High Level Parallel Processing

1. Communication between Programs
   - client/server interaction $\sim$ telephone exchange
2. Monte Carlo for π
   - a pleasingly parallel computation
   - server distributes seeds for random number generator
3. Forking Processes in Python
   - the multiprocessing module
   - estimating π, again
   - speedup and quality up

MCS 260 Lecture 40
Introduction to Computer Science
Jan Verschelde, 20 April 2016

High Level Parallel Processing

1. Communication between Programs
   - client/server interaction $\sim$ telephone exchange
2. Monte Carlo for π
   - a pleasingly parallel computation
   - server distributes seeds for random number generator
3. Forking Processes in Python
   - the multiprocessing module
   - estimating π, again
   - speedup and quality up

Analogy with Telephone Exchange

client/server communication with sockets

Analogy between a telephone exchange and sockets:

1. dial company on 1-312-666-9000
   connect to IP address 127.0.0.1
2. call answered by reception
   connection established to remote host
3. ask for computer center
   route using specified port (8732)
4. call answered by computer center
   server handles request from client
5. hang up the phone
   close sockets

Methods on Socket Objects

Most commonly used methods:

method	description
`accept()`	accepts connection and returns new socket for passing data
`bind()`	binds a socket to an address
`close()`	closes the socket
`connect(a)`	connects to address `a`
`listen(c)`	listend for connections
`recv(b)`	receives data in buffer of size `b`
`send(d)`	sends data in `d`

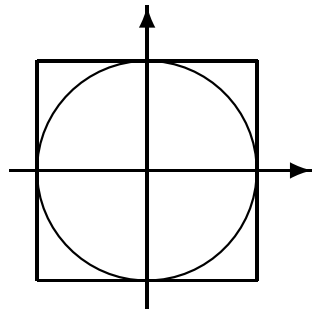
High Level Parallel Processing

1. Communication between Programs
   - client/server interaction $\sim$ telephone exchange
2. Monte Carlo for π
   - a pleasingly parallel computation
   - server distributes seeds for random number generator
3. Forking Processes in Python
   - the multiprocessing module
   - estimating π, again
   - speedup and quality up

Estimating π

Monte Carlo methods: throwing darts

The area of the unit disk is π:



Generate random uniformly distributed points with coordinates $(x, y) \in [-1, +1] \times [-1, +1]$.
We count a success when $x^2 + y^2 \leq 1$.

1. generate n points P in $[0, 1] \times [0, 1]$
2. $m := \{ (x, y) \in P : x^2 + y^2 \leq 1 \}$
3. the estimate is then $4 \times m/n$

$\rightarrow$ dual core processor: use two clients.
pleasingly parallel: optimal speedup (twice as fast)

Server and Client Terminals

The server distributes seeds for the random number generators.

```
$ python mc4pi2.py
server waits for connections...
server waits for results...
approximation for pi = 3.141487

$ python mc4pi_client.py
client is connected
client received 1
client computes 0.785253200000

$ python mc4pi_client.py
client is connected
client received 2
client computes 0.785490300000
```

High Level Parallel Processing

1. Communication between Programs
   - client/server interaction $\sim$ telephone exchange
2. Monte Carlo for π
   - a pleasingly parallel computation
   - server distributes seeds for random number generator
3. Forking Processes in Python
   - the multiprocessing module
   - estimating π, again
   - speedup and quality up

server distributes seeds: mc4pi2.py

```
...
SERVER.listen(2)

print 'server waits for connections...'
FIRST, FIRST_ADDRESS = SERVER.accept()
SECOND, SECOND_ADDRESS = SERVER.accept()

FIRST.send('1')
SECOND.send('2')
print 'server waits for results...'

PART1 = FIRST.recv(BUFFER)
PART2 = SECOND.recv(BUFFER)
RESULT = 2*(float(PART1)+float(PART2))
print 'approximation for pi =', RESULT
SERVER.close()
```

Code for the Client in `mc4pi_client.py`

```
from random import seed, uniform
from socket import socket as Socket
from socket import AF_INET, SOCK_STREAM

HOSTNAME = 'localhost'   # on same host
NUMBER = 11267           # same port number
BUFFER = 80              # size of the buffer

SERVER_ADDRESS = (HOSTNAME, NUMBER)
CLIENT = Socket(AF_INET, SOCK_STREAM)
CLIENT.connect(SERVER_ADDRESS)

print 'client is connected'
DATA = CLIENT.recv(BUFFER)
print 'client received %s' % DATA

seed(int(DATA))
```

code for the client continued ...

```
N = 10**7
CNT = 0
for i in range(0, N):
    xpt = uniform(0, 1)
    ypt = uniform(0, 1)
    if xpt**2 + ypt**2 <= 1:
        CNT = CNT + 1
R = float(CNT)/N
print 'client computes %.12f' % R

CLIENT.send(str(R))
CLIENT.close()
```

High Level Parallel Processing

1. Communication between Programs
   - client/server interaction $\sim$ telephone exchange
2. Monte Carlo for π
   - a pleasingly parallel computation
   - server distributes seeds for random number generator
3. Forking Processes in Python
   - the multiprocessing module
   - estimating π, again
   - speedup and quality up

the multiprocessing module

The multiprocessing module exports `Process`.

