HOOVER

Max Grossman[1], Howard Pritchard[2], Tony Curtis[3], and Vivek Sarkar[4]

[1] Rice University (`max.grossman@rice.edu`)
[2] Los Alamos National Laboratory
[3] Stony Brook University
[4] Georgia Institute of Technology

1 What is HOOVER?

HOOVER is a distributed framework for solving streaming/dynamic graph problems where the structure of a graph changes over time based on the state of its vertices. HOOVER emphasizes flexibility without sacrificing scalability, allowing users to plug in nearly-arbitrary callbacks for application-specific logic while:

1. Using OpenSHMEM 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.

The following sections serve as both a design document as well as a user guide, describing the high level software architecture of HOOVER as well as how one uses it.

Any questions on HOOVER can be posted on the HOOVER Github (`https://github.com/agrippa/hoover`) or sent to max.grossman@rice.edu.

2 High-Level Usage

HOOVER exposes a C/C++ API to a SPMD programming model, inherited from the OpenSHMEM programming model that it is built on top of. While most of the application code does not need to be aware of the SPMD-ness of the underlying runtime, application initialization is performed in parallel across all PEs in the simulation. This choice was made deliberately so that large datasets could be loaded across all PEs, rather than exposing sequential semantics by only loading the dataset on a single PE and then broadcasting it.

The core data structure of HOOVER is a graph vertex. In HOOVER, a vertex can have zero or more attributes attached to it, stored as a sparse vector. During the execution of a HOOVER simulation, the user-level, application-specific code is primarily responsible for reading and updating attributes on local and remote vertices based on application-specific semantics.

Attributes in HOOVER are split into two categories: positional and logical. Positional attributes on a vertex define its relationship and distance to other vertices in the simulation. Positional attributes are used by HOOVER to automatically update edges between vertices as they move "closer" or "farther" away. For example, positional attributes might be the physical location of a vertex in a 2D or 3D space, defining its location relative to other vertices. Logical attributes are additional vertex attributes used by the programmer for application-specific information, but which are not used by the HOOVER runtime system.

At a high level, a HOOVER program starts with the user configuring a graph problem by defining the initial state/attributes of all vertices in it. This initial state is then passed into the HOOVER runtime along with several application-specific callbacks. All future execution is then coordinated by the HOOVER runtime, with application callbacks made when application-specific behavior or updates are needed.

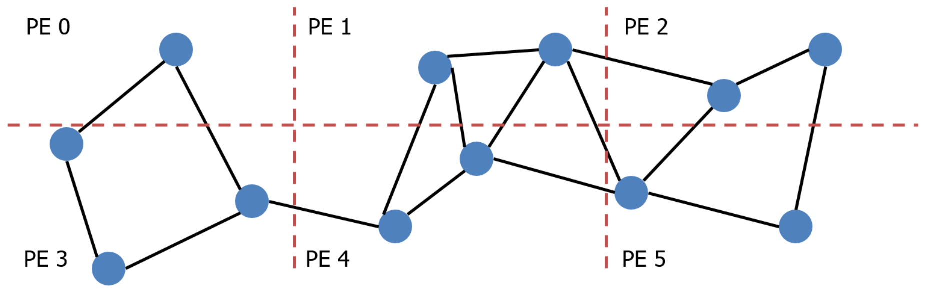

Fig. 1. Illustration of a HOOVER graph distributed across PEs with cross-PE edges.

2.1 Target Application Pattern

While HOOVER is a general dynamic graph processing, simulation, and analytics framework it also offers specialized capabilities for a particular class of dynamic graph problems. The classical HOOVER problem follows the following high level execution flow:

1. The application defines a large number of vertices partitioned across PEs, as well as callbacks to update their state (among other things).
2. The HOOVER framework begins iterative modeling of vertex behavior through repeated callbacks to user-level functions, evolving vertex 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 is ever blocked on or performing two-sided communication with any other PE.
3. After some time, two or more PEs discover they are related. This "relationship" is entirely user-directed and in the control of user callbacks. After this connectivity is discovered, those PEs will enter lockstep execution with each

other and share data on each timestep. 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 the application.
4. Eventually, application termination is either signalled by a PE or controlled by all PEs reaching a maximum number of timesteps.

