COMP 322: Fundamentals of Parallel Programming

Lecture 32: Apache Spark framework for cluster computing

Vivek Sarkar, Eric Allen
Department of Computer Science, Rice University
Contact email: vsarkar@rice.edu

https://wiki.rice.edu/confluence/display/PARPROG/COMP322

Worksheet #31 solution: impact of distribution on parallel completion time (rather than locality)

```
public void sampleKernel(
    int iterations, int numChunks, Distribution dist) {
  for (int iter = 0; iter < iterations; iter++) {
    finish(() -> {
      forseq (0, numChunks - 1, (jj) -> {
        asyncAt(dist.get(jj), () -> {
          doWork(jj);
          // Assume that time to process chunk jj = jj units
       });
      });
    });
  } // for iter
} // sample kernel
```

- Assume an execution with n places, each place with one worker thread
- Will a block or cyclic distribution for dist have a smaller abstract completion time, assuming that all tasks on the same place are serialized with one worker per place?

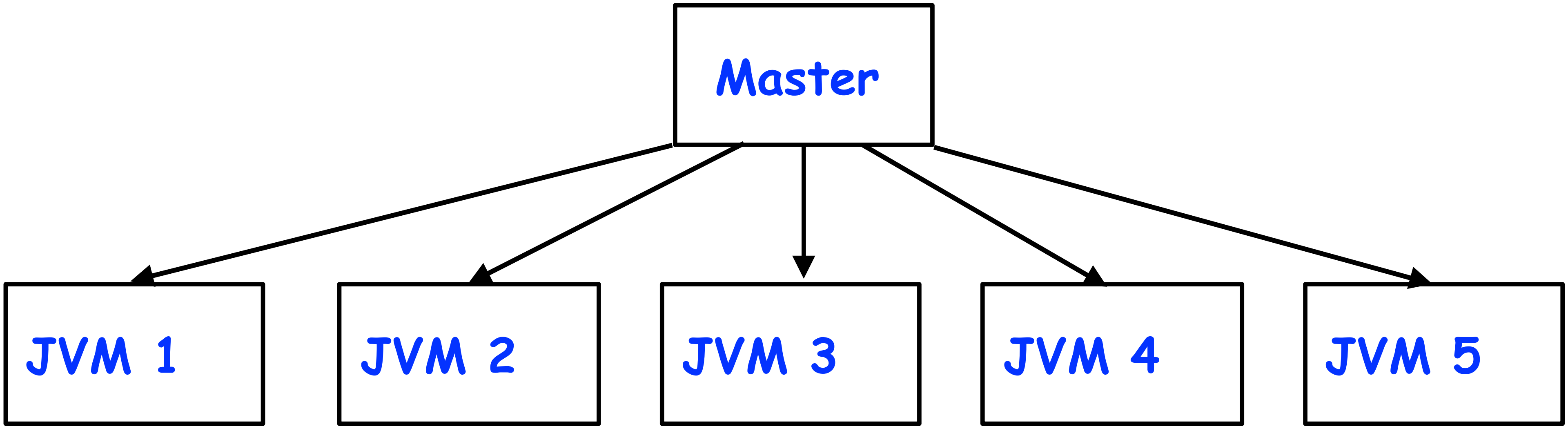
Answer: Cyclic distribution because it leads to better load balance (locality was not a consideration in this problem)

Spark and Iterative Map/Reduce

- After experience with Map/Reduce, users started realizing that a much larger class of algorithms could be expressed as an iterative sequence of map/reduce operations
    - Many machine learning algorithms fall into this category
- Tools started to emerge to enable easy expression of multiple map/reduce operations, along with smart scheduling
- But it is also useful to interactively query large datasets
- Apache Spark: General purpose functional programming over a cluster
    - Caches results of map/reduce operations in memory so they can be used on subsequent iterations
    - Tends to be 10-100 times faster than Hadoop for many applications

Apache Spark

- Distributed computing framework based on the Scala programming language (on the JVM)
- Multiple JVMs (one per machine in a cluster) are coordinated by a master JVM



The Scala Programming Language

- Scala is a programming language that combines object-oriented and functional language features
- Scala comes from "SCAlable LAnguage": Intended to have the feel of a scripting language (read-eval-print loop, type inference) but support for programming in the large (efficient JVM-based implementation, powerful static type system, etc.)
- Many object-oriented design patterns are natively supported (singletons via object definitions, visitors via pattern matching)
- Deep interoperability with Java: Classes can be freely mixed between languages
- Full-fledged functional language: Anonymous functions, higher order functions, efficient immutable data structures, currying

The Scala Programming Language

- Small example Scala program:

```
object Main {
  def main(args: Array[String]) {
    val result = for (i <- 1:10) yield i*i
    println("Squares: " + result.toString)
  }
}
```

