# CS 377P: Programming for Performance

## Assignment 6: Parallel pagerank algorithm

### Due date: 9 PM, April 22nd, 2020

**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.

Recall that in Assignment 4, you wrote a sequential program for push-style pagerank computation on graphs represented in DIMACS format on file and in Compressed Sparse Row (CSR) format in memory. We will provide you with a sequential program for this algorithm, which implements the following approach.

i) Topology-driven algorithm: the algorithm executes in rounds, and in each round, it applies a push-style algorithm to all vertices.
ii) Jacobi-style iteration: vertices have two labels: *current* and *next*. In each round, the *current* label is read and the *next* label is written.
iii) Convergence: pagerank iterations are terminated when no node changes its pagerank value by more than $10^{-4}$ between successive iterations.
iv) After convergence, the pagerank values of all vertices are scaled so that their sum is one.
v) The scaled pagerank values are sorted in descending order. If there is a tie, vertices are sorted by their node IDs in ascending order.

In this assignment, you will write parallel programs to implement the same algorithm. You may use classes from the C++ STL and boost libraries if you wish. In your implementation, you can use either pthreads or C++ threads.

# Coding

0. Run the sequential program given to you. Your parallel code should produce the same output (round-off errors might affect the output slightly but the answers from your parallel program should be roughly the same).

1. (60 points) Work assignment: One way to assign work to threads is to divide the vertices 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 process the outgoing edges of all vertices assigned to it.

The main complexity in a parallel push-style program is ensuring that updates to vertex labels are done atomically. Implement these different forms of synchronization (different versions of the same algorithm):

1. **Mutex on each node:** Before relaxing any edge, acquire a lock on the destination node. Release it after relaxation. This is fine-grain locking.
2. **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.
3. **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).

   In the versions that use locks, you will need to implement a preprocessing step in which you allocate a lock for each vertex. Therefore, each vertex will have an *current* pagerank value, a *next* pagerank value, and a *lock*. Use an array of structures to represent all the vertex labels to get good locality.

   You should also parallelize the loop that checks convergence after each sweep since it is executed in each round.

   In your studies, you should time only the main computation loop. You can ignore any preprocessing step for timing.

2. (40 points) Better work assignment: Assigning equal number of vertices 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. In this approach, you will need to find the source vertex for each edge. Do not precompute the source vertex for each edge because that will require a lot of storage. Each thread should perform binary search to determine the range of source vertices for the edges assigned to it, and then iterate over outgoing edges of each of these vertices during the computation (you may need to process a subset of the edges for the first and last vertices in the range). It is fine for threads to precompute and store the range of vertices for the edges assigned to them.

   Implement this approach, using only the compare-and-swap approach for synchronization.

# Input graphs

Input graphs: use rmat15, rmat23, and road-NY in DIMACS format. Note that rmat23 is pretty huge, more than 2GB. You should be able to read the graph files directly on orcrist machines from TA's directory **/u/rbchen/public_html/graphs/**

# What to turn in

1) Report the running time of the serial code for each input graph.
2) Write a short description of your implementations to help us understand your programs.
3) Report the running times and speedups for 1,2,4,8,16 threads for rmat15, rmat23 and road-NY (baselines for speedup are the times for your serial code) using the three ways of implementing atomic updates discussed above. Graph the running times and speedups for each input graph as a function of the number of threads. Based on these experiments, what is the best way to implement the atomic updates for pagerank?
4) Repeat part 2 with the edge-based assignment of work to threads. Do you get better performance?

# Submission

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

# 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