Pacific Northwest
NATIONAL LABORATORY

Chapel Graph Library (CGL)

June 22, 2019

Louis Jenkins
Marcin Zalewski

U.S. DEPARTMENT OF ENERGY BATTELLE

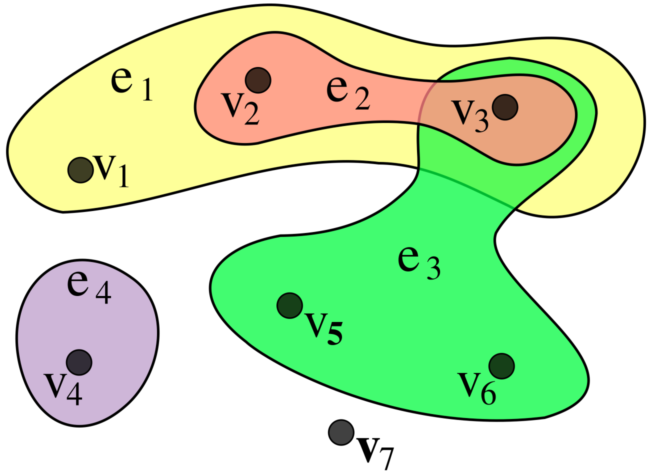
PNNL is operated by Battelle for the U.S. Department of Energy

Pacific Northwest
NATIONAL LABORATORY

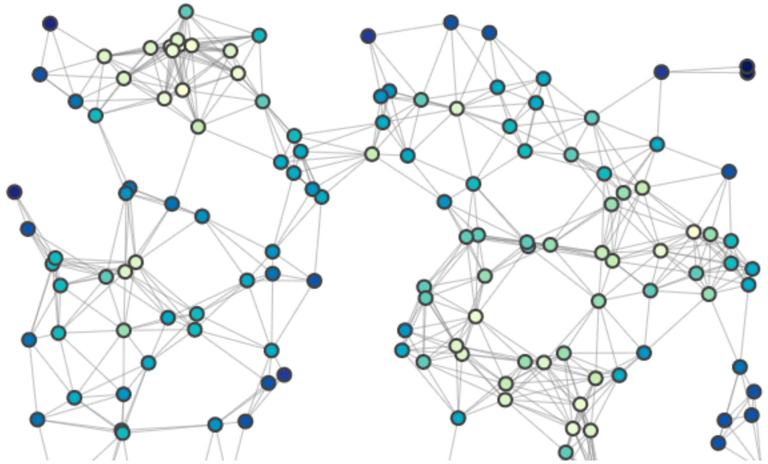
What Is This Talk About

- Chapel HyperGraph Library (CHGL)
  - **Hypergraph** algorithms in Chapel
  - Open-Source project
  - Global-View distributed data structures
  - Chapel Aggregation Library (CAL)
    - ✓ Support for fine-grained computation

CHGL
CHAPEL HYPERGRAPH LIBRARY



- How to apply this experience to **graphs**?
  - Chapel Graph Library (CGL)
  - How much of CHGL can be reused?
    - ✓ Maybe CHGL == CGL
  - What are the performance bounds?
    - ✓ Is there a penalty for using hypergraph data structures to represent graphs?
    - ✓ What is the abstraction penalty in general?
    - ✓ Chapel performance issues?

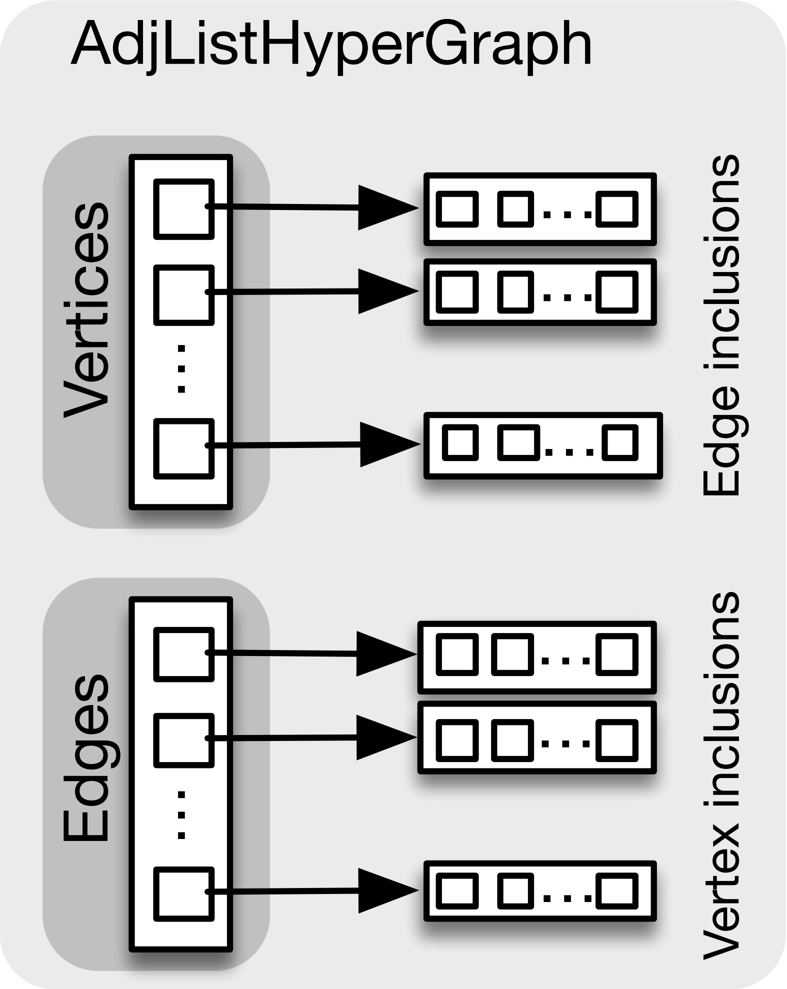


Schedule

- Background
  - Chapel HyperGraph Library (CHGL)
  - Chapel Aggregation Library (CAL)
  - Global-View Distributed Data Structures effort
- Graphs as 2-Uniform Hypergraphs
- Triangle Counting Benchmark
  - Analysis of performance results
  - Profile and report performance issues
  - Comparison to UPC++
- Conclusion

Background: Chapel HyperGraph Library (CHGL)

