Nolen Young, ID: 11517296

CptS 411, Final White Paper

12-9-19

Introduction

The internet is made up of millions of web pages, with each web page linking to other web pages, creating an impossibly vast network of websites to navigate through. In order to efficiently browse the web, we need to be able to determine the importance of any specific website in the network of the internet. In the search to create an effective search engine, google invented the PageRank algorithm and named it after one of the founders of Google, Larry Page.

Because the internet is so large, the PageRank algorithm stands to gain a lot from parallelization. The algorithm may take a very long time serially to work across such a large network, but having multiple processors running the algorithm at once would give much needed speed up. In this paper, I will discuss the specifics of the problem the PageRank algorithm seeks to solve, I will discuss challenges involved in parallelizing the PageRank algorithm, I will outline my own implementation of a multithreaded version of PageRank, as well as a theoretical distributed system version of PageRank, and I will test the speed up and efficiency of my multithreaded PageRank algorithm and analyze the results.

Problem Statement

The goal of the PageRank algorithm is to take in a directed graph and output the top 5 nodes with the highest PageRank scores.

Input: Directed graph G, consisting of nodes representing webpages and edges representing links. Integer K, the length of the random walks. Decimal D, the damping ratio, where $0 < D < 1$.

Output: Top 5 nodes of PageRank

Challenges in Parallelization

Making a parallel version of the PageRank algorithm brings up a few problems. The problems all stem from data accessibility, but the problems facing a distributed system and a parallel system are very different.

First, I will talk about the problems facing a distributed system. The most obvious problem is that a distributed system must distribute the graph across all processes. So, if a given process is running on a random walk on G, it is likely that the next node in the random walk is stored in memory on a different computational node and is directly inaccessible by the processor running the walk. This presents the problem of how we want communication between the nodes to work. My goal was to think of an algorithm that would do these things while maintaining acceptable levels of speed up and efficiency. After some brief brainstorming I came up with a couple possible solutions.

The first solution I came up with to this problem is also most likely the worst solution. The graph would be divided evenly between each process. Each process would start a random walk on their portion of the graph, but when a walk leads to a section of the graph handled by another process, it would send the walk to that process to be handled by it. In this way the walks would be sent between processes, wherever the walk takes them. This is a bad solution for a few obvious reasons. The walks are likely not to be distributed between the nodes evenly, and many of the walks could end up on one process, and one process would be left to compute the bulk of the random walk, giving us very poor efficiency. Communication would also be difficult with this solution, it seems likely to me that because of the inconsistent need for communication between nodes, desynchronization is likely and missed communications between nodes will be common. Accounting for these problems will only make our speed and efficiency worst. Obviously, this solution is not what we should go for.

My second solution was much better for this problem. This solution involves using a master process and the reduce and broadcast functions. First the root process broadcasts all the walks to all the other processes, the nodes compute where the walk will go next given their portion of the graph, then reduce is called and all the information is sent back to the root node, probabilities for each node are computed and stored, then the next set of nodes in the walks are sent out to the processes. This repeats until the user is satisfied. This is a much better solution because it retains consistent communication times and partners, makes use of the very efficient reduce and broadcast, and generally is much more efficient and faster than the previous algorithm.

Now I will talk about problems with a multithreaded solution. The main problem with a multithreaded solution is data corruption. This can be cause by two processes trying to write and read data to the same location at the same time. To prevent this, we must simply code the multithreaded sections so that they do not allow data corruption. Besides that issue, there are no other real issues I could identify while programming this algorithm. There are no issues with communication overhead and desync issues like in the distributed memory approach.

