Towards A General Aggregation Framework in Chapel

Oliver Alvarado Rodriguez, Engin Kayraklioglu, Bartosz Bryg, Mohammad Dindoost, David Bader, and Brad Chamberlain

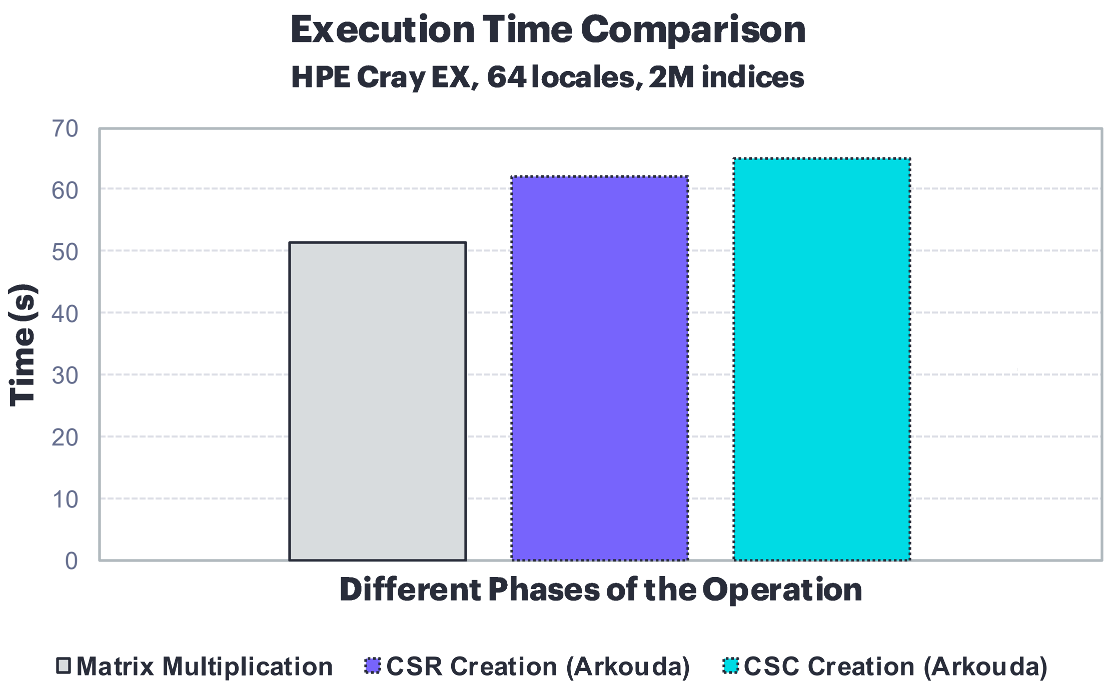
October 10, 2024

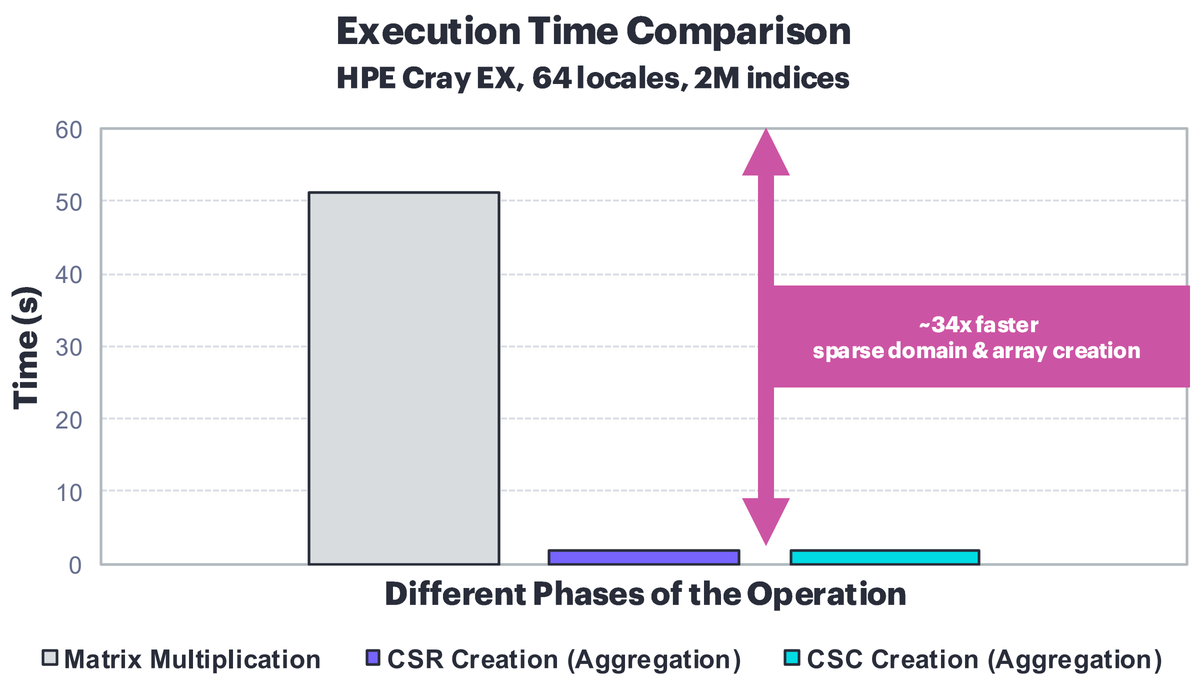
Introduction

- In distributed-memory, parallel programming, one of the most common bottlenecks is the quantity of communications performed between remote, parallel tasks.
  - Increases in communication are especially noticeable in irregular workloads such as those that use sparse matrices and graphs.
  - For this talk the focus will be on optimizing sparse matrix creation but also taking a minor look at sparse matrix multiplication and RMAT matrix generators for background.
- Chapel has the CopyAggregation module to facilitate the batching of fine-grained communications for array-specific operations.
  - However, there is not a more general framework to support the aggregation of more user-specific operations.
  - We will introduce a framework prototype for more general aggregation.
  - The work for this talk did not only involve the framework prototype.
    - Compressed sparse layouts in Chapel were modified to add parallel safety.
