HOOVER: Distributed, Flexible, and Scalable Streaming Graph Processing on OpenSHMEM

Max Grossman[1], Howard Pritchard[2],
Tony Curtis[3], and Vivek Sarkar[1]

[1] Rice University (`jmg3@rice.edu`)
[2] Los Alamos National Laboratory
[3] Stony Brook University

Abstract. Many problems can benefit from being phrased as a graph processing or graph analytics problem: infectious disease modeling, insider threat detection, fraud prevention, social network analyis, and more. These problems all share a common property: the relationships between entitites in these systems are crucial to understanding the overall behavior of the systems themselves. However, relations are rarely if ever static. As our ability to collect information on those relations improve (e.g. on financial transactions in fraud prevention), the value added by large-scale, high-performance, dynamic/streaming (rather than static) graph analysis becomes significant.
This paper introduces HOOVER, a distributed software framework for large-scale, dynamic graph modeling and analyis. HOOVER sits on top of OpenSHMEM, a PGAS programming system, and enables users to plug in application-specific logic while handling all runtime coordination of computation and communication. HOOVER has demonstrated scaling out to 24,576 cores, and is flexible enough to support a wide range of graph-based applications, including infectious disease modeling and anomaly detection.

1 Motivation

The value of graph analytics has grown over the past decade, as new applications arise in the areas of intrusion detection, infectious disease modeling, social networks, fraud prevention, and more. The value of graph analytics lies in the emphasis on analyzing relationships between elements of a system, rather than simply attributes of the elements themselves.

In many high-value applications of graph analytics, timeliness is key; while detecting a network intrusion one month after it occurs is still useful, detecting it as it occurs is much more so. As a result, focus is shifting from static graphs towards dynamic or streaming graph analyses.

However, with the growth in the use of streaming graph analysis has come a growth in the size and diversity of the graph datasets that graph analytics frameworks are applied to. Graphs have grown in scale, with increased numbers of vertices and edges. Graphs have also grown in complexity and imbalance, with widely varying densities and connectivity within a single graph. To support the continuation of these trends into the future, graph analysis frameworks will need to:

1. Support bringing to bear larger amounts of memory and compute.
2. Use sufficiently high level abstractions such that the framework's runtime can make automatic performance tuning decisions transparently, and so that user workloads can be mapped to new and exotic hardware.
3. Use sufficiently low level and flexible abstractions such that the framework does not overly restrict the problems that a user can express on top of it.
4. Demonstrate good scalability, such that adding memory and compute leads to an increase in the problem sizes that can be solved.

Without these properties, future graph datasets will be un-analyzable because of their size, or because of how long processing them requires.

In this paper, we introduce the HOOVER graph analysis and simulation framework. HOOVER is a general purpose, distributed, scalable, and flexible framework for (1) modeling systems that are naturally expressed as a graph, and (2) running analyses on the graph representation of that system. HOOVER is a framework for modeling dynamic graphs and supports addition and removal of vertices and edges in the graph, as well as updates to attributes on graph elements. This paper offers an overview of the problem scope of HOOVER, HOOVER's runtime, HOOVER's API, and uses two mini-apps to evaluate its scalability. HOOVER is available open source at `https://github.com/agrippa/hoover`.

2 Design

HOOVER is a C/C++ distributed framework for modeling and analyzing systems represented as streaming/dynamic graph problems. HOOVER emphasizes flexibility without sacrificing scalability, allowing users to plug in application-specific logic while:

1. Using OpenSHMEM [2] as a scalable backend for inter-PE communication.
2. Using communication-avoiding techniques to reduce inter-PE communication.
3. Being PGAS-by-design from the beginning, leveraging one-sided communication and de-coupled execution to reduce blocking and increase asynchrony.

HOOVER is, to some extent, specialized for a particular class of dynamic graph problems. The archetypical HOOVER problem follows this high level execution flow:

1. The application defines a large number of vertices partitioned across PEs, as well as callbacks to update the state of the graph. This information is passed to HOOVER.
2. The HOOVER framework begins iterative modeling of vertex behavior through repeated callbacks to user-level functions, evolving vertex and graph state over time. All PEs execute entirely de-coupled from each other. While each PE is asynchronously made aware of summaries of the state change in other PEs, no PE ever blocks on or performs two-sided communication with any other PE.

