Complexity Measures for Map-Reduce, and Comparison to Parallel Computing [*]

Ashish Goel
Stanford University and Twitter

Kamesh Munagala
Duke University and Twitter

November 11, 2012

The programming paradigm Map-Reduce [3] and its main open-source implementation, Hadoop [1], have had an enormous impact on large scale data processing. Our goal in this expository writeup is two-fold: first, we want to present some complexity measures that allow us to talk about Map-Reduce algorithms formally, and second, we want to point out why this model is actually different from other models of parallel programming, most notably the PRAM (Parallel Random Access Memory) model. We are looking for complexity measures that are detailed enough to make fine-grained distinction between different algorithms, but which also abstract away many of the implementation details.

1 An overview of Map-Reduce

Map-Reduce is commonly used to refer to both a programming model for *Bulk Synchronous Parallel Processing* [7], as well as a computational infrastructure for implementing this programming model. From the infrastructure point of view, a Map-Reduce job has three phases:

While many good descriptions of Map-Reduce exist [3, 5], we still would like to present a description since one of the phases (shuffle) is typically given less attention, and this phase is going to be crucial in our complexity measures and in the distinction that we draw with PRAM.

Map: In this phase, a User Defined Function (UDF), also called *Map*, is executed on each record in a given file. The file is typically striped across many computers, and many processes (called Mappers) work on the file in parallel. The output of each call to Map is a list of $\langle \text{KEY}, \text{VALUE} \rangle$ pairs.

Shuffle: This is a phase that is hidden from the programmer. All the $\langle \text{KEY}, \text{VALUE} \rangle$ pairs are sent to another group of computers, such that all $\langle \text{KEY}, \text{VALUE} \rangle$ pairs with the same KEY go to the same computer, chosen uniformly at random from this group, and independently of all other keys. At each destination computer, $\langle \text{KEY}, \text{VALUE} \rangle$ pairs with the same KEY are aggregated together. So if $\langle x, y_1 \rangle, \langle x, y_2 \rangle, \ldots, \langle x, y_K \rangle$ are all the key-value pairs produced by the Mappers with the same key x, at the destination computer for key x, these get aggregated into a large $\langle \text{KEY}, \text{VALUE} \rangle$ pair $\langle x, \{y_1, y_2, \ldots, y_K\} \rangle$; observe that there is no ordering guarantee. The aggregated $\langle \text{KEY}, \text{VALUE} \rangle$ pair is typically called a *Reduce Record*, and its key is referred to as the *Reduce Key*.

[*]One Disclaimer: This is an expository article, as opposed to new research. We are not claiming that the measures defined here have never been used before, nor are we claiming that our observations regarding the difference between PRAM and Map-Reduce are novel. Our motivation for writing this is two-fold: (a) We have often felt the need to establish a common vocabulary while discussing the performance of Map-Reduce algorithms with engineers and fellow researchers, and (b) Whenever we talk to more theoretical students, the question of whether Map-Reduce is just parallel programming in another guise always comes up.

Reduce: In this phase, a UDF, also called *Reduce*, is applied to each Reduce Record, often by many parallel processes. Each process is called a *Reducer*. For each invocation of Reduce, one or more records may get written into a local output file.

The reduce phase starts after all the Mappers have finished, and hence, this model is an example of *Bulk Synchronous Processing* (BSP). The shuffle phase is typically implemented by writing all the data that comes to a destination computer to disk. The task of separating out the data into different Reduce Records on each destination computer is also done off of disk. We are going to assume that the total amount of work done in the shuffle phase is proportional only to the size of the data being shuffled, both overall as well as for any one destination computer.

