Lecture Notes for MACS 30123
Large-Scale Computing for the Social Sciences

Linghui Wu

November 28, 2020

Contents

1 Fundamentals of Large-Scale Computing

1.1 Introduction to Large-Scale Computing

Key Concerns in the Pursuit of Large-Scale Computing

1. Large-Scale Computation
2. Large-Scale Data

Solutions: Split the computation up into a series of parallel smaller computations.

Large-Scale Data have **4 Vs**

1. Volume: hard to store in a disk
2. Velocity: real-time data comes in fast
3. Variety: geographic, image information . . .
4. Variability: clean & process data in variable levels:

Solutions: Split data up, Process it in parallel streams and batches, Employ alternative flexible data structures, . . .

1.1.1 Parallel Computing Architectures

Serial Computation problems are broken up in a discrete set of instructions and instructions are executed sequentially, typically done in a single processor

Parallel Computation simultaneously use multiple computer resources to solve single computation tasks, typically done on different processors

Why Use Parallel Computing?

- Save time/money
- Solve larger problems
- Make better use of available hardware to achieve full tolerance for replication

Parallel Computer Architectures Flynn's Taxonomy describes the degree to which is computer is paralleled at hardware level.

- Single Instruction, Single Data (SISD)
- MISD, relatively rare but exists in space shuttles
- **SIMD**, special account for vectorization, e.g. GPU processes pixels in images
- **MIMD**, seen in multicore CPUs

1.1.2 General Considerations for Parallel Programming

High-level Parallel Programming Models

- Single Program, Multiple Data (SPMD), one of the most dominant programs for cluster computing
- MPMD

Designing Parallel Program

First, Understanding the Problem and the Program

- Where is most of the work being done?

- Can it be parallelized?

Easy to parallelize where we perform independent operations on a set of data

EX. Monte Carlo Simulations, Text Processing

Difficult to parallelize of anything with data dependencies, sequential operations

EX. Fibonacci Sequence $f(n) = f(n-1) + f(n-2)$

Then, Partitioning

- Domain Decomposition: Each processor gets even-sliced data to work on – equal team work.
- Functional Decomposition: Split up the system into subsystems onto different processors.

Finally, Assessing the Scalability of Parallel Solutions

- Amdahl's Law Speedup = $\frac{1}{1-P}$ where P = parallel fraction of work.
- Amdahl's Law Speedup = $\frac{1}{\frac{P}{N}+S}$ where P = parallel fraction of work, S = serial fraction of work & N = # of Processors.

But there are limits in the scalability due to serial parts.

- Gustafson's Law Scaled Speedup: to solve large problems rather than fixed problem size and the serial part becomes negligible $S + P \times N$.

1.2 Introduction to On-Premises CPU Clusters

1.2.1 CPUs and Resrarch Computing Clusters: Hardware Considerations

Central Processing Unit (CPU) is composed of:

- Memory – stores instructions and data;
- Control Unit – fetches instructions and data from the memory and decodes the instructions into commands;
- Execution Unit (Arithmetic Logic Unit, ALU) – executes the command and moves the results back into memory.

Multicore CPUs Each CPU has an associated individual memory and shared memory that across different cores.

Major CPU Processing Bottlenecks

- Compute: CPU can only execute certain #operations per cycle.

  Calculating Peak Performance $FLOP/s = Cores \times Hertz/Core \times FLOPs/Cycle$ where FLOP is the floating points of operations.

  EX. $FLOP/s = 4 \times 2.5GHz \times 4FLOPs/Cycle = 40b/s$ over the entire chip for the CPU.

- Memory Bandwidth: It takes longer than you expect for CPUs to access the memory and dramatically slows down the codes.

  The time for read and write data before and after performing the computation.

Memory Hierarchy (from smaller, faster, costlier to larger, slower, cheaper) CPU, Cache, Main Memory (RAM), Disk storage, Remote storage (e.g. Cloud, tapes)

Memory Bottleneck Takeaways

- Keep your data close to your CPU
- Increase the size of your faster memory sources, if necessary

Working on a Research Computer Cluster (RCC)

CPUs in clusters are organized by Nodes, where one or more processor fits into a Socket of the nodes.