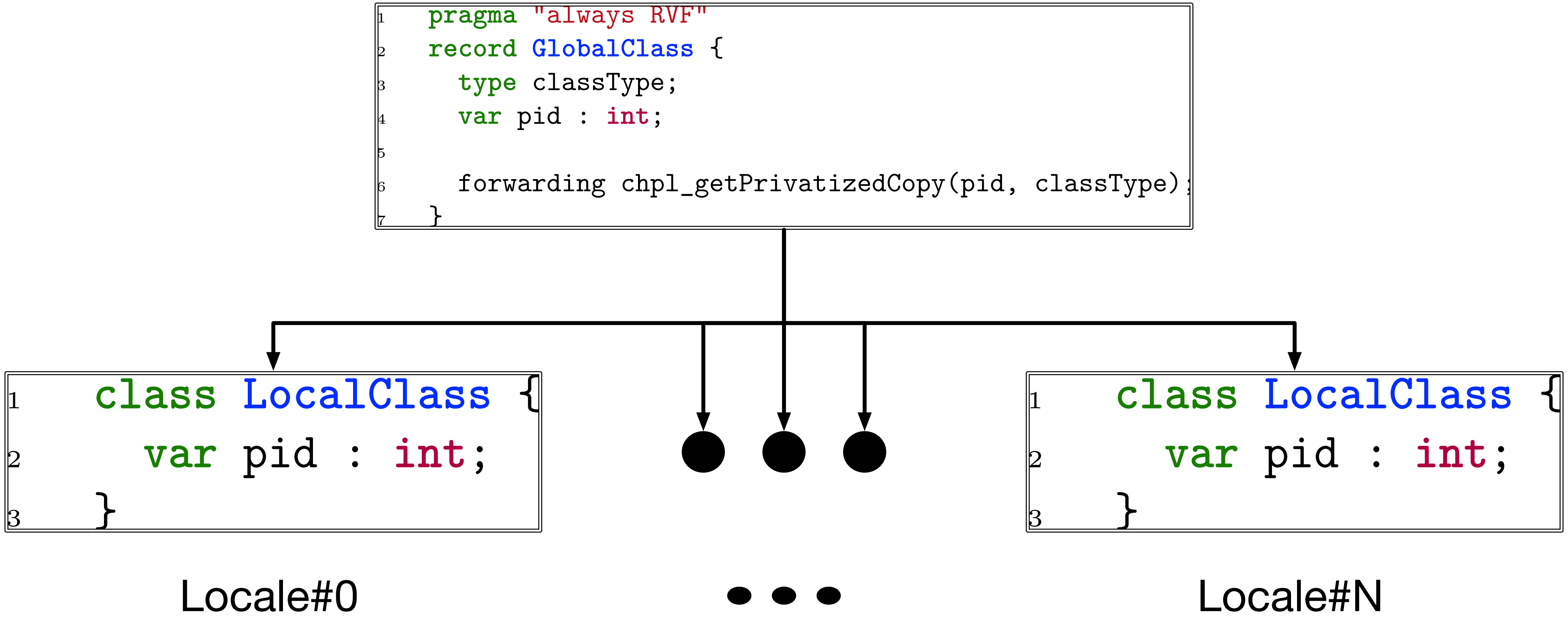
- One of the few software packages specifically targeted at hypergraphs
- Dual Hypergraph (AdjListHyperGraph)
  - Vertices have an incidence list of hyperedges they are incident in
  - Hyperedges have an incidence list of vertices that are incident in it
- Provides a good initial set of methods and data structures
  - Hypergraph metrics
  - Hypergraph generation algorithms
  - Hypergraph algorithms (s-walks, s-connected components, …)
- Generic design: high-level, conceptual, write once



Pacific Northwest
NATIONAL LABORATORY

Background: Chapel HyperGraph Library (CHGL)

- Global-View Distributed Data Structure
  - Offers semantics equivalent to Chapel's distributed arrays via *privatization*
    - ✓ Creates a clone of a class instance on each *locale* (compute node)
    - ✓ All privatized instances are obtained in O(1) time via pid (offset into runtime privatization table)
    - ✓ Wraps pid and type into a *record* and *forwards* method calls and field access to privatized instance



Background: Chapel HyperGraph Library (CHGL)

```
var graph = new AdjListHyperGraph(
  numVertices=1024, verticesMapping = new Cyclic(startIdx=0),
  numEdges = 1024, edgesMapping = new Block(boundingBox={0..#1024})
);
forall v in graph.getVertices() do
  forall e in graph.getEdges() do
    if randomSelection() then
      graph.addInclusion(v,e);
```

- Global-View Distributed Data Structure
  - Offers semantics equivalent to Chapel’s distributed arrays via *privatization*
    - ✓ Creates a clone of a class instance on each *locale* (compute node)
    - ✓ All privatized instances are obtained in O(1) time via pid (offset into runtime privatization table)
    - ✓ Wraps pid and type into a `record` and `forwards` method calls and field access to privatized instance
    - ✓ Abstracts and optimizes locality from the user; usable from within all language constructs

Background: Chapel HyperGraph Library (CHGL)

```
var vPropMap = new PropertyMap(string); // Assume is filled
var ePropMap = new PropertyMap(string); // Assume is filled
var graph = new AdjListHyperGraph(
  vPropMap, verticesMapping = new Cyclic(startIdx=0),
  ePropMap, edgesMapping = new Block(boundingBox={0..#ePropMap.size})
);
// Add between vertices and edges
graph.addInclusion(
   vPropMap.getProperty("Hello"), ePropMap.getProperty("World")
);
forall v in graph.getVertices() {
   var vProp = graph.getProperty(v);
   forall e in graph.incidence(v) {
      var eProp = graph.getProperty(e);
      // do something with vProp and eProp
   }
}
```

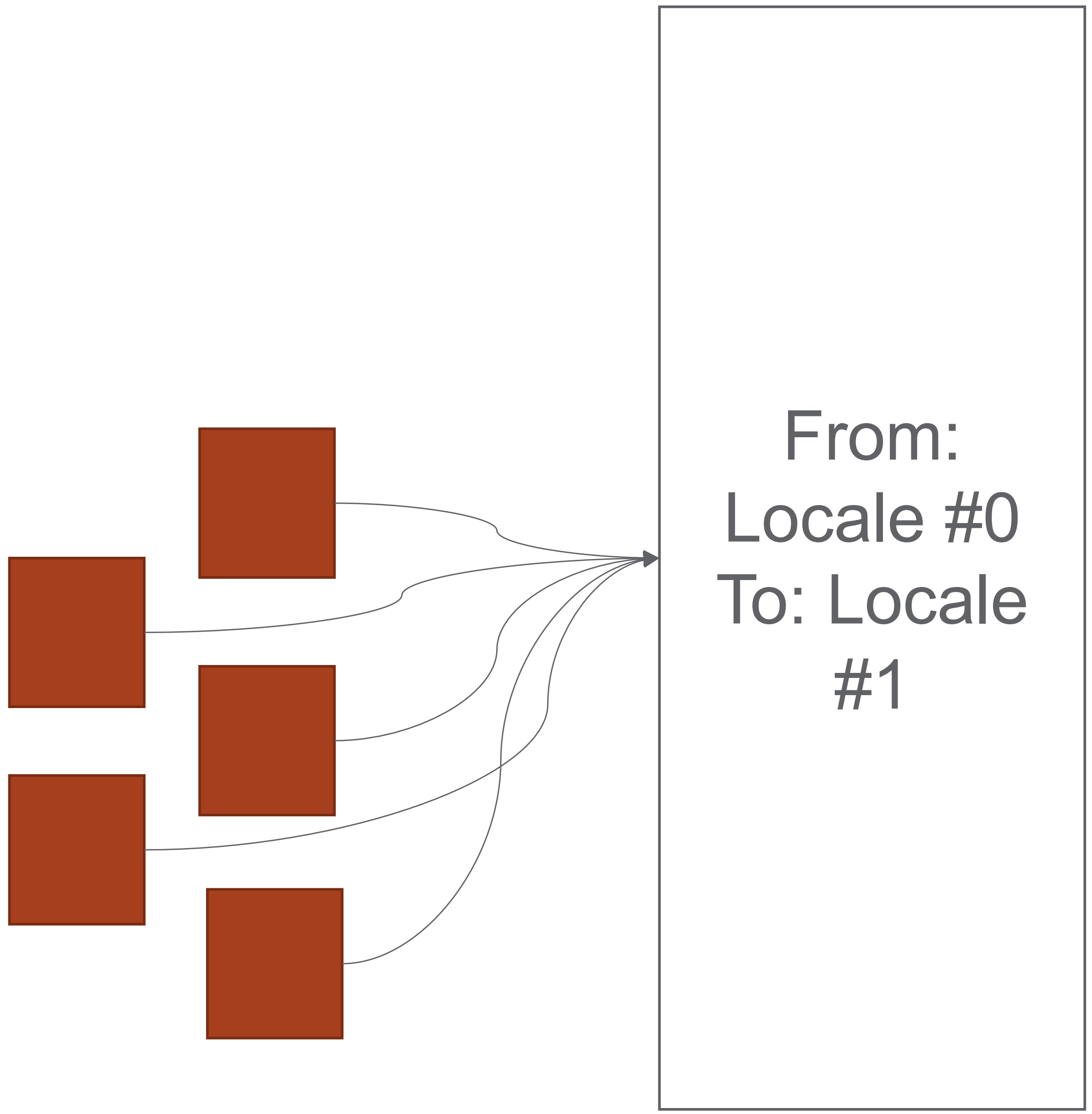
- Dual **Property** Hypergraph (AdjListHyperGraph)
  - Vertices and Hyperedges can be associated to user-defined properties

