Chapel Aggregation Library (CAL)

By Louis Jenkins
The Problem

• Accessing remote data is slow
  • Multiple orders of magnitude slower to access than local memory
The Problem

- Accessing remote data is slow
  - Multiple orders of magnitude slower to access than local memory
- “Moving the computation to the data” not always the best solution
  - Using an `on` statement requires migrating tasks to another locale

```
on Locales[1] do ...;
```
The Problem

• Accessing remote data is slow
  • Multiple orders of magnitude slower to access than local memory
• “Moving the computation to the data” not always the best solution
  • Using an on statement requires migrating tasks to another locale
    • Can become bottleneck if fine-grained
The Problem

• Accessing remote data is slow
  • Multiple orders of magnitude slower to access than local memory
• “Moving the computation to the data” not always the best solution
  • Using an `on` statement requires migrating tasks to another locale
    • Can become bottleneck if fine-grained
    • Task creation is relatively expensive
      • Tasks are too large to spawn in a fire-and-forget manner (issue #9984)
      • Migrating tasks require individual active messages (issue #9727)
The Solution

• Coarsen the granularity of the data
  • Buffer units of data to be sent to a locale in *destination buffers*
The Solution

• Coarsen the granularity of the data
  • Buffer units of data to be sent to a locale in *destination buffers*
The Solution

- Coarsen the granularity of the data
  - Buffer units of data to be sent to a locale in *destination buffers*
The Solution

- Coarsen the granularity of the data
  - Buffer units of data to be sent to a locale in destination buffers
  - When buffer is full, it can be *flushed* to be handled by the user
The Solution

- Coarsen the granularity of the data
  - Buffer units of data to be sent to a locale in destination buffers
  - When buffer is full, it can be flushed to be handled by the user
  - User can perform coalescing to combine aggregated data

From: Locale #0
To: Locale #1

on Locales[1] do ...;

Locale #0 Task

Coalesced Data

Send to Locale #1

Locale #1 Task

Locale #0 Coalesced Data
The CAL API

Design principles and library semantics
Chapel Aggregation Library (CAL)

• Distributed
  • Privatization makes it possible to use on any locale efficiently
  • Aggregates data in per-locale destination buffers

• Minimal
  • Generic on user data
  • User determines how data is handled

• Parallel-Safe and Parallel-Encouraged
  • Can be called from any task on any locale
  • Provides significant performance improvements

• Chapel Module
  • No run-time nor compiler changes required
  • Written in Chapel, for Chapel
Usage of CAL
Initialization

• The ‘Aggregator’ is generic on the type of data it is to aggregate
• Privatization is handled ‘behind-the-scenes’, no extra effort by user

```javascript
var aggregator = new Aggregator(int);
```
Usage of CAL Deinitialization

• The ‘Aggregator’ is record-wrapped
  • No reference counting nor life-time support
  • Cannot `delete` a record

• User must explicitly invoke `destroy` to actually cleanup resources used

```c
1 aggregator.destroy();
```
Usage of CAL
Aggregating Data

• User can *aggregate* data
  • Returns a buffer which must be handled explicitly by user
    • Lack-luster first-class function support prevents automation and optimization
      • Discussed more later

```plaintext
1 for i in 1..N { 
2   var buf = aggregator.aggregate(i, Locales[1]);
3   if buf != nil {
4     handleBuffer(buf, Locales[1]);
5   }
6 }
```
Usage of CAL
Aggregating Data in Parallel

- User can aggregate data from multiple tasks
- Extremely fast and scales up to maximum parallelism

```plaintext
1 forall i in 1..N {
2    var buf = aggregator.aggregate(i, Locales[1]);
3    if buf != nil {
4        handleBuffer(buf, Locales[1]);
5    }
6 }
```
Usage of CAL
Aggregating Data in
Distributed Context

- The Aggregator can be used across multiple locales
  - Uses privatization where each locale has their own local instance
  - Will not jump to a single locale, all accesses are forwarded to the privatized instance (important!!!)
- User currently must declare `forall`-intent as `in` to prevent communication
  - Discussed later

```plaintext
1 var A : [someCyclicDom] int;
2 var B : [someBlockDom] int;
3 // Wait for all asynchronous tasks to finish
4 sync forall a in A with (in aggregator) {
5   const loc = B[a].locale;
6   var buf = aggregator.aggregate(a, loc);
7   if buf != nil {
8     // Handle buffer asynchronously
9       begin handleBuffer(buf, loc);
10   }
11 }
```
Usage of CAL
Flushing Buffers