Midway RCC Architecture (see the slides.)

Key point: avoid communications b/w processors as much as possible unless it is necessary.

1.2.2 Foster's Method (PCAM)

Foster's Method (PCAM)

- Partitioning: identify things that can be paralleled and divide the data & computation into smaller tasks.
- Communication: what communication needs to be carried out among the tasks.
- gglomeration or Aggregation: combine tasks and communication into larger composite tasks.

  Enhance concurrency – plays concurrentable tasks on different processors

  Increase locality – plays frequently-communicated tasks on the same processor.
- Mapping: assign the composite tasks to processors or threads.

Partitioning

- (n tasks) Train ML model using "n" different sets of hyperparameters
- (1 task) Identify optimal set of hyperparameters and model
- (m tasks) Use optimal model to predict archaeological site locations in "m" images in test dataset

Communications

- Training images, what hyperparameter values to work with -¿ training tasks
- Synchronize training tasks s.t. they are all finished training using their subset of hyperparameters
- Communicate b/w training tasks to identify where and what the optimal set of hyperparameters is
- Communicate optimal model configuration to all prediction tasks
- Test images -¿ prediction tasks

Aggregation

- Subset of n training tasks and communications can be aggregated on different processors
- Synchronization b/w training tasks, identify/broadcast optimal model to all the processors
- Make predictions on subsets of m test images, aggregated on different processors

Mapping

- On 10 CPU cores
- n/10 training tasks performed on each CPU core
- Synchronization, then MPI "All Reduce" to find model that produces maximum accuracy and broadcast it to all processors, also clears the cache to reserve faster memory space for making predictions on test images
- Read m/10 test data images onto each processor and make predictions using the model

1.2.3 Introduction to MPI and mpi4py

Goal communication on a distributed memory architecture

The MPI Standards

- A specification and NOT a single library
- Originally designed for distributed memory architectures, but works on hybrid memory architecture so that messages can be passed through CPU cores and distant nodes
- Programmer is responsible for identifying parallelism and implementing parallel algorithms - lower-level programming introduces errors

Programming MIMD systems

- MPI: Send, Receive. Codes on the sending processor will be different from the codes in the receiving processor.
- Commonly MPI programs use the SPMD model

```
if processor == 0:
    # send
elif processor == 1:
    # receive
```

```
# 'hello_world.py' Print out "Hello World" on different processors
from mpi4py import MPI
comm = MPI.COMM_WORLD # COMM_WORLD designation so all available processors communicate
    with each other
rank = comm.Get_rank() # Each processor has own local copy of rank that cannot be accessed
    by other processors
size = comm.Get_size() # Total # of processors

print("Hello World from rank", rank, "out of ", size, "processors.")
```

```
mpirun -n 4 python hello_world.py # The processors perform tasks in different order.
```

Programming SPMD Example

```
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))
```

Point-to-Point Communication (Arbitrary Python Objects)

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

# Send the dict from rank 0 to rank 1
if rank == 0:
    data = {'a': 7, 'b': 3.14}
    comm.send(data, dest=1)
```

```
elif rank == 2:
    data = comm.recv(source=0)
```

Point-to-Point Communication (NumPy Arrays)

```
from mpi4py import MPI
import numpy as np

rank = MPI.COMM_WORLD.Get_rank()

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

# Send the numpy array from rank 0 to rank 1
if rank == 0:
    data = np.arange(100, dtype=np.float)
    comm.Send(data, dest=1)
elif rank == 1:
    data = np.empty(100, dtype=np.float)
    comm.Recv(data, source=0)
```

Collective Communication - Broadcast(comm.Bcast()) Enable every processor to communicate with one another.

- broadcast a message to all the processors
- tree-based approach

