HOOVER: Leveraging OpenSHMEM for High Performance, Flexible Streaming Graph Applications

Max Grossman[1], Howard Pritchard[2], Vivek Sarkar[1], Steve Poole[2]

[1]Georgia Tech

[2]Los Alamos National Laboratory

Just in case you tune out…

HOOVER: Iterative dynamic graph modeling/analysis framework.
- Be able to update/mutate graphs
- Then analyze impact those updates have had on the graph.

C/C++ library built on OpenSHMEM 1.4 – PGAS-by-design.

Emphasis on de-coupled execution – communication is one-sided and localized.

Users provide callbacks that implement application-specific functionality.

Runtime manages all computation and communication.

HOOVER sucking up your data…

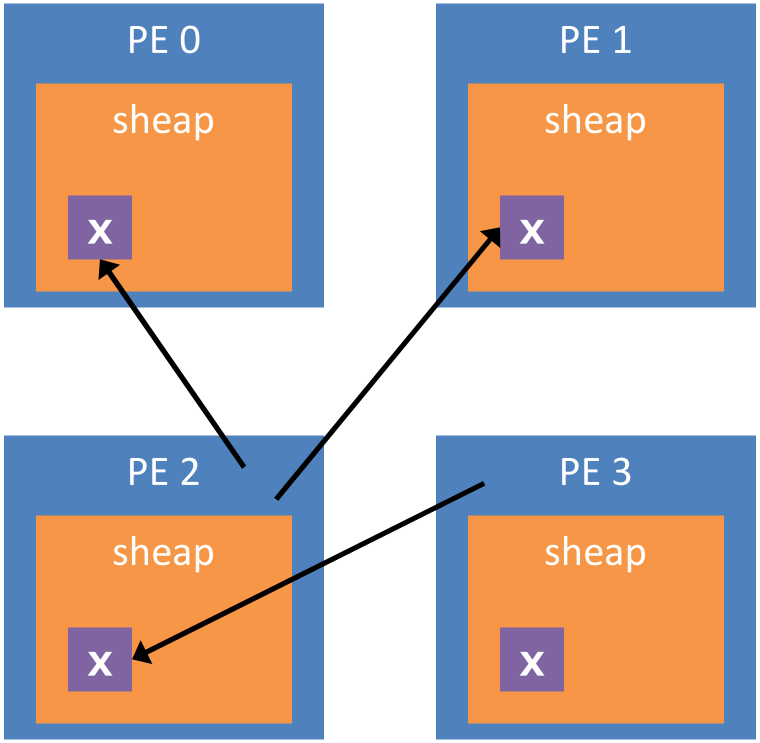
OpenSHMEM

PGAS, SPMD programming model.

Every process (i.e. PE) has a single remotely accessible heap from which objects are allocated symmetrically.

“Key feature of OpenSHMEM is that data transfer operations are one-sided”

“Allows for overlap between communication and computation to hide data transfer latencies, which makes OpenSHMEM ideal for unstructured, small/medium size data communication patterns.”



Target Class of Problems

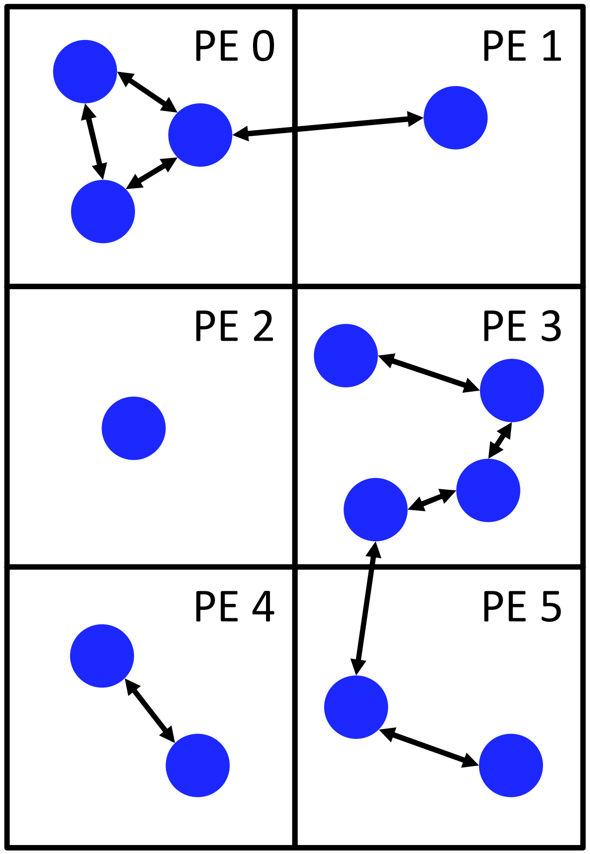
Streaming, dynamic graphs with edge and vertex additions/removals.

Connectivity is localized in the graph (few long distance edges).

- Implies loose consistency requirements.

As graph evolves over time, connectivity may grow and consistency requirements may increase.

