexec -n 2 -f machinefile python /mirror/passRandomDraw.py

El siguiente código permite realizar la estimación de la integral x^2 empleando el método numérico de la regla trapezoidal. A continuación se muestra la derivación matemática empleada para la implementación del algoritmo (n es la cantidad de trapezoides):



$$\begin{aligned}
 \int_a^b f(x) dx &\cong \frac{b-a}{2n} (f(x_1) + 2f(x_2) + 2f(x_3) + \dots 2f(x_n) + f(x_{n+1})) \\
 &\frac{b-a}{n} \left(\frac{f(x_1)}{2} + \frac{2f(x_2)}{2} + \frac{2f(x_3)}{2} + \dots \frac{2f(x_n)}{2} + \frac{f(x_{n+1})}{2} \right) \\
 &\frac{b-a}{n} \left(\frac{f(x_1)}{2} + f(x_2) + f(x_3) + \dots f(x_n) + \frac{f(x_{n+1})}{2} \right) \\
 &\frac{b-a}{n} \left(\frac{f(x_1)}{2} + \frac{f(x_{n+1})}{2} + f(x_2) + f(x_3) + \dots f(x_n) \right) \\
 &\frac{b-a}{n} \left(\frac{f(x_1) + f(x_{n+1})}{2} + f(x_2) + f(x_3) + \dots f(x_n) \right) \\
 &\quad \text{Reemplazando con:} \\
 &\frac{f(x_1) + f(x_{n+1})}{2} = f(x_1) + f(x_{n+1}) - \frac{f(x_1) + f(x_{n+1})}{2} \\
 &\quad \text{Se tiene finalmente:} \\
 &\frac{b-a}{n} \left(-\frac{f(x_1) + f(x_{n+1})}{2} + f(x_1) + f(x_2) + f(x_3) + \dots f(x_n) + f(x_{n+1}) \right)
 \end{aligned}$$

trapSerial.py

```

#trapSerial.py
#example to run: python trapSerial.py 0.0 1.0 10000

import numpy
import sys

#takes in command-line arguments [a,b,n]
a = float(sys.argv[1])
b = float(sys.argv[2])
n = int(sys.argv[3])

def f(x):
    return x*x

def integrateRange(a, b, n):
    '''Numerically integrate with the trapezoid rule on the interval
    from
    a to b with n trapezoids.
    '''
    integral = -(f(a) + f(b))/2.0
    # n+1 endpoints, but n trapazoids
    for x in numpy.linspace(a,b,n+1):
        integral = integral + f(x)
    integral = integral* (b-a)/n
    return integral

integral = integrateRange(a, b, n)
print "With n =", n, "trapezoids, our estimate of the integral\
from", a, "to", b, "is", integral

```

trapParallel_1.py

```
#trapParallel_1.py
#example to run: mpiexec -n 4 python trapParallel_1.py 0.0 1.0 10000
import numpy
import sys
from mpi4py import MPI
from mpi4py.MPI import ANY_SOURCE

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

#takes in command-line arguments [a,b,n]
a = float(sys.argv[1])
b = float(sys.argv[2])
n = int(sys.argv[3])

#we arbitrarily define a function to integrate
def f(x):
    return x*x

#this is the serial version of the trapezoidal rule
#parallelization occurs by dividing the range among processes
def integrateRange(a, b, n):
    integral = -(f(a) + f(b))/2.0
    # n+1 endpoints, but n trapezoids
    for x in numpy.linspace(a,b,n+1):
        integral = integral + f(x)
    integral = integral* (b-a)/n
    return integral

#h is the step size. n is the total number of trapezoids
h = (b-a)/n
#local_n is the number of trapezoids each process will calculate
#note that size must divide n
local_n = n/size

#we calculate the interval that each process handles
#local_a is the starting point and local_b is the endpoint
local_a = a + rank*local_n*h
local_b = local_a + local_n*h

#initializing variables. mpi4py requires that we pass numpy objects.
integral = numpy.zeros(1)
recv_buffer = numpy.zeros(1)

# perform local computation. Each process integrates its own interval
integral[0] = integrateRange(local_a, local_b, local_n)

# communication
# root node receives results from all processes and sums them
if rank == 0:
    total = integral[0]
```

```

        for i in range(1, size):
            comm.Recv(recv_buffer, ANY_SOURCE)
            total += recv_buffer[0]
    else:
        # all other process send their result
        comm.Send(integral)

# root process prints results
if comm.rank == 0:
    print "With n =", n, "trapezoids, estimate of the integral from"\
        , a, "to", b, "is", total

