**P434 Final Project Report**

**Using PageRank to Analyze Components and Properties of Distributed Systems**

Jared Anson, Aaron Schwartz

**Introduction**

Overview of the Project

In this report we will detail a speedup performance analysis of outbound PageRank in sequential and MPI implementations using a fixed number of iterations or a fixed delta difference with various sizes of input data.  In lecture we have often discussed the application of using both, however, our previous experiments used 1,000 iterations and we believe that the PageRank series converges before that many iterations.  Our experiment will allow direct comparison by measuring the time of completion of both implementations in the same environment.

Elements of Distributed Systems and Technologies

In order to grasp the intentions of our experiment, a brief understanding of distributed systems is required.  The intention of a distributed system is to allow a user control over multiple computers that can be masked to look like a single cohesive system.  Communication between multiple systems acting as a server for synchronization, virtualization, sharing of cloud resources, and a single client interface to control them simultaneously were necessary elements for our purposes.  There are many other technologies for distributed systems, such as fault tolerance and p2p systems, that we learned of in class, but are not required knowledge for the scope of this report.

**Architecture and Implementation**

Brief Explanation of PageRank Algorithm and MPI PageRank

The PageRank algorithm aims to sort pages in the web by assigning a rank to each page.  The rank is based on a converging summation involving links between each page that allow the host of the link to influence the link recipient with its own rank, and other static factors.  After many iterations, each page will be ranked such that more important pages, considered to be ones that are linked to often by many other important pages, are ranked higher while other pages, that are not linked to often by other important pages, are ranked lower.  This is done by simulating a random user's random walk through the web by visiting other pages linked to on each page visited.  The equation is as follows:

PR, pagerank (a probability value)

pi , a page under consideration

L(pi), the number of outbound links on page pj

d, damping factor which can be set

N, total number of pages

The sum of all PageRank values will be one.

In the algorithm, each page has an initial rank such that all of the ranks of each node when added together sum to 1.  Typically, each node starts with a rank of 1/N where N is the total number of nodes.  Random restarts are included by using a value called a damping factor, which is usually 0.85.  The series converges after many iterations by using the ranks calculated for each node in the previous iteration as the new rank to operate on.

Running MPI PageRank on a HPC Cluster and OpenStack Cloud Infrastructure

The sequential implementation is run on a single core and is done by looping through every node's key and its values.  The algorithm was implemented to run in parallel using MPI in OpenStack Cloud using FutureSystems.  When run in parallel, each processor in the cloud is given a partition of the nodes and each processor iterates on its partition.  When every processor completes its own local calculations, they synchronize by using allreduce on their partial sums to compute the new global ranks for each node. A new iteration begins by using the previously global rank data as a local cache for local computations again.  These iterations continued until 1,000 iterations were run, if the setting was iterative, or until the largest difference between each node's new and previous ranks was less than 10^-6, if the setting was delta.

**Experiments** (2 points)

Settings

Our implementation of PageRank was coded in Java and run on a grid using the FutureSystems OpenStack Cloud.  The parallel code was implemented with MPJ for communicating between systems using an MPI interface.  The cloud allowed creation of virtual machines of various flavors in which to test our program.  We used a single m1.medium flavor VM for our sequential runs and two m1.medium VMs for parallel runs.  The sequential code was run on a single core of the VM while the parallel code was run using 8 MPI processes on 4 cores, with 2 cores per m1.medium flavor VMs.

Input

Compilation of our sequential code is done using javac, while the parallel code includes MPJ arguments.  Compilation of our code uses the following commands:

javac SequentialPageRank.java

javac itSeqPageRank.java

javac -cp .:$MPJ\_HOME/lib/mpj.jar MPIPageRank.java

javac -cp .:$MPJ\_HOME/lib/mpj.jar itPageRank.java

Here is a sample command line argument for running our parallel iterative code:

mpjrun.sh –dev niodev -np 8 itPageRank <input\_file\_name> <output\_file\_name> <iterations> <damping factor>

Here is a sample command line argument for running our parallel delta code:

mpjrun.sh –dev niodev -np 8 MPIPageRank <input\_file\_name> <output\_file\_name> <delta> <damping factor>

Here is a sample command line argument for running our sequential iterative code:

java itSeqPageRank <input\_file\_name> <output\_file\_name> <iterations> <damping factor>

Here is a sample command line argument for running our sequential delta code:

java SequentialPageRank <input\_file\_name> <output\_file\_name> <delta> <damping factor>

To run the parallel code locally, rather than on OpenStack VMs, remove –dev niodev.

For our inputs, we varied the size of the input file between 1000 and 1000000 nodes, using the generated samples.  We used 1000 iterations for every iterative run and a delta 0.000001 for delta runs.  Every run used a damping factor of 0.85.

Output

While the output of all implementations prints the top 10 PageRank values to the console, it also saves them to a text file to view all of the values. MPI\_output.txt corresponds to MPIPageRank, itMPI\_output.txt corresponds to itPageRank, Seq\_output.txt corresponds to SequentialPageRank, and itSeq\_output.txt corresponds to itSeqPageRank.