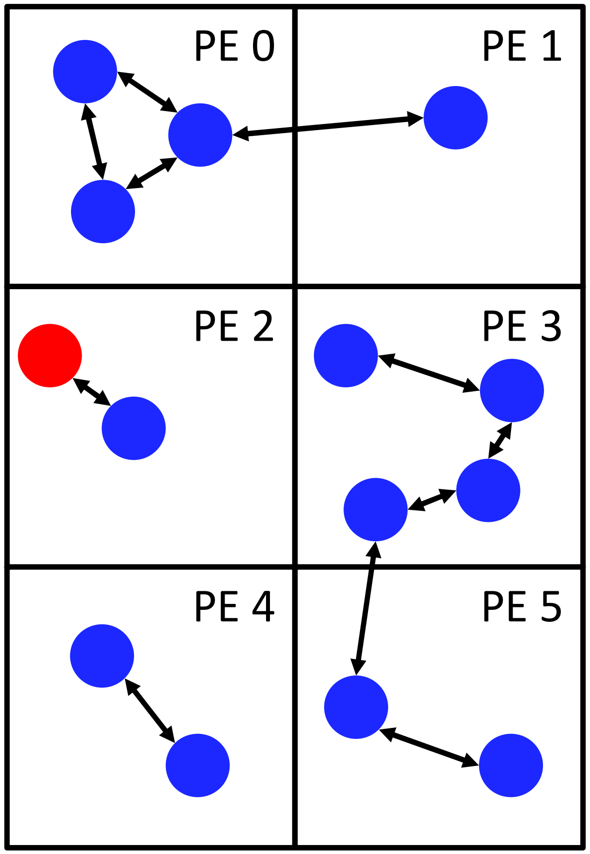
Want a system that can combine modeling and analyzing a dynamic graph with these properties.



One Challenge With Distributed Graph Frameworks

New vertex arrives.

How would this be handled in existing graph frameworks?



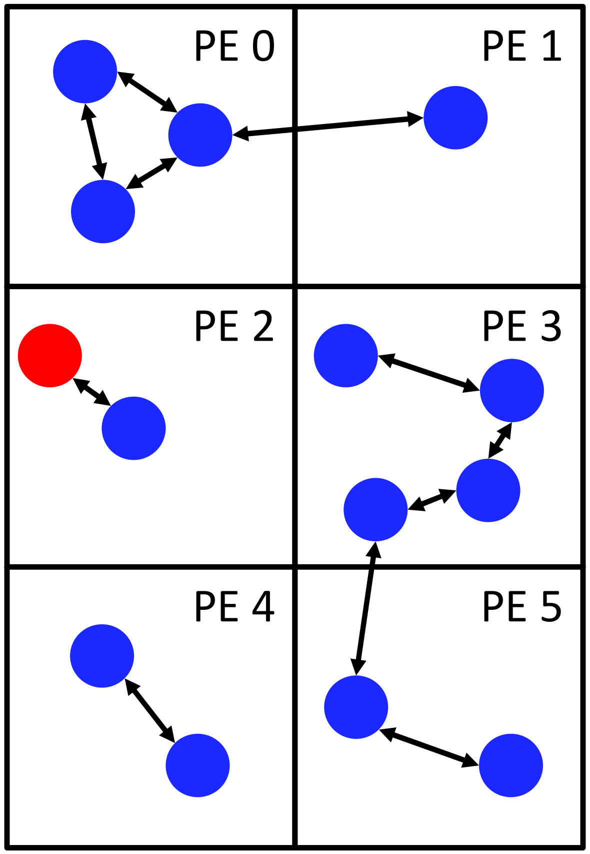
One Challenge With Distributed Graph Frameworks

New transaction arrives.

How would this be handled in existing graph frameworks?

GraphX (https://spark.apache.org/graphx/):

```
val transacts : RDD[(VertexID, ...)] = …
val new_transacts = sc.parallelize(local_new)
val next_transacts = transacts.join(new_transacts)
```



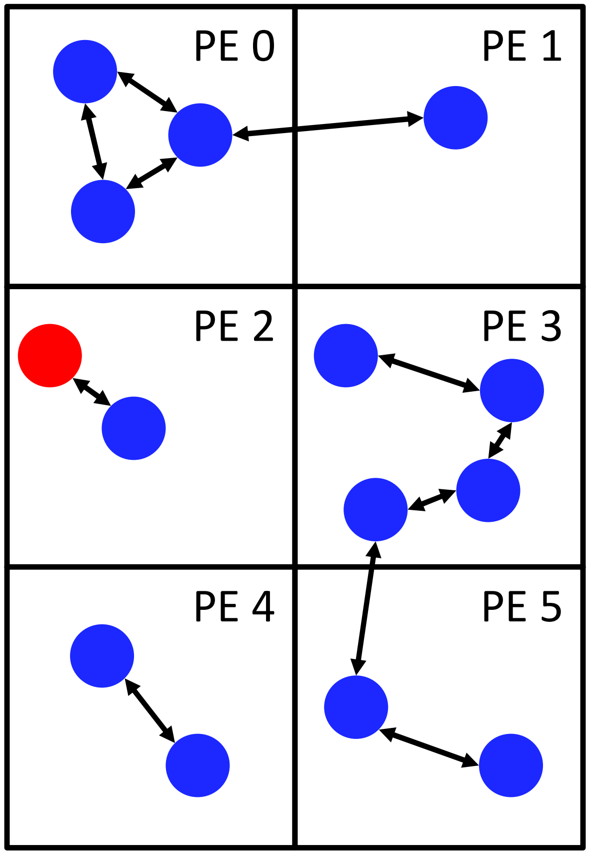
One Challenge With Distributed Graph Frameworks

GraphX (https://spark.apache.org/graphx/):

```
val transacts : RDD[(VertexID, ...)] = …
val new_transacts = sc.parallelize(local_new)
val next_transacts = transacts.join(new_transacts)
```

Problems:

- Globally bulk synchronous, implies global barriers and probably a shuffle.
- Only PE 2 really needs to be involved at this point – vertex insertion is entirely local.



