

Pregel+ Triangle Count and Label Propagation Benchmarking

David Hochberg

Summer 2026 term report for CSS 499 with Professor Fukuda

Introduction

The work done this quarter involved collecting computation time data for graph algorithms running in the Pregel+ framework. Pregel+ is a vertex-centric parallel computing graph processing program developed by Google and the Chinese University of Hong Kong for large computations on distributed memory systems. This quarter, triangle counting and label propagation algorithms were coded and their execution times were benchmarked. The main purpose of this data is to be compared with computation times of the same algorithms in the MASS framework. However, it is also useful for studying the efficiency of Pregel+ across multiple computing nodes.

Related work

Pregel is a parallel computing graph computation framework made by Google for distributed memory systems. It includes the superstep model, a vertex-centric structure, message passing, and the core utilities that make graph processing possible. Pregel+ is an open-source extension of Pregel optimized for efficiency, especially in how messages are handled. It was made by researchers at the Chinese University of Hong Kong.

MASS is also a graph processing library for distributed memory systems, developed at the University of Washington Bothell, and built on the Bulk Synchronous Parallel model. Being an agent-based computation library, MASS has a significantly different architecture than the vertex-centric Pregel+, but both libraries can be used for large graph computation. There are 3 versions of MASS, those being MASS Java, MASS C++, and MASS CUDA, each having different code and performance statistics. This quarter's work is most concerned with MASS C++. The results of these Pregel+ benchmarks will be used as a baseline of industry standards to compare with MASS C++'s execution times for triangle counting and label propagation.

Program environment

The large graph computations were performed on an array of 24 cssmpi computing nodes running Linux. The programs were run on 2, 4, 8, 12, 16, 20, and 24 machines respectively. They were also run on an additional array of 8 machines running Rocky Linux, on 2, 4, and 8 machines respectively.

Pregel+ was used to run and benchmark the algorithms. Pregel+ requires the dependencies MPI and Hadoop. MPI is a low level message passing interface between machines, and Hadoop is a framework that allows for heavy computation to run across a cluster of machines. The Pregel+ framework is a 200 kilobyte bundle of C++ code, and a Pregel+ program can be created by making a C++ file that interacts with that bundle. To run a Pregel+ program, one compiles it and uploads the executable to each machine in the cluster. The input graph file must be uploaded to Hadoop's shared filesystem. Then, MPI is used to execute the program on all machines at once. The program's output can be accessed from the Hadoop shared filesystem, but since this project is mainly benchmarking, outputs were only checked to ensure the accuracy of the computation.

The execution of a Pregel+ program is split into supersteps. On each superstep, a universal compute function is run on all active nodes. The compute function has many capabilities, including sending messages to other nodes along outgoing edges, or voting to halt, which makes the current node inactive. An inactive node can only be awoken when it receives a message from another node. The program ends once all nodes are inactive. Pregel+ also has an aggregator feature for making calculations involving all nodes, which proved useful for the triangle counting code.

Algorithm explanation

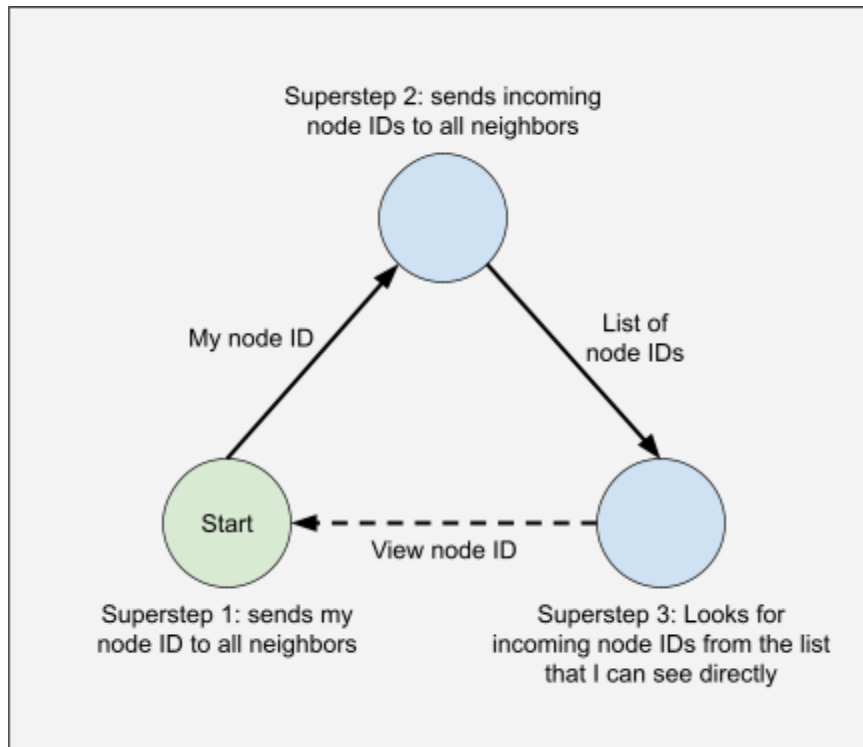
Triangle Count

The first major consideration when designing a Pregel+ triangle counting algorithm was choosing which exact type of triangle to count. We chose cyclic triangles, which occur when each node has a directed edge to exactly 1 other node in the triangle, making a cycle if one were to follow the edges. (The other two potential triangle types were transitive triangles and undirected doubly-linked triangles).