```
from mpi4py import MPI
import numpy as np

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

# Send the numpy array from rank 0 to rank 1
if rank == 0:
    data = np.arange(100, dtype='i')
else:
    data = np.empty(100, dtype='i')
comm.Bcast(data, root=0) # `Bcast` is fast designed for `NumPy`, `bcast` is slow designed
     for all obj.
```

Scatter (comm.Scatter()) takes an array of elements and distributes these elements in an order of the processor's rank. It might serve as evenly partitioning the data so tasks can ben run on different portions.

Gather (comm.Gather()) collects the data back into a single processor.

Reduce (comm.Reduce()) can be used for calculating summary statistics and performing simulations.

Allreduce (comm.Allreduce()) simultaneously brings the maximum accuracy machine learning model to all processors at once.

1.3 On-Premises GPU-computing with OpenCL

1.3.1 Introduction to GPUs

Recall from last week that CPUs can be

- **Compute-bound** Deal with communication costs between different cores within one CPU in order to coordinate them
- **Memory Bandwidth-bound** Different levels of memory hierarchy have different access restrictions.

GPUs (Graphic Processing Units) are capable of

- High computational density and memory bandwidth
- Running 1000s of concurrent threads to hide latency (i.e. high throughput)

A Brief History

- Early 2000s: scientific applications via Graphics APIs
- 2006: NVIDIA launched CUDA
- 2008: Khronos Group defined OpenCL

CPU vs GPU Applications

CPUs consist of a few cores that are optimized for serial processing but they have memory-bandwidth problems.

GPUs consist of thousands of smaller cores designed for parallel computing such as graphic shading in the graphic pipeline, which are often ten-times the bandwidth with CPUs. However, each tiny GPU core is slower than a CPU core.

Performance

- CPUs measured in GigaFLOP/s
- GPUs measured in TeraFLOP/s

Trade-offs

- CPUs can multi-task and perform complicated sequential operations (MIMD + SIMD)
- GPUs tend to be able to run simple functions over large parallelized data (SIMD)

Will Execution on GPU Accelerate my Application?

- Computationally intensive
- Massively parallel
- Well-suited to GPU architectures

Some prime candidates for GPU Accelerations

- Elementwise linear algebra (addition, scalar multiplication)
- Image Processing: spread the work on pixels onto different GPU cores
- Monte Carlo simulations
- Brute-force optimization: run a set of possible hyperparameters onto different GPU cores
- Random Number Generatoin

1.3.2 Introduction to OpenCL and GPU Programming

Basic "formula" for GPU Programming

- Setup inputs on the host(CPU-accessible memory), which sends the functions and the in-format data from the kernel to GPUs

- Allocate memory for inputs on the GPU, and indicate where our data is actually going to go on the GPU devices
- Copy inputs from host to GPU (transfering from CPU to GPU memory is the major bottleneck)
- Allocate memory for outputs on the host
- Allocate memory for outputs on the GPU
- Start GPU kernels so they can run on the information being put on the GPUs
- Copy output from GPU to host

Working with OpenCL

First, identifying Host and Devices where Host tells each device what to do (e.g. send/receive data, execute kernel code) and OpenCL Devices are NVIDIA GPUs, AMD GPUs as well as Intel Xeon CPUs.

Once on the Compute Device, there are series of Compute Units which contain thousands of Processing Elements which allows GPU to perform operations concurrently.

(OpenCL Device Model) Each Processing Element has a certain amount of private memory and Compute Units have Local Memory shared by different Processing Elements. Altogether, the Compute Units have Constant Memory and Global Memory available to them.

Submmiting Work to Compute Devices

Kernels (Work Items) a function that run concurrently on the devices' processing units. The big idea is to replace loops with functions that can process chunks of data simultaneously on a variety of processing elements.