• User must manually *flush* the buffers when finished
  • Ways to automate this is discussed later
  • For now we use a parallel iterator

```python
1 forall (buf, loc) in aggregator.flush() {
2   handleBuffer(buf, loc);
3 }
```
Example Usage of CAL

Practical example of CAL and performance improvements
Histogram - Naive

• \textit{rindex} is a block distributed array of random indices into \textit{A}  
• \textit{A} is a cyclic distributed array of atomic counters  
• Both \textit{rindex} and \textit{A} are distributed differently

```python
1 forall r in rindex {
2 A[r].add(1);
3 }
```
Histogram - Aggregated

- Coalescing of aggregated data
  - Combine duplicate increments to same index
  - May reduce needed computation needed for destination
  - May reduce size of data being sent

- Benefit of dealing with bulk data
  - Can take extremely large stream of data and process them in windows
    - Might not be possible with entirety of stream

```java
1 // Aggregation Handler
2 proc handleBuffer(buf : Buffer(int), loc : locale) {
3   // Coalescing...
4   var counters : [A.domain.localSubdomain(loc)] int(64);
5   for idx in buf do counters[idx] += 1;
6   // Recycle buffer
7   buf.done();
8
9   // Process coalesced data
10  on loc {
11     // Copy data locally
12     const _tmp = counters;
13     for (cnt, idx) in zip(_tmp, _tmp.domain) {
14       if cnt > 0 {
15         A[idx].add(cnt);
16       }
17     }
18  }
19  }
20 }
21 // Aggregating Indices
22 var aggregator = new Aggregator(int);
23
24 // Aggregate and wait for asynchronous tasks to finish
25 sync forall r in rindex with (in aggregator) {
26   const loc = A[r].locale;
27   // If its local, handle it
28   if loc == here {
29     A[r].add(1);
30   } else {
31     var buf = aggregator.aggregate(r, loc);
32     if buf != nil {
33       // Handle buffer asynchronously
34       begin handleBuffer(buf, loc);
35     }
36   }
37 }
38}
39 // Flush
40 forall (buf, loc) in aggregator.flush() {
41   handleBuffer(buf, loc);
42 }
```
• Recycling the buffer when finished allows other tasks to use it
  • Safe to call from other locales
    • Not as efficient as calling on host locale
• No longer needed as we already coalesced the data.

```plaintext
// Aggregation Handler
proc handleBuffer(buf : Buffer(int), loc : locale) {
  // Coalescing...
  var counters : [A.domain.localSubdomain(loc)] int(64);
  for idx in buf do counters[idx] += 1;

  // Recycle buffer
  buf.done();

  // Process coalesced data
  on loc {
    // Copy data locally
    const _tmp = counters;
    for (cnt, idx) in zip(_tmp, _tmp.domain) {
      if cnt > 0 {
        A[idx].add(cnt);
      }
    }
  }

  // Aggregating Indices
  var aggregator = new Aggregator(int);

  // Aggregate and wait for asynchronous tasks to finish
  sync forall r in rindex with (in aggregator) {
    const loc = A[r].locale;
    // If its local, handle it
    if loc == here {
      A[r].add(1);
    } else {
      var buf = aggregator.aggregate(r, loc);
      if buf != nil {
        // Handle buffer asynchronously
        begin handleBuffer(buf, loc);
      }
    }
  }

  // Flush
  forall (buf, loc) in aggregator.flush() {
    handleBuffer(buf, loc);
  }
```
Histogram - Aggregated