Introducing HOOVER

Iterative dynamic graph modeling/analysis framework.

- Be able to update/mutate graphs
- Then analyze impact those updates have had on the graph.

C/C++ library built on OpenSHMEM 1.4 – PGAS-by-design.

Emphasis on de-coupled execution – communication is one-sided and localized.

Users provide callbacks that implement application-specific functionality.

Runtime manages all computation and communication.

HOOVER sucking up your data…

A HOOVER Example

```
void start_iteration(hvr_ctx_t ctx) {
  for (e = 0; e < n_to_add; e++)
    hvr_create_edge(rand_vert_id(), rand_vert_id(),
      BIDIRECTIONAL, ctx);
}

void update_vertex(hvr_vertex_t *vert) {
  uint64_t min_lbl = hvr_vertex_get_uint64(LBL, vert,
    ctx);

  hvr_neighbors_t neighbors;
  hvr_get_neighbors(vert, &neighbors, ctx);

  while (hvr_neighbors_next(&neighbors, &nbr)) {
    uint64_t nbr_lbl = hvr_vertex_get_uint64(LBL, nbr);
    min_lbl = MIN(min_lbl, nbr_lbl);
  }

  hvr_vertex_set_uint64(LBL, min_lbl, vert);
}
```

Example of connected components as a label propagation problem.

start_iteration: insert logic at the start of internal runtime iterations.

update_vertex: called on every vertex to recompute its state.

hvr_create_edge: Create an edge between any two vertices

hvr_get_neighbors: Get an iterator over the neighboring vertices.

hvr_vertex_set/get: Update/fetch attributes on vertices.

A HOOVER Example

```
int main(int argc, char **argv) {
  shmem_init();

  hvr_ctx_t hvr_ctx;
  hvr_ctx_create(&hvr_ctx);

  for (int v = 0; v < nvertices_per_pe; v++) {
    hvr_vertex_t *vert = hvr_vertex_create(hvr_ctx);

    // Initially each vertex is its own component
    hvr_vertex_set_uint64(LBL, vert->id, vert);
  }

  hvr_init(update_vertex, start_iteration,
    time_limit_s, 1, hvr_ctx);

  hvr_body(hvr_ctx);

  hvr_finalize(hvr_ctx);

  shmem_finalize();
}
```

`hvr_ctx_create`: Allocate a context for the new HOOVER job

`hvr_vertex_create`: Allocate a new vertex.

`hvr_init`: Set up the HOOVER problem by providing callbacks and other parameters.

`hvr_body`: Launch the HOOVER problem.

`hvr_finalize`: Wait for all PEs to complete and clean up runtime resources.

HOOVER Feature Set

The Usual

Callback-based application logic.

Iteration-driven execution – run application-specific logic once per runtime iteration (`start_iteration`).

Data-driven execution – vertices are updated when needed based on changes to neighborhood (`update_vertex`).

Support explicit message passing between vertices (`hvr_send_msg`).

Support creating and deleting vertices and edges.

Support set/get on vertex attributes.

The Unusual

Fully decoupled execution by default.

Ability to force lockstep execution between PEs (`update_coupled`).

PEs may exit the simulation arbitrarily, benefit of PGAS (`should_terminate`).

Support both implicit and explicit edge creation.

HOOVER API

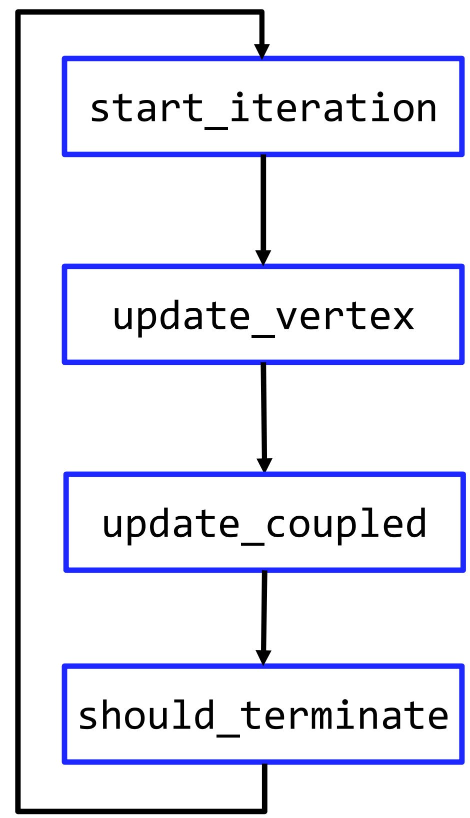
Like other frameworks, callbacks are used to implement application-specific functionality.

`start_iteration`: Hook for logic to be executed at the start of every runtime iteration.

