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

Graph Connectivity with OpenMP and MPI

Due Date: Monday 19 Sept. 2016, 16:00

This assignment will you to the concepts introduced in the first three lectures:

1. Vertex-centric graph computations
2. Push vs. pull
3. OpenMP parallelism
4. MPI parallelism

We're going to be looking at the above by implementing a couple parallel graph connectivity algorithms. Graph connectivity is a basic but important concept in graph analysis. It is a way to determine which vertices have links to each other, or which parts of a network are disconnected to each other. We going to focus on graph connectivity in its simplest form on undirected networks. For reference, you can read:

- `https://en.wikipedia.org/wiki/Connectivity_(graph_theory)`
- `https://en.wikipedia.org/wiki/Connected_component_(graph_theory)`
- `https://www.cs.cornell.edu/home/kleinber/networks-book/networks-book.pdf` Section 2.2

1 Fill out CCI forms (10 pts)

For this class, we'll eventually be using the CCI clusters. If you have not already had an account previously created for you, you'll need to fill out and submit a CCI User Information Form and CCI User responsibility agreement. These are on the class website along with assignment 1 as "Form 1" and "Form 2":

- `CCI_User_Information_20140612.pdf`
- `CCI_User_Responsibility_Agreement_20140612.pdf`

Fill them out then email them to `accounts@ccni.rpi.edu`. If you have an account, this is not necessary.

2 Getting Started

First, we want to make sure the environment on your system is setup to write and compile parallel OpenMP and MPI code. It is highly recommended you work under a *nix variant, such as Linux or OS X. Windows is less well supported for the work we'll be doing in this class, future assignments will potentially use frameworks that won't run under Windows, the CCI systems (and most high performance clusters in general) use an *nix OS, and all your assignments will be tested and graded on Linux. There are Linux labs and shared systems (`linux.cs.rpi.edu`) available on campus. Use Windows at your own risk.

1. Make sure your compiler supports at least OpenMP 3.1. For GCC, you'll need version 4.7 or higher. A way to check is to successfully compile the final code from Lecture 2 (`lec02-bfs-complete.cpp`).
2. Install Open MPI or MPICH. Although any MPI variant (e.g. Intel) should be fine since the API will be consistent. Install the most recent version you can, but we won't be using any advanced features so this is less important than the OpenMP version.

3 Parallel Graph Connectivity Algorithms

We're going to implement two basic parallelizable graph connectivity algorithms. The first is based on breadth-first search, and the second is based on the *color propagation* technique. The BFS algorithm will use vertex-centric pushing, while color propagation will use pulling. For each algorithm, we'll implement serial, OpenMP, and MPI versions. The functions to implement are all listed in the code file on the website.

What each implementation aims to do is to assign some numeric label to all vertices in the graph. For vertices in the same component, this label should be identical. E.g., if there exists a path from vertex 5 to vertex 10, ccLabels[5] == ccLabels[10]; if a path doesn't exist, ccLabels[5] != ccLabels[10].

To begin, first download the Assignment 1 code template and graphical datasets:

- `hw01-intro.cpp`
- `google.graph`
- `random.graph`

The code template contains empty functions that you'll fill in. The Google graph is a web graph of Google's domain from the SNAP database and is representative of the connectivity structure of most real-world networks. The synthetic random graph is a just a randomly generated graph with a large number of small disconnected components.

3.1 Breadth-first Search (35 pts)

By using breadth-first search, a straightforward parallel connectivity decomposition algorithm can be implemented. We loop through all vertices in the graph, and when we come across a vertex that hasn't been assigned a component label yet (*ccLabel*), we begin a parallel BFS using that vertex as the root. For the root as well as all vertices discovered during that BFS, they will get labeled with the vertex identifier of the root (e.g. all vertices in the component with vertex 0 will all be labeled with a value of 0).

```
procedure BFS-CONNECTIVITY($G(V, E)$)
    for all $v \in V$ do
        $ccLabels[v] \leftarrow -1$
    for all $v \in V$ do
        if $ccLabels[v] = -1$ then
            $Q \leftarrow v$
            $ccLabels[v] \leftarrow v$
            while $Q \neq \varnothing$ do
                for all $u \in Q$ do                                ▷ in parallel
                    for all $\langle u, w \rangle \in E$ do
                        if $ccLabels[w] = -1$ then
                            $ccLabels[w] \leftarrow v$
                            $Q_{next} \leftarrow w$
                swap($Q, Q_{next}$)
                $Q_{next} \leftarrow \varnothing$
```

