AGGREGATION IN ARKOUDA

Elliot Ronaghan

July 20, 2021

AGGREGATION NEED

- Initial Arkouda development done in a few months by 1 developer, production ready in a year by 2 devs
  - Small team necessitated high level and easy to maintain code
    - Productivity benefits are one of the reasons Chapel was chosen over traditional HPC technologies

- This led to algorithms with lots of fine-grained communication
  - Initially, used 'unorderedCopy', which is efficient on Aries, but not other systems

- Sample performance for copying a local array to a remote array

```
// bulk
rArr = lArr;


// unordered
forall (r, l) in zip(rArr, lArr) do
  unorderedCopy(r, l);
```

Performance (MiB/s)		
config	**bulk**	**unordered**
Aries	8000.0	510.0
FDR IB	6000.0	2.0

USER AGGREGATION

- Added copy aggregators to Arkouda
  - Created for each task, must specify whether source or destination is remote

```
// bulk
rArr = lArr;


// unordered
forall (r, l) in zip(rArr, lArr) do
  unorderedCopy(r, l);


// aggregated
forall (r, l) in zip(rArr, lArr) with (var agg = new DstAggregator(int)) do
  agg.copy(r, l);
```

Performance (MiB/s)			
config	**bulk**	**unordered**	**aggregated**
Aries	8000.0	510.0	2275.0
FDR IB	6000.0	2.0	1850.0

CHAPEL INDEXGATHER SOURCE

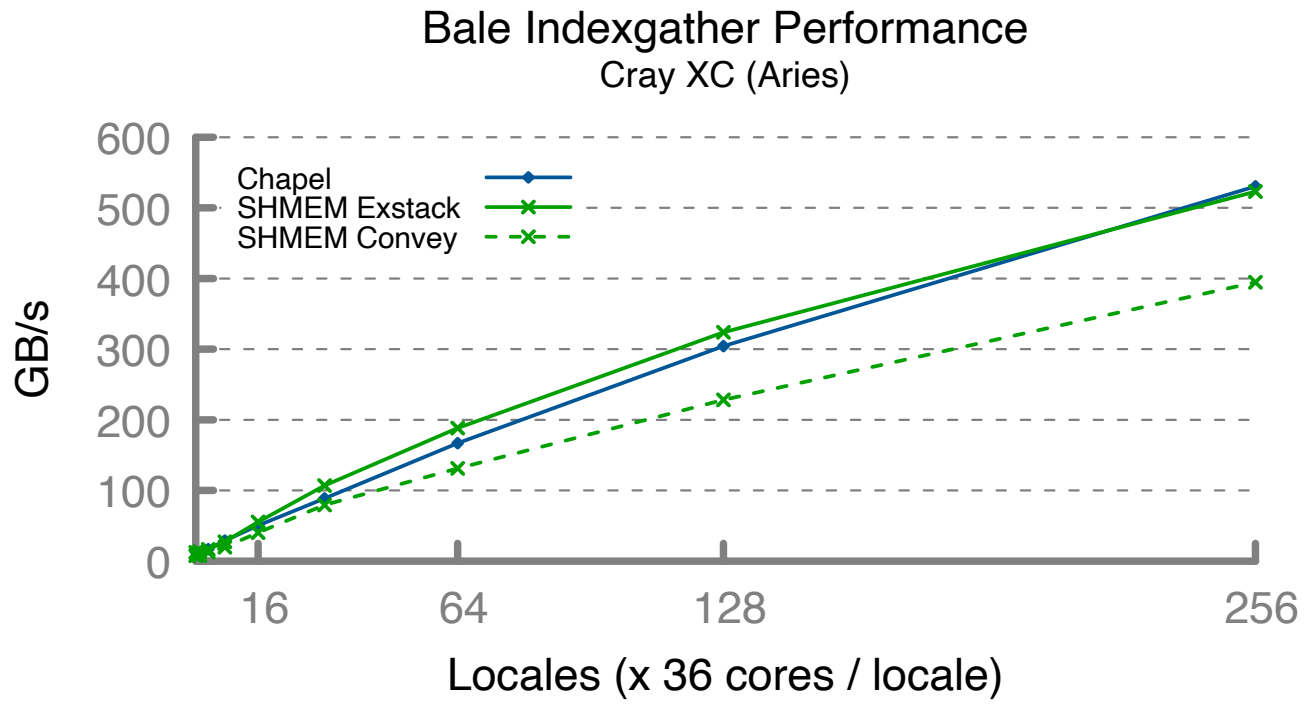
```
use BlockDist, Random, CopyAggregation;

const numTasks = numLocales * here.maxTaskPar;
config const N = 1000000; // number of updates per task
config const M = 10000;   // number of entries in the table per task

const D =  newBlockDom(0..# M*numTasks);
var A: [D] int = D;
const UpdatesDom = newBlockDom(0..# N*numTasks);
var Rindex: [UpdatesDom] int;

fillRandom(Rindex, 208);
Rindex = mod(Rindex, M*numTasks);
var tmp: [UpdatesDom] int;

// Unaggregated (though can be aggregated by compiler with --auto-aggregation)
forall (t, r) in zip (tmp, Rindex) do
  t = A[r];

// Manually aggregated
forall (t, r) in zip (tmp, Rindex) with (var agg = new SrcAggregator(int)) do
  agg.copy(t, A[r]);
```