Proposed Approaches

Distributed Memory

My distributed memory approach relies on two main MPI functions, broadcast and reduce. We use these functions to send out data to our processes then gather the results back to a root process to complete the walk iterations. Each walk step is synchronized together to prevent desynchronization between processes. The pseudo code is as follows.

```
Read and distribute graph to each process
Nodes = current nodes in walks

While (max_error > threshold)
    Sum = 0
    For each node in nodes // nodes = webpages
        Update probabilities matrix

    For each node in nodes
        Compute total probability, contributed by nodes neighbor
        Sum += a nodes probability / N nodes

    Mpi_reduce(MPI_SUM)

    If rank == 0 // root node
        Compute new probabilities and find max error

    Mpi_bcast(max_error, new prob matrix)
```

The pseudo code is rough, but I do hope it gives you a good idea on how the algorithm should flow. The space complexity of this is N, as each node must be stored somewhere for it to be accessed and all other variables are constant size and can be disregarded. The runtime complexity should be O((n + m) / processes), where n = the number of nodes and m = the number of edges. This is because at worst case the algorithm must visit each node once and each edge twice to finish. The processes will reduce this time by the number of processes running theoretically, but in actuality when you consider communication overhead and a slightly inefficient algorithm the runtime may actually be substantially longer than O((n + m) / processes) predicts.

Shared Memory

The shared memory solution is not nearly as complex as the distributed memory, as it is basically the serial code with a parallelized for loop. This makes the code a lot cleaner because we don't haveto send jobs to nodes, receive results, and then redistribute jobs. The pseudo code is as follows

```
Read in graph
malloc vistedNodes of type int and length of # of nodes
set lock for visitedNodes

#pragma omp parallel for
For each node in the graph // start at each node
        For I = 0, I < steps
                #pragma omp atomic
                visitedNodes[graph[node]]++;

                if(coin toss == heads)
                        move to next random node from edges
                else
                        jump to a random node
```

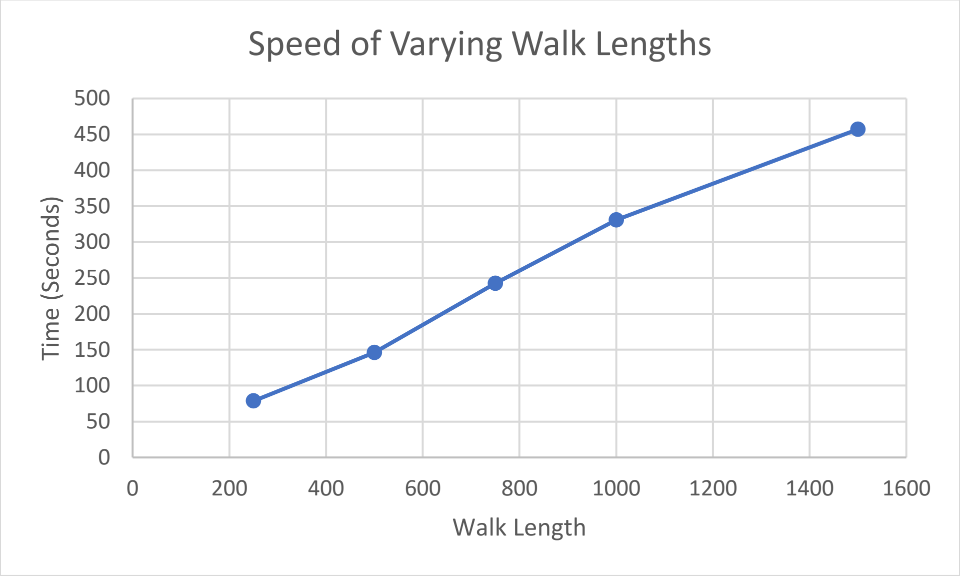
As you can see this code is much cleaner and follows the flow of the serial code almost exactly. The space complexity is 2n + m, where n is the number of nodes and m is the number of edges. The runtime complexity is O(((n * steps) / processes), where n is the number of nodes, m is the number of edges. I found this number from the algorithm starting at each node possible, and running for steps iterations, divided by the number of processes running the job

Experimental Results and Discussion

To test the effectiveness of the shared memory PageRank algorithm I will be running two different tests. The first test I will keep the number of processes static but increase the number of steps in the algorithm. I will run the test on one of the data sets provided. With this test I hope to measure speed as well as accuracy over time. The second test I will keep the number of steps constant and increase the number of processes. With this test I hope to effectively measure the speedup and efficiency of the algorithm. With these two tests I will get an idea of how effective I was at parallelizing the PageRank algorithm.

