# CS 378: Programming for Performance
# Assignment 5: Parallel Bellman Ford
# Due: April 19th

April 20, 2015

**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).

Implement a parallel Bellman Ford algorithm in C++. Divide the nodes in the graph uniformly between threads. Each thread should relax the edges of all nodes assigned to it. Decide if any thread updated any node and if so, iterate. Use a barrier to ensure that all threads have finished relaxation before deciding to iterate. Relaxation and decision to iterate involve synchronization across threads. 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.
2. **Mutex on each node:** Before relaxing any edge, acquire a lock on the destination node. Release it after relaxation.
3. **Spin-lock on each node:** For each edge to be relaxed, try acquiring a lock on the destination node. If it succeeded, relax the edge and release the lock. Otherwise, try relaxing it later, after relaxing the other edges.
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). You can also implement a variant which tries relaxing the other edges if the compare and swap fails, before trying to relax this edge again (similar to spin-lock).

In your implementation:

- Use either pthreads or OpenMP or C++11 threads.
- Read a graph from a command line that is formatted in the DIMACS graph specification.

- Use the Compressed Row Storage (CRS) format (more information below) for storing the graph in memory.

For each form of synchronization:

- Compile your code in ICC with flags '-O3 -ipo -std=c++0x'.
- Submit your run to the job scheduler on Stampede at TACC - use the 'serial' queue.
- Use the DIMACS full USA road network (travel time) graph as input (choose the first node as the start node).
- Run your code on 1, 2, 4, 8, and 16 threads.
- Validate that your implementation produces the correct answer.
- Plot a graph of execution time vs. number of threads (strong scaling) and compare its performance with that of other forms of synchronization. Explain your results briefly.
  Note: Report the execution time of the algorithm, not of the entire program; do not include the time taken for reading a graph into memory in the execution time reported.

# Compressed Row Storage (CRS) format

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 0 & 6.3 & 0 & 7.7 & 0 & 8.8 \\ 0 & 0 & 1.1 & 0 & 0 & 0 \\ 0 & 0 & 2.9 & 0 & 3.7 & 2.9 \\ 9.0 & 0 & 0 & 1.1 & 4.5 & 0 \\ 1.1 & 0 & 2.9 & 3.7 & 0 & 1.1 \end{pmatrix}$$

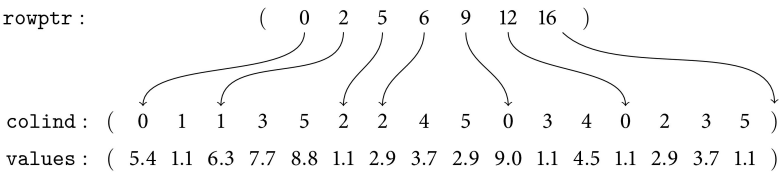


Figure 1: Example of a sparse matrix stored in CRS format

Compressed Row Storage (CRS) is a format to store a sparse matrix compactly. It only stores the non-zero values in the sparse matrix in a contiguous array, one row after another starting from the first (in order). In each row, the non-zero column values are stored one after another. Along with the values, it maintains the column index of each value. It maintains another contiguous array whose elements point to the first non-zero element of each row.

```
for (i=0; i<N; ++i)
  for (j=0; j<N; ++j)
    y[i] += A[i][j] * x[j];
```

(a) Dense matrix

```
for (i=0; i<N; ++i)
  for (j=rowptr[i]; j<rowptr[i+1]; ++j)
    y[i] += values[j] * x[colind[j]];
```

(b) Sparse matrix

Figure 2: Matrix Vector Multiply

Figure 1 shows a way in which a sparse matrix A is stored in CRS format. `rowptr` is an array in which each element points to the first non-zero element of that row. `colind` is an array which contains the column index of the non-zero values, one row after another. `values` is an array which contains the corresponding non-zero values, one row after another. To multiply the matrix with a vector, Figure 2 shows the difference between accessing the values stored in a CRS sparse matrix and those stored in a dense matrix.
Note: The column index and the values can be stored as distinct arrays or in a single array of a structure that contains both.

# Atomic compare and swap (C++11)

```
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

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.

## Notes

- Since the values you obtain will depend a lot on the machine and the input you use, you must use Stampede and the DIMACS full USA road network (travel time) graph for the numbers you report.
- You are being directed to write code in C++ because the atomic compare and swap is directly supported in C++11. The rest of the code can be written C-style without using C++ features. As long as your code is in a `.cpp` file, ICC will compile the code using its C++ compiler.
- Your implementation can use any standard C++ containers if you wish (it is not mandatory).
- Validation can be done by inspecting every node in the graph and ensuring that each neighbor is labeled no further than the node's label plus the edge length (in other words, running another iteration of Bellman-Ford should not change the shortest distance of any node). In addtion, you can verify that the output (the shortest distance to each node) of the parallel execution is same of that of the single-threaded execution.
- During development, you can use smaller DIMACS graphs like the New York travel time graph and the Florida travel time graph to validate your programs.

## Bonus points

You are welcome to experiment with and implement more optimizations (that improve strong scaling). Briefly describe them and report their performance in the report. Bonus points will be awarded based on the performance improvement of the optimizations.

## Deliverables

Submit (to canvas) source code of all your implementations and a report containing all plots and analysis. Include a Makefile to build your program and a README that contains instructions to verify your program. You will not be given any points if we are not able to run your code on the DIMACS graph and verify correctness.