COMP 322: Fundamentals of Parallel Programming

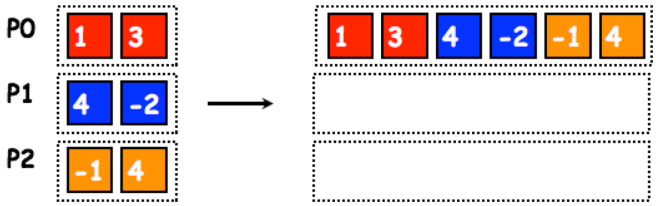
Lecture 37: Distributed Computing, Apache Spark

Vivek Sarkar, Shams Imam
Department of Computer Science, Rice University
vsarkar@rice.edu, shams@rice.edu

comp322.rice.edu

Worksheet #34: MPI Gather

Indicate what value should be provided instead of ??? in line 6 to minimize space, and how it should depend on myrank.




```
1.    MPI.Init(args) ;
2.    int myrank = MPI.COMM_WORLD.Rank() ;
3.    int numProcs = MPI.COMM_WORLD.Size() ;
4.    int size = ...;
5.    int[] sendbuf = new int[size];
6.    int[] recvbuf = new int[???];
7.    . . . // Each process initializes sendbuf
8.    MPI.COMM_WORLD.Gather(sendbuf, 0, size, MPI.INT,
9.                          recvbuf, 0, size, MPI.INT,
10.                         0 /*root*/);
11.   . . .
12.   MPI.Finalize();
```

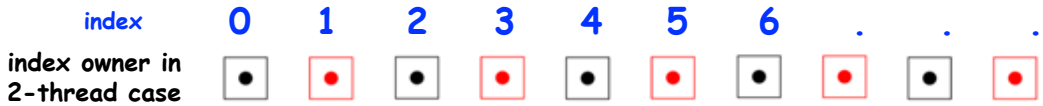
Solution: `myrank == 0 ? (size * numProcs) : 0`

Worksheet #36: UPC data distributions

In the following example from Lecture 36 slide 20, assume that each UPC array is distributed by default across threads with a *cyclic* distribution. In the space below, identify an iteration of the upc_forall construct for which all array accesses are local, and an iteration for which all array accesses are non-local (remote).

Assume 2 <= THREADS < 100. Explain your answer in each case.

```
1. shared int a[100],b[100], c[100];
2. int i;
3. upc_forall (i=0; i<100; i++; (i*THREADS)/100)
4.     a[i] = b[i] * c[i];
```



Solution:

- Iteration 0 has affinity with thread 0, and accesses a[0], b[0], c[0], all of which are located locally at thread 0
- Iteration 1 has affinity with thread 0, and accesses a[1], b[1], c[1], all of which are located remotely at thread 1

MapReduce Pattern (Recap from Lecture 9)

