CS 240A Assignment 4: Betweenness Centrality in Graphs

Assigned April 26, 2011

Due by 11:59 pm Monday, May 9

In this assignment you'll parallelize a sequential program that explores a sparse graph. The program computes a value called the "betweenness centrality" (or BC) for each vertex in the graph. The BC of a vertex is a measure of how "central" that vertex is in the graph, defined in the 1970's by social scientists studying graphs of relationships in societies.

The sequential algorithm that's your starting point is from the paper [Brandes], which contains a good description of BC. BC has also been used as a parallel graph benchmark, and implementations exist in several parallel languages. You will use Cilk to parallelize a sequential C implementation of the Brandes algorithm. All these references are linked to the course web site.

Getting good speedups on parallel graph algorithms is tricky. Your goal for this problem will be to run as fast as possible on as big a graph as possible; but don't be discouraged when the timings don't match the theory!

As usual, you'll do this in a team of two from different departments, and after its due you'll swap submissions with another team to write a review.

1 Betweenness centrality

A *path* in a graph is just a sequence of edges joining one vertex to another, and the *length* of the path is the number of edges. Our graphs will be *undirected*, meaning that an edge can be used in either direction as part of a path. (Betweenness centrality can be defined for directed graphs too; also, BC can be defined for *weighted* graphs in which each edge has a weight and the length of a path is the sum of the weights of its edges. But for this assignment we'll stick to unweighted, undirected graphs.)

If I'm a vertex, my betweenness centrality answers the question: How many of the shortest paths between other vertices go through me? To be precise, the betweenness centrality $C_B(v)$ of vertex v is defined as

$$C_B(v) = \sum_{s \neq v} \sum_{t \neq v} \frac{\sigma_{st}(v)}{\sigma_{st}}, \tag{1}$$

where σ_{st} is the number of shortest paths (not their length!) from vertex s to vertex t, and $\sigma_{st}(v)$ is the number of shortest paths from s to t that go through vertex v.

Here's a simple example. Consider the graph with four vertices a, b, c, and d, arranged in a cycle of four edges. The shortest paths from a to c are abc and adc. Therefore $\sigma_{ac} = 2$, and $\sigma_{ac}(b) = \sigma_{ac}(d) = 1$. The sum defining $C_B(b)$ has two nonzero contributions, of value 1/2 each: one from shortest paths from a to c and one from shortest paths from c to a; thus $C_B(b) = 1$. (Notice that $\sigma_{ab}(b) = 0$ by definition.) Since this four-vertex graph is completely symmetric, all

four vertices have the same C_B. You can verify this by running the sequential code in the harness on the web site with `./bc -s -g 2 2`.

For a slightly larger example, here's the output of `./bc -s -g 6`, which runs `bc` on a graph with 36 vertices arranged in a 6-by-6 grid:

```
  6.3   48.6   66.5   66.5   48.6    6.3
 48.6  133.0  173.7  173.7  133.0   48.6
 66.5  173.7  227.9  227.9  173.7   66.5
 66.5  173.7  227.9  227.9  173.7   66.5
 48.6  133.0  173.7  173.7  133.0   48.6
  6.3   48.6   66.5   66.5   48.6    6.3
```

The middle vertices are most "central", the corners least.

2 The sequential algorithm

The sequential algorithm is explained in [Brandes], and we won't repeat all of that here. There's also a cartoon of part of a sample run in the slides by Robinson linked to the web site.

The basic idea is to identify shortest paths by breadth-first search (or BFS). Performing a BFS from a starting vertex s gives the lengths of the shortest paths from s to every other vertex. With a little bit of extra bookkeeping, this BFS can also count shortest paths from s to every t, and can even keep track of how many of those paths go through every other vertex v. Actually, the search from s requires two sweeps over the graph: The first sweep is the BFS, which computes σ_{st} for every t and also records information about the "predecessors" of each vertex reached in the search; the second sweep goes through the BFS tree in reverse order, updating the centrality scores of each vertex using its predecessors.

The search starting from vertex s computes the contribution to $C_B(v)$ from the inner sum in Equation (1) for every v. The whole algorithm consists of doing the search from every possible starting vertex s, adding up the contributions as they're computed.

There are several places to introduce parallelism in the algorithm. Probably the simplest is to parallelize the outermost loop over s; that is, to do several breadth-first searches on different processors at the same time. A second approach is to parallelize the individual breadth-first searches, either by working on multiple nodes on a level simultaneously or by working on multiple neighbors of a single node. The fastest parallel benchmark codes for BC use more than one source of parallelism.

For this assignment, you should probably stick to one source of parallelism. We suggest the first one, parallelizing the outermost loop. Changing `for` to `cilk_for` is easy. The challenging part is to figure out which working data structures in the loop need to be duplicated in parallel and which don't; you probably only need one copy of the graph itself, for example. Also, you'll need to handle the data race that arises from multiple BFS's trying to update the C_B values at the same time. It is possible to create an array of reducers, though generally speaking you don't want to use more reducers than you can help.

