Towards A General Aggregation Framework in Chapel

Oliver Alvarado Rodriguez
Hewlett Packard Enterprise
Houston, Texas, USA
oliver.alvarado-rodriguez@hpe.com

Engin Kayraklioglu
Hewlett Packard Enterprise
Houston, Texas, USA
engin@hpe.com

Bartosz Bryg
New Jersey Institute of Technology
Newark, New Jersey, USA
bb474@njit.edu

Mohammad Dindoost
New Jersey Institute of Technology
Newark, New Jersey, USA
md724@njit.edu

David A. Bader
New Jersey Institute of Technology
Newark, New Jersey, USA
bader@njit.edu

Brad Chamberlain
Hewlett Packard Enterprise
Houston, Texas, USA
bradford.chamberlain@hpe.com

Abstract

Distributed and parallel applications with fine-grained communication experience significant performance overheads from large communication volumes between compute nodes. Communication aggregation techniques can help improve performance, particularly for irregular computational patterns that are susceptible to fine-grained communication bottlenecks. Programming languages like Chapel address these challenges through aggregation modules designed to mitigate these fine-grained communication bottlenecks. Chapel provides a lexically partitioned global namespace that enables global distributed arrays, user-defined parallel iterators, and asynchronous tasking through on-clauses. These features provide a strong foundation for implementing effective aggregation strategies. While Chapel's `CopyAggregation` module supports aggregation of remote read and write operations for arrays, it does not allow for the aggregation of arbitrary remote operations. Experienced users can develop aggregation strategies for specific use cases; but a generalize aggregation framework would enhance developer productivity. The Arkouda software package, which supports large-scale distributed array operations, pioneered aggregation concepts within Chapel. The recent addition of sparse matrix support exemplifies the need for a generalized aggregation framework to address fine-grained communication challenges. This work addresses these limitations by presenting a prototype for custom destination aggregation within Chapel. Additionally, we provide a comparative study demonstrating how this framework improves fine-grained, parallel-safe sparse matrix creation performance.

CCS Concepts

• **Computing methodologies** → **Parallel programming languages**; **Distributed programming languages**.

Keywords

parallel programming, distributed programming, optimization

SC25-W, St. Louis, MO

ACM ISBN XXX-X-XXXX-XXXX-X/XXXX/XX
https://doi.org/XXXXXXX.XXXXXXX

ACM Reference Format:
Oliver Alvarado Rodriguez, Engin Kayraklioglu, Bartosz Bryg, Mohammad Dindoost, David A. Bader, and Brad Chamberlain. 2025. Towards A General Aggregation Framework in Chapel. In *SC25-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis, November 16–21, 2025, St. Louis, MO.* ACM, New York, NY, USA, 8 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Chapel [4] is a modern high-level programming language supporting a lexically-based global namespace partitioned across memories. This model enables users to execute programs across multiple compute nodes without explicitly writing messaging functions to manage data transfers between the memories of different compute nodes. By abstracting away communication details, Chapel significantly enhances productivity in parallel programming where developers can write high-level code that the compiler transforms into distributed executables, with the runtime system automatically handling inter-node communication. However, while the global namespace permits referring to any variable regardless of location, this convenience can entice programmers into fine-grained, random access patterns that perform worse than larger data transfers capable of amortizing network overheads. Compiler optimizations can help offset some of these overheads [15], but relying on such optimizations can lead to performance uncertainty whereas using aggregation provides the performance benefits more explicitly.

An alternative approach that bypasses the need for compiler optimizations involves providing users with a way to get coarser-grained messages explicitly without complicating their code unduly. This approach leverages aggregation, which buffers data and tracks transfer destinations before executing bulk operations. For basic array operations involving assignment, the `CopyAggregation` module provides both destination aggregation (where the destination is remote) and source aggregation (where the source can be remote). Chapel users can directly utilize this module to perform array operations of the form `A[i] = B[j]` where the accessed elements could live on distinct compute nodes.