In order to detect all cyclic triangles in a graph, a 3-superstep program was used. On the first superstep, each node sent its unique ID to all of its neighbors. These messages were received by neighbors on the second superstep, giving each node a list of its incoming neighbors. The end of the second superstep was much like the first superstep, except that the messages being sent were now lists of incoming node IDs, being forwarded to all outgoing neighbors by each node. When received on the third step, these lists naturally contained an entry for each 2-edge path that ended at the current node. (If there were multiple 2-edge paths starting from the same node, then that node's ID was in multiple of these lists). On the third superstep, each node iterated its incoming lists. If one of the nodes being iterated also had an edge pointing to that same node, then a new triangle had been discovered, and the current node's individual triangle count was increased by 1. At the end of the third superstep, triangle counting was finished, and all nodes voted to halt. An aggregator added every node's individual triangle count (and divided

the total by 3 to prevent duplicate triangles), to get the total number of triangles. Figure 1 visualizes how this process is used to detect a cyclic triangle.

Figure 1: Pregel+ Cyclic Triangle Counting Algorithm



Label Propagation

Like many other implementations of label propagation, all nodes started with a unique label. Since messages in Pregel+ are received on the superstep after they are sent, it is difficult to communicate between nodes in the same superstep. Therefore, our Pregel+ implementation computed what every node should update to, and then updated them all at once at the end of each superstep.

2 additional supersteps were added at the beginning to build an initial list of each node's incoming neighbors and their labels. In the first superstep, each node sent its own label in a message to all neighboring nodes. In the second superstep, each node received all the labels and stored them in an adjacency map. The map data structure was chosen so that each label could store the number of times it occurs adjacently, avoiding repeated labels as groups converge to the same value. Once the adjacency map was constructed, the main loop was ready to begin.

The main propagation loop was a function that ran once per superstep until the propagation completely finished, with a hard limit of 300 total supersteps. Its first step was to read incoming messages, which signified that a neighbor's label had changed, and to update the adjacency

map accordingly. Then, each node used its adjacency map to find the most frequently occurring neighbor, breaking ties at random. A laziness factor of 0.25 (a 0.25 chance not to update any given node's label on each superstep) was added to help the algorithm terminate in a timely manner and prevent infinite cycles. Finally, all nodes whose values changed sent a message to all neighboring nodes so they could update their map in the next superstep. At the end of each superstep, all nodes voted to halt. This meant the program terminated unless at least one node changed its label and woke up another node.

Programmability Metrics

Figure 2 is a list of programmability metrics for the C++ programs that ran triangle count and label propagation in Pregel+.

Figure 2: Programmability Metrics for Pregel+ Triangle Count and Label Propagation

Algorithm	Triangle Count	Label Propagation
Files	2	2
Functions	17	12
Non-comment lines of code	179	196
Average non-comment LoC	7.353	11.25
Max non-comment LoC	47	72
Boilerplate percentage	12.6% (16/127)	17.5% (22/126)
Cyclomatic Complexity	16	20
Average CC	1.88	2.58
Max CC	12	17
High CC count	1	1

The file count for both programs was 2, although if needed, this could be easily decreased to 1 since both programs use a short script that calls methods in the other file. The triangle count's aggregator had 5 functions that, while only a couple lines each, increased the triangle count's number of functions by 5 over label propagation. The total lines of code were similar between programs because triangle count had shorter, less complex code, but its aggregator increased the line count. The max NLoC for each program came from the compute method that held the algorithmic logic for all supersteps. However, the average NLoC for both programs was brought down significantly by the numerous short, 1-5 line functions. The boilerplate rate, cyclomatic complexity, and average CC for both programs were moderately low, although a bit higher for label propagation. The max CC again came from the compute method of each program, and was responsible for the 1 method in each program that added to the high CC count. Since the

max CCs were moderately high, a next step for further research might be splitting these methods into smaller parts by superstep.

Methods of data collection

Three graphs were obtained from the [Stanford Large Network Dataset Collection](#) (SNAP) to use for benchmarking the triangle count and label propagation algorithms. The smallest of these graphs was web-Google, which had 875,713 nodes and 5,105,039 edges. The other two were soc-Pokec, which had 1,632,803 nodes and 30,622,564 edges, and com-Friendster, which had 65,608,366 nodes and 1,806,067,135 edges. However, there were difficulties running the com-Friendster graph due to its size and the space capacity of the computing array, so no data was able to be collected on that graph in this term.