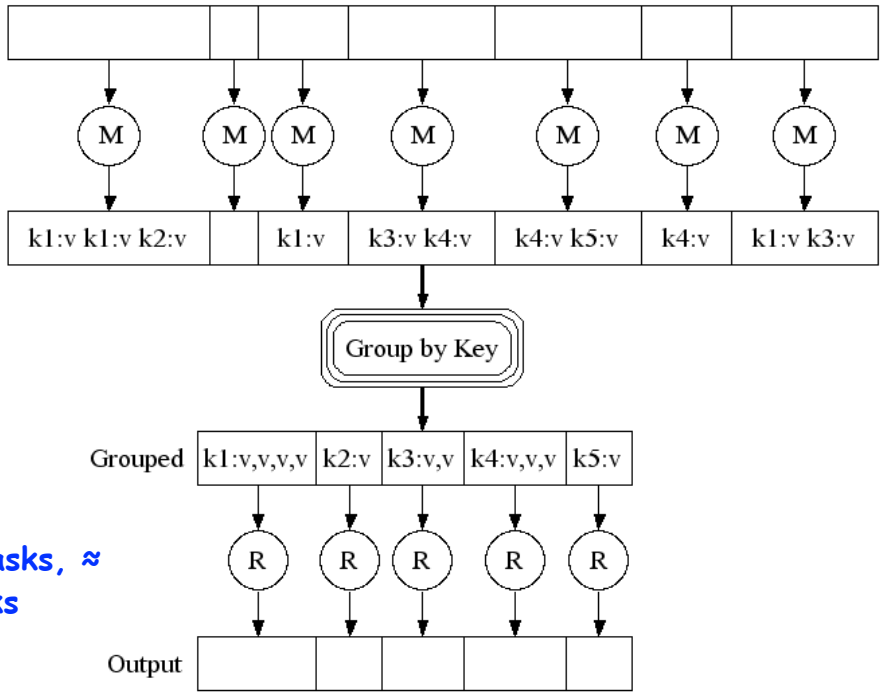
- Apply `Map` function `f` to user supplied record of key-value pairs
- Compute set of intermediate key/value pairs
- Apply `Reduce` operation `g` to all values that share same key to combine derived data properly
  - —Often produces smaller set of values
- User supplies Map and Reduce operations in functional model so that the system can parallelize them, and also re-execute them for fault tolerance