3. After some time, two or more PEs discover their state is related. This "relationship" is entirely user-directed and in the control of user callbacks. After this connectivity is discovered, those PEs enter lockstep execution with each other and share data on each iterative update to their local graph state. Multiple clusters of "coupled" PEs may evolve over time, with separate groups of PEs becoming interconnected or all PEs evolving into a single, massive cluster depending on application behavior.
4. Individual PEs may decide to leave the simulation at any time. A PE exiting a simulation does not imply a barrier; hence, all other PEs may continue in the simulation. PEs may also be configured with a maximum number of iterations to perform. Of course, this says nothing about process termination: all PEs would be expected to call `shmem_finalize` eventually.

An illustrative example may be useful: malware spread over Bluetooth. Malware propagation can be expressed as a graph problem, where vertices in the graph represent Bluetooth devices and edges represent direct connectivity between two devices. Malware propagation and analysis could be modeled on the HOOVER framework:

1. The application developer would define the actors in the simulation as vertices. Each actor would represent a device, and may include attributes such as the range of its Bluetooth hardware, the model of its Bluetooth hardware/-software, the speed at which it can move, or its initial infected/uninfected status.
2. HOOVER would then begin execution, updating device infection status, position, and connectivity with other devices based on user callbacks and other information passed in by the application developer. As iterations progress, more and more devices might become infected from a small initial seed of infected devices.
3. Eventually, two or more PEs may become coupled at the application developer's direction. For example, the developer might instruct two PEs to become coupled if a device resident on one PE infects a device resident on another. By entering coupled, lockstep execution those two PEs can now compute several joint metrics about the infectious cluster they collectively store, such as number of infected devices or rate of infection progression. Note that even when PEs create a tightly coupled cluster, they still interact as usual with any other PEs in the simulation which they are not coupled with.

2.1 OpenSHMEM

HOOVER is built on top of the PGAS OpenSHMEM programming model, and derives much of its scalability from being designed for the PGAS/OpenSHMEM execution model.

The SHMEM programming model was first created by Cray Research for the Cray* T3D machine and has subsequently been supported by a number of vendors across many platforms. The OpenSHMEM specification was created in an effort to improve the consistency of the library across implementations and, more importantly, to provide a forum for the user and vendor communities to discuss and adopt extensions to the SHMEM API.

The OpenSHMEM library provides a single program, multiple data (SPMD) execution model in which N instances of the program are executed in parallel. Each instance is referred to as a processing element (PE) and is identified by its integer ID in the range from 0 to $N-1$. PEs exchange information through one-sided *get* (read) and *put* (write) operations that access remotely accessible *symmetric objects*. Symmetric objects are objects that are present at all PEs and they are referenced using the local address to the given object. By default, all objects within the data segment of the application are exposed as symmetric; additional symmetric objects are allocated through OpenSHMEM API routines. OpenSHMEM's communication model is unordered by default. Point-to-point ordering is established through *fence* operations, remote completion is established through *quiet* operations, and global ordering is established through *barrier* operations.

3 HOOVER's API

This section describes the user-facing HOOVER data structures, concepts, and APIs to illustrate how an application developer interacts with HOOVER.

3.1 Vertex APIs

The core data structure of HOOVER is the graph vertex, represented by objects of type `hvr_vertex_t`. A graph vertex is represented as a sparse vector-like data structure.

Creating new vertices is accomplished with `hvr_vertex_create_n` (before or during the simulation). This will return initialized but empty vertices to the user, to be populated with initial state. Vertices are deleted using `hvr_vertex_delete_n`.