SHMEM INDEXGATHER KERNEL

- https://github.com/jdevinney/bale_private/tree/master/src/bale_classic/apps/ig_src

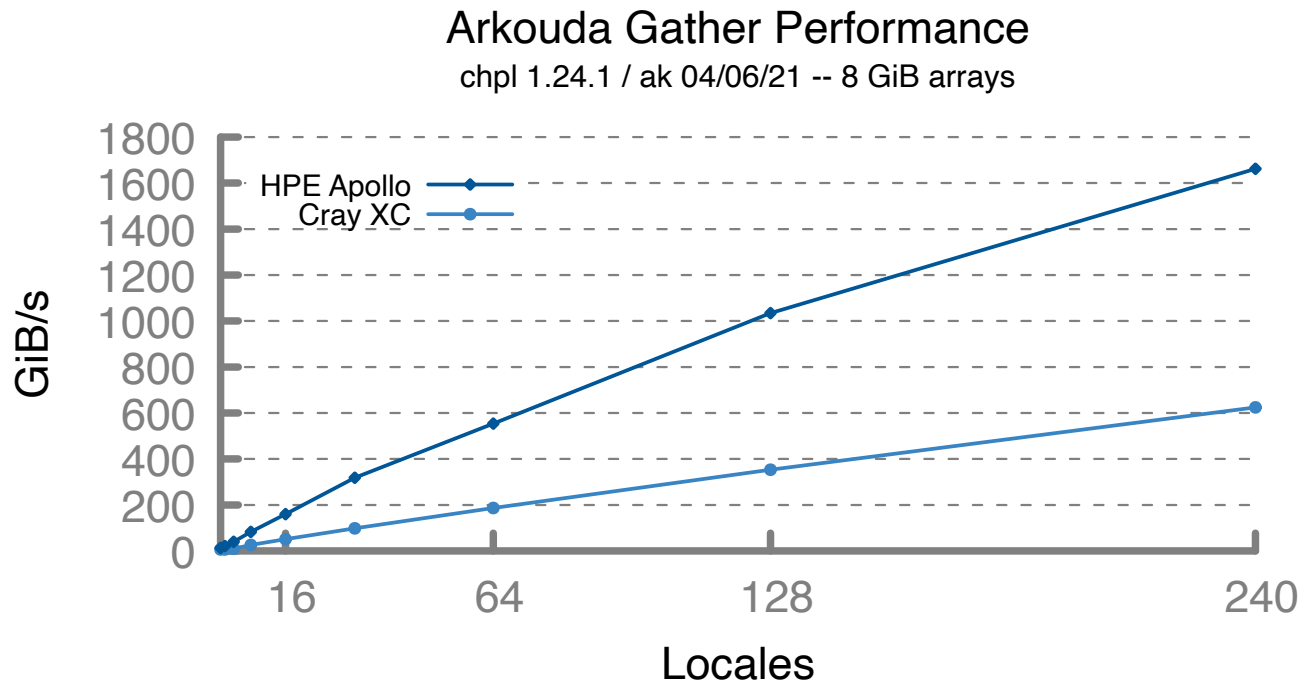
AGGREGATION PERFORMANCE

- For Bale indexgather we're competitive with Exstack up to 256 nodes on XC
  - This is from October 2020, we have since done ~600 node runs with similar results