2 Complexity Measures

A good characterization of the class of problems for which the Mpa-Reduce computation model can give a performance advantage over a single machine already exists [5]. However, our goal here is to provide complexity measures that are sufficient to make a fine-grained distinction between the performance of different Map-Reduce algorithms. There are many different operations that happen in Map-Reduce, and an exhaustive list of complexity measures such as the one in [4] does not lead to easy algorithmic analysis. We will focus on a smaller set of measures that we believe capture essential performance bottlenecks. In particular, we keep track of the aggregate work done by the entire system, and the work done at the finest granularity (i.e. Mappers and Reducers) separately. Our measures are:

Key Complexity: This itself consists of three parts:

1. The maximum size of a $\langle$KEY, VALUE$\rangle$ pair input to or output by a Mapper/Reducer,
2. The maximum running time for a Mapper/Reducer for a $\langle$KEY, VALUE$\rangle$ pair.
3. The maximum memory used by a Mapper/Reducer to process a $\langle$KEY, VALUE$\rangle$ pair, and

Sequential Complexity: This time, we sum over all Mappers and Reducers as opposed to looking at the worst.

1. The size of all $\langle$KEY, VALUE$\rangle$ pairs input and output by the Mappers and the Reducers,
2. The total running time for all Mappers and Reducers.

Notice that we omit the total memory from our sequential complexity measure, since that depends on the number of Reducers operating at any given time and is a property of the Map-Reduce deployment as opposed to the algorithm.

For some problems, the key complexity can depend on whether we assume streaming Reducers (as in Hadoop streams [2] which read a Reduce Record one value at a time serially from disk or batched Reducers which take the entire Reduce Record as input and store it in memory.

2.1 Two Illustrative Examples and Discussion

We will discuss two simple and oft-used examples, Word Count and a single PageRank iteration, assuming the trivial Map-Reduce algorithms in each case [3]:

Word Count: Assume N documents, M words, total document size S, and word frequencies $f_1, f_2, \ldots, f_M$. We get the following complexity (assuming batched Reducers):

- Key complexity: The size, time, and memory are all $O(f_{MAX})$ where $f_{MAX} = \max_i f_i$.
- Sequential complexity: The total size and running time are both $O(S)$.

With streaming Reducers, the key complexity becomes $O(f_{MAX})$ (size and time), and $O(1)$ (memory), whereas the sequential complexity remains the same.

PageRank: Given a directed graph $G = (V, E)$ with M edges, N nodes, and maximum in- or out-degree d_{MAX}, each iteration of PageRank (assuming each edge is already annotated with the out-degree of its source node) for batched Reducers is:

- Key complexity: The size, time, and memory are all $O(d_{MAX})$.
- Sequential complexity: The total size and running time are both $O(M)$.

With streaming Reducers, the key complexity becomes $O(d_{MAX})$ (size and time), and $O(1)$ (memory), whereas the sequential complexity remains the same.

In each of the two cases, the complexity measures are simple, and capture natural properties of the algorithms while avoiding implementation and deployment details. In order to be broadly useful, any complexity measure must capture essential aspects of the problem. We describe several such aspects:

1. Our key complexity measures capture the performance of an idealized Map-Reduce system with infinitely many Mappers and Reducers and no coordination overheads. A small key complexity guarantees that a Map-Reduce algorithm will not suffer from "the curse of the last Reducer" [6], a phenomenon where the average work done by all Reducer may be small, but due to variation in the size of Reduce Records, the total wall clock time may be extremely large, or even worse, some Reducers may run out of memory.

2. The sequential complexity measures capture the total system resources consumed. In other words, this would be the amount of effort spent if the entire Map-Reduce installation had a single Mapper and a single Reducer. If the sequential complexity of a Map-Reduce algorithm is small (eg. if it matches the best known PRAM or message passing algorithm, or even better, the best known sequential algorithm for a problem), and the sequential complexity is small as well, then we can immediately conclude that we have an optimum or near-optimum Map-Reduce algorithm. Note that the total size input/output by all Mappers/Reducers captures the total filesystem I/O done by the algorithm, and is often of the order of the shuffle size.

3. Our complexity measures depend only on the algorithm, *not on details of the Map-Reduce installation* such as the number of machines, the number of Mappers/Reducers etc., which is a desirable property for the analysis of algorithms.

In our experience at Twitter, the above measures have proved to be a valuable guide in the design of efficient Map-Reduce algorithms; while subjective, this is arguably the ultimate test of any set of complexity measures.

