CS 377P: Programming for Performance

Assignment 6: Parallel Bellman-Ford algorithm

Due date: April 25th, 2019

You can work independently or in groups of two.

Late submission policy: Submission can be at the most 2 days late. There will be a 10% penalty for each day after the due date (cumulative).

Clarifications to the assignment will be posted at the bottom of the page.

In this assignment, you will implement a parallel program to implement the Bellman-Ford algorithm for single-source shortest-path computation. You may use classes from the C++ STL and boost libraries if you wish.

Recall that the Bellman-Ford algorithm solves the single-source shortest path problem. It is a topology-driven algorithm, so it makes sweeps over all the nodes of the graph, terminating sweeps when node labels do not change in a sweep. In each sweep, it visits all the nodes of the graph, and at each node, it applies a push-style relaxation operator to update the labels of neighboring nodes.

In this assignment, you will implement sequential and parallel codes for the Bellman Ford algorithm in C++. In your implementation, you can use either pthreads or C++ threads.

One way to assign work to threads is to divide the nodes in the graph uniformly between threads. This will give good load balance for uniform-degree graphs but not for power-law graphs, but it is a start. Each thread should relax the edges of all nodes assigned to it. Decide if any thread updated any node and if so, continue iterating over the graph. Use a barrier to ensure that all threads have finished relaxation before deciding whether to perform another iteration.

Coding

- Implement a sequential program for Bellman-Ford and measure its running time on the input graphs given to you. These times will be the baseline for computing parallel speedups.
- The main complexity in a parallel Bellman-Ford implementation is ensuring that updates to node labels are done atomically. Implement these different forms of synchronization (different versions of the same algorithm):

    1. **Mutex on the graph:** Before relaxing any edge, acquire a lock on the graph. Release it after relaxation. This is coarse-grain locking.

2. **Mutex on each node:** Before relaxing any edge, acquire a lock on the destination node. Release it after relaxation. This is fine-grain locking.
3. **Spin-lock on each node:** For each edge to be relaxed, try acquiring a lock on the destination node. If it succeeds, relax the edge and release the lock. Otherwise, try relaxing it again.
4. **Compare and swap:** To relax an edge, perform an atomic update on the destination node using std::atomic::compare exchange weak() in C++11 standard atomics library (more information below).

- **Extra credit (25 points):** Assigning equal number of nodes to threads will result in poor load-balancing for power-law graphs. You can get better load-balancing for these graphs by assigning equal number of edges to threads although this complicates the algorithm a little. Implement this approach, using only the compare-and-swap approach for the relaxations.

Input graphs

Input graphs: use rmat15, rmat23, road-FLA and road-NY in DIMACS format. Note that we allow rmat graphs to have multiple edges for the same pair of nodes in this assignment.

You can use the graph files on orcrist machines from TA's directory **/u/rbchen/public_html/graphs** or go to https://www.cs.utexas.edu/~rbchen/ and download your own copy. (They are huge, 2GB in total!)

Source nodes: **node 1 for rmat graphs, node 140961 for road-NY, node 316607 for road-FL. These are the nodes with the highest degree. These are the DIMACS node numbers.**

The output for the SSSP algorithm should be produced as a text file containing one line for each node, specifying the number of the node and the label of that node. Please specify how your represent infinity in the README or report.

What to turn in

1) Find the running time of your serial code for each input graph.
2) Find the running times and speedups for 1,2,4,8,16 threads for rmat15 and road-FLA (baseline for speedup is the time for your serial code) using the four ways of implementing atomic updates discussed above. Plot these results. You can use different plots for the running times for different input graphs since the sizes and therefore the running times will be very different but use a single plot for the speedups. Based on these experiments, what is the best way to implement the atomic updates for Bellman-Ford? In the rest of the experiments, use only this implementation.
3) Find the running times and speedups for 1,2,4,8,16 threads for all four input graphs and plot them. You can use different plots for the running times for different input graphs since the sizes and therefore the running times will be very different but use a single plot for the speedups.

4) Do you observe good speedups for rmat-style graphs? How about road networks?
5) Extra credit implementation: if you did this part, repeat parts 2 and 3 with your implementation.

Submission

Submit (in canvas) your code and all the items listed in the experiments above.

Graph formats

Input graphs will be given to you in *DIMACS format*, which you used in assignment 2.

Compare-and-swap

Figure 3 shows a way to use C++11 atomic compare and swap to achieve the same functionality as a mutex. var is the variable whose value is of type double to be synchronized. It is declared as std::atomic<double>. The function call compare exchange weak(old, new, ..) on var compares its value with old. If it is equal, it sets new as its value and returns true. Otherwise, it copies its value to old and returns false. All this is done atomically.

```
double old, new, var;
...
new = ...;
acquire lock on var;
old = var;
if (condition(old,new))
  var = new;
release lock on var;
```

(a) Mutex

```
double old, new;
std::atomic<double> var;
...
new = ...;
old = var;
do
  if (condition(old,new))
    done = var.compare_exchange_weak(
        old, new,
        std::memory_order_acq_rel,
        std::memory_order_relaxed);
  else
    done = true;
while (!done);
```

(b) Compare and swap

Figure 3: Synchronization primitives