A Distributed Greedy Graph Coloring Algorithm

Ryan Williams

CSCI 693. Spring 2016.

Instructor: Harris, Elena Ph.D.

Advisor: Gibson, Todd Ph.D.

Problem Statement

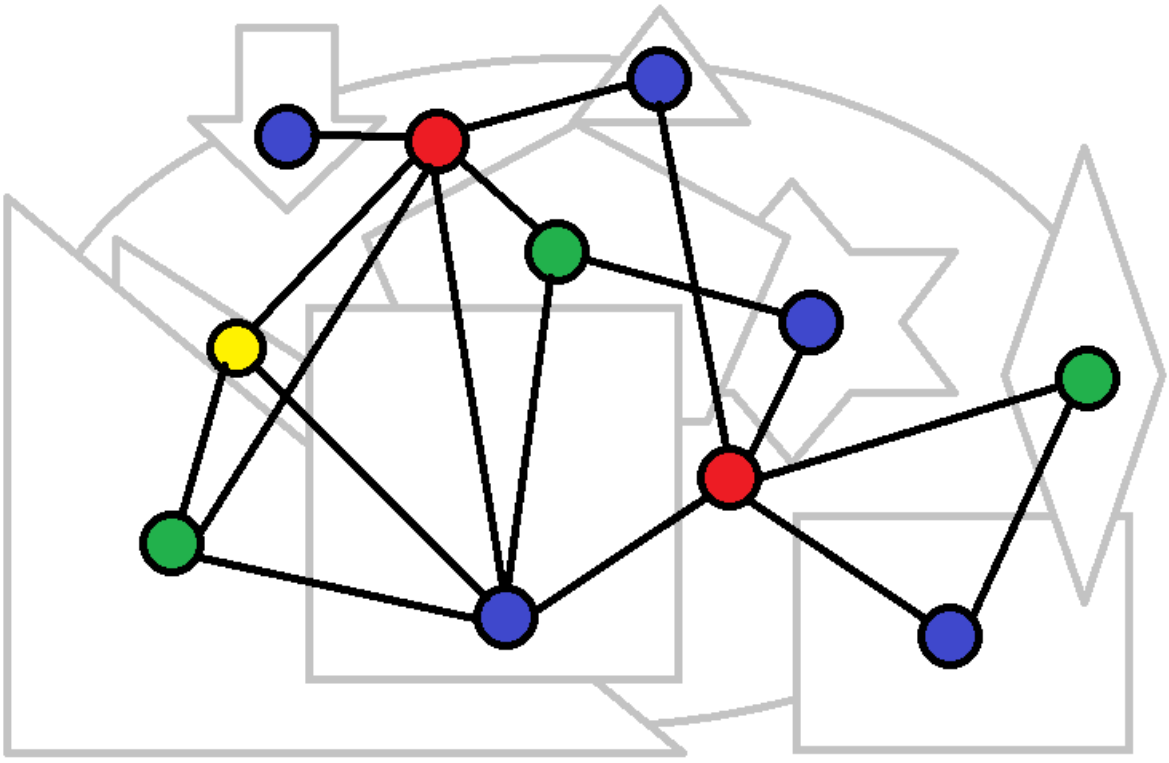
Existing Distributed Graph Coloring Algorithms often rely upon a multi-round cycle to perform tie-breaking amongst candidate vertices that are ready to finalize their coloring. The DLFDAG algorithm presented in this paper instead establishes parent-child relationships amongst all of the vertices in the graph in O(1) rounds, effectively orienting all of the edges in the graph and transforming it into a Directed Acyclic Graph (DAG). Vertices wait to receive a ColorMessage from each of its parents, if there are any, and then immediately calculates its own color and sends a ColorMessage to each of its children for the next round. The goal of this implementation is to speedup the algorithm, reducing the number of rounds required to color a Graph.