This `CopyAggregation` module demonstrates how users can gain more granular control over specific aggregated operations. A generalized framework that could aggregate arbitrary remote operations would significantly enhance developer productivity by

providing a standard, universal, and customizable approach to aggregation. Although experienced users can develop custom aggregation strategies for specific use cases, a generalized custom aggregation framework would enhance developer productivity. Chapel users who would benefit from such a standardized framework are precisely those working with highly irregular applications, such as sparse matrix operations and graph algorithms, as demonstrated by researchers developing sparse data structures and graph algorithms [1, 14].

The Arkouda software package [12] was the first to explicitly motivate the concept of aggregation for Chapel programs. Arkouda is a modular and extensible Python library that uses a client-server model for performance and scalability, with its server implemented in Chapel. Users can write new operations and register them with Arkouda's framework, as demonstrated by projects like Arachne [1]. Arkouda relies heavily on aggregation for operations such as `argsort`, where the returned array `iv` specifies how the indices of an array `A` should be reordered to produce a sorted array in increasing order. Performing an indexing operation `sortedA = A[iv]` creates a new array containing the elements of `A` arranged in sorted order. This operation utilizes the `CopyAggregation` module when implemented in Chapel.

Recently, we have begun developing prototypical sparse linear algebra capabilities for Arkouda, including sparse matrix creation and multiplication, by leveraging Chapel's multidimensional sparse domains and arrays. This represents an early step toward introducing new data distributions beyond the dense block distributions that Arkouda has traditionally supported. However, while the matrix multiplication algorithm itself functions correctly, sparse matrix creation exhibits significant performance bottlenecks due to fine-grained communication patterns. In this use case, users provide three distributed arrays `R`, `C`, and `V` containing row indices, column indices, and values stored at index $(r \in \texttt{R}, c \in \texttt{C})$ of the sparse matrix. The primary limitation is that the current sparse matrix creation implementation generates excessive fine-grained remote operations that could benefit significantly from aggregation, particularly in sparse workloads with characteristics reflecting power-law distributions.

The main contributions of this work are as follows.

(1) A prototype for custom aggregation in Chapel that extends beyond the foundational `CopyAggregation` module to support arbitrary remote operations.
(2) Enhanced parallel safety mechanisms for distributed sparse matrix operations in Chapel, specifically for compressed sparse row (CSR) and compressed sparse column (CSC) layouts.
(3) A performance evaluation demonstrating the benefits of aggregation for sparse matrix creation, comparing fine-grained communication patterns against aggregated approaches.
(4) A roadmap toward developing a standard, universal, and customizable aggregation framework for Chapel.

2 Related Work

2.1 Aggregation in Other Programming Models

The concept of message aggregation for parallel communication has been explored across various lower-level systems and programming models. Maley and DeVinney [11] present "Conveyors," a software package that provides high-bandwidth, memory-efficient streaming communication for parallel applications. Their conveyor abstraction supports many-to-many communication patterns through message aggregation and multi-hop routing, demonstrating scalability to over 100,000 processes. Conveyors are typically used in conjunction with Exstack, a predecessor communication library that provides complementary functionality for parallel message handling. Unlike higher-level language constructs, conveyors operate at the library level and can be integrated into SHMEM, UPC, or MPI applications.

Nathan Wichmann at Hewlett Packard Enterprise developed HABU, a C aggregation library that provides similar aggregation capabilities for lower-level parallel programming environments that has utilized the conveyors work above [17]. Additionally, recent work by Hati et al. presented an asynchronous distributed-memory parallel algorithm for k-mer counting that utilized custom message aggregation protocols shown the importance of aggregation in modern data applications [7].

2.2 Aggregation in Chapel

In 2018, Louis Jenkins proposed aggregation and buffering of asynchronous remote tasks via `begin on` [8]. His idea was to buffer communication operations within a parallel loop, such that updates to arrays from within a `forall` loop would be aggregated and issued as "fire-and-forget" active messages, improving efficiency for large-scale iterations. Michael Ferguson raised important questions regarding this approach, such as how to distinguish between buffered and unbuffered `begin on` operations, and how to handle cases where `begin on` requires synchronization mechanisms like locks. He noted that acquiring a lock for every `begin on` could significantly degrade performance. After extensive discussion, it was concluded that aggregating computation, rather than tasks, might be more effective. This led Jenkins to propose the Chapel Aggregation Library [9], focusing on computation aggregation.

