Gregory Granito
gdg38
CS 5220

PageRank Computation Optimization

All code and data can be found at: https://github.com/ggranito/pagerank

Background

PageRank is a graph algorithm that assigns a score to each node in the graph. The more “important” the higher the score. Importance is measured by considering the number of nodes that are inbound to the node in question and considering their PageRanks.

The algorithm was originally developed by Google and was used as a basis for improving search result ordering. Google considered each webpage on the internet to be a node and each hyperlink to be an edge. Once this graph structure was created, they determined each page’s PageRank. Pages with higher PageRank were then more likely to be served as results. While PageRank is no longer a major factor in Google’s ranking algorithm, PageRank is still used for a variety of other analytic purposes including traffic management, social network analysis, and natural language processing.

The Formula

The PageRank for node u, $PR(u)$, is defined as follows:

$$PR(u) = \frac{(1-d)}{N} + d * \sum_{v \in B_u} \frac{PR(v)}{L(v)}$$

Where N is the number of nodes in the graph, B_u is the set of nodes with an edge going to u, $L(u)$ is the number of outbound edges the u has, and d is the dampening factor.

The dampening factor is equivalent to the probability that a jump to a random node is made. It is usually around 0.15.

Computation

The PageRank definition stated above describes a valid PageRank value, but does not describe how to compute it. Since it is a recurrence, PageRank is typically calculated by assigning every node the equal value of $\frac{1}{N}$. The algorithm is then run repeatedly until the value of every node has converged to a single value. This has been shown to converge after surprisingly few iterations.

Traditionally, Google has used MapReduce to compute these iterations. MapReduce is a highly parallelized computation framework that is designed to run across numerous (usually low quality) computers.

Project Goals

The goal of this project is to implement a PageRank algorithm that is designed to run on the totient cluster. To parallelize the algorithm, I will use OpenMP.

Once an algorithm is written, I will analyze it and find the places that require the most improvement. I will continue tuning to algorithm in an attempt to minimize the algorithm's running time.

Step 0: The Basics

In preparation for tuning a PageRank algorithm, I needed a way to generate graphs for the algorithm.

Unlike other algorithms, PageRank's running time depends on the graph structure. Different structures can take longer to converge. To account for this, I had to modify the gen_graph provided for the Floyd Warshall project. I modified the function by changing seed to the random number generator. Originally, a constant was used. This results in identical pseudo-graphs for every execution of the algorithm. To improve the randomness, I seeded the random number generator with the current number of

microseconds. This is essentially an integer between 0 and 999,999 that updates very quickly. This served as a perfect seed because of the large domain and relative randomness.

Step 1: The Basic Algorithm

At first I implemented the iterative PageRank algorithm as simply as possible. The algorithm simply iterated through each node of the graph and computed its new PageRank value by examining the adjacency matrix and the previous PageRank values for the other nodes. Once all new values are computed, the new results are compared with the previous results. If every node's new value is equal to the old value (within a very small tolerance) the algorithm has found the steady-state scores and it terminates. Otherwise, it saves the new values and recurses.

To achieve parallelism, the for-loop over all of the nodes is parallelized using OpenMP. Since no iteration mutates the values in any data-structures accessed by other iterations, the parallelism is safe and effective.

Analysis

After implementing the first version of PageRank, I began by ensuring that it worked by running it on various random graphs.

After ensuring correctness I thought about ways to improve performance.

Vectorization

Since the algorithm made substantial use of for-loops, my first thought about improvement was to make better use of vectorization. However, when I examined the compiler optimization report I found that all of the innermost loops had been already been vectorized. The one exception was in the main function and it parsed the values of the command line arguments. Since the algorithm already had substantial vectorization, I decided to look for other ways to get performance enhancements.

VTune Analysis

In order to find the slowest parts of the algorithm, I turned to VTune. When examining the report I found that of the 19.2 seconds that the program was running, 18.8 seconds were spent in the “run_iteration” function.

VTune Excerpt:

```
Function                  Module                    CPU Time
------------------------  ------------------------  ---------------
run_iteration             pagerank.x                 18.837s
__kmp_wait_template       libiomp5.so                 1.951s
```

While the majority of the time was spent in that function, comparatively little time was spent waiting. So I examined the report for that function in particular.