```
# Change:
for i in range(N):
   b[i] = a[i] * 2

# To:
def kernel(a, b)"
# Each ‘id’ == OpenCL work-item
   id = get_global_id(0)
   b[id] = a[id] * 2
```

Full OpenCL (Host) Program does the following things.

- Define platform and queues
- Define memory objects
- Create program
- Build the program
- Create and setup kernel
- Execute the kernel
- Read results on the host

1.3.3 Harnessing GPUs with PyOpenCL

A PyOpenCL Program

```
import pyopencl as cl
import numpy as np
```

```
ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)
```

Memory Allocation and Data Transfer

```
a = np.random.rand(50000).astype(np.float43)
a_buf = cl.Buffer(ctx, cl.mem_flags.READ_WRITE, size=a.nbytes)
cl.enqueue_copy(queue, a_buf, a)
```

Build Program, Write Kernel(s)

```
prg = cl.Program(ctx,"""
    __kernel void twice(__global float *a)
    {
        int gid = get_global_id(0);
        a[gid] = 2 * a[gid];
    }
    """).build()
```

Running a Kernel

```
prg.twice(queue, a.shape, None, a_buf)
```

Copy Data Back to Host

```
result = np.empty_like(a)
cl.enqueue_copy(queue, result, a_buf)
```

PyOpenCL Arrays

```
import pyopencl as cl
import pyopencl.array as cl_array
import numpy as np

ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)

a = np.random.rand(50000).astype(np.float32)
a_dev = cl_array.to_device(queue, a)

# Double all entries on GPU
twice = 2 * a_dev

# Turn back into Numpy Array
twice_a = twice.get()
```

Generating Random Numbers on Device

```
import pyopencl as cl
import pyopencl.clrandom as clrand
import numpy as np

ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)

n = 10 ** 6
a = clrand.rand(queue, n, np.float32)
b = clrand.rand(queue, n, np.float32)
```

Map Operations

```
import pyopencl as cl
import pyopencl.clrandom as clrand
import numpy as np

ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)

n = 10 ** 6
a = clrand.rand(queue, n, np.float32)
b = clrand.rand(queue, n, np.float32)

c1 = 5 * a + 6 * b
result_np = c1.get()
```

```
lin_comb = ElementwiseKernel(ctx,
    "float a, float *x, float b, float *y, float *c",
    "c[i] = a * x[i] + b * y[i]")
c2 = c1.array.empty_like(a)
lim_comb(5, a, 6, b, c2)
result_np = c2.get()
```

Reduce Operations

Associativity: $a \cdot (b \cdot c) = (a \cdot b) \cdot c$

Identity: $e \cdot a = a \cdot e = a$ where e is the neural operator

```
import pyopencl as cl
import pyopencl.clrandom as clrand
from pyopencl.reduction import ReductionKernel
import numpy as np

ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)

n = 10 ** 7
x = clrand.rand(queue, n, np.float64)

rknl = ReductionKernel(ctx, np.float64,
    neutral="0",
    reduce_expr="a+b", map_expr="x[i]*x[i]",
    arguments="double *x")

result = rknl(x)
result_np = result.get()
```

Scan Operations

It handles data dependencies that might emerge in loops.

```
import numpy as np

np.cumsum([1, 2, 3])
np.arrary([1, 3, 6])
```

```
import numpy as np
import pyopencl as cl
import pyopencl.clrandom as clrand
from pyopencl.scan import GenericScanKernel

np.cumsum([1, 2, 3])
np.array([1, 3, 6])

n = 10 ** 7
x = clrand.rand(queue, n, np.float32)

ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)
sknl = GenericScanKernel(ctx, np.float32,
   arguments="double *y, double *x",
   input_expr="x[i]",
   scan_expr="a+b",
   neutral="0",
   output_statement="y[i+1] = item")