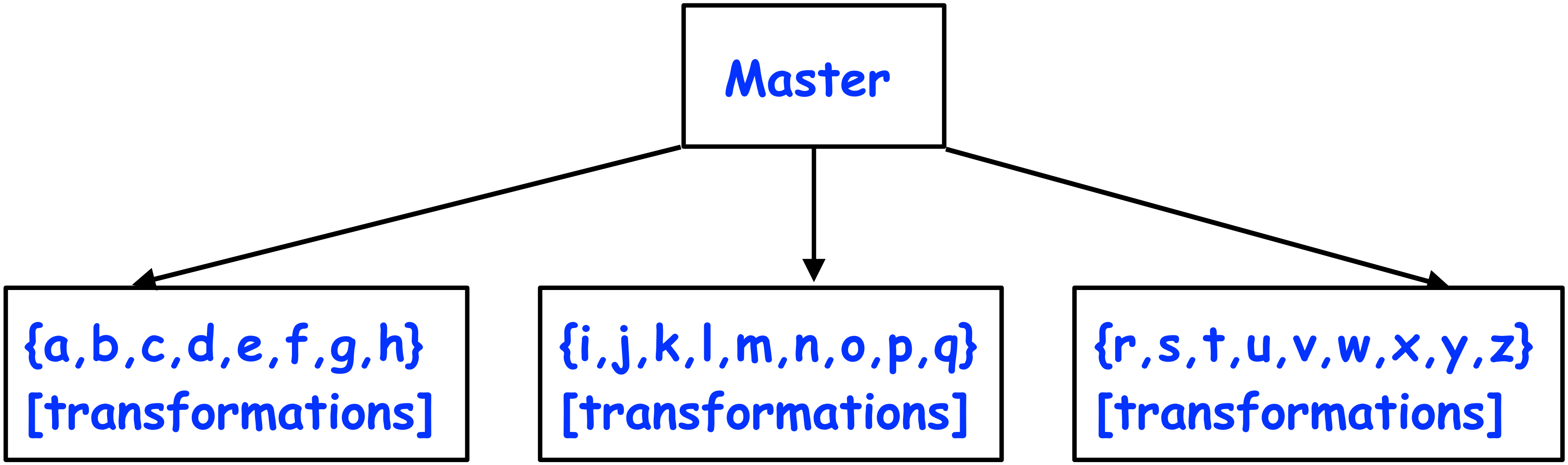
- For more exposure to Scala and functional programming check out Comp 311 this Fall

Spark: Resilient Distributed Datasets

- The key construct in Spark is the Resilient Distributed Dataset (RDD)
- An RDD is an <u>immutable</u> collection, distributed in a reliable way over the machines in a cluster
- The types of the elements in the RDD can be arbitrary elements
- If the elements are pairs, then the RDD acts like a key-value map or table
- Computations on an RDD (including Map/Reduce) can be expressed as functional programming operations

Apache Spark

- **Resilience is achieved without significant data replication:**
  - The transformations used to compute an RDD are necessarily shared across an nodes, enabling efficient recompilation of elements
  - Transformations are not applied until forced (an advantage of immutability)



Master

{a,b,c,d,e,f,g,h}
[transformations]

{i,j,k,l,m,n,o,p,q}
[transformations]

{r,s,t,u,v,w,x,y,z}
[transformations]

Advantages of Immutability

- The distributed nature of RDDs is not evident in the programming model
- RDD elements can be replicated for fault tolerance
- Purely functional operations can be easily defined on RDDs
- The runtime has great flexibility in scheduling operations on RDDs

Wordcount in Apache Spark

```
val file = spark.textFile(“hdfs://...")

val counts = file.flatMap(line => line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey((x,y) => x + y)

counts.saveAsTextFile("hdfs://...")
```

Wordcount in Apache Spark

```
val file = spark.textFile(“hdfs://...")

val counts = file.flatMap(line => line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)

counts.saveAsTextFile("hdfs://...")
```

Wordcount in Apache Spark

```
val file = spark.textFile(“hdfs://...")

val counts = file.flatMap(line => line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)

counts.saveAsTextFile("hdfs://...")
```

x.flatMap(*f*) = x.map(*f*).flatten()

Wordcount in Apache Spark

```
(“this is a line”,
 “this is another line”,
 “this is yet another line”)
.map(line => line.split())
.flatten()
```

Wordcount in Apache Spark

```
((“this”, “is”, “a”, “line”),
 (“this”, “is”, “another”, “line”),
 (“this”, “is”, “yet”, “another”, “line”))
.flatten()
```

Wordcount in Apache Spark

```
(“this”, “is”, “a”, “line”, “this”, “is”,
 “another”, “line”, “this”, “is”, “yet”,
 “another”, “line”)
```

