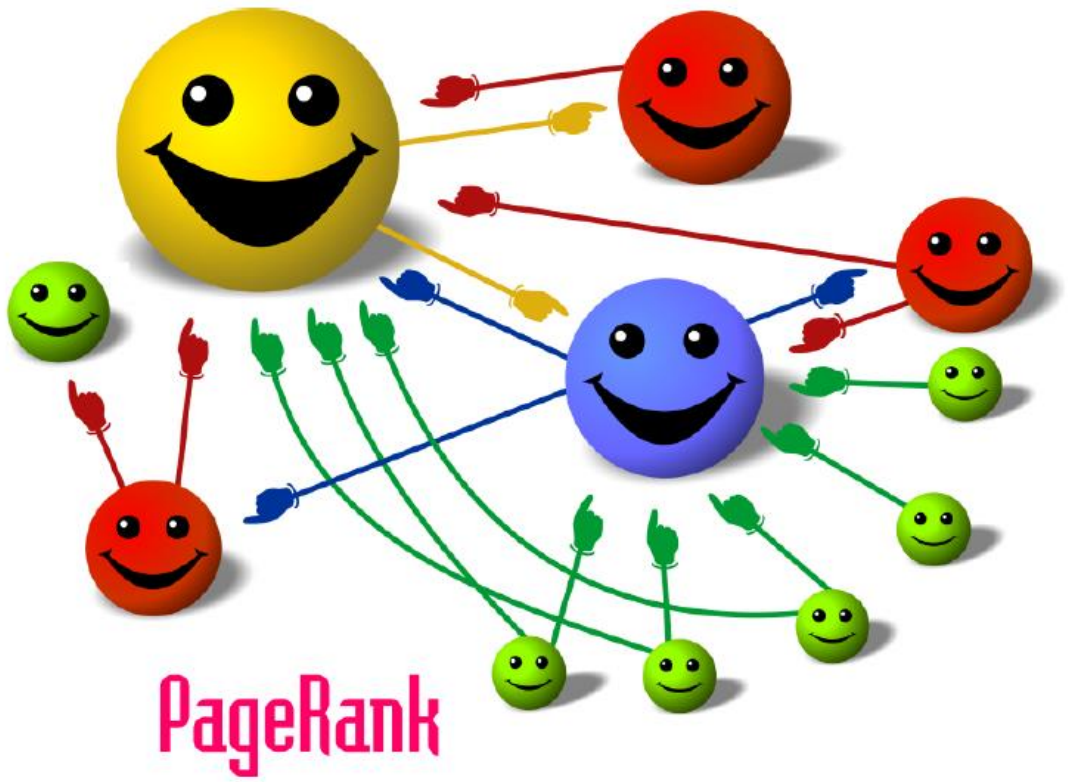
ECE5640 High Performance Computing
Preliminary Project Report

Student: Papadopoulos Charalampos
Topic: Parallelization of the Google Pagerank Algorithm on the Discovery Cluster



1. Project Description

In this project, the well-known Google Pagerank Algorithm is going to be implemented and parallelized on the Northeastern University Discovery Cluster. Initially, a serial version was implemented. After that, this version was parallelized using OpenMP, MPI. There is going to be a next version in CUDA.

The Google Pagerank Algorithm is an algorithm used by Google Search to rank websites in their search engine results, according to how many other websites provide a link to that website. More important websites are expected to be pointed by more other websites.

The algorithm used is

$$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)}$$

where $p_1, p_2, \ldots, p_N$ are the pages that we examine, $M(p_i)$ is the set of pages that have a link to p_i, $L(p_j)$ is the number of outbound links on page p_j and N is the total number of pages.

2. Source of the Code and GIT Repository

The code that I use here comes from here . There have been some modifications because the code on the site had bugs.

The git repository that I built is here . Currently, there are a serial version, an mpi version and an openmp version. There is soon going to be a CUDA version. Each folder has a Readme file with compile instructions.