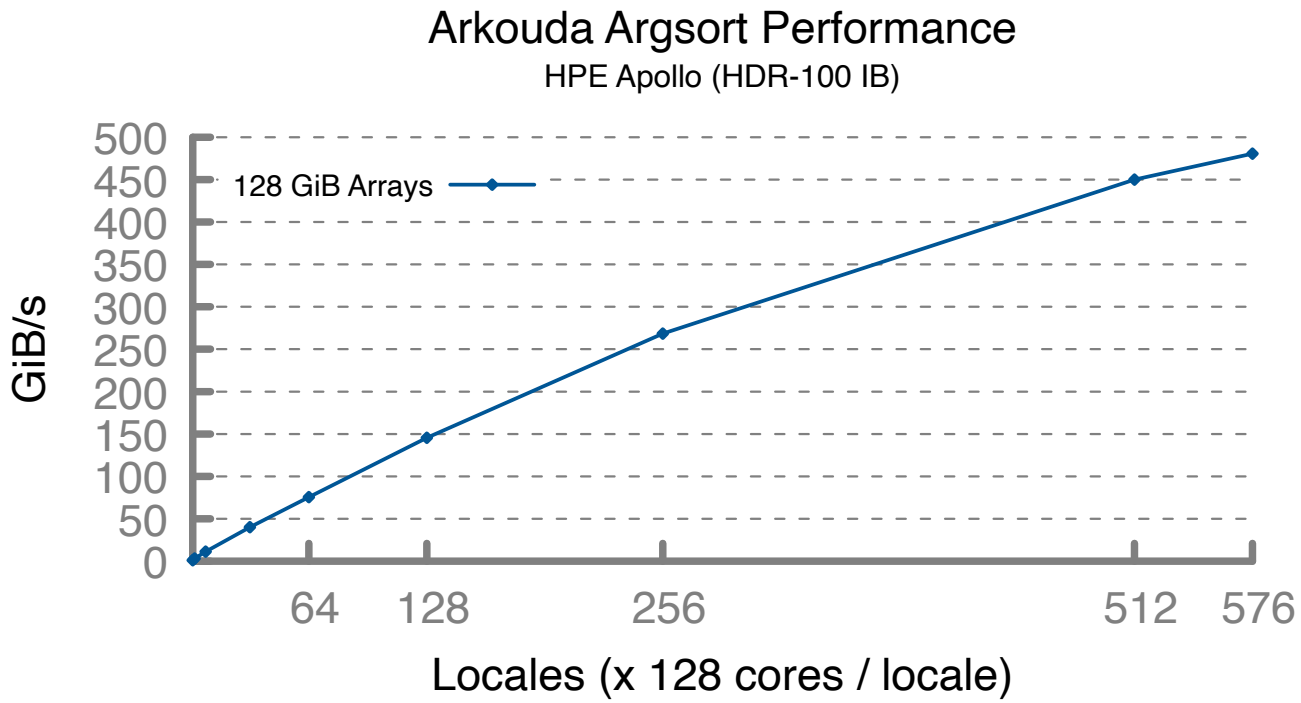
AGGREGATION PERFORMANCE

- For Arkouda gather, Chapel performance on HPE Apollo systems is well ahead of XC
  - This is from April 2021
  - Firmware and/or HPC-X SHMEM bugs hurt reference performance, so no direct comparisons



AGGREGATION PERFORMANCE

- For Arkouda Argsort, we have seen positive scaling using 576 nodes (~75K cores) on HPE Apollo
  - This is from June 2021



AGGREGATOR REQUIREMENTS AND EXAMPLES

- Aggregators must be created on a per-task basis (they are not parallel safe)
- Either the destination or source must be local
  - This requirement may be relaxed over time
- Aggregators are not consistent until they go out of scope or an explicit 'flush()' is performed
  - Think of it as a non-blocking or async operation

- Examples in Arkouda:
  - Simple numeric/bool gather:
    - https://github.com/Bears-R-Us/arkouda/blob/4d32ebe/src/IndexingMsg.chpl#L164-L166
  - More complicated string gather (requires localization of source index)
    - https://github.com/Bears-R-Us/arkouda/blob/4d32ebe/src/SegmentedArray.chpl#L244-L281
  - SPMD aggregation in RadixSort (uses manual flush)
    - https://github.com/Bears-R-Us/arkouda/blob/4d32ebe/src/RadixSortLSD.chpl#L230-L240


