Chapel Graph Library (CGL)

June 22, 2019

Louis Jenkins
Marcin Zalewski
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

• 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
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

```plaintext
pragma "always AVF"
record GlobalClass {
  type classType;
  var pid : int;
  forwarding chpl_getPrivatizedCopy(pid, classType);
}

class LocalClass {
  var pid : int;
}
```

Locale#0 Locale#N
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
    ✓ Abstracts and optimizes locality from the user; usable from within all language constructs

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

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

```chapel
var vPropMap = new PropertyMap(string); // Assume is filled
def 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
  }
}
```
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”

```
locale #0

on Locales[1] do ...;

locale #0 task

locale #1 task

locale #0 data
```

From: Locale #0
To: Locale #1

Send to Locale #1
Global-View Distributed Data Structures
All Working Together

- Global-View Distributed Data Structures working together…
  - Chapel Arrays
  - Aggregator
  - AdjListHyperGraph

- What’s next…
  - Composition!

```plaintext
// 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();
}
```
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

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

```
2-Uniform Graph
1 var numTriangles : int;
2 forall v in graph.getVertices() with (+ reduce numTriangles) do
3     for u in graph.neighbors(v) do
4         if v < u then
5             numTriangles += graph.intersectionSize(v,u);
6     numTriangles /= 3;
```
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!

```c
1 proc intersectionSize(A : [ ] ?t, B : [ ] t) {
2   if isLocalArray(A) && isLocalArray(B) {
3     return intersectionSize(A, B);
4   } else if isLocalArray(A) && !isLocalArray(B) {
5     const BD = B.domain; // Make by-value copy so domain is not remote.
6     var _B : [ ]BD t = B;
7     return intersectionSize(A, _B);
8   } else if !isLocalArray(A) && isLocalArray(B) {
9     const AD = A.domain; // Make by-value copy so domain is not remote.
10    var _A : [ ]AD t = A;
11    return intersectionSize(_A, B);
12  } else {
13    const AD = A.domain; // Make by-value copy so domain is not remote.
14    const BD = B.domain;
15    var _A : [ ]AD t = A;
16    var _B : [ ]BD t = B;
17    return intersectionSize(_A, _B);
18  }
19 }
20 proc intersectionSize(A : [ ] ?t, B : [ ] t) {
21   var match : int;
22   local {
23     var idxA = A.domain.low;
24     var idxB = B.domain.low;
25     while idxA <= A.domain.high && idxB <= B.domain.high {
26       const a = A[idxA];
27       const b = B[idxB];
28       if a == b {
29         match += 1;
30         idxA += 1;
31         idxB += 1;
32       }
33       else if a > b {
34         idxB += 1;
35       } else {
36         idxA += 1;
37       }
38     }
39   }
40   return match;
41 }
```
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.
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)
Explaination for 'Plateau Effect' (2 threads)
Explanation for ‘Plateau Effect’ (44 threads)
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
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
• 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)

```java
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 scalability
Thank you
// local triangle count iterator
size_t local_triangle_count = 0;

// the start of the conjoined future
upcxx::future<> fut_all = upcxx::make_future();

// For each vertex
for (uint64_t i = 0; i < numVerticesPerRank; 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 (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();
// 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([](gp理解和neighbour){
                    // 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([](std::vector<uint64_t>& 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

• Explicit Asynchronous Communication
  ▪ Overlap computation with communication
UPC++ Triangle Counting

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

```cpp
// local triangle count iterator
type size_t local_triangle_count = 0;

// 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([=] (gp_t_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([](auto 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…

```cpp
// local triangle count iterator
template<> size_t local_triangle_count = 0;

// 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([&](gp::and_<lp>) {
                    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(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();
```
// local triangle count iterator
size_t local_triangle_count = 0;

// 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([=] {
              // 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

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