MapReduce and Spark (and MPI)  
(Lecture 22, cs262a)
Context (1970s—1990s)

Supercomputers the pinnacle of computation

- Solve important science problems, e.g.,
  - Airplane simulations
  - Weather prediction
  - ...
- Large national racing for most powerful computers
- In quest for increasing power → large scale distributed/parallel computers (1000s of processors)

Question: how to program these supercomputers?
Shared memory vs. Message passing

**Shared memory**: all multiple processes to share data via memory

Applications must locate and map shared memory regions to exchange data

**Message passing**: exchange data explicitly via IPC

Application developers define protocol and exchanging format, number of participants, and each exchange
Shared memory vs.

Easy to program; just like a single multi-threaded machines

Hard to write high perf. apps:
  • Cannot control which data is local or remote (remote mem. access much slower)

Hard to mask failures

Message passing

Message passing: can write very high perf. apps

Hard to write apps:
  • Need to manually decompose the app, and move data

Need to manually handle failures
MPI - Message Passing Interface

- Library standard defined by a committee of vendors, implementers, and parallel programmers
- Used to create parallel programs based on message passing

Portable: one standard, many implementations

- Available on almost all parallel machines in C and Fortran
- De facto standard platform for the HPC community
Groups, Communicators, Contexts

**Group**: a fixed ordered set of $k$ processes, i.e., 0, 1, .., k-1

**Communicator**: specify scope of communication
- Between processes in a group
- Between two disjoint groups

**Context**: partition of comm. space
- A message sent in one context cannot be received in another context

This image is captured from: “Writing Message Passing Parallel Programs with MPI”, Course Notes, Edinburgh Parallel Computing Centre The University of Edinburgh
Synchronous vs. Asynchronous Message Passing

A **synchronous communication** is not complete until the message has been received.

An **asynchronous communication** completes before the message is received.
Communication Modes

**Synchronous:** completes once ack is received by sender

**Asynchronous:** 3 modes

- **Standard send:** completes once the message has been sent, which may or may not imply that the message has arrived at its destination
- **Buffered send:** completes immediately, if receiver not ready, MPI buffers the message locally
- **Ready send:** completes immediately, if the receiver is ready for the message it will get it, otherwise the message is dropped silently
Blocking vs. Non-Blocking

**Blocking**, means the program will not continue until the communication is completed

- Synchronous communication
- Barriers: wait for every process in the group to reach a point in execution

**Non-Blocking**, means the program will continue, without waiting for the communication to be completed
MPI library

Huge (125 functions)

Basic (6 functions)
Many parallel programs can be written using just these six functions, only two of which are non-trivial:

- MPI_INIT
- MPI_FINALIZE
- MPI_COMM_SIZE
- MPI_COMM_RANK
- MPI_SEND
- MPI_RECV
Skeleton MPI Program (C)

```c
#include <mpi.h>

main(int argc, char** argv)
{
    MPI_Init(&argc, &argv);

    /* main part of the program */

    /* Use MPI function call depend on your data partitioning and the parallelization architecture */

    MPI_Finalize();
}
```
A minimal MPI program (C)

```c
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    printf("Hello, world!\n");
    MPI_Finalize();
    return 0;
}
```
A minimal MPI program (C)

#include “mpi.h” provides basic MPI definitions and types.

MPI_Init starts MPI

MPI_Finalize exits MPI

Notes:
• Non-MPI routines are local; this “printf” run on each process
• MPI functions return error codes or MPI_SUCCESS
#include <mpi.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
    int rank, size;
    MPI_Init(&argc, &argv);
    /* rank of this process in the communicator */
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    /* get the size of the group associates to the communicator */
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("I am %d of %d\n", rank, size);
    MPI_Finalize();
    return 0;
}
/* Find out rank, size */
int world_rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);
int number;
if (world_rank == 0) {
    number = -1;
    MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
} else if (world_rank == 1) {
    MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("Process 1 received number %d from process 0\n", number);
}
Many other functions...

**MPI_Bcast**: send same piece of data to all processes in the group

**MPI_Scatter**: send different pieces of an array to different processes (i.e., partition an array across processes)

From: http://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/
Many other functions...

**MPI_Gather**: take elements from many processes and gathers them to one single process
  - E.g., parallel sorting, searching

From: http://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/
Many other functions...