3. Explanation of the serial algorithm

In order to better understand the structure of the code, we explain more the functionality of the algorithm in this section.

Websites may be represented as nodes in a graph. The edges represent the hyperlinks between the websites. So if there is an edge between two nodes, then there is a hyperlink between the relevant websites.

We represent each website (node) as a struct, as following:

```
typedef struct
{
  double p_t0;
  double p_t1;
  double e;
  int *To_id;
  int con_size;
}Node;
```

where:

- p_t0 is the probability of the node at the current moment
- p_t1 is the probability of the node at the next moment
- e is a constant used when this node is not connected to any other node, to avoid the “rank sink” problem
- *To_id is a pointer to a matrix of integers. This matrix contains the ids of the nodes to which this node is pointing.
- Con_size is the size of the above matrix. It is called connectivity of the node and it measures how many nodes this node points to.

The algorithm that we implement in c comes from the Matlab “PageRank by power method” that uses no matrix operations. This version was written my Mathworks employee Cleve Moler. The algorithm was translated to c by a student and was posted on github, where I took it from.

The pagerank.c file starts by reading the graph from a file. In my example the file is web-Google.txt and it comes from a Google programming contest. It can be obtained from https://snap.stanford.edu/ This graph has 916428 nodes. Reading this file with function Read_from_txt_file() updates the Nodes[] table and some initializations are done with Random_P_E() .

Following that, the main body of the iterative while loop of the algorithm starts. At each loop iteration, the current probabiblity p_t0 becomes equal to p_t1 and the new p_t1 of each Node is updated. This is mainly done in the double for loop that you can see in the algorithm. In fact this loop takes approximately 95% of the computation time of the algorithm. This loop, together with the for loop below implement the equation mentioned above as well:

$$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)}$$

where $p_1, p_2, \ldots, p_N$ are the pages that we examine, $M(p_i)$ is the set of pages that have a link to p_i, $L(p_j)$ is the number of outbound links on page p_j and N is the total number of pages.

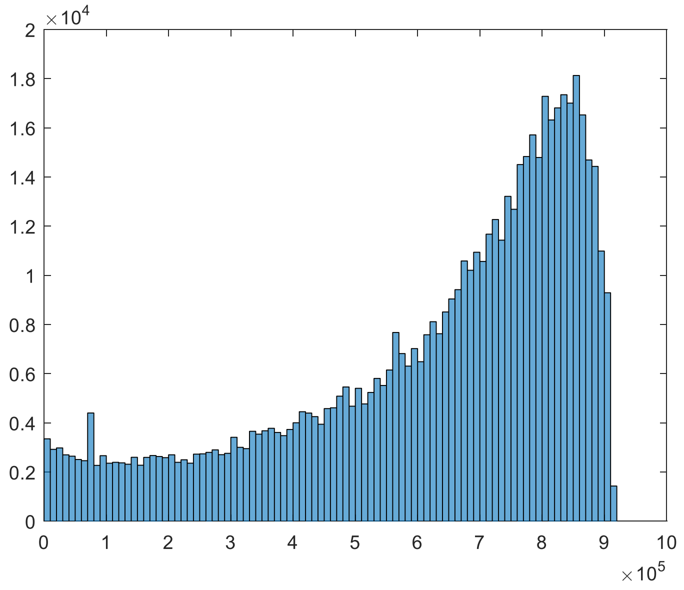
After that, there is a convergence test, which is carried out by checking the differences of the current probabilities p_t0 to the new probabilities p_t1. If all of them are below a threshold then the algorithm stops.

The execution time of the algorithm on discovery cluster is 1.21 seconds.

4. Bad spatial locality issue

A major issue with Pagerank algorithm is that it is not characterized by good spatial locality. In other words, the data that it reads/modifies are not located closed to each other in memory. That creates degradation in performance. Each thread in fact modifies the whole range of the memory used. So each thread modifies the whole range of the p_t1 matrix, which consumes around 8mb in memory.

