Title

Essentials of Parallel Graph Analytics

Authors

Osama, Muhammad
Porumbescu, Serban D
Owens, John D

Publication Date

2022-03-17

Peer reviewed

Essentials of Parallel Graph Analytics

Muhammad Osama
University of California, Davis
mosama@ucdavis.edu

Serban D. Porumbescu
University of California, Davis
sdporumbescu@ucdavis.edu

John D. Owens
University of California, Davis
jowens@ucdavis.edu

***Abstract*—We identify the graph data structure, frontiers, operators, an iterative loop structure, and convergence conditions as essential components of graph analytics systems based on the native-graph approach. Using these essential components, we propose an abstraction that captures all the significant programming models within graph analytics, such as bulk-synchronous, asynchronous, shared-memory, message-passing, and push vs. pull traversals. Finally, we demonstrate the power of our abstraction with an elegant modern C++ implementation of single-source shortest path and its required components.**

***Index Terms*—parallel, graph analytics, graph traversal, algorithms**

I. Introduction

Graph analytics are used to study graphs and relationships between objects. Unfortunately, efficient analysis is increasingly difficult given increases in graph scale and workload diversity. Research in parallel graph analytics has tried to keep up by introducing new programming models, abstractions, and algorithms. Here, we explore one approach called **"native-graph" graph analytics** and make the following contributions:

1) We propose a framework, centered around an abstraction, that supports many design choices in each of the four pillars of graph analytics in the McCune et al. [1] survey paper *Thinking Like a Vertex (TLAV)*.
2) An example, using modern C++ concepts, highlighting design choices and exercising our proposed abstraction.

Our abstraction focuses on the graph as a "native" data structure, the active working set, and the ability to express iterative loops with convergence conditions to organize and schedule the computation and completion of a graph algorithm.

II. Overview

The native-graph approach for graph analytics focuses on operating directly on the graph's vertices or edges, or their relationships within the graph data structure, as opposed to linear-algebra-based graph analytics, which exploits the duality between graphs and sparse matrices to perform graph algorithms as sparse linear algebra operations. We focus on the native-graph approach and on implementing graph analytics with the different design choices described in TLAV [1] while using a general abstraction that targets many of these design choices. TLAV describes the four pillars of graph analytics as deeply connected, interesting design decisions to be made in a graph framework implementation targeting specific hardware architectures. The pillars and relevant models we target are:

1) **Timing**: Synchronous and asynchronous.
2) **Communication**: Shared-memory and message-passing.
3) **Execution Model**: Vertex programs and push vs. pull.
4) **Partitioning**: Heuristics.

We refer readers to the TLAV paper for an extensive discussion [1] of these design choices. Many graph libraries carve a slice of the TLAV design space to implement graph algorithms; for example, Gunrock's [2] and Pregel's [3] bulk-synchronous programming model on a shared-memory system, or PowerGraph's [4] asynchronous and message-passing system. These libraries focus on a small subset of possible design decisions and are limited by abstractions that are tightly coupled to those design decisions. Although these systems make useful design decisions, we desire greater flexibility. We show that with an abstraction focused around a native graph data structure, essential components of graph analytics can target many of the models within TLAV in a single graph framework. A general abstraction that is able to capture more than a single slice of these four pillars allows for more versatility when expressing graph algorithms and targeting different hardware architectures, and also leaves room for scalability in the future. This paper is our vision of such an abstraction and how it is implemented.

III. Abstraction for Each of the Four Pillars

Our goal is to build a graph framework able to support many (or more) of the design choices within TLAV. We first describe the essential components:

1) A **graph data structure** that expresses the underlying graph representation;
2) **Frontiers**, active sets of vertices or edges in each iteration of a graph algorithm;
3) **Operators**, programs operating on the graph data structure or the frontiers. Operators are often defined as traversals or transformations on the frontiers/graphs; and
4) **Loop structure/convergence condition(s)** to organize and schedule the computation and completion of a graph algorithm.

Before we show how our abstraction, built on the essential components above, targets the different programming models, we emphasize that these models are heavily interdependent, and describing how our abstraction targets them independently of each other will provide an incomplete understanding. Therefore, the following sections may cross-reference the models that are not strictly within the same "pillar". We

also summarize the models we capture and ignore within our abstraction and framework in Table I.