For the purposes of this experiment, we were mostly interested in the runtime to compare efficiency.  We measured the total execution time to capture the full effects of each part of our implementations.  We felt that measuring only the PageRank loop and not the initialization would favor an inbound implementation since one of the drawbacks of that implementation includes its lengthy creation of an inverse adjacency matrix. When comparing our code efficiency, we took the average of 3 runs on each input variation.  We computed speedup by dividing the sequential time by the parallel time.  All of our data is shown below.

PageRank Results (Projects #1 and #2)

For all sequential runs we used our Project 1 SequentialPageRank running on a single VM. All Delta runs were used with a delta difference of 0.000001.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Config 1 | Iterative Sequential (T1) | Iterative Parallel (T4) | Delta Sequential (T1) | Delta Parallel (T4) |
| Run 1 | 131.526 s | 260.432 s | 8.293 s | 20.989 s |
| Run 2 | 128.397 s | 264.966 s | 8.307 s | 22.355 s |
| Run 3 | 127.811 s | 255.518 s | 7.911 s | 22.806 s |
| Average | 129.245 s | 260.305 s | 8.170 s | 22.050 s |

VMs: 2 x m1.medium

Cores: 2 per m1.medium = 4 cores

Size: 100,000 URLs

MPI Processes: 8

Iterations: 1,000

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Config 2 | Iterative Sequential (T1) | Iterative Parallel (T4) | Delta Sequential (T1) | Delta Parallel (T4) |
| Run 1 | 46.094 s | 138.089 s | 3.446 s | 15.711 s |
| Run 2 | 46.947 s | 136.747 s | 3.151 s | 13.214 s |
| Run 3 | 45.273 s | 140.452 s | 3.222 s | 11.289 s |
| Average | 46.105 s | 138.429 s | 3.273 s | 13.405 s |

VMs: 2 x m1.medium

Cores: 2 per m1.medium = 4 cores

Size: 50,000 URLs

MPI Processes: 8

Iterations: 1,000

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Config 3 | Iterative Sequential (T1) | Iterative Parallel (T4) | Delta Sequential (T1) | Delta Parallel (T4) |
| Run 1 | 326.750 s | 502.953 s | 18.84 s | 36.318 s |
| Run 2 | 330.369 s | 513.624 s | 18.923 s | 37.866 s |
| Run 3 | 323.792 s | 508.384 s | 18.233 s | 37.850 s |
| Average | 326.970 s | 508.320 s | 18.665 s | 37.345 s |

VMs: 2 x m1.medium

Cores: 2 per m1.medium = 4 cores

Size: 200,000 URLs

MPI Processes: 8

Iterations: 1,000

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Config 4 | Iterative Sequential (T1) | Iterative Parallel (T4) | Delta Sequential (T1) | Delta Parallel (T4) |
| Run 1 | 0.654 s | 6.503 s | 0.300 s | 1.728 s |
| Run 2 | 0.664 s | 6.567 s | 0.294 s | 1.639 s |
| Run 3 | 0.638 s | 6.396 s | 0.313 s | 1.676 s |
| Average | 0.652 s | 6.489 s | 0.302 s | 1.681 s |

VMs: 2 x m1.medium

Cores: 2 per m1.medium = 4 cores

Size: 1,000 URLs

MPI Processes: 8

Iterations: 1,000

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Config 5 | Iterative Sequential (T1) | Iterative Parallel (T4) | Delta Sequential (T1) | Delta Parallel (T4) |
| Run 1 | 1208.824 s | 1321.977 s | 61.060 s | 92.457 s |
| Run 2 | 1105.025 s | 1321.856 s | 60.520 s | 93.696 s |
| Run 3 | 1087.121 s | 1325.683 s | 87.191 s | 93.793 s |
| Average | 1133.657 s | 1323.172 s | 59.590 s | 93.315 s |

VMs: 2 x m1.medium

Cores: 2 per m1.medium = 4 cores

Size: 500,000 URLs

MPI Processes: 8

Iterations: 1,000

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Config 6 | Iterative Sequential (T1) | Iterative Parallel (T4) | Delta Sequential (T1) | Delta Parallel (T4) |
| Run 1 | 4171.019 s | 2916.967 s | 192.405 s | 202.654 s |
| Run 2 | 4029.967 s | 2886.005 s | 192.705 s | 207.249 s |
| Run 3 | 3902.367 s | 2906.185 s | 188.644 s | 197.472 s |
| Average | 4034.451 s | 2903.052 s | 191.251 s | 202.458 s |

VMs: 2 x m1.medium

Cores: 2 per m1.medium = 4 cores

Size: 1,000,000 URLs

MPI Processes: 8

Iterations: 1,000

**Improvements**

Featured

Our featured improvement, along with a reduction in communication overhead for parallelized versions, is the inclusion of a delta to limit the number of iterations once a degree of convergence is reached.  The delta is a minimum absolute difference between values of consecutive iterations required for termination of the algorithm.  This is calculated by taking the absolute difference between iterations of computed values.  If any page's value changes by more than the given delta value, another iteration is performed until every page's difference is below the delta value.