We considered the total processing time for a Pregel+ program as the sum of the time it takes to upload the graph to Hadoop, and the time it takes for the algorithm to run. Both of these statistics were collected individually and summed in each benchmark to produce the final result. To collect the upload time to Hadoop, a shell script was used that collected the execution time of the upload command. This was run 3 times for each graph and the average was used. For benchmarking computation times, the variables that changed between runs were the algorithm (triangle count or label propagation), the graph (web-Google or soc-Pokec), and number and type of machines used for the computation. Each unique combination of these variables was run 3 times and an average was taken to get the benchmark.

Data

Figure 3: File Upload Times to Hadoop

	web-Google	soc-Pokec	com-Friendster
Average Upload Time	2.031660333	2.747158	144.740712

Figure 4: Average Pregel+ Triangle Counting Times Raw Data

	Average triangle counting time				Total processing time (upload + tri count)			
	Csshpi		Rocky		Csshpi		Rocky	
# machines	web-Google	soc-Pokec	web-Google	soc-Pokec	web-Google	soc-Pokec	web-Google	soc-Pokec
2	12.627	210.684	14.25	329.81	14.659	213.431	16.282	332.557
4	7.289	112.024	8.752	178.601	9.321	114.771	10.784	181.348
8	4.973	61.31	5.161	89.974	7.005	64.057	7.193	92.721
12	4.189	46.397			6.221	49.144		
16	3.143	35.658			5.175	38.405		

20	3.075	29.713			5.107	32.46		
24	2.662	26.148			4.694	28.895		

Figure 5: Pregel+ Triangle Counting Time for Varying Computing Nodes

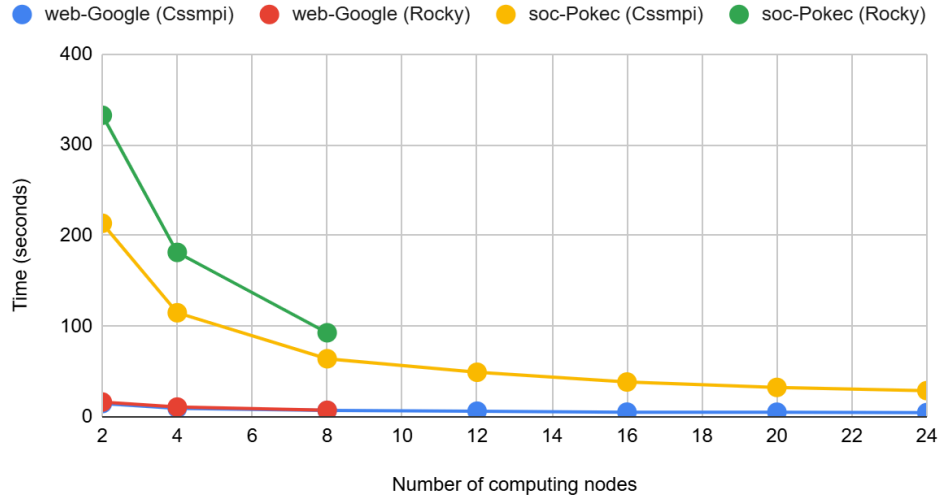


Figure 6: Pregel+ Triangle Counting Time for Varying Computing Nodes on web-Google Dataset

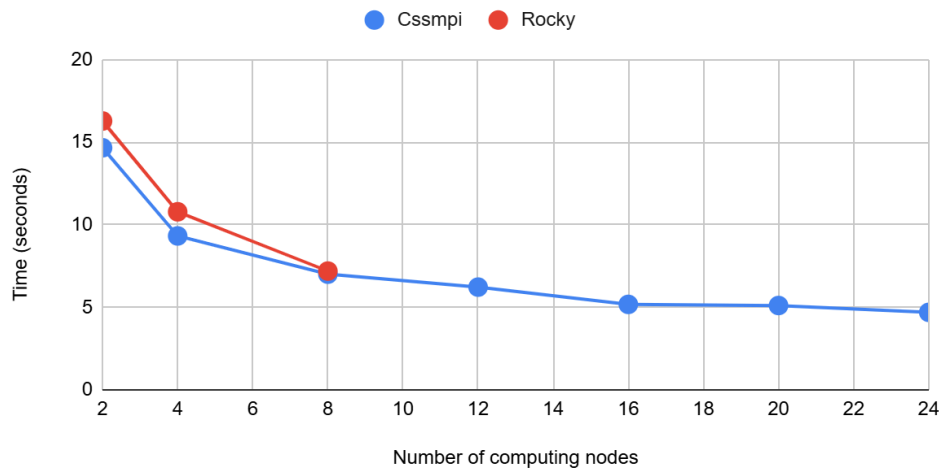


Figure 7: Average Pregel+ Label Propagation Times Raw Data

# machines	Average label propagation time				Total processing time (upload + propagation)			
	Cssmpi		Rocky		Cssmpi		Rocky	
	web-Google	soc-Pokec	web-Google	soc-Pokec	web-Google	soc-Pokec	web-Google	soc-Pokec
2	21.479	215.974	13.518	145.123	23.511	218.721	15.55	147.87
4	11.874	113.202	8.473	91	13.906	115.949	10.505	93.747