A. Synchrony and Asynchrony

In the execution domain, a timing model is often the building block for a graph framework. The choice of a bulk-synchronous programming model implies that the computation is performed in bulk *supersteps*, and a global synchronization barrier is used to synchronize the completion of each super-step. In contrast, asynchronous programming models have no explicitly defined barriers, and work is performed whenever the required resources are available [1]. In the TLAV survey paper, asynchronous execution models are defined to be more "complex" than bulk-synchronous execution models but allow for better workload balance.

The abstraction that targets the timing model in our framework is structuring operators with support for *execution policies*. We center our execution domain around operations on graphs or frontiers (an active vertex or edge set) [2], and how these operators are structured within an iterative loop. Since operators are loosely defined compute and memory transformations, our abstraction additionally allows them to be expressed with different *execution policies* as a parameter to control synchronization behavior and parallelism. Much like the C++ standard library's execution policies [5], these policies are unique types to allow for overloading of traversal and transformation operators to support parallelism and synchronization behaviors. Parallelism is supported by the work of an operator permitted to execute either in the invoking thread or in threads implicitly created by the operator's implementation. Synchronization behavior is supported by avoiding or introducing barriers on the invoking threads, based on the need of the timing model and the graph algorithm being implemented. These overloadings, now disambiguated with a unique type (*execution policies*), allow for the operator's functionality to be identical, even as its underlying execution changes.

B. Communication

The two conventional communication models are shared-memory and message-passing. In a shared-memory model, all data is directly available to all processes, whereas in a message-passing model, data is made available through messages passed between processes [1]. Expressing both models under the same framework can potentially allow for performance benefits in hierarchical distributed systems. **The abstraction that enables support for multiple communication models is the use of frontiers with multiple underlying representations, which individually support shared-memory and message-passing models.** When represented as an asynchronous queue [6], a frontier can communicate its elements using messages. When represented as a sparse vector or a dense bitmap [2] stored in shared memory, its elements are directly available to all processes. With thoughtful design, regardless of the underlying representation, the top-level interface to query the frontier (or presence of an active vertex or edge) remains the same.

The communication model also goes hand-in-hand with the timing defined in the previous section. In a shared-memory communication model, performing bulk-synchronous operations on the frontiers with global synchronizations is a common, effective practice. However, depending on the size and workload imbalance of a frontier, an asynchronous execution model with message-passing to communicate the active working set can be more efficient.

C. Vertex Programs and Push vs. Pull Traversals

The abstractions for supporting vertex or edge programs and traversal directions are exposed using three components: (1) the previously described operators, now with C++ lambda expressions [5] applied on the tuple {source and destination vertices, and their corresponding edge}, for every traversal or transformation (Section IV-C); (2) the frontier type, expressed as either a set of active vertices or a set of active edges, which allows for both edge and vertex-centric programs [2]; and (3) the graph data structure stored as the original representation and the transposed representation, the former for push traversals and the latter for pull traversals, at the cost of memory space.

D. Partitioning Schemes

In our framework, partitioning space is largely left unexplored and is work in progress. However, since parts of our graph abstraction allow for multiple underlying representations, partitioned graphs could also simply be expressed as another such representation. Inheriting a partitioned graph within our framework would imply that when the top-level graph data structure is queried, the APIs will need to support the use of the corresponding partitioned sub-graph to return the result of a query.

IV. Implementation

In this section, we show a vision of how to implement a graph framework using modern C++ that captures a wide range of models described in the TLAV survey paper. Due to space constraints, we will only show the implementation of one algorithm (single-source shortest path), implemented with a bulk-synchronous timing model and shared-memory communication scheme in mind. Although we show a very limited scope of TLAV's four pillars, we highlight where the abstraction extends to support asynchrony, message-passing, and push- and pull-based graph processing.

