CME 323: Distributed Algorithms and Optimization, Spring 2015
`http://stanford.edu/~rezab/dao`.
Instructor: Reza Zadeh, Matroid and Stanford.

Lecture 15, 5/16/2016. Scribed by Jayanth Ramesh, Hao Wu.

15.1 Node Iterator Algorithm

The work done by the mappers in the node iterator algorithm can be captured in the following steps

Algorithm 1 MAP($G(V, E)$

```
1: for $v \in V$ do
2:     for $(u, w) \in \Gamma(v) \times \Gamma(v)$ do
3:         Emit $((u, w)$, ”to be checked”)
4:     end for
5: end for
```

The total time taken by the algorithm can be seen as the time taken by the following three stages

- Last Mapper
- Shuffle size
- Last Reducer

Naively, the last mapper would take time $\mathcal{O}(n^2)$. But by using a clever trick, this can be brought to $\mathcal{O}(m^{\frac{3}{2}})$.

The Trick
Instead of looking at all pairs of edges in the neighborhood of a particular node , we only look at those neighbors of a node v whose degree is higher than that of v. Let $x \prec y$ if $deg(x) < deg(y)$, where we break ties arbitrarily but consistently (say, based on the node id). This new neighborhood list is denoted by $\Gamma^*(v)$

Analysis
Introduce $t > 0$ All nodes with $deg(v) \geq t$ are called high degree nodes and nodes with $deg(v) < t$ are called low degree nodes. **Observation:** One property here is that the modified neighborhood list of a node with high degree will consist of only high degree nodes. The number of computations by the mapper is

$$\sum_{v \in V} (deg^*(v))^2$$

where $deg^*(v)$ denotes the size of $\Gamma^*(v)$. In this particular problem , the last mapper can't take any longer than the shuffle size. The shuffle size is upper bounded by

$$\sum_{\substack{v\in V\\ deg(v)<t}} (deg^*(v))^2 + \sum_{\substack{v\in V\\ deg(v)\geq t}} (deg^*(v))^2$$

Let

$$A = \sum_{\substack{v\in V\\ deg(v)<t}} (deg^*(v))^2$$

and

$$B = \sum_{\substack{v\in V\\ deg(v)\geq t}} (deg^*(v))^2$$

There are at most $\frac{2m}{t}$ high degree nodes. Hence, the number of triangles formed by higher degree nodes is bounded by

$$B \leq \left(\frac{2m}{t}\right)^3$$

Moreover,

$$A = \sum_{\substack{v\in V\\ deg(v)<t}} (deg^*(v))^2$$

$$A \leq \sum_{\substack{v\in V\\ deg(v)<t}} t(deg^*(v))$$

$$A \leq t(2m)$$

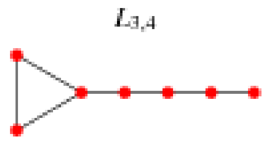
Therefore,

$$A + B \leq 2mt + \left(\frac{2m}{t}\right)^3$$

Minimizing the above expression with respect to t, we have at $t = \sqrt{m}$

$$A + B \leq \mathcal{O}\left(m^{\frac{3}{2}}\right)$$

This clever method will have to do the same order of work as the naive method if the graph were dense, but for a sparse graph, this method has much better time complexity. The above analysis is also tight if an algorithm has to list all the triangles in a graph. Consider a lollipop graph as shown in the figure below, where the complete graph part has $\sqrt{n}$ nodes and the path graph part has $n - \sqrt{n}$ nodes. The number of triangles is given by $T = \binom{\sqrt{n}}{3} = \mathcal{O}(m^{1.5})$ triangles, and hence this bound is tight for an algorithm that has to list all the triangles in a graph.



Credit: *Wolfram Alpha : http://mathworld.wolfram.com/LollipopGraph.html*

Figure 1: A lollipop Graph [1]

15.2 Combiners

When the reduce function is an associative binary operator, a *Combiner* can be used in between the *Map* and the *Reduce* to reduce the volume of data transfer between them. A *Combiner* performs a local *Reduce* to summarize the map output records with the same key before the emission occurs. The following MapReduce task diagram shows the *Combiner* phase (Fig. 2).

