STA 221: LECTURE 15

KRISHNA BALASUBRAMANIAN

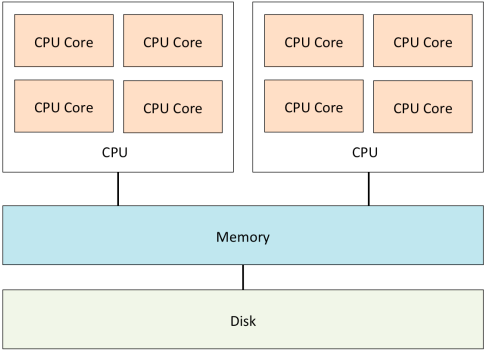
(University of California, Davis)

Outline

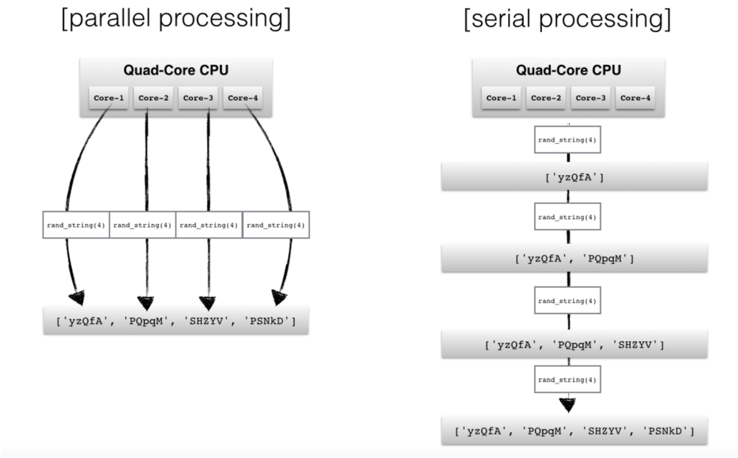
- Introduction to computer architecture
- Multi-core computing, distributed computing
- Multi-core computing tools
- Distributed computing tools

Computer Architecture

- Each computer can have multiple CPUs, each CPU has multiple cores
  (e.g., two quad-core CPUs)
- All the CPUs are connected to memory (e.g., 64G memory)
- CPU cores can execute in parallel

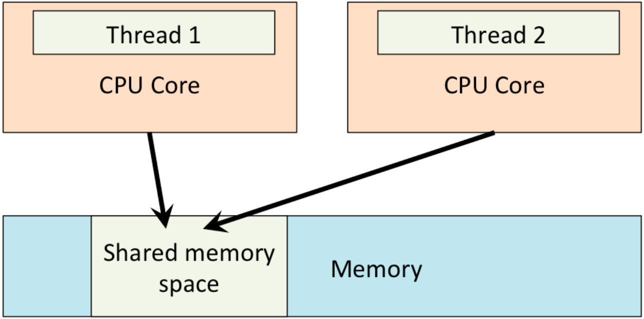


- Execute tasks simultaneously on many CPU cores



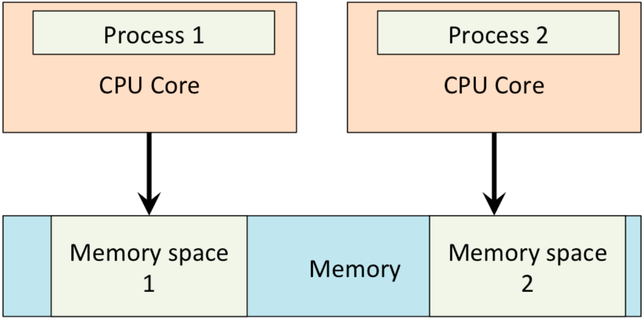
What is a thread?

- Multiple threads share the memory.
- Don't need inter-process communication.
- They are "light-weighted" (not much overhead to fork multiple threads)



What is a process?

- Processes are "share nothing"
  (independent executing without sharing memory or state)
- Easier to turn into a distributed application.