At this point, an illustrative example might be useful: malware spread over Bluetooth. Malware propagation modeling can be expressed as a graph problem, where vertices in the graph represent Bluetooth devices and edges represent direct connectivity between two devices. Executing malware propagation on the HOOVER framework might look something like:

1. The application developer would define the actors in the simulation as vertices passed down to the HOOVER framework. Each actor would represent a device, and may include attributes such as the range of its Bluetooth hardware, the model its Bluetooth hardware, 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 infection 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.

3 API

3.1 Vertex APIs

The core data structure of HOOVER is the graph vertex, represented by objects of type `hvr_sparse_vec_t`. Application developers are not expected to manipulate the internal state of this object directly, but rather by using the following APIs.

Below, several terms may be used interchangeably to refer to a simulation vertex, including vertex, sparse vector, or actor.

hvr_sparse_vec_create_n

```
hvr_sparse_vec_t *hvr_sparse_vec_create_n(
        const size_t nvecs);
```

Create `nvecs` sparse vectors/vertices. This API must be called collectively on all PEs with the same value of `nvecs` on each PE. This will return initialized but empty vertices to the user, to be populated with initial state.

hvr_sparse_vec_set_id

```
void hvr_sparse_vec_set_id(const vertex_id_t id,
        hvr_sparse_vec_t *vec);
```

Set a globally unique ID `id` for vertex `vec` in the current simulation. It is the programmer's responsibility to provide each vertex a globally unique ID, and the runtime performs no checks to verify the uniqueness of each ID. This API must be called once for each vertex in a given simulation before launching the simulation with the `hvr_body` API (described below).

Vertex IDs do not need to follow any pattern or be contiguous, but application developers are advised to assign their vertex IDs in such a way that looking up the PE for a given vertex based on its ID is a quick, efficient, and preferrably constant time operation.

hvr_sparse_vec_get_id

```
vertex_id_t hvr_sparse_vec_get_id(hvr_sparse_vec_t *vec);
```

Fetch the globally unique ID assigned to the passed vertex.

hvr_sparse_vec_get_owning_pe

```
int hvr_sparse_vec_get_owning_pe(hvr_sparse_vec_t *vec);
```

Fetch the PE that owns the passed vertex, i.e. the PE that created and initialized it. Information on PE ownership is automatically maintained by the HOOVER runtime. No dynamic load balancing of vertices is performed in HOOVER, and so vertex ownership is a static and one-to-one relationship.

hvr_sparse_vec_set

```
void hvr_sparse_vec_set(const unsigned feature,
        const double val, hvr_sparse_vec_t *vec,
        hvr_ctx_t in_ctx);
```

Set the vertex attribute identified by `feature` to store the value `val` in the vertex `vec`.

hvr_sparse_vec_get

```
double hvr_sparse_vec_get(const unsigned feature,
        hvr_sparse_vec_t *vec, hvr_ctx_t in_ctx);
```

Fetch the value stored for attribute `feature` in vertex `vec`. If the specified attribute was never set on this vertex, the default behavior is to abort.

hvr_sparse_vec_dump

```
void hvr_sparse_vec_dump(hvr_sparse_vec_t *vec, char *buf,
        const size_t buf_size, hvr_ctx_t in_ctx);
```

A utility function for generating a human-readable string for the passed vertex into the provided character `buf`, which is at least of length `buf_size` bytes. This API can be useful for debugging or other user diagnostics.

3.2 Core APIS

The core of HOOVER is encapsulated in four APIS.

hvr_ctx_create

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

`hvr_ctx_create` initializes the state of a user-allocated HOOVER context object to be used in later HOOVER user and runtime operations. This API does not allocate space for the context, the `out_ctx` parameter is expected to point to at least `sizeof(hvr_ctx_t)` bytes of valid memory. `out_ctx` may be on the stack or heap memory segment.

The HOOVER context is used to store several pieces of global state for a given HOOVER simulation, such as a pointer to the local vertices in the simulation being managed by the current PE. This should be considered an opaque object by application developers, and internal context state should not be directly manipulated.