VTune Excerpt for the function “run_iteration”:

```
Source Line      Source                                  CPU Time
---------------  --------------------------------------  ------------
29               int jDegree = 0;                          0.015s
30               for (int k=0; k<n; ++k) {                 0.176s
31                   jDegree += g(j,k);                   18.557s
32               }
33               sum += w[j]/(double)jDegree;              0.058s
```

The report indicated that almost all of the computation time was spent on that one inner loop. This code was responsible for computing the $L(v)$ component of PageRank.

Step 2: Precomputing Graph Properties

After observing that the loop computing the degree of each node, $L(v)$, was taking the longest amount of time. I decided to precompute the values. $L(v)$ is static for any node for a given graph. Therefore it makes sense to compute it only once and to save the result.

In this modified algorithm, the first step of PageRank is to compute an array of integers, degree. This array maintains the values of all the $L(v)$ and makes it easily and instantly accessible later in the algorithm.

The rest of the algorithm is nearly identical to the algorithm in step 1. However, this preprocessing modification resulted in a speed-up of about 40%. A detailed comparison can be found later in this document.

Analysis

In order to find ways to further improve the algorithm, I again looked for vectorization improvements. However, as with before the algorithm was already largely vectorized.

I then looked at the VTune reports for more detailed information. I found that the total run time had dropped from about 19 seconds to about 0.5 seconds.

VTune Excerpt:

```
Function                  Module                    CPU Time
------------------------  ------------------------  ---------------
__kmp_wait_template       libiomp5.so                 0.445s
__kmp_wait_template       libiomp5.so                 0.118s
run_iteration             pagerank.x                  0.070s
```

This report indicated that waiting and thread idling is now the largest factor in the algorithm's performance and is a likely candidate for improvement.

Step 3: Blocking

In an effort to minimize waiting times and communication delays I thought back to the other projects we worked on through the semester. PageRank, like many of these other projects, can be split into distinct sub-problems that are worked on independently and then merged back together.

After thinking about the problem and doing some research, I found a paper by Taher Haveliwala (http://ilpubs.stanford.edu:8090/386/1/1999-31.pdf) that discusses a technique for solving PageRank in this manner called Blocking.

Blocking is a process that involves partitioning the graph into chunks called blocks. Once blocks are formed, PageRank iterations are run on each block until the nodes of each block converge to a value. Once this occurs, the results are merged and the process repeats until the whole graph has converged.

By allowing each block to come to a steady-state before merging results, the algorithm aims to reduce the total number of iterations required. Additionally, since each block is spatially local in memory, the communication overhead is lessened.

I implemented this blocking algorithm. The paper was specifically focussed on analyzing the internet and suggested blocking by domain. However, since I am implementing it for a general graph I simply grouped adjacently numbered nodes.

Overall I found that the changes made some improvement. The algorithm ran an average of 15.3% faster. Additionally the number of iterations needed substantially decreased from an average of 8.2 to an average of 4.87. A more detailed analysis can be seen later in this document.

Step 4: Improved Blocks

The paper on blocking heavily emphasized the importance of blocks having mostly internal edges. When blocks have largely internal edges, their convergence is more significant which eventually results in fewer required iterations.

When considering the internet, there is substantial metadata to help create these largely connected blocks. However, my general graphs have little data to help the grouping.

However, in an attempt to improve performance, I implemented a simple grouping technique. I started by selecting a random node for each group and then I picked other nodes adjacent to the selected node to be in the group. I repeated this process until all nodes were in a group.

Once grouped, I ran the algorithm from step 3. Unfortunately, I found that this did not improve my results substantially. In general the execution times were very similar, but they did increase an average of 1.7%, but the speed up was not very reliable. Additionally, I had hoped that it would decrease the number of iterations needed, but the average number of iterations increased from 4.87 to 4.99.