8	6.934	63.724	5.346	54.279	8.966	66.471	7.378	57.026
12	5.386	45.289			7.418	48.036		
16	5.383	38.756			7.415	41.503		
20	6.195	47.115			8.227	49.862		
24	7.454	43.222			9.486	45.969		

Figure 8: Pregel+ Label Propagation Time for Varying Computing Nodes

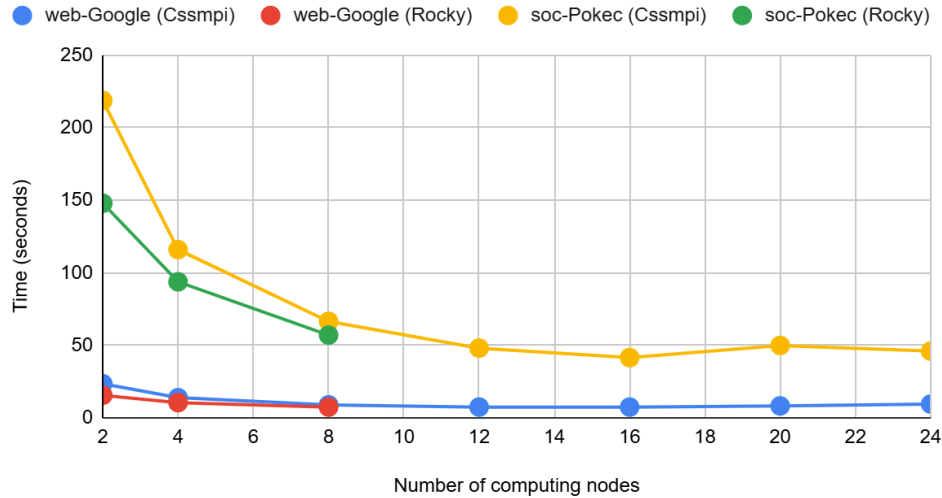
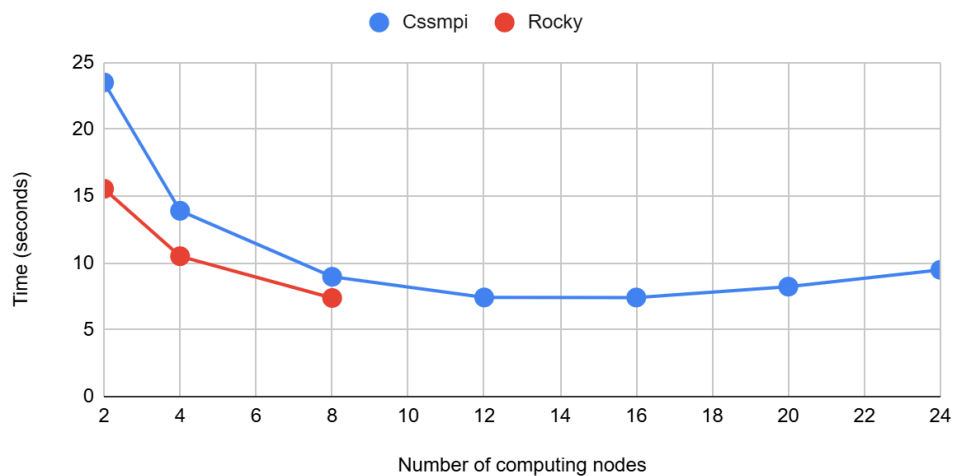


Figure 9: Pregel+ Label Propagation Time for Varying Computing Nodes on web-Google dataset



Discussion

The results from these execution times show us the best ways to run different graph algorithms in Pregel+. Triangle counting gets faster as more machines are added. However, there are

significant diminishing returns starting at about 8 machines, so in most cases it is only worth pursuing more than 8 nodes if you have the resources.

The computing times for label propagation across multiple amounts of nodes produced an upward-facing curve with the lowest time at approximately 12 or 16 computing nodes. The initial decline in computation time is likely due to sheer computing power, which decreases the required time to process these graphs. The increase during the larger numbers of nodes is likely caused by the computational overhead of connecting many machines as labels propagate all around the graph. The data for soc-Pokec fluctuates from the described pattern at 24 nodes, getting lower than the computing time on 20 nodes. This raises questions about if this result is an outlier or if more machines are required to understand the full picture. But in general, the most practical number of nodes for this type of computation is around 12 or 16.

The results also allow us to compare the two computing arrays with each other. For triangle counting on 2, 4, and 8 computers, the Cxsmapi machines processed both datasets faster. For label propagation on 2, 4, and 8 computing nodes, the Rocky machines consistently ran the computation faster than the Cxsmapi machines. This discrepancy between the two algorithms demands further research to determine what factors play a role in making one cluster faster than the other.

Conclusion

Through this project, we determined the computing times for running Pregel+ programs on different sizes of graphs. This allows for comparison with other graph processing frameworks like MASS, playing a role in helping people decide which graph framework to use. This data also helps us learn which types of machines run different Pregel+ programs best.