HOOVER assumes that the user has already called `shmem_init` to initialize the OpenSHMEM runtime before calling `hvr_ctx_create`.

hvr_init

```
extern void hvr_init(const uint16_t n_partitions,
        const vertex_id_t n_local_vertices,
        hvr_sparse_vec_t *vertices,
        hvr_update_metadata_func update_metadata,
        hvr_might_interact_func might_interact,
        hvr_check_abort_func check_abort,
        hvr_vertex_owner_func vertex_owner,
        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_time_t max_timestep, hvr_ctx_t ctx);
```

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

1. `n_partitions` - During execution, HOOVER divides the simulation space up into partitions as directed by the `actor_to_partition` callback (described below). These partitions are used to approximately detect interactions between actors by first finding actor-to-partition interaction. This is similar but not identical to techniques used in the Fast Multipole Method. This argument specifies the number of partitions the application developer will use.
2. `n_local_vertices` - The number of vertices managed by the local PE.
3. `vertices` - The vertices managed by the local PE. This array should be allocated using `hvr_sparse_vec_create_n`, and each vertex in it should have been initialized using `hvr_sparse_vec_set_id` and `hvr_sparse_vec_set`.
4. `update_metadata` - A user callback. On each timestep, `update_metadata` is passed each local vertex one-by-one along with all vertices that the current vertex has edges with (including remote vertices). update_metadata is responsible for updating the local state of the current vertex, and deciding if based on those updates any remote PEs should become coupled with the current PE's execution.
5. `might_interact` - A user callback. `might_interact` is called with a single partition ID and a set of partitions to determine if a vertex in the provided partition may interact with any actor in any partition in the passed set.
6. `check_abort` - A user callback. `check_abort` is used by the application developer to determine if the current simulation should exit based on the state of all local vertices following a full timestep. `check_abort` also computes any local metrics, which are then shared with coupled PEs to compute coupled metrics.
7. `vertex_owner` - A user callback. Given a globally unique vertex ID, the user is expected to return the PE owning that vertex and the offset in that PE's vertices of the specified vertex.
8. `actor_to_partition` - A user callback. Given a vertex, return the partition it belongs in.
9. `connectivity_threshold`,
   `min_spatial_feature_inclusive`,
   `max_spatial_feature_inclusive` - These arguments are all used to update edges. Recall that HOOVER automatically updates inter-vertex edges based on their "nearness" to other vertices in the simulation, by some distance measure. Today, that is simply an N-dimensional 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.
10. `max_timestep` - A limit on the number of timesteps for HOOVER to run.
11. `ctx` - The HOOVER context to initialize. This ctx should already have been zeroed using `hvr_ctx_create`.

hvr_body

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

Launch the simulation problem, as specified by the provided `ctx`. `hvr_body` only returns when the local PE has completed execution, either by exceeding the maximum number of timesteps or through a non-zero return code from the `check_abort` callback.

hvr_finalize

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

Perform cleanup of the simulation state. HOOVER assumes that `shmem_finalize` is called after `hvr_finalize`.

3.3 HOOVER Application Skeleton

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

```
hvr_ctx_t ctx;
hvr_ctx_create(&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 the results of the simulation
...

hvr_finalize(ctx);
```

4 Runtime

With our understanding of the user-facing APIs for HOOVER, we can now discuss how HOOVER operates on the backend. We will split this into two discussions: 1) a description of the vertex/sparse vector data structure used, and 2) a description of the steps taken by the HOOVER runtime on each timestep of an iterative simulation.

4.1 Versioned Sparse Vectors

While HOOVER sparse vectors 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 PE being accessed. As such, the sparse vector data structure must be designed to be always consistent, such that remote accessors can get information from it even if the owning PE is currently modifying it.

Additionally, because HOOVER is iterative it has some measure of progress, time, and ordering between timesteps. Indeed, de-coupled PEs may have reached very different timesteps in the simulation before their first interaction. However, it would be undesirable for the slower PE to be able to read data from the future on the faster PE - we would like any information accessed to be mostly consistent for a given timestep (though perhaps not from that precise timestep). 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 timesteps can still fetch consistent data from each other.