MapReduce Execution

Fine granularity tasks: many more map tasks than machines

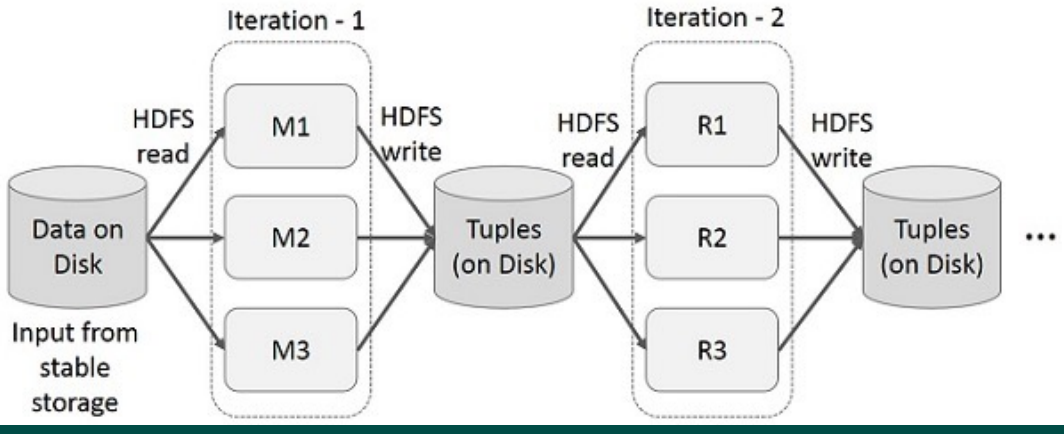
Bucket sort to get same keys together

2000 servers => ≈ 200,000 Map Tasks, ≈ 5,000 Reduce tasks



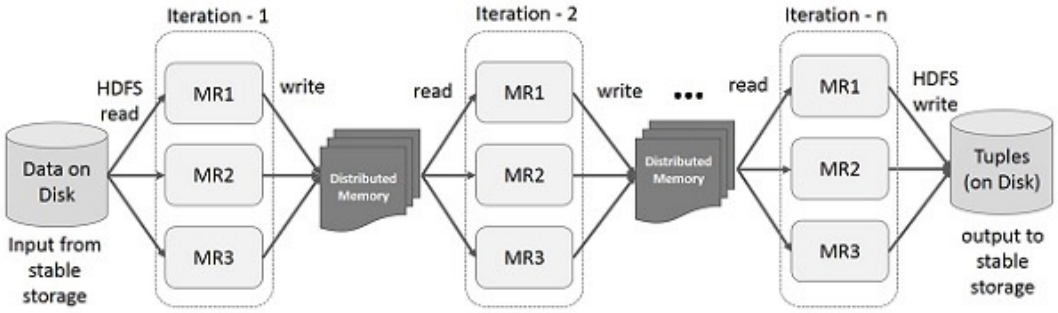
Map/Reduce and Iterative Algorithms

- Apache Hadoop now dominates use of the Map/Reduce framework
- Algorithms could be expressed as an iterative sequence of map/reduce operations
  - —Many machine learning algorithms,e.g. gradient descent, fall into this category
- This iterative pattern is also useful to interactively query large datasets



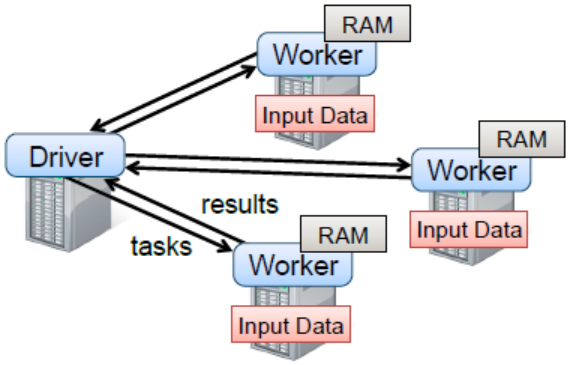
Spark and Iterative Map/Reduce

- **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**



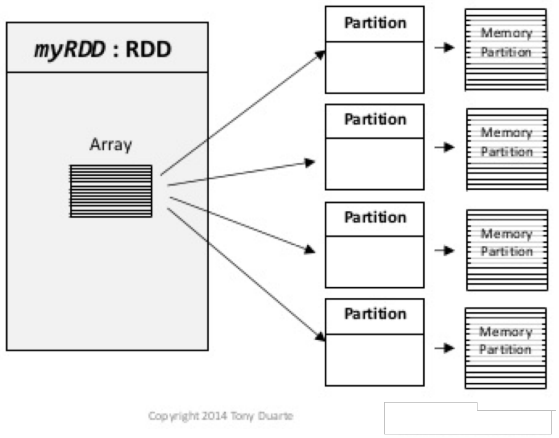
Apache Spark

- **Spark is a data parallel processing framework, which means it will execute tasks as close to where the data lives as possible (i.e. minimize data transfer).**
- **Spark follows a paradigm of keeping as much data in-memory and spilling excess to disk rather than pulling data from disk when needed.**
- **Spark revolves around the concept of a resilient distributed datasets (RDD).**
- **Spark decomposes your program into tasks and handles dispatching and scheduling of these tasks on worker nodes in your cluster.**



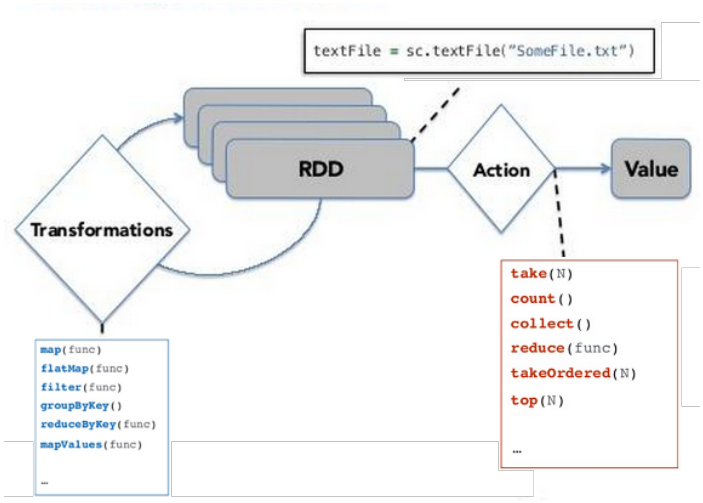
Resilient Distributed Datasets

- The key construct in Spark is the Resilient Distributed Dataset (RDD)
  - RDDs can be though of as a collection of key-value pairs
- An RDD is a giant immutable collection, distributed in a redundant way over all 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 table
- Computations on an RDD (including Map/Reduce) can be expressed as functional programming operations