`update_vertex`: Given a vertex and its neighborhood, update its attributes.

`update_coupled:` Update the value shared with coupled PEs.

`should_terminate`: Called at end of iteration, check if this PE will exit.



HOOVER Runtime

OpenSHMEM PE

Manage all communication of vertices/edges with remote PEs.

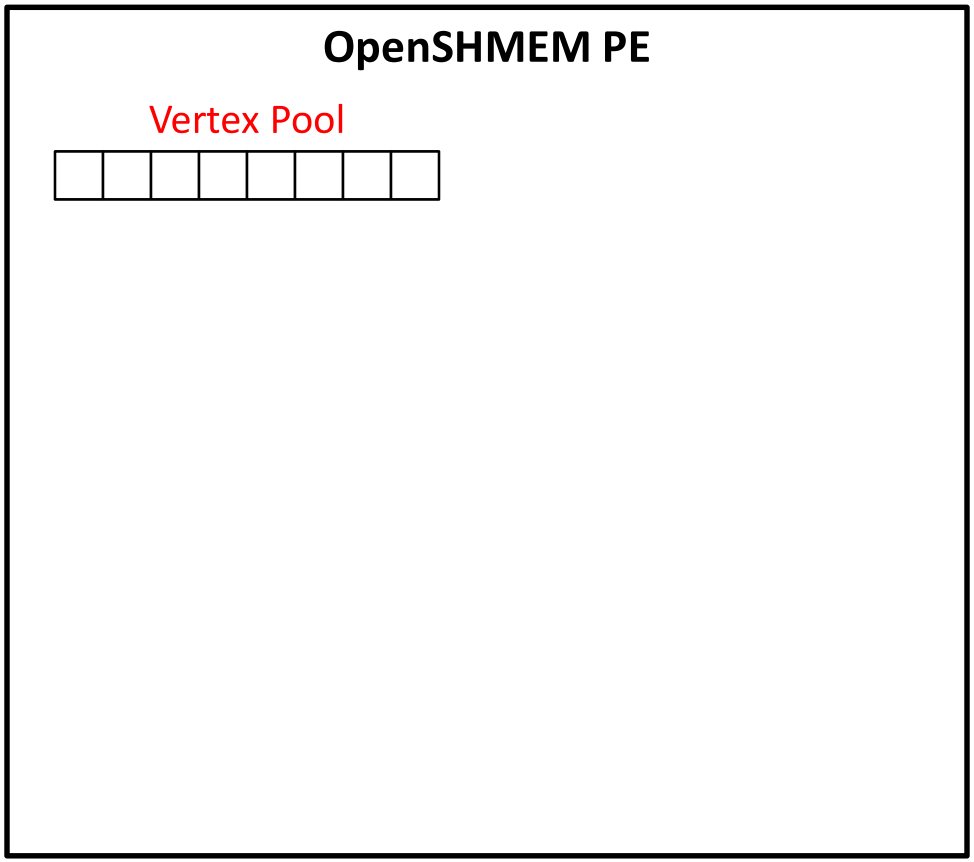
- Relies heavily on distributed mailboxes for asynchronous communication between PEs.

Schedule all local computation needed to update graph state.

Respond to requests from user-level application code.

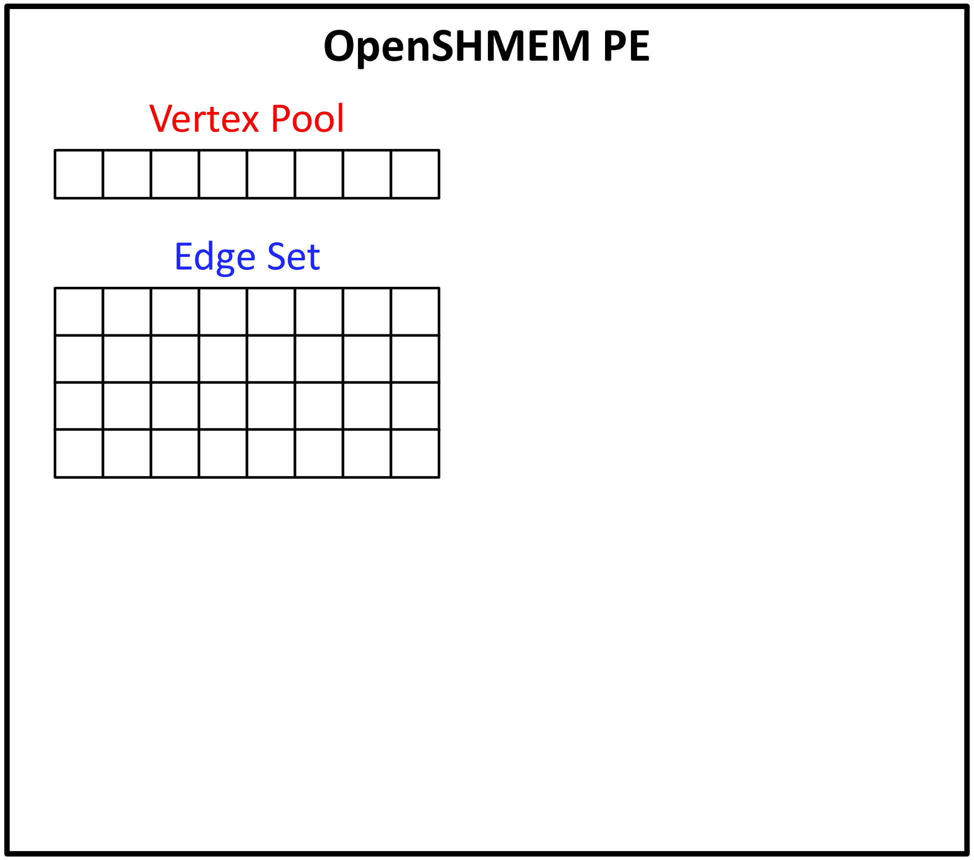
Heavily event-driven: message arrives in mailbox or user mutates some state, triggers downstream work.