    - The sparse domain buffer functionality, for faster adding of indices into sparse domains, was updated to pass uniqueness and sorted flags.

Motivating Use Case – Sparse Operations in Arkouda

```
1. n = 20                                                    # Matrices will be of size (2**n X 2**n).
2. rows, cols = rmat(n)                                      # Create 1D pdarrays of row and column indices.
3. vals = ak.randint(1, len(rows), len(rows))                # Generate random values.
4. A = create_sparse_matrix(2**n, rows, cols, vals, "CSR")   # Create sparse matrix with CSR layout.
5. B = create_sparse_matrix(2**n, rows, cols, vals, "CSC")   # Create sparse matrix with CSC layout.
6. C = ak.sparse_matrix_matrix_mult(A, B)                    # Do the sparse matrix multiplication.
```





Background

Background – Array Aggregation

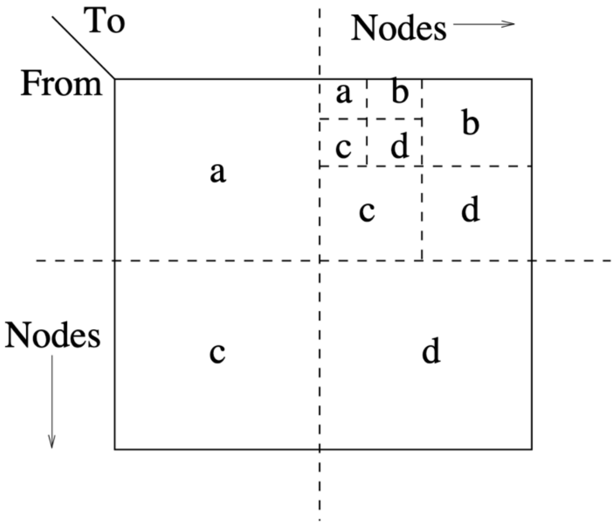
```
use BlockDist, CopyAggregation;

const size = 10000;
const space = {0..size};
const D = space dmapped new blockDist(space);
var A, rA: [D] int = D;

forall (ra, i) in zip(rA, D) with (var agg = new SrcAggregator(int)) do
    agg.copy(ra, A[size-i]);
```