3 PRAM vs Map-Reduce: Exploiting the Power of the Shuffle Phase

Let us consider the simple PageRank example in the PRAM model, where the input edges reside on shared disk (to make it similar to Map-Reduce and avoid penalizing the PRAM model for storing the M edges). If we have K machines, then the total I/O and the total running time are both $O(M)$, which are matched by Map-Reduce. However, the total memory needed by all the PRAM machines is $O(N)$ where N is the number of nodes, and the memory needed by each PRAM machine is $O(N/K)$. In Map-Reduce with K Reducers, by contrast, the total memory needed by all Reducers (assuming streaming Reducers) is $O(K)$ and the memory needed by each Reducer is $O(1)$. Similar differences exist in the even simpler Word Count example.

This seems surprising, and on first glance, might appear to be a flaw in our modeling. However, we believe this gets exactly to one of the reasons why Map-Reduce is so successful as a computational paradigm (beyond the obvious ease-of-use reasons). In Map-Reduce, the shuffle phase aggregates all the $\langle \text{KEY}, \text{VALUE} \rangle$ pairs into Reduce Records, and the cost for this step is not being charged to the algorithm by our complexity measures. This accurately captures the practical design of Map-Reduce platforms: the shuffle phase first writes everything onto disk at each destination machine, and then aggregates the received $\langle \text{KEY}, \text{VALUE} \rangle$ pairs into Reduce Records. The writing on disk is something that needs to happen anyway because of the BSP model, and dominates the cost of the aggregation phase. Hence, by using the disk as temporary memory, Map-Reduce isolates the cost of aggregation from system performance[1].

The difference in memory usage can be substantial (eg. for small K), and hence, designing efficient Map-Reduce algorithms is an interesting research question in its own right, *distinct from the design of efficient PRAM algorithms*. Many of the algorithms that we are currently working on (eg. [8]) exploit the fact that we get the aggregation step for free as part of a shuffle.

While not germane to this article, we would like to point out another important reason behind the success of Map-Reduce. In modern systems, the network is much faster than disk, but the network is a shared resource. By having many Mappers and Reducers, the same shared network bandwidth drives many disks during the shuffle phase. Just like writing to disk in the shuffle phase hides the cost of aggregation, using a shared network hides the cost of disk accesses. It would be very interesting to see how large-scale adoption of faster solid state disks changes this equation.

4 The Aggregation Exception

It is also important to point out that there are several aspects of typical Map-Reduce systems that we do not model, most notably the *Combine* operation, which is like running a Reducer locally at each Mapper. The combine operation is the most beneficial for aggregation operations, where we need to apply a simple operator such as sum, max, or average to all Map Records. To capture the benefit of Combination, we need to introduce the number of Mappers, K, into our complexity measures. This gives the following complexity for sum/max/average and many similar aggregation functions, where N is the number of Map Records (assuming batched Reducers):

- Key complexity: The size, time, and memory are all $O(K)$.
- Sequential complexity: The total size and running time are both $O(N)$.

[1]Of course, if we were to take disk usage into account, a Map-Reduce algorithm would use more memory than the PRAM model.

With streaming Reducers, the key complexity becomes $O(K)$ (size and time), and $O(1)$ (memory). In practical installations, $O(K)$ is typically negligible compared to the coordination overhead in a Map-Reduce phase. Hence, we recommend just treating the key complexity as $O(1)$ for these operations.

For researchers who disagree with our recommendation (or where the distinction is important in the problem), using K explicitly in the complexity measures is a reasonable alternative.

References

[1] http://hadoop.apache.org.

[2] http://wiki.apache.org/hadoop/hadoopstreaming.

[3] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150. USENIX Association, 2004.

[4] Herodotos Herodotou. Hadoop performance models. *CoRR*, abs/1106.0940, 2011.

[5] H. Karloff, S. Suri, and S. Vassilvitskii. A model of computation for mapreduce. *Proceedings of the Twenty-First Annual Symposium on Discrete Algorithms (SODA '10)*, 2010.

[6] S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. *Proceedings of the 20th International Conference on World Wide Web (WWW '11)*, pages 607–614, 2011.

[7] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[8] Reza Bosagh Zadeh and Ashish Goel. Dimension independent similarity computation. *CoRR*, abs/1206.2082, 2012.