Hence, internally the sparse vector data structure stores its state going back many timesteps. Additionally, when updating a sparse vector with new values, those values are tagged with the current timestep. A simplified version of the actual sparse vector declaration is shown below:

```
typedef struct _hvr_sparse_vec_t {
    // Globally unique ID for this node
    vertex_id_t id;

    // PE that owns this vertex
    int pe;

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

    // Feature IDs, all entries in each bucket guaranteed unique
    unsigned features[HVR_BUCKETS][HVR_BUCKET_SIZE];

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

    // Timestamp for each bucket
    hvr_time_t timestamps[HVR_BUCKETS];
} hvr_sparse_vec_t;
```

Here, the key fields are `timestamps`, `bucket_size`, `values`, and `features` each of which is a circular buffer. The sparse vector above has the ability to store history for this sparse vector's state going back `HVR_BUCKETS` timesteps, with up to `HVR_BUCKET_SIZE` features in the sparse vector.

Each time the first attribute is set on a new timestep, a bucket is allocated to it by finding the oldest bucket. The most recent state of the sparse vector from the most recent timestep is copied to the new bucket, including its `features`, `values`, and `bucket_size`. Then, additional changes for the current timestep are made on top of those copied values.

Anytime a feature needs to be read from a sparse vector, a timestep to read the value for is also passed in (either explicitly if from the HOOVER runtime or implicitly using the calling PE's context). The bucket that is closest to that timestep but not past it is then used to return the requested feature.

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. This design also limits how out-of-sync two PEs can become as a result of using a fixed-size circular buffer. If one PE becomes more than `HVR_BUCKETS` behind the other PE, it will no longer be able to fetch valid values from the other's vertices.

4.2 Core Runtime Execution Flow

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

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

1. **Update Local Actors**: All local vertices have their attributes updated using the `update_metadata` user callback.
2. **Update Local Partitions**: Information on the problem space partitions that contain local actors is updated on the local PE and made accessible by remote PEs. This step uses the `actor_to_partition` user callback.
3. **Find Nearby PEs**: Based on the partition information of other PEs, this PE constructs a list of all PEs which have actors that local actors may interact with using the `might_interact` user callback.
4. **Update Graph Edges**: Communicating only with the PEs that may have nearby actors, update all inter-vertex edges.
5. **Check Abort**: Check if any updates to local actors lead to this PE aborting using the `check_abort` user callback, and compute the local actor's contribution to any coupled metric.
6. **Compute Coupled Metric**: If coupled with other PEs, jointly compute a coupled metric with them through an all-reduction.
7. Continue to the next iteration if no abort was indicated and we have not reached the maximum number of timesteps.

Below we offer more details on how some of the steps above are implemented.

Local Actor Attribute Updates Updating each local actor's state via the `update_metadata` callback requires collecting the state of all other actors that it has edges with.

Internally, HOOVER stores the edge information for each actor as an AVL tree where each node contains the globally unique vertex ID of edge-connected vertices (local or remote). An AVL tree is used to limit the amount of memory consumed while offering better than O(log(N)) insertions and lookups.

When collecting the state of all actors with edges on a local actor, we simply linearize the AVL tree storing the neighboring actors and iterate over this edge list. The `shmem_getmem` and `shmem_fence` OpenSHMEM APIs are used together to ensure we fetch a consistent view of the remote actor even if bytes are delivered out-of-order. Then, the collected actors are passed with the actor into the `update_metadata` user callback.

Edge Updates Because edge updating is an expensive operation and each check to see if an edge should be added may include both a remote vector fetch as well as a Euclidean distance measure, 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 sub-chunk 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 in the simulation. 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 maximum number of partitions (passed to `hvr_init`).
2. A callback for returning the partition for a given actor (note that this may change over time as the attributes on an actor change).
3. A callback that tests for the possibility of partition-to-partition interaction.

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 timestep. Then, for each actor on the remote PE that is in a partition which one of our locally active partitions may interact with (based on the `might_interact` callback) we take a Euclidean distance with each of our local actors to determine which should have edges added. Hence, partitions can dramatically reduce the number of these checks necessary. In general, it is advisable to create many more partitions than there are PEs. At a minimum, the number of partitions should equal the number of PEs.

5 Performance of an Example Use Case

As an example, we use a simplified infectious disease modeling problem to demonstrate and benchmark the current state of the HOOVER framework.