Working with RDDs

- There are two kinds of operations one can perform over an RDD.
- Transformations: Operations like map, filter, join etc. that just return another RDD. They are *lazy* operations.
- Actions: These are operations that *actually produce* results like count, collect, save etc. These operations don't return an RDD.
- Similar to *intermediate* and *terminal* operations in Java 8 Streams.



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**
- **Because RDDs are immutable, all the operations from purely functional programming can be applied and parallelized in a straightforward way.**
- **The runtime has great flexibility in scheduling operations on RDDs and executing them in parallel on partitions.**
- **Partitions of RDDs can be recomputed from their lineage.**

Word Count in Apache Spark

```
JavaRDD<String> file = context.textFile(inputFile);

JavaPairRDD<String, Integer> counter =
  file.flatMap(s -> Arrays.asList(s.split(" ")))
      .mapToPair(s -> new Tuple2<>(s, 1))
      .reduceByKey((a, b) -> a + b);

counter.collect().forEach(System.out::println);
```


Word Count in Apache Spark

```
JavaRDD<String> file = context.textFile(inputFile);

JavaPairRDD<String, Integer> counter =
  file.flatMap(s -> Arrays.asList(s.split(" ")))
      .mapToPair(s -> new Tuple2<>(s, 1))
      .reduceByKey((a, b) -> a + b);

counter.collect().forEach(System.out::println);
```

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


Word Count in Apache Spark

```
["this is a line",
 "this is another line",
 "this is yet another line"]
.map(s -> Arrays.asList(s.split(" ")))
.flatten()
-->
[["this", "is", "a", "line"],
 ["this", "is", "another", "line"],
 ["this", "is", "yet", "another", "line"]]
.flatten()
-->
["this", "is", "a", "line", "this", "is",
 "another", "line", "this", "is", "yet",
 "another", "line"]
```


Word Count in Apache Spark

```
JavaRDD<String> file = context.textFile(inputFile);

JavaPairRDD<String, Integer> counter =
  file.flatMap(s -> Arrays.asList(s.split(" ")))
      .mapToPair(s -> new Tuple2<>(s, 1))
      .reduceByKey((a, b) -> a + b);

counter.collect().forEach(System.out::println);
```


Word Count in Apache Spark

```
["this", "is", "a", "line", "this", "is",
 "another", "line", "this", "is", "yet",
 "another", "line"]
.map(s -> new Tuple2<>(s, 1))

—->

[["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]]
```


Word Count in Apache Spark

```
JavaRDD<String> file = context.textFile(inputFile);

JavaPairRDD<String, Integer> counter =
  file.flatMap(s -> Arrays.asList(s.split(" ")))
      .mapToPair(s -> new Tuple2<>(s, 1))
      .reduceByKey((a, b) -> a + b);

counter.collect().forEach(System.out::println);
```

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


Word Count 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))
—->
[["this", [1,1,1]],
 ["is", [1,1,1]],
 ["a", [1]],
 ["line", [1,1,1]],
 ["another", [1,1]],
 ["yet", [1]]].map(xs -> xs.reduce((a,b) -> a + b))
```


Word Count in Apache Spark

```
[["this", [1,1,1]],
 ["is", [1,1,1]],
 ["a", [1]],
 ["line", [1,1,1]],
 ["another", [1,1]],
 ["yet", [1]]].map(xs -> xs.reduce(
                                (a,b) -> a + b)
—->
[["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))]
```


Word Count 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))]

—->

[["this", 3], ["is", 3], ["a", 1],
 ["line", 3], ["another", 2], ["yet", 1]]
```