Bounded Asynchrony and Nested Parallelism for Scalable Graph Processing

Dissertation Summary

Adam Fidel
Texas A&M University
https://parasol.tamu.edu/people/afidel

Introduction

Processing large-scale graphs has increasingly become a critical component in a variety of fields, from scientific computing to social analytics. The size of graphs of interest are becoming explosively large, preventing them from fitting into the memory of a single-processor system and highlighting the need for fast and efficient methods to process such graphs. Because of this, there exists a clear need for distributed data structures and parallel algorithms to facilitate the processing of these large graphs. However, the irregular access pattern for graph workloads, coupled with complex graph structures, varying topology, and large data sizes makes efficiently executing parallel graph workloads challenging.

In this dissertation, we develop two broad techniques for improving the performance of graph traversals and general parallel graph algorithms:

1. **Bounded Asynchrony.** Increasing asynchrony in a bounded manner allows one to avoid costly global synchronization at scale, while still avoiding the penalty of unbounded asynchrony including redundant work. In addition, asynchronous processing enables a new family of approximate algorithms when applications are tolerant to a fixed amount of error.

2. **Nested parallelism.** Allowing to express graph algorithms in a naturally nested parallel manner enables us to fully exploit all of the available parallelism inherent in graph algorithms.

Using bounded asynchrony, we are able to scale a breadth-first search (BFS) workload to 98,304 cores, where traditional techniques stop scaling at smaller core counts. Through the use of nested parallelism, we are able to process scale-free graphs up to 1.6x faster and are able to fit graphs into memory that would otherwise overload the memory capacity of a node using traditional methods.

Bounded Asynchrony

Parallel graph algorithms are generally expressed in level-synchronous or asynchronous paradigms. Level-synchronous paradigms iteratively process vertices of a graph level by level. This model guarantees the current level's computation to have completed before starting the next one through the use of global synchronizations at the end of each level. Level-synchronous algorithms tend to perform well when the number of levels is small. The asynchronous paradigm replaces expensive global synchronizations with less expensive point-to-point fine-grained synchronizations, which potentially increases the available degree of parallelism and thus may perform better on graphs with many levels. However, asynchronous algorithms may sometimes perform redundant work. Choosing the right paradigm depends on the system, input graph, and algorithm. This implies different implementations and optimizations for algorithms, with no easy way to switch between them.

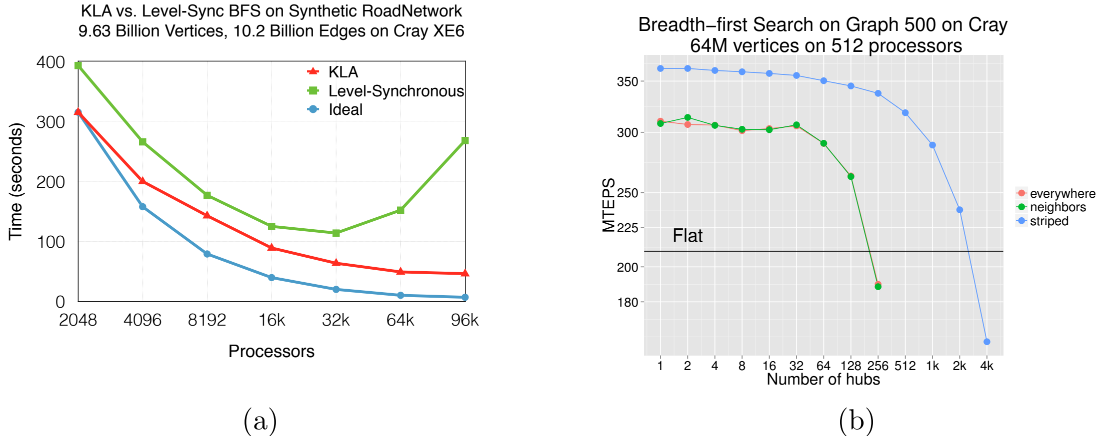

Figure 1: (a) Performance of KLA vs. level-sync BFS on synthetic road network to 98,304 cores on Hopper and (b) Improvement of using nested parallelism for BFS.