Python threads

- Package “threading”
- Unfortunately, python only allows a single thread to be executing at once

  (due to GIL (global interpreter lock))
- Usually no or little speedup

  only useful when you want to interleave I/O and CPU execution

- Package "multiprocessing"
- You can create multiple processes
  - Automatically run on multiple CPU cores
  - Default no shared memory, each process has its own memory space
    (larger memory overhead)
  - Can declare some part of memory to be shared
    (but often harder to use)
- You can also check some tutorials:
  - `http://sebastianraschka.com/Articles/2014_multiprocessing.html`
  - `https://pymotw.com/2/multiprocessing/basics.html`

Example: HelloWorld

```
import multiprocessing as mp

def helloworld(x):
    print ``Hello World %d\n''%x

# Setup a list of processes
plist = []
for x in range(4):
    plist.append(mp.Process(target=helloworld, args=(x,)))

# Run processes
for p in plist:
    p.start()

# Exit the completed processes
for p in plist:
    p.join()
```

Example: HelloWorld (output)

Output of the program:

```
Hello World 0

Hello World 1

Hello World 2

Hello World 3
```

- (Check `https://docs.python.org/2/library/multiprocessing.html`)
- "Process(target=helloworld, args=(x,))":
  - Specify the target function to run (helloworld)
  - Specify the input argument of the function (only one argument x)
  - Create an object belongs to "Process" type
- The process will run when execute "process.start()"
- The process will terminate when execute "process.join()"

Example: Exchanging objects using Queue

```
import multiprocessing as mp

def f(x,q):
    q.put(x**2)
    return

q = mp.Queue()
processes = []
for x in range(4):
    processes.append(mp.Process(target=f, args=(x,q)))

for p in processes:
    p.start()
for p in processes:
    p.join()

while (q.empty==False):
    print q.get()
```

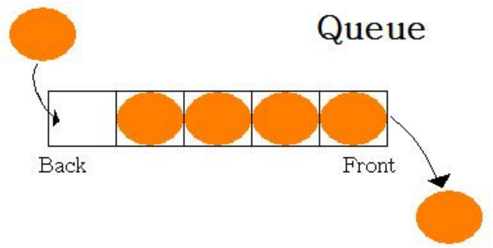
Example: Exchanging objects using Queue (Output)

Output of the program:

```
0
1
4
9
```

Use of Queue

- mp.Queue is a concurrent and "first in first out" data structure
- Can be used to communicate, or gather the results from the processes
- Queue.put(): insert an object to the end of queue
- Queue.get(): remove the first element in the queue
- Queue.empty: check whether the queue is empty



Using Pool

- Pool class is another and more convenient approach for parallel processing in python.
- Use "mp.Pool(processes=4)" to create 4 processes
- Use "$[r_1, r_2, \cdots, r_k]$ = pool.map(f, $[x_1, x_2, \cdots, x_k]$)" to run multiple processes and get the results
  - f is the function to run for the processes
  - $[x_1, \cdots, x_k]$ are the input arguments we want to run for the function (this is a size k list)
  - $[r_1, \cdots, r_k]$ are the output arguments we get after running the functions for each input (this is a size k list)
  - k may be larger than number of processes

Example: using Pool

```
import multiprocessing as mp

def f(x):
    return x**2

pool = mp.Pool(processes=4)
results = pool.map(f, range(4))
print results
```

Output of the program:

```
[0, 1, 4, 9]
```

Example: Compute sum of square

```
import multiprocessing as mp

def f(x):
    return x**2

pool = mp.Pool(processes=4)
results = pool.map(f, range(100))
print sum(results)
```

Output of the program:

```
328350
```

Parzen-window kernel density estimator:

$$p(x) = \frac{1}{n} \sum_{i=1}^{n} \frac{1}{h^2} \phi \left(\frac{x_i - x}{h} \right)$$

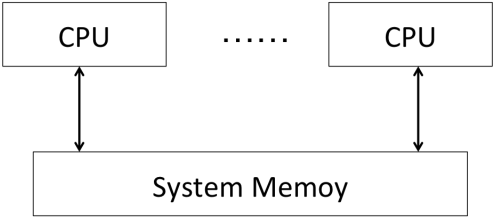
Parallel Implementation in Lecture15.ipynb

Parallel algorithms can be different in the following two cases:

- Shared Memory Model (Multiple cores)
  - Independent L1 cache
  - Shared/independent L2 cache
  - Shared memory
- Distributed Memory Model
  - Multiple processes in single machine
  - Multiple computers

Shared Memory Model (Multiple cores)

- Shared memory model: each CPU can access the same memory space
- Programming tools:
  - C/C++: openMP, C++ thread, pthread, intel TBB, ...