HOOVER Runtime



Vertex Pool: statically sized pool of all vertices stored on this PE. Includes both locally-owned vertices and mirrored remotely-owned vertices. (hybrid hashmap/linked lists)

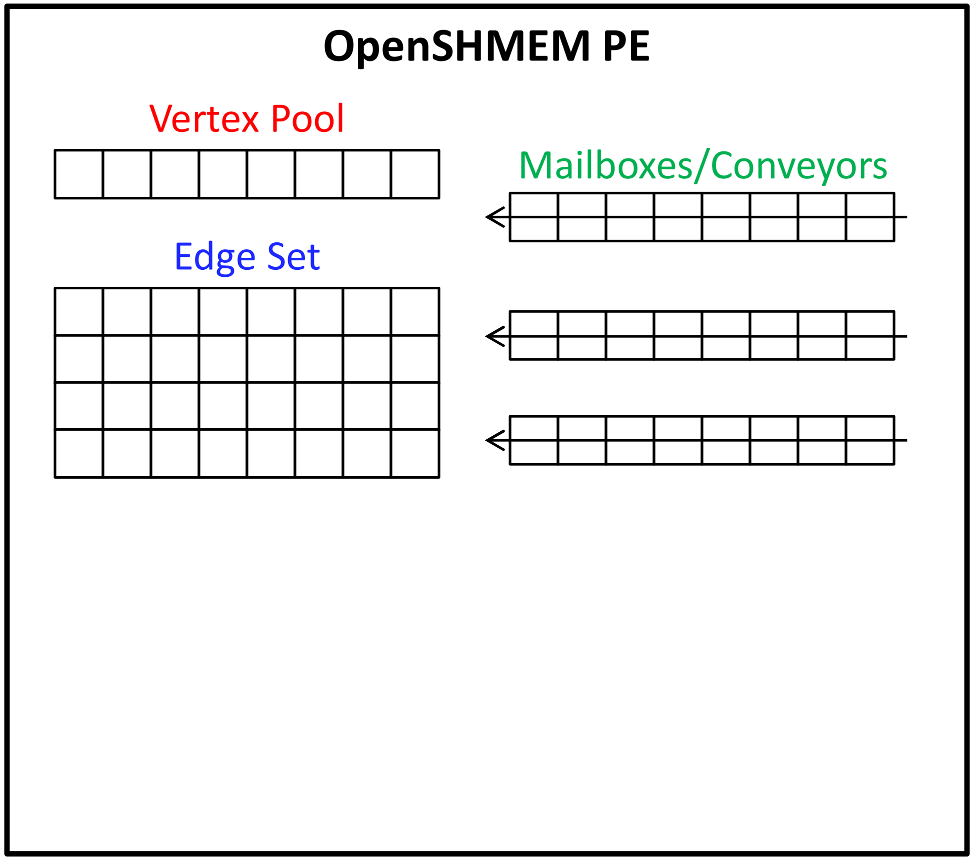
HOOVER Runtime



Vertex Pool: statically sized pool of all vertices stored on this PE. Includes both locally-owned vertices and mirrored remotely-owned vertices. (hybrid hashmap/linked lists)

Edge Set: CSR matrix storing edge information for each locally-stored vertex.