In our initial work, we introduced a new paradigm, k-level-asynchronous (KLA) [2], that allows parametric control of asynchrony ranging from completely asynchronous execution to partially asynchronous execution to level-synchronous execution. Partial asynchrony is achieved through the use of a global synchronization after a certain number (k) of asynchronous levels. In this respect, KLA may be viewed as a generalized BSP model where each superstep runs an asynchronous algorithm. A related approach is also used in the Δ-stepping single-source shortest path algorithm, which can be viewed as a special-case of KLA. This paradigm works in phases, similar to the level-synchronous paradigm. However, each phase is allowed to proceed asynchronously up to k steps by creating asynchronous tasks on active vertices.

Experimental Evaluation. We evaluate the improvement in performance and scalability of KLA by using breadth-first search as an example. We generated a synthetic road network by stitching together multiple copies of the European road network, which resulted in an input with 9.63 billion vertices and 10.2 billion edges. Figure 1(a) compares the scalability of KLA BFS with level-synchronous BFS on this road network. The level-synchronous BFS scales to 32,768 cores, but not beyond, while the KLA BFS is able to scale better until 98,304 cores (96k – the maximum available processor count) and yield faster running times due to better balancing the synchronization costs (which become increasingly expensive for large core counts) with wasted work.

Approximation Through Asynchrony

Computing shortest paths in networks is a fundamental operation that is useful for multiple reasons and many graph algorithms are built on top of shortest paths. For example, computing centrality metrics and network diameter relies on distance queries. For many large real-world graphs, computing exact shortest paths is prohibitively expensive and recent work explores efficient approximate algorithms for this problem. In unweighted graphs, an online distance query can be answered through the use of breadth-first search (BFS).

We introduce a novel approximate parallel breadth-first search algorithm [1] based on the k-level-asynchronous paradigm. A high amount of asynchrony in breadth-first search may lead to redundant work, as the lack of a global ordering could cause a vertex to receive many updates

with smaller distances until the true breadth-first distance is discovered. Each update to the vertex's state will trigger a propagation of its new distance to its neighbors, potentially leading to all reachable vertices being reprocessed many times and negating the benefit of asynchronous processing. Our novel algorithm controls the amount of redundant work performed by controlling how updates trigger propagation and allowing for vertices to contain some amount of error. In short, by only sending the improved values to neighbors if the change is large enough, we limit the amount of redundant work that occurs during execution. We modify the KLA breadth-first search algorithm by conditionally propagating improved values received from a neighbor update.

To this end, we introduce a new visitor (neighbor operator) for breadth-first search that allows for the correction of an error and repropagation of the corrected distance under certain conditions. We use tolerance $0 \leq \tau < 1$ to denote the amount of error a vertex will allow until it propagates a smaller distance. For a visit with current distance d and better distance d_{new}, we will propagate the new distance if $(d - d_{new})/d \geq \tau$.

The parameter τ controls the amount of tolerated error. This is a user-defined parameter that trades accuracy for performance in a KLA BFS. In applications that can tolerate a high degree of error, a large value of τ could result in signficant performance gains at the cost of accuracy. We prove an upper bound on the error as a function of degree of approximation τ and k.

Experimental Evaluation. In Figure 2, we evaluate both the execution time and error on the Texas road network from the SNAP collection on 32,768 cores on the IBM-BG/Q platform. This graph has 1.3 million vertices and 1.9 million edges. As expected, a lower value of τ results in slower execution time as more repropagations occur with lower tolerance. In the extreme case of $\tau = 0$, every message that contains a better distance is propagated and thus it is the same as the exact version of the algorithm. Figure 2(b) shows the mean error in distance $((d_k^\tau(v) - d_0(v))/d_0(v))$ as we vary the error tolerance τ. As expected, higher values of τ induce higher error in the result. At 32,768 cores, the approximate version is 2.27x faster with around 17% mean error.