- Forall loops spawn a source aggregator per-task, therefore there are per-task buffers, making aggregation memory-intensive.
- In this case the destination is ra, which is local, but the source of the data is remote, which is A[size-i].
  - This example shows source aggregation, but the rest of the talk focuses on destination aggregation.
- Whenever the buffer gets full, or the all the iterations of the task are finished, then a flush gets issues, that moves all the saved values in the buffer to the memory location they belong in.

Background – Power-Law Matrices

What? Experiment graphs were **r**ecursive **mat**rix (R-MAT) random graphs:

- $|V| \leq 2^{SCALE}$
- $|E| \leq (2^{SCALE} * eFACTOR)$

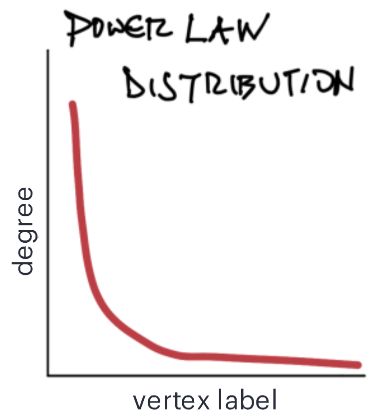


How?

1. Split adjacency matrix into four equal parts.
2. Choose part & subdivide again into four equal parts.
3. Once you reach a 1x1 cell, assign it 0 or 1 to keep that edge in the graph

Why?

- Probabilities a=0.57, b=c=0.19, and d=0.05 give a **Kronecker** graph.
- This type of graph exhibits a power-law vertex degree distribution.
- Real-world graphs exhibit power-law vertex degree distributions.
- This makes them an ideal random graph for benchmarking.



Background – Sparse Matrix Layouts

Compressed Sparse Row (CSR) format
(sorted within each row)

values:	1	2	3	4	5	6	7	8	9	1	2	3
row starts:	1	2	3	6	8	10	11	12	13			
col inds:	2	8	3	4	6	1	2	1	5	6	6	4

Space: 2·*nnz* + *n*

$$\begin{array}{c|cccccccc|}
 & 1 & & & & & & & n=8 \\
\hline
1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 \\
 & 0 & 0 & 3 & 4 & 0 & 5 & 0 & 0 \\
 & 6 & 7 & 0 & 0 & 0 & 0 & 0 & 0 \\
 & 8 & 0 & 0 & 0 & 9 & 0 & 0 & 0 \\
 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 \\
n=8 & 0 & 0 & 0 & 3 & 0 & 0 & 0 & 0 \\
\hline
\end{array}$$

Compressed Sparse Column (CSC) format
(sorted within each column)

values:	6	8	1	7	3	4	3	9	5	1	2	2
row inds:	4	5	1	4	3	3	8	5	3	6	7	2
col starts:	1	3	5	6	8	9	12	12	13			

Space: 2·*nnz* + *n*

Coordinate (COO) format (sorted in row-major order)

values:	1	2	3	4	5	6	7	8	9	1	2	3
row indices:	1	2	3	3	3	4	4	5	5	6	7	8
column indices:	2	8	3	4	6	1	2	1	5	6	6	4

Space: 3·*nnz* (*nnz* = number of nonzeroes)

Background – Sparse Matrix Multiplication

LHS matrix (CSC storage)

$$
\begin{array}{c|cccccccc|}
 & 1 & & & & & & & 8 \\
1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 \\
 & 0 & 0 & 3 & 4 & 0 & 5 & 0 & 0 \\
 & 6 & 7 & 0 & 0 & 0 & 0 & 0 & 0 \\
 & 8 & 0 & 0 & 0 & 9 & 0 & 0 & 0 \\
 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 \\
8 & 0 & 0 & 0 & 3 & 0 & 0 & 0 & 0
\end{array}
$$