Performance Measurements

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| URLs | Iterative Sequential (T1) | Iterative Parallel (T4) | Delta Sequential (T1) | Delta Parallel (T4) |
| 1,000 | 0.652 s | 6.489 s | 0.302 s | 1.681 s |
| 50,000 | 46.105 s | 138.429 s | 3.273 s | 13.405 s |
| 100,000 | 129.245 s | 260.305 s | 8.170 s | 22.050 s |
| 200,000 | 326.97 s | 508.320 s | 18.665 s | 37.345 s |
| 500,000 | 1133.657 s | 132.172 s | 59.590 s | 93.315 s |
| 1,000,000 | 4034.451 s | 2903.052 s | 191.251 s | 202.458 s |

|  |  |  |
| --- | --- | --- |
| URLs | Iterative Speedup | Delta Speedup |
| 10,000 | 0.1 | 0.18 |
| 50,000 | 0.33 | 0.244 |
| 100,000 | 0.497 | 0.371 |
| 200,000 | 0.643 | 0.509 |
| 500,000 | 0.847 | 0.639 |
| 1,000,000 | 1.39 | 0.945 |

Analysis of Results

The PageRank Analysis graph above shows that using sequential iterative PageRank was faster than using sequential parallel PageRank for datasets containing less than approximately 600,000 URLs.  Delta sequential PageRank ran faster than delta parallel PageRank on all data sizes.  These observations can also be seen in the Speedup Analysis graph, where any value above 1 on the y-axis indicates parallel-favored sizes, values below one indicate sequential-favored sizes, and values equal to one indicate that both versions run at the same speed.  Communication overhead for parallel PageRank was more costly for datasets below approximately 600,000 for iterative runs and below approximately 1,000,000 for delta runs.  We attempted to use a generator to create datasets larger than 10^6 URLs, but the VMs would run out of memory trying to process them.  The slopes for sequential performance analysis show a seemingly linear increase for parallel and an exponentially increasing slope for sequential, This pattern can also be seen in the delta versions, although on a smaller scale. Both of these slopes show that parallel scales better for larger data.  Delta sequential scales much better than iterative sequential because there is less focus on the PageRank calculating loop.  Calculating for fewer iterations favors the sequential algorithm since the calculations are the only parts that are parallelized.

**Conclusion**

Summary of Achievements

We feel that all of our implementations of PageRank were successful in accuracy.  Our sequential implementation was very efficient, but we believe that our parallelized implementations could be faster by reducing code blocking in order to mitigate the cost of communication.  Other speedups could be gained by changing data types to more primitive types to sacrifice precision for speed and by switching from using a hashmap and instead using arrays to cut the cost of deserialization.

Findings

Our most difficult hurdle was coordinating the communication in the parallelized version.  Managing global and local values was relatively simple, but we found that Send and Receive was more complicated than Broadcast.  Getting our parallel code to run on OpenStack was only a matter of following instructions that were given to us, and we were grateful that they were clear enough to guide us successfully.

Communication vs. Computation Overhead of MPI PageRank

The main drawback of parallelization is the communication overhead required to allow several processors to compute as though one.  The overhead comes through the need to synchronize at various steps in an algorithm which requires that every processor send a message to each other one and that each processor receives a message from every other processor as well.  Another drawback includes data partitioning in the initialization phase, though this cost is minimal compared to the overall cost of computation, assuming many iterations are needed.  The advantage of using parallelization is that the computation overhead is split amongst other processors, and that the computation does not have a single failure weakness if data partitioned with copies.  When deciding whether or not to opt for a parallel implementation, these costs must be weighed against each other, with larger amounts of data and iterations favoring the parallel implementation.

Synchronization Issues in a Distributed System

One of the most complicated issues in distributed systems is synchronization.  Since there are many computers acting as a single unit, and these computers may be any distance apart, they must agree on certain conditions and be careful when manipulating their local data and uploading it as global data.  There are many styles of synchronization that establish several relationships between processors, but ultimately they all revolve around handling state and time.  The more one system of synchronization tries to preserve state, the more time it takes to achieve its goal, thus looser synchronization is more efficient, but also has more mismatches in state that need to be resolved to function cohesively.

**Acknowledgements**

The lectures allowed for brief introduction and discussion of the topics without requiring too heavily on reading the required textbook, which is rather dry.  We found the labs to be extremely helpful in guiding our code writing and our lab AI, Supun Kamburugamuve, to be very knowledgeable, helpful, and friendly.  When we were seeking additional help outside of lab, Ethan Li helped us greatly with guiding our communication problems in our parallel implementation.  While we found the class to be challenging, we think our design choices in our programs speak for themselves in representing our knowledge of distributed systems.

**References**

The majority of our reference material can be found in Canvas under the Files and Assignments tabs.

We used a custom comparator when sorting our PageRank:

<http://www.tutorialspoint.com/java/java_using_comparator.htm>

We used the website above to learn about custom comparators, and this helped in coding our own comparator to short an Arraylist of Tuples based on the second item in each Tuple.