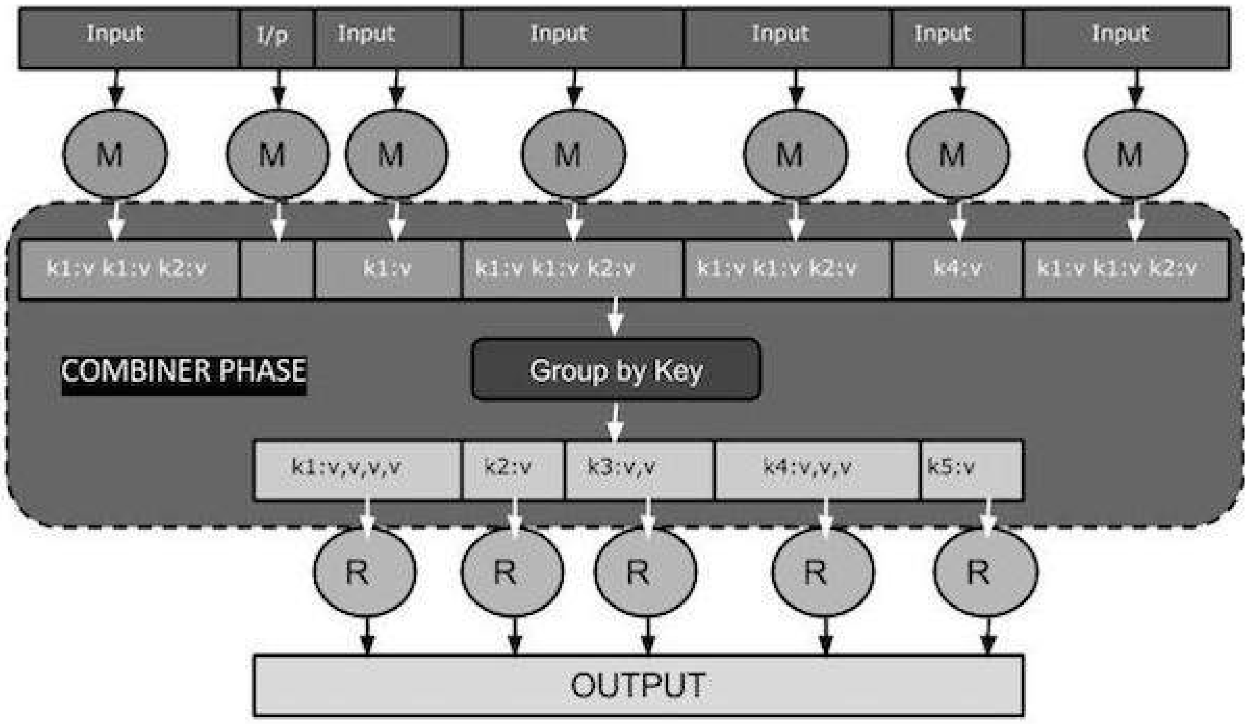


Figure 2: A MapReduce task diagram showing the combiner phase. [2]

15.3 Broadcasting

For optimization problems on the distributed system, it is often needed to propagate current guess for optimization variables to all machines. Such one-to-all communication operation is called *broadcasting*. The exact wrong way to do it is with "one machines feeds all", which requires p rounds of communication, while using the Bit-torrent broadcasting instead needs only $\log(p)$ rounds (Fig. 4).

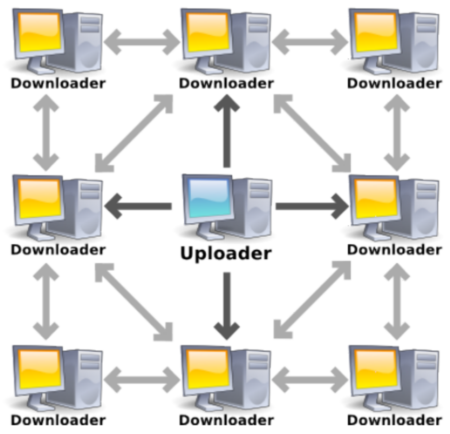


Figure 3: The schematic of Bit-torrent broadcasting.

15.3.1 Broadcast Rules

Broadcast variables allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks. They can be used, for example, to give every node a copy of a large input dataset in an efficient manner.