Our simple infectious disease model is expressed on a two-dimensional problem space, which represents some geographic area. Partitions are created as a regular, two-dimensional grid across the whole problem space.

Each actor in the problem is given 7 attributes:

1. A two-component position.
2. A two-component home location for this vertex.
3. A two-component current destination location for this vertex.
4. A single attribute indicating if a vertex is infected or uninfected.

On each timestep, each actor does the following:

1. If the current actor is still uninfected, it iterates over all vertices with which it shares edges and checks if any are infected. If any are infected, the current actor marks itself infected. If the actor which infects this actor comes from another PE, the current PE becomes coupled with the remote PE.
2. The current actor then updates its current position based on its home location and its destination location. An actor's home location is some point in the 2D problem space which it never moves more than a certain distance from. A point's destination location is the current point in the 2D problem space that an actor is moving towards, which must be near its home location.
   (a) If the actor has not yet reached its destination location, it simply updates its current location to continue moving towards it.
   (b) If the actor has reached its destination location, it computes a new destination location within some radius of its home and begins moving towards its new destination.

Benchmarking scalability of irregular problems like those that the HOOVER framework targets can be difficult, as even small changes in the scale of the problem or compute resources available can drastically impact the communication or computational patterns of the application.

Additionally, we would like to strongly emphasize that all performance results shown here are works-in-progress and that this report will be frequently updated as scaling improves. Performance bottlenecks and issues continue to be ironed out of the HOOVER framework, and scalability improves on a weekly basis. However, this report will serve as a useful document for tracking those improvements.

Still, we try to use this example problem to illustrate the current scaling characteristics of the HOOVER framework as several parameters and tunables are changed.

These experiments are run on a small cluster at Los Alamos National Laboratory consisting of SGI/HPE C1104-GP2 servers connected with Mellanox ConnectX5. Each server has 2 sockets, each containing 8 hyperthreaded cores, and 64 GB of memory. In the experiments below, we run with one PE per core (i.e. 8 PEs per socket).

As performance continues to improve over time, this report will be updated with those improvements. Table 1 maps from version numbers used in the text below to commit hashes in the Github repo. In general, any figures/tables will include in their caption which version of HOOVER the results are collected from.

Version Number	Git Hash	Date
0.1	`385911bb27f74fd74a8f038f95a2447cda372ec4`	Feb 10 2018
0.2	`38591d9adf732c144ced382175348031f8518bad`	Feb 27 2018
0.3	`4f9350fcc2d624ae14cffde6aae08b63a7d3ad16`	Mar 02 2018
0.4	`0129e2a81c2a14959e8966965b25c3fc7cf5c752`	Mar 17 2018

Table 1. Strong scaling of the HOOVER framework on OSSS OpenSHMEM running a simple infectious disease model with varying # of iters and infection radius.

These experiments are run with OSSS OpenSHMEM over UCX. HOOVER is compiled using gcc 6.3.0 with -O2 optimization turned on.

5.1 Strong Scaling

These experiments test the strong scaling of HOOVER while varying two simulation parameters but keeping the problem space fixed. We vary the number of timesteps/iterations to run the simulation for, and the infection radius. Infection radius controls how quickly the infection spreads from one actor to another, and so an increased infection radius leads to more infected actors across more PEs (and hence, more rapid PE coupling). Tables 2 and 3 show the results of these experiments.

# Iters	**10**		**100**		**200**	
Infection Radius	4.0	10.0	4.0	10.0	4.0	10.0
1 node	22,360	29,299	239,527	298,234	459,211	676,952
4 nodes	1,725	3,075	16,167	32,620	32,915	66,724
16 nodes	4,389	2,916	31,517	50,068	60,293	115,186

Table 2. Strong scaling of the HOOVER framework v0.1 on OSSS OpenSHMEM running a simple infectious disease model with varying # of iters and infection radius.

Tables 2, 3, 4, and 5 exhibit a few interesting trends:

1. HOOVER does not always exhibit linear strong scaling from 4 to 16 nodes (16 to 256 PEs), but in most cases does see significant speedup.
2. In many cases, HOOVER sees super-linear scaling from 1 node to 4 nodes, likely due to drastic changes in communication patterns as we spread the same dataset more thinly across PEs connected by a network.