HOOVER Runtime

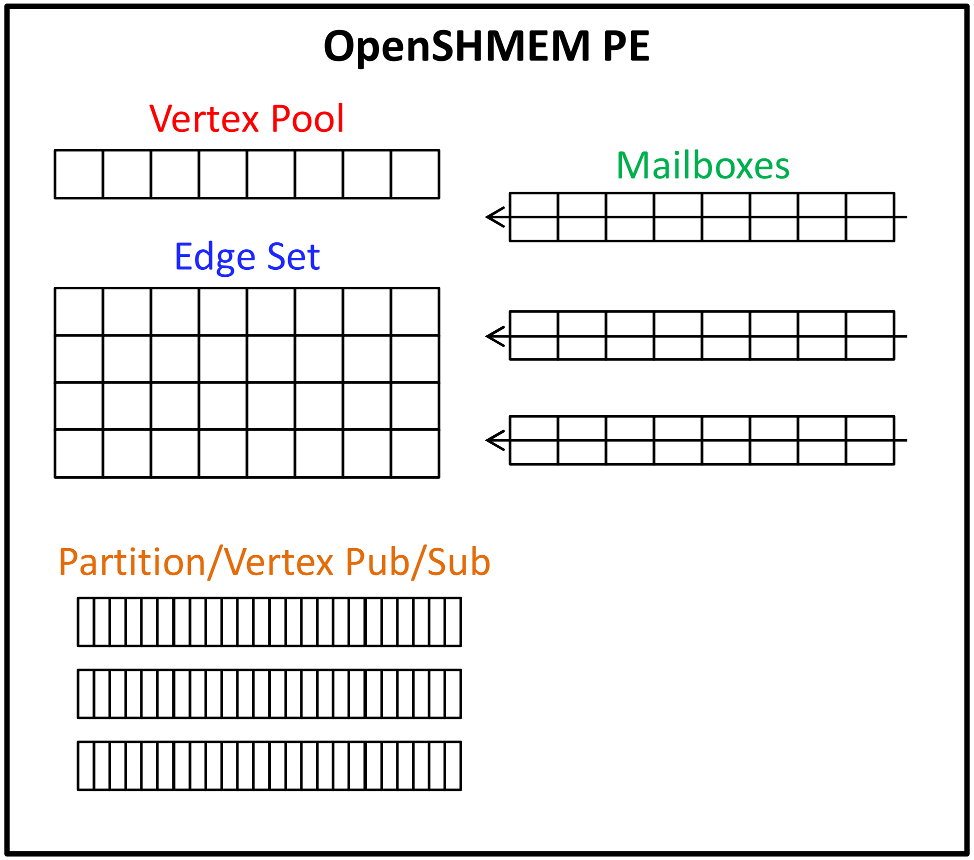


Vertex Pool: statically sized pool of all vertices stored on this PE. Includes both locally-owned vertices and mirrored remotely-owned vertices. (hybrid hashmap/linked lists)

Edge Set: CSR matrix storing edge information for each locally-stored vertex.

Mailboxes/Conveyors: Primary inter-PE communication protocol, used to share updates to graph and other metadata.

HOOVER Runtime



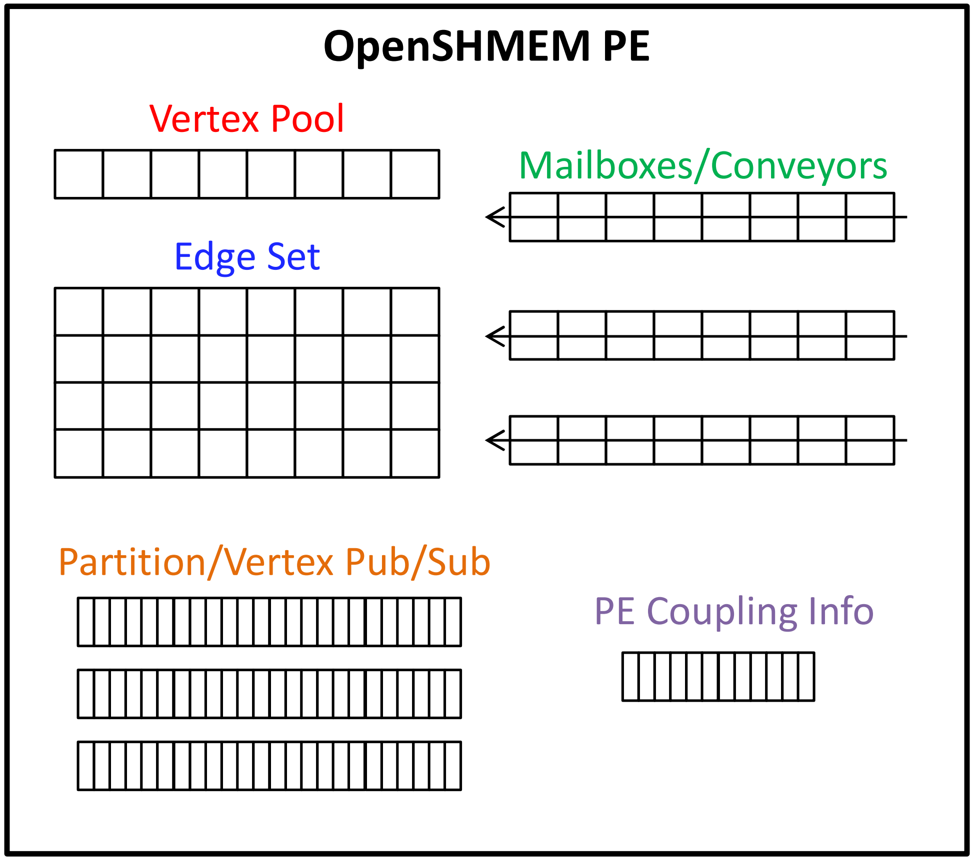
Vertex Pool: statically sized pool of all vertices stored on this PE. Includes both locally-owned vertices and mirrored remotely-owned vertices. (hybrid hashmap/linked lists)

Edge Set: CSR matrix storing edge information for each locally-stored vertex.

Mailboxes: Primary inter-PE communication protocol, used to share updates to graph and other metadata.

Partition/Vertex Pub/Sub: Producer-consumer metadata on graph partitions and individual vertices that dictate which PEs receive updates. (bit vectors, AVL trees, etc).

HOOVER Runtime



Vertex Pool: statically sized pool of all vertices stored on this PE. Includes both locally-owned vertices and mirrored remotely-owned vertices. (hybrid hashmap/linked lists)

Edge Set: CSR matrix storing edge information for each locally-stored vertex.