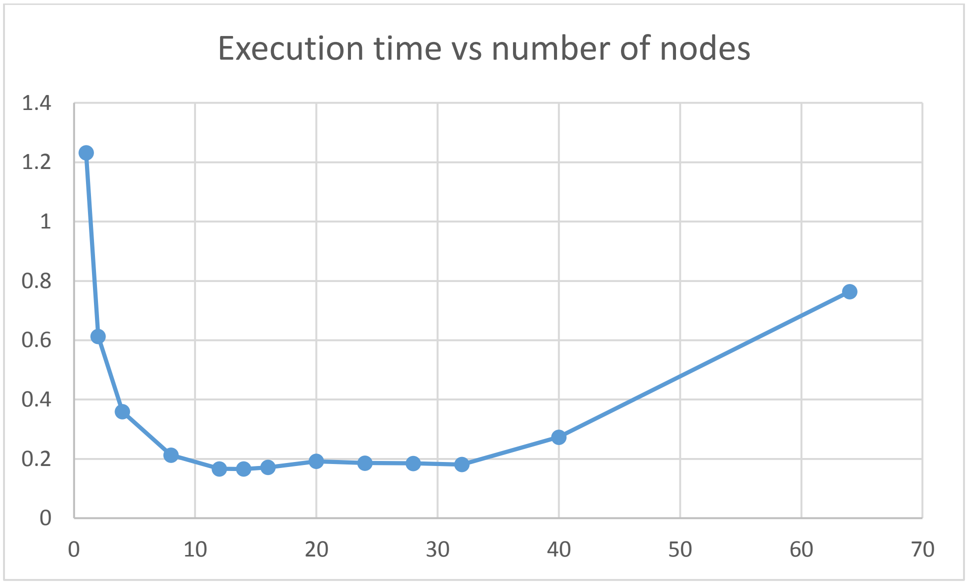
To show that, I computed the maximum minus minimum id of the Nodes that each Node points to. I created the histogram of these values which is shown below. The graph below shows that most of the Nodes point to other Nodes whose ids differ by around 800 000, while the total ids are 916428. We conclude that there indeed is no good spatial locality in this algorithm. All the threads modify the whole range of their p_t1 matrix, (around 8mb), in big steps.



5. First parallel version of the algorithm. OpenMP

As explained above, 95% of the execution time of the algorithm is spent in the double for loop. However, I used openmp pragmas to parallelize all the 3 for loops inside the while loop of the algorithm. Special attention was needed on what to declare as private in the pragma of the double for loop. The execution time versus the number of processes is presented below. Each execution was carried out 10 times and the mean value was computed. There seems to be an optimal approximately at 14 processes.

#processes	Execution time
1	1.232342
2	0.613277
4	0.359246
8	0.212763
12	0.166481
14	0.165796
16	0.171043
20	0.192035
24	0.185979
28	0.184676
32	0.181218
40	0.273122
64	0.764412



6. Second parallel version of the algorithm. MPI

In this parallel version of my code I experimented with MPI programming.

As described in the previous report (parallel version 1), 95% of the time is spent inside the double for loop that is seen at the code. There are two more for loops in the while loop. I experimented with parallelizing those with MPI but the overhead was more than the gain. Therefore, I present here an MPI version that only parallelizes the double (nested) for loop.