# Iters	**10**		**100**		**200**	
Infection Radius	4.0	10.0	4.0	10.0	4.0	10.0
1 node	15,169	18,321	145,107	197,610	291,928	399,285
4 nodes	1,813	2,804	13,043	22,007	25,785	44,847
16 nodes	1,870	1,877	7,201	12,364	9,658	27,461

Table 3. Strong scaling of the HOOVER framework v0.2 on OSSS OpenSHMEM running a simple infectious disease model with varying # of iters and infection radius.

# Iters	**10**		**100**		**200**	
Infection Radius	4.0	10.0	4.0	10.0	4.0	10.0
1 node	12,919	18,139	142,675	193,895	303,930	401,899
4 nodes	1,817	3,043	12,321	22,917	24,266	42,755
16 nodes	340	657	3,204	12,166	6,996	28,947

Table 4. Strong scaling of the HOOVER framework v0.3 on OSSS OpenSHMEM running a simple infectious disease model with varying # of iters and infection radius.

3. Increasing the infection radius significantly increases simulation time. A likely cause of this is that it increases the number of edges on each actor, increasing the amount of time it takes to update each actor's attributes when checking for infection.

5.2 Weak Scaling

These experiments test the weak scaling of HOOVER. We test in multiples of 4 PEs. In each quadrupling of PEs, we increase the problem space size by 2× on each axis while keeping the number of actors per PE fixed. Hence, with each quadrupling of PEs the problem space size and number of actors quadruples. Tables 6 and 7 show the results of these experiments.

Like the strong scaling results, the weak scaling results in Tables 6, 7, and 9 show imperfect scaling from 1 to 4 to 16 nodes as more PEs require more compute time.

5.3 Larger Scale Tests

The strong and weak scaling tests described in Sections 5.1 and 5.2 were only run on up to 16 nodes on the small hickok cluster at LANL.

To better evaluate the scalability of HOOVER, larger strong scaling experiments were run on the Edison supercomputer at NERSC. Edison has 2x12-core

# Iters	**10**		**100**		**200**	
Infection Radius	4.0	10.0	4.0	10.0	4.0	10.0
4 nodes	1,811	2,939	14,606	24,815	30,957	45,612
16 nodes	576	574	4,618	5,141	9,530	9,981
Speedup	3.14×	5.12×	3.16×	4.83×	3.25×	4.57×

Table 5. Strong scaling of the HOOVER framework v0.4 on OSSS OpenSHMEM running a simple infectious disease model with varying # of iters and infection radius.

# Iters	**10**		**100**		**200**	
Infection Radius	4.0	10.0	4.0	10.0	4.0	10.0
1 node	14,957	21,729	160,171	225,808	310,550	446,786
4 nodes	17,245	22,664	178,586	235,632	344,221	482,851
16 nodes	22,696	28,988	232,528	297,360	469,306	600,663

Table 6. Weak scaling of the HOOVER framework v0.1 on OSSS OpenSHMEM running a simple infectious disease model with varying # of iters and infection radius.

# Iters	**10**		**100**		**200**	
Infection Radius	4.0	10.0	4.0	10.0	4.0	10.0
1 node	9,415	12,875	96,341	135,000	198,050	289,314
4 nodes	11,820	18,629	124,190	158,152	250,276	323,039
16 nodes	18,377	20,299	181,467	216,053	374,546	443,111

Table 7. Weak scaling of the HOOVER framework v0.2 on OSSS OpenSHMEM running a simple infectious disease model with varying # of iters and infection radius.

CPUs in each node with 64 GB of memory. All nodes are connected by the Cray Aries high performance interconnect.

For these experiments, we continue to use the infectious disease modeling mini-app with a domain size of 16,000 $\times$ 24,000, and 460,800 actors in total. This problem size is kept fixed across all runs. Experiments are run with 1 PE per core on 4, 16, 64, and 256 nodes (i.e. 96, 384, 1536, and 6144 PEs).

5.4 Parameter Design Space Exploration

The HOOVER runtime itself has many parameters that can be varied. In the sections that follow, we explore how varying these parameters impacts execution time for a fixed problem size and number of nodes. We focus on 16 nodes, 100 timesteps, and an infection radius of 10.0.