Given a vertex in the graph, a new attribute can be set or an old attribute updated to a new value using `hvr_vertex_set`. Similarly, `hvr_vertex_get` can be used to fetch the current value of an attribute.

```
hvr_vertex_t *hvr_vertex_create_n(size_t nvecs,
        hvr_graph_id_t graph, hvr_ctx_t ctx);
void hvr_vertex_set(unsigned feature, double val,
        hvr_vertex_t *vec, hvr_ctx_t in_ctx);
double hvr_vertex_get(unsigned feature, hvr_vertex_t *vec,
        hvr_ctx_t in_ctx);
void hvr_vertex_delete_n(hvr_vertex_t *vecs, size_t nvecs,
        hvr_ctx_t ctx);
```

3.2 Core APIS

The core of HOOVER is encapsulated in four APIS.

`hvr_ctx_create` initializes the state of a user-allocated HOOVER context object. The HOOVER context is used to store global state for a given HOOVER simulation. HOOVER assumes that the user has already called `shmem_init` to initialize the OpenSHMEM runtime before calling `hvr_ctx_create`.

```
extern void hvr_ctx_create(hvr_ctx_t *out_ctx);
```

`hvr_init` completes initialization of the HOOVER context object by populating it with several pieces of user-provided information (e.g. application callbacks) and allocating internal data structures. `hvr_init` does not launch the simulation itself, but is the last step before doing so.

```
void hvr_init(const uint16_t n_partitions,
        hvr_start_iteration start_iteration,
        hvr_update_metadata_func update_metadata,
        hvr_check_abort_func check_abort,
        hvr_might_interact_func might_interact,
        hvr_actor_to_partition actor_to_partition,
        const double connectivity_threshold,
        const unsigned min_spatial_feature_inclusive,
        const unsigned max_spatial_feature_inclusive,
        const hvr_iter_t max_iteration, hvr_ctx_t ctx);
```

The arguments passed are described below:

1. `n_partitions` - During execution, HOOVER divides the simulation space up into partitions. These partitions are used to detect possible interactions between vertices in different PEs by first checking for vertex-to-partition interaction. This argument specifies the number of partitions the application developer would like used.
2. `start_iteration` - A user callback that is called at the beginning of each iteration and passed an iterator over the vertices in the local PE.
3. `update_metadata` - On each iteration, `update_metadata` is called on each local vertex one-by-one along with the vertices that vertex has edges with (including remote vertices). update_metadata is responsible for making any changes to the state of the vertex, and deciding if based on those updates any remote PEs should become coupled with the current PE's execution.
4. `check_abort` - A callback used by the application developer to determine if the current PE should exit the simulation based on the state of all local vertices following a full iteration. `check_abort` also computes local metrics, which are then shared with coupled PEs.
5. `might_interact` - A callback used by the runtime to determine if a vertex in the provided partition may interact with any vertex in another partition.
6. `actor_to_partition` - A callback that computes the partition for a given vertex.
7. `connectivity_threshold`,
   `min_spatial_feature_inclusive`,
   `max_spatial_feature_inclusive` - These arguments are all used to update graph edges. HOOVER automatically updates edges based on their

"nearness" to other vertices in the simulation, by some distance measure. Today, that is simply a Euclidean distance measure on the features in the range [min_spatial_feature_inclusive, max_spatial_feature_inclusive]. If the computed distance is less than connectivity_threshold those vertices have an edge created between them.

8. max_iteration - A limit on the number of iterations for HOOVER to run.
9. ctx - The HOOVER context to initialize.

hvr_body is then used to launch the simulation problem, as specified by the provided ctx, and hvr_finalize is used to clean up HOOVER's state. hvr_body only returns when the local PE has completed execution, either by exceeding the maximum number of iterations or through a non-zero return code from the check_abort callback. HOOVER assumes that shmem_finalize is called after hvr_finalize.

```
extern void hvr_body(hvr_ctx_t ctx);
extern void hvr_finalize(hvr_ctx_t ctx);
```

3.3 HOOVER Application Skeleton

Given the above APIs, a standard HOOVER application has the following skeleton:

```
hvr_ctx_t ctx;
hvr_ctx_create(&ctx);
hvr_graph_id_t graph = hvr_graph_create(hvr_ctx);

// Create and initialize the vertices in the simulation
hvr_sparse_vec_t *vertices = hvr_sparse_vec_create_n(...);
...

hvr_init(...);

// Launch the simulation
hvr_body(ctx);

// Analyze and display final results of the simulation
...

hvr_finalize(ctx);
```