**MPI_Reduce**: Takes an array of input elements on each process and returns an array of output elements to the root process given a specified operation.

**MPI_Allreduce**: Like MPI_Reduce but distribute results to all processes.

MPI Discussion

Gives full control to programmer
  • Exposes number of processes
  • Communication is explicit, driven by the program

Assume
  • Long running processes
  • Homogeneous (same performance) processors

Little support for failures, no straggler mitigation

Summary: achieve high performance by hand-optimizing jobs but requires experts to do so, and little support for fault tolerance
Today’s Papers

MapReduce: Simplified Data Processing on Large Clusters, Jeffrey Dean and Sanjay Ghemawat, OSDI’04

Spark: Cluster Computing with Working Sets,
Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica, NSDI’12
Context (end of 1990s)

Internet and World Wide Web taking off

Search as a killer applications
- Need to index and process huge amounts of data
- Supercomputers very expensive; also designed for computation intensive workloads vs data intensive workloads

Data processing: highly parallel
Bulk Synchronous Processing (BSP) Model

MapReduce as a BSP System

Partitions --> Maps --> Shuffle --> Partitions --> Reduce

Super-step (Map phase) --> Super-step (Reduce phase)
Example: Word Count
Context (2000s)

MapReduce and Hadoop de facto standard for big data processing ➔ great for batch jobs

… but not effective for

- Interactive computations
- Iterative computations
Spark, as a BSP System

- all tasks in same stage impl. same operations,
- single-threaded, deterministic execution

RDD

• Immutable dataset

Barrier implicit by data dependency

 Immutable dataset

Barrier implicit by data dependency

• all tasks in same stage impl. same operations,
• single-threaded, deterministic execution

 RDD

 tasks (processors) 

 Shuffle 

 RDD 

 tasks (processors)
Spark, really a generalization of MapReduce

DAG computation model vs two stage computation model (Map and Reduce)

Tasks as threads vs. tasks as JVMs

Disk-based vs. memory-optimized

So for the rest of the lecture, we’ll talk mostly about Spark
More context (2009): Application Trends

Iterative computations, e.g., Machine Learning
  • More and more people aiming to get insights from data

Interactive computations, e.g., ad-hoc analytics
  • SQL engines like Hive and Pig drove this trend
More context (2009): Application Trends

Despite huge amounts of data, many working sets in big data clusters fit in memory.
## 2009: Application Trends

<table>
<thead>
<tr>
<th>Memory (GB)</th>
<th>Facebook (% jobs)</th>
<th>Microsoft (% jobs)</th>
<th>Yahoo! (% jobs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>69</td>
<td>38</td>
<td>66</td>
</tr>
<tr>
<td>16</td>
<td>74</td>
<td>51</td>
<td>81</td>
</tr>
<tr>
<td><strong>32</strong></td>
<td><strong>96</strong></td>
<td><strong>82</strong></td>
<td><strong>97.5</strong></td>
</tr>
<tr>
<td>64</td>
<td>97</td>
<td>98</td>
<td>99.5</td>
</tr>
<tr>
<td>128</td>
<td>98.8</td>
<td>99.4</td>
<td>99.8</td>
</tr>
<tr>
<td>192</td>
<td>99.5</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>256</td>
<td>99.6</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>

*G Ananthanarayanan, A. Ghodsi, S. Shenker, I. Stoica, ”Disk-Locality in Datacenter Computing Considered Irrelevant”, HotOS 2011*
# 2009: Application Trends

<table>
<thead>
<tr>
<th>Memory (GB)</th>
<th>Facebook (% jobs)</th>
<th>Microsoft (% jobs)</th>
<th>Yahoo! (% jobs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>69</td>
<td>38</td>
<td>66</td>
</tr>
<tr>
<td>16</td>
<td>74</td>
<td>51</td>
<td>81</td>
</tr>
<tr>
<td>32</td>
<td>96</td>
<td>82</td>
<td>97.5</td>
</tr>
<tr>
<td>64</td>
<td>97</td>
<td>98</td>
<td>99.5</td>
</tr>
<tr>
<td>128</td>
<td>98.8</td>
<td>99.4</td>
<td>99.8</td>
</tr>
<tr>
<td>192</td>
<td>99.5</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>256</td>
<td>99.6</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>

*G Ananthanarayanan, A. Ghodsi, S. Shenker, I. Stoica, ”Disk-Locality in Datacenter Computing Considered Irrelevant”, HotOS 2011*
Operations on RDDs