5.5 Sparse Vector Buckets

One tunable parameter is the amount of history to retain per sparse vector. By default in the tests run above, each sparse vector retains its history for the past 1,024 timesteps.

Varying the number of timesteps retained offers an interesting performance tradeoff. While reducing the retained timesteps reduces the size of the sparse vector data structure and the bytes over the network, it also can reduce the

# Iters	**10**		**100**		**200**	
Infection Radius	4.0	10.0	4.0	10.0	4.0	10.0
1 node	9,064	12,542	96,229	134,777	203,198	267,624
4 nodes	15,510	16,826	122,886	158,256	259,487	337,783
16 nodes	17,130	20,889	190,952	214,298	381,488	434,551

Table 8. Weak scaling of the HOOVER framework v0.3 on OSSS OpenSHMEM running a simple infectious disease model with varying # of iters and infection radius.

# PEs	**Execution Time (ms)**	**Speedup Relative to Previous**
96	307,611	
384	22,259	13.82×
1536	6,568	3.39×
6144	2,418	2.72×

Table 9. Large scale tests of the HOOVER framework on the Edison supercomputer.

amount of inter-PE asynchrony that is possible as PEs will begin to throttle themselves if they detect they are beginning to exceed the number of retained timesteps.

Tables 10 and 11 clearly shows the performance benefit of reduced bytes over the wire from a reduced number of retained timesteps. It is possible that for only 100 timesteps, the PE throttling does not become a major factor.

Number of timesteps retained	Execution time (ms)
100	31,169
50	20,639
25	28,604
10	18,442
2	3,964

Table 10. Execution time as a function of retained timesteps in each sparse vector on HOOVER v0.1.

Number of timesteps retained	Execution time (ms)
100	83,020
50	78,255
25	102,503
10	65,855
2	66,070

Table 11. Execution time as a function of retained timesteps in each sparse vector on HOOVER v0.2 (with a larger dataset than the previous tests, the same dataset as was used for weak scaling in the previous section).

5.6 Remote Sparse Vector Cache Size

As a simple optimization, HOOVER supports caching of remote sparse vectors in a local PE. This optimization is particularly useful when fetching remote actors with edges shared with local actors, as many remote actors may have multiple edges shared with local actors. This cache allows us to fetch a read-only copy of the remote actor once, and then re-use it.

The remote actor cache is designed as a simple hashmap, with a fixed number of top-level buckets (keyed on the offset of the remote actor in the remote PE's list of actors) and a maximum number of entries allowed per bucket. Hence, we can expect that reducing both the number of top-level buckets and the number of entries allowed per bucket will both increase cache bucket evictions.

Table 12 illustrates that for very small numbers of top-level buckets and entries per bucket, performance suffers. However, HOOVER appears to be fairly resilient to varying values for each cache parameter, with the same performance across many different configurations.

Number of top-level buckets	Max entries per bucket	Execution time (ms)
64	64	43,620
64	16	41,276
64	4	41,505
16	64	49,284
16	16	53,973
16	4	47,930
4	64	42,128
4	16	59,965
4	4	81,023

Table 12. Execution time as a function of remote cache configuration on HOOVER v0.1.

5.7 Number of Partitions

The importance of partitions was explained in Section 4.2. In this section, we study how varying the number of partitions for the infectious disease simulator affects performance. By default, in previous experiments we have used 170×170 = 28,900 partitions formatted as a regular grid on the two-dimensional problem space. Table 13 shows how performance changes with partition configuration.

Partition Configuration	Execution time (ms)
50×50	13,090
100×100	11,047
150×150	18,875
200×200	14,472

Table 13. Execution time as a function of partition configuration for the infectious disease modeling application on HOOVER v0.1.

6 Visualization

As a helpful utility, HOOVER offers basic support for visualizing HOOVER simulations. Currently this support is limited to 2D simulations where there are only two positional attributes for each actor. The HOOVER-viz tool will plot the positions of each actor on a 2D plane for each timestep. It supports color coding each actor by either the owning PE or by some attribute on the actor.