Background

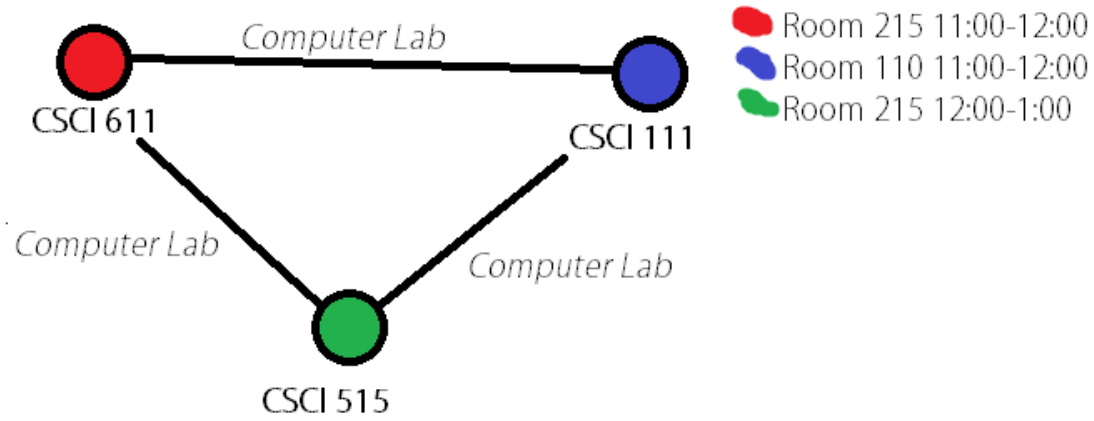
Back in the 1850's, one of Augustus De Morgan's students came to him seeking confirmation of a conjecture they had: any map of continuous planar regions, such as a map of England, could be colored using just four colors such that no two adjacent regions shared the same color. De Morgan was unable to confirm or deny their conjecture, but he became interested in the problem – asking other researchers for their thoughts on the problem, and writing a review of the problem in the academic journal Athenaeum in 1860.

In 1879, Alfred Kempe put forth the first attempted proof for what became known as the Four Color Theorem. However, in 1889 John Heawood demonstrated that there were errors in Kempe's proof. Simultaneously, Heawood proved the Five Color Theorem. There were many more attempts and failures over the years. Finally in 1976 Kenneth Appel and Wolfgang Haken used extensive computer computations to prove the Theorem. However, it has been a highly contested proof – in no small part because of the sheer number of calculations performed using computers that prevent one from validating the proof by hand. It is the first mathematical proof to be established in this manner.

Researchers have used various abstractions to evaluate the Four Color Theorem and to develop Coloring Algorithms. One of the most important abstractions is to represent a map as a graph. Below we can see an example of a map of continuous planar regions translated into a corresponding graph such that each region is represented as a vertex. Where two regions share a boundary (not just a point), the vertices for those two regions are connected together by an undirected edge. A 4-coloring has been applied to the graph so that no two adjacent vertices share the same color.



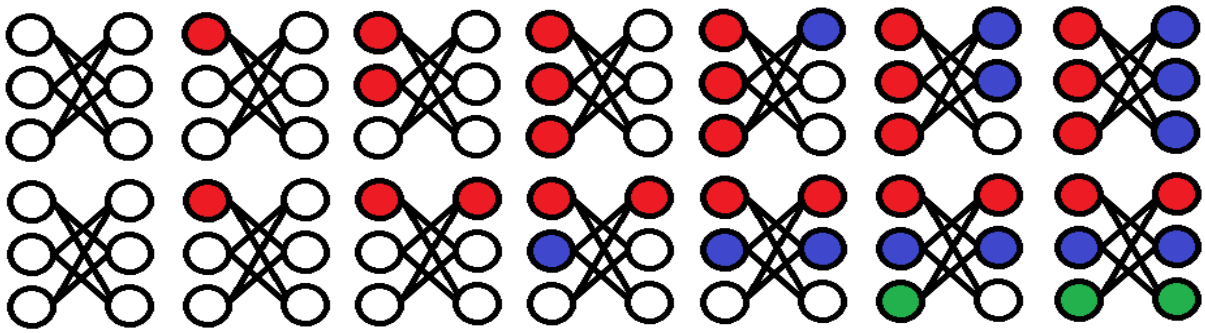
In Computer Science, Graph Colorings have been found to be useful for such applications as Resource Allocation and Task Scheduling. Below is a simple example of how we might use Graph Coloring to assign a (room, time) pair for different classes that need a Computer Lab. Instead of representing planar regions, each vertex represents a class. The edges between the vertices indicate that the (room, time) assignments are mutually exclusive. The colors represent the (room, time) pair assignment based upon the corresponding legend.