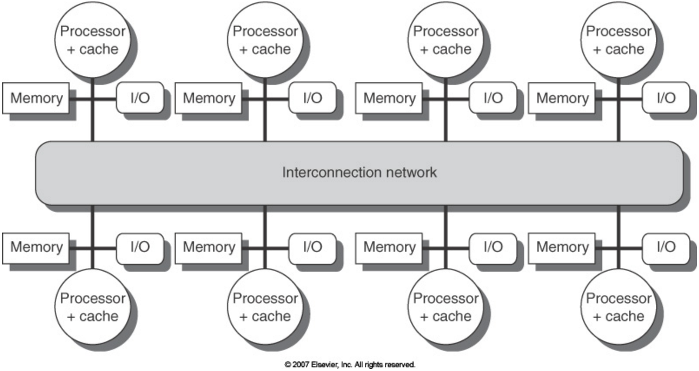


Parallel for loop in OpenMP

```
#pragma omp parallel for private(i)
    for(i=0;i<w_size;i++)
        g[i] = w[i] + g[i];
```

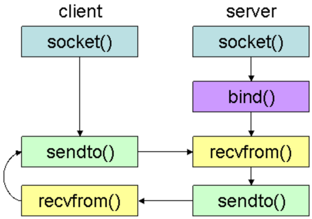
Distributed Memory Model

- Programming tools: MPI, Hadoop, Spark, ...



(Figure from http://web.sfc.keio.ac.jp/ rdv/keio/sfc/teaching/architecture/computer-architecture-2013/lec09-smp.html)

- Low-level programming:
  Socket programming
- Need to write code to send/receive sockets (messages) through network
  - Initialize socket in each processor
  - Sender send message "sendto()"
  - Receiver get message "recvfrom()"
- Use this only when you need very low level control



(Figure from https://people.eecs.berkeley.edu/~culler/WEI/

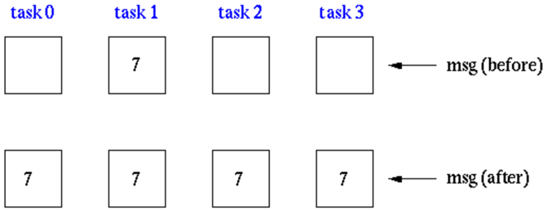
Message Passing Interface (MPI)

- An easier (and standard) way to pass messages in distributed systems
- C, python, ...
- Several types of "Collective Communication Routines"
  - Broadcast
  - Reduce
  - Gather, Allgather
  - ...
- Check `http://materials.jeremybejarano.com/MPIwithPython/`

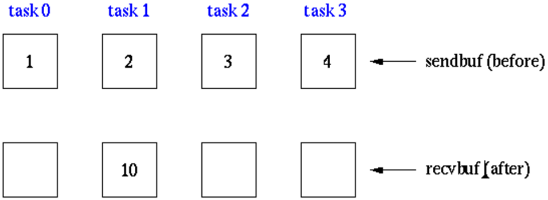
```
import numpy
from mpi4py import MPI
comm = MPI.COMM_WORLD

rank = comm.Get_rank()
rankF = numpy.array(float(rank))
total = numpy.zeros(1)
comm.Reduce(rankF,total, op=MPI.MAX)
```

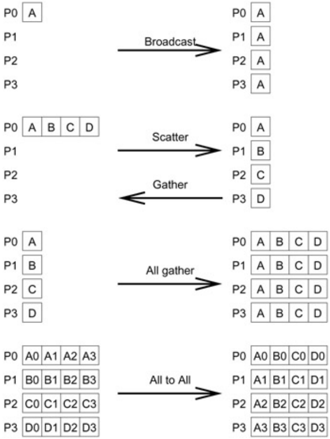
- MPI_Broadcast: Broadcasts a message to all other processes of that group



▷ MPI_Reduce: Reduces values on all processes to a single value (Can specify an operator, e.g., $+, -, \times, /$)



▷ Many other operations.