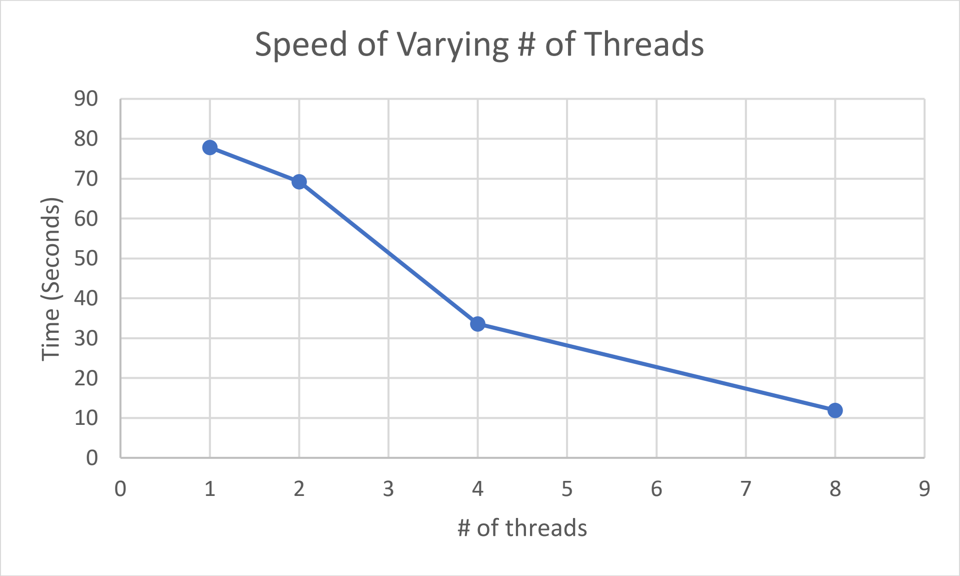
Test 1 Results

This test was ran on 4 threads using the Notre Dame data set, on the Pleiades cluster, and using a damping factor of .85. The results of the speed are as follows, I will discuss the accuracy of the results in the discussion section



Test 2 Results

This test was ran on varying threads using the Notre Dame data set, on the Pleiades cluster, and using a damping factor of .85. I will discuss these results in the discussion section.



Discussion

It is clear to see that the parallelization had major benefits on the performance of the algorithm. I will go into detail on the results of each of the tests.

Test 1 showed us most obviously that the speed of the algorithm slows down roughly linearly as you increase the walk length. What this test was really trying to pinpoint thought was the accuracy of the results from the algorithm. Since this result is much harder to quantify I will have to use my subjective observations for this. I observed that as the number of walks increased, the results of the algorithm began to converge, and become consistent. The lower walk values show disagreement between results, but as the walk length increases, the algorithm started to give similar results. This tells me that with a higher walk length, the algorithm spits out more accurate results compared to lower walk lengths. This however is a small assumption, and the most I can conclusively conclude is that algorithm becomes more consistent in result than walks over lower length. The only way for me to conclude the accuracy of the algorithm would be to have the expected results of the data set, which I do not have access to.

Test 2 showed obvious speed up when increasing the number of threads. As the number of threads increased, the time taken to compute the PageRank decreased. The exact speed ups are:

2 threads: 1.12404
4 threads: 2.3194
8 threads: 6.519

Obviously, this is sub-linear speed up, and something in the implementation is slowing down the results. However, despite this being sub theoretical speed up, I still think this is an acceptable and significant speed up for the algorithm. From this we can calculate the efficiency of the algorithm. The efficiencies are as follows:

2 threads: .56202
4 threads: .57985
8 threads: .81487

This is a strange trend as the efficiency is not consistent throughout. Mid 50% efficiency isn’t the best theoretically, and I am unsure what caused those speed ups to be so low. However, on 8 processors the speed up was significant, all the way up to 81%. This tells me that the algorithm could possibly be more efficient at 8 threads using the parameters specific to this test. Any threads less than 8 are not efficient for the data, walk length, etc.

Acknowledgements

None.

References

None.