$$
\begin{bmatrix} 0 \\ 0 \\ 0 \\ 6 \\ 8 \\ 0 \\ 0 \\ 0 \end{bmatrix}
\qquad
\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}
$$

$$
C_1 = \begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 6 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 8 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
$$

$$
\begin{array}{c|cccccccc|}
 & 1 & & & & & & & 8 \\
1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 \\
 & 0 & 0 & 3 & 4 & 0 & 5 & 0 & 0 \\
 & 6 & 7 & 0 & 0 & 0 & 0 & 0 & 0 \\
 & 8 & 0 & 0 & 0 & 9 & 0 & 0 & 0 \\
 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 \\
8 & 0 & 0 & 0 & 3 & 0 & 0 & 0 & 0
\end{array}
$$

RHS matrix (CSR storage)

Background – Sparse Matrix Multiplication

$$
\begin{array}{c|cccccccc|}
 & 1 & & & & & & & 8 \\
\hline
1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 \\
 & 0 & 0 & 3 & 4 & 0 & 5 & 0 & 0 \\
 & 6 & 7 & 0 & 0 & 0 & 0 & 0 & 0 \\
 & 8 & 0 & 0 & 9 & 0 & 0 & 0 & 0 \\
 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 \\
8 & 0 & 0 & 0 & 3 & 0 & 0 & 0 & 0 \\
\hline
\end{array}
$$

LHS matrix (CSC storage)

$$
\begin{array}{c|cccccccc|}
 & 1 & & & & & & & 8 \\
\hline
1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 \\
 & 0 & 0 & 3 & 4 & 0 & 5 & 0 & 0 \\
 & 6 & 7 & 0 & 0 & 0 & 0 & 0 & 0 \\
 & 8 & 0 & 0 & 9 & 0 & 0 & 0 & 0 \\
 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 \\
8 & 0 & 0 & 0 & 3 & 0 & 0 & 0 & 0 \\
\hline
\end{array}
$$

$$
\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 \end{bmatrix}
$$

$$
\begin{bmatrix} 1 \\ 0 \\ 0 \\ 7 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}
\quad
\begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 6 & 0 & 0 & 0 & 0 & 0 & 14 \\
0 & 8 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
\quad C_1 + C_2
$$

etc.

Proposed Solution

Proposed Solution – High-Level Description

- We take the current CopyAggregation module and modify it to accept two new user-defined records.
  - The source handler to dictate where data is going to be transferred to and how it will be stored within the buffer.
  - The destination handler to perform a flushing operation to move the data from the buffers to their physical memory location.
- Gives more power to the user to let them specify what data structures they want to aggregate into and how should the data be treated.

Proposed Solution – Source Handling

```
1  class SourceHandler {
2    var dVal;
3    var aVal;
4    type elemType = (int,int,int);
5
6    proc init(D, A) {
7      // workaround as ref domain is not implemented
8      this.dVal = D._value;
9
10     // workaround as ref array is not implemented
11     this.aVal = A._value;
12   }
13
14   proc sourceCopy() {
15     return new unmanaged DestinationHandler(dVal,aVal);
16   }
17
18   proc getDestinationLocale(val: elemType) {
19     var (i,j,_) = val;
20     return dVal.parentDom.dist.dsiIndexToLocale((i,j));
21   }
22 }
```

Line 4: we have the expected format for the source data, and eventually the destination data of type (i,j,v) where (i,j) is the sparse domain index and v is the data we are passing.

Lines 6-12: the initializer extracts the underlying record for domains and arrays; this is a workaround as current classes in Chapel do not let us take a ref (pointer) of a domain or array.

Lines 14-16: the aggregator has a backend handler that uses this function to instantiate a destination handler whenever it comes time to perform a flush.

Lines 18-21: gets the locale that owns the index (i,j) during a copy step that is used to add a full tuple (i,j,v) to the buffer.