```

`mpiexec -n 2 -f machinefile python /mirror/trapParallel_1.py 0.0 1.0 10000`

`trapParallel_2.py`

```

#trapParallel_2.py
#example to run: mpiexec -n 4 python26 trapParallel_2.py 0.0 1.0 10000
import numpy
import sys
from mpi4py import MPI
from mpi4py.MPI import ANY_SOURCE

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

#takes in command-line arguments [a,b,n]
a = float(sys.argv[1])
b = float(sys.argv[2])
n = int(sys.argv[3])

#we arbitrarily define a function to integrate
def f(x):
    return x*x

#this is the serial version of the trapezoidal rule
#parallelization occurs by dividing the range among processes
def integrateRange(a, b, n):
    integral = -(f(a) + f(b))/2.0
    # n+1 endpoints, but n trapazoids
    for x in numpy.linspace(a,b,n+1):
        integral = integral + f(x)
    integral = integral* (b-a)/n
    return integral

#h is the step size. n is the total number of trapezoids
h = (b-a)/n
#local_n is the number of trapezoids each process will calculate
#note that size must divide n
local_n = n/size

#we calculate the interval that each process handles
#local a is the starting point and local b is the endpoint

```

```

local_a = a + rank*local_n*h
local_b = local_a + local_n*h

#initializing variables. mpi4py requires that we pass numpy objects.
integral = numpy.zeros(1)
total = numpy.zeros(1)

# perform local computation. Each process integrates its own interval
integral[0] = integrateRange(local_a, local_b, local_n)

# communication
# root node receives results with a collective "reduce"
comm.Reduce(integral, total, op=MPI.SUM, root=0)

# root process prints results
if comm.rank == 0:
    print "With n =", n, "trapezoids, estimate of the integral from"\
, a, "to", b, "is", total

```

`mpiexec -n 2 -f machinefile python /mirror/trapParallel_2.py 0.0 1.0 10000`

`dotProductParallel_1.py`

```

#dotProductParallel_1.py
#"to run" syntax example: mpiexec -n 4 python26 dotProductParallel_1.py 40000
from mpi4py import MPI
import numpy
import sys

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

#read from command line
n = int(sys.argv[1])    #length of vectors

#arbitrary example vectors, generated to be evenly divided by the number of
#processes for convenience

x = numpy.linspace(0,100,n) if comm.rank == 0 else None
y = numpy.linspace(20,300,n) if comm.rank == 0 else None

#initialize as numpy arrays
dot = numpy.array([0.])
local_n = numpy.array([0])

#test for conformability
if rank == 0:
    if (n != y.size):
        print "vector length mismatch"
        comm.Abort()

    #currently, our program cannot handle sizes that are not evenly divided by
    #the number of processors
    if(n % size != 0):

```

```

        print "the number of processors must evenly divide n."
        comm.Abort()

    #length of each process's portion of the original vector
    local_n = numpy.array([n/size])

#communicate local array size to all processes
comm.Bcast(local_n, root=0)

#initialize as numpy arrays
local_x = numpy.zeros(local_n)
local_y = numpy.zeros(local_n)

#divide up vectors
comm.Scatter(x, local_x, root=0)
comm.Scatter(y, local_y, root=0)

#local computation of dot product
local_dot = numpy.array([numpy.dot(local_x, local_y)])

#sum the results of each
comm.Reduce(local_dot, dot, op = MPI.SUM)

if (rank == 0):
    print "The dot product is", dot[0], "computed in parallel"
    print "and", numpy.dot(x,y), "computed serially"

```

mpiexec -n 2 -f machinefile python /mirror/dotProductParallel_1.py 4000

```

#scatter.py
import numpy
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()
LENGTH = 3
#create vector to divide
if rank == 0:
    #the size is determined so that length of recvbuf evenly divides
    the
    #length of sendbuf
    x = numpy.linspace(1,size*LENGTH,size*LENGTH)
else:
    #all processes must have a value for x
    x = None
#initialize as numpy array
x_local = numpy.zeros(LENGTH)
#all processes must have a value for x. But only the root process
#is relevant. Here, all other processes have x = None.
comm.Scatter(x, x_local, root=0)
#you should notice that only the root process has a value for x that
#is not "None"
print "process", rank, "x:", x
print "process", rank, "x_local:", x_local

