Chapel Aggregation Library (CAL)

Louis Jenkins
Pacific Northwest National Laboratory
louis.jenkins@pnnl.gov

Marcin Zalewski
Pacific Northwest National Laboratory
marcin.zalewski@pnnl.gov

Michael Ferguson
Cray Inc.
mferguson@cray.com

***Abstract*—Fine-grained communication is a fundamental principle of the Partitioned Global Address Space (PGAS), which serves to simplify creating and reasoning about programs in the distributed context. However, per-message overheads of communication rapidly accumulate in programs that generate a high volume of small messages, limiting the effective bandwidth and potentially increasing latency if the messages are generated at a much higher rate than the effective network bandwidth. One way to reduce such fine-grained communication is by coarsening the granularity by aggregating data, or by buffering the smaller communications together in a way that they can be handled in bulk. Once these communications are buffered, the multiple units of the aggregated data can be combined into fewer units in an optimization called coalescing.**

The Chapel Aggregation Library (CAL) provides a straightforward approach to handling both aggregation and coalescing of data in Chapel and aims to be as generic and minimal as possible to maximize code reuse and minimize its increase in complexity on user applications. CAL provides a high-performance, distributed, and parallel-safe solution that is entirely written as a Chapel module. In addition to being easy to use, CAL improves the performance of some benchmarks by one to two orders of magnitude over naive implementations at 32 compute-nodes on a Cray XC50.

I. Introduction and Related Work

The Chapel programming language [1], [2] is an open-source language for parallel computing on large-scale systems that is easy-to-use, portable, and scalable. Chapel is a partitioned global address space (PGAS) language, but it differs from other PGAS languages such as UPC [3] in that it abstracts communication and performs (most of) it implicitly. Chapel transforms individual reads and writes of remote memory into GET and PUT RDMAs, and it similarly transforms remote procedures and tasks into active messages. While this significantly reduces the complexity involved in writing correct Chapel user programs, it can often make it difficult for the user to write programs that perform well because of the overhead of small messages. In this work, we address this problem with *aggregation* and *coalescing*. Aggregation coarsens the granularity of communication by pooling together individual units of data into larger chunks, and coalescing reduces the amount of data to be communicated by combining some of these units.

Other works in aggregation, such as AM++ [4], perform aggregation and coalescing directly on active messages [5]. Chapel, however, is a higher-level language composed of *layers*, which are internal libraries that handle specific tasks. The *communication layer* is the runtime abstraction layer responsible for handling RDMAs and active messages, such as GASNet [6], [7] and uGNI [8]. The communication layer must also coordinate with the *tasking layer* responsible for the creation, scheduling, and management of *tasks*, which are multiplexed on top of threads, such as qthreads [9]. Both layers must interact with the *memory management layer*, the abstraction layer that is responsible for handling requests to allocate and de-allocate memory and is implemented with a low-level allocator such as jemalloc [10].

The Chapel Aggregation Library (CAL), presented in this work, is not the first project to address the problem of fine-grained communication. Ferguson and Buettner [11] developed a software write-back cache that is integrated into Chapel's run-time and compiler. The software cache enables aggregation and overlap of individual PUT and GET operations, and provides support for programmer-specified prefetching. Where the software cache is entirely automated, Kayraklioglu, Ferguson, and El-Ghazawi [12] describe the concept of locality-aware productive prefetching support (LAPPS), where unlike the software cache, the user determines what to prefetch and the run-time handles how the data is prefetched in an implementation-specific manner. Unlike LAPPS and the software cache, CAL does not directly support prefetching and instead takes the approach of aggregating and coalescing user-defined data instead of individual PUT and GET operations. Unlike the software cache and LAPPS, CAL does not require any compiler or run-time changes or integration, and it exists entirely as a Chapel module, and as such, it gains the benefit of being able to work with Chapel's rich data and task parallelism and locality constructs.

In Section II, we discuss the core design details and philosophy behind CAL, describe the library API, and detail the interaction between the library and the user. In Section III, we provide a sketch of the algorithms used in Chapel-like pseudo-code. In Section IV, we present examples of the usage of the library in Chapel and provide a comparison between a simple naive implementation and one using CAL to demonstrate changes in design complexity of the algorithm. In Section V, we provide performance improvements over the naive examples demonstrated in Section IV. In Section VI, we discuss design choices and difficulties we faced. Finally, in Section VII we discuss planned improvements and plans to integrate the aggregation library into Chapel and how it may be used in other PGAS languages.

II. Design

CAL is designed according to the following design principles:

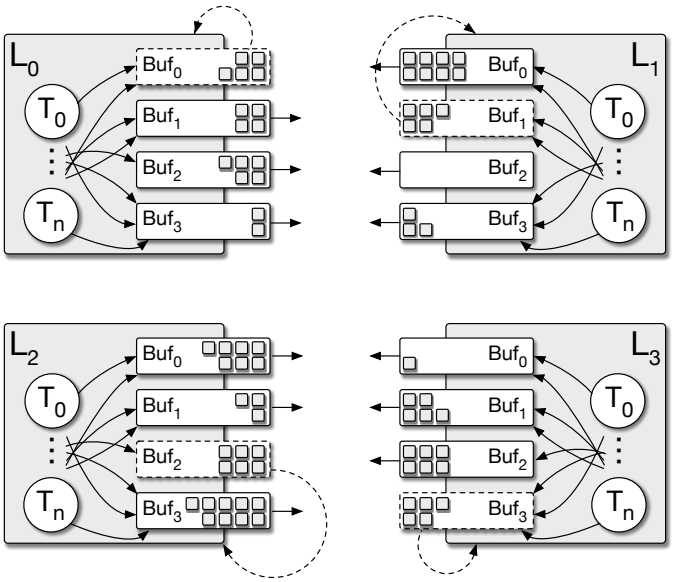


Fig. 1: The design of CAL aggregation.

```
class Buffer {
  type dataType;
  var data : [0..#BufferSize] dataType;
  var claimed : atomic int;
  var filled : atomic int;
  var stolen : atomic bool;
  var pool : MemoryPool(Buffer(dataType));
  proc done();
}
class Aggregator {
  type dataType;
  var pools :
    [LocaleSpace] MemoryPool(Buffer(dataType));
  var destBuffers :
    [LocaleSpace] Buffer(dataType);
  proc aggregate(data : dataType, loc : locale) :
    Buffer(dataType);
  iter flush() : (Buffer(dataType), locale);
}
```

Fig. 2: The basic API of CAL.

- **Minimalism**: CAL is designed to be small and unassuming, deferring decisions such as how the buffer is coalesced or processed to the user. This allows CAL to be reusable and modular. In other words, CAL's only responsibility is aggregating outgoing units of data; how the aggregated data is handled is at the discretion of the user. This minimalist design works well in conjunction with the high-level features of Chapel, which can seamlessly handle transfers of buffers, parallel execution, and all other aspects that a library like CAL might have to handle in a lower-level runtime.
- **Distributed**: CAL objects appear local but act distributed, following the spirit of Chapel. CAL employs *privatization* (similar to Chapel arrays) and optimized such that accesses to an instance of the library is forwarded to the appropriate local privatized instance. That is, it is inherently safe and efficient for multiple locales to share access to the same CAL instance.
- **Parallel-Safe**: CAL is designed to be transparently thread-safe and efficient. It employs lock-free data structures to aggregate units of data without race conditions and errors.

The basic design of CAL is illustrated in Fig. 1. Every locale, L_0, L_1, L_2, and L_3 in the figure, maintains per-destination buffers (in Section III we discuss how the buffers are managed in concurrent buffer pools). User tasks ($T_0, \cdots, T_n$ in Fig. 1) generate data items and insert them into a CAL aggregator along with the target locale. When a buffer is full, it is sent to the destination locale, and the data items included in the buffer are processed there. Since, as stated above, CAL only provides the basic aggregation functionality, details are left to the user. For example, the same-locale buffers indicated by dashed lines in Fig. 1 are optional and can be used if local aggregation is beneficial.

In the remainder of this section, we briefly cover the facets of CAL's design. The simplified version of the API of CAL, which will be discussed in more detail in later subsections, is summarized in Fig. 2. The API consists of two classes, the `Aggregator` that manages buffers and provides aggregation API and the `Buffer` class that encapsulates all details of a thread-safe buffer.

A. Minimalism

Whereas the software cache of Ferguson and Buettner [11] aggregates individual PUT and GET operations, CAL aggregates individual generic units of data defined by the user. CAL strives for maximizing code-reuse, and as such, it is generic on the type of data to be aggregated. The `Aggregator` is generic on the type specified by the user, which will be the type of data to aggregate.

```
// Create an aggregator that aggregates integers
var aggregator = new Aggregator(int);
```

The user can then begin aggregating data via `aggregate`, which will return a `Buffer` if that task took the last available slot in the buffer for the requested destination. If a buffer has been returned, the user can then apply their own coalescing and data processing. Once the user is finished they must invoke the `done` function on the buffer so that it can be recycled by the `Aggregator`.

```
// Aggregate the integer '1' to Locale 1
const data = 1;
const loc = Locales[1];
var buf = aggregator.aggregate(data, loc);
if buf != nil {
  handleBuffer(buf, loc);
  buf.done();
}
```

When manually flushing the `Aggregator`, it can be handled in the same way as if a buffer was returned via aggregating data.

```
forall (buf, loc) in aggregator.flush() {
  handleBuffer(buf, loc);
}
```

Besides adjusting the inner workings of a CAL `Aggregator` such as buffer size or the number of buffers in buffer pools, the above three snippets of code represent the total complexity of using CAL.

B. Distributed

In Chapel, class instances are stored on the heap of the locale that it is allocated on, and these instances do not migrate to other locales during their lifetime. Hence accesses such as class fields to class instances allocated in remote memory will result in an implicit GET operation to the locale it is hosted on. To eliminate this implicit communication, the `Aggregator` makes use of a process called privatization where a deep-copy of the data structure is created and managed on each locale, where each access of the `Aggregator` is forwarded to its privatized class instance. Privatization allows us to access data in a *locale-private* manner such that updates to the class fields can be performed independently of all other locales. Hence each `Aggregator` instance stores their own `destBuffers`, the destination buffers, and `pool`, the memory pool for recycling buffers. The end result is that the user is able to access the `Aggregator` without needing to worry about locality and can use it in a very intuitive manner.

```
on Locales[1] {
  const data = 1;
  const loc = Locales[0];
  var buf = aggregator.aggregate(data, loc);
  if buf != nil then handleBuffer(buf, loc);
}
```