A. Graph representations

The duality of graphs and sparse matrices can be exploited even in the native-graph approach for graph analytics. The underlying graph data structure can be expressed using common sparse matrix formats such as compressed-sparse row (CSR), compressed-sparse column (CSC), or an adjacency list. Sparse-matrix formats make for great graph representations due to the sparsity inherent in many large graphs and their

TLAV Pillars	Models Captured	Abstraction	Mechanism	Models Ignored (not captured)
Timing	Bulk-Synchronous, Asynchronous	Operators, Loop structure	Execution policies	
Communication	Shared-Memory, Message Passing	Graph and Frontier Representations	Queue-based (messages) or bitmap, sparse frontiers	Active Messages
Execution Model	Vertex Programs, Push vs. Pull	Operators, Frontiers and Graph Representations	Vertex/edge-centric frontiers and compressed sparse row/column graph representations	
Partitioning	Heuristics (Mostly Unexplored)	Graph and Frontier Representations	Random partitioning, METIS [7]	Streaming, Vertex Cuts, Dynamic Repartitioning

TABLE I: Summary of what models are captured within the four pillars of TLAV [1] by our abstraction, and the corresponding element within the abstraction that implements the captured models.

ability to store such graphs in a compressed space. Listing 1 shows one example of a graph internally represented as a CSR matrix, but queried with a graph-focused API.

++Push vs. Pull Traversal

Our abstraction encompasses the ability to inherit and retain multiple underlying data structures for a single graph at the same time. We can leverage multiple representations profitably; for instance, storing both CSR and CSC graph representations enables traversal models that support both push and pull.

Listing 1 An example of requesting data from a sparse-matrix representation (CSR) with a native-graph API.

```
// Compressed-Sparse Row (CSR) matrix.
struct csr_t {
    int rows, cols;
    std::vector<int> row_offsets, column_indices;
    std::vector<float> values;
};
// In our framework, we rely on variadic inheritance
// to support multiple underlying data structures.
struct graph_t : public csr_t {
    // Get edge weight for a given edge.
    float get_edge_weight(int const& e) {
        return values[e];
    }
};
```

B. Frontier representations

Like graphs, frontiers can be represented with many different underlying representations. A sparse frontier can be simply represented as a vector of active vertex or edge indices. A dense frontier can be represented as a boolean array, where each element is true only if the corresponding vertex or edge is active. Depending on the scheduling and communication model, these frontier representations can be partitioned or be streamed to the compute units for processing. For a bulk-synchronous programming model with shared memory communication, the frontier can simply be stored in the shared memory (like the graph), and each element within the frontier can be processed in parallel. After each processing step, the synchronization step can be performed based on the operator's execution policy. In Listing 2, we show one example of how a sparse frontier can be implemented with a `std::vector` as its underlying data structure.

++Asynchrony and Message-Passing

Based on Chen et al, we experiment with an asynchronous queue as an underlying structure to represent the frontier to allow for asynchrony and message-passing [6].

Listing 2 Sparse frontier of active vertices represented as a simple vector from the C++ standard library.

```
struct frontier_t {
    // Underlying representation of a frontier.
    std::vector<int> active_vertices;
    // Get the number of active vertices.
    int size() {
        return active_vertices.size();
    }
    // Get the active vertex at a given index.
    int get_active_vertex(int const& i) {
        return active_vertices[i];
    }
    // Add a vertex to the frontier.
    void add_vertex(int const& v) {
        active_vertices.push_back(v);
    }
};
```

C. Parallel Operators

A high-performance graph analytics implementation relies on efficient parallel operators that transform, expand, or contract the frontiers or graphs. This is where the bulk of optimizations can be introduced, such as utilizing data parallelism and load balancing. In Listing 3 we show an example of how a parallel operator performing a traversal (frontier expansion) can be expressed using modern C++, implemented on a BSP model with data stored in the shared memory space.

++Timing Model and Parallelism

We extend our operators to support execution policies that overload the operator implementations to allow for parallel synchronous (`execution::par`) and asynchronous (`execution::par_nosync`) models.