Background: Chapel Aggregation Library (CAL)

- Aggregation – “Collect individual units of data to be sent in batch”

Background: Chapel Aggregation Library (CAL)

- Aggregation – "Collect individual units of data to be sent in batch"

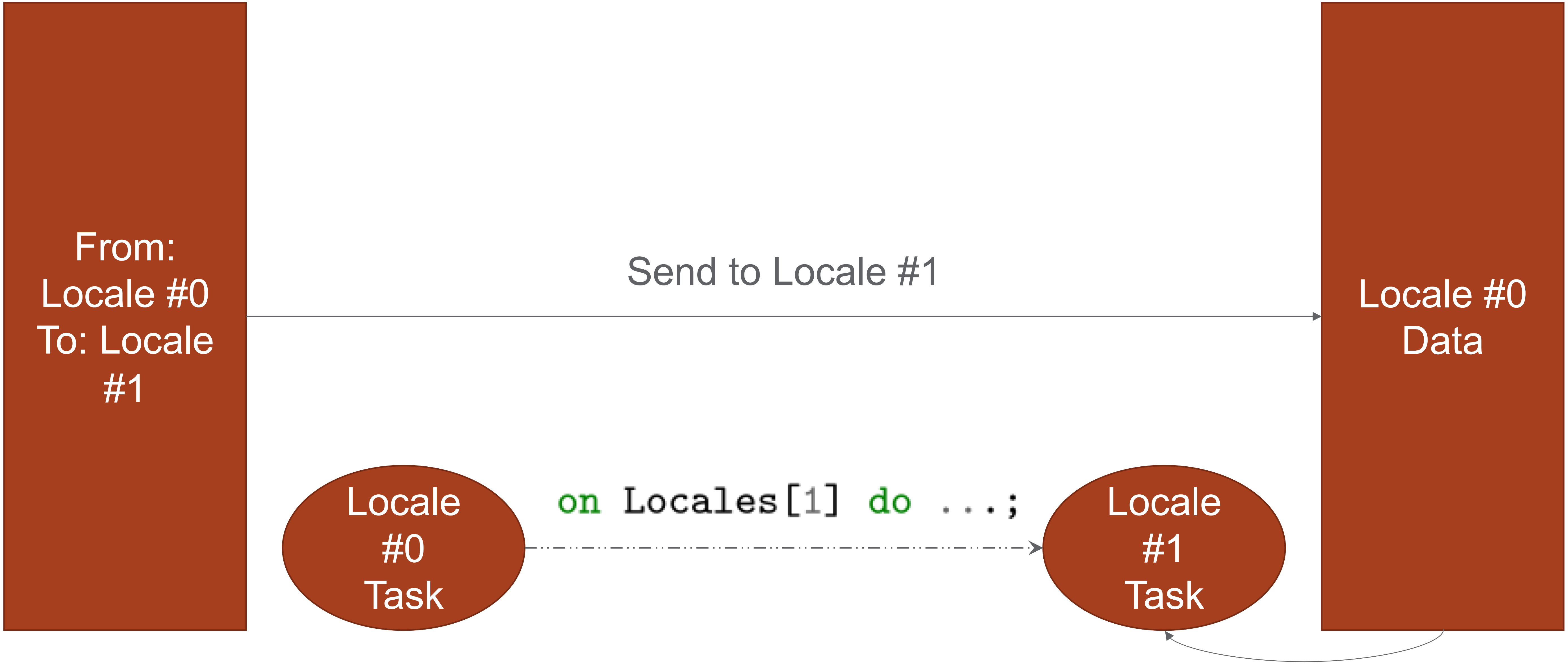


Background: Chapel Aggregation Library (CAL)

- Aggregation – "Collect individual units of data to be sent in batch"

Background: Chapel Aggregation Library (CAL)

- Aggregation – "Collect individual units of data to be sent in batch"



Global-View Distributed Data Structures All Working Together

- Global-View Distributed Data Structures working together…
  - Chapel Arrays
  - Aggregator
  - AdjListHyperGraph
- What's next…
  - Composition!

```
// Find largest degree of all vertices in distributed graph
var N = max reduce [v in graph.getVertices()] graph.degree(v);
// Histogram is cyclically distributed over all locales
var histogramDomain = {1..N} dmapped Cyclic(startIdx=1);
var histogram : [histogramDomain] atomic int;

// Aggregate increments to histogram
var aggregator = new Aggregator(int);
forall v in graph.getVertices() {
  const deg = graph.degree(v);
  const loc = histogram[deg].locale;
  var buffer = aggregator.aggregate(deg, loc);
  if buffer != nil {
    on loc do [deg in buffer] histogram[deg].add(1);
    buffer.done();
  }
}

// Flush
forall (buf, loc) in aggregator.flush() {
  on loc do [deg in buf] histogram[deg].add(1);
  buffer.done();
}
```

Pacific Northwest NATIONAL LABORATORY

2-Uniform Hypergraphs

- Goals
  - Implement a graph on top of a hypergraph
    - ✓ Restrict all in-use hyperedges to have exactly two vertices incident in them
    - ✓ Reuse and recycle as much of the hypergraph as possible to save time and effort
  - Implement graph algorithms with said graph
    - ✓ Measures overhead of such approach
- Implementation…
  - Static graph that requires number of edges to be known in advance
    - ✓ Uses an atomic counter and aggregation to 'claim' edges
  - Maintains a cache of vertex adjacency lists
    - ✓ Populated eagerly, eliminates overhead of having to 'walk' hyperedges

```
var graph = new Graph(
  numVertices = 1024, verticesMapping = new Cyclic(startIdx=0),
  numEdges = 1024, edgesMapping = new Block(boundingBox={0..#1024})
);
graph.addEdge(0..#1024 by 2,0..#1024 by 2 align 2); // (1,2), (3,4), ...
forall (v1,v2) in graph do writeln(v1, " is connected to ", v2);
```

Benchmark – Triangle Counting

- 2-Uniform Graph vs Minimal Implementation
  - Measure overhead of abstraction…
- Why Triangle Counting?
  - Simple enough example to be implemented with just distributed arrays and vectors

Minimal Graph

```
var numTriangles : int;
forall v in A.domain with (+ reduce numTriangles) do
  for u in A[v] do
    if v < u then
      numTriangles += A[v].intersectionSize(A[u]);
numTriangles /= 3;
```