C. Parallel-Safe

As Chapel is a language focusing on parallelism, we ensure that CAL is both thread-safe and scalable. The `Aggregator` is safe to call from multiple tasks as well as locales, making it easy to use with Chapel's concurrent constructs such as in `forall` loops.

```
forall i in 1..N {
  const loc = Locales[1];
  var buf = aggregator.aggregate(i, loc);
  if buf != nil then handleBuffer(buf, loc);
}
```

III. Implementation

Using privatization, the `Aggregator` is privatized such that an instance representing the local part of its data structures is created on each locale. Each instance is *locale-private* in that mutations are performed independently and locally, on the local part of the distributed data structure. In summary, the privatized instances are referenced via a privatization identifier, or *pid*, which is the index into a run-time data structure, the privatization table. The privatization table is replicated across all locales such that the pid associated with an instance on one locale will also refer to an instance on another locale. To enable efficient retrieval of the pid, we *record-wrap* it by storing it in a `record`, which is equivalent to a C struct that can be passed by value. By making use of `forwarding`, we can forward any and all method calls and field accesses to the privatized instance associated with the pid. An additional compiler optimization, called *remote-value forwarding*, ensures that the record-wrapper of an `Aggregator` is always handled by-value, eliminating most communication involved with retrieving the pid. With the combination of record-wrapping, forwarding, and remote-value forwarding, the `Aggregator` can be used seamlessly with Chapel's language constructs and from multi-locale contexts.

When an `Aggregator` is constructed, we create a memory pool for recycling buffers for all destinations. Each locale has a destination buffer for itself to handle cases where data is aggregated for local consumption. In the current implementation, we utilize a synchronized free list that enables obtaining buffer via `getBuffer` and recycling an existing buffer via `recycleBuffer(buf : Buffer(dataType))`; `getBuffer` will create buffers on-demand with the option of limiting the number of buffers allocated at any given time. The `Buffer` itself is a simple class wrapper for a Chapel array with three additional fields. The `claimed` field is an atomic counter used to claim an index in the buffer; the `filled` field is an atomic counter use to keep track of the number of finished stores into the buffer; and the `stolen` field is used to keep track of whether or not the buffer is about to be flushed.

When the buffer is about to be recycled, we explicitly avoid *sanitizing* the buffer (by calling `reset()`), performed by resetting all fields to their default values, until it is about to be returned from the memory pool for reasons that we discuss later.

```
proc Buffer.done() {
  on this do pool.recycleBuffer(this);
}
```

Sanitizing the buffer is performed in a specific order, so it is imperative that each atomic write is performed with sequential consistency, the default for atomic read and writes in Chapel.

```
proc Buffer.reset() {
  stolen.write(false);
  filled.write(0);
  claimed.write(0);
}
```

Appending to the buffer involves looping until satisfying a specific condition. Upon each iteration of the loop, the task will read the current destination buffer and perform a *fetch-and-add* on the `claimed` counter to 'claim' an index. If the claimed index is not within a valid range, the task will yield the current thread it is multiplexed on and loop again. When the index is considered valid, the task is safely able to write its value into that position in the buffer. After the task completes its write, it

will perform another fetch-and-add, this time on the `filled` counter, and if that task is the last to fill the buffer it will attempt to 'steal' the buffer for flushing. If it can successfully steal the buffer, it swaps out the buffer and returns it to the user for processing.

```
proc aggregate(data : dataType, loc : locale) :
       Buffer(dataType) {
  while true {
    // Get current buffer
    var buf = destBuffers[loc.id];
    // Claim an index in the buffer
    var idx = buf.claimed.fetchAdd(1);
    // Buffer is full or going to be flushed soon
    // Yield and try again
    if idx >= buf.cap {
      chpl_task_yield();
      continue;
    }
    // Store and notify we have filled up another slot
    buf[idx] = data;
    var nFilled = buf.filled.fetchAdd(1) + 1;
    // If we are the last to fill, see if we can steal
    if nFilled == buf.cap &&
       !buf.stolen.testAndSet() {
      // Swap buffers and return old for processing
      destBuffers[loc.id] = pools[loc.id].getBuffer();
      return buf;
    }
    // Not the last to fill buffer or already stolen
    return nil;
  }
}
```

Flushing is performed on each locale's privatized instance. Flushing the privatized instance is performed by iterating in parallel over all destination buffers, and for each buffer, we attempt to steal it. Stealing the buffer involves performing an atomic *test-and-set* on the *stolen* flag. If we successfully set the flag, we can ensure that no other task can steal the buffer. We then atomically *exchange* the `claimed` counter's current value with the buffer's capacity to ensure that future tasks will fail their fetch-and-add on the buffer, then swap out the buffer. Knowing the number of currently claimed indices in the buffer, we spin until the `filled` count is equal to the value we exchanged for. Once all tasks have finished their writes, we can safely yield the buffer to the user for processing and to notify when they are `done`.

```
iter flush() : (Buffer(dataType), locale) {
  // Spawn a task on each locale
  coforall loc in Locales do on loc {
    // Get privatized instance.
    var instance =
      chpl_getPrivatizedCopy(this.type, pid);
    // Attempt to flush all buffers in parallel
    forall loc in Locales {
      var buf = destBuffers[loc.id];
      // Try to steal the buffer
      if !buf.stolen.testAndSet() {
        // Simultaneously obtain the claimed indices
        // and prevent newer tasks from claiming more.
        var claimed = buf.claimed.exchange(buf.cap);
        if claimed > 0 {
          // Swap buffers
          destBuffers[loc.id] =
            pools[loc.id].getBuffer();
          // Wait for claimed stores to complete
          buf.filled.waitFor(claimed);
          // Yield buffer and locale for processing
          yield (buf, loc);
        } else {
          // Reset buffer
          buf.reset();
        }
      }
    }
  }
}
```