MapReduce Paradigm

- Framework for parallel computing
- Handles parallelization, data distribution, load balancing & fault tolerance
- Allows once to process huge amounts of data (terabytes & petabytes) on thousands of processors
- Google
  - Original implementation
- Apache Hadoop MapReduce
  - Most common (open-source) implementation
  - Built to specs defined by Google
- Amazon MapReduce
  - On Amazon EC2

MapReduce Paradigm

- Framework for parallel computing
- Handles parallelization, data distribution, load balancing & fault tolerance
- Allows once to process huge amounts of data (terabytes & petabytes) on thousands of processors
- Google

  Original implementation
- Apache Hadoop MapReduce

  Most common (open-source) implementation

  Built to specs defined by Google
- Amazon MapReduce

  On Amazon EC2

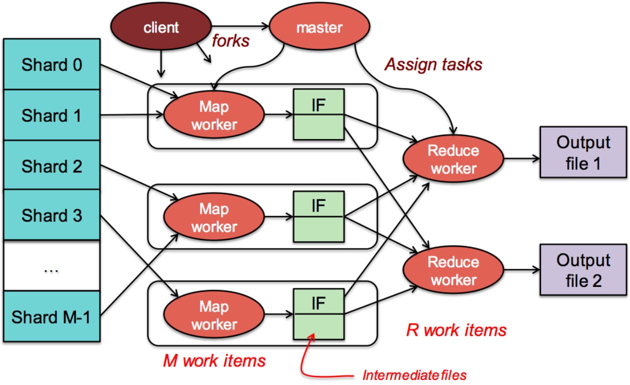
MapReduce concept

- Map
  - Grab the relevant data from the source
  - User function gets called for each chunk of input (key, value) pairs
- Reduce
  - Aggregate the results
  - Gets called for each unique key

MapReduce concept

- Map: (input shard) $\rightarrow$ (intermediate key, intermediate value)
  - Automatically partition input data to each computer
  - Group together all intermediate values with the same intermediate key
  - Pass to the reduce function
- Reduce: (intermediate key, intermediate value) $\rightarrow$ result files
  - Input: key, and a list of values
  - Merge these values together to form a smaller set of values
  - Automatically partition the reducers distributedly

MapReduce: the complete picture



(Figure from `https://www.cs.rutgers.edu/~pxk/417/notes/content/16-mapreduce-slides.pdf`)

Example

- Count number of each word in a collection of documents
- Map: parse data, output each word with a count (1)
- Reduce: sum together counts for each key (word)
- Mapper:

```
map(key, value):
// key: document name, value: document contents
for each w in value:
    output (w, '1')
```

- Reducer:

```
reduce(key, values):
// key: a word; values: a list of counts
for each v in values:
    result += Int(v)
output(result)
```

Example

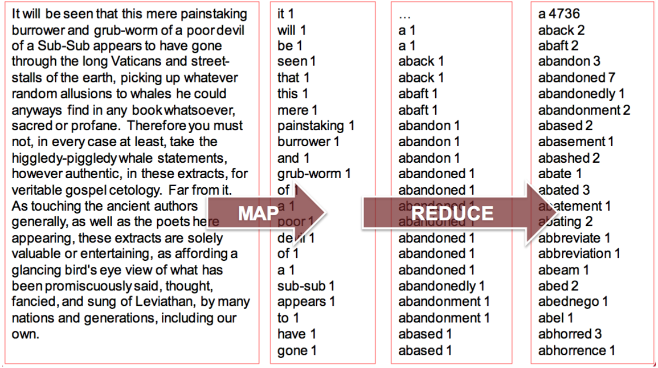
- Mapper:

```
for line in sys.stdin:
    line = line.strip.split()
    for word in words:
        print '%s\t%s'%(word,'1')
```