Greedy Graph Coloring Algorithms

An Algorithm which applies a color to each vertex in a graph G so that no two adjacent vertices share the same color assignment is called a Graph Coloring Algorithm, or, more specifically, a Vertex Coloring Algorithm. A Greedy Coloring Algorithm is one such kind of Graph Coloring Algorithm which works by sequentially visiting each vertex in the graph G in some order. As each vertex is visited, it is assigned the lowest possible color value at the time, where the color is represented as an integer value. Greedy Coloring Algorithms will use, at most, (Δ + 1) colors where Δ is the largest vertex degree in the graph G.

Different Greedy Coloring Algorithms use different algorithms for determining the order in which to visit the vertices. The order that the vertices are visited can make a big difference in the performance of the Algorithm. Below are two different colorings for the same graph. The top graph has an ideal 2-coloring; no fewer colors could have been used. If we were to expand the graph, adding more rows and coloring them in the same fashion, we could continue to use just two colors. Beneath it is the worst-case coloring for the graph, using 3 colors. If we were to expand the graph, adding more rows and coloring them in the same fashion, we would use one more color per row.



One kind of Greedy Graph Coloring Algorithm is the Largest First (LF) Coloring Algorithm, which prioritizes assigning lower color values to vertices of higher degree before those of a lower degree. This is the approach that DLFDAG will use, as it has been shown to be able to outperform other Greedy Algorithms when distributed. [TO DO: REFERENCE NEEDED]

Distributed Computational Model

Linial developed a computational model for distributed graph algorithms which is commonly used by other researchers, and is the model we will use here. In this model, each vertex has its own processor and each edge represents a bi-directional communication link between processors. Each vertex knows its own id, the id of its neighbors, and whatever additional local information the algorithm calls for. There is no shared memory between vertices, but adjacent vertices may send messages to one another. In this model, there is no restriction on the size of such messages.[2]

Distributed Graph Algorithms using this model will execute in synchronous rounds such that every vertex will run an algorithm on its processor for the round. Every vertex must complete executing before the next round can begin. This will continue until some end condition is met –

such as every vertex in the graph being colored so that no two adjacent vertices share the same color.

For Distributed Graph Coloring Algorithms, there are two primary performance metrics. $C_{DA}(G)$ is the number of distinct color values used by the algorithm DA to color the graph G. Ideally as few colors are used as possible. The minimum number of colors needed to color a graph G is called the Chromatic Number χ(G). Determining χ(G) is an NP-Hard problem. The other primary performance metric is $T_{DA}(G)$, which is the number of synchronous rounds used by algorithm DA to generate a coloring for graph G. Once again, it is ideal to use as few rounds as possible.

DLF Algorithm

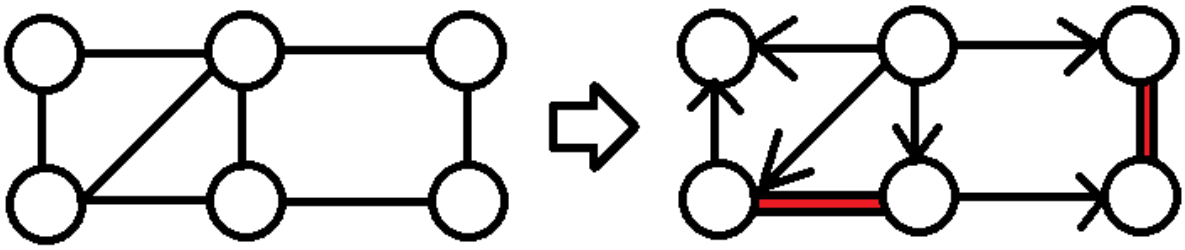
Below is a near-optimal Distributed LF Greedy Graph Coloring Algorithm [1]. Note how there is a 3-round cycle used for attempting to finalize the colors for some set of vertices. The first round in the cycle updates the colors of all vertices that haven't yet finalized their colors so that they do not conflict with previously colored vertices. The second round in the cycle generates a local random value for each vertex that is ready to be colored. The third round each vertex compares its local random value against that of its neighbors. If a vertex has the greatest random value amongst all of its neighbors, that vertex wins and can finalize its color, else it loses and must wait until the next 3-round cycle to try again. Thus 2/3 of the rounds for this algorithm are dedicated to tie-breaking.

