ECE 420 Parallel and Distributed Programming
Lab 4: Implementing PageRank with MPI

Winter 2024

1 Introduction

PageRank is the first algorithm used by Google search engine to evaluate the popularity of webpages and rank the results from the search query. The intuition of this algorithm is that if a webpage is popular, then many webpages will link to it. Since a popular webpage draws more visitors, it, in turn, contributes to the popularity of any webpages it links to. In a probabilistic view, the PageRank algorithm outputs a probability distribution representing the likelihood that an Internet surfer randomly clicking through links on the Internet will arrive at a particular webpage.

Let us consider a group of webpages as nodes in a graph. If an arbitrary webpage A links to an arbitrary webpage B, then there exists a directed edge from node A to node B. We refer to this edge as an *outgoing edge of node* A or an *incoming edge of node* B. In PageRank, the surfer starts by visiting a random node in the graph with equal probability. In each subsequent step, the surfer will arrive at node A, select a random outgoing edge of node A with equal probability, and traverse to node B through the selected edge.

We introduce here the **Iterative Approach** for solving the PageRank algorithm, where the solution is approximated through iterative updates. Suppose that there are N nodes in the graph. Let vector $\vec{r}(t) = (r_1(t), \ldots, r_i(t), \ldots, r_N(t))$ denote the PageRank score of all the nodes[1] in the t^{th} iteration, with $r_i(t)$ being

[1]In this manual, node indices start from 1.

the probability that the surfer is on node i. At the start, we would have

$$r_i(0) = \frac{1}{N} \quad i = 1, \ldots, N. \tag{1}$$

Then, according to the random surfing process described above, a recursive relationship between $\vec{r}(t)$ and $\vec{r}(t+1)$ is defined,

$$r_i(t+1) = \sum_{j \in D_i} \frac{r_j(t)}{l_j}, \quad i = 1, \ldots N, \tag{2}$$

where D_i denotes the set of nodes that have an outgoing edge leading to node i, and l_j denotes the total number of outgoing edges from node j.

Additionally, let us assume that the surfer, who is randomly clicking through the links, will eventually stop clicking at some point. In PageRank, this can be denoted by a damping factor $d < 1$, representing the probability that the surfer will continue to traverse to the next node at each step. Alternatively, $(1 - d)$ represents the probability that the surfer will stop surfing at each step. Therefore, instead of computing Equation (2), we usually incorporate the damping factor and obtain the following iterative updates:

$$r_i(t+1) = (1-d) \cdot \frac{1}{N} + d \cdot \sum_{j \in D_i} \frac{r_j(t)}{l_j}, \quad i = 1, \ldots, N. \tag{3}$$

In this lab, let us set $d = 0.85$.

Webpages that do not link to any other webpages are represented by nodes with no outgoing edges. These nodes are problematic since they introduce a zero denominator in $\frac{r_j(t)}{l_j}$. To solve this problem, nodes with no outgoing edges are artificially assigned outgoing edges to every node in the graph, including itself. All l_j and D_i terms are updated accordingly before starting the PageRank computation. (The provided "datatrim.c" in the development kit already does this.)

Computing $\vec{r}(t)$ thus entails starting with the initialization from Equation 1 and iteratively updating $\vec{r}(t)$ based on Equation 3. This process should terminate if the difference between $\vec{r}(t+1)$ and $\vec{r}(t)$ is sufficiently small. A good measure of the difference is the relative change of $\vec{r}(t)$ in each iteration over a certain norm. Therefore, we can terminate the algorithm once the following condition is reached:

$$\frac{\|\vec{r}(t+1) - \vec{r}(t)\|}{\|\vec{r}(t)\|} < \epsilon, \tag{4}$$

where ϵ is some predefined positive constant. In this lab, let us set $\epsilon = 1 \times 10^{-5}$.

2 Task and Requirements

Task: Implement a distributed version of the PageRank algorithm described in the previous section with MPI.