Due to having per-destination buffer pools, we can ensure that no task that is delayed for an extended duration of time can end up writing to a buffer that gets recycled for another destination. By sanitizing the buffer immediately before its use, we can ensure that no similarly delayed task can write to a buffer that is sitting in the buffer pool, as the `claimed` counter will still show as full. The buffer pools are not just needed for efficiency, but also to eliminate potential race conditions. Currently, buffers cannot be safely reclaimed without more advanced memory reclamation techniques like epoch-based reclamation, which is possible in Chapel despite lack of thread-local storage [13], or quiescent state-based reclamation, which is an in-progress effort [14] to implement in Chapel. Usage of hazard pointers [15] is also another option that can be considered, which has its own trade-offs [16], but such explorations are reserved for future work.

IV. Examples

CAL can be deployed wherever data parallelism cannot be exploited. Iterating over a distributed array while updating another distributed array is an example of a situation where CAL can be employed. Another example is when the user may need to spawn remote tasks to perform a complex operation of some distributed array. In the examples below, we show how we go from the naive version to an aggregated one.[1] As a disclaimer, the code examples provided are generally simplified from the actual implementation, and merely serve as high-level descriptions of the algorithms used.

A. Histogram

For this example, we use a modified version of Chapel's study on distributed histograms [17]. Consider the situation where you have one distributed array that contains values that you wish to count by category such as by occurrence. As well, assume that the table that keeps track of the number of occurrences for each category is also distributed. Now assume that both the array of values and the array of occurrences are distributed differently, making the access pattern irregular.

[1]We leave out the final flush of the buffer as it is handled in the exact same way as shown in Section II-A

```
// Distributed table of occurrences
const tableSpace = {0..#tableSize};
const tableDomain = tableSpace
        dmapped Cyclic(startIdx=tableSpace.low);
var table : [tableDomain] atomic int;
// Distributed values to categorize
const valueSpace = {0..#numValues};
const valueDomain = valueSpace
        dmapped Block(valueSpace);
var values : [valueDomain] int;
```

Furthermore, the array of values can be randomized so that their values are valid indices into the table of occurrences.

```
fillRandom(values);
forall value in values {
   value = mod(value, tableSize);
}
```

Accessing the histogram in a naive fashion is straightforward.

```
forall value in values {
        table[value].add(1);
}
```

In Chapel, iterators are implemented by every iterable data type, and in the case of Chapel's distributions like `Block` and `Cyclic`, they spawn a task on each core on each locale where each task will yield elements hosted on that locale by reference. While the iterator is ideal in situations where we have data parallelism, such as when operating on the reference directly, it is inefficient when accessing remote memory, such as to increment the count of occurrences in the table.

Accessing the histogram using CAL isn't as straight-forward, but it is easy to reason about. First, we define a small function to handle the buffer, which will perform coalescing and processing of the buffer on the target locale. Coalescing is performed by locally combining all repeated occurrences. After coalescing, the original buffer is no longer required and can be recycled for later use. Finally, the new coalesced buffer, which could be smaller than the original, is copied and processed remotely.

```
proc handleBuffer(buf : Buffer(int), loc : locales) {
  // Coalesce the multiple increments
  var subdom = tableDomain.localSubdomain(loc);
  var occurrences : [subdom] int;
  for idx in buf do counters[idx] += 1;

  // Finished coalescing buffer... Recycle buffer...
  buf.done();

  // Dispatch coalesced data on target locale
  on loc {
  // Copies occurrences array in bulk to target locale
    var _occurences = occurences;
    for (cnt, idx) in zip(_occurences,
                          _occurencies.domain) {
       if cnt > 0 then table[idx].add(cnt);
    }
  }
}
```

Then we perform iteration, in parallel, over the distributed array of values in a similar fashion, but we wrap it in a 'sync' block so that we wait for all spawned tasks to finish before proceeding (this is a form of termination detection [18]). This is required as, even though the `forall` loops wait for its spawned tasks to finishes, they do not wait for asynchronous tasks spawned in their scope to finish. When the buffer is full we spawn an asynchronous task via Chapel's `begin` construct to handle flushing the buffer.

```
var aggregator = new Aggregator(int);
sync forall value in values with (in aggregator) {
  const loc = table[value].locale;
  var buf = aggregator.aggregate(value, loc);
  if buf != nil then begin handleBuffer(buf, loc);
}
```

B. Dual Hypergraph Generation

In more complex irregular applications where remote accesses are not as simple as an increment of an atomic counter, such as one which requires spawning a remote task to accomplish, is where CAL shows its utility and truly shines. Graph generation is an application that is the subject of numerous publications and studies and has many uses such as the study of community structure [19]. CAL was originally developed for hypergraph generation, so we use it as our next example. A *hypergraph* [20] is a generalization of a graph that allows edges to contain any number of vertices; a graph is a hypergraph where edges connect exactly two vertices. A *dual hypergraph* is a hypergraph that allows vertices to contain any number of edges, with the constraint that if a vertex connects to an edge then that edge also connects back to that vertex. Put simply, a dual hypergraph provides a bidirectional mapping from vertices and edge.