Proposed Solution – Destination Handling

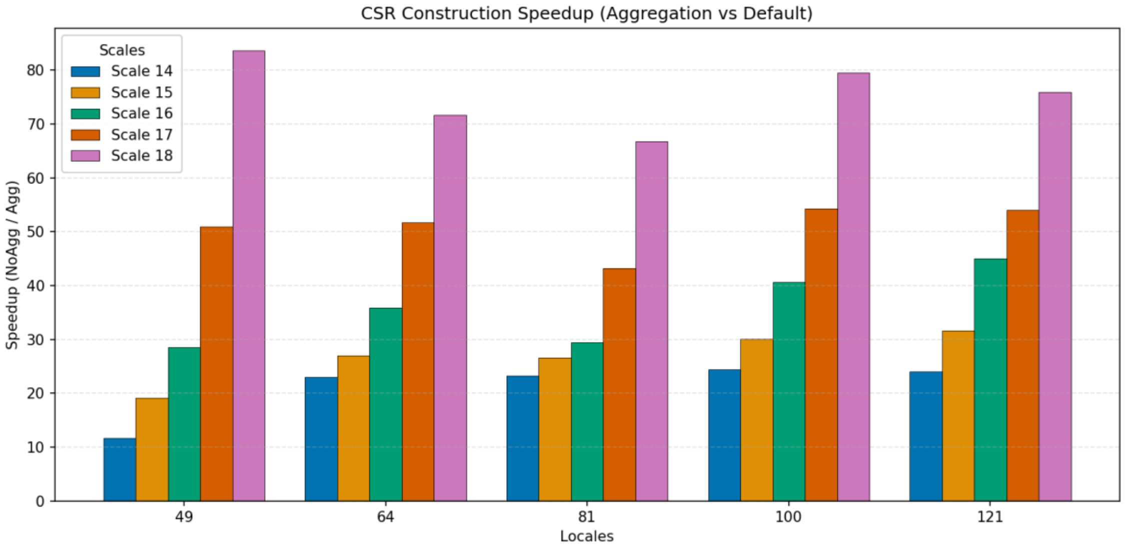
```
23 class DestinationHandler {
24     var domVal;
25     var arrVal;
26
27     proc init(domVal, arrVal) {
28         this.domVal = domVal;
29         this.arrVal = arrVal;
30     }
31
32     inline proc flush(ref rBuffer, const ref remBufferPtr, const ref myBufferIdx) {
33         const (_, locid) = this.domVal.dist.chpl__locToLocIdx(here);
34         var locIdxBuf = this.domVal.locDoms[locid]!.mySparseBlock._value.createIndexBuffer(bufSize);
35         for (dstAddr, srcVal) in rBuffer.localIter(remBufferPtr, myBufferIdx) {
36             assert(dstAddr == nil);
37             var (i,j,_) = srcVal;
38             locIdxBuf.add((i, j));
39         }
40         locIdxBuf.commit();
41         for (dstAddr, srcVal) in rBuffer.localIter(remBufferPtr, myBufferIdx) {
42             assert(dstAddr == nil);
43             var (i,j,v) = srcVal;
44             var (_,loc) = this.domVal.locDoms[locid]!.mySparseBlock._value.find((i,j));
45             this.arrVal.locArr[locid]!.myElems._value.data[loc] = v;
46         }
47     }
48 }
```