One consideration for future work is the efficiency gained from splitting one large input graph into smaller pieces. When the com-Friendster graph was split into 64 files, the “Load time” step in the Pregel+ took around 30 seconds, while as one big file it had taken over 200 seconds to load (the load time for com-Friendster was still readable since this step occurred before we encountered errors). Since the runtime changed based on how much the file split, further work is required to find the optimal number of files to split different graphs into. This concept hints at the idea of a graph splitting tool that automatically finds the most efficient amount of files to split a graph into.

Appendix A: How To Run Pregel+ Programs

Graph Format Conversion

Since the graphs were obtained from the [SNAP](#) dataset, which stores them in adjacency matrix format, they needed to be converted into adjacency list format. Functionally, this meant converting a file with 1 edge per line to a file with 1 node and its outgoing edges on each line.

For this project, a C++ tool was created to automate this process. It can be found at the following GitHub:

<https://github.com/David-Hochberg/pregel-plus-graph-benchmark/blob/main/graphconvert.cpp>

The command usage for this script is `./graphconvert <input file> <output file> [-d]`, where `-d` is the option to include the degree of the starting node on each line (and should be enabled for the Pregel algorithms in this project).

Pregel+ Triangle Count and Label Propagation Source Code

The following GitHub links contain the C++ code for the triangle count program:

https://github.com/David-Hochberg/pregel-plus-graph-benchmark/blob/main/pregel_app_triangle_count.h

<https://github.com/David-Hochberg/pregel-plus-graph-benchmark/blob/main/trianglecount-run.cpp>

The following GitHub links contain the C++ code for the label propagation program:

https://github.com/David-Hochberg/pregel-plus-graph-benchmark/blob/main/pregel_app_labelpropagation.h

<https://github.com/David-Hochberg/pregel-plus-graph-benchmark/blob/main/labelpropagation-run.cpp>

Setting Up the Pregel+ Environment

Before a Pregel+ program can be run, its dependencies, Hadoop and MPI, need to be set up. This project used MPICH version 5.0.0, and Hadoop 3.4.1. To set up the environment, you can follow these steps:

1. Download Pregel+ from the following URL:

<https://www.cse.cuhk.edu.hk/pregelplus/download.html>

A tutorial on running Pregel+ programs can be found at the following URL:

<https://www.cse.cuhk.edu.hk/pregelplus/console.html>

2. Download and install MPICH 5.0.0, and set up all required configuration files, including `mpd.hosts` (replacing `slaveN.domain.name` with your actual machine names):

```
slave1.domain.name
slave2.domain.name
.....
slaveN.domain.name
```

3. Download and install Hadoop 3.4.1 and set up all required configuration files, including `$HADOOP_HOME/etc/hadoop/worker` (replacing `master` and `slaveN` with your actual machine names):

```
master
slave1
slave2
.....
slaveN
```

4. You might have to update some environment variables in `.bash_profile`, as follows:

```
export HADOOP_HOME=$HOME/hadoop-3.4.1
export HADOOP_VERSION=3.4.1
export PATH=$PATH:$HADOOP_HOME/bin
export JAVA_HOME=<your java home>
export
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$JAVA_HOME/lib/server:$HADOOP_
HOME/lib/native
export
CLASSPATH=$CLASSPATH:$HOME/hadoop-3.4.1/bin/hadoop:($HOME/hadoo
p-3.4.1/bin/hadoop classpath)
```

5. Set up passwordless SSH on all machines you are using:

- Run the command `ssh-keygen -t rsa -P '' -f ~/.ssh/id_rsa`
- If you have never used SSH on these machines, you might have to sign in with your password on each machine to set up encryption first:
`ssh <username>@<machine ip>`
`exit`
- For each machine, run the following command:
`ssh-copy-id -f <username>@<machine ip>`
- If working properly, running `ssh <username>@<machine ip>` should start an SSH session in the specified machine without prompting you for a password

6. Make a program folder containing your C++ source code and the 4 Pregel+ source code folders (basic, ghost, regresp, and utils)

7. Create your Makefile in your program folder. An example is located at the following link, although you may have to make some changes depending on your setup:

<https://github.com/David-Hochberg/pregel-plus-graph-benchmark/blob/main/Makefile>

8. Compile your code using the command `make`. (If you get a compilation error, you may have to go into `<your program folder>/utils/ydhdfs1.h` and `ydhdfs2.h` and replace “default” and “0” with your master machine’s name and port). Once compiled, make sure all machines have copies of the compiled program. One way to do this is by running the following command for each machine:

```
scp <path to executable> <other machine>:<path to executable>
```

9. Create your conf file in your program folder using the same format as follows, replacing `master.domain.name` and `slaveN.domain.name` with your actual machine names, and the numbers after the colons with your desired amount of processes per machine:

```
master.domain.name:1
slave1.domain.name:1
slave2.domain.name:1
.....
slaveN.domain.name:1
```