Transformations $f(\text{RDD}) \Rightarrow \text{RDD}$
- Lazy (not computed immediately)
- E.g., “map”, “filter”, “groupBy”

Actions:
- Triggers computation
- E.g. “count”, “collect”, “saveAsTextFile”
Working With RDDs

textFile = sc.textFile("SomeFile.txt")
Working With RDDs

```python
textFile = sc.textFile("SomeFile.txt")
linesWithSpark = textFile.filter(lambda line: "Spark" in line)
```
Working With RDDs

```
linesWithSpark = textFile.filter(lambda line: "Spark" in line)
linesWithSpark.count()  # 74
linesWithSpark.first()  # Apache Spark
```
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```python
lines = spark.textFile("hdfs://...")
```
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile("hdfs://...")
```
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```python
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
```
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
```
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```python
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
```
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```python
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
```
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```python
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
```
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```python
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
```
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```python
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t") [2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
```
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```python
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
```
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```python
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
```
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```python
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
messages.filter(lambda s: "php" in s).count()
```
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```python
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()
messages.filter(lambda s: "mysql" in s).count()
messages.filter(lambda s: "php" in s).count()
```
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```python
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
messages.filter(lambda s: "php" in s).count()
```
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```python
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
messages.filter(lambda s: "php" in s).count()
```
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns.

```python
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
messages.filter(lambda s: "php" in s).count()
```

Cache your data ➔ Faster Results

*Full-text search of Wikipedia*

- 60GB on 20 EC2 machines
- 0.5 sec from mem vs. 20s for on-disk
Language Support

**Python**

```python
lines = sc.textFile(...)
lines.filter(lambda s: "ERROR" in s).count()
```

**Scala**

```scala
val lines = sc.textFile(...)
lines.filter(x => x.contains("ERROR")).count()
```

**Java**

```java
JavaRDD<String> lines = sc.textFile(...);
lines.filter(new Function<String, Boolean>() {
    Boolean call(String s) {
        return s.contains("error");
    }
}).count();
```

**Standalone Programs**

Python, Scala, & Java

**Interactive Shells**

Python & Scala

**Performance**

Java & Scala are faster due to static typing

…but Python is often fine
Expressive API

map  reduce
Expressive API

map
filter
groupBy
sort
union
join
leftOuterJoin
rightOuterJoin
reduce
count
fold
reduceByKey
groupByKey
cogroup
cross
zip
sample
take
first
partitionBy
mapWith
pipe
save
...
Fault Recovery: Design Alternatives

Replication:
- Slow: need to write data over network
- Memory inefficient

Backup on persistent storage:
- Persistent storage still (much) slower than memory
- Still need to go over network to protect against machine failures

Spark choice:
- Lineage: track seq. of operations to efficiently reconstruct lost RRD partitions
- Enabled by determinist execution and data immutability
Fault Recovery Example

Two-partition RDD $A=\{A_1, A_2\}$ stored on disk

1) filter and cache $\rightarrow$ RDD $B$

2) join $\rightarrow$ RDD $C$

3) aggregate $\rightarrow$ RDD $D$
Fault Recovery Example

$C_1$ lost due to node failure before “aggregate” finishes
Fault Recovery Example

$C_1$ lost due to node failure before reduce finishes
Reconstruct $C_1$, eventually, on different node
Fault Recovery Results

![Bar chart showing iteration times and failure point]

- Iteration 1: 119 s (Failure happens)
- Iteration 2: 57 s
- Iteration 3: 56 s
- Iteration 4: 58 s
- Iteration 5: 58 s
- Iteration 6: 81 s
- Iteration 7: 57 s
- Iteration 8: 59 s
- Iteration 9: 57 s
- Iteration 10: 59 s
Spark Streaming: Motivation

Process large data streams at second-scale latencies
- Site statistics, intrusion detection, online ML

To build and scale these apps users want
- **Fault-tolerance:** both for crashes and stragglers
- **Exactly one semantics**
- **Integration:** with offline analytical stack
Spark Streaming

Data streams are chopped into batches
- A batch is an RDD holding a few 100s ms worth of data

Each batch is processed in Spark
How does it work?

Data streams are chopped into batches
  • A batch is an RDD holding a few 100s ms worth of data

Each batch is processed in Spark

Results pushed out in batches
Streaming Word Count