Implementation Requirements and Remarks:

1. Use the scripts in "datatrim.c" to generate the input data, a graph with directed links. Specifically, the input data should consist of two files: "data_input_link" and "data_input_meta".
2. Use the helper functions in "Lab4_IO.c" to load and save data. Note that the `node` structure is defined in "Lab4_IO.h".
3. Use the "serialtester" executable to verify your results. Instructions on using this test program are provided in the "ReadMe". Use the `double` data type when storing your PageRank scores to ensure high accuracy and compatibility with the test program.
4. You can use the incomplete serial implementation of PageRank, "main_template.c", as a basis to develop your distributed program. However, this is not mandatory.
5. Time measurements should be properly implemented. Specifically, it should only measure the time required to run the PageRank algorithm and the necessary distributed computing overhead. Graph loading and results saving time should not be included.
6. Do not pass any command line arguments to your program.
7. Your program must be able to run with both a single (master) VM and with a cluster of VMs, for different problem (graph) sizes and under different numbers of processes.
8. Your program needs to handle cases where the number of nodes in the graph is not perfectly divisible by the number of processes.
9. When running your program with a cluster of VMs, you can store an identical copy of "data_input_link" and "data_input_meta" in every VM of the cluster.

10. *Optimize* the execution time of your implementation as much as possible using the techniques learned in class.

Submission: One member of each team is required to submit a zip file to eClass before the submission deadline. While we do not enforce a naming convention for the submissions, a good template is "user**_lab4.zip" (where ** is the user number). The zip file should contain the following:

1. "Makefile": By executing the `make` command, the solution executable named "main" should be generated. Please do not use any optimization flags (i.e. no `-O3`).

2. Solution source files: You should include all source files necessary to compile your solution executable. Note that you do not need to include the "datatrim.c", "serialtester", or "web-Stanford.txt" files. Also, do not include any input/output data file or compiled executable.

3 Marking Guideline

3.1 Marking Session

Each group is required to present a short demo of their code within an appointed in-lab time-slot. Each demo consists of the following components:

1. **Demo:** Upload and compile your eClass submission and verification code (to be provided during the marking session) in your VM. Demonstrate that your code works with both a single (master) VM and with a cluster of VMs, for different problem (graph) sizes and under different numbers of processes, as specified by the marker (a TA or LI). This includes verifying the correctness of your results and comparing the PageRank computation time achieved relative to the optimal results prepared by the LI and TAs.

2. **Presentation:** Provide a short (1-minute) verbal explanation of the program design. Focus on the strategy implemented to minimize the execution time of the PageRank process.

3. **Group Response:** The marker will ask some questions to the group. Marks will be assigned to the group based on their collective response to these questions.

4. **Individual Response:** The marker will ask some questions to each group member. Marks will be assigned to each individual based on their individual response (without assistance from other group members) to these questions.

To expedite the marking process, please rehearse your demo beforehand. Be aware that:

1. You should practice the process of loading, unzipping, and compiling your eClass submission in the VM in an efficient manner.

2. The marking process is timed. While there should be ample time for the demo, the marker may cut you off if the demo runs over the time limit.

3. The questions are sampled from a pool, and may be different for different groups.

4. The contents of the questions asked by the marker may include, but are not limited to:

   (a) explaining observations from the results,

   (b) describing anticipated program performance under certain scenarios,

   (c) providing potential improvements on the existing solution to address specific scenarios,

   (d) describing and modifying certain components within the solution code,

   (e) explaining and applying concepts and techniques learned in the lecture that are relevant to this lab.

3.2 Marking Rubric

Successful code compilation:	1
Presentation of implemented distributed strategy:	1
Successful execution and correct results:	2
Runtime is competitive and comparable to the optimal results:	2
Group questions:	2
Individual questions:	2
Total:	**10**

A Appendix: Testing on a VM Cluster

To test your MPI program on the VM cluster, you need to follow the procedures below:

1. Log into the master VM of your cluster. In the home directory, there is a file named *hosts* that contains IP addresses of the one master and three slave VMs.

2. Copy (via `scp`) your executable file and any supporting files from your master VM to every slave VM. The directory path of these files should be the **same** for all master and slave VMs.

3. Launch your program in the master VM by running the following command.
   ```
   mpirun -np 4 -f ~/hosts ./main
   ```
   This will use all the VMs listed in *hosts* to run the "main" program with 4 processes.