2-Uniform Graph

```
var numTriangles : int;
forall v in graph.getVertices() with (+ reduce numTriangles) do
  for u in graph.neighbors(v) do
    if v < u then
      numTriangles += graph.intersectionSize(v,u);
numTriangles /= 3;
```

Pacific Northwest
NATIONAL LABORATORY

Computing Intersection Sizes – Locality Optimizations

- STL faithful implementation
- Uses locality optimizations…
  - *local* blocks get rid of locality checks
  - Explicitly copy *both domain and array* if they are remote
  - ~2 orders of magnitude performance improvement!

```
proc intersectionSize(A : [] ?t, B : [] t) {
  if isLocalArray(A) && isLocalArray(B) {
    return _intersectionSize(A, B);
  } else if isLocalArray(A) && !isLocalArray(B) {
    const _BD = B.domain; // Make by-value copy so domain is not remote.
    var _B : [_BD] t = B;
    return _intersectionSize(A, _B);
  } else if !isLocalArray(A) && isLocalArray(B) {
    const _AD = A.domain; // Make by-value copy so domain is not remote.
    var _A : [_AD] t = A;
    return _intersectionSize(_A, B);
  } else {
    const _AD = A.domain; // Make by-value copy so domain is not remote.
    const _BD = B.domain;
    var _A : [_AD] t = A;
    var _B : [_BD] t = B;
    return _intersectionSize(_A, _B);
  }
}
```

```
proc _intersectionSize(A : [] ?t, B : [] t) {
  var match : int;
  local {
    var idxA = A.domain.low;
    var idxB = B.domain.low;
    while idxA <= A.domain.high && idxB <= B.domain.high {
      const a = A[idxA];
      const b = B[idxB];
      if a == b {
        match += 1;
        idxA += 1;
        idxB += 1;
      }
      else if a > b {
        idxB += 1;
      } else {
        idxA += 1;
      }
    }
  }
  return match;
}
```

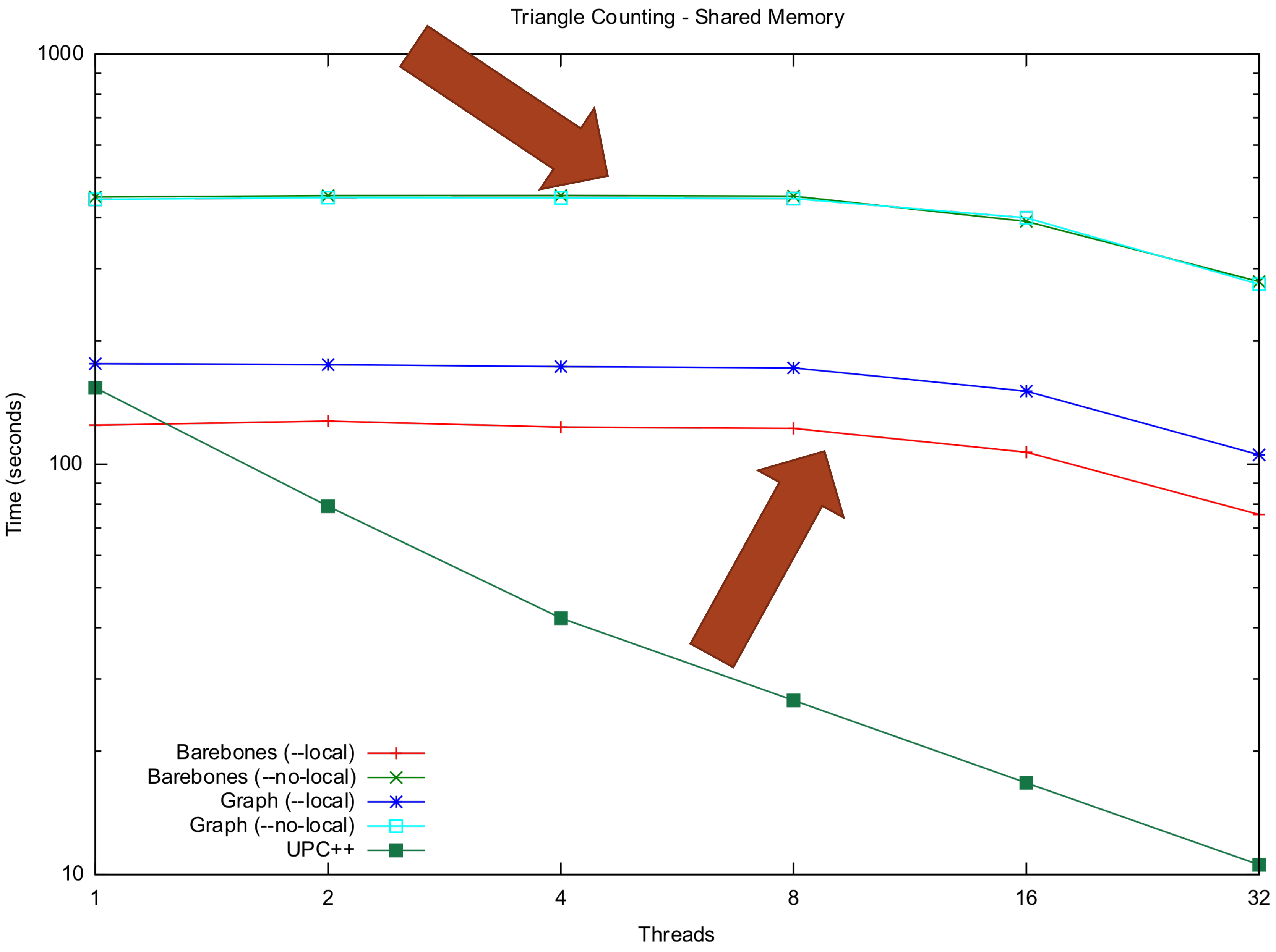
Disclaimer: UPC++ vs Chapel Performance

Performance results for Chapel were preliminary and the cause for the ‘plateau effect’ mentioned in the next slides are due to a severe load imbalance caused by the dataset. While both Chapel and UPC++ implementations act on the same data, UPC++ cyclically distributes the data for each rank, while Chapel chunks up the data in a way that would resemble a ‘block’ distribution via ‘forall’. After implementing a round-robin iteration scheme, Chapel was **as fast as UPC++** under *--local and 2 – 3x slower under --no-local.* In distributed memory, Chapel was just 3 – 5x slower.