As contention is unlikely due to the random sampling of vertices and edges, we make use of a *Test-and-Test-and-Set* spinlock over Chapel's `sync` variable due to the favorable performance under low contention.

```
record SpinLock {
  var lock : chpl__processorAtomicType(bool);

  proc acquire() {
    // Fast path
    if lock.testAndSet() == false then return;
    // Slow path
    while true {
       var ret = lock.peek();
       if ret == false && lock.testAndSet() == false {
         return;
       }
       chpl_task_yield();
    }
  }
  proc release() {
    lock.clear();
```

```
  }
}
```

For this example, we use an adjacency list to represent the *neighbors* of each vertex and edge.

```
record Edge {
  var verticesDom = {0..-1};
  var vertices : [verticesDom] int;
  var lock : SpinLock;

  proc addVertex(v) {
    lock.acquire();
    vertices.push_back(v);
    lock.release();
  }
}

record Vertex {
  var edgesDom = {0..-1};
  var edges : [edgesDom] int;
  var lock : SpinLock;

  proc addEdge(e) {
    lock.acquire();
    edges.push_back(e);
    lock.release();
  }
}
```

As each connection, or *inclusion*, between an edge and vertex requires adding each to the others' adjacency list, we distribute both over non-intersecting sets of locales.

```
const vertexSpace = {0..#numVertices};
const vertexDom = vertexSpace dmapped Block(
  boundingBox=vertexSpace,
  targetLocales = Locales[0..numLocales by 2]
);
var vertex : [vertexDom] Vertex;

const edgeSpace = {0..#numEdges};
const edgeDom = edgeSpace dmapped Block(
  boundingBox = edgeSpace,
  targetLocales = Locales[1..numLocales by 2]
);
var edge : [edgeDom] Edge;
```

We use a fast bipartite hypergraph variant of Erdős-Rényi [21], [22], originally introduced for graphs in [23]. In the algorithm, we compute the number of inclusions to generate to between vertices and edges selected at random with replacement.[2] For each random vertex and edge selected, we add them to each others' adjacency list. For simplicity, we create two distributed arrays of the same shape and fill them both with random values to `zip` over. In Chapel, zipping over two arrays of the same shape and distribution will create tuples containing pairs of elements from the arrays, described in more detail in [25].

[2]In a real-world application, one may require a "coupon-collectors adjustment" [24], which we omit for brevity.

```
var space = {0..#numInclusions};
var dom = space dmapped Block(space);
var verticesRNG : [dom] real;
var edgesRNG : [dom] real;
fillRandom(verticesRNG);
fillRandom(edgesRNG);

forall (vRNG, eRNG) in zip(verticesRNG, edgesRNG) {
  var v = round(vRNG * (numVertices - 1)) : int;
  var e = round(eRNG * (numEdges - 1)) : int;
  on vertex[v] do vertex[v].addEdge(e);
  on edge[e] do edge[e].addVertex(v);
}
```

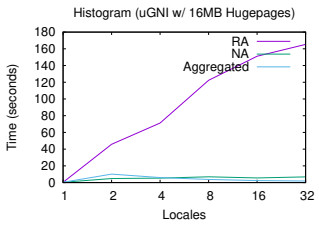
While iterating over the already-computed random values is efficient, accessing random indices in two separate distributed arrays is not. By making use of Chapel's "data-driven" on-clauses, we can move the computation to the locale where the data is hosted rather than performing multiple PUT and GET operations. However since `on` statements are entirely synchronous, the edge must be added to the adjacency list of the vertex before the vertex can be added to the adjacency list of the edge. While it is possible to make these asynchronous, the overhead of remote task creation and migration, the size of the tasks themselves, and the sheer number of tasks can quickly overwhelm the cluster.

In the aggregated case, we can aggregate data destined for each locale using the `Aggregator` as expected. First, we determine the type of data we wish to aggregate, which in this case would be the vertex and edge. Since we are aggregating data to both vertices and edges, we also create a descriptor to differentiate between them.

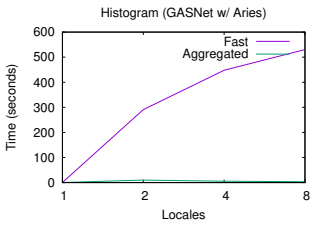
```
enum Inclusion {
  Vertex,
  Edge
}

type dataType = (int, int, Inclusion);
```

As well, we define the handler for the buffer, which will handle adding to the adjacency list of the specified vertex or edge. Note that we do not perform coalescing as the chance of there being multiple insertions into the adjacency list of the same vertex or edge in a single buffer is very small, and shrinks as the graph we're trying to generate GETs larger.

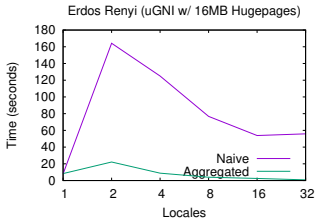
```
proc handleBuffer(buf : Buffer(dataType),
                  loc : locale) {
  on loc {
    forall (src, dest, srcType) in buf {
      select srcType {
        when Vertex {
          vertex[src].addEdge(dest);
        } when Edge {
          edge[src].addVertex(dest);
        }
      }
    }
    buf.done();
```