Visualizing a HOOVER simulation is a two-step process with online and offline components. Online, HOOVER PEs generate traces of actor updates and write them to disk as CSV files. This tracing is enabled whenever the environment variable `HVR_TRACE_DUMP` is defined. Offline, the HOOVER-viz tool (which simply consists of an HTML and Javascript file) reads the CSV files into your browser and then plays back the activity on an HTML Canvas.

Figure 2 shows the HOOVER-viz tool. In the upper left corner are controls for selecting a CSV to replay, toggling how to color code actors, and resetting the simulation. In the upper right corner is a counter for the current timestep. The remainder of the canvas is devoted to plotting actors.

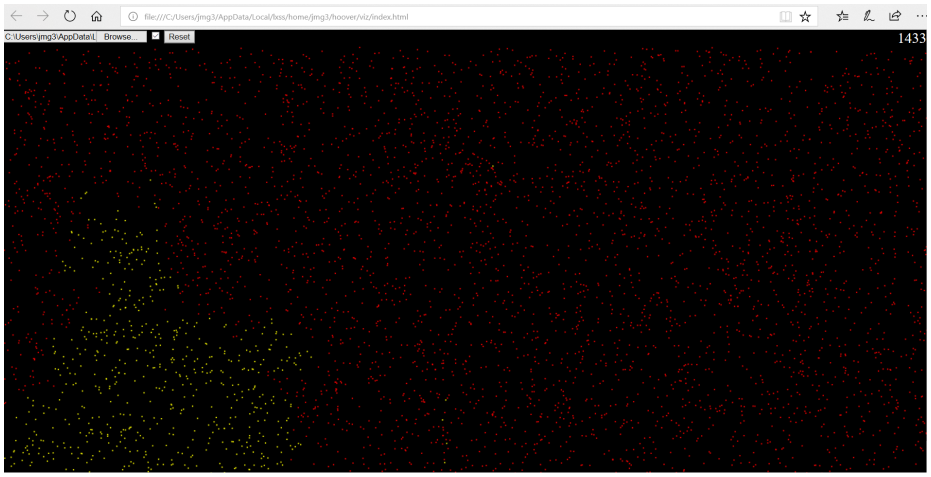
file:///C:/Users/jmg3/AppData/Local/lxss/home/jmg3/hoover/viz/index.html
C:\Users\jmg3\AppData\L
Browse...
Reset
1433

Fig. 2. Screenshot of the HOOVER-viz tool.

7 Future Work

HOOVER is an actively developed project, with several major features on its roadmap.

First, continued performance analysis and improvement is key as the project matures. Primarily, this is done through built-in profiling hooks in HOOVER that offer detailed information on time spent in different application regions. Additionally, recent work with HOOVER has begun using Jupyter notebooks to analyze and visualize performance trends in HOOVER executions, which has already proven valuable. For example, plotting histograms of overall iteration execution time against histograms of execution time spent in specific code regions uncovers visual similarities between the overall distribution and the per-region distribution of execution time. These similarities (e.g. long tails, sharp Gaussian curves, etc) offer insight into how different code regions positively or negatively impact overall execution time (both its mean and variance).

Second, explorations of both multi-threaded and multi-GPU execution are on HOOVER's roadmap. Hybrid distributed and multi-threaded execution will enable more state to fit in each PE, reducing the amount of communication needed between PEs.

Finally, we are also interested in exploring a history-less execution mode. Much of the complexity and overhead of HOOVER results from the need to keep the history of changes for each actor going back several timesteps, to ensure that remote PEs can fetch consistent data for their present without seeing into the "future". However, some applications may simply be interested in the latest state from each PE, regardless of when it was computed. This simplifies HOOVER's execution and offers new opportunities for optimizations.

In conclusion, HOOVER is a unique, distributed framework for processing of large-scale and dynamic applications that can be expressed as graph problems. Unlike other frameworks, it focuses on scale out performance and datasets that do not fit in CPU or GPU memory. HOOVER's construction on the OpenSHMEM runtime enables entirely de-coupled, asynchronous execution through RDMA without the need for interrupting the target PE. This runtime model has positive long term implications for the scalability of HOOVER, particularly as hardware platforms grow more parallel and memory access latencies become more irregular.

8 Acknowledgements

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.