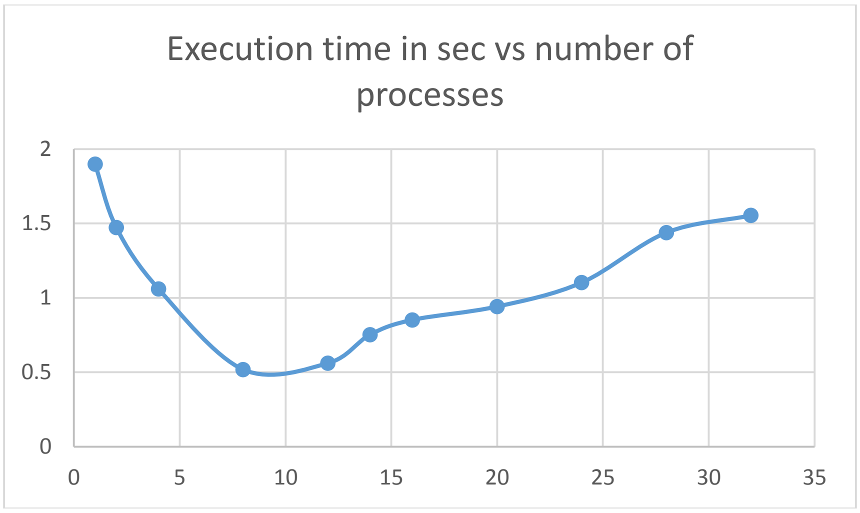
In order to parallelize the code, I assigned different Nodes to different processors. Each processor had to do computation related to a range of Nodes. The N nodes, where N is equal to 916428 for my example, are divided among the processors. At the end there are two reduction operations in order to collect the data and pass them to the master process to complete the algorithm. After the master process completes its tasks, it broadcast the correct p_t1 values to the rest of the processes and a new iteration begins.

The problem here was that my version of the pagerank algorithm is the "power" version, which uses only structs and does not use matrices. I did a trick to overcome that, I noticed that the p_t1 fields of the struct are the important ones. I converted the field p_t1 in the struct to a pointer and then allocated memory for an array and pointed all that pointers to that array. The name of the array is matrix_pt1. That way I was able to perform reductions on this matrix.

It has been validated that the code works correctly!

I measured the time of execution of the algorithm for several number of processes. Each execution was carried out 10 times and the mean value was computed. The results may be seen below. There seems to be an optimal for 8 processes.

1	1.896921
2	1.473117
4	1.059444
8	0.51771
12	0.560706
14	0.752775
16	0.851799
20	0.941125
24	1.102737
28	1.438129
32	1.553267



7. Profiling MPI

In order to understand the medium performance of MPI and see where the bottlenecks are, I profiled the code using the function MPI_Wtime() multiple times. The results are shown below:

1st for loop	0.0051	3.7%
double for loop	0.1274	91%
3rd for lop	0.0075	5.4%
total	0.1402	

Table for 1 thread

1st for loop	0.0187	19%
double for loop	0.0545	56%
2 reductions	0.0063	6.6%
3rd for lop	0.0112	11.6%
2 broadcasts	0.0053	5.5%
total	0.0961	