Pacific Northwest NATIONAL LABORATORY

Triangle Counting – Shared Memory

- 650K Vertices, 32M Edges (synthetic kronecker generated graph)
- Locality checks overhead
  - *After* locality optimizations
- Plateau Effect?
  - No shared-memory scalability
    - ✓ Why? (next slide)



Explanation for ‘Plateau Effect’ (1 thread)

GOOD

```
Samples: 33K of event 'cycles:ppp', Event count (approx.): 448125821878
  Children      Self  Parent symbol
-  100.00%   100.00%  [other]
   - 95.06% 0
      - 93.21% wrapcoforall_fn_chpl13
         - 93.17% coforall_fn_chpl13.constprop.450
            + 91.85% intersectionSize_chpl4
              0.39% this3
              0.08% remove4
              0.08% deinit8.isra.193
              0.07% chpl_localeID_to_locale
              0.04% dsiGetBaseDom
              0.04% localeIDtoLocale2
              0.03% chpl_count_help2.constprop.559
              0.02% _new13.constprop.543
           0.01% intersectionSize_chpl4
           0.01% chpl_je_free
           0.01% this3
           0.00% chpl_track_malloc
           0.00% _new13.constprop.543
           0.00% remove4
           0.00% chpl_track_free
           0.00% deinit8.isra.193
        0.29% remove3
        0.22% chpl_memhook_check_pre
      + 0.22% _new13.constprop.543
        0.20% chpl_je_malloc
        0.14% this3
        0.13% _delete_arr
        0.12% chpl_je_free
        0.09% deinit8.isra.193
        0.09% chpl_track_malloc
        0.08% chpl_track_free
      + 0.06% append_chpl
        0.06% chpl__auto_destroy_DefaultRectangularDom
        0.04% _delete_dom3
        0.03% dsiLinksDistribution2
        0.02% chpl__auto_destroy_ArrayViewSliceArr
        0.01% chpl_memhook_check_post
        0.01% dsiDestroyArr
        0.01% chpl_je_tcache_event_hard
      + 0.01% coforall_fn_chpl11.isra.420.constprop.467
        0.01% dsiDestroyDom
        0.01% decEltCountsIfNeeded2
        0.00% 0xffffffff81524320
```

Pacific Northwest
NATIONAL LABORATORY

Explanation for ‘Plateau Effect’ (2 threads)

BAD

BAD

```
Samples: 69K of event 'cycles:ppp', Event count (approx.): 949639489519
  Children      Self  Parent symbol
-  100.00%   100.00%  [other]
   - 45.42% wrapcoforall_fn_chpl12
      + 44.63% intersectionSize_chpl4
        0.30% this3
        0.05% remove4
        0.03% localeIDtoLocale2
        0.03% deinit8.isra.193
        0.03% chpl_localeID_to_locale
        0.02% _new13.constprop.543
        0.01% chpl_count_help2.constprop.559
        0.01% dsiGetBaseDom
   - 29.08% 0
        6.11% qthread_master
      + 5.44% qt_scheduler_get_thread
        3.94% qt_swapctxt
        3.48% qthread_yield_
        2.67% qt_getmctxt
        1.90% qt_setmctxt
        1.55% qthread_shep
        1.46% qt_mpool_free
        0.59% chpl_task_yield
        0.52% _waitEndCount3
        0.52% qt_threadqueue_enqueue_yielded
        0.15% remove3
        0.12% chpl_memhook_check_pre
        0.09% chpl_je_malloc
      + 0.09% _new13.constprop.543
      + 0.09% this3
        0.06% _delete_arr
        0.05% chpl_je_free
        0.04% deinit8.isra.193
        0.03% chpl_track_free
        0.03% _waitEndCount.constprop.509
        0.03% chpl_track_malloc
        0.03% append_chpl
        0.02% chpl__auto_destroy_DefaultRectangularDom
        0.02% _delete_dom3
        0.01% dsiLinksDistribution2
        0.01% chpl_memhook_check_post
        0.01% dsiMyDist2
        0.01% chpl__auto_destroy_ArrayViewSliceArr
        0.00% decEltCountsIfNeeded2
        0.00% dsiDestroyArr
```

Pacific Northwest
NATIONAL LABORATORY

Explanation for ‘Plateau Effect’ (44 threads)