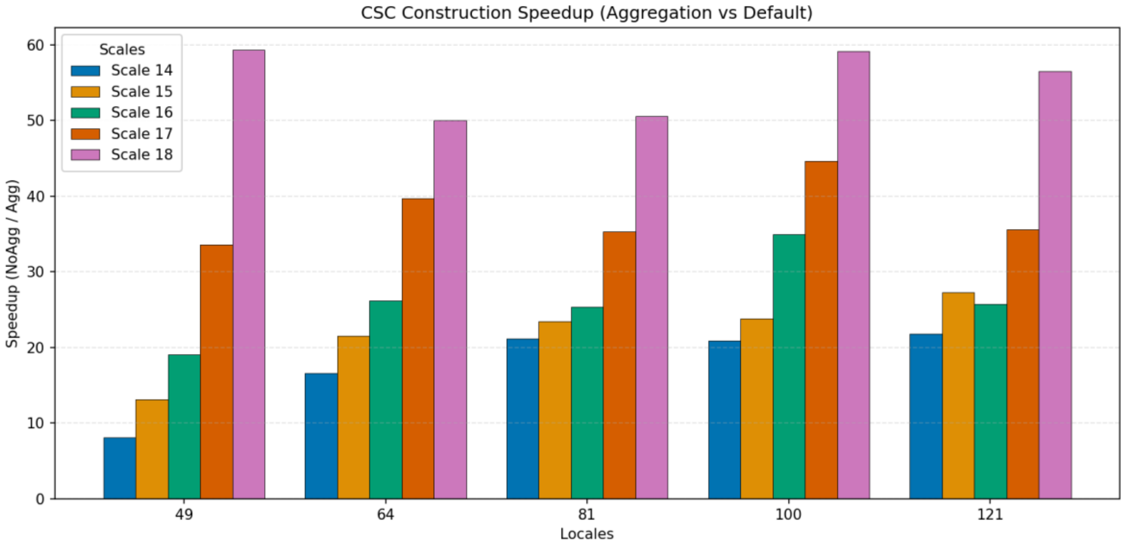
```
53 forall (i,j,v) in zip(rows, cols, vals) with (var agg = new CustomDstAggregator(new shared SourceHandler(SparseDom, SparseArr))) do
54         agg.copy((i,j,v));
```

Lines 32-45: we have the works for the flushing operation, which currently is a lot more scary-looking than intended, I just couldn't help myself and wanted to optimize as much as possible ☺.
Lines 33-34: create an index buffer for faster addition of indices into a sparse domain.
Lines 35-39: add a given index (i,j) into the buffer, once the buffer is full, or hits the commit() in line 40, the buffer gets flushed and those indices get added to the domain. **This is not the same as the remote buffer within the aggregator.**
Lines 41-45: once the indices have been added, we add in the actual data, which requires finding the index for (i,j) in the backend data array.
Lines 53-54: we see the aggregation prototype in action where we create a custom aggregator with the source handler shown in the previous slide.