Table for 8 threads

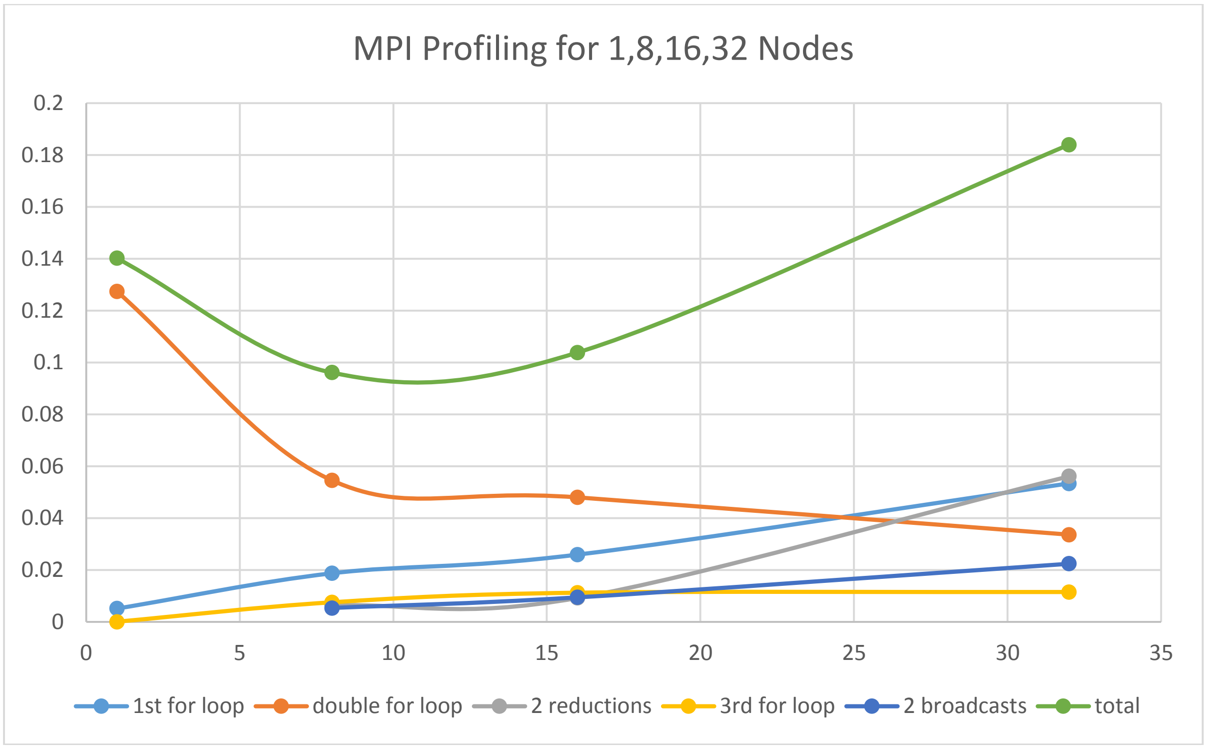
1st for loop	0.025884	25%
double for loop	0.047924	46%
2 reductions	0.009171	9%
3rd for lop	0.011474	11%
2 broadcasts	0.009388	9%
total	0.103841	

Table for 16 threads

1st for loop	0.053314	29%
double for loop	0.033565	18%
2 reductions	0.05608	30%
3rd for lop	0.018655	10%
2 broadcasts	0.022365	12%
total	0.183979	

Table for 32 threads

From the above tables, we have the following graph:



We observe that as the number of threads increases, the time impact of the reductions and the 1st for loop grow significantly and overpass the time impact of the double for loop, which we try to decrease.

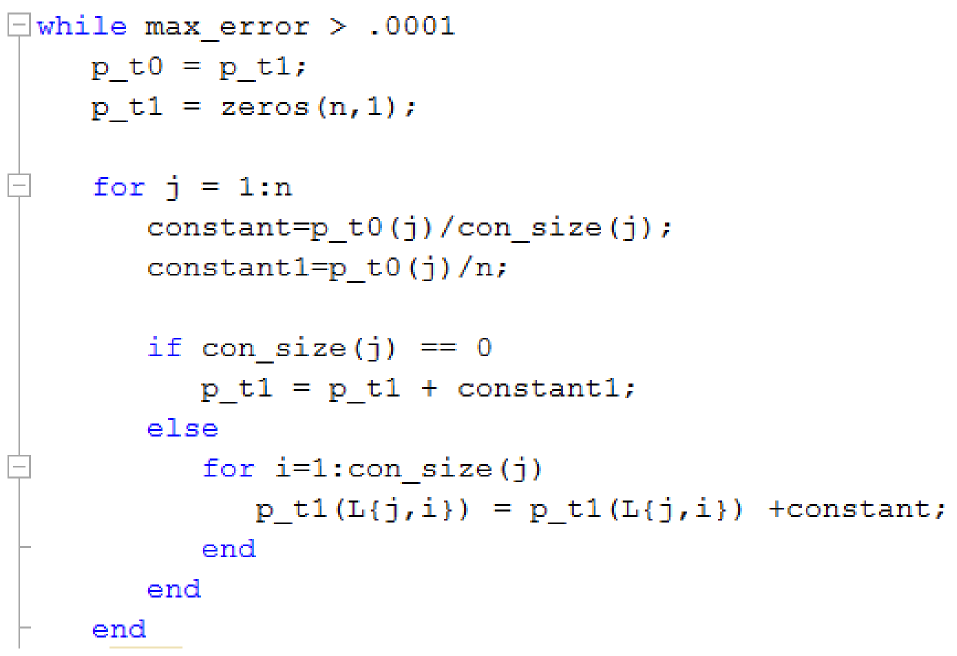
The cost of the reductions may not be decreased, since we need to add matrices of approximately 8 MB each and we cannot do anything about that. Furthermore, the cost of the 1st loop increases due to contention in memory and this may also not be reduced. It is av effect of the bad spatial locality that we described above.