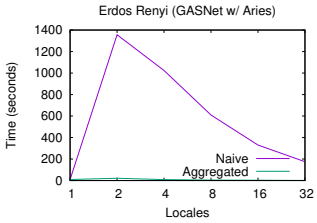
(a) uGNI Histogram



(b) GASNet Histogram



(a) uGNI Erdos Renyi



(b) GASNet Erdos Renyi

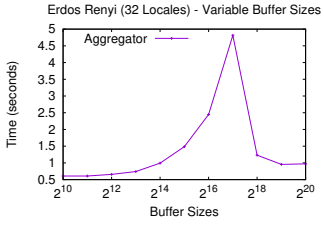
```
  }
}
```

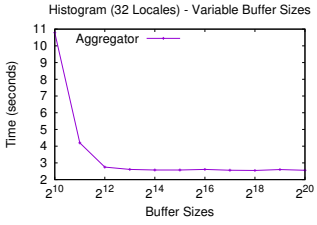
Finally, we iterate over the `zip` of both pre-computed random numbers like in the naive.

```
forall (vRNG, eRNG) in zip(verticesRNG, edgesRNG) {
  const v = round(vRNG * (numVertices - 1)) : int;
  const vLoc = vertex[v].locale;
  const vData = (v, e, Inclusion.Vertex);
  var vBuf = aggregator.aggregate(vData, vLoc);
  if vBuf != nil then begin handleBuffer(vBuf, vLoc);

  const e = round(eRNG * (numEdges - 1)) : int;
  const eLoc = edge[e].locale;
  const eData = (e, v, Inclusion.Edge);
  var eBuf = aggregator.aggregate(eData, eLoc);
  if eBuf != nil then begin handleBuffer(eBuf, eLoc);
}
```

V. Evaluation

Performance benchmarks are performed on Intel Broadwell compute nodes of a Cray-XC50 cluster. For the communication layer, we use both uGNI with 16MB hugepages and GASNet with the Aries substrate and 'fast' segment. For the tasking layer, we use qthreads as it is the most optimized and highest performance tasking layer available. For the memory management layer, we use jemalloc as it provides the most performance. All benchmarks have been tested using Chapel pre-release version 1.18.0, hash 732b550d52. We compile the benchmarks with the `--fast` flag, which instructs the Chapel compiler to perform all optimization on the Chapel source code, turn off any and all safety run-time checks such as out-of-bounds checking, and optimizes the generated C code, which Chapel gets compiled down to, with the highest optimization settings for the C compiler used. We allocate compute nodes, each with 44 cores, by powers of 2 up to 32 nodes, and so performance results are presented in a logarithmic base-2 scale.





(a) Erdos Renyi at 32 locales with variable buffer sizes (uGNI)

(b) Histogram at 32 locales with variable buffer sizes (uGNI)

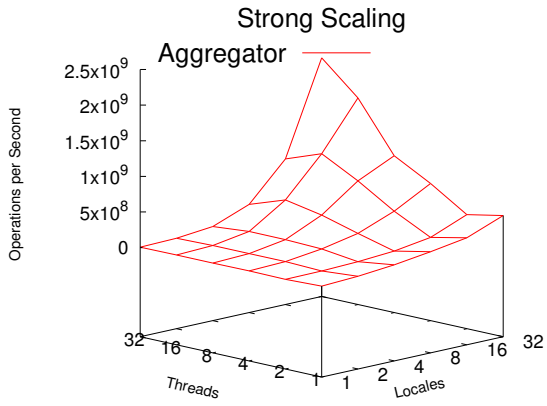


Fig. 6: Aggregation with varying locales and threads per locale

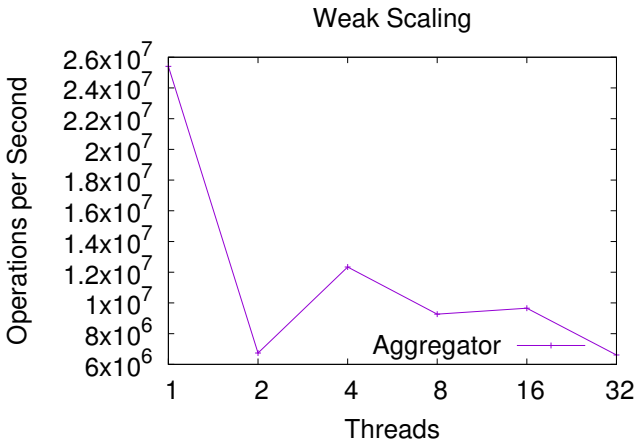


Fig. 7: Aggregation at single locale showing contention

A. Histogram

For our first benchmark, we use the histogram examples shown in Section IV-A and show the performance of both naive and aggregated implementations under various configurations. The naive histogram relies heavily on the performance of remote atomic operations, which the uGNI communication layer will make use of *network atomics*, which are implemented with atomic memory operators, or *AMO*, descriptors which

perform remote direct memory access, or *RDMA*, that bypass the hardware that normal put and get operations go through. In other words, network atomics are extremely well optimized by the hardware and the uGNI communication layer. When not using network atomics, the Chapel compiler turns `on` statements which do not require synchronization or communication into *fast* on statements. These fast `on` statements, which are executed on the dedicated progress threads instead of becoming its own task, are labeled as RA. Finally, the aggregated performance is labeled as *Aggregated*. The portion of the code being profiled in this benchmark is the incrementing of the atomic counters in the table, and the aggregated and coalesced version. The performance results are shown in Fig. 3a, and *RA* shows degrading performance as we increase the number of locales, showing no signs of weak scaling. *NA* shows a considerable boost in overall performance by approximately 25 times over *RA* at 32 locales, but plateaus and also shows no signs of weak scaling. *Aggregated* not only beats *RA* by almost two orders of magnitude, but also beats *NA* by approximately 4 times.