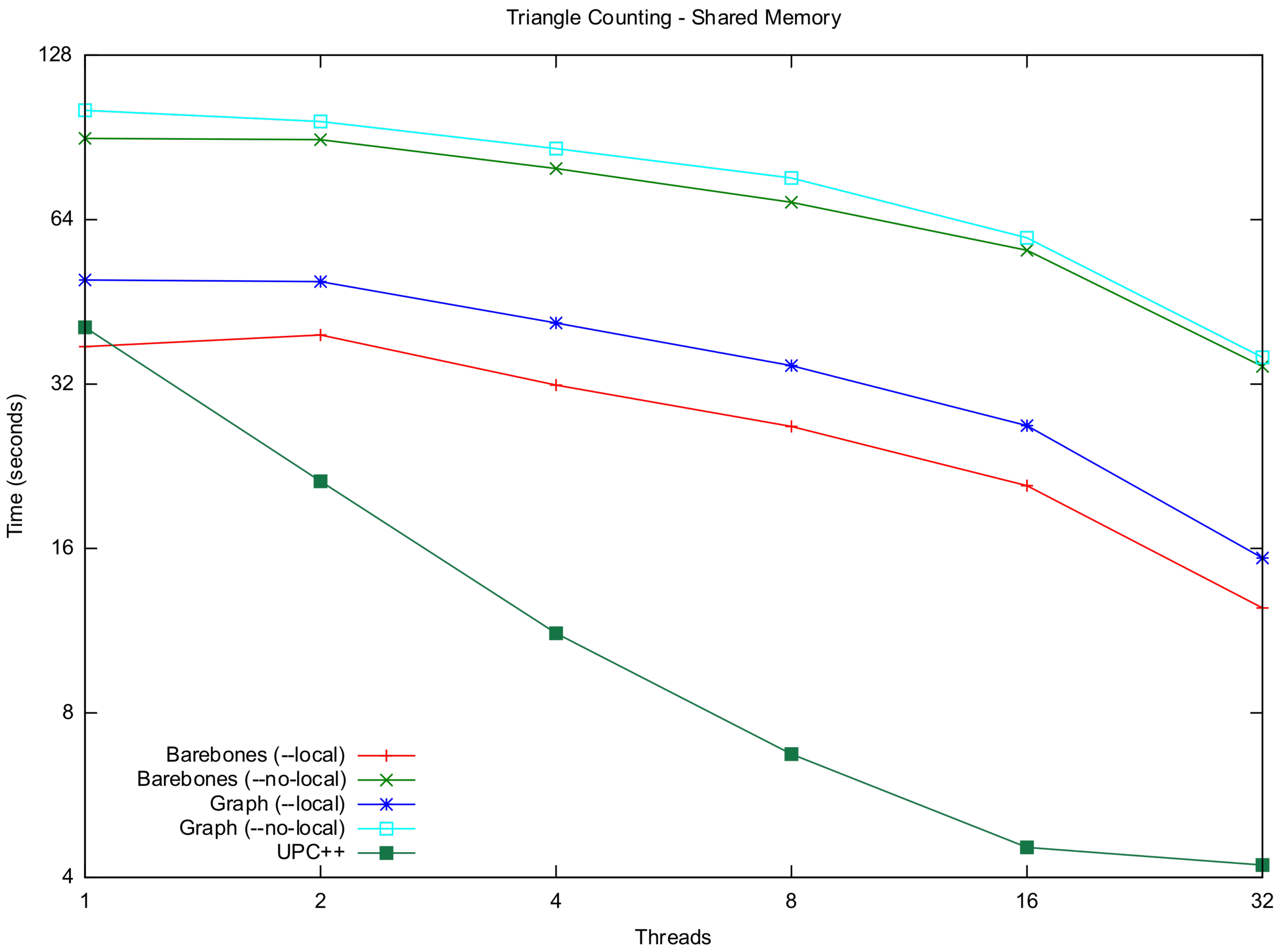
BAD

BAD

```
Samples: 755K of event 'cycles:ppp', Event count (approx.): 961655142110
  Children      Self  Parent symbol
-  100.00%   100.00%   [other]
   - 43.49% wrapcoforall_fn_chpl12
      + 42.76% intersectionSize_chpl4
        0.23% this3
        0.04% remove4
        0.03% chpl_localeID_to_locale
        0.02% localeIDtoLocale2
        0.02% deinit8.isra.193
        0.02% _new13.constprop.543
        0.01% chpl_count_help2.constprop.559
        0.01% dsiGetBaseDom
   + 25.90% __memcpy_avx_unaligned
   + 8.28% 0x8
   + 5.85% 0x10bf6274c085fb89
   + 5.59% 0
   + 2.94% 0x10000
   + 2.68% 0x7
     2.08% qt_mpool_free
   + 0.79% pthread_getspecific
     0.24% pthread_getspecific@plt
   + 0.23% coforall_fn_chpl11.isra.420.constprop.467
   + 0.19% chpl_je_free
   + 0.18% chpl_je_malloc
   + 0.16% chpl_memhook_check_pre
   + 0.08% __memset_avx2
     0.06% chpl_track_free
   + 0.04% qio_channel_create
   + 0.04% chpl_mem_inited
   + 0.03% 0x1
   + 0.02% chpl_track_malloc
   + 0.02% 0x4
   + 0.02% 0x28
   + 0.02% 0x2
   + 0.02% 0x26
   + 0.02% 0x3
   + 0.02% 0x29
   + 0.02% 0x20
   + 0.02% 0x23
   + 0.02% 0x27
   + 0.02% 0x6
   + 0.02% 0x5
   + 0.02% 0x1d
   + 0.02% 0x24
```

Triangle Counting – Shared Memory

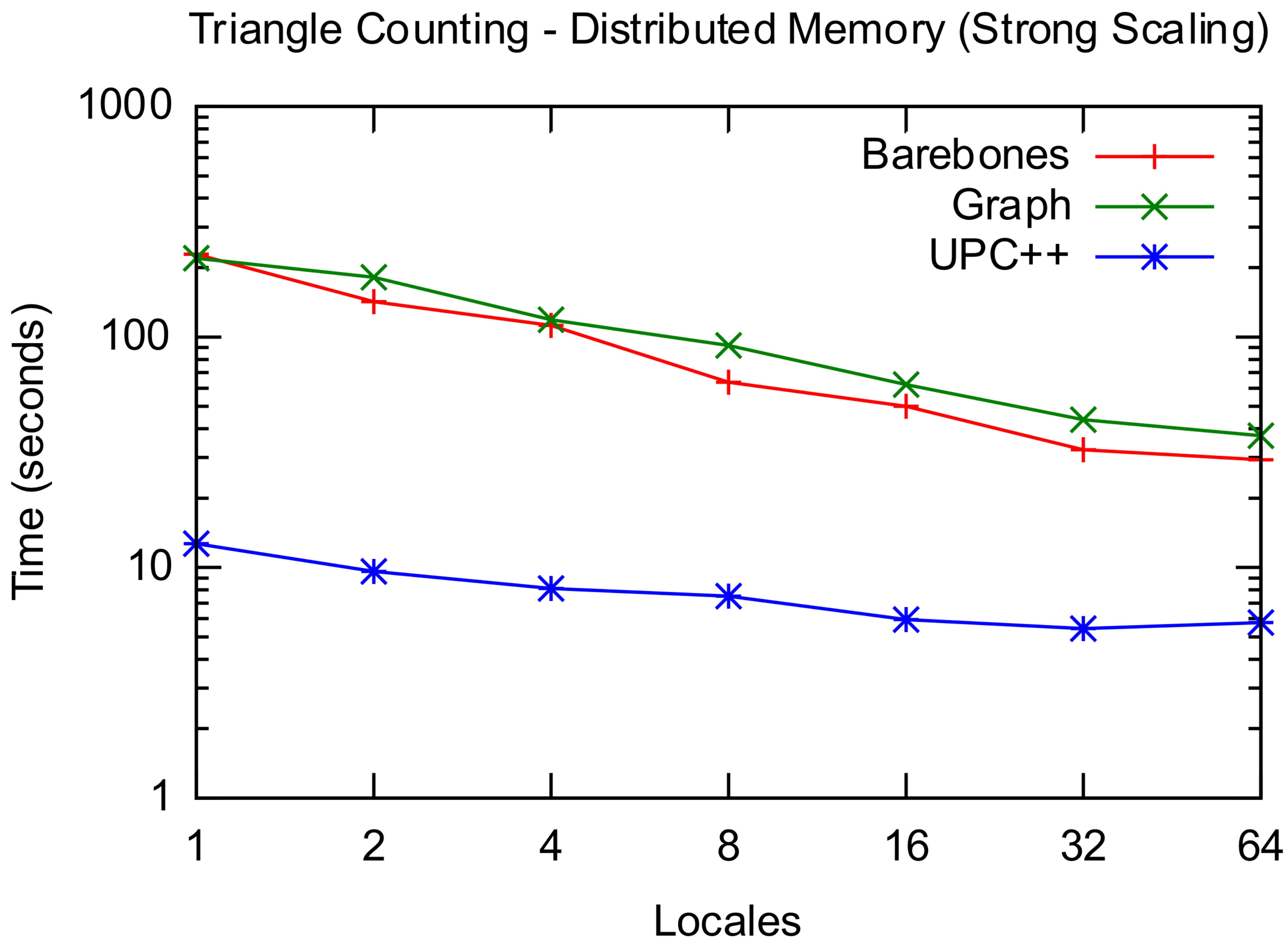
- 4M Vertices, 34.6M Edges (com-LiveJournal)
- Locality checks overhead
  - *After* locality optimizations
- Shallow improvement
  - Little shared-memory scalability
    - ✓ Better than plateau
  - Depends on kind of graph