Ferguson also documented common patterns that require aggregation [5], such as updating or gathering data from distributed data structures and performing topological sorts. He observed that destination aggregators are particularly beneficial when updating distributed data structures in parallel loops, as the destination is typically remote and the loop iterations are local. Additionally, Ferguson proposed language support for critical sections and aggregation [6]. In contrast to Jenkins' approach of aggregating tasks created with `begin on`, Ferguson suggested that `sync on` could be used to mark critical sections, ensuring they execute on the locale that owns the relevant object.

The proposal for user-facing aggregators [16] resulted in the development of the `CopyAggregation` package module (covered in the next section). This work, completed by Elliot Ronaghan, highlighted the need for improved naming, support for third-party and user-defined operations, and additional enhancements to the module. Building on this foundation, Engin Kayraklioglu added the `–auto-aggregation` flag to the Chapel compiler to automatically apply copy aggregators to certain well-formed loops, reducing the burden on programmers to manually insert aggregation calls where beneficial [10].

3 A Brief Overview of the `CopyAggregation` Module

The `CopyAggregation` module in Chapel provides efficient batch copy operations for trivially copyable types through aggregated communication. This module offers two primary aggregators: `DstAggregator` for scenarios where the source is local and the destination may be remote, and `SrcAggregator` for when the destination is local and the source may be remote. For this work, we focus primarily on destination aggregation using `DstAggregator`, though source aggregation follows similar principles.

`DstAggregator` optimizes remote assignments by batching multiple copy operations into fewer communication events. When a `copy(ref dst, src)` operation is performed through the aggregator, the operation is not immediately executed. Instead, the source value and destination address are buffered locally on a per-destination locale basis. The aggregator maintains separate buffers for each remote locale, allowing it to coalesce multiple assignments destined for the same locale into a single communication operation. This batching significantly reduces the communication overhead that would otherwise occur with individual remote assignments. An example can be seen in Listing 1.

```
use BlockDist, CopyAggregation;

const space = {0..9999};
const D = space dmapped new blockDist(space);
var A: [D] int;

// Aggregate assignments to remote array elements
forall i in D with (var agg = new DstAggregator(int)) do
    agg.copy(A[i], i * 2);
```

Listing 1: Using `DstAggregator` for Remote Array Updates

The key behavioral characteristic of destination aggregation is that updates are not immediately visible at the destination. Operations remain buffered until the buffer becomes full or either the aggregator is deinitialized or an explicit `flush()` call is made. This deferred execution model is essential for performance but requires careful consideration in program design, as intermediate reads of destination variables may not reflect recent aggregated writes until flushing occurs.

4 Motivating Use-Case: Sparse Matrix Creation from Three Block-Distributed Arrays

The core challenge of this use-case in Arkouda and Chapel involves efficiently converting three distributed arrays into a sparse matrix while minimizing communication overhead. This use case is communication-bound rather than compute-bound. The primary computational work resembles a distributed sorting or binning algorithm: determining which locale should own specific matrix indices and values ("On which node should these indices and values that I own end up?") and organizing incoming data from other locales ("What indices and values will others be sending me that I should own?"). The target distribution block-distributes the overall logical matrix indices across locales, with each locale storing its local indices, referred to as local subdomains, and values using either CSR or CSC format. The main operations involve adding index pairs to local subdomains and storing array elements corresponding to those index pairs.

The arrays may contain multiple elements with the same indices, requiring a strategy for handling these cases. Common approaches include adding the values, retaining the minimum value, maximum value, or applying a custom aggregate function to elements with duplicated indices. The current prototype removes duplicates without preserving specific values, but this limitation can be addressed by allowing users to specify their preferred reduction operation. This aggregation introduces minimal overhead since the data can be processed efficiently during the distribution phase, and Arkouda's existing function for identifying unique values can be extended to perform the specified aggregation.