Wordcount in Apache Spark

```
val file = spark.textFile(“hdfs://...")

val counts = file.flatMap(line => line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)

counts.saveAsTextFile("hdfs://...")
```

Wordcount in Apache Spark

```
(“this”, “is”, “a”, “line”, “this”, “is”,
 “another”, “line”, “this”, “is”, “yet”,
 “another”, “line”)
.map(word => (word, 1))
```

Wordcount in Apache Spark

```
((“this”,1), (“is”,1), (“a”,1), (“line”,1),
 (“this”,1), (“is”,1), (“another”,1),
 (“line”,1), (“this”,1), (“is”,1),
 (“yet”,1),(“another”,1), (“line”,1))
```

Wordcount in Apache Spark

```
val file = spark.textFile(“hdfs://...")

val counts = file.flatMap(line => line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)

counts.saveAsTextFile("hdfs://...")
```

```
x.reduceByKey(f) = x.groupByKey()
                    .map(xs =>
                           xs.reduce(f))
```

Wordcount in Apache Spark

```
((“this”,1), (“is”,1), (“a”,1), (“line”,1),
 (“this”,1), (“is”,1), (“another”,1),
 (“line”,1), (“this”,1), (“is”,1),
 (“yet”,1),(“another”,1), (“line”,1))
.groupByKey().map(xs =>
                     xs.reduce
                         (a,b => a+b)
```

Wordcount in Apache Spark

```
((“this”, (1,1,1),
 (“is”, (1,1,1),
 (“a”, (1)),
 (“line”, (1,1,1)),
 (“another”, (1,1)),
 (“yet”, (1))).map(xs => …)
```

Wordcount in Apache Spark

```
((“this”, (1,1,1).reduce(a,b => a+b),
 (“is”, (1,1,1).reduce(a,b => a+b),
 (“a”, (1)).reduce(a,b => a+b),
 (“line”, (1,1,1)).reduce(a,b => a+b),
 (“another”, (1,1)).reduce(a,b => a+b),
 (“yet”, (1)).reduce(a,b => a+b))
```

Wordcount in Apache Spark

```
((“this”, 3), (“is”, 3), (“a”, 1),
 (“line”, 3), (“another”, 2), (“yet”, 1))
```

Lazy Evaluation of RDDs

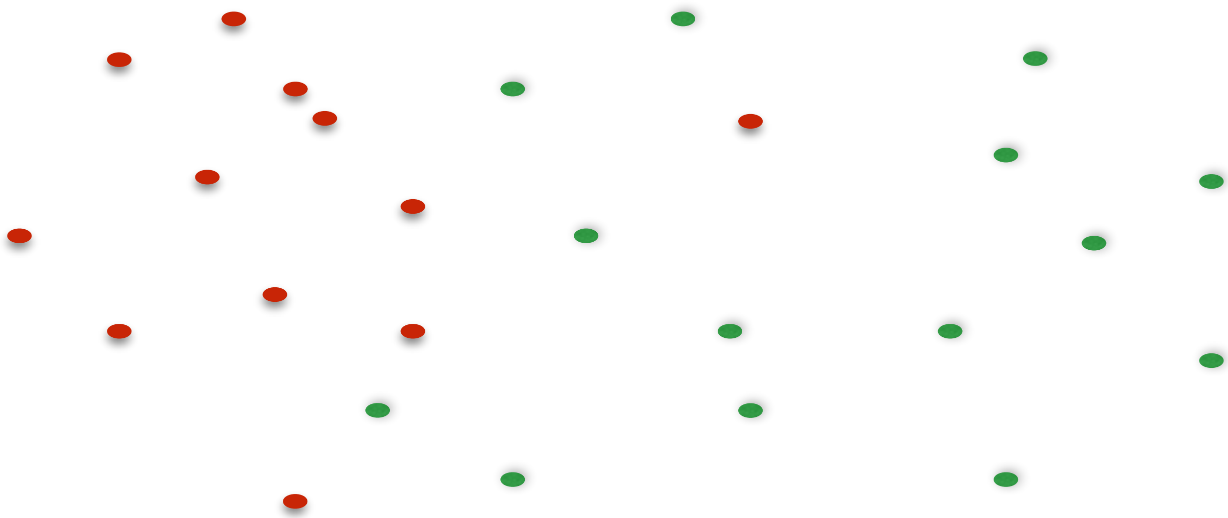
- Map operations (transformations) on RDDs are applied "lazily":
  - The sequence of operations are built up on elements as a closure
  - The closure is not applied until forced by a reduce operation (actions)
- Many other operations are available on RDDs:
  - map, reduce, sample, groupByKey, reduceByKey, join, …
- Because RDDs are immutable, all the operations from purely functional programming can be applied and parallelized in a straightforward way