result = cl.array.empty_like(x)
sknl(result, x)
result_np = result.get()
```

2 Architecting Computational Social Science Data Solutions in the Cloud

2.1 An Introduction to Cloud Computing and Cloud HPC Architectures

2.1.1 Introduction to the Cloud Landscape

Reasons you might want to scale up into the Cloud:

- Larger scale computing than in traditional RCC.
    1. access to the latest technology and customizability,
    2. scale up as many cores as possible
    3. wait time is shorter
- Virtually unlimited storage space spread across multiple data centers
- Immediate access to large datasets and models: replicable software, data and **hareware configurations**
- Adjustable Costs
- "Serverless" Architectures

For now, Amazon Web Service, Microsoft Azure and Google Cloud are the major players.

2.1.2 Fundamental AWS Vovabulary

AWS Regions Physical location around the world where data centers are clustered. Also konwn as availability zones or AZs. Each AWS region consists of multiple isolated physical AZs within a geographic area. 'us-east-1' will be used in class.

Availability Zones There are multilple availability zones within each region. The availablity zones consist of one or more dicrete data. All the AZs are inter-connected with high bandwidth and low latency networking, and the traffic between the connection is encrypted. The setup enables the application to be partitioned between different zones and the data is thereofore distribution in one given data center.

EC2 Instance - Elastic Compute Cloud Instance Whatever the service is, it is expandable, adaptable and customizable. It is equilevant to a node in Midway RCC. The default operation system is Amazon Linux distribution. Data cannot be persistent on an EC2 instance

AMI - Amazon Machine Image Machine set-ups and programs can be saved on AMIs, which is convenient for academic replications.

EBS - Elastic Block Storage Data can be retained longer on EBS disk storage. EBS can be connected and disconnected b/w different EC2 instances.

S3 - Simple Storage Service Keep data for a long while. The fundamental location where data and funciton of any size and are stored persistently in structure. S3 is replicated across multiple availability zones automatically. S3 can make data public so the data can be shared with the scientific community, e.g. the Common Crawl. The unit is S3 bucket.

Auto-Scaling Amazon will automatically dignosed when is programming will be benefited by adding more EC2 instances, e.g. Zoom and Netflix.

Putting it all together.

- Single-node Computing Workflow
- Multi-node Computing Workflow

Abstracting hardware considerations all away: ”Serverless” Computing

lambda Instead of setting EC2 instances and etc., we only need write a function using AWD lambda server which automatically transforms the data according to your function and taking care of all the hardware considerations underneath, even auto-scaling when necessary. Note that lambda only runs on EC2 CPU.

2.1.3 Bursting HPC into Cloud

PyWren A prototype approach for scaling out “map” style tasks for lambda. Send python functions from local machine and lambda will automatically scale whatever sizes it needs to solve the problem. Results will be sent back to local machines and the entire parallel operations will be hided behind.

```
def my_function(b):
    x = np.random.normal(0, b, 1024)
    A = np.random.normal(0, b, (1024, 1024))
    return np.dot(A, x)

pwex = pywren.default_executor()
res = pwex.map(my_function, np.linspace(0.1, 100, 1000))
```

PyWren Scalability: Almost linear scaled workflow for the most part, computer performance improves as the number of workers increases and the memory operations does well.

```
import pywren
import numpy as np



def addone(x):
    return x + 1

pwex = pywren.default_executor()
```

```
xlist = np.arange(10)
futures = pwex.map(addone, xlist) # placeholders

print([f.result() for f in futures])
# The results will be blocked until the remote job is completed.
```

Applications includes but not limited to

- Common Crawl Keyword Search
- News Article Sentiment Analysis
- Normalized Vegetation Index (NDVI) Calculation on Landsat 8 Data

  NDVI tells the degree to which the certain image contains live green vegetation which is condusive in assessing the changes in argricultural and urbanization patterns

Bottom Line for (Current) Serverless HPC: Good for embarrasingly parallel solutions (Why bother with lower-level operations if we can use one python function to perform parallel and scale instantly?)

Limitations of (Current) Serverless HPC

- Non-mapping collective operations (Reduce, Scan, etc.)
- Low limit of simultaneous workers ($\sim$ 3k)
- Atomic tasks that last longer than 300s
- Function invocation overhead
- No GPU backends

2.2 Large-Scale Cloud Storage

2.2.1 S3 Data Lake Architectures

What is Distributed Storage?

A distributed storage system is an infrastructure that can split data across multiple physical servers and more than one data center. It typically takes the form of storage units with the mechanism for data synchronization and coordination between cluster nodes.

Distributed storage systems can store several types of data:

- Files: distributed file systems that allow devices to mount to virtual drives with the actual files distributed across several machines
- Block storage: stores data in volumes, alternative to files that provides slightly higher performance
- Objects: wraps data into objects identified by unique ID or hash

Advantages of Distributed Storage Systems:

- Scalability: adding more storage spaces by adding more storage nodes in the clusters
- Redundancy: storing more than one copy of the data for backup, disastory restoring and etc.
- Cost: cheaper to use commidity hardware storage for large-volumes of data
- Performance: better than a single server and enables massive access to parallel files

Distributed Storage Features

- Partitioning: the ability to distribute between cluster nodes and retrieve data from different nodes
- Replication: the ability to reproduce different copies of data across nodes
- Fault tolerance: the ability to retain availability to data even when one or more nodes breaks down
- Elastic Scalability: the ability to enable data users to receive more storages spaces they need by adding or moving more storage units in the cluster

CAP Theorem A distributed system cannot maintain all three - consistency (all cores in the clusters have the same copy of data at the same time), availabity (read and write data at all times) and partition tolerance (the ability to recover from a failure of partition that contains part of the data). Consistency is often given up to guarantee availability and partition tolerance.

S3 and CAP Theorem S3 offers eventual consistency model which may take time for changes to replicate across S3.

S3 Data Lake = the foundation for your cloud workflow

Working with S3 Data via Boto3

