CSCI-4974/6971: Assignment 2 (100 pts)

Web Graph Analysis

Due Date: Monday 29 Sept. 2016, 16:00

For this assignment, we're going to perform an analysis of several web crawls. Specifically, we will analyze:

1. The *bowtie* structure of several web crawls
2. How effective it is to use a *random surfer* to calculate PageRanks

To do so, we're going to implement a parallel strongly connected components algorithm and a parallelizable random surfer algorithm. For background material and reference, use the lecture slides and code along with:

- `https://en.wikipedia.org/wiki/Strongly_connected_component`
- `https://en.wikipedia.org/wiki/PageRank`
- `https://www.cs.cornell.edu/home/kleinber/networks-book/networks-book.pdf` Ch. 13 and 14.3
- `http://www.sandia.gov/ sjplimp/papers/jpdc05.pdf` Paper describing the Forward-Backward (DCSC as they call it) algorithm

Submit both code files to `slotag@rpi.edu` by the due day and time listed above. Put your responses to the short answer questions in comments in the code files.

1 Parallel Strongly Connected Components (50 pts)

To begin, first download the Assignment 2 code outline and datasets that will be used for this part:

- `hw02-scc.cpp`
- `google.graph`
- `stanford.graph`
- `berkstan.graph`

- `notredame.graph`

The code template contains empty functions that you'll fill in. The graph datasets are web graphs of various domains from the SNAP database. What we're trying to find is how closely these crawls match the *bowtie* structure of the Internet as a whole. To do so, we're going to implement a shared-memory parallel version of the Forward-Backward algorithm discussion in Lecture 5.

In the code outline, we will fill in the `do_connectivity_info()` function. What we want to do is use a single step of the Forward-Backward (FW-BW) algorithm to label vertices using the `conn` array as being in the `SCC_SET` (in the largest SCC), in the `IN_SET` (can reach the large SCC), in the `OUT_SET` (reachable from the large SCC), or `NOT_SET` (can not be reached from the large SCC following either incoming or outgoing edges).

A single step of the FW-BW background algorithm consists of two breadth-first searches from a given root (the root to use is given in the code file). The first search discovers all vertices reachable from that root by following outgoing edges. The second search discovers all vertices that can reach the root by following incoming edges. The overlap between these sets forms the largest SCC (the root I give you is known to be a part of this SCC for all of the graphs), while vertices not in this SCC but are reachable from the root form the `OUT_SET` and vertices not in the SCC but can reach the root form the `IN_SET`. The `output_connectivity_info()` function will output the sizes of these sets. Implement these searches using OpenMP parallelism as shown in class. You can run the code by supplying the graph file as an input argument. Answer the following questions in a comment in your submitted code:

1. What are the sizes out each of the sets (`IN`, `OUT`, `SCC`) for each of the graphs?
2. Discuss how closely do the relative sizes of these sets for these web graphs correlate with the observed structure of the Internet as a whole (compared to the most recent analysis of the 2012 Web Data Commons crawl we talked about)?
3. Suppose we wanted to find out how many vertices comprise the tendrils and tubes of these crawls, how might we go about doing that?
4. Did you implement your BFS using a *pushing* or *pulling* method? What might be a potential advantage of using a pulling method here? Hint: we only care about determining reachability and not distances or parent information.

This will be evaluated on:

- Correctness of output - 15 pts
- OpenMP parallelization using work queue with no race conditions - 15 pts
- Responses to questions - 5 pts each

2 Implementing a Random Surfer (50 pts)

To begin, first download the Assignment 2 code outline and datasets that will be used for this part:

- `hw02-surf.cpp`
- `cs-stanford.graph`

The cs-stanford graph is a crawl of the computer science domain at Stanford from the Koblenz Network Collection. What we're trying to do here verify that an iterative PageRank calculation can be considered equivalent to the *random surfer* model, and observe how effective it is to actually use a random surfer model to get these PageRank values.

Remember that the PageRanks of each vertex in a graph can be considered as probabilities that a random surfer is visiting those pages at a given moment in time. When a random surfer visits a page, they will either follow an out link (with probability `OUTLINK_PROB`, as named in the code) or randomly jump to another page in the graph (with probability (1 - `OUTLINK_PROB`). If they follow an out link, they will select with equal probability from one of the available outgoing links. If the random surfer reaches a page without any outgoing links, they'll randomly jump to another page in the graph.

What we want to do is explicitly implement and track a random surfer as they browse a supplied web graph. Track the number of times our random surfer visits each the pages in the graph in `visit_counts`, and use those counts to calculate each page's PageRank. As output you'll see how those PageRanks compare to those calculated by the iterative algorithm. First, implement a serial version of your random surfer in `get_pageranks_walk()`. Second, implement a version using MPI in `get_pageranks_walk_mpi()`. For the MPI version, you can assume that each MPI task has access to the entirety of the graph structure and all associated data (i.e., we don't need to partition and parallelize over specific task-owned vertices).

You can run the code by supplying the graph file as a first argument and number of total clicks for your random surfer to model as the second argument. After implementing these functions and running your code, answer the following questions in a comment block in your code file:

1. About how many *clicks* or page visits is required of the random surfer before the "Total error" is below 0.01? (relative to 50 iterations of the standard algorithm)
2. Based on observed computational performance, which approach (surfer vs. iterative algorithm) do you think is more efficient?
3. What about in terms of parallelism? Describe potential benefits and drawbacks of each approach (surfer vs. iterative algorithm) in a parallel setting.
4. Suppose we instead wanted to parallelize our random surfer code with OpenMP instead of MPI, how might we do so? What race conditions and usage of public/private variables would we need to account for?

This will be evaluated on:

- Correctness of output - 15 pts
- MPI parallelization - 15 pts
- Responses to questions - 5 pts each