Internally, the kernel of hvr_body follows the following workflow:

```
while not abort and iter < max_iter:
  start_time_step(local_vertices)

  foreach vert in local_vertices:
    neighbors = gather_neighbors_along_edges(vert)
    update_metadata(vert, neighbors)
```

```
iter += 1

update_my_partitions()

foreach vert in local_vertices:
  update_edges(vert)

abort = check_abort()

block_on_coupled_pes()
```

4 HOOVER's Runtime

The core of HOOVER's coordination logic is included under the `hvr_body` API. `hvr_body` is responsible for coordinating the execution of the simulation from start to end.

The core of `hvr_body` is a loop. On each iteration, the following high level actions are taken:

1. **Start Iteration**: The user-provided `start_iteration` is called, which is passed an iterator over the vertices in the local part of the graph. This gives the user the opportunity to (optionally) perform any application-specific, per-iteration logic.
2. **Update Local Vertices**: All local vertices have their attributes updated using the `update_metadata` user callback.
3. **Update Local Partitions**: Information on the problem space partitions that contain local vertices is updated on the local PE and made visible to remote PEs.
4. **Find Nearby PEs**: Based on the partition information of other PEs, construct a list of all PEs which have vertices that local vertices may have edges with.
5. **Update Graph Edges**: Communicating only with the PEs that may have nearby vertices, update all inter-vertex edges.
6. **Check Abort**: Check if any updates to local vertices lead to this PE aborting using the `check_abort` user callback, and compute the local PE's contribution to any coupled metric.
7. **Compute Coupled Metric**: If coupled with other PEs, jointly compute a coupled metric with them.
8. Continue to the next iteration if no abort was indicated and we have not reached the maximum number of iterations.

The following sections provide additional details on subtleties in HOOVER's execution and data structures.

4.1 Versioned Vertices

While HOOVER vertices expose simple get and set APIs to the user, they are subtely complex.

The root of this complexity is the decoupled nature of HOOVER's execution. For scalability reasons, HOOVER was designed to avoid all two-sided, blocking, or collective operations between any two de-coupled PEs. As such, any PE may fetch vertex data from any other PE at any time during the simulation without any involvement from the remote PE. As such, the sparse vector data structure used to represent vertices must be designed to be remotely consistent.

Additionally, because HOOVER is iterative it has some measure of ordering of operations. De-coupled PEs may have reached very different iterations in the simulation before their first interaction. It may be undesirable (in some applications) for the slower PE to be able to read data from future iterations on the faster PE - we would like any information accessed to be mostly consistent for a given iteration. As a result, it is necessary to have some history or versioning built in to HOOVER's sparse vector data structure such that de-coupled PEs on different iterations can still fetch consistent data from each other.

Hence, internally the vertex data structure stores its state going back many iterations. Additionally, when updating a vertex with new values, those values are tagged with the current iteration. A simplified version of the actual sparse vector data structure used to represent graph vertices is shown below:

```
typedef struct _hvr_vertex_t {
    // Features, all entries in each bucket guaranteed unique
    unsigned features[HVR_BUCKETS][HVR_BUCKET_SIZE];

    // Values for each feature in each bucket
    double values[HVR_BUCKETS][HVR_BUCKET_SIZE];

    // Number of features present in each bucket
    unsigned bucket_size[HVR_BUCKETS];

    // Creation iteration for each bucket
    hvr_iter_t iterations[HVR_BUCKETS];
} hvr_vertex_t;
```

The sparse vector above has the ability to store history for this sparse vector's state going back `HVR_BUCKETS` iterations, with up to `HVR_BUCKET_SIZE` features in the sparse vector.

Each time the first attribute is set on a new iteration, a bucket is allocated to it by finding the oldest bucket (i.e. least recently used eviction policy). The most recent state of the sparse vector from the most recent iteration is copied to the new bucket. Then, additional changes for the current iteration are made on top of those copied values.