```
import boto3

# Initialize Boto Client for S3
s3 = boto3.resource("s3")

# Create bucket
s3.create_bucket(Bucket="my-bucket") # Universally unique name is requied

# Put objects into bucket
data = open("test.jpg", "rb")
s3.Bucket("my-bucket").put_object(Key="test.jpg", Body=data)
```

Performing SQL Database Queries in Place

S3 Select and **Athena** splits a cluster in serverless fashion and runs queries in the clusters. It is hard to change the observations because they are stored in objects and not in data points.

2.2.2 Large-Scale Database Solutions

What is a Database?

- Flat File Databases
  - Data stored in plain files
  - Separated via commas, tabs, etc.
  - Simple data
- Non-relational Database (No SQL)
  - Miniamlly structured and data can be stored as key-value pair
  - *EX.* MongoDB, Couchbase, Amazon DynamoDB
- Relational Database (SQL) **Structured Query Language (SQL)** A language that makes it possible to easily manage the requested data.
  - *Use Case* Customer's information, billing information, and images uploaded
  - *EX.* PostgreSQL, MSSQL, MySQL

How to Choose the Right Database? (See the YouTube video.)

Which Database to Use When? (idem)

2.3 Large-Scale Data Ingestion and Processing

2.3.1 Ingesting Streaming Data with Kafka and Kinesis

Increase thoughput to allow more data to move in a given time in the data ecosystem and decrease the bottleneck on the computation side.

The Apache Kafka Clusters: Add more **Brokers** in the cluster in order to increase the amount of data from producers instances to consumers instances.

Serverless Throughout Approach - Kinesis

See slides for Kinesis Architecture.

Kinesis can scale the input and output steam by changing the number of **Shards** engaged according to the demands.

```
import boto3

# Create a Kinesis stream
kinesis = boto3.client("kinesis")

kinesis.create_stream(
    StreamName="stream_name",
    ShardCount=1 # Only 1 shard for AWS Educate - 1M input and 2M output
    )

# Put data into a Kinesis stream
kinesis.put_record(
    StreamName="stream_name",
    Data=data,
    PartitionKey="patitionkey"
    )

# Get data from a Kinesis stream
shard_it = kinesis.get_shard_iterator(
    StreamName="stream_name",
    ShardId="shardId-000000000000",
    ShardIteratorType="LATEST"
    )["ShardIterator"]

while 1 == 1:
    out = kinesis.get_records(ShardIterator=shard_it, Limit=1)