10. Format your Hadoop file system using:

```
~/hadoop-3.4.1/bin/hdfs namenode -format
```

11. Before you can actually run your program, you need to start MPI, dfs, and yarn:

```
$HADOOP_HOME/sbin/start-dfs.sh
```

```
$HADOOP_HOME/sbin/start-yarn.sh
CSSmpdboot -n <number of machines> -v
```

12. Now that Hadoop is on, you can configure your Hadoop filesystem if you haven't already:

```
~/hadoop-3.4.1/bin/hdfs dfs -mkdir /user
~/hadoop-3.4.1/bin/hdfs dfs -mkdir /user/<your username>
~/hadoop-3.4.1/bin/hdfs dfs -chown <username>:<username>
    /user/<username>
~/hadoop-3.4.1/bin/hdfs dfsadmin -setSpaceQuota <space>
    /user/<username>
```

13. Create your Hadoop program folder. The folder name can be found (or changed) in your run.cpp:

```
~/hadoop-3.4.1/bin/hdfs dfs -mkdir /user/<username>/<folder
name>
```

14. To process your graph, it needs to be in adjacency list format (see [Graph Format Conversion](#)). Place your adjacency list graph file into your Hadoop program folder:

```
~/hadoop-3.4.1/bin/hdfs dfs -put <graph file>
    /user/<username>/<folder name>
```

Alternatively, you can use the [file upload timer](#) to automatically remove any previous graphs and upload a new one.

15. You are now ready to run your Pregel+ program. Use the following command:

```
mpiexec -machinefile <path to program folder>/conf -n <number
of machines> <path to program folder>/run
```

This will execute the Pregel+ program and print the processing times, including load time, total computation time, and the time for each superstep, to the console. If you get an error about safe mode when trying to execute, try again after turning off safe mode:

```
hdfs dfsadmin -safemode leave
```

16. After your program runs, the results from your run can be accessed in the Hadoop filesystem at the following path:

```
/user/<username>/<program folder>/<output folder name>
```

The output folder name can be found inside your run.cpp.

17. When you are done, make sure to turn off MPI, dfs, and yarn:

```
$HADOOP_HOME/sbin/stop-dfs.sh
$HADOOP_HOME/sbin/stop-yarn.sh
mpdallexit
```

Timing Hadoop File Uploads

As part of the data collection for this project, uploads to the Hadoop filesystem needed to be timed. To do so, a shell script was created that automatically uploads a file to Hadoop and prints the runtime to console. This script can be found at the following GitHub:

<https://github.com/David-Hochberg/pregel-plus-graph-benchmark/blob/main/upload-to-hadoop.sh>

To use this script, you have to edit the .sh file and replace <username> with your username for all occurrences. The command usage for the script is as follows:

```
./upload-to-hadoop.sh <path to local graph file> <name of hadoop
program folder>
```

Changing the Number of Machines

Since the number of machines needed to be changed many times, a custom shell script, called change-machine-num.sh, was created to do so automatically. This script can be found at the following GitHub:

<https://github.com/David-Hochberg/pregel-plus-graph-benchmark/blob/main/change-machine-num.sh>

The change-machine-num.sh script requires another script, rm-hadoop-tmp-files.sh, to be placed at your home directory:

<https://github.com/David-Hochberg/pregel-plus-graph-benchmark/blob/main/rm-hadoop-tmp-files.sh>

To use change-machine-num.sh, you have to edit the file and replace <username> with your username for all occurrences. Then, you need to create files called conf, mpd.hosts, and workers in the same folder as your script, and format them according to steps 2, 3, and 9 of [Setting Up the Pregel+ Environment](#). Make sure these files are in the same folder as your shell script because they will be copied to the correct locations automatically. The command usage for the change-machine-num.sh is as follows:

```
./change-machine-num.sh <number of machines> <path to local program
folder>
```

Appendix B: Raw Data

Figure 10: Raw File Upload Times

Average Hadoop upload time			web-Google upload times			soc-Pokec upload times			com-Friendster upload times		
web-Google	soc-Pokec	com-Friendster	Run 1	Run 2	Run 3	Run 1	Run 2	Run 3	Run 1	Run 2	Run 3
2.031660333	2.747158	144.740712	2.056058	2.019493	2.01943	3.716126	2.244839	2.280509	145.519858	144.908778	143.7935

Figure 11: Raw Triangle Count Data

		web-Google timings (seconds)			soc-pokec timings (seconds)		
# of machines	Task	Run 1	Run 2	Run 3	Run 1	Run 2	Run 3