algorithm DLF(G):
Round 0:
  $f(v) := \mathit{false};\ d(v) = \deg_G(v);$
Round $3k+1$:
  if $f(v) = \mathit{false}$
    then while $\exists_{u \in N_{\geq}(v)}(c(u) = c(v) \wedge f(u) = \mathit{true})$
      do $c(v) := c(v) + 1;$
Round $3k+2$:
  $r(v) := 0;$
  if $f(v) = \mathit{false} \wedge c(v) < \min_{u \in N_{>}(v)}\{c(u) : f(u) = \mathit{false}\}$
    then if $1 = \mathrm{rnd}[1, 2 \cdot |\{u \in N(v) : d(u) = d(v) \wedge c(u) = c(v)\}|]$
      then $r(v) := \mathrm{rnd}[0, d(v)^4];$
Round $3k+3$:
  if $r(v) > \max_{\{u \in N(v) : d(u) = d(v) \wedge c(u) = c(v)\}} r(u)$
    then $f(v) := \mathit{true};$

Designing the DLFDAG Algorithm

To design a more time-efficient algorithm, using less synchronous rounds than other Distributed Greedy Algorithms like the DLF algorithm presented above, an alternative to the repetitive tie-breaking cycles was necessary. For DLFDAG this was accomplished by assigning parent-child relationships and orienting all of the edges in the graph in such a way that the graph effectively becomes a Directed Acyclic Graph (DAG).

Parent-Child relationships are assigned in O(1) rounds. This is accomplished by having each vertex send an OrientationMessage to each of its neighbors in the first round. This OrientationMessage contains the vertex's id, degree, and a random priority value. In the second round, each vertex processes each of the OrientationMessages that it has received, comparing its id, degree, and priority values against those reported by its neighbors. First the degrees are compared. Where a vertex has a greater degree than its neighbor, that vertex is treated as the parent of the neighbor so that it must be colored prior to the child neighbor vertex. This will partially orient the graph, as seen below:



However, this does not fully orient the graph. Note the edges that have been colored red in the example graph above. These are edges that connect vertices of equal degree. To fully orient the graph and produce a DAG, these must be oriented as well. This is where the random priority values come into play, vertices that have a lower priority value will become the parent of neighboring vertices with a higher priority value. Finally, in the event that two vertices of equal degree happen to generate the same priority value, we fall back to comparing the ids of the vertices so that vertices with a lower id value will become the parent of vertices with a greater degree.

Once the graph has been effectively transformed into a DAG, coloring the graph becomes trivial. Starting with any root nodes in the graph, we assign them the lowest possible color. We then send a ColorMessage to any children vertices. Once a child vertex has received a ColorMessage for each of its parents, the child vertex will calculate its own color and will in turn send a ColorMessage to each of its children. This continues until all vertices have been assigned a color.

The DLFDAG Algorithm

Below is the pseudo-code for the DLFDAG Algorithm. The algorithm will execute in L + 2 rounds, where L is the longest path in the now oriented graph G.

```
Algorithm DLFDAG(G):
Round 0
        c(v) = 1;
        priority(v) = Random.nextLong();
        UncoloredParents(v) = {};
        Children(v) = {};
        ParentColors(v) = {};
        send OrientationMessage(id(v), deg(v), priority(v)) to all neighbors

Round 1
        Foreach OrientationMessage(id(u), deg(u), priority(u))
        {       if(deg(v) > deg(u)) Children(v) += id(u)
                else if(deg(v) < deg(u)) UncoloredParents(v) += id(u)
                else if(priority(v) < priority(u)) Children(v) += id(u)
                else if(priority(v) > priority(u)) UncoloredParents(v) += id(u)
                else if(id(v) < id(u)) Children(v) += id(u)
                else UncoloredParents(v) += id(u)
        }
        if(UncoloredParents(v).isEmpty())
        {       c(v) = 0;
                send ColorMessage(id(v), c(v)) to all Children
        }

Round 2+
        Foreach ColorMessage(id(u), c(u))
        {
                UncoloredParents(v) = id(u)
                ParentColors(v) += c(u)
        }
        if(c(v) == 1 && UncoloredParents(v).isEmpty())
        {       c(v) = 0;
                ParentColors(v) = sort(ParentColors(v))
                for(i = 0; i < ParentColors(v).size; i++)
                {
                        if(c(v) == ParentColors(v)[i]) c(v) += 1
                        else if(c(v) < ParentColors(v)[i]) break;
                }
                send ColorMessage(id(v), c(v)) to all Children
        }
```

Methodology

In order to test the performance of my algorithm, an implementation was written in the Scala Programming Language (version 2.10.6) using the GraphX Library provided as part of Spark (version 1.6.1). I used the scripts provided by Spark for launching a cluster of 16 servers (1 master, 15 slaves) with Amazon Web Services EC2. The cluster was configured to also serve as a

Hadoop (version 2.4.0) Distributed File System, which is where the resulting graphs and results were stored.