Anytime a feature needs to be read from a sparse vector, an iteration to read the value for is also passed in (either explicitly from the HOOVER runtime or implicitly using the calling PE's context). The bucket that is closest to that iteration but not past it is then used to return the requested feature. Finding the correct bucket is `O(HVR_BUCKETS)` in the worst case, but HOOVER maintains two indices into each vertex's buckets to accelerate lookups: (1) the index of the last bucket requested and the iteration that was requested, and (2) the index of the most recently created bucket.

While this design is flexible and solves the problem of de-coupled data accesses in a massively distributed system, it naturally comes with drawbacks. It is memory inefficient, consuming many times the number of bytes than what would be needed to simply store the current state of the sparse vector. Of course, this also has implications for bytes transferred over the network.

Edge Updates Updating the edges on a given vertex is an expensive operation. Each check to see if an edge should exist between two vertices may include both a remote vector fetch as well as a distance measure. Hence, edge updating is a multi-step process during which we try to eliminate as many remote vertices from consideration as possible without fetching the vertex itself. Key to this is the concept of partitions.

Partitions were introduced earlier, but will be described in more detail here. A partition is simply some subset of the current simulation's problem space, where the problem space is defined as all possible values that may be taken on by the positional attributes of any vertex. One of the simplest forms of partition would be a regular two-dimensional partitioning/gridding of a flat, two-dimensional problem space. However, the concept of a partition in HOOVER is more flexible than that as the user is never asked to explicitly specify the shape or bounds of any partition. They simply must define:

1. A total number of partitions (passed to `hvr_init`).
2. A callback for returning the partition for a given vertice's state.
3. A callback that tests for the possibility of partition-to-partition interaction (i.e. the possibility of any vertex in partition `A` interacting with any vertex in partition `B`)..

Partitions are key to reducing the number of pairwise distance checks needed during edge updating.

During an update to the edges of local actors, we iterate over all other PEs. For each PE we fetch the current actor-to-partition map of that PE. The actor-to-partition map is simply an array storing the partition of each actor on a PE, which is updated on each iteration. Then, for each actor on the remote PE in a partition which one of our locally active partitions may interact with we take a Euclidean distance with each of our local actors to determine which should have edges added.

To further reduce the number of remote memory accesses required we also use a fixed-size, LRU cache for remotely fetched vertices.

OpenSHMEM Read-Write Locks One common pattern repeated throughout HOOVER was the desire to atomically fetch a large, contiguous region of memory from a remote PE (similar to `shmem_atomic_fetch` but on larger numbers of bytes). In general, these regions of memory are remotely read and only locally written.

Currently, HOOVER supports this requirement by implementing read-write locks on top of OpenSHMEM APIs. Like the standard OpenSHMEM lock APIs, a read-write lock is a symmetrically allocated `long`, though in our case we add a custom allocator to allow for custom initialization:

```
long *hvr_rwlock_create_n(const int n);
```

These read-write locks have some semantic differences with standard OpenSHMEM locks (beyond the differences between read-write locks and standard locks). When locking an OpenSHMEM lock, mutual exclusion is guaranteed globally across all PEs for that lock. If a user has a distributed data structure and would like to lock only the chunk of it sitting in a particular PE, this leads to an (undesirable) pattern of allocating `npes` locks, each for mutual exclusion on a different PE's chunk.

Instead, allocating a single read-write lock in HOOVER is semantically allocating a lock per PE. When acquiring or releasing a read-write lock, a target PE must be specified along with the symmetrically allocated lock object. Mutual exclusion is only guaranteed for a given lock targeting a given PE. The APIs for read-write locks are listed below:

```
void hvr_rwlock_rlock(long *lock, const int target_pe);
void hvr_rwlock_runlock(long *lock, const int target_pe);
void hvr_rwlock_wlock(long *lock, const int target_pe);
void hvr_rwlock_wunlock(long *lock, const int target_pe);
```

Under the covers, the highest order bit in the symmetrically allocated `long` on each PE is set to acquire a write lock for that PE, while the remaining bits are used to count readers. If a reader attempts to lock and finds the highest order bit set, it will spin until the write lock clears. If a writer attempts to lock and finds one or more readers in the critical section, it will spin until they have all unlocked their read locks.