```


mpiexec -n 3 -f machinefile python /mirror/scatter.py

```
#scatterv.py
import numpy
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
if rank == 0:
    x = numpy.linspace(0,100,11)
else:
    x = None
if rank == 2:
    xlocal = numpy.zeros(9)
else:
    xlocal = numpy.zeros(1)
if rank == 0:
    print "Scatter"
comm.Scatterv([x, (1,1,9), (0,1,2), MPI.DOUBLE], xlocal)
print ("process " + str(rank) + " has " + str(xlocal))
comm.Barrier()
if rank == 0:
    print "Gather"
    xGathered = numpy.zeros(11)
else:
    xGathered = None
comm.Gatherv(xlocal, [xGathered, (1,1,9), (0,1,2), MPI.DOUBLE])
```

mpiexec -n 3 -f machinefile python /mirror/scatterv.py -u

En el directorio ***home/mpi/mpi4py-1.3.1/demo*** encontrará más ejemplos de mpi4py.

5. Actividades Propuestas


- a. Modifique el algoritmo de la regla trapezoidal para poder ser ejecutado en el caso en que la cantidad de trapezoides no sea divisible entre la cantidad de procesos. Tenga en cuenta realizar una distribución equitativa de los trapezoides sobrantes
- b. Modifique el algoritmo para calcular el producto punto para poder ser ejecutado en el caso en que el tamaño de los vectores no sea divisible entre la cantidad de procesos. Tenga en cuenta realizar una distribución equitativa de las tareas.


6. Referencias


<http://jeremybejarano.zzl.org/MPIwithPython/>


<http://lateblt.tripod.com/bit41.txt>

Anexo – Métodos MPI

<code>Comm.Send(buf, dest = 0, tag = 0)</code> 	
Performs a basic send. This send is a point-to-point communication. It sends information from exactly one process to exactly one other process.	
Parameters	Comm (<i>MPI comm</i>) – communicator we wish to query buf (<i>choice</i>) – data to send dest (<i>integer</i>) – rank of destination tag (<i>integer</i>) – message tag

<code>Comm.Recv(buf, source = 0, tag = 0, Status status = None)</code> 	
Basic point-to-point receive of data	
Parameters	Comm (<i>MPI comm</i>) – communicator we wish to query buf (<i>choice</i>) – initial address of receive buffer (choose receipt location) source (<i>integer</i>) – rank of source tag (<i>integer</i>) – message tag status (<i>Status</i>) – status of object

<code>Comm.Scatter(sendbuf, recvbuf, root)</code> 	
Sends data from one process to all other processes in a communicator (Length of recvbuf must evenly divide the length of sendbuf. Otherwise, use <i>Scatterv</i>)	
Parameters	sendbuf (<i>choice</i>) – address of send buffer (significant only at root) recvbuf (<i>choice</i>) – address of receive buffer root (<i>int</i>) – rank of sending process

<code>Comm.Scatterv([choice sendbuf, tuple_int sendcounts, tuple_int displacements, MPI_Datatype sendtype], choice recvbuf, root=0)</code> 	
Scatter data from one process to all other processes in a group providing different amount of data and displacements at the sending side	
Parameters	Comm (<i>MPI comm</i>) – communicator across which to scatter sendbuf (<i>choice</i>) – buffer sendcounts (<i>tuple_int</i>) – number of elements to send to each process (one integer for each process) displacements (<i>tuple_int</i>) – number of elements away from the first element in the array at which to begin the new, segmented array sendtype (<i>MPI_Datatype</i>) – MPI datatype of the buffer being sent (choice of sendbuf) recvbuf (<i>choice</i>) – buffer in which to receive the sendbuf root (<i>int</i>) – process from which to scatter

Anexo - Operaciones MPI

MPI.MAX	maximum
MPI.MIN	minimum
MPI.SUM	sum
MPI.PROD	product
MPI.LAND	logical and
MPI.BAND	bit-wise and
MPI.LOR	logical or
MPI.BOR	bit-wise or
MPI.LXOR	logical xor
MPI.BXOR	bit-wise xor
MPI.MAXLOC	max value and location
MPI.MINLOC	min value and location