5.3 Paper Review

5.3.1 Parallel Breadth-First Search on Distributed Memory Systems

This paper is generally motivated by the fact that parallel BFS and its implementations on modern computer architectures is understudied, particularly in applications that involve irregular graphs. RAM (random access machine) models for algorithm design neglect the computational overhead of synchronizing and passing data between multiple processors.

As BFS is a key fundamental algorithm for graph operations, particularly in parallel systems where the naturally serial DFS is not as eligible for parallelization, much work has been invested into improving the efficiency of BFS. There have been meaningful advances in designing BFS for specific hardware systems (e.g. the massively parallel MTA-2 system used by Bader and Madduri), as well as for distributed memory systems that require intelligent partitioning of the graph to avoid coherence traffic and duplicate work (Scarpazza et al.). The most effective of these methods, however, operate on graphs with constant degrees, which do not reflect the typical distribution of graphs found in the wild. Another outstanding challenge is all-to-all communication at each level in BFS in order to synchronize knowledge about newly found vertices amongst the processors, which is especially expensive on distributed systems.

The novel contributions of this paper focus on improving distributed-memory BFS on graphs with irregular degree distributions, and are twofold. First, the authors present a one-dimensional distributed adjacency array for representing graphs. Second, they offer an alternative 2-dimensional partitioning that can be used amongst multiple processes.

I thought the discussion about potential bottlenecks in section 4.2 was very interesting. I would have expected that the synchronization and copying of the new frontier stack FS (i.e. merging thread-local stacks at every level) would have induced greater overhead than 3% of execution time, and that cache coherence would have been a greater issue for updating the distance array. However, the barrier and memory fence ensure that the correct distance is written and propagated to all the cores, making these "benign races" on insertions (into per-thread new frontier stacks NS_i) indeed benign. In the 2D case, use of the sparse accumulator in forming $\bigcup A_{ij}(:, k)$ for all k where $f_i(k)$ exists (i.e. unioning neighbour columns of A for the frontier to build the new frontier) is also fascinating, since I would not have anticipated a dense vector of values to perform better than a method that works better than the multiway merge, which, given the previous readings about parallel algorithms, seems like it would have been the preferable option (more memory efficient, sorted output).

The following sections talk more about memory efficiency, particularly focusing on the distinction between local and network memory accesses. Since, in the distributed approach, the array size for distance array checks (which make up a large fraction of the runtime) are reduced by a factor of p processors, multithreading has meaningful performance benefits. Similar tenets hold for the 2D case, but the working sets are larger (p_c and p_r are both smaller than the total p).

The experimental studies seem to align with the requests that Johnson makes in A Theoretician's Guide to the Experimental Analysis of Algorithms, 2001, in that the hardware and experimental conditions are clearly described and likely replicable. A comparison is made to previous implementations in order to situate the work in the literature, and are open about failures (inability to compile on Cray machines). They also reveal unexpected findings, for instance that on some architectures, the 1D algorithm perform 1.5-1.8x as fast as the 2D (Franklin), but that when scaling up, the 2D performs faster due to reduced time in communication (Hopper).

If we go up to section 2.2, the authors state that prior work has primarily optimized by ensuring the parallelization of edge visit steps is load balanced, synchronization costs are mitigated, and that locality of memory references is improved. Although the authors are implementing a novel approach, they do pay attention to load balancing (randomly relabeling vertices), reducing synchronization overhead (the "benign races" as a mechanism for merging local thread frontier stacks), and improving locality of memory references (CSR for 1D case, DCSC for sparse 2D matrices).

It would be interesting to see what happens when higher-diameter graphs are run on this algorithm, and whether any of these methods can be further optimized to create hardware specific solutions.