    # Do something with streaming data

    # Set shard iterator to next position in the data stream
    shard_it = out["NextShardIterator"]
```

Applications in the Social Sciences

- Real-time feedback loops in digital experiments
- Deploying ”listeners”, already employed in stock analysis in financial decisions

2.3.2 The Map Reduce Framework

MPI Workflow

- Manually partition tasks
- Manually establish communication checkpoints between processors to ensure each processor is still online and to guarantee tolerance

Motivation Google Web Search

Needs

- Automatic parallelization at scale
- Optimize network and disk operations
- Handle machine failures

Fundamental MapReduce Unit: (Key, Value) *EX.* (Word, Count), (Student, Grade), (Employee, "Salary": salary, "Social Security Number": ssn), and so on...

Typical MapReduce Workflow

1. Read Data
2. Map
   - Extract relevant information from data
   - Output: (key, value) pairs
3. Reduce
   - Group (key, value) pairs by their keys and summarize/fileter/aggregate to obtain new value
   - Output: (key, value2)

```
# Word count pseudocode

def mapper(line):
    for word in line.split():
        output(word, 1)

def combiner(key, values):
    output(key, sum(values))

def reducer(key, values):
    output(key, sum(values))
```

MapReduce jobs requires one resource manager, **"master" process**, that splits data evenly among the workers and one or more **"worker" process**.

If a task crashes, retry on another process or if it fails repeated, end the job. Data is replicated on disk and it's slower to process the data in memory.

MapReduce is ideal for batch jobs such as calculating summary statistics.

Hadoop is an open-source implementation, while the challenge is that we need to manage the machines on clusters.

AWS Elastic MapReduce (EMR) is the manage application for MapReduce and we can only pay for what we use. EMR allows us to use S3 buckets for input and output operations instead of performing on local machines.

2.3.3 Writing MapReduce Programs in Python with mrjob

A “mrjob” script

```
from mrjob.job import MRJob
import re

WORD_RE = re.compile(r"[\w’]+") # Identify all the words in each line of text

class MRWordFreqCount(MRJob):

    def mapper(self, _, line):
        for word in WORD_RE.findall(line):
            yield (word.lower(), 1)

    def combiner(self, word, counts):
        yield (word, sum(counts))

    def reducer(self, word, counts):
        yield (word, sum(counts))

# Brief Refresher: Python Generators

def generator():
    for i in range(3):
        yield i
print(generator()) # <generator object generator at 0x7fa222ce64d0>

for i in generator():
    print(i)

l = [0, 1, 2]
print(l == [i for i in generator()])
sum([0, 1, 2]) == sum(generator())
```

MapReduce ”Steps” and ”Jobs”

1. Step: a single map -¿ combine -¿ reduce ”chain”

   Note that a step need not contain all three of map, combine, and reduce

2. Job: consists of one or more steps

A Multi-Step “mrjob” Script

```
from mrjob.job import MRJob
from mrjob.step import MRStep
import re

WORD_RE = re.compile(r"[\w’]+") # Identify all the words in each line of text

class MRMostUsedWord(MRJob):

    def steps(self):
        return [
            MRStep(mapper=self.mapper_get_words,
                   combiner=self.combiner_count_words,
                   reducer=self.reducer_count_words),
            MRStep(reducer=self.reducer_find_max_word)
        ]
    def mapper_get_words(self, _, line):
        for word in WORD_RE.findall(line):
            yield (word.lower(), 1)

    def combiner_count_words(self, word, counts):
        yield (word, sum(counts))

    def reducer_count_words(self, word, counts):
        yield None, (sum(counts), word)

    def reducer_find_max_word(self, _, word_count_pairs):
        # So yielding one results in key=counts, value=word
        yield max(word_count_pairs)

if __name__ == '__main__':
    MRMostUsedWord.run()

    # max() identifies the maximum *first* value in a tuple
    word_counts = [("cat", 1), ("bat", 2)]
    print(max(word_counts))

    # This time, let's get the word that occurs most frequently
    word_counts = [(1, "cat"), (2, "bat")]
    print(max(word_counts))
```

3 Large-Scale Data Analysis and Prediction

3.1 Introduction to Large-Scale Data Analysis and Prediction with Spark

3.1.1 Parallel Computing with Spark

Why is Spark so Fast?

First, it caches data in RAM on different individual nodes being processed.

Additionally, it persists data in memory for multiple operations it performs. Unlike MapReduce that transfering data between different mappers and reducers.

It also uses an optimized parallel execution model that automatically parallizes the programs.

Basic Unit of Spark: the Resilient Distributed Dataset (RDD)

With RDD, the data are partitioned across different nodes among clusters that can be operated on in parallel.

- Transformations: Any sort of operation that results in another RDD
  - Map
  - Filter
  - Sample
  - Group by key
  - Reduce by key
  - Join
  - ...
- Actions: Compute and report the value of the program results

  - collect
  - reduce
  - count
  - ...

Transformation is not really executed until Action is run. The reason why Spark does this is to train together a pipeline and produce the parallelized solutions. The order of transformations is what we need to think about to perform on the dataset.

RDD Transformation + Action