Working with an instance of `Process` is similar to threads:

1. A process is made with a new object instance.

   The parameters of the instantiation are
   - the function that the process will execute; and
   - values for the arguments of that function.

2. If the process is called `p`,
   then we start the process with `p.start()`.

   In the proper terminology, we *fork* a process.

3. To wait for a child process `p` to finish,
   the parent process does `p.join()`.

a process says hello

```
from time import sleep

def say_hello(name, tslp):
    """
    Process with name says hello,
    prints its process id, and
    sleeps for tslp seconds.
    """
    from os import getpid
    prt = 'hello from ' + name \
        + ' with process id ' + str(getpid())
    print prt
    print name, 'sleeps', tslp, 'seconds'
    sleep(tslp)
    print name, 'wakes up'
```

the main function

```
from multiprocessing import Process

def main():
    """
    Defines two processes.
    """
    from os import getpid
    print('the process id of main is', getpid())
    apr = Process(target=say_hello, args = ('A', 4))
    bpr = Process(target=say_hello, args = ('B', 1))
    print('starting two processes...')
    apr.start()
    sleep(1) # to make printing look nice
    bpr.start()
    print('waiting for processes to wake up...')
    apr.join()
    bpr.join()
    print('processes are done')
```

running the script

```
$ python hello_multiprocess.py
the process id of main is 20892
starting two processes...
hello from A with process id 20893
A sleeps 4 seconds
waiting for processes to wake up...
hello from B with process id 20894
B sleeps 1 seconds
B wakes up
A wakes up
processes are done
hit enter to close window
$
```

On Windows, run by double clicking on `.py` file.

High Level Parallel Processing

1. Communication between Programs
   - client/server interaction $\sim$ telephone exchange
2. Monte Carlo for π
   - a pleasingly parallel computation
   - server distributes seeds for random number generator
3. Forking Processes in Python
   - the multiprocessing module
   - estimating π, again
   - speedup and quality up

communication between processes

At the start, forking process `p`:

- Input to the function is passed as argument when the process is instantiated.

How to pass results computed in functions executed by a child process to the parent process?

The module multiprocessing exports `Queue`:

- An instance `q` of `Queue` is passed as input to the function executed by the child process.
- With `q.put(r)`, executed in the function, a result `r` is added to `q`.
- The parent process does `q.get()` to obtain the result(s) computed by the child process.

the multiprocessing module

The multiprocessing module provides two classes.

1. Objects of the class `Process` represent processes:
   - We instantiate with a function the process will execute.
   - The method `start` starts the child process.
   - The method `join` waits till the child terminates.

   In UNIX, we say: the main process *forks* a child process.
2. Objects of the class `Queue` are used to pass data from the child processes to the main process that forked the processes.

The two classes suffice for a simple manager-worker model:

- The manager distributes the work with functions.
- The workers are the child processes which run the functions.

processes report back via a queue

```
from multiprocessing import Process, Queue
from math import pi

def monte_carlo4pi(nbr, nsd, result):
    """
    Estimates pi with nbr samples,
    using nsd as seed.
    Adds the result to the queue q.
    """
    from random import uniform as u
    from random import seed
    seed(nsd)
    cnt = 0
    for _ in range(nbr):
        (x, y) = (u(-1, 1), u(-1, 1))
        if x**2 + y**2 <= 1:
            cnt = cnt + 1
    result.put(cnt)
```

defining, starting, and joining processes

```
def main():
    """
    Prompts the user for the number of samples
    and the number of processes.
    """
    nbr = int(input('Give the number of samples : '))
    npr = int(input('Give the number of processes : '))
    queues = [Queue() for _ in range(npr)]
    procs = []
    for k in range(1, npr+1):
        procs.append(Process(target=monte_carlo4pi, \
            args=(nbr, k, queues[k-1])))
    for process in procs:
        process.start()
    for process in procs:
        process.join()
    app = 4*sum([q.get()/nbr for q in queues])/npr
    print(app, 'error : %.3e' % abs(app - pi))
```

High Level Parallel Processing

1. Communication between Programs
   - client/server interaction $\sim$ telephone exchange
2. Monte Carlo for π
   - a pleasingly parallel computation
   - server distributes seeds for random number generator
3. Forking Processes in Python
   - the multiprocessing module
   - estimating π, again
   - speedup and quality up

speedup and quality up

We end with some observations:

1. Almost all computers have *multicore* processors, capable of executing multiple processes at the same time.
2. With a parallel algorithm we can achieve *speedup*: with p processors we may solve the same problem $p\times$ faster.

$$\text{speedup} = \frac{\text{time spent by one process}}{\text{time spent by } p \text{ processes}}$$

3. If we can afford the same amount of time for a problem, then we can ask how to solve the problem not faster, but *better*.

$$\text{quality up} = \frac{\text{quality obtained by } p \text{ processes}}{\text{quality obtained by one process}}$$

Quality often refers to accuracy and reliability.