CSCI-4974/6971: Assignment 1 (100 pts)

Parallel Strong Connectivity

Due Date: Thursday 28 Sep. 2017, 16:00 - before class

For this assignment, we're going to analyze the strong connectivity of several web crawls. To do so, we're going to implement and examine a parallel strongly connected components algorithm based on breadth-first search. For background material and reference, use:

- `https://en.wikipedia.org/wiki/Strongly_connected_component`
- `https://www.cs.cornell.edu/home/kleinber/networks-book/networks-book.pdf` – Ch. 13 - discussing the web structure
- `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 Implementation (80 points)

The forward-backward algorithm is given below. We've also gone over it in class. We've discussed breadth-first search on several occasions. The challenge here is going to modifying and implementing the algorithm to perform the strong connectivity decomposition on your own system.

To begin, first download the Assignment 1 code outline and datasets that will be used for this part:

- `hw01-scc.cpp`
- `google.graph`
- `stanford.graph`
- `berkstan.graph`
- `notredame.graph`

Algorithm 1 Forward-Backward Algorithm

```
1: procedure FW-BW(V)
2:     if V = ∅ then
3:         return
4:     Select a pivot u ∈ V
5:     D ← BFS(G(V, E(V)), u)                    ▷ Descendant set
6:     A ← BFS(G(V, E'(V)), u)                   ▷ Ancestor set
7:     R ← (V \ (P ∪ D)                          ▷ Remainder set
8:     S ← (P ∩ D)                               ▷ Assignment to global SCC set
9:     new task do FW-BW(D \ S)
10:    new task do FW-BW(P \ S)
11:    new task do FW-BW(R)
```

The code template contains empty functions that you'll fill in. The graph datasets are web crawls of various domains from the SNAP database (e.g., stanford is a crawl of the stanford web domain). We're going to implement a shared-memory parallel version of the Forward-Backward algorithm. You can compile the code and just use the supplied graph files as-is for an input argument. Search through the code for `TODO` to find where you'll need to edit. The main goal to set all labels in the global `scc` array such that any two vertices in the same SCC are labeled the same, and any two vertices in different SCCs are labeled differently.

In the code outline, you will fill in the `void fwbw(digraph* g)` function. What we want to do is use it recursive for the Forward-Backward (FW-BW) algorithm to label vertices using the global `scc` array. A single step of the FW-BW background algorithm consists of two breadth-first searches from some root (select the root however you wish). The first search (`bfs_forward(g, root)`) discovers all vertices reachable from that root by following out edges. The second search (`bfs_backward(g, root)`) discovers all vertices that can reach the root by following in edges. The overlap between these sets forms the largest SCC, while vertices not in this SCC but are reachable from the root form the `D_minus_SCC` set and vertices not in the SCC but can reach the root form the `A_minus_SCC` set. The `Remainder` set is all other vertices that are both not assigned to an SCC and where not visited on either search. You can create new graphs for recursion by calling e.g. `create_new_graph(g, A_minus_SCC)`. Check the code comments for further detail. Especially note the usage of the `map` array in the graph object for properly assigning values to `scc`.

For parallelization, implement these three combinations:

1. Task-based parallelization on the recursive calls using `#pragma omp task` – check the OpenMP tutorial sites for details if necessary.

2. Loop-based parallelization during the breadth-first searches as we've done in class.

3. Both task-based and loop-based – you'll need to call `omp_set_nested(1)` to enable nested parallelization

This will be evaluated on:

- Correctness of final scc output - 20 pts
- Correctness of search outputs - 20 pts
- Correctness of fwbw function and recursive calls - 20 pts
- Use of parallelization - 10 pts
- No race conditions - 10 pts

Some additional hints:

- There's an optional and simple-to-implement trimming procedure (`trim(digraph* g)`) you can call. It's described in the paper listed above. This might help accelerate your code and ease your debugging.
- The `count_components(g, scc)` function can be used to check your output. To aid in debugging, the expected final output for cs-stanford.graph is: `Num SCCs: 4391, Max SCC: 2759, Nontrivial SCCs: 184, Unassigned: 0`.
- Be careful with memory management. Since we're doing recursion, both the stack and un-freed allocated memory can quickly explode.

2 Questions to Answer (5 points each)

In addition to submitting your source code, also include responses to the following:

1. What are the speedups you observe for each of the parallelization methods for each graph? Give serial and parallel numbers. Try and explain these observations.
2. What are the output sizes of the max SCCs for each graph? How do the relative sizes compare to the internal as a whole (at least according to the figure on page 389 of the book linked above).
3. Look more in-depth at the large number of trivial (single-vertex) components on a single graph. Do most of these have no in-edges? no out-edges? Do they have both in-edges and out-edges? Try to explain your observation in the context of the fact that this is a crawl of a given domain.
4. Look again at the figure on page 389. Describe a simple way that we could determine the assignment of vertices to IN, SCC, and OUT. Also describe a way we could determine which vertices comprise the tendrils. Describe a way we could determine which vertices comprise the tubes.