- Iterate over local index and coalesced increment count
  - Perform increment to counter that corresponds to local index.

```java
// Aggregation Handler
proc handleBuffer(buf : Buffer(int), loc : locale) {
    // Coalescing...
    var counters : [A.domain.localSubdomain(loc)] int64;
    for idx in buf do counters[idx] += 1;

    // Recycle buffer
    buf.done();

    // Process coalesced data
    on loc {
        // Copy data locally
        const _tmp = counters;
        for (cnt, idx) in zip(_tmp, _tmp.domain) {
            if cnt > 0 {
                A[idx].add(cnt);
            }
        }
    }

    // Aggregating Indices
    var aggregator = new Aggregator(int);

    // Aggregate and wait for asynchronous tasks to finish
    sync forall r in rindex with (in aggregator) {
        const loc = A[r].locale;
        // If its local, handle it
        if loc == here {
            A[r].add(1);
        } else {
            var buf = aggregator.aggregate(r, loc);
            if buf != nil {
                // Handle buffer asynchronously
                begin handleBuffer(buf, loc);
            }
        }
    }

    // Flush
    forall (buf, loc in aggregator.flush()) {
        handleBuffer(buf, loc);
    }
}
```
Perform increments that are local immediately
- Even though overhead is relatively small for aggregating data, nothing beats free
- Coalescing also is counter-productive here

```javascript
1 // Aggregation Handler
2 proc handleBuffer(buf : Buffer(int), loc : locale) {
3    // Coalescing...
4    var counters : [A.domain.localSubdomain(loc)] int(64);
5    for idx in buf do counters[idx] += 1;
6
7    // Recycle buffer
8    buf.done();
9
10   // Process coalesced data on loc {
11      // Copy data locally
12      const _tmp = counters;
13      for (cnt, idx) in zip(_tmp, _tmp.domain) {
14          if cnt > 0 {
15              A[idx].add(cnt);
16          }
17      }
18  }
19
20 // Aggregating Indices
21 var aggregator = new Aggregator(int);
22
23 // Aggregate and wait for asynchronous tasks to finish
24 sync forall r in rindex with (in aggregator) {
25    const loc = A[r].locale;
26    // If its local, handle it
27    if loc == here {
28        A[r].add(1);
29    } else {
30        var buf = aggregator.aggregate(r, loc);
31        if buf != nil {
32            // Handle buffer asynchronously
33            begin handleBuffer(buf, loc);
34        }
35    }
36}
37
38 // Flush
39 forall (buf, loc) in aggregator.flush() {
40    handleBuffer(buf, loc);
41 }
```
Performance Analysis

• Performance under aggregation is the same for both GASNet and uGNI
  • Even though GASNet is significantly slower than uGNI
  • Why? Aggregation reduces the required communication
    • Less PUT and GET
    • Less remote tasks (on-statements)
    • Overall less work for communication layer
Feature Requests

How to make it easier to use CAL
Feature Request
Remote-Value Forwarding on User Types

• Remote-Value Forwarding
  • Compiler-optimization that changes default `forall`-intent as by-value
    • Default is normally by reference where each access is a round-trip to the original locale
  • Would be appreciated for user-defined types (issue #9717)

With Remote-Value Forwarding

```plaintext
1 var A : [someCyclicDom] int; 1 var A : [someCyclicDom] int;
2 var B : [someBlockDom] int; 2 var B : [someBlockDom] int;
3 // Wait for all asynchronous tasks to finish 3 // Wait for all asynchronous tasks to finish
4 sync forall a in A with (in aggregator) { 4 sync forall a in A {
  5     const loc = B[a].locale;  5     const loc = B[a].locale;
  6     var buf = aggregator.aggregate(a, loc);  6     var buf = aggregator.aggregate(a, loc);
  7     if buf != nil {  7     if buf != nil {
        // Handle buffer asynchronously  8         // Handle buffer asynchronously
        9         begin handleBuffer(buf, loc);  9         begin handleBuffer(buf, loc);
 10   } 10 }
```
Benefit to CAL
Remote-Value Forwarding on User Types

- Remove need for user to explicitly define `forall`-intent as `in`
  - Eliminates cases where user forgets to do so leading into bad performance
  - Eliminates cases where user mistakes bad performance for bad library design
Feature Request

Improved First-Class Functions

• First-Class Functions that are similar to inlined iterators
  • Should be efficient to call across multiple locales
    • Accessing ‘A’ should be remote-value forwarded

• Allow First-Class Functions that access local variables to be returned
  • Copy arguments on stack to the heap if returned and redirect all accesses there
    • Maybe by creating variant of original function object similar to lifetime-checking?
    • Maybe cleaning up heap-allocated data could be done using reference-counting?

```javascript
var handleBuffer = lambda(buf : Buffer(int), loc : locale) {
    begin on loc do [idx in buf] A[idx].add(1);
}
var aggregator = new Aggregator(int, handleBuffer);
sync [idx in indices] aggregator.aggregate(idx, A[idx].locale);
aggregator.flush();
```
Benefit to CAL
Improved First-Class Functions