Unfortunately, GASNet, even with the Aries substrate and fast segments, significantly lags behind uGNI in terms of communication in general, but aggregation and coalescing using CAL seems to fix this problem. Even though GASNet uses Aries, it does not yet support network atomics, and so it settles for fast *on* statements, labeled as *Fast*. As shown in Fig. 3b, performance is significantly worse, and due to issues that are likely with the implementation of the GASNet communication layer, the benchmark could not complete without crashing after 8 locales.[3] *Aggregated* shows that regardless of the communication layer used, CAL proves to significantly improve overall performance.

B. Erdős-Rényi

For this benchmark, we are testing non-trivial operations which require the use of task migration for even reasonable performance. In this case, we cannot use any specialized hardware or optimization like AMOs and RDMA. This type of benchmark tests the benefits of aggregating multiple units of data and coalescing them into active message. The naive version is labeled *Naive* and the aggregated version is labeled *Aggregated*. As can be seen in Fig. 4a and Fig. 4b, aggregation not only enables the code to scale, but it also reduces the significant jump in execution time due to introducing communication at 2 locales, as the amount of communication is greatly reduced. GASNet is still slower than uGNI at first, but begins to catch up at 32 locales, and uGNI shows a plateau at 16 locales, likely due to the bottleneck of communication. *Aggregated* beats GASNet *Naive* by over two orders of magnitude at approximately 180 times, and almost beat uGNI *Naive* by almost two orders of magnitude at approximately 70 times. Erdős-Rényi helps to highlight the issue of fine-grained communication as there is hardly any contention due to the random selection of vertices and edges.

[3]This is the exact same benchmark that is executed with uGNI.

C. Aggregation Buffer Sizes

In the above benchmarks, we use an arbitrary constant buffer size of 1M elements. In the following benchmark, we test the performance of the `Aggregator` with variable buffer sizes from powers of two between $2\hat{1}0$ to $2\hat{2}0$ at 32 locales to determine how performance varies based on the size of the buffer.[4] For the Histogram, shown in Fig. 5b, performance significantly improves when we jump 1024 to 4096 and plateaus onward. For Erdős-Rényi, shown in Fig. 5a, we see a significant spike in execution time between 32K to 256K, before returning to normal. This is likely due to the maximum amount of data that gets sent over the network at any given time and shows that even though CAL is easy to use, careful profiling is required for obtaining the most optimal performance.

D. Overhead of Aggregation

In this benchmark, we extract the raw performance CAL by profiling the overhead of aggregating data. When the `Buffer` is returned from the `Aggregator`, we immediately invoke `done()` and continue aggregating, eliminating the overhead of any communication. The results shown in Fig. 6 show the scalability of the aggregation across multiple locales in the best case. Also analyzed is the performance at one locale shown in Fig. 7, which shows raw contention on the buffer and shows a scalability bottleneck. It is believed that this is caused by a combination of false sharing and poor NUMA locality, as Chapel currently does not offer satisfactory NUMA-awareness tools.

VI. Discussion

CAL significantly improves benchmarks involving irregular access patterns, which is currently being studied in Chapel. On Cray's Aries NIC and uGNI communication layer, network atomics have hardware-specific support and so CAL is not meant to compete with them; however, even in the case of irregular access involving network atomics, CAL has proven itself, and even more so in the cases where the user finds themselves using remote execution.

While the `Aggregator` is meant to be minimal, it would be more aesthetically pleasing if the user could provide their own functions to be invoked to handle the buffer rather than having to check and handle the buffer after each called to aggregate. Once remote value forwarding is available for user data types, the user will no longer need to explicitly specify the forall-intent as `in`. To show the ideal look and feel of the library, the histogram without coalescing can be seen below.

```
proc handleBuffer(buf : Buffer(int), loc : locale) {
  on loc do forall idx in buf do table[idx].add(1);
}
var aggregator = new Aggregator(int, handleBuffer);
forall value in values {
  aggregator.aggregate(value, table[value].locale);
```

[4]We lower the task stack sizes from the default 8MBs to 32KB to prevent running out of memory.

```
}
aggregator.flush();
```

However, due to incomplete support for first-class functions, forcing the user to handle aggregating the buffer is currently the most reasonable approach. However, with better support for first-class functions, it would enable things like automatic flushing of the buffer based on a heuristic like time and rate of change, similar to the AM++ framework.

The destination buffers in CAL are not performing as well as they could, primarily due to the `Aggregator`'s lack of NUMA awareness. While Chapel offers a very rich set of tools and abstractions for handling locality between compute nodes, support for NUMA is currently in its experimental phase. As Chapel does not yet support thread-local or task-local storage, it is currently not possible to use per-thread buffers to eliminate contention and the need for NUMA awareness. The ideal solution would involve having NUMA-specific destination buffers which would eliminate cache-line ping-ponging between NUMA nodes that occur from performing atomic operations.

One future work is to integrate CAL into Chapel's run-time, where thread-local and task-local storage and NUMA domains are accessible. There is also some incentive to examine whether CAL can work cooperatively with the software cache to avoid excessive invalidation due to its heavy usage of atomics. Furthermore, automatic flushing can become significantly easier and more efficient to implement with run-time support. Perhaps, in the end, aggregation can become more of a first-class construct instead of a user library.