Pacific Northwest NATIONAL LABORATORY

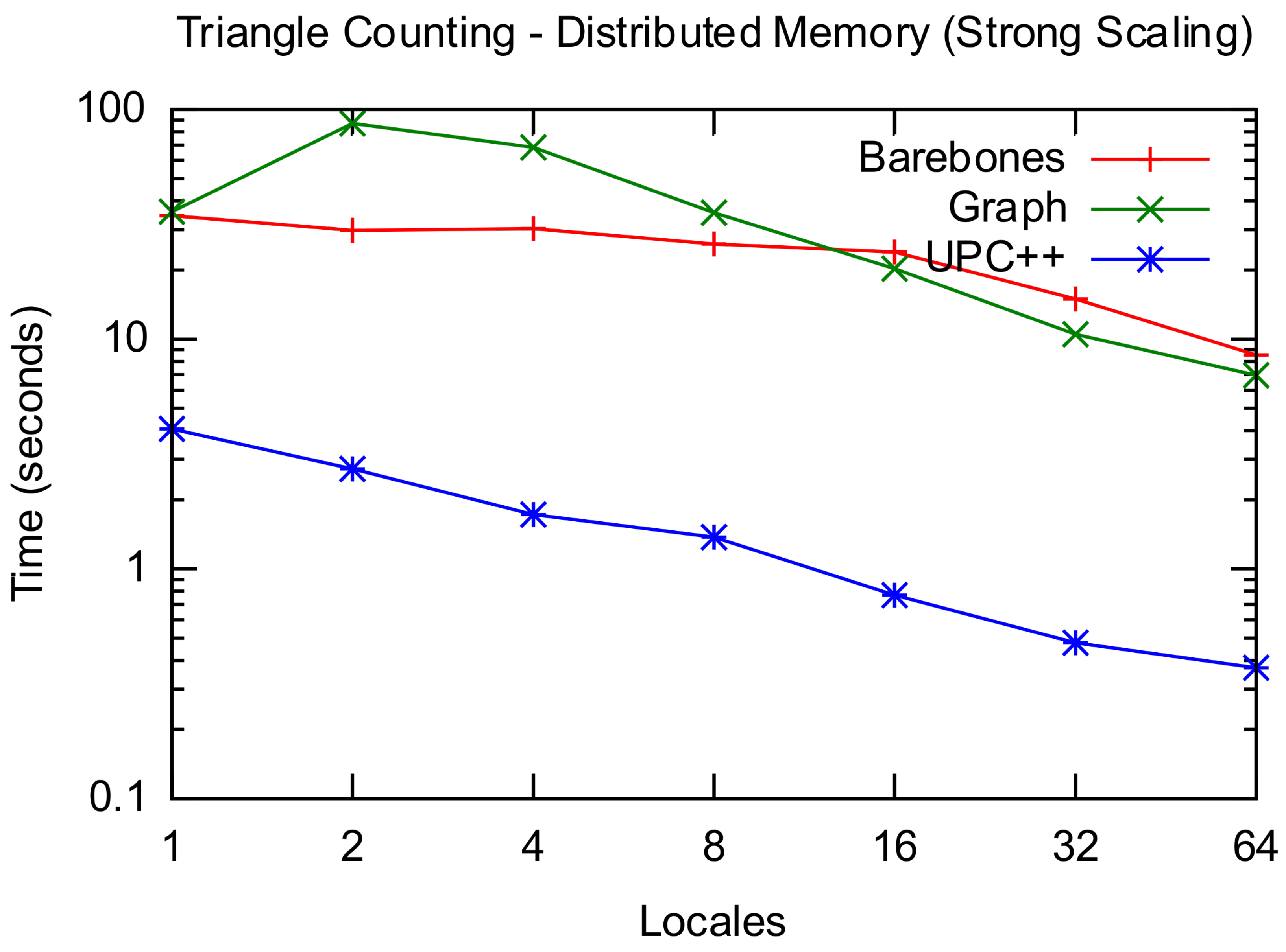
Triangle Counting—Distributed Memory

- Strong Scaling
  - 650K Vertices, 32M Edges (synthetic kronecker generated graph)
- Graph not much slower than barebones
  - Abstraction is not too bad!



Triangle Counting—Distributed Memory

- Strong Scaling
  - 4M Vertices, 34.6M Edges (com-LiveJournal)
- Graph spikes at 2 locale but scales after



Advertisement: IrregularToolkit

- Soon-to-be IrregularToolkit (Global-View Distributed Data Structures)
  - Aggregation Library (Dynamic and Static sized aggregation buffers)
  - WorkQueue (Send aggregated or fine-grained data to locales for processing)
  - TerminationDetection (Explicit termination detection)

```
var current = new WorkQueue(graph.vDescType, aggregationSize=-1);
var next = new WorkQueue(graph.vDescType, aggregationSize=-1);
var currentTD = new TerminationDetector(1);
var nextTD = new TerminationDetector(0);
current.addWork(graph.toVertex(0)); // add source
var visited : [graph.verticesDomain] atomic bool;
while !current.isEmpty() {
  forall vertex in doWorkLoop(current, currentTD) {
    if visited[vertex.id].testAndSet() == false { // already visited?
      for neighbor in graph.neighbors(vertex) {
        nextTD.started(1);
        next.addWork(neighbor, graph.getLocale(neighbor));
      }
    }
    currentTD.finished(1);
  }
  next <=> current;
  nextTD <=> currentTD;
}
```

Conclusion

- Building specialized graphs on top of a hypergraph isn’t a bad idea
  - Some overhead, but small compared to other things such as locality checking
  - Enables code-reuse on a whole new scale!
    - ✓ Building global-view distributed property graph from a global-view distributed property hypergraph
- Composition of Distributed Data Structures
  - Works exceptionally well with somewhat minimal overhead
  - Graph builds on AdjListHyperGraph and Aggregator
    - ✓ AdjListHyperGraph builds on Distributed Arrays and Aggregator
- Identified performance bottleneck from excess locality checks
  - Discovered importance of `local` blocks (R.I.P)
  - Hopeful for ‘local’ modifier on variables and class/record fields
- Identified performance bottleneck in runtime tasking layer
  - Prevents shared-memory scalability, hinders distributed memory scalalbility

Pacific Northwest NATIONAL LABORATORY

Thank you

Pacific Northwest
NATIONAL LABORATORY

UPC++ Triangle Counting

- No global-view programming
  - Forced to do local index calculations

```
// local triangle count iterator
size_t                   local_triangle_count = 0;
counting_output_iterator counter(local_triangle_count);
// the start of the conjoined future
upcxx::future<> fut_all = upcxx::make_future();
// For each vertex
for (uint64_t i = 0; i < num_vertices_per_rank; i++) {
  auto vtx_ptr = bases[upcxx::rank_me()].local()[i];
  auto adj_list_start = vtx_ptr.p.local();
  auto adj_list_len = vtx_ptr.n;
  auto current_vertex_id = index_to_vertex_id(i);
  // For each neighbor of the vertex, first get the
  // global pointer to the adjacency list and size
  for (auto j = 0; j < vtx_ptr.n; j++) {
    auto neighbor = adj_list_start[j];
    if (current_vertex_id < neighbor) {
      auto rank   = vertex_id_to_rank(neighbor);
      auto offset = vertex_id_to_offset(neighbor);
      upcxx::future<> fut = upcxx::rget(bases[rank] +
                                 offset)
      .then(
        [=] (gptr_and_len pn) {
        // Allocate a buffer of the same size
        std::vector<uint64_t> two_hop_neighbors(pn.n);
        // rget the actual list
        return upcxx::rget(pn.p,
          two_hop_neighbors.data(), pn.n)
        .then([=, two_hop_neighbors =
          std::move(two_hop_neighbors]() {
          // set intersection
          std::set_intersection(adj_list_start,
                         adj_list_start + adj_list_len,
                         two_hop_neighbors.begin(),
                         two_hop_neighbors.end(),
                         counter);
          });
        });
        // conjoin the futures
        fut_all = upcxx::when_all(fut_all, fut);
    }
  }
}
// wait for all the conjoined futures to complete
fut_all.wait();
...
auto done_reduction = upcxx::reduce_one(
   &local_triangle_count, &total_triangle_count, 1,
   [](size_t a, size_t b) { return a + b; }, 0);
done_reduction.wait();
```