3 Where's the data? The graph data structure

The graph data structure uses what's called "compressed sparse adjacency lists". This is a very compact structure that's almost exactly like the CSR sparse matrix data structure described in class. (An annoying difference is that we usually number the vertices of a graph from 0 to $n-1$ and the rows of a matrix from 1 to n.)

There are just two arrays for a CSR graph. Array `nbr[]` lists all the neighbors of vertex 0, followed by all the neighbors of vertex 1, then vertex 2, and so on. For an undirected graph, each edge is represented twice; if $\{v, w\}$ is an edge then v is listed as a neighbor of w and w is listed as a neighbor of v. Thus the number of entries in `nbr[]` is equal to the number of edges for a directed graph, or twice the number of edges for an undirected graph.

Array `firstnbr[]` gives the index in `nbr[]` of the first neighbor of each vertex. Thus, for example, the neighbors of vertex 5 are `nbr[firstnbr[5]]`, `nbr[firstnbr[5]+1]`, `nbr[firstnbr[5]+2]`, ..., `nbr[firstnbr[6]-1]`. If the graph has n vertices, the `firstnbr[]` array has $n+1$ elements; `nbr[firstnbr[n-1]]` is the first neighbor of the last vertex, and `nbr[firstnbr[n]-1]` is the last neighbor of the last vertex.

This data structure is not very flexible for a dynamic graph—it's hard to add or delete an edge or a vertex—but it's quite good for a graph that doesn't change, because it's easy to write an efficient loop over the all the neighbors of a given vertex.

The graph lives in shared memory, and it doesn't cause data races because it's only read, not written, in BC. The other data structures, which keep track of the BFS and the updates to the σ, δ, and centrality values, need to be parallelized with care. Different threads will need different copies of some of these data structures, but you want to make sure there are not n copies of them—that would use $O(n^2)$ memory in all, which is too much for a large graph. Rather, you should aim to use only $O(np)$ memory, where p is an upper bound on the number of cores you intend to use. Figuring out how to do this, and orchestrating it to be reasonably efficient, is the largest challenge of this assignment.

4 Test harnesses

The test harness on the course web site includes a main program `bc.c`, routines to generate grid and torus graphs, and utility routines to do things like read in a graph from an edge list and print out the graph data structure. It also includes a complete sequential implementation of BC. (Like most legacy codes, this one has been through several rounds of modification for various purposes by various people, and it isn't as clean as you might wish!)

The torus graph, which `bc` generates with the `-t` flag, is unusual in that the value $B_C(v)$ is the same for every vertex v. This makes it uninteresting as an example, but useful as a test case, and when you use the `-t` flag the harness will actually check to verify that the answer is correct.

The harness code reports both elapsed time and a score called "TEPS", which stands (more or less) for Edges Traversed Per Second. Just as the standard speed measure for high-performance numerical codes is FLOPS, or FLoating-point Operations Per Second, the standard speed measure for high-performance graph codes is TEPS.

5 What to implement

Implement `betweennessCentrality_parallel()` in Cilk. Your code should just plug into the existing harness, which you can use to generate and validate test cases.

We suggest you start with `betweennessCentrality_serial()`, and parallelize the outer loop over breadth-first searches as described above. However, you can take any approach you want—you can write your code from scratch, and you can explore parallelizing the BFS itself. Just make sure your routine has the same calling sequence so it will work from the harness.

Your timing results should be based just on the runtime of your betweenness centrality routine, not on the graph generators or validator. You don't have to parallelize anything in the harness besides the centrality routine, though you may if you wish.

6 What experiments to do

This is open-ended. Your goals are, first, to run BC on the largest possible graph; second, to get the largest possible TEPS score on that graph; and, third, to get the largest possible speedup or parallel efficiency.

The work in BC scales as $t_1 = O(n^2)$, since BC does a BFS from every vertex. This means that runs on very large graphs might take too long for your patience or your Triton account. Once you have debugged and verified your parallel code on graphs with, say, a few hundred vertices, it is alright to use partial BC runs to obtain TEPS scores and speedups on larger graphs. In the harness code, you can change the variable `numV` to something less than n—probably 100 or so is enough to get good measurements on any size graph, with as many cores as you'll be able to use.

The reason to emphasize the size of the graph is that many graph computations are so limited by communication volume that they get better TEPS scores on small graphs running on one or a few processors than on large graphs on many processors. This is in contrast to the numerical Top500 benchmark, where the best FLOPS scores generally come from the largest matrices.

Do the usual experiments with parallel efficiency and TEPS as a function of both graph size and number of cores. In parallel graph experiments, it is pretty much guaranteed that your performance results will not exactly follow your theoretical expectations. See if you can explain the behavior you observe, and the reasons for it.

7 References

[Brandes] U. Brandes, "A faster algorithm for betweenness centrality," J. Mathematical Sociology, vol. 25, no. 2, pp. 163177, 2001. A copy of this paper, and some other references, is linked to the course web site.