Dynamic Vertex Allocation and Deallocation To support adding and removing vertices, we must support dynamic allocation and de-allocation of HOOVER vertices from OpenSHMEM's symmetric heap. Today, that is accomplished with a memory pool that tracks free and used vertices in a pre-allocated chunk of the symmetric heap.

One subtlety of vertex deletion in the presence of de-coupled execution is that remote PEs may still request information on a deleted vertex after it is locally deleted, depending on how the problem is configured and which iteration they are on. As a result, deleted vertices are retained until all PEs have progressed past the point where any would request information on the deleted vertices. PEs share information on their current iteration with neighboring PEs, which in term share this information with their neighbors, leading to all PEs asynchronously receiving slightly out-of-date information on the current iteration of all other PEs. Once all PEs have passed the iteration on which a given vertex was deleted, it is safe to delete that vertex.

5 Performance Results

5.1 Mini-Apps

We focus our evaluation on two mini-apps developed as part of this work: a simplified infectious disease model and an intrusion detection model.

Infectious Disease Model In our infectious disease model each node in the graph represents an actor, i.e. a person or device that could be infected by a bacterial or electronic bug. Actors are assigned a home location, and then repeatedly given random destinations to travel to that are within some radius of their home. Edges between actors indicate some physical proximity to each other, allowing infection to spread between nearby actors. One or more actors in the simulation are initialized to be infected, with the remainder initialized to uninfected. On each HOOVER iteration, an actor's location is updated based on the current destination it is traveling towards, and it becomes infected if any of the vertices it shares edges with are also infected. PEs couple when an actor from one PE infects an actor on another PE.

Intrusion Detection Model Our intrusion detection model is based on the GBAD graph-based anomaly detection algorithm [4]. In GBAD, the goal is to find anomalies (i.e. rarities) in the structure of a graph which look similar to common patterns, but which are not the same. Nodes in this graph might represent system events, network packets, or other user activities. In our implementation, each PE computes a local set of normative/common subgraph patterns. These patterns are then shared globally and asynchronously among PEs, and used to compute a global set of normative patterns. Each PE then locally looks for patterns which are similar to the normative patterns, but not the same. Note that these patterns may contain edges that cross PEs and include remote vertices. PEs couple when an anomalous pattern is discovered with cross-PE edges.

5.2 Evaluation Platform

HOOVER has been tested on the OSSS, SoS, Cray, HPE, and OpenMPI OpenSHMEM implementations. It has also been tested on ARM- and Intel-based platforms.

The experiments presented here were run on the NERSC Edison machine. Edison is a Cray* XC30 with 2×12-core Intel® Xeon® Processors E5-2695 v2 and 64 GB DDR3 in each node. Edison nodes are connected by the Aries interconnect. All experiments are run on Cray SHMEM 7.7.0. All tests are run with one PE per core (24 PEs per node).

5.3 Scaling Results

For strong scaling experiments of the infectious disease model, we use a problem consisting of a 16,000 × 24,000 two-dimensional grid with 9,830,400 actors moving on it. Strong scaling results out to 256 nodes are shown in Table 1. Thanks to its decoupled-by-default design, HOOVER is able to continue to show strong scaling performance improvements out to over 6,000 PEs.

# PEs	Execution Time (ms)	Speedup Relative to Previous
384	56,296	
1,536	13,229	4.26×
6,144	6,642	1.99×
24,576	4,472	1.49×

Table 1. Strong scaling tests with 9,830,400 actors in the infectious disease model on the Edison supercomputer.

Table 2 shows the results of weak scaling experiments of our intrusion detection model out to 3,072 PEs. In these tests, each PE inserts a random number of random nodes in the graph on each iteration as part of the `start_iteration` callback. This emulates the ingestion of new events in a real world auditing system. Tests are run for a fixed walltime. Table 2 reports the number of nodes that were handled by the end of the simulation, demonstrating that with more hardware the system is able to process events at a consistently higher rate.