VII. Conclusion

CAL provides a very simple and minimal approach to aggregating and coalescing user-defined data. CAL can offer a relatively low increase in program complexity compared to the significant performance improvement it brings. The library is written entirely in Chapel and as such makes use of the rich features and language constructs that Chapel offers, while also working with language constructs to make CAL as easy to use as possible. While there are issues with the language, such as the lack-luster support for first-class functions to prevent automatic flushing, in the future the library may be improved and integrated into the language itself as a first-class construct assuming significant demand. CAL will soon be released as an open source library either as a 'package' module or a mason third-party module.

Acknowledgment

We would like to thank Brad Chamberlain (Cray Inc.) for providing insight and valuable comments that impacted this work significantly. Furthermore, we would like to thank Cray Inc., and especially the Chapel development team, for providing computational resources to perform experiments shown in this paper.

References

[1] B. Chamberlain, D. Callahan, and H. Zima, "Parallel Programmability and the Chapel Language," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 3, pp. 291–312, Aug. 2007.

[2] "The Chapel Parallel Programming Language," https://chapel-lang.org/, Jul. 2018.

[3] U. Consortium *et al.*, "UPC language specifications v1. 2," 2005.

[4] J. J. Willcock, T. Hoefler, N. G. Edmonds, and A. Lumsdaine, "AM++: A generalized active message framework," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. ACM, 2010, pp. 401–410.

[5] T. Von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active messages: a mechanism for integrated communication and computation," in *ACM SIGARCH Computer Architecture News*, vol. 20, no. 2. ACM, 1992, pp. 256–266.

[6] D. Bonachea and J. Jeong, "Gasnet: A portable high-performance communication layer for global address-space languages," *CS258 Parallel Computer Architecture Project, Spring*, 2002.

[7] D. Bonachea and P. Hargrove, "GASNet Specification, v1. 8.1," 2017.

[8] Y. Sun, G. Zheng, L. V. Kale, T. R. Jones, and R. Olson, "A uGNI-based Asynchronous Message-driven Runtime System for Cray Supercomputers with Gemini Interconnect," in *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*. IEEE, 2012, pp. 751–762.

[9] K. B. Wheeler, R. C. Murphy, and D. Thain, "Qthreads: An API for programming with millions of lightweight threads," 2008.

[10] P. Argyroudis and C. Karamitas, "Exploiting the jemalloc memory allocator: Owning FirefoxâĂŹs heap," *Blackhat USA*, 2012.

[11] M. P. Ferguson and D. Buettner, "Caching Puts and Gets in a PGAS language runtime," in *2015 9th International Conference on Partitioned Global Address Space Programming Models (PGAS)*. IEEE, 2015, pp. 13–24.

[12] E. Kayraklioglu, M. Ferguson, and T. El-Ghazawi, "LAPPS: Locality-Aware Productive Prefetching Support for PGAS ," *ACM Transactions on Architecture and Code Optimization (TACO)*, 2018.

[13] L. Jenkins, "RCUArray: An RCU-Like Parallel-Safe Distributed Resizable Array," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2018, pp. 925–933.

[14] "Quiescent-State Based Reclamation - Overhaul [W.I.P]," https://github.com/chapel-lang/chapel/pull/8842.

[15] M. M. Michael, "Hazard pointers: Safe memory reclamation for lock-free objects," *IEEE Transactions on Parallel & Distributed Systems*, no. 6, pp. 491–504, 2004.

[16] T. E. Hart, P. E. McKenney, and A. D. Brown, "Making lockless synchronization fast: Performance implications of memory reclamation," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*. IEEE, 2006, pp. 10–pp.

[17] "test/studies/bale/histogram/histo-atomics.chpl," https://github.com/chapel-lang/chapel/blob/25bf5188b904300f444524198b7a9faba9b00180/test/studies/bale/histogram/histo-atomics.chpl.

[18] E. W. Dijkstra, W. H. Feijen, and A. M. Van Gasteren, "Derivation of a termination detection algorithm for distributed computations," in *Control Flow and Data Flow: concepts of distributed programming*. Springer, 1986, pp. 507–512.

[19] S. G. Aksoy, T. G. Kolda, and A. Pinar, "Measuring and Modeling Bipartite Graphs With Community Structure," *Journal of Complex Networks*, vol. 5, no. 4, pp. 581–603, mar 2017.

[20] C. Berge, *Hypergraphs: Combinatorics of Finite Sets*. Elsevier, 1989.

[21] J. C. Miller and A. Hagberg, "Efficient Generation of Networks with Given Expected Degrees," in *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2011, pp. 115–126.

[22] M. Winlaw, H. DeSterck, and G. Sanders, "An In-Depth Analysis of the Chung-Lu Model," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2015.

[23] P. Erdos and A. Rényi, "On the evolution of random graphs," *Publ. Math. Inst. Hung. Acad. Sci*, vol. 5, no. 1, pp. 17–60, 1960.

[24] T. G. Kolda, A. Pinar, T. Plantenga, and C. Seshadhri, "A Scalable Generative Graph Model with Community Structure," *SIAM Journal on Scientific Computing*, vol. 36, no. 5, pp. C424–C452, jan 2014.

[25] B. L. Chamberlain, S.-E. Choi, S. J. Deitz, and A. Navarro, "User-defined parallel zippered iterators in Chapel," in *Proceedings of Fifth Conference*

on Partitioned Global Address Space Programming Models, 2011, pp. 1–11.