Pacific Northwest
NATIONAL LABORATORY

UPC++ Triangle Counting

- Explicit Asynchronous Communication
  - Overlap computation with communication

```
// local triangle count iterator
size_t                   local_triangle_count = 0;
counting_output_iterator counter(local_triangle_count);
// the start of the conjoined future
upcxx::future<> fut_all = upcxx::make_future();
// For each vertex
for (uint64_t i = 0; i < num_vertices_per_rank; i++) {
  auto vtx_ptr = bases[upcxx::rank_me()].local()[i];
  auto adj_list_start = vtx_ptr.p.local();
  auto adj_list_len = vtx_ptr.n;
  auto current_vertex_id = index_to_vertex_id(i);
  // For each neighbor of the vertex, first get the
  // global pointer to the adjacency list and size
  for (auto j = 0; j < vtx_ptr.n; j++) {
    auto neighbor = adj_list_start[j];
    if (current_vertex_id < neighbor) {
      auto rank   = vertex_id_to_rank(neighbor);
      auto offset = vertex_id_to_offset(neighbor);
      upcxx::future<> fut = upcxx::rget(bases[rank] +
                                 offset)
      .then(
        [=] (gptr_and_len pn) {
        // Allocate a buffer of the same size
        std::vector<uint64_t> two_hop_neighbors(pn.n);
        // rget the actual list
        return upcxx::rget(pn.p,
          two_hop_neighbors.data(), pn.n)
        .then([=, two_hop_neighbors =
          std::move(two_hop_neighbors]() {
          // set intersection
          std::set_intersection(adj_list_start,
                        adj_list_start + adj_list_len,
                        two_hop_neighbors.begin(),
                        two_hop_neighbors.end(),
                        counter);
          });
        });
        // conjoin the futures
        fut_all = upcxx::when_all(fut_all, fut);
    }
  }
}
// wait for all the conjoined futures to complete
fut_all.wait();
...
auto done_reduction = upcxx::reduce_one(
   &local_triangle_count, &total_triangle_count, 1,
   [](size_t a, size_t b) { return a + b; }, 0);
done_reduction.wait();
```

Pacific Northwest NATIONAL LABORATORY

UPC++ Triangle Counting

- Continuations
  - First-Class Functions (C++)
  - Asynchronous execution
  - Nested Continuations

```
// local triangle count iterator
size_t                  local_triangle_count = 0;
counting_output_iterator counter(local_triangle_count);
// the start of the conjoined future
upcxx::future<> fut_all = upcxx::make_future();
// For each vertex
for (uint64_t i = 0; i < num_vertices_per_rank; i++) {
  auto vtx_ptr = bases[upcxx::rank_me()].local()[i];
  auto adj_list_start = vtx_ptr.p.local();
  auto adj_list_len = vtx_ptr.n;
  auto current_vertex_id = index_to_vertex_id(i);
  // For each neighbor of the vertex, first get the
  // global pointer to the adjacency list and size
  for (auto j = 0; j < vtx_ptr.n; j++) {
    auto neighbor = adj_list_start[j];
    if (current_vertex_id < neighbor) {
      auto rank   = vertex_id_to_rank(neighbor);
      auto offset = vertex_id_to_offset(neighbor);
      upcxx::future<> fut = upcxx::rget(bases[rank] +
                                          offset)
      .then(
        [=] (gptr_and_len pn) {
        // Allocate a buffer of the same size
        std::vector<uint64_t> two_hop_neighbors(pn.n);
        // rget the actual list
        return upcxx::rget(pn.p,
          two_hop_neighbors.data(), pn.n)
        .then([=, two_hop_neighbors =
          std::move(two_hop_neighbors]() {
          // set intersection
          std::set_intersection(adj_list_start,
                                adj_list_start + adj_list_len,
                                two_hop_neighbors.begin(),
                                two_hop_neighbors.end(),
                                counter);
          });
        });
        // conjoin the futures
        fut_all = upcxx::when_all(fut_all, fut);
    }
  }
}
// wait for all the conjoined futures to complete
fut_all.wait();
...
auto done_reduction = upcxx::reduce_one(
   &local_triangle_count, &total_triangle_count, 1,
   [](size_t a, size_t b) { return a + b; }, 0);
done_reduction.wait();
```

UPC++ Triangle Counting

- Wait for all communication to finish…
  - Queue up all communication and NIC and runtime handle the rest…

```
// local triangle count iterator
size_t                   local_triangle_count = 0;
counting_output_iterator counter(local_triangle_count);
// the start of the conjoined future
upcxx::future<> fut_all = upcxx::make_future();
// For each vertex
for (uint64_t i = 0; i < num_vertices_per_rank; i++) {
  auto vtx_ptr = bases[upcxx::rank_me()].local()[i];
  auto adj_list_start = vtx_ptr.p.local();
  auto adj_list_len = vtx_ptr.n;
  auto current_vertex_id = index_to_vertex_id(i);
  // For each neighbor of the vertex, first get the
  // global pointer to the adjacency list and size
  for (auto j = 0; j < vtx_ptr.n; j++) {
    auto neighbor = adj_list_start[j];
    if (current_vertex_id < neighbor) {
      auto rank   = vertex_id_to_rank(neighbor);
      auto offset = vertex_id_to_offset(neighbor);
      upcxx::future<> fut = upcxx::rget(bases[rank] +
                                         offset)
      .then(
        [=] (gptr_and_len pn) {
        // Allocate a buffer of the same size
        std::vector<uint64_t> two_hop_neighbors(pn.n);
        // rget the actual list
        return upcxx::rget(pn.p,
          two_hop_neighbors.data(), pn.n)
        .then([=, two_hop_neighbors =
          std::move(two_hop_neighbors]() {
          // set intersection
          std::set_intersection(adj_list_start,
                                adj_list_start + adj_list_len,
                                two_hop_neighbors.begin(),
                                two_hop_neighbors.end(),
                                counter);
          });
        });
        // conjoin the futures
        fut_all = upcxx::when_all(fut_all, fut);
    }
  }
}
// wait for all the conjoined futures to complete
fut_all.wait();
...
auto done_reduction = upcxx::reduce_one(
   &local_triangle_count, &total_triangle_count, 1,
   [](size_t a, size_t b) { return a + b; }, 0);
done_reduction.wait();
```

Pacific Northwest
NATIONAL LABORATORY

UPC++ Triangle Counting

- Explicit reduction step
  - No first-class language support for reductions

```
// local triangle count iterator
size_t                    local_triangle_count = 0;
counting_output_iterator counter(local_triangle_count);
// the start of the conjoined future
upcxx::future<> fut_all = upcxx::make_future();
// For each vertex
for (uint64_t i = 0; i < num_vertices_per_rank; i++) {
  auto vtx_ptr = bases[upcxx::rank_me()].local()[i];
  auto adj_list_start = vtx_ptr.p.local();
  auto adj_list_len = vtx_ptr.n;
  auto current_vertex_id = index_to_vertex_id(i);
  // For each neighbor of the vertex, first get the
  // global pointer to the adjacency list and size
  for (auto j = 0; j < vtx_ptr.n; j++) {
    auto neighbor = adj_list_start[j];
    if (current_vertex_id < neighbor) {
      auto rank   = vertex_id_to_rank(neighbor);
      auto offset = vertex_id_to_offset(neighbor);
      upcxx::future<> fut = upcxx::rget(bases[rank] +
                                        offset)
      .then(
        [=] (gptr_and_len pn) {
        // Allocate a buffer of the same size
        std::vector<uint64_t> two_hop_neighbors(pn.n);
        // rget the actual list
        return upcxx::rget(pn.p,
          two_hop_neighbors.data(), pn.n)
        .then([=, two_hop_neighbors =
          std::move(two_hop_neighbors]() {
          // set intersection
          std::set_intersection(adj_list_start,
                                adj_list_start + adj_list_len,
                                two_hop_neighbors.begin(),
                                two_hop_neighbors.end(),
                                counter);
          });
        });
        // conjoin the futures
        fut_all = upcxx::when_all(fut_all, fut);
    }
  }
}
// wait for all the conjoined futures to complete
fut_all.wait();
...
auto done_reduction = upcxx::reduce_one(
   &local_triangle_count, &total_triangle_count, 1,
   [](size_t a, size_t b) { return a + b; }, 0);
done_reduction.wait();
```