# PEs	Nodes Processed	Improvement Relative to Previous
384	2,228,526	
768	3,788,324	1.70×
1,536	6,143,523	1.62×
3,072	9,087,829	1.48×
6,144	12,787,045	1.41×

Table 2. Weak scaling tests of the intrusion detection model on the Edison supercomputer.

Decoupled execution is a foundational component of HOOVER's performance, but also leads to overheads that other systems lack. In particular, decoupled execution eliminates synchronization but requires that the history of a vertex's attributes be kept and communicated between PEs. It is important to test that the overheads removed by decoupled execution are greater than the overheads introduced. To that end, experiments were run using the intrusion detection model at 3,072 PEs with vertex history tracking disabled and a global barrier per iteration to ensure PEs remain in-sync. While decoupled execution was able to process 9,087,829 nodes, the synchronized version only processed 3,672,764 ($\sim$40%).

6 Related Work

6.1 Distributed Graph Analytics

While most of today's graph analytics frameworks are single node and shared memory, we summarize the distributed frameworks here.

GraphX [5] is a popular graph processing framework built on top of the Apache Spark framework. As a result, it supports scaling out to large distributed systems (though the original paper only measures scalability out to 16 nodes) and composability with other frameworks built on Spark. GraphX represents graphs as distributed arrays of vertex and edge attributes stored in Spark RDDs, which are often presented as arrays of "triplets" where each triplet contains an edge and references to the vertices it connects. GraphX adds graph-specific operators on top of Spark to make processing GraphX graphs easier (e.g. a `mapE` operation that maps a function across edges).

LFGraph [6] is a distributed graph processing framework that focuses on value propagation problems (rather than computing on the graph itself) on static graphs. The dataflow API focuses on fetching the updated values of neighboring vertices, and updating the current value of the current vertex. Hence, LFGraph is focused on computing classical graph statistics (e.g. PageRank, Undirected Triangle Count, etc) rather than modeling more complex systems. While LFGraph is fault tolerant and is designed to run on commodity hardware, its experimental evaluation only measures its scalability out to 64 nodes.

Distributed GraphLab [8] is a distributed graph processing framework for static graphs. GraphLab's programming abstractions consist of a "data graph" which allows user's to attach metadata to each vertex and edge in the static graph, "update functions" which updates the state of a vertex and may schedule processing of other vertices, and "sync operations" which update global data structures based on read-only access to the data graph. GraphLab uses a preprocessing step to initially over-partition the target graph into many files on the storage system, and then loads these partitions (similar to HOOVER's partitions) in a distributed fashion across nodes. GraphLab uses its own custom execution engine to manage computation and communication across nodes (i.e., does not sit on top of Spark or some other framework).

Pregel [9] offers a message-based programming model for dynamic, distributed graph processing: "Programs are expressed as a sequence of iterations, in each of which a vertex can receive messages sent in the previous iteration, send messages to other vertices, and modify its own state and that of its outgoing edges or mutate graph topology". Pregel has its own execution engine that coordinates the communication of messages and their processing at each vertex.

Apache Flink [1] is an open source system for distributed stream and batch processing, which exposes a graph API called Gelly [7]. Gelly offers common graph operations, supports dynamic graph mutation, and supports both vertex-centric and edge-centric APIs.

Many of the frameworks above and in other literature have several properties in common that contrast them with HOOVER:

1. Bulk synchronous execution: Most frameworks make frequent use of global barriers to coordinate execution, limiting scalability.
2. Focus on static rather than dynamic graphs: Most frameworks focus on static rather than dynamic graphs (with some exceptions, such as Pregel and Gelly).
3. Custom execution/coordination engines: Many frameworks implement their own custom execution, coordination, and communication engines for schedul-

ing work. HOOVER, on the other hand, leverages years of work tuning OpenSHMEM runtimes for performance and stability.

4. Support for fault tolerance: HOOVER does not currently support fault tolerance, though active work is exploring this avenue of research.