Benchmarks

Aggregated vs. Non-Aggregated Sparse Matrix Creation Time

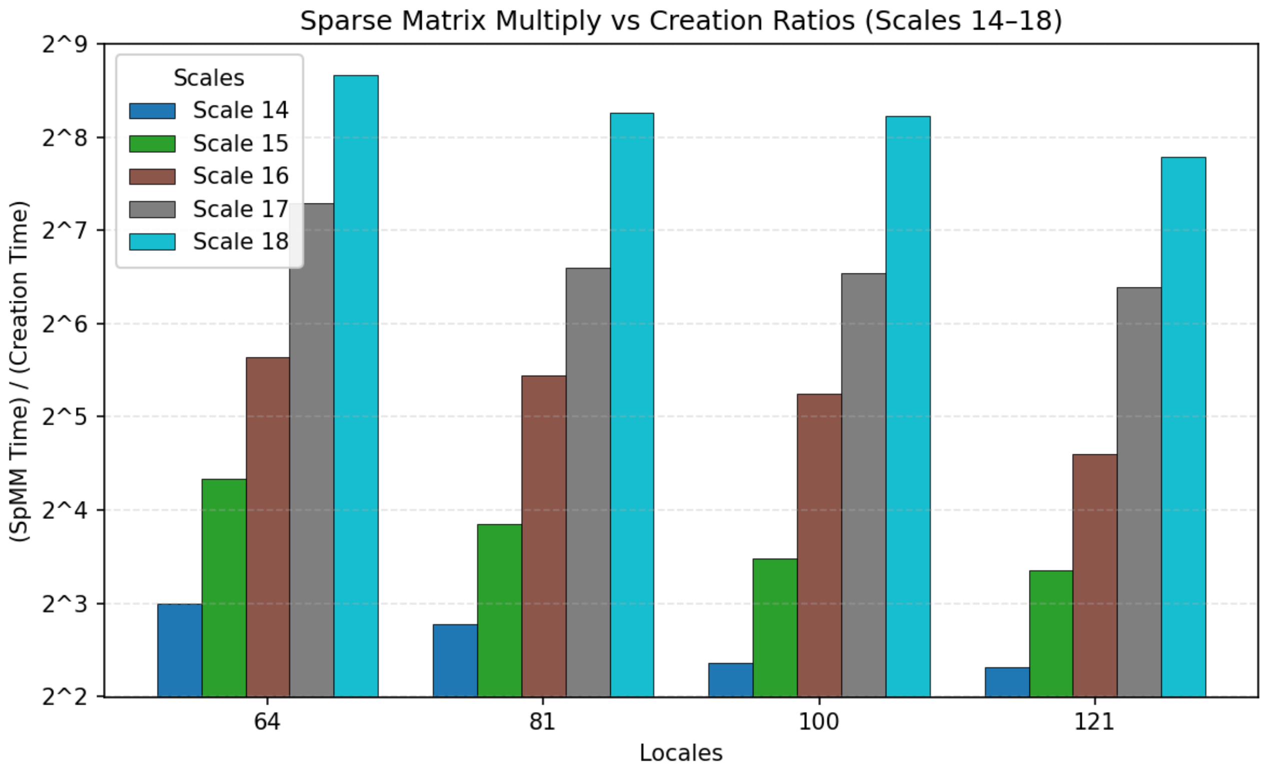




- We see consistent speed-ups across all scales for the aggregated construction times vs. the current Arkouda construction times.
- The speed-up for CSR is greater because CSC non-aggregated construction was generally slower.
  - I do not have a strong reason or sense of "why" and I do not want to speculate, but it would be a good study to figure out why.
- Generally, as the number of locales increased, the performance got better for aggregated construction whereas the non-aggregated construction slightly degraded in performance as locales increased.
- System-specific issues with compute nodes caused odd results like the ones for CSC construction at 121 locales, where the aggregated code did not perform as well as it should have.
  - Re-running the test with one trial showed much better performance, but I kept the "bad" result to showcase how system-specific issues can affect aggregation.

System: HPE Cray EX.
Slingshot-11 interconnect with communication managed through libfabric.
2 AMD EPYC 7763 processors with 256 cores total and 512GB memory per locale.

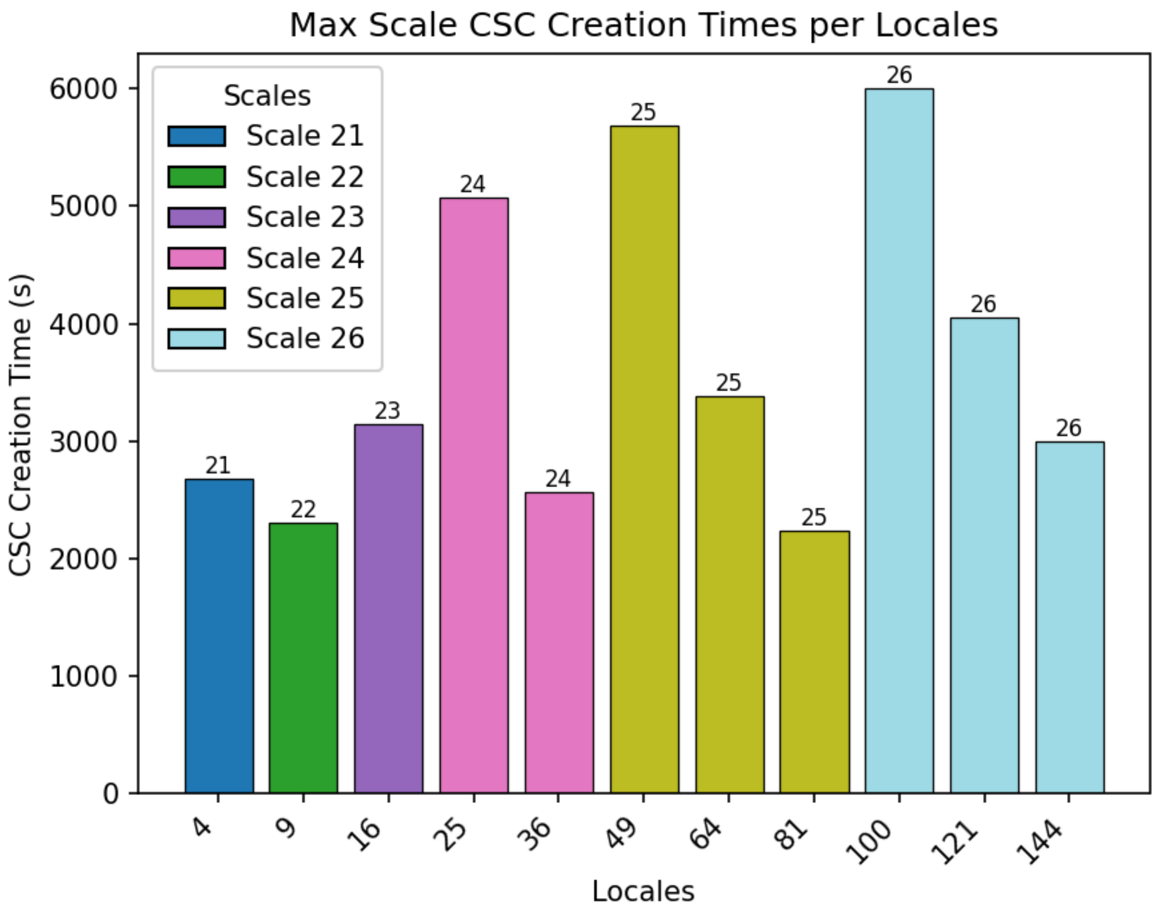
Sparse Matrix Multiplication vs. Aggregated Creation



