6.3 Paper Review

6.3.1 A Simple and Practical Linear-Work Parallel Algorithm for Connectivity

This paper describes the first parallel algorithm for identifying connected components in a graph that is work-efficient and can be reasonably implemented. It builds off literature that describes (a) parallel algorithms that are not theoretically work-efficient or (b) highly efficient theoretical algorithms that would be very challenging to practically implement. As such, the paper fits perfectly into the corpus of Algorithm Engineering, a positioning which is further reinforced by the blend of theory and experimental results presented in the work.

The idea underpinning the algorithm is quite elegant. It uses Miller et al.'s (β, d)-decomposition as a subroutine to partition the graph into small diameter clusters (any two vertices in the cluster are at most hop distance d away from each other). The number of edges connecting clusters is less than βm. In other words, β, a fraction between 0 and 1, tells us what proportion of edges we are willing to cut in order to create distinct clusters. A graph so decomposed is then collapsed into a smaller graph $G'(V', E')$ where each V' represents a cluster, and each E' represents one of the at most βm edges in between the clusters. The goal, to be accomplished through recursive applications of (β, d)-decomposition (DECOMP) to the graph, is to fully collapse the graph so that individual connected components are represented by only one super-vertex V''. The label L'' given to V'' is finally applied to all of the vertices that have been collapsed into that super-vertex.

An important practicality consideration that the authors make is using arbitrary decomposition, which uses rounded δ_v instead of fractional tie-breaking. Although it has the same asymptotic guarantees as regular DECOMP, the expectation of cut edges doubles to $2\beta m$. Other optimizations include in-place/on-the-fly filtering of edges that will be carried forward during the BFS. Intra-component edges are no longer needed by the algorithm and therefore culled. Memory is also saved by using a single structure C for storing both the component ID and resolving conflicts. There is some inefficiency inherent in parallel BFS as multiple nodes might write to the same child node at the same time, leading to a conflict that is resolved at the next barrier. The barrier effectively maintains synchronicity and also allows intra-component edges to be dropped in the min-decomp version of the code. The arbitrary version of the decomposition algorithm avoids the synchronization barrier and only stores one integer per vertex (its component ID).

A last note on implementation is made with respect to hybrid BFS, which implements bottom-up search per Beamer et al. when the frontier is large. Although the connected cluster algorithm must inspect every edge in order to determine if it is inter- or intra-cluster, using cache-friendly read-based computation when the frontier is large does improve performance.

The algorithms were tested on six graphs (random 5-degree graph, two power-law degree distribution graphs of differing densities, a 3D-grid graph structured like a lattice, a straight line, and a social network graph com-Orkut). The performance of the serial versions of these algorithms varies depending on the graph they are applied to. In general, arbitrary decomposition which requires only one pass over the edges of the frontier wins out over regular decomposition. When the frontier grows large the hybrid algorithm is beneficial, as expected. Parallelizing the algorithms achieved up to a 13x speedup, and is robust across graph types.

One of the clear strengths of this paper is its novelty. It is the theoretical work-efficient algorithm with polylogarithmic-depth and an implementation of its kind. The paper also provides a simple, comprehensible version of the algorithm alongside some clever optimizations that improve its practical runtime. The results suggest, as might be expected, that this algorithm works much better on small-diameter graphs than on large-diameter graphs, and I wonder what its performance on more real datasets (e.g. Wikipedia, or the Flickr dataset we discussed last week) would be. Also, the choice of β is and remains a little arbitrary.

I think future work could lead in the direction of additional experiments with real-world graphs. A way that one might be able to build on this paper is finding some novel implementation (what real-world problems are there that would benefit from finding disconnected elements quickly? I imagine there are some interesting applications in finite element analysis, which suffers from numerical instability when there are disconnected mesh elements.)