2	Load	0.890867	0.809682	0.977646	2.745406	2.537638	2.458987
2	Superstep 1	3.98076	4.00544	3.83954	21.9399	21.9315	22.6058
2	Superstep 2	4.96676	4.79396	4.92661	48.4734	49.5651	49.4502
2	Superstep 3	2.89033	2.9045	2.89192	137.105	134.409	138.827
	Total computation time						
2		11.83831	11.704389	11.658628	207.519515	205.905726	210.88414
2	Total	12.729177	12.514071	12.636274	210.264921	208.443364	213.343127
2	Average total	12.62650733			210.683804		
4	Load	0.99736	0.911659	0.813218	2.488919	2.569099	2.523749
4	Superstep 1	1.9469	1.98333	1.98976	12.6389	12.5756	12.8011
4	Superstep 2	2.90802	2.90867	2.92017	30.737	31.3902	32.8939
4	Superstep 3	1.52607	1.44487	1.51336	65.0863	65.0277	65.3383
	Total computation time						
4		6.381911	6.337964	6.424281	108.463039	108.994404	111.03409
4	Total	7.379271	7.249623	7.237499	110.951958	111.563503	113.557839
4	Average total	7.288797667			112.0244333		
8	Load	1.029155	1.275765	1.018788	2.935035	3.241253	3.080767
8	Superstep 1	1.12376	1.16754	1.11935	7.6078	7.91443	7.67935
8	Superstep 2	2.13832	1.86098	1.95342	22.3151	23.3913	22.2222
8	Superstep 3	0.750561	0.736231	0.738745	27.9773	27.673	27.8874
	Total computation time						
8		4.017004	3.766103	3.813258	57.901688	58.980433	57.79077
8	Total	5.046159	5.041868	4.832046	60.836723	62.221686	60.871537
8	Average total	4.973357667			61.309982		
12	Load	1.482535	1.421409	1.141362	3.089039	3.101692	3.001366
12	Superstep 1	0.887199	0.821134	0.864926	5.7835	6.05771	5.85573
12	Superstep 2	1.40086	1.42027	1.4714	15.3494	15.4569	15.1515
12	Superstep 3	0.545693	0.54431	0.559739	22.1377	22.0917	22.1095
	Total computation time						
12		2.835521	2.787344	2.897815	43.27253	43.608434	43.118414
12	Total	4.318056	4.208753	4.039177	46.361569	46.710126	46.11978
12	Average total	4.188662			46.39715833		
16	Load	1.184444	1.046455	1.200923	3.053108	3.075029	3.189116

16	Superstep 1	0.590562	0.599034	0.569464	4.63716	4.58546	4.75334
16	Superstep 2	1.00028	0.986947	0.954933	13.2028	13.1093	13.1839
16	Superstep 3	0.441949	0.436667	0.411588	14.7076	14.7144	14.7538
16	Total computation time	2.034849	2.02496	1.938613	32.549664	32.412137	32.693449
16	Total	3.219293	3.071415	3.139536	35.602772	35.487166	35.882565
16	Average total	3.143414667			35.657501		
20	Load	1.364166	1.529642	1.272951	3.461203	3.396915	3.227803
20	Superstep 1	0.487289	0.478326	0.645239	3.83032	3.76702	3.81333
20	Superstep 2	0.754588	0.783864	0.775746	9.8443	9.88282	9.57093
20	Superstep 3	0.375572	0.376793	0.376043	12.7481	12.8355	12.7518
20	Total computation time	1.619425	1.641137	1.799014	26.424798	26.487849	26.140634
20	Total	2.983591	3.170779	3.071965	29.886001	29.884764	29.368437
20	Average total	3.075445			29.71306733		
24	Load	1.209081	1.258118	1.286165	3.446998	3.417389	3.400597
24	Superstep 1	0.428265	0.422967	0.427235	3.44358	3.52103	3.29988
24	Superstep 2	0.667573	0.65336	0.649692	8.81476	8.72872	8.72899
24	Superstep 3	0.3264	0.325383	0.322656	10.4817	10.6143	10.536
24	Total computation time	1.42463	1.404473	1.40243	22.742786	22.870343	22.567133
24	Total	2.633711	2.662591	2.688595	26.189784	26.287732	25.96773
24	Average total	2.661632333			26.14841533		
2 (Rocky)	Load	0.920606	0.842171	0.860165	2.664877	2.848336	2.560813
2 (Rocky)	Superstep 1	3.35194	3.36658	3.48984	20.9478	26.8257	21.3461
2 (Rocky)	Superstep 2	4.68608	4.64348	4.78624	48.4516	56.1253	52.3495
2 (Rocky)	Superstep 3	5.12135	5.90234	4.77804	229.915	263.637	261.758
2 (Rocky)	Total computation time	13.159594	13.912753	13.054359	299.315027	346.588227	335.454129
2 (Rocky)	Total	14.0802	14.754924	13.914524	301.979904	349.436563	338.014942
2 (Rocky)	Average total	14.24988267			329.8104697		
4 (Rocky)	Load	0.959682	0.896908	0.906473	2.958003	3.051247	2.787131
4 (Rocky)	Superstep 1	2.27717	2.03707	2.37585	12.772	13.9204	12.2306