Iterative Map/Reduce Example: Logistic Regression

- Given a collection of examples with various attributes and a label, we wish to predict the labels for new examples:
- <height, weight, age, systolic bp, diastolic bp>: medicine?
- <170 cm, 72 kg, 52, 120, 80>: YES
- <150 cm, 60 kg, 34 years, 130, 70> : NO
- …

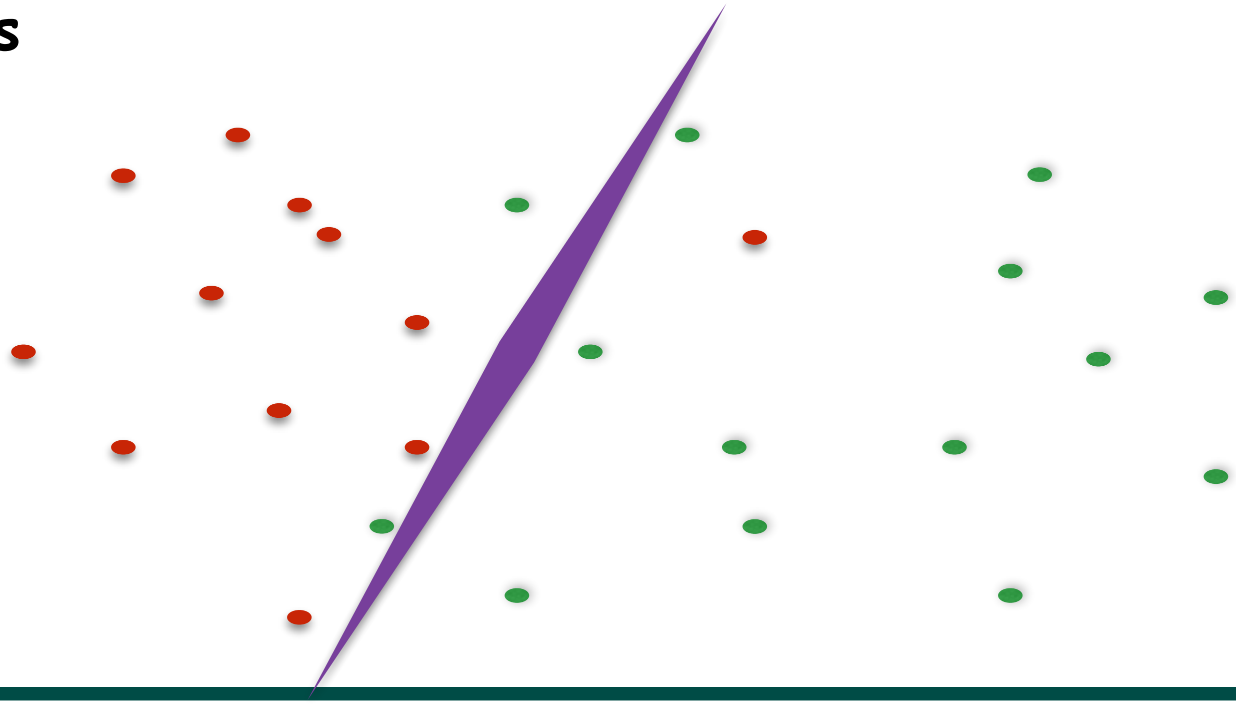
Iterative Map/Reduce Example: Logistic Regression

- We can view the examples as vectors in a high-dimensional vector space
- The problem of labeling yes/no can be solved by finding the best hyperplane that divides the given examples according to their labels
- This new hyperplane can be used to predict labels for new examples



Iterative Map/Reduce Example: Logistic Regression

- We can view the examples as vectors in a high-dimensional vector space
- The problem of labeling yes/no can be solved by finding the best hyperplane that divides the given examples according to their labels
- This new hyperplane can be used to predict labels for new examples



Iterative Map/Reduce Example: Logistic Regression

```
val points = spark.textFile(…).map(parsePoint).cache()

var w = Vector.random(D) // current separating plane

for (i <- 1 to ITERATIONS) {
  val gradient = points.map(p =>
    (1 / (1 + exp(-p.y*(w dot p.x))) - 1) * p.y * p.x
  ).reduce(_ + _)

  w -= gradient
}

println("Final separating plane: " + w)
```

Example presented in:

Zaharia, Matei, et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing." *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012.

Worksheet #32: impact of distribution on parallel completion time (rather than locality)

```
val points = spark.textFile(…).map(parsePoint).cache()

var w = Vector.random(D) // current separating plane

for (i <- 1 to ITERATIONS) {
  val gradient = points.map(doWork(1)).reduce(_ + _)

  w -= gradient
}

println("Final separating plane: " + w)
```

Consider the above simplified regression program.
Let each doWork operation cost 1 unit of work.
What is the total work? What is the CPL?