As part of the implementation, a UnitTest class was provided with a command-line interface. Based upon the arguments passed to it, it would randomly generate X graphs and color them Y times. Two parameters controlled the random generation of the graphs. The order argument specified the order of the generated graphs, aka the number of vertices. The density argument controlled how many edges to use the generated graphs, where density is the ratio between the number of edges in the graph versus the total possible edges in the graph (|V| * (|V| - 1))/2.

Density = 2 * |E| / (|V| * (|V| - 1))

As the UnitTest program generates and colors the graphs, it keeps track of various performance metrics, like the average number of colors used C_{DLFDAG} and the average number of rounds used to color the graph T_{DLFDAG}. The final results are saved to a file.

Results

For the results seen below, 1000 graphs were generated and colored for each row of the DLFDAG results. However, the DLF results were taken straight from the paper [TO DO: REFERENCE NEEDED]. The DLF results were thus generated from a separate set of graphs and thus the comparison below must be taken with a grain of salt. However, the graphs that the DLF were generated from did share the same **n** and **density** values. More in-depth benchmark testing is necessary to properly compare the performance of these two algorithms. Speedup is calculated for each row with the following formula: $100 * (T_{DLF} - T_{DLFDAG}) / T_{DLF}$.

n	**density**	C_{DLFDAG}	T_{DLFDAG}	C_{DLF} [1]	T_{DLF} [1]	Speedup
100	.001	2.00	3.02	2.00	9.67	68.77 %
	.005	2.11	4.28	2.11	12.14	64.75 %
	.025	3.42	7.97	3.48	15.68	49.17 %
500	.001	2.45	5.03	2.52	15.57	67.69 %
	.005	3.93	9.60	3.97	19.21	50.02 %
	.025	4.46	28.20	7.71	33.73	16.40 %
2500	.001	4.00	10.96	4.00	22.59	51.48 %
	.005	4.99	32.56	8.00	37.67	13.57 %

We see that for each set of graphs that were tested against, DLFDAG completed in fewer synchronous rounds than the DLF algorithm. In the best case we see a speedup by as much as 68.77 %, or about 2/3. This is the about the same number of rounds used by the DLF algorithm for its tie-breaking.

As the graphs get larger, with more edges, the speedup decreases to as low as a 13.57 % speedup. In these cases I think that this decrease can be attributed to differences in the underlying datasets that are being tested against. Again, more testing is needed to properly evaluate how much of a speedup DLFDAG provides over the DLF algorithm.

Conclusions and Future Research

A new Distributed LF Greedy Graph Coloring Algorithm has been presented in this paper: DLFDAG. It effectively converts the graph into a DAG in O(1) rounds, eliminating the need for repetitive tie-breaking cycles as is seen in other Distributed Graph Coloring Algorithms. Initial results and comparisons were provided that showed as much as a 68.77 % speedup as compared to another, near optimal, Distributed LF Greedy Graph Coloring Algorithm. However, while these initial results are promising they must be further evaluated with more in-depth benchmark testing, since the results from the two algorithms are based upon two different data sets.

For future research, such benchmark testing should take priority. The testing should also be expanded to compare against other Distributed Greedy Graph Coloring Algorithms.

Pre-requisites to compiling the code

While the code can be compiled on any machine using JDK 1.7, the accompanying scripts used are written for a Unix/Linux OS and will not work properly as-is in Windows. Additionally, to avoid binary compatibility issues, make sure that you have installed and are using JDK 1.7 and not an earlier or later version. You can find the Java Development Kit at http://www.oracle.com/. Follow the installation instructions for Linux.

The source code is written in Scala (version 2.10.6), an open-source programming language built on top of Java. You can download Scala from http://www.scala-lang.org. Note that while there are newer versions of Scala available at the time of this writing, 2.10.6 is a stable release that is relatively easy to use with Spark and Amazon Web Services (AWS) EC2. Make sure to follow the installation instructions for Linux. If you use version 2.11+, do not expect these instructions to work as-is.