4 (Rocky)	Superstep 2	3.60579	2.91827	3.26747	37.5406	39.2041	35.2385
4 (Rocky)	Superstep 3	2.50924	2.25551	2.24419	123.25	124.843	128.006
4 (Rocky)	Total computation time	8.392643	7.21116	7.887858	173.563218	177.968028	175.47557
4 (Rocky)	Total	9.352325	8.108068	8.794331	176.521221	181.019275	178.262701
4 (Rocky)	Average total	8.751574667			178.6010657		
8 (Rocky)	Load	1.056068	0.981539	0.946332	2.918398	2.844064	3.101573
8 (Rocky)	Superstep 1	1.25603	1.17983	1.33912	6.83772	6.80955	7.34198
8 (Rocky)	Superstep 2	1.74587	1.68578	2.05641	20.1037	19.7269	19.8238
8 (Rocky)	Superstep 3	1.09061	1.02962	1.11391	60.0003	61.6228	58.7888
8 (Rocky)	Total computation time	4.094319	3.895821	4.510054	86.942613	88.160033	85.955113
8 (Rocky)	Total	5.150387	4.87736	5.456386	89.861011	91.004097	89.056686
8 (Rocky)	Average total	5.161377667			89.97393133		

Figure 12: Raw Label Propagation Data

# of machines	Task	web-Google results			soc-pokec results		
		Run 1	Run 2	Run 3	Run 1	Run 2	Run 3
2	Load time	0.884826	0.849288	1.108352	3.072869	2.647979	3.031007
2	Total computation time (s)	20.305715	21.190375	20.099268	213.794897	211.03072	214.34566
2	Superstep #	63	123	68	133	168	135
2	Total time	21.190541	22.039663	21.20762	216.867766	213.678699	217.376667
2	Average time	21.47927467			215.9743773		
4	Load time	1.083344	1.042701	0.938317	3.111693	2.66733	2.801536
4	Total computation time (s)	10.792633	10.713926	11.049743	106.729419	110.707029	113.589043
4	Superstep #	77	49	134	141	168	136
4	Total time	11.875977	11.756627	11.98806	109.841112	113.374359	116.390579
4	Average time	11.87355467			113.2020167		
8	Load time	1.034503	1.131453	1.1251	2.759493	2.830494	2.859684

8	Total computation time (s)	5.71453	6.036186	5.761559	59.261103	61.545829	61.914707
8	Superstep #	77	129	68	151	202	205
8	Total time	6.749033	7.167639	6.886659	62.020596	64.376323	64.774391
8	Average time	6.934443667			63.72377		
12	Load time	1.130047	1.063634	1.07532	2.976896	2.944493	2.885048
12	Total computation time (s)	4.152931	4.342278	4.392613	42.968294	42.529954	41.561425
12	Superstep #	91	122	127	142	182	146
12	Total time	5.282978	5.405912	5.467933	45.94519	45.474447	44.446473
12	Average time	5.385607667			45.28870333		
16	Load time	1.288959	1.424279	1.320096	3.179987	3.200413	3.237401
16	Total computation time (s)	3.882633	4.656748	3.576596	35.008885	36.959601	34.682869
16	Superstep #	97	212	56	107	184	143
16	Total time	5.171592	6.081027	4.896692	38.188872	40.160014	37.92027
16	Average time	5.383103667			38.75638533		
20	Load time	1.677589	1.498951	1.621608	3.412601	3.465585	3.429041
20	Total computation time (s)	4.469371	4.721726	4.596598	35.254011	63.228886	32.554623
20	Superstep #	112	133	154	153	278	132
20	Total time	6.14696	6.220677	6.218206	38.666612	66.694471	35.983664
20	Average time	6.195281			47.11491567		
24	Load time	1.731998	1.730798	1.401666	3.550727	3.509451	3.649513
24	Total computation time (s)	4.566165	7.719256	5.21062	35.387632	35.214182	48.355522
24	Superstep #	58	245	116	141	177	166
24	Total time	6.298163	9.450054	6.612286	38.938359	38.723633	52.005035
24	Average time	7.453501			43.22234233		
2 (Rocky)	Load time	0.956081	0.882415	0.877001	2.588335	2.734414	2.656487
2 (Rocky)	Total computation time (s)	12.5601	12.624789	12.65414	141.143876	144.629694	141.615989

2 (Rocky)	Superstep #	51	70	47	118	117	146
2 (Rocky)	Total time	13.516181	13.507204	13.531141	143.732211	147.364108	144.272476
2 (Rocky)	Average time	13.51817533			145.1229317		
4 (Rocky)	Load time	0.952024	0.944317	0.967176	3.012715	3.103401	2.922249
4 (Rocky)	Total computation time (s)	7.348452	7.829197	7.377523	92.207924	86.112678	85.639828
4 (Rocky)	Superstep #	72	228	43	269	258	179
4 (Rocky)	Total time	8.300476	8.773514	8.344699	95.220639	89.216079	88.562077
4 (Rocky)	Average time	8.472896333			90.99959833		
8 (Rocky)	Load time	1.035238	0.989233	1.022959	2.958661	3.193781	2.937237
8 (Rocky)	Total computation time (s)	4.197662	4.120326	4.673604	54.863773	51.352031	47.532932
8 (Rocky)	Superstep #	91	64	261	180	100	174
8 (Rocky)	Total time	5.2329	5.109559	5.696563	57.822434	54.545812	50.470169
8 (Rocky)	Average time	5.346340667			54.27947167		