System: HPE Cray EX.
Slingshot-11 interconnect with communication managed through libfabric.
2 AMD EPYC 7763 processors with 256 cores total and 512GB memory per locale.

- Overall, the aggregated sparse matrix creation is now always significantly faster than sparse matrix multiplication.
  - This is good for Arkouda users to quickly load in data and immediately start their analyses without having to spend too much time waiting for data to load.
- I present the y-axis on a log_2 scale. There is no specific reason other than that without the log scale, the bars for Scale 14 looked almost non-existent, even though the creation code was consistently 4x or faster.

Testing the Limits of Aggregated Sparse Matrix Creation



System: HPE Cray EX.
Slingshot-11 interconnect with communication managed through libfabric.
2 AMD EPYC 7763 processors with 256 cores total and 512GB memory per locale.

- Here, we see the maximum scale for the RMAT matrices constructed.
- As a reminder, the matrices occupy an area of 2^scale X 2^scale and have NNZ of about 2^scale * 32.
  - Give or take repeated edges getting parsed out.
  - If the scale is 20, then that will give a ratio of about 1/32768 or 0.00003 of NNZ values to the area of the matrix.
- We can see for locales 25-36 the creation time significantly improving for scale 24, and then similarly for locales 49-81 and 100-144.
  - This exhibits strong scalability for the aggregated sparse matrix creation code.
- A graph of scale 26 is considered a “toy” size by the Graph500 benchmark (they run on supercomputers like Fugaku & Frontier).
  - Assuming 64 bits per edge, a graph of this scale takes up about ~20GB in memory.

In Conclusion...

Conclusion

- This work introduces early steps toward a general aggregation framework in Chapel beyond CopyAggregation to allow more general operations such as modifying sparse matrix domains and arrays.
- This prototype gives finer control of distributed communication while keeping Chapel’s productive global namespace.
- Aggregated sparse matrix creation showcases benefits for irregular, power-law style workloads.
- Our performance results show aggregation alleviates communication bottlenecks in sparse workloads.
- We validates that explicit aggregation control yields predictable performance gains without sacrificing Chapel’s high-level model.
- Going forward this work would benefit from the following.
  - A comparison against state-of-the-art methods like conveyors.
  - A more in-depth look at the framework prototype itself. How well does it support source aggregation? What other workloads can it be applied to?
  - “Hyperparameter” tuning. Aggregation has a ton of toggles like buffer sizes. Is there a Goldilocks–space for buffer sizes to number of aggregations that is most optimal?

Thank You!

Oliver Alvarado Rodriguez

oliver.alvarado-rodriguez@hpe.com

© 2025 Hewlett Packard Enterprise Development LP