Mailboxes/Conveyors: Primary inter-PE communication protocol, used to share updates to graph and other metadata.

Partition/Vertex Pub/Sub: Producer-consumer metadata on graph partitions and individual vertices that dictate which PEs receive updates. (bit vectors, AVL trees, etc).

PE Coupling Info: Structures needed to safely handle coupling requests.

Performance Evaluation

Experiments are run on Cori (CraySHMEM 7.7.6) – 1 PE per core (32 PEs per node).

Two streaming graph kernels, measuring performance relative to Apache Flink:

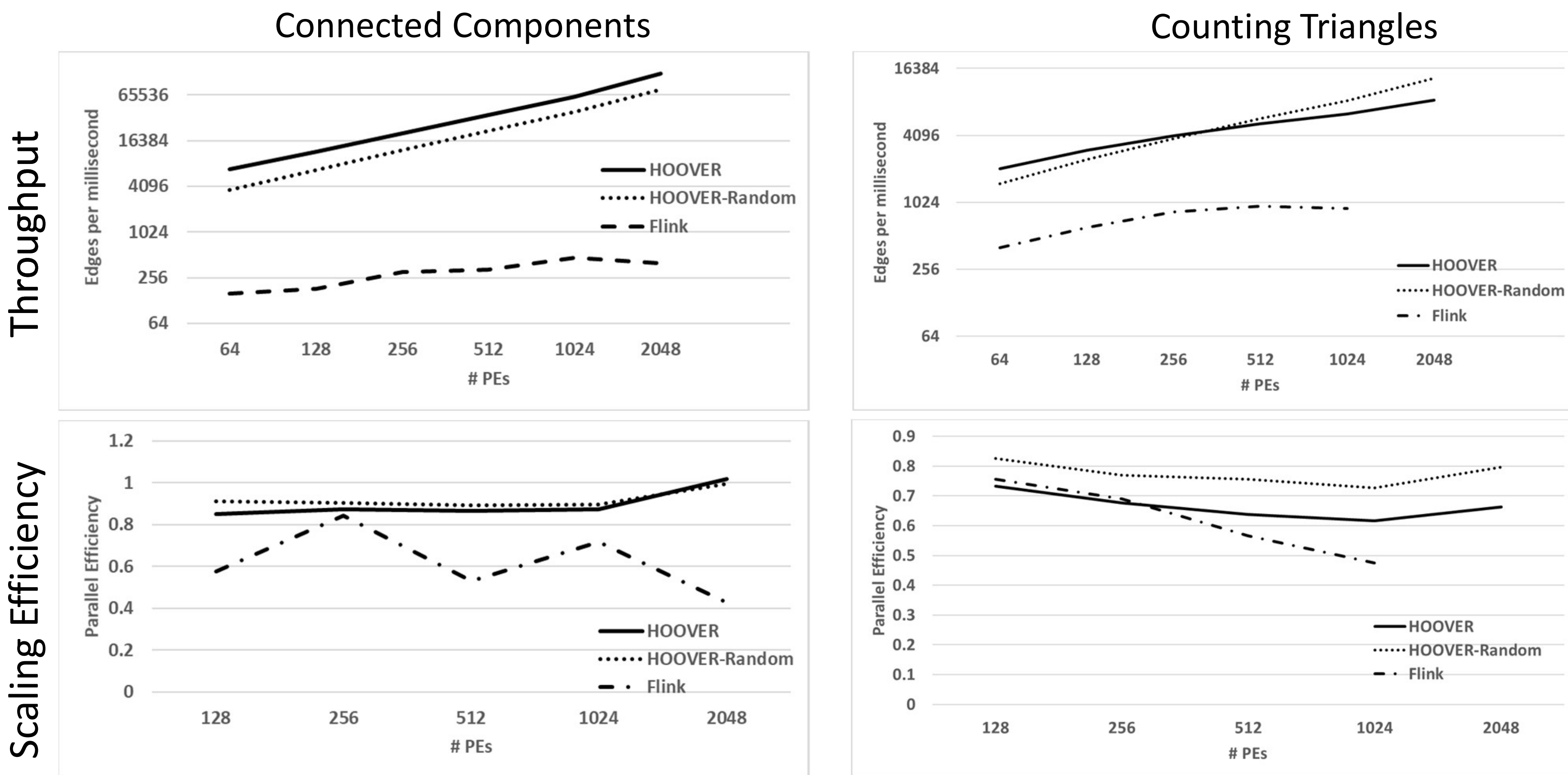
- *Connected Components*
- *Triangle Counting*

Two mini-apps, measuring strong scaling of HOOVER:

- *Graph-based anomaly detection* – Stream random vertices into the graph, search for normative patterns, identify anomalies as patterns that are similar but not identical to normative.
- *Community detection* – Clique percolation method

Other implemented applications include infectious disease modeling, n-body simulations, graph convolutional networks, mosquito-borne illness modeling.

