Distributed Graph Coloring with Spark

Sue Anne Davis and Kelsey Pownell

Project Overview

For our project, we implemented the Kuhn-Wattenhofer distributed graph coloring algorithm sequentially in python and concurrently in Spark.

Graph Coloring Problem

The problem of graph coloring involves coloring a graph with undirected edges in such a way that, if two nodes have an edge between them, they must have different colors. Finding the minimum number of colors with which a graph can be colored is difficult, but it is always possible to color a graph with $\Delta + 1$ colors, where Δ is the maximum degree of any node in the graph. Therefore, we focused our project on finding a $(\Delta + 1)$-coloring for randomly generated graphs.

The Kuhn-Wattenhofer Algorithm

The Kuhn-Wattenhofer algorithm is a greedy algorithm for finding a $(\Delta + 1)$-coloring for a graph in a distributed setting. It is known as a "color reduction" algorithm, because it first assigns each node its own individual color, and then iteratively reduces the number of colors down to $\Delta + 1$. The algorithm is as follows:

1. Begin with each node assigned to its own unique color
2. Separate colors into bins each containing $(\Delta + 1) * 2$ colors
3. In parallel, reduce each bin to $\Delta + 1$ colors:
   a. Choose one color c at a time to reduce, beginning with the $(\Delta + 1)$th color in the bin and proceeding to the $((\Delta + 1) * 2)$th color
   b. In parallel, assign each node of color c to one of the first $\Delta + 1$ colors with which it is compatible
   c. Repeat a and b until the bin contains only $\Delta + 1$ colors
4. Repeat steps 2 and 3 until only $\Delta + 1$ colors remain in the overall dataset

Sequential Solution

Our sequential solution consists of the Kuhn-Wattenhofer algorithm, but with all parallel steps executed serially. We used a naive coloring solution in this version for reducing colors in bins, which consisted of:

1. Begin with each node assigned to its own unique color
2. For each node $\Delta + 2$ and on:
   a. Loop over the first $\Delta + 1$ until we find a color that we don't have an edge with
   b. Place into that color
3. Once we have finished placing all nodes $\Delta + 2$ to the end, then we have $\Delta + 1$ colors.

Spark

The Kuhn-Wattenhofer algorithm only performs better than a naive graph coloring solution when running in a distributed setting. We decided to use Spark for this purpose because it is well-suited to iterative algorithms such as this one. In addition, Spark naturally divides data into partitions, which we exploited to divide the colors into bins. We used pyspark to program our project because python is the language with which we are both most familiar.

Design and Implementation Decisions

Inputs

For our inputs, we randomly generated graphs, where for each graph on n nodes, all nodes had a degree of log(n). This ensures that, while the number of colors required for each graph was far less than the number of nodes, and each graph was uniform in its distribution of edges, the maximum degree increased as the size of the graph did. It seemed to us that, as our input graphs grew in size, they should also grow in complexity. Because we already knew the maximum degree of each input file, we did not loop through each node and count the edges to find it.

Implementation

As previously stated, we decided to exploit Spark's partitioning capabilities to divide the colors into bins. Our RDDs consisted of pairs mapping integers (representing colors) to lists of nodes (represented as an integer index and a list of integers representing nodes which that node has an edge to). At each iteration, we divide the RDD into (number of elements / ((Δ + 1) * 2)) partitions, and call mapPartitions to reduce the number of colors in each bin.

For the color reduction process, we met with some difficulty, because this step, which is already occurring in parallel, requires additional parallelization (each node for a given color must be processed in parallel). To solve this problem, we combined Spark's distributed processing with regular Python thread pools. While the parallelization of bins is handled by Spark, nodes of a given color are processed by threads in a thread pool which is spawned during the mapPartitions step and destroyed afterwards.

Cluster

We ran our Spark program on a cluster using the trial version of Google's Cloud Dataproc service. The machines provided ran on Debian 9, and job distribution was handles with Yarn. We used the Google File System to store our input files and provide access to them from our cluster workers. Details about cluster configuration can be found in the next section.