THANK YOU
elliot.ronaghan@hpe.com


NEXT STEPS BACKUP
SLIDES

NEXT STEPS

- Autotune aggregation buffer sizes
  - Manually tuned for Aries and InfiniBand, but not for other networks
  - Ideal setting also depends on scale and CPU architecture

- Reduce memory overhead, per peer/destination buffers have high usage
  - Lower than SHMEM since we use a process per node, so 1/numCores as many peers
  - Not a priority with high memory nodes, but important for general adoption and scale
  - Have a prototype multi-hop implementation, reduces memory overhead by 16x, achieves 60% the performance

- Move aggregators from Arkouda to Chapel standard library
  - In doing so eliminate need to specify whether src/dst is remote
  - Allow for arbitrary user-facing aggregation instead of just copy aggregation

- Do more comparisons to Bale reference versions
  - Should be a good litmus test for arbitrary aggregation completeness while allowing perf comparisons

USER DEFINED AGGREGATION

- Copy aggregators are all that's needed for Arkouda
  - But aggregation is useful in general, want to provide a way for anybody to access user defined aggregators
    - https://github.com/chapel-lang/chapel/issues/16963

- Chapel has some support for first-class-functions / closures, which should help
  - FCF have not been a high priority, so they need some more work and aren't a stable part of the language

- Sketch:

```
// Indexgather
proc copy(ref lhs, rhs) { lhs = rhs; }

type CopyAgg = Aggregator(lhsT=int, rhsT=int, flushFunc=copy);

forall (t, r) in zip (tmp, Rindex) with (var agg = new CopyAgg()) do
  agg.copy(t, A[r]);
```

```
// Histogram
proc add(ref lhs, rhs) { lhs.add(rhs); }

type AtomicAgg = Aggregator(lhsT=atomic int, rhsT=int, flushFunc=add);

forall r in Rindex with (var agg = new AtomicAgg()) do
  agg.add(A[r], 1);
```

IMPLEMENTATION BACKUP SLIDES

INITIAL AGGREGATION IMPLEMENTATION

- Chapel features make aggregation pretty straightforward to implement
  - on-statements (active messages) makes flushing simple
    - Does not require “symmetrically one-sided updates”
    - Does not require all threads/tasks to participate to make progress
  - User-level tasks (tasks multiplexed over system threads) makes the implementation efficient

```
// Migrate execution to the remote node
on Locales[loc] {
  ...
}
```

INITIAL AGGREGATION IMPLEMENTATION

```
config const bufferSize = 4096;


/*
 * Aggregates copy(ref dst, src). Optimized for when src is local.
 * Not parallel safe and is expected to be created on a per-task basis
 * High memory usage since there are per-destination buffers
 */
record DstAggregator {

  type elemType;
  var buffer: [LocaleSpace][0..#bufferSize] (addr, elemType);
  var bufferIdxs: [LocaleSpace] int;
```

INITIAL AGGREGATION IMPLEMENTATION

```
inline proc copy(ref dst: elemType, srcVal: elemType) {
  // Get the locale of dst and the local address on that locale
  const loc = dst.locale.id;
  const dstAddr = getLocalAddr(dst);


  // Get our current index into the buffer for dst's locale
  ref bufferIdx = bufferIdxs[loc];


  // Buffer the address and desired value
  buffer[loc][bufferIdx] = (dstAddr, srcVal);
  bufferIdx += 1;


  // If full, flush
  if bufferIdx == bufferSize then
    flushBuffer(loc, bufferIdx);
}
```

INITIAL AGGREGATION IMPLEMENTATION

```
proc flushBuffer(loc: int, ref bufferIdx) {

  // Migrate execution to the remote node
  on Locales[loc] {
    // GET the buffered dst addrs and src values, and assign
    var localBuffer = buffer[loc][0..#bufferIdx];
    for (dstAddr, srcVal) in localBuffer do
      dstAddr.deref() = srcVal;
  }
  bufferIdx = 0;

}
```

CURRENT AGGREGATION IMPLEMENTATION

- Have optimized implementation since then, but still less than ~100 lines
  - https://github.com/Bears-R-Us/arkouda/blob/661fab1/src/CommAggregation.chpl#L32-L123
    - More manual memory management
    - Cache remote allocations
    - Yield periodically