```
countsByAge = (df.groupBy("age").count())
```

3.1.2 Accelerating Spark with GPUs (Optional)

(See the YouTube video)

3.1.3 Web-Scale Graph Analytics with Apache Spark

(idem)

3.2 Introduction to Dask

3.2.1 Introduction to Dask

Dask automatically partitions the dataset and constructs optimized graphs, communications and computations before performing. Dask, in contrast to Spark, is a Python-native solution that builds in the Python data analysis ecosystem.

Existing Python ecosystem includes but not limited to ‘Numpy’, ‘Pandas’ and ‘Scikit-Learn’ which Dask is based on and scales up.

```
# Numpy
import numpy as np
x = np.ones((1000, 1000))
print(x + x.T - x.mean(axis=0))

import dask.array as da
x = da.ones((1000, 1000))
print(x + x.T - x.mean(axis=0))

# Pandas
import pandas as pd
df = pd.read_csv("file.csv")
df.groupby("x").y.mean()

import dask.dataframe as dd
df = dd.read_csv("s3://*.csv")
df.groupby("x").y.mean()

# Scikit-Learn
from scikit_learn.linear_model import LogisticRegression
lr = LogisticRegression()
lr.fit(data, labels)

from dask_ml.linear_model import LogisticRegression
lr = LogisticRegression()
lr.fit(data, labels)
```

Under the hood, what Dask is actually doing is to partition data structures into small pieces that coordinated by the Dask scheduler.

Dask constructs the optimal task graphs and only executes the code after the task graphs are completed.

```
import dask.array as da
# Create a 1D dask array that splits into 5 chunks
x = da.ones(15, chunks=(5, ))
# Actually computes the object and returns the result

# More complicated tasks
x_sum = x.sum()
print(x_sum.compute())

x = da.ones((15, 15), chunks=(5, 5))
x_sum = x.sum(axis=0)
print((x + x.T).compute())
print(x.compute())
```

dask.delayed() Dask also allows users to conveniently set up customized codes as well that automatically assesses the codes provided by the user.

```
results = []
for x in A:
   for y in B:
      if x < y:
         results.append(f(x, y))
      else:
         results.append(g(x, y))

import dask
results = []
results = []
for x in A:
   for y in B:
      if x < y:
         results.append(dask.delayed(f(x, y)))
      else:
         results.append(dask.delayed(g(x, y)))
Results = dask.compute(results)
```

dask.delayed Decorators

```
# dask.delayed decorators
@dask.delayed
def inc(x):
   return x + 1

data = [i + 1 for i in range(4)]

output = []
output = [inc(x) for x in data]
dask.compute(output)
```

Dask on GPU Clusters

cuDF provides Pandas-like APIs that allows users to work on GPUs.

```
import cudf

df = cudf.read_csv("myfile.csv")
df = df[df.name == "Alice"]
df.groupby("id").value.mean()
```

Dask allows us to scale datasets up to a cluster of GPUs by setting up the optimized graph. 'dask_cudf' is what works under the hood to perform the necessary operations.

```
from dask_cuda import LocalCUDACluster
import dask_cudf
from dask.distributed import Client

cluster = LocalCUDACluster()
client = Client(cluster)

gdf = dask_cudf.read_csv("data/nyc-taxi/*.csv")
gdf.passenger_count.sum().compute()
```

Takeaways

The functionality of GPU on Dask is currently very limited but has lots of potential.

3.2.2 Natively Scaling the Python Ecosystem with Dask