6.2 Graph-Based Intrusion Detection

While not the primary contribution of this work, the intrusion detection mini-app described in Section 5.1 is inspired by earlier work.

GBAD [4] is the seminal graph-based anomaly detection algorithm on which our intrusion detection mini-app is based. The GBAD paper introduced the idea of thinking about anomalies as patterns in a graph which look similar to a common, normative pattern but which is not exactly identical.

Eberle et al. [3] introduced a distributed version of the GBAD algorithm. While this distributed extension is bulk synchronous, it shares similarities to our intrusion detection mini-app by computing local normative patterns, using them to find global normative patterns, and then reporting local anomalies based on those global normative patterns.

7 Future Work

While HOOVER's decoupled approach to graph processing offers promise for scaling to larger graph problems than are solveable with existing frameworks, HOOVER is still an active and evolving project with several avenues of future and ongoing investigation:

1. Explicit edge creation: HOOVER's current approach to edge creation is implicit – edges are created when two vertices become close by some distance measure. We plan to explore alternative, explicit ways to prescribing edge creation and study their impact on performance.
2. Automatic load balancing of vertices between PEs.
3. Experiment with hybrid and heterogeneous parallelism.
4. Improved infectious disease model: Work is actively exploring making the simple infectious disease model into a more realistic application.
5. Improved vertex memory efficiency: Versioned vertices consume large amounts of space to store their state over many iterations. This costs memory and bytes over the wire. Exploring ways to compress these large data structures without a loss of information would be beneficial for performance.
6. Cross OpenSHMEM implementation performance comparison: While HOOVER has been tested across several OpenSHMEM implementations for correctness, we are interested in using it as a point-of-comparison for performance.

8 Conclusion

When it comes to distributed streaming graph processing, the choice of frameworks is extremely limited today. Most graph processing frameworks do one or the other (distributed or streaming), but not both. The underlying reason for this is the challenge of efficiently scaling graph applications on rapidly mutating graphs with highly irregular computation and memory access, all using a bulk synchronous model.

HOOVER avoids this problem by using OpenSHMEM to enable fully decoupled parallel execution, minimizing communication and synchronization by keeping it local to only those PEs that must interact. HOOVER offers a sufficiently flexible API to support a wide range of graph processing applications, while enabling scaling out to thousands of PEs and terabytes of memory.

Acknowledgments

The authors would like to thank Steve Poole (LANL) for his valuable feedback on the HOOVER project and this manuscript.

Work on HOOVER was funded in part by the United States Department of Defense, and was supported by resources at Los Alamos National Laboratory. This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

1. Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., Tzoumas, K.: Apache flink: Stream and batch processing in a single engine. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering 36(4) (2015)
2. Chapman, B., Curtis, T., Pophale, S., Poole, S., Kuehn, J., Koelbel, C., Smith, L.: Introducing openshmem: Shmem for the pgas community. In: Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model. p. 2. ACM (2010)
3. Eberle, W., Holder, L.: Scalable anomaly detection in graphs. Intelligent Data Analysis 19(1), 57–74 (2015)
4. Eberle, W., Holder, L.B.: Mining for structural anomalies in graph-based data. In: DMIN. pp. 376–389 (2007)
5. Gonzalez, J.E., Xin, R.S., Dave, A., Crankshaw, D., Franklin, M.J., Stoica, I.: Graphx: Graph processing in a distributed dataflow framework. In: OSDI. vol. 14, pp. 599–613 (2014)
6. Hoque, I., Gupta, I.: Lfgraph: Simple and fast distributed graph analytics. In: Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems. p. 9. ACM (2013)
7. Kalavri, V.: Gelly: Large-Scale Graph Processing with Apache Flink. `https://www.slideshare.net/vkalavri/gelly-in-apache-flink-bay-area-meetup` (2015)
8. Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., Hellerstein, J.M.: Distributed graphlab: a framework for machine learning and data mining in the cloud. Proceedings of the VLDB Endowment 5(8), 716–727 (2012)
9. Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. pp. 135–146. ACM (2010)