- Reducer:

```
word2count = {}
for line in sys.stdin:
    line = line.strip()
    word, count = line.split('\t', 1)
    word2count[word] = word2count[word]+count
for word in word2count.keys():
    print '%s\t%s'%(word,word2count[word])
```

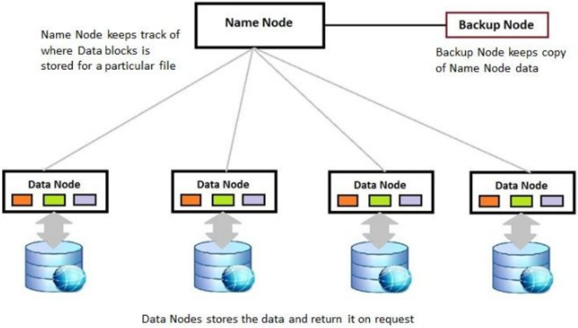
Example



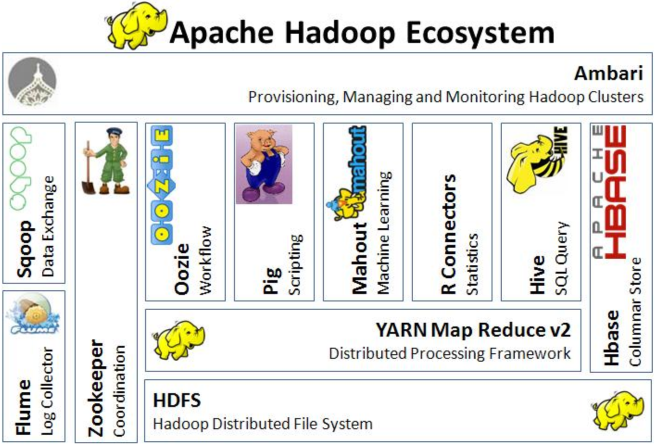
(Figure from https://www.cs.rutgers.edu/~pxk/417/notes/content/16-mapreduce-slides.pdf)

Hadoop Distributed File System (HDFS)

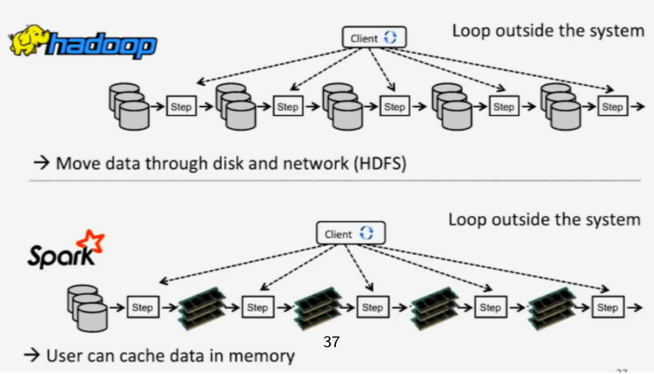
- The Hadoop Distributed File System (HDFS) is designed to store very large data sets on multiple servers.
- To use hadoop MapReduce, input/output files are in HDFS



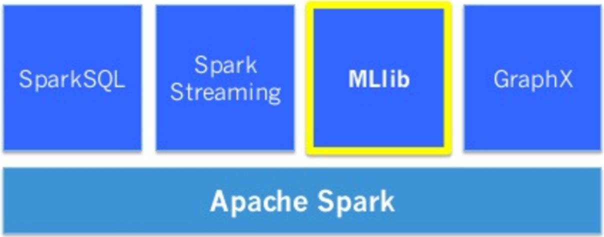
Hadoop Ecosystem



- Hadoop: Need to read/write HDFS for all the mapper/reducer
  - Main bottleneck is disk reading time
  - Not suitable for machine learning (iterative computing)
- Spark: Also a MapReduce framework with
  - In memory data flow
  - Optimize for multi-stage jobs



- Machine Learning using Spark: MLLib



Parameter Server

- A concept mainly for parallelizing machine learning algorithms (deep learning)
- Maintain a set of "shared parameters"
- Local machine communicate with parameter server to get the latest parameters