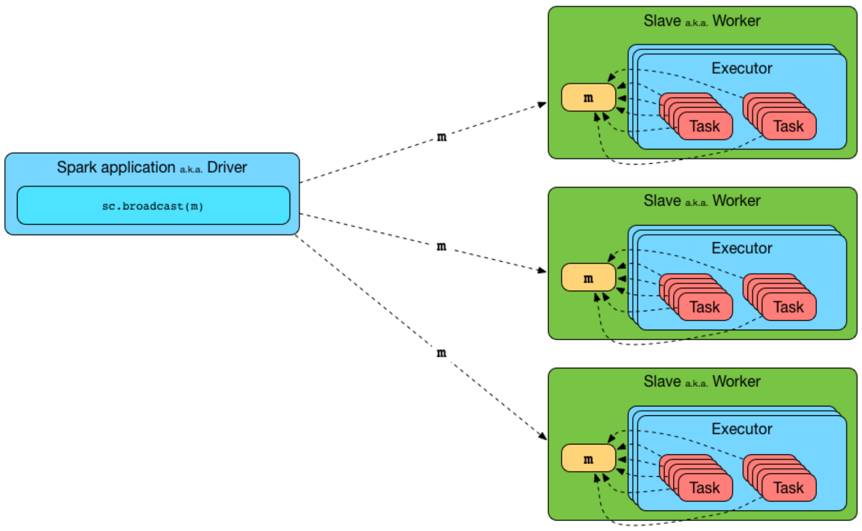
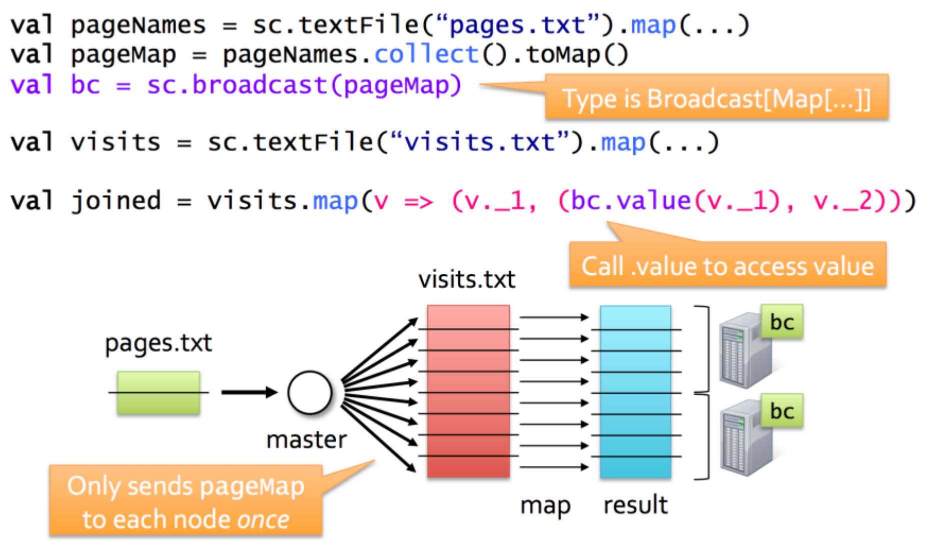


Figure 4: Broadcasting a value to executors. [3]

- Create with `SparkContext.broadcast(initialVal)`
- Access with `.value` inside tasks (first task on each node to use it fetches the value)
- Cannot be modified after creation

15.3.2 Example: Replicated Join



```
val pageNames = sc.textFile("pages.txt").map(...)
val pageMap = pageNames.collect().toMap()
val bc = sc.broadcast(pageMap)

val visits = sc.textFile("visits.txt").map(...)

val joined = visits.map(v => (v._1, (bc.value(v._1), v._2)))
```

Type is Broadcast[Map[...]]
Call .value to access value
visits.txt
pages.txt
master
bc
bc
map
result
Only sends pageMap to each node *once*

Figure 5: Replicated join.

15.4 Spark for Python (PySpark)

The Spark Python API (PySpark) exposes the Spark programming model to Python. Spark core is written in Scala. PySpark calls existing scheduler, cache and networking layer (2K-line wrapper) to automatically ship Python functions to workers, along with any objects that they reference. Instances of classes will be serialized and shipped to workers by PySpark. Therefore, no changes to Python is required.

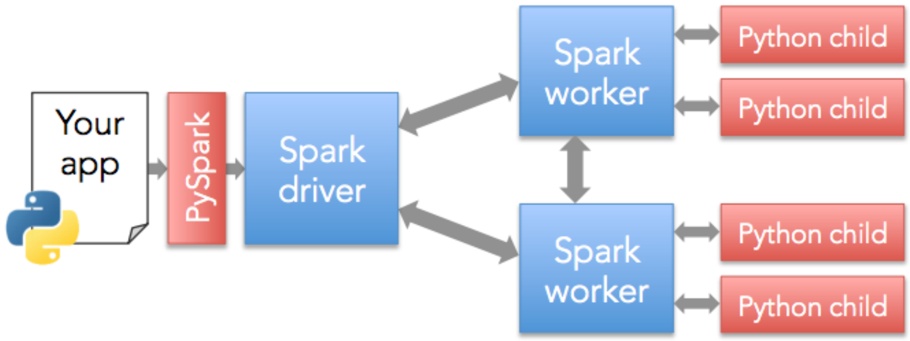


Figure 6: Schematic of pySpark.

References

[1] Lollipop Graph. Retrieved from http://mathworld.wolfram.com/LollipopGraph.html

[2] MapReduce-Combiners. Retrieved from http://www.tutorialspoint.com/map_reduce/map_reduce_combiners.htm.

[3] Broadcast Variables. Retrieved from https://jaceklaskowski.gitbooks.io/mastering-apache-spark/content/spark-broadcast.html.