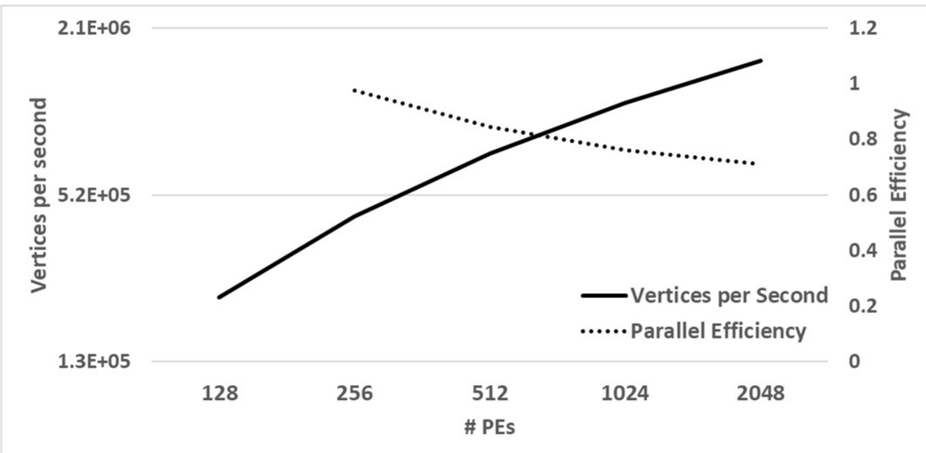
Performance Evaluation



Streaming Connected Components: Similar benchmarking studies in the past report 1.1 million edges per second, versus 17.7 million in the worst case and 371.5 million edges per second in the best case for HOOVER and scalability beyond that (Berry et al, 2013).

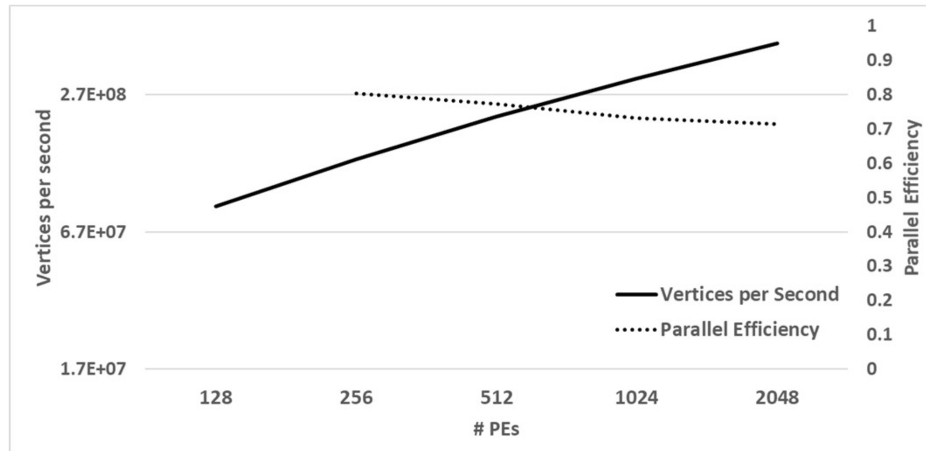
Performance Evaluation

Graph-Based Anomaly Detection



Based on "Mining for Structural Anomalies in Graph-based Data", William Eberle and Lawrence Holder.

Community Detection



Clique Percolation Method

Run each application for a fixed number of seconds and then measure average throughput over all of execution.

Conclusions

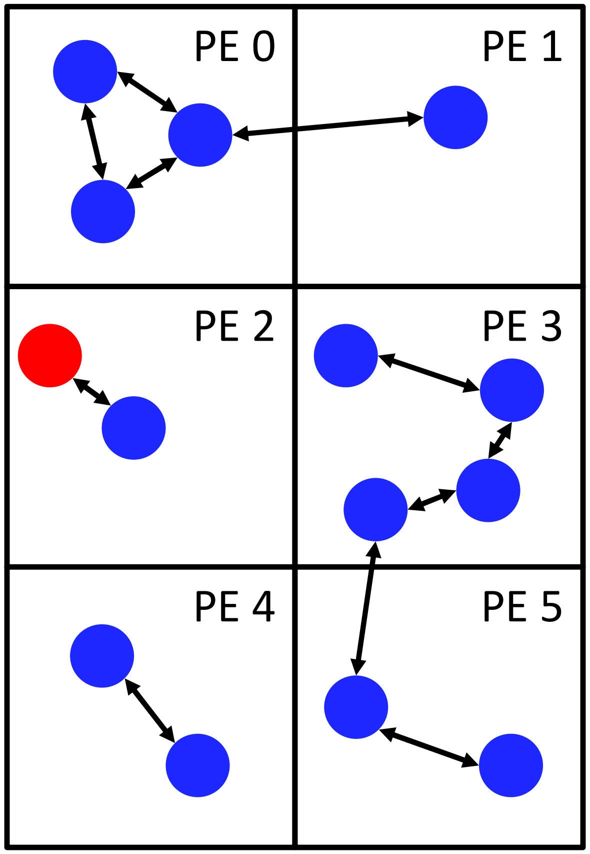
HOOVER: an iterative dynamic graph modeling and analysis framework.

Emphasis on de-synchronized, de-coupled execution – one-sided and PGAS by default.

This adds complexity to the runtime.

But enables scalability in a way that bulk synchronous models can't.

Github: https://github.com/agrippa/hoover
Contact: max.grossman@gatech.edu



Acknowledgements