Listing 3 Synchronous parallel neighbor-expand, an operator derived from a traditional textbook graph algorithm [8] implemented using modern C++. Note, due to the limitations in C++20's execution policies, the following version simply uses a parallel synchronous `std::for_each`; however, that can be replaced with parallel threads with no explicit barriers.

```
#include <algorithm>
#include <execution>
#include <mutex>
// Neighbor-expand implemented in C++20.
template<typename expand_cond_t, typename policy_t,
    std::enable_if_t<
        !std::is_same_v<policy_t,
        decltype(execution::par_nosync)>, int> = 0>
frontier_t neighbors_expand(
    policy_t execution_policy,
    graph_t& g, frontier_t& f,
    expand_cond_t condition) {
    std::mutex m;
    frontier_t output;
    auto expand = [=] (int const& v) {
        // For all edges of vertex v.
        for (auto e : g.get_edges(v)) {
             auto n = g.get_dest_vertex(e);
             auto w = g.get_edge_weight(e);
             // If expand condition is
             // true, add the neighbor into
             // the output frontier.
            if (condition(v, n, e, w)) {
                std::lock_guard<std::mutex>
                    guard(m);
                output.add_vertex(n);
            }
        }
    };
    // For all active vertices in the
    // frontier, process in parallel.
    std::for_each(execution_policy,
        f.active_vertices.begin(),
        f.active_vertices.end(),
        expand);
    // Synchronized here and return output.
    return output;
}
// An alternative asynchronous version (par_nosync
// is true) could launch parallel C++ threads and
// avoid the synchronization entirely.
template<typename expand_cond_t, typename policy_t,
    std::enable_if_t<
        std::is_same_v<policy_t,
        decltype(execution::par_nosync)>, int> = 0>
frontier_t neighbors_expand(/*...*/) {/*...*/}
```

D. Example: Parallel Native-Graph SSSP

To illustrate the abstraction, we show a simple example of a parallel SSSP algorithm implemented using the native-graph API, building on the previously described essential components. In Listing 4, we organize the algorithm in a BSP iterative while-loop with a convergence condition to schedule the completion. The key insights of the provided example are the use of (1) C++ lambda expressions to define a vertex program, (2) the `neighbor_expand` operator to perform the push-based traversal, and (3) `std::execution::par` to define the parallel execution policy.

V. A Look Ahead

As future work, we wish to explore many of TLAV's design decisions under a single framework targetting a wide-range of graph algorithms. Given the proposed abstraction, we make available "essentials", a graph library that targets GPUs: https://github.com/gunrock/essentials.

Listing 4 Parallel Single-Source Shortest Paths (SSSP) implemented in C++ using key components of native-graph graph analytics. Complete code: https://github.com/gunrock/essentials-cpp

```
std::vector<float> sssp(
    graph_t& g, int const& source) {
    // Initialize data.
    std::vector<float> dist(g.get_num_vertices(),
        std::numeric_limits<float>::max());
    dist[source] = 0;
    frontier_t f;
    f.add_vertex(source);
    while(f.size() != 0) {  // Main-loop.
        // Expand the frontier.
        f = neighbors_expand(
            std::execution::par, g, f,
            // User-defined condition for SSSP.
            [=](int const& src,          // source
                int const& dst,          // dest
                int const& edge,         // edge
                float const& weight) {   // weight
                float new_d = dist[src] + weight;
                // atomic::min atomically updates
                // the distances vector at dst with
                // the minimum of new_d or its
                // current value, then returns the
                // old value. (eq: mutex updates)
                float curr_d =
                    atomic::min(&dist[dst], new_d);
                return new_d < curr_d;
            });
    }
    return dist;
}
```

References

[1] R. R. McCune, T. Weninger, and G. Madey, "Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing," *ACM Comput. Surv.*, vol. 48, no. 2, pp. 25:1–25:39, Oct. 2015.

[2] Y. Wang, Y. Pan, A. Davidson, Y. Wu, C. Yang, L. Wang, M. Osama, C. Yuan, W. Liu, A. T. Riffel, and J. D. Owens, "Gunrock: GPU graph analytics," *ACM Transactions on Parallel Computing*, vol. 4, no. 1, pp. 3:1–3:49, Aug. 2017. [Online]. Available: http://escholarship.org/uc/item/9gj6r1dj

[3] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10, Jun. 2010, pp. 135–146.

[4] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed graph-parallel computation on natural graphs," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI '12. USENIX Association, Oct. 2012, pp. 17–30.

[5] "ISO international standard ISO/IEC 14882:2017(E) - programming language C++," International Organization for Standardization (ISO), Tech. Rep., 2017, https://isocpp.org/std/the-standard.

[6] Y. Chen, B. Brock, S. Porumbescu, A. Buluç, K. Yelick, and J. D. Owens, "Atos: A task-parallel GPU dynamic scheduling framework for dynamic irregular computations," *CoRR*, vol. abs/2112.00132, no. 2112.00132, Dec. 2021.

[7] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Sci. Comput.*, vol. 20, no. 1, pp. 359–392, Dec. 1998.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Second Edition*. The MIT Press, Sep. 2001.