Comparison

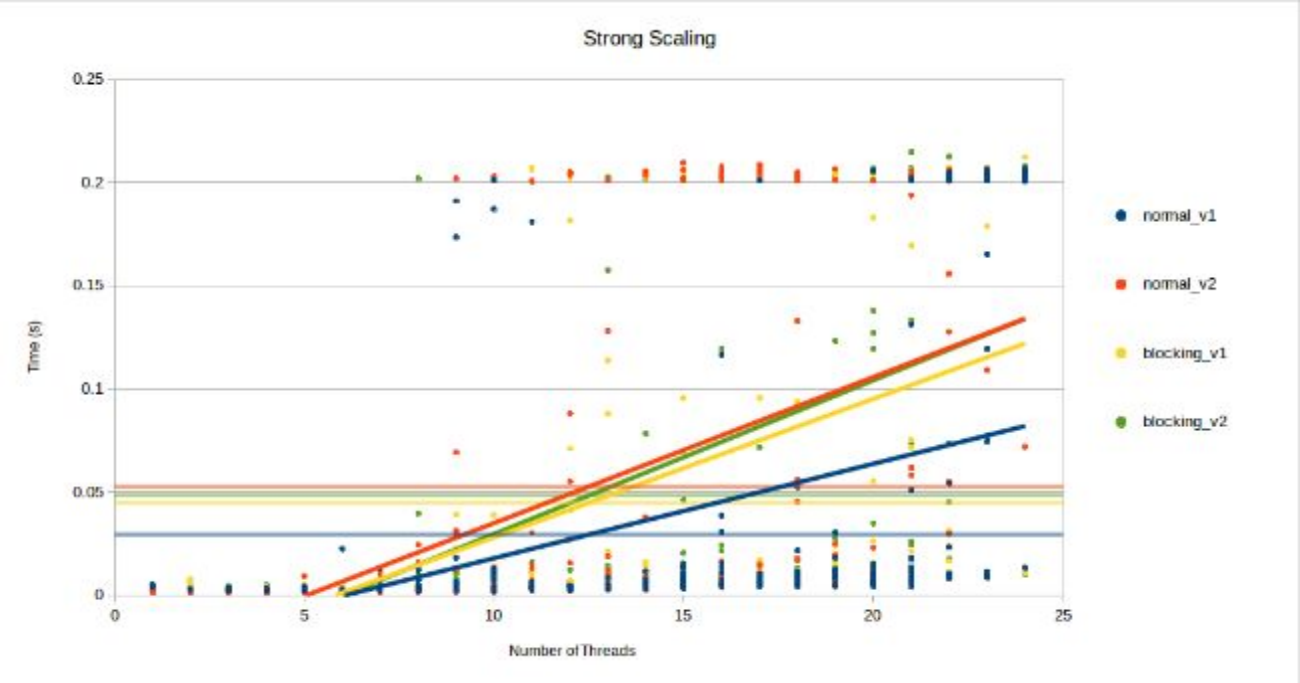
Overall I implemented and tested 4 different algorithms:

- normal_v1: the simplest algorithm possible from Step 1
- normal_v2: the algorithm that introduced graph preprocessing from Step 2
- blocking_v1: the algorithm that introduced blocking from Step 3
- blocking_v2: the algorithm that attempted smarter blocking from Step 4

Strong Scaling

To test strong scaling, I started by running all 4 algorithms on a 200 node graph. For each technique, I gradually turned up the parallelism by giving OpenMP access to additional threads.

Here is a graph of the results:

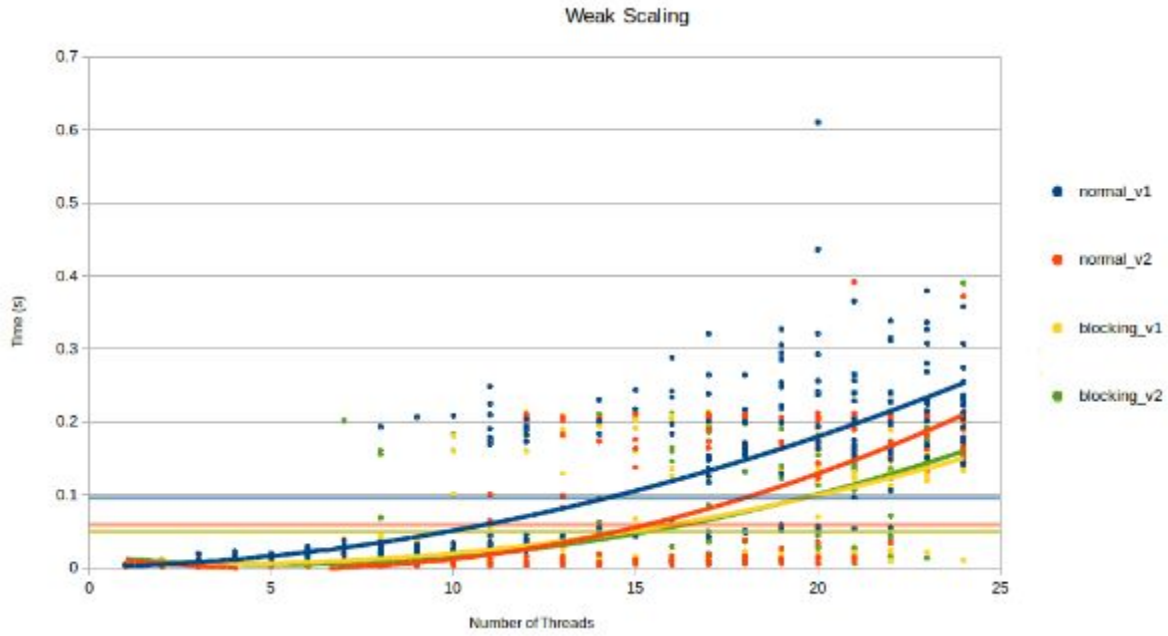


Unfortunately the data seems very erratic and contrary to earlier numerical analysis, it seems that the original algorithm performed best.

Weak Scaling

To test the weak scaling abilities of the algorithms, I ran each algorithm multiple times. Each time, the parallelism was increased, the size of the problem was also increased. Since the adjacency matrix grows with respect to the square of the number of nodes, the size of the graph was set to $\sqrt{(400n)}$ where n is the maximum number of threads. This ensures that amount of work in solving the problem grows linearly with respect to the number of threads.

Here is a graph of the results:



From this graph, it can be seen that as the problem size increases, the different algorithms perform differently. Adding the graph pre-processing and doing the first version of blocking both seem to make a substantial difference on the overall performance. However, attempting to group the blocks more intelligently seems to have made the performance slightly worse.

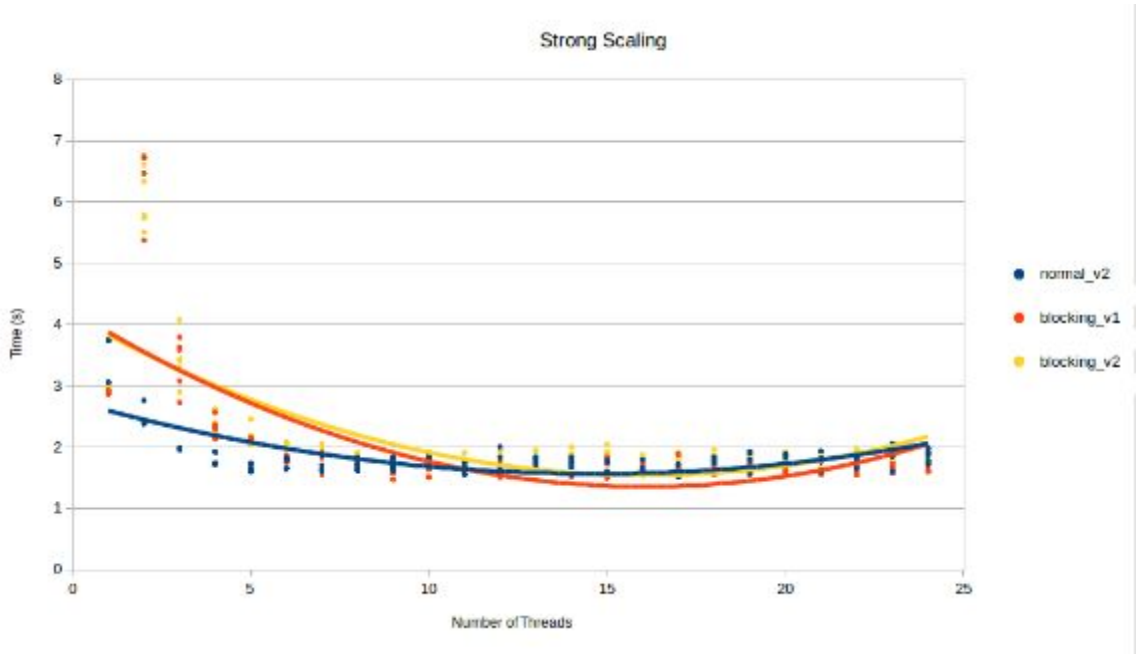
Large Problems

I hypothesize that the results in the previous two sections are so volatile because the problems are too small in size. To test this, I re-ran the tests with substantially larger graphs.

Strong Scaling

In this test, I used a graph that had 20,000 nodes in it. I then gradually increased the number of threads available. Unfortunately, the normal_v1 algorithm did not finish in a reasonable amount of time and was excluded from the graph.