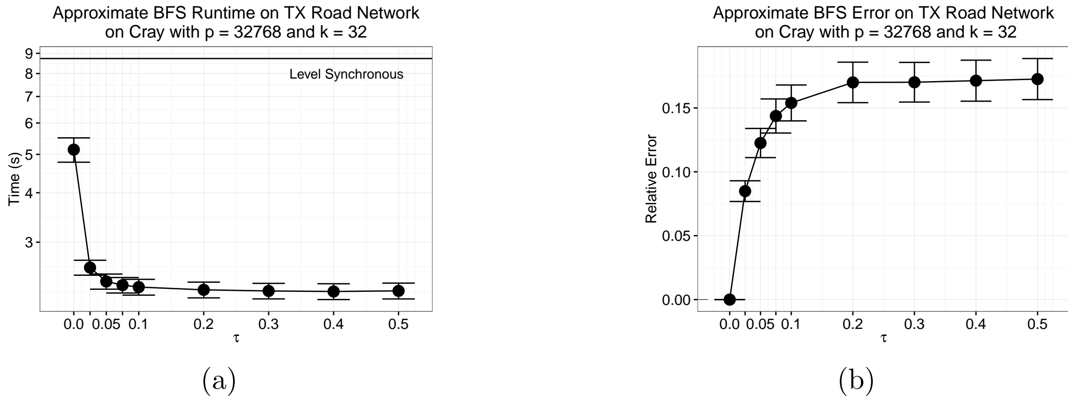

Figure 2: Approximate BFS on IBM-BG/Q evaluating sensitivity of (a) runtime and (b) error.

Nested Parallelism in Graph Processing

An important class of graphs are *scale-free* networks, where the vertex degree distribution follows a power-law. These graphs are known for their presence of *hub vertices* that have extremely high degrees and present challenges for parallel computations on these types of graphs. In the presence

of hub vertices, simple 1D partitioning (i.e., vertices distributed, edges stored sequentially with corresponding vertex) of scale-free networks presents challenges to keeping a balanced number of vertices and edges per processor, as the placement of a hub could overload any one processor.

In this dissertation, we propose a framework [3] to independently control the distribution of edges on a per-vertex basis, allowing the possibility to express orthogonal strategies for the various kinds of vertices in the graph. First, we represent the graph as a distributed array of vertices, with each vertex having a (possibly) distributed array of edges, partitioned in a fine-tuned manner. Using nested parallel constructs, we can define several strategies for distributing the edges of hub vertices, that can be interchanged without changing the graph algorithm itself. We present three initial strategies: partitioning a hub across all processors (`EVERYWHERE`), partitioning a hub only on the locations of the target vertices (`NEIGHBORS`), or partitioning a hub across shared-memory nodes (`STRIPED`).

Even though the distribution strategy of the edges changes dynamically, the expression of the graph algorithm itself remains unchanged. The nested parallel algorithm that executes over the edges is specified at the algorithmic level. We augment the existing k-level-asynchronous vertex-centric programming model to support execution on a graph whose edges may not be colocated with either its source or target vertex.

Experimental Evaluation. Figure 1(b) shows the benefit of using this nested parallel visitation strategy for the Graph 500 benchmark on 512 cores of the Cray platform. We observe that all strategies show improvement over flat processing for reasonable number of hubs. Specifically, we see more than a 60% speedup with a single hub using the `STRIPED` strategy.

Conclusion and Future Work

In this dissertation, we introduced two broad techniques for improving the performance and scalability of parallel graph algorithms: bounded asynchrony and nested parallelism. Through these techniques, we were able to show large performance improvements for highly irregular and dynamic graph workloads on large numbers of processors. For future work, we would like to explore further use cases of bounded asynchrony as it applies to streaming graph computations and heterogeneous computing environments with accelerators.

References

[1] A. Fidel, F. C. Sabido, C. Riedel, N. M. Amato, and L. Rauchwerger. Fast approximate distance queries in unweighted graphs using bounded asynchrony. In *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2016.

[2] Harshvardhan, A. Fidel, N. M. Amato, and L. Rauchwerger. KLA: A new algorithmic paradigm for parallel graph computations. In *Proc. Intern. Conf. Parallel Architecture and Compilation Techniques (PACT)*, PACT '14, pages 27–38, New York, NY, USA, 2014. ACM. Conference Best Paper Award.

[3] I. Papadopoulos, N. Thomas, A. Fidel, D. Hoxha, N. M. Amato, and L. Rauchwerger. Asynchronous nested parallelism for dynamic applications in distributed memory. In *Int. Workshop on Languages and Compilers for Parallel Computing (LCPC), in Lecture Notes in Computer Science (LNCS)*, Raleigh, NC, USA, September 2015.