• Allows user to pass around Aggregator
  • Returning aggregator from a function means returning the handler from a function

• Significantly reduces boilerplate code
  • Makes aggregation quick, easy, and painless

• Allows optimization and automation
  • Flushing of the buffer can be automated and optimized behind-the-scenes
Feature Request
Assigning scheduling priority to tasks

• Not all tasks should be scheduled fairly
  • Some tasks do not need to be scheduled as often as others
  • Some tasks should not be taken into account for calculations
    • I.E forall loops for Chapel’s standard arrays and distributions

• Categories
  • Background – Tasks that are IO-Bound or are dedicated to book-keeping
  • Normal – Your average task
  • Computational – Tasks that are CPU-bound such as those used in forall loops

• Possible approach
  • Implement new scheduler in qthreads as experimental testing ground
    • FIFO tasking layer has no priority, nothing needs to be done for it
    • Massive threads is defunct anyway...
Benefit to CAL
Assigning scheduling priority to tasks

• Single background task per locale that handles flushing buffer
  • Based on rate-of-change heuristic to save time for computational tasks

• Tasks that are aggregating data can be made computational
  • Background tasks can take up less of a time slice from a computational task

```
1 begin(TaskPriority.Background) {
2   while keepAlive {
3     doBackgroundWork();
4     chpl_task_yield();
5   }
6 }
7 coforall loc in Locales do on loc {
8   forall x in X with (priority TaskPriority.computational) {
9     computeWith(x);
10   }
11 }
```
Possible Runtime Integration
Squeezing out more performance

• Aggregation is extremely fast, but...
  • Processing the aggregated data can lead to performance issues

• Add some kind of *routing* of aggregation buffers?
  • Aggregate the aggregation buffers
  • Send buffer to single-hop neighbor
  • Repeat?

• Send data to destination locale
  • Currently have to explicitly retrieve it from source on destination
Extras

Erdős–Rényi – HyperGraph Generation
Hypergraph - Description

• Hypergraph – Consists of vertices $V$ and hyperedges $H$
  • Hyperedges represent relationship between 2 or more vertices
    • Such as authors of a paper; vertices are authors, hyperedge is co-authorship on paper
    • A graph is a hypergraph with hyperedges that all have a cardinality of 2

• Dual Hypergraph – Vertices represent relationship between 2 or more hyperedges
  • If a hyperedge contains a vertex, then that vertex contains that hyperedge
    • Bidirectional mapping of vertices and hyperedges

• Hypergraph generation – Synthetically create hypergraph based on information
Erdős–Rényi – Naïve

• A Vertex is an object with an adjacency list of indices that map to Edge objects
• A Edge is an object with an adjacency list of indices that map to Vertex objects
• vertices is a distributed array of Vertex objects
• edges is a distributed array of Edge objects
• verticesRNG and edgesRNG are pre-computed random indices for vertices and edges respectively

```python
// Iterate over distributed array of pre-computed random numbers
forall (randVertex, randEdge) in zip(verticesRNG, edgesRNG) {
    on vertices[randVertex] do vertices[randVertex].addEdge(randEdge);
    on edges[randEdge] do edges[randEdge].addVertex(randVertex);
}
```
Erdős–Rényi – Aggregated (Idealized)

```plaintext
enum Inclusion { Vertex, Edge }

type dataType = (int, int, Inclusion);

proc handleBuffer(buf : Buffer(dataType), loc : locale) {
    on loc {
        forall (src, dest, opType) in buf {
            select opType {
                when Inclusion.Vertex do vertices[src].addEdge(dest);
                when Inclusion.Edge do edges[src].addVertex(dest);
            }
        }
        buf.done();
    }
    var aggregator = new Aggregator(dataType, handleBuffer);
    sync forall (vRNG, eRNG) in zip(verticesRNG, edgesRNG) {
        const vData = (vertex, edge, Inclusion.Vertex);
        const eData = (edge, vertex, Inclusion.Edge);
        const vLoc = vertices[vertex].locale;
        const eLoc = edges[edge].locale;
        aggregator.aggregate(vData, vLoc);
        aggregator.aggregate(eData, eLoc);
    }
    aggregator.flush();
```