Limitations/Roadblocks

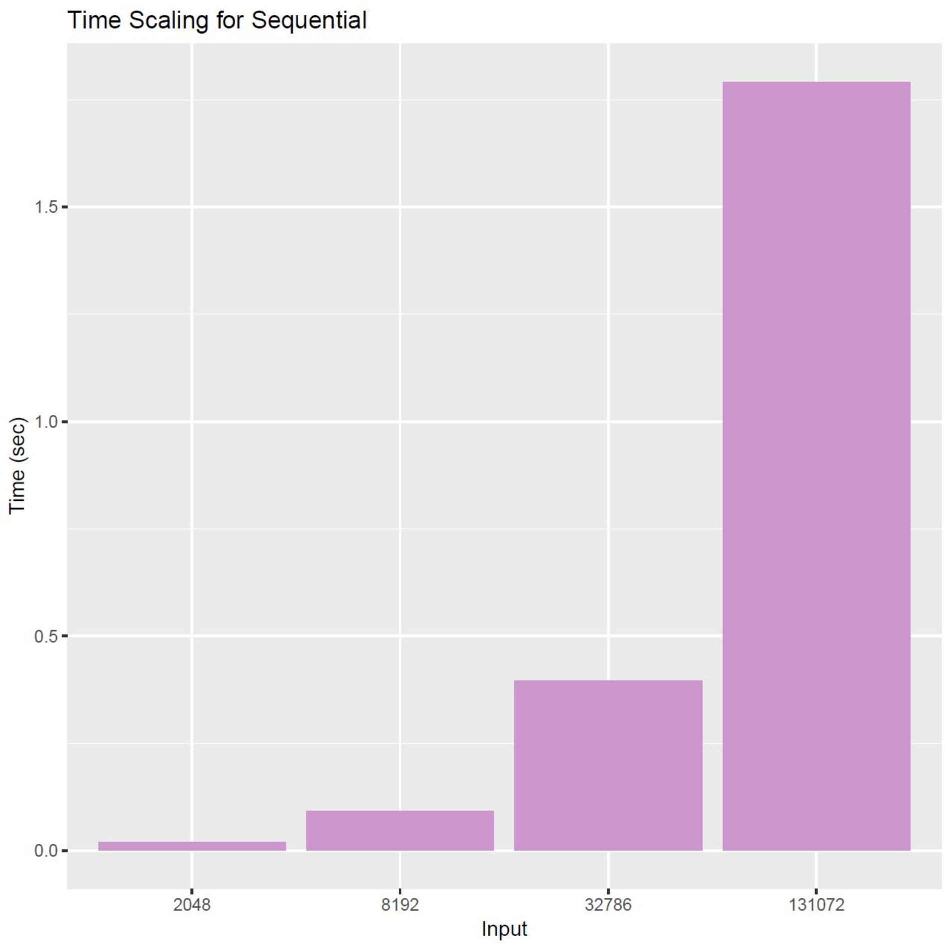
Our main difficulty was in learning to use Spark and Cloud Dataproc, both of which have very poor documentation. We were also limited in the scale at which we could run our program, because Cloud Dataproc imposed quotas on how many workers and CPUs we could allocate to one cluster. With more time and resources, we would have experimented with larger cluster scales, and we then would have had more points of comparison between the impact of horizontal and vertical scaling.

In our original project plan, we intended to also implement the Linial Algorithm for distributed graph coloring. However, we later realized that this algorithm does not result in a $\Delta + 1$ coloring, so it could not be compared as accurately to our Kuhn-Wattenhofer implementation. We decided that the investment into implementing the Linial algorithm versus understanding Dataproc more thoroughly was not a wise one.