Maven is an open-source Apache Project for managing projects. It is necessary for compiling both Spark and the source code. It can be downloaded from https://maven.apache.org/. Maven version 3.3.3+ is required.

The code makes heavy use of Spark (version 1.6.1) and its GraphX library. You can download Spark from http://spark.apache.org/. You will be provided will several download options, make sure to download the source code for version 1.6.1. Their website provides documentation for

how to compile spark. After downloading Spark locally, updating the PATH to include the Spark bin directory and setting up the SPARK_HOME environment variable, as per the instructions provided on the website, perform the following commands to compile Spark with Hadoop 2.4.0 and Yarn:

```
cd $SPARK_HOME

mvn -Pyarn -Phadoop-2.4 -Dhadoop.version=2.4.0 -DskipTests clean package
```

If the above completes successfully, you should be able to find the following file: $SPARK_HOME/assembly/target/scala-2.10/spark-assembly-1.6.1-hadoop2.4.0.jar. This file must exist for the source code to compile correctly based upon its current configuration. If you use a different versions of Scala, Spark, or Hadoop, the path and filename for this jar will be different and you will need to go through and modify the Project Object Model XML file (pom.xml) accordingly.

Optionally, you may also adjust the level of output that come from Spark, as it can be quite noisy by default. To do this, navigate to $SPARK_HOME/conf. Copy the log4j.properties.template and rename the copy "log4j.properties". Open log4j.properties and edit the line that reads "`log4j.rootCategory=INFO, console`" to "`log4j.rootCategory=WARN, console`"

Cloning, Compiling and Running Locally

The source code is hosted in my public Github repository https://github.com/iam1me/masters. You can download Git from https://git-scm.com/. You may also need to make an account on GitHub.com. After you have installed Git, you can use the following commands to clone the source code locally.

```
cd $home

mkdir repos

git clone https://github.com/iam1me/masters.git
```

You now have a local copy of my Masters project located under $home/repos/masters. Now run the following command to clean and build:

```
mvn clean package
```

Providing the above completes successfully, the source code will be compiled into a jar file located at $home/repos/masters/target/masters.jar. You can run the code locally with the following commands:

```
cd $home

cd repos/masters/target

$SPARK_HOME/bin/spark-submit --class csci.ryan_williams.masters.UnitTest --master
local[*] masters.jar --order=100 --density=0.001 --sample-size=10 --trials=5 output_dir
```

This will run the UnitTest class, instructing it to randomly generate 10 graphs with 100 vertices each and with an edge density of 0.001. Each graph will be colored 10 times. The program will be ran locally, distributed across all cores. The results of this process will be saved under output_dir. Note: make sure output_dir does not exist or is empty. Don't attempt to re-use the same output directory for different runs of UnitTest.

Also be aware that this process can consume a good amount of memory, can output a lot of data, and can take many hours to run depending upon the configuration parameters. When running locally, keep the settings low.

Below is a description of the resulting paths in the output directory:

Path	Description
output_dir/results	Contains the net results of the UnitTest. There will be several part_XXX files – the results are generally stored in the last such file
output_dir/graphs	All graphs that were generated as part of the UnitTest are stored here
output_dir/graphs/**i**	The directory for the **i**th graph generated by the UnitTest
output_dir/graphs/**i/j**	The directory for the **j**th coloring of the **i**th graph generated by the UnitTest
output_dir/graphs/**i/j**/edges	The directory containing the edges for the **j**th coloring of the **i**th graph. The edges are serialized in json format, distributed amongst a set of part_XXX files.
output_dir/graphs/**i/j**/vertices	The directory containing the vertices for the **j**th coloring of the **i**th graph. The vertices are serialized in json format, distributed amongst a set of part_XXX files.
output_dir/graphs/**i/j**/unittest	The directory containing the local unit test results for the **j**th coloring of the **i**th graph. The results are serialized in json format, distributed amongst a set of part_XXX files. However, usually only the last such part file contains any data.

Configuring and Running on an EC2 Cluster

Included in the root directory of the source code are some scripts used to launch an AWS EC2 Cluster, Deploy the Code to the Cluster, Login the Cluster, and Terminate the Cluster. In order to use these scripts you must first create an account with Amazon Web Services (http://aws.amazon.com).