Furthermore, the time cot of the 3rd for loop as well as the broadcasts increase, but it remains around half of the cost of the double for loop.

Unfortunately, as described above, the double for loop writes to a memory area of 8mb for each thread at a random manner, not showing spatial locality and therefore we cannot break the pt1 matrix in pieces to process it by different nodes. Therefore no further significant improved may be obtained with mpi.

8. Matlab implementation

I continued by implementing the code in Matlab. The basic loop without parallelization may be seen below:



```
while max_error > .0001
    p_t0 = p_t1;
    p_t1 = zeros(n,1);

    for j = 1:n
        constant=p_t0(j)/con_size(j);
        constant1=p_t0(j)/n;

        if con_size(j) == 0
            p_t1 = p_t1 + constant1;
        else
            for i=1:con_size(j)
                p_t1(L{j,i}) = p_t1(L{j,i}) +constant;
            end
        end
    end
```

The code seems more compact than the C version. The L{j,i} is the equivalent of the To_id matrix of the Node struct. However, L{j,i} contains the whole connectivity matrices for all the Nodes.

In the parallel version I used spmd, gplus, gop statements. However, I had to convert the cell array L{j,i} to L(j,i), otherwise the cell array took too much space in memory and I had memory issues. Furthermore, I faces special difficulties such as with the following line of code

constant=double(p_t0(j))/double(sizes(j));

where when I was doing it inside an spmd statement I needed to convert to double and needed to declare the sizes matrix inside the spmd statement, even though the values are the same for all threads. That needed several hours of debugging.

Also, unfortunately I had issues with the discovery cluster during the last days of the project (could not get an interactive node, couldn't get access to the license sever). **Therefore I only timed this code <u>on my own laptop</u>, where I only had up to 2 workers.** Results are shown below.

#workers	Time(sec)
1	1600
2	1402

The Matlab code together with a Readme containing instructions is on the github under 3rd parallel folder.

9. Comparison

We compare below the two parallel version in terms of execution time:

Version	**Execution time (sec)**
serial	1.21
openmp	0.17
mpi	0.51
matlab	1400

We now compare the two versions in terms of programming effort:

Version	**Programming effor**
serial	
openmp	low
mpi	medium / high
matlab	Medium/high

Given the above results, it is definitely not worth spending the extra programming effort needed to implement the mpi version. The openmp version, just by adding a few lines of code, performs significantly better. The MPI version took me several days to implement, while the openmp version less than a day.

The Matlab version was also not worth programming, apart from being able to understand the algorithm better due to immediate access to the variables in the workspace. Matlab' s performance compared to the other two version was very slow and the programming effort was significant.

8. Sources

1. blackboard.neu.edu
2. https://github.com/nikos912000/parallel-pagerank
3. http://www.cse.buffalo.edu/faculty/miller/Courses/CSE633/Li-Fall-2012-CSE633.pdf
4. https://en.wikipedia.org/wiki/PageRank
5. http://www.mathworks.com/moler/exm/chapters/pagerank.pdf
6. http://arxiv.org/pdf/1208.3071v2.pdf
7. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1488928&tag=1
8. http://delab.csd.auth.gr/~dimitris/courses/ir_spring06/page_rank_computing/01531136.pdf