```scala
val lines = context.socketTextStream("localhost", 9999)
val words = lines.flatMap(_.split(" "))
val wordCounts = words.map(x => (x, 1)).reduceByKey(_ + _)
wordCounts.print()
ssc.start()
```

- create DStream from data over socket
- split lines into words
- count the words
- print some counts on screen
- start processing the stream
object NetworkWordCount {
    def main(args: Array[String]) {
        val sparkConf = new SparkConf().setAppName("NetworkWordCount")
        val context = new StreamingContext(sparkConf, Seconds(1))

        val lines = context.socketTextStream("localhost", 9999)
        val words = lines.flatMap(_.split(" "))
        val wordCounts = words.map(x => (x, 1)).reduceByKey(_ + _)

        wordCounts.print()
        ssc.start()
        ssc.awaitTermination()
    }
}
public class WordCountTopology {
    public static class SplitSentence extends ShellBolt {
        public static class Splitsentence extends ShellextendsShellBolt implements ShellBolt {
            public static class Splitsentence extends ShellBolt {
                public static class Splitsentence () {
                    super ("python", "splitsentence.py");
                }
            }
        }
    }
}

public class WordCountTopology {
    public static class SplitSentence extends ShellBolt implements ShellBolt {
        public static class Splitsentence extends ShellextendsShellBolt {
            public static class Splitsentence extends ShellBolt {
                public static class Splitsentence () {
                    super ("python", "splitsentence.py");
                }
            }
        }
    }
}

public class WordCountTopology {
    public static class SplitSentence extends ShellBolt implements ShellBolt {
        public static class Splitsentence extends ShellextendsShellBolt {
            public static class Splitsentence extends ShellBolt {
                public static class Splitsentence () {
                    super ("python", "splitsentence.py");
                }
            }
        }
    }
}

public class WordCountTopology {
    public static class SplitSentence extends ShellBolt implements ShellBolt {
        public static class Splitsentence extends ShellextendsShellBolt {
            public static class Splitsentence extends ShellBolt {
                public static class Splitsentence () {
                    super ("python", "splitsentence.py");
                }
            }
        }
    }
}
Spark 2.0

Dataframes/datasets instead of RDDs

- Like tables in SQL
- Far more efficient:
  - Can directly access any field (dramatically reduce I/O and serialization/deserialization)
  - Can use column oriented access

Dataframe APIs (e.g., Python, R, SQL) use same optimizer: Catalyst

New libraries or old libraries revamped to use dataframe APIs

- Spark Streaming → Structured Streaming
- GraphX
General

Unifies *batch, interactive, streaming* workloads

Easy to build sophisticated applications

- Support iterative, graph-parallel algorithms
- Powerful APIs in Scala, Python, Java, R

```
Spark SQL  Spark Streaming  MLlib  GraphX  SparkR
```

Spark Core
Summary

MapReduce and later Spark jump-started Big Data processing

Lesson learned

• Simple design, simple computation model can go a long way
• Scalability, fault-tolerance first, performance next
  - With Dataframes and SparkSQL now Spark implements DB like optimizations which significantly increase performance
## Discussion

<table>
<thead>
<tr>
<th>Environment, Assumptions</th>
<th>OpenMP/Cilk</th>
<th>MPI</th>
<th>MapReduce / Spark</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Single node, multiple core, shared memory</td>
<td>Supercomputers</td>
<td>Commodity clusters</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Sophisticate programmers</td>
<td>Java programmers</td>
</tr>
<tr>
<td></td>
<td></td>
<td>High performance</td>
<td>Programmer productivity</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hard to scale hardware</td>
<td>Easier, faster to scale up cluster</td>
</tr>
<tr>
<td>Computation Model</td>
<td>Fine-grained task parallelism</td>
<td>Message passing</td>
<td>Data flow / BSP</td>
</tr>
<tr>
<td>Strengths</td>
<td>Simplifies parallel programming on multi-cores</td>
<td>Can write very fast asynchronous code</td>
<td>Fault tolerance</td>
</tr>
<tr>
<td>Weaknesses</td>
<td>Still pretty complex, need to be careful about</td>
<td>Fault tolerance</td>
<td>Not as high performance as MPI</td>
</tr>
<tr>
<td></td>
<td>race conditions</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>