3.1.1 Serial (5 pts)

Implement the above algorithm first without parallelism. This should only require minor modification from from the BFS code we wrote during Lecture 2. Note we aren't tracking levels here, only the connectivity labels. This will be evaluated on:

- Correctness - 5 points

3.1.2 OpenMP (15 pts)

Next, parallelize the primary *for loop* over the vertices in the queue using OpenMP. You don't need to implement the thread-owned queues as we did in Lecture 2, but you do need to handle any race conditions that might arise from updating the global queue. Remember to use atomics instead of critical sections. This will be evaluated on:

- Correctness - 5 pts
- Parallelizing the loop - 5 pts
- Handling race conditions (no `omp critical`!) - 5 pts

3.1.3 MPI (15 pts)

Finally, parallelize the algorithm using MPI. Implement communications between **2 tasks**. Assume that you have sufficient memory to store the full array of *ccLabels* on each task, but only have access to the adjacency information (`out_degree`, `out_vertices`) for a vertex owned by that task. Each task should own approximately half of the vertices. Use the code from class as your guide.

- Correctness - 5 pts
- Distributing work among tasks - 5 pts
- Updating queues with MPI calls - 5 pts

3.2 Color Propagation (35 pts)

Color propagation (also called min/max-label propagation, among many other names) is another connectivity algorithm. This algorithm can be more performant relative to the BFS algorithm above due to improved parallel efficiency, especially when using a *pulling* methodology, as we will do here. Color propagation works by initially coloring each vertex with its numeric vertex identifier and then iteratively propagating minimum numeric color values among neighbors. The end result is that every vertex in a given connected component all end up colored with the minimal vertex id appearing in that component (e.g. all vertices in the component with vertex 0 will all be colored with a value of 0).

```
procedure COLORPROP-CONNECTIVITY($G(V, E)$)
    for all $v \in V$ do
        $ccLabels[v] \leftarrow v$                                ▷ note different initialization
    $updates \leftarrow 1$
    while $updates > 0$ do
        $updates \leftarrow 0$
        for all $v \in V$ do                                       ▷ in parallel
            for all $\langle v, u \rangle \in E$ do
                if $ccLabels[v] > ccLabels[u]$ then
                    $ccLabels[v] \leftarrow ccLabels[u]$
                    $updates \leftarrow updates + 1$
```

3.2.1 Serial (5 pts)

Implement the above algorithm first without parallelism. This will be evaluated on:

- Correctness - 5 points

3.2.2 OpenMP (15 pts)

Next, parallelize the primary *for loop* over the vertices using OpenMP. Handle any race conditions that arise. Mitigate any race conditions by using either an OpenMP reduction or explicit private variables. For performance, avoid using any atomics or critical sections within the parallel loop. This will be evaluated on:

- Correctness - 5 pts
- Parallelizing the loop - 5 pts
- Handling race conditions (no `atomic` or `critical`!) - 5 pts

3.2.3 MPI (15 pts)

Finally, parallelize the algorithm using MPI by communicating updated color values assuming 2 tasks. This will be evaluated on:

- Correctness - 5 pts
- Distributing work among tasks - 5 pts
- Updating colors (`ccLabel`) with MPI calls - 5 pts

3.3 Short answer questions (20 pts)

Answer these questions within a comment block in your submitted `.cpp` file.

1. Which implementation runs fastest for each algorithm and each graph? Be sure to look at the timing while running with mpirun for the MPI code and without using mpirun for the OpenMP codes. Can you explain these results by looking at the output component info? - 5 pts
2. For graph `google.graph`, are vertices 0 and 643 in the same component? How did you determine your answer? - 5 pts
3. Suppose you wanted to turn you OpenMP color propagation implementation into a *pushing* algorithm.
   What modifications would be necessary? - 5 pts
   What race conditions might arise and how would you mitigate them? - 5 pts