Once you have created an account and are logged in, you will need to acquire your AWS Access Key ID and AWS Secret Key. These can be acquired by navigating the Security Credentials section under your username in the top-bar. Under the Access Keys section, click the Create New Access Key button. This will generate a new Access Key ID and Secret Access Key Pair. You will need to download or copy down this information at the time you create it, you will not be able to recover the Secret Access Key afterwards.

Once you have this information you will need to update the exportClusterSettings script located in the root directory of the source code. Namely, you will need to update the AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY settings accordingly.

Next navigate to the EC2 Management Console. From there locate the Key Pairs page under Network & Security. Create a new Key Pair. Name the key something like “masters”. You will be prompted to download a corresponding file “masters.pem.” Make sure to download and hold onto this file as you will need it for the scripts to work, and you will not be able to recover it later. After you have the file stored locally, update the exportClusterSettings script again. You will need to set AWS_KEY_PAIR to the name of the key pair you created. You will also need to update AWS_IDENTITY_FILE to point to wherever you put the .pem file that you got when you created the Key Pair. You will also need to change the permissions on the file to 600.

Also pay attention to whichever region you created the Key Pair under. You will need to update AWS_REGION in the exportClusterSettings file to match this region. You may additionally modify the type of server instances you want to create, the number of slaves, and the name of the cluster.

Once all of that configuration is out of the way, you can launch a cluster by simply running the launchCluster script in the root directory of the source code. Expect to wait a while for the servers to be launched and configured.

Once the servers are up and running, confirm that the source code has been compiled and that the masters.jar file is sitting in the target directory. From the root directory of the source code, run the deployToCluster script. This will copy the masters.jar file from your local box onto all of the servers in the cluster under the path /root/app.

Now run the loginToCluster script to ssh into the cluster’s master server. Navigate to /root/spark/conf. Edit the spark-defaults.properties file by adding the following line: “`spark.default.parallelism X`” where X is the number of partitions to use by default for the graphs being processed and other data. If you used the default settings with 15 slaves of instance type t2.large, which has 2 cores per server, then I suggest setting X to 30. You can set this value to less than the number of cores, but you don’t want to set it higher.

It is highly suggested that you also adjust the level of output that come from Spark, as it can be quite noisy by default. To do this, copy the log4j.properties.template and rename the copy "log4j.properties". Open log4j.properties and edit the line that reads "`log4j.rootCategory=INFO, console`" to "`log4j.rootCategory=WARN, console`"

Now you are ready to run the program across the cluster. Use the following command:

```
cd /root

spark/bin/spark-submit --class csci.ryan_williams.masters.UnitTest masters.jar
--order=500 --density=0.025 --sample-size=10 --trials=1 output_dir
```

This will generate 10 graphs with 500 vertices and an edge density of 0.025. It will color each graph 1 time, distributing the work across the cluster. The results will be stored on the cluster's Ephemeral Hadoop Distributed File System. While logged into the master server in the cluster, you can use the following command to pull these files down from Hadoop:

```
cd /root

ephemeral_hdfs/bin/hadoop dfs –get /user/root/output_dir /root/output_dir
```

After the above executes, you will have all the files generated by the UnitTest available on the master server's file system under /root/output_dir. You can then zip these files with the following command:

```
tar –zcvf output.tar.gz output_dir
```

You can end the ssh session by running the exit command. From your local box, you can pull the zipped output file down from the master server by running the following commands from the top-level source code directory:

```
source exportClusterSettings

source exportMasterAddr

scp –i $AWS_IDENTITY_FILE root@$AWS_MASTER_ADDR:/root/output.tar.gz
output.tar.gz
```

When you are done with the cluster, you can call the terminateCluster script to go through and terminate all of the servers on the cluster. They cost money, so don't leave them running unnecessarily. You can login to the EC2 Management Console to confirm that they have all been terminated, and/or to manually terminate the servers if the script failed for some reason.

References

[1] Kosowski, Adrian, and Lukasz Kuszner. "On Greedy Graph Coloring in the Distributed Model." *Euro-Par 2006 Parallel Processing* (2006)

[2] N. Linial. Locality in distributed graph algorithms. *SIAM Journal on Computing*, 21(1):193–201, 1992.

[3] Chartrand, Gary, and Ping Zhang. *Chromatic Graph Theory*. Boca Raton: CRC Press. 2009.