Evaluation

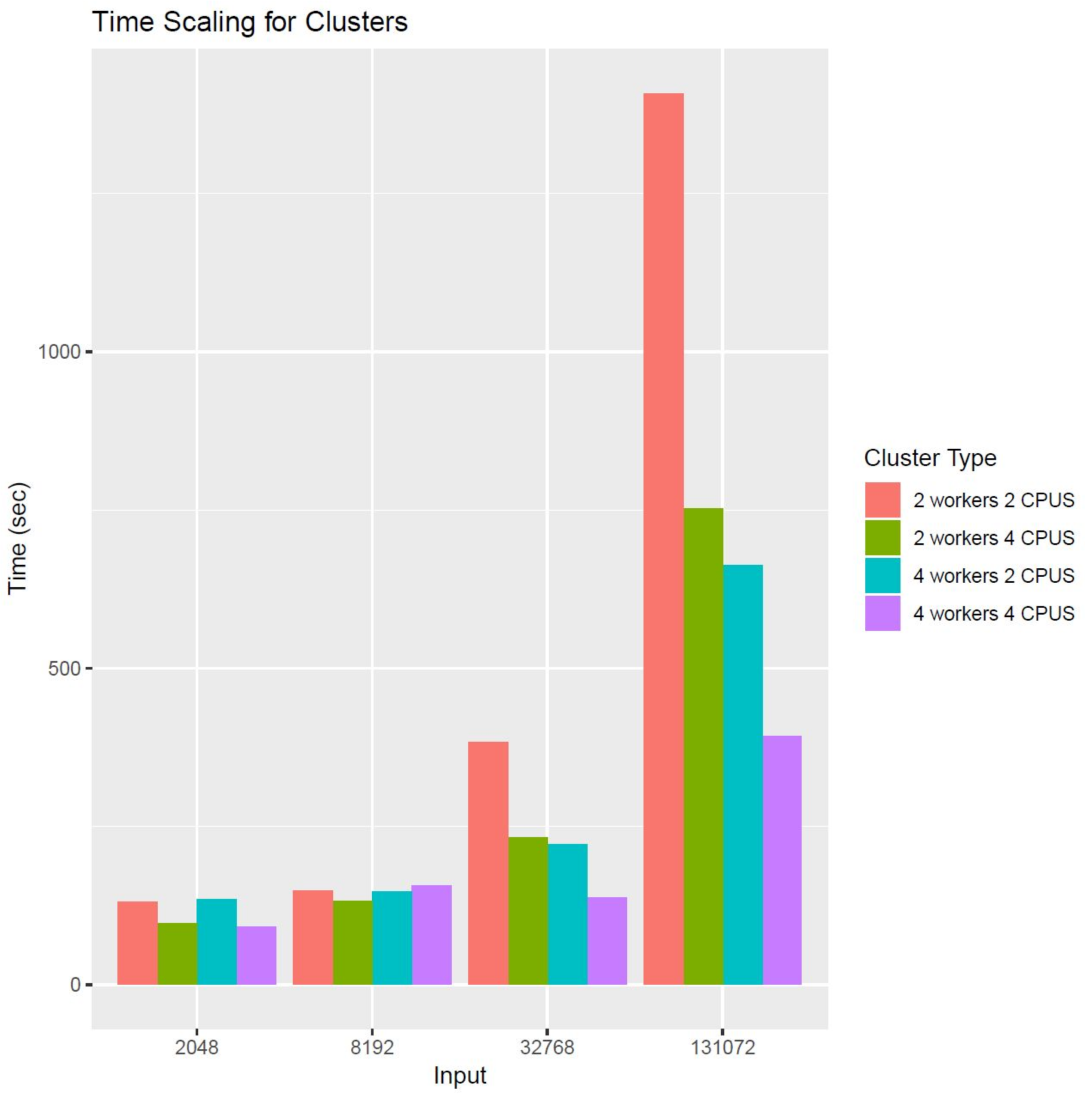
Sequential Solution

We ran our sequential solution on one of our personal computers, a two-core, four-hyperthread machine running Ubuntu 16.04. We used inputs of sizes 2048 (with $\Delta = 7$), 8192 ($\Delta = 9$), 32,786 ($\Delta = 10$), and 131,072 ($\Delta = 11$).



Distributed Solution

We ran our program using Cloud Dataproc, with clusters of two workers and four workers. For each number of workers, we experimented with worker machines that had two cores and four cores. We used the same inputs as for the sequential solution.



As you can see, the result scales well as more machines and cores are added to the cluster. Since the 2-worker, 4-CPU cluster and 4-worker, 2-CPU cluster had the same number of cores, they had similar runtimes. However, the 2-worker, 4-CPU cluster scales slightly better for larger inputs, likely because of the increased overhead of communicating across a network for a 4-worker cluster. There is also the fact that Spark is not optimized for smaller inputs, making the process of creating many smaller RDDs rather expensive, to negate the benefits of parallelization.

Overall, our spark implementation was much slower than our sequential implementation. Again, this is likely because of the overhead of communicating between processors and machines, and Spark's optimizations for larger data sets. However, our Spark implementation displays better scalability trends, particularly for the larger clusters. If we had the resources to use larger clusters and larger inputs, we would expect to see more performance gains from the distributed setting.