Here are the results:

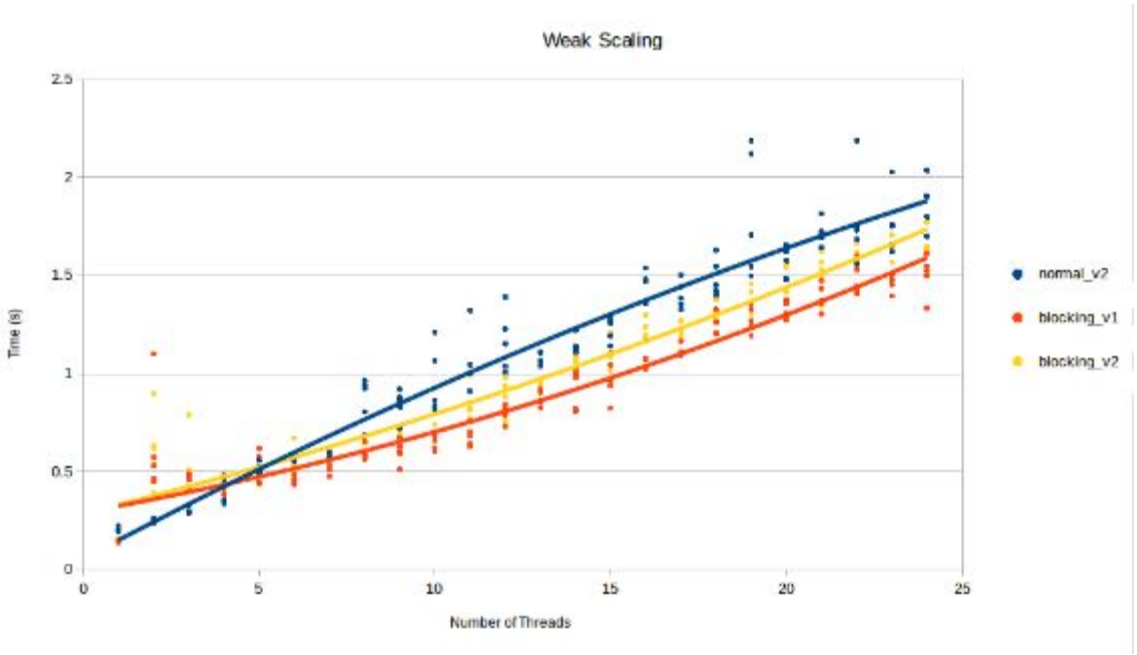


In this case, the 3 algorithms performed comparable. However, the blocking_v1 algorithm seemed to perform the best at higher thread counts.

Weak Scaling

The larger problem was also used for weak scaling. In this case, the graph has $\sqrt{4000000n}$ nodes in it where n is the number of threads available. Again, in this case the preliminary normal_v1 algorithm, was omitted.

Here is the graph:



This graph seems to agree with the previous one, that blocking_v1 performs best. Both of the blocking algorithms perform better than normal_v2, but overall the blocking seems to help.

Discussion

The transition from normal_v1 to normal_v2 that was made by preprocessing the graph seems to be a clear improvement. In almost every case, and certainly the average, the overhead of running the preprocessing seems to be more than made up in performance improvements.

The transition from normal_v2 to blocking_v1 also seems to be an overall improvement. While the results are not as dramatic as from normal_v1 to normal_v2, they seem to be positive overall. It seems that reducing the number of total iterations was definitely worth the additional overhead.

Unfortunately it seems that attempting to improve the blocks from blocking_v1 to blocking_v2 does not appear to have much of an impact. The overhead of doing the preprocessing is not worth the benefit. I hypothesize that this is because the graphs

being tested are generated by adding random edges. If the graphs were more similar to the structure of the internet and had larger connected components, the blocking could make a more substantial difference.

Additionally, the algorithm was fairly difficult to monitor. Unlike other problems, the running time for PageRank depends highly on the structure of the graph in addition to the size since the structure determines the number of iterations that are required.

Future Enhancements

There are two main areas that would be interesting to explore in the future:

- Better Block Grouping Algorithms
- Algebraic Solution Finding

Better Block Grouping

It is very difficult to figure out the best blocking scheme for an arbitrary graph. Ideally, better blocking results in fewer iterations and faster run times. However, finding ideal, or close to ideal, blockings is very complex. In this project I examined only 2 very simplistic approaches. Balancing the grouping overhead and payoff seems like it would be a very interesting extension.

Algebraic Solutions

In addition to iterative PageRank computation, scores can be computed in an alternative Algebraic manner. Since PageRank scores are steady-state parameters, they are very similar to eigenvalues. Therefore, it may be possible to compute PageRank by utilizing or modifying existing matrix libraries.