Parallel k-Core Decomposition: Theory and Practice

Author(s): Youzhe Liu, Xiaojun Dong, Yan Gu, Yihan Sun

17.1 Review

This paper presents an algorithm for parallel k-core decomposition. k-core decomposition is a form of graph processing that identifies a sequence of subgraphs that have "coreness" κ no less than k. The goal is to identify κ for all the vertices, and is achieved by sequentially removing vertices with lower connectivity from the graph. For example, a 2-core decomposition may yield a subgraph whose vertices have a connectivity of two or more, but vertices with a connectivity of 1 are disallowed. Ordinarily, k-decomposition is performed serially by "peeling" away the lower-order vertices. Translating it into parallel has historically been challenging, because of the high synchronization overhead required, the establishment of race conditions if the k-values of all neighboring vertices are atomically decremented at the same time, and the high work intensity of parallel models.

The authors approach this problem in an interesting way–they begin with a theoretical analysis of the efficiency of existing parallel k-core decompositions, such as Julienne. By formalizing the work-efficiency of Julienne, they find that it is theoretically efficient and are able to use this framework to develop an implementation that is also more efficient in practice. This seems like a very nice demonstration of the strengths of algorithm engineering as an intellectual approach toward designing functional, efficient code. They show that their algorithm has $O(n+m)$ work.

The discussion is structured around the parts of the algorithm that cost the most time. These are primarily concentrated in extracting the frontier, peeling the frontier, and updating $\mathcal{A}$. Although theoretically true, the authors argue that it requires careful implementation to have this be the case in practice, especially for the peel function (since frontier update and $\mathcal{A}$ update can be efficiently performed by the Pack function.) They find that the online (i.e. induced degrees $\tilde{d}$ of neighbors are updated immediately during the peeling operation), asynchronous approach to peeling used by ParK and PKC is efficient and easily improved simply by maintaining an active frontier $\mathcal{A}$ rather than generating a new $\mathcal{F}$ each round. (I'm amused that the authors say that "unfortunately" the previous work does not retain an active set because this actually enables their own publication.) The primary bottleneck, which the authors seek to resolve, are race conditions or contentions where multiple vertices attempt to decrement the induced degree $\tilde{d}$ of the same high-degree neighbor.

As I was reading the paper I was wondering whether a buffered approach might work (i.e. you store all the decrements in a buffer and then update the vertices that pull from the buffer after the peeling step is completed or when the buffer is full). The authors take a more elegant approach by probabilistically sampling for the $\tilde{d}$ of high-degree vertices instead of allowing it to be calculated directly by atomic decrement. If a vertex is likely to approach the target $\tilde{d} == k$ for that round, which would place it in the new frontier to be peeled in round $k+1$, the sampling is dropped for the more accurate measurement. The sampler is reset periodically as $\tilde{d}$ approaches k. The authors introduce a safety factor to ensure that this does not happen (changing their algorithm from Monte Carlo to Las Vegas, i.e. with guaranteed correctness) that will reset the run with a higher tolerance for accurately sampling the vertices. They claim that this reset has never been necessary, validating the robustness of their theoretical guarantee.

The authors also achieve a dramatic improvement in practical performance (though work efficiency stays the same) by leveraging vertical granularity control and bucketing. Instead of neighboring vertices being added to the frontier when their $\tilde{d} == k$, they are added to the current vertex queue so that they can be processed in round. Additionally, the size of the queue is limited for load balancing. This approach is especially effective for sparse graphs, and demonstrates up to a 31.2x improvement. Bucketing is used to organize the elements in the active frontier by their $\tilde{d}$, up to $\tilde{d} > k + 16$, in which case the vertices land in the overflow bucket. This structure allows buckets to be automatically updated, so that the frontier is automatically exposed.

A comprehensive set of experiments are performed across a range of graphs and using both self-implemented and state-of-the-art sequential and parallel k-core decomposition algorithms as baselines. Although not always the fastest (existing algorithms perform very well on roads and k-NN, for example, because of the low-degree of all the vertices), the authors' algorithm generally outperforms alternatives, and vastly outperforms on social and web graphs which feature vertices with high connectivity or high regularity (e.g. GRID). This indicates that the authors correctly identified the bottlenecks in existing algorithms and have effectively ameliorated them.

It would be nice to see future work that ports this method into a distributed setting, or even to scale the testing up onto very large multicore systems (the authors use a 96-core machine). There is also not much discussion of the additional memory cost or required I/O operations incurred by the hierarchical bucketing structure. Additionally, I think the bucketing structure breaks down if the graph undergoes dynamic updates.