In Chapel, distributed two-dimensional arrays (essentially sparse matrices) are created by defining a two-dimensional space, creating a block-distributed domain over this space, and then constructing a sparse domain and array using the dense domain as the parent. Chapel also provides user-configurable options for data representation, including compressed sparse row (CSR) and compressed sparse column (CSC) formats. The creation of these distributed sparse domains and arrays is shown in Listing 2.

```
const Space = {1..N, 1..N};
const DenseDom = Space dmapped new blockDist(
    Space,
    sparseLayoutType = csrLayout(parSafe=true)
);
var SparseDom: sparse subdomain(DenseDom);
var SparseArr: [SparseDom] int;
```

Listing 2: Chapel Sparse Domain and Array Creation

The problem reduces to bulk-writing indices with their corresponding values to the locales that own the respective index positions. Once all indices have been transferred to the sparse domain and values have been assigned, the sparse matrix construction is complete. Users can then perform sparse matrix multiplications through the sparse matrix interface found in Arkouda [2].

5 Implementation

In this work, the existing `CopyAggregation` module in Chapel was extended to support user-defined classes for both source and destination handling. In this architecture, the source class manages data buffering for writes, while the destination class handles buffer flushing operations. The implementation details of these specialized classes are discussed in Subsections 5.1 and 5.2.

We implemented a record that accepts the destination handler responsible for buffering operations. When an instance of this record is initialized on a remote locale, it creates a local copy of the destination handler, ensuring proper buffer flushing at that specific locale.

Additionally, we implemented a customized `copy` method that accepts only the data to be written, eliminating the need for a specific location address. In this implementation, the destination address is irrelevant since only the target locale matters as the destination handler class manages the underlying data structure operations. This modified function, omitted here for brevity, mirrors the original implementation except that it maintains the destination address as `nil` and determines the target locale using `handler.getDestinationLocale(srcVal).id`. An example of the `getDestinationLocale` function is provided in Subsection 5.1.

The internal `_flushBuffer` function was also modified to eliminate individual element processing. Instead of handling buffer elements independently, the entire buffer is passed directly to the user-defined destination handler's `flush` method. This design grants users complete control over bulk buffer operations, enabling more efficient batch processing when aggregating elements into their target data structures. For this specific use-case, the data structure to aggregate into is a two-dimensional, sparse block-distributed array in Chapel where both the domain and array must be modified.

```
class SourceHandler {
  var dVal;
  var aVal;
  type elemType = (int,int,int);

  proc init(D, A) {
    // workaround as ref domain is not implemented
    this.dVal = D._value;

    // workaround as ref array is not implemented
    this.aVal = A._value;
  }

  proc sourceCopy() {
    return new unmanaged DestinationHandler(dVal,aVal);
  }

  proc getDestinationLocale(val: elemType) {
    var (i,j,_) = val;
    return dVal.parentDom.dist.dsiIndexToLocale((i,j));
  }
}
```

Listing 3: User-Defined Source Handler

5.1 Source Handling

The interface for a user-defined source handler is shown in Listing 3. The source handler serves two primary functions in the current prototype implementation. First, it creates and transmits a copy of the destination handler to the locale responsible for processing a specific buffer. This copying mechanism parallels the process used when transferring buffered items from source to destination locales. Second, it determines the appropriate destination locale for incoming data. This function is critical for distributed data structures, as it requires probing operations to identify the correct data placement. For sparse matrix operations, this calculation is straightforward through Chapel's domain interface, which allows users to query a domain for specific indices using the standard `dsiIndexToLocale(idx)` function. In this context, the function accepts a tuple of two values representing the row and column indices of the sparse matrix.

It should be noted that aggregator initialization requires only the source handler in the current prototype implementation. During the buffer flushing operation, the source handler manages the transfer of buffered data to the remote locale and automatically handles the creation and transmission of the destination handler copy. The destination handler performs the actual flushing operations at the target locale. However, the source handler enables this process by facilitating the copying mechanism and determining the appropriate destination locale for data placement.

5.2 Destination Handling

The interface for a user-defined destination handler is shown in Listing 4. The destination handler serves one primary role: specifying the flushing operation that defines how data is added to the target data structure. In the sparse matrix user-case example, two main variables are utilized: `domVal` and `arrVal`, which represent the internal record representations for domains and arrays in Chapel. While these variables are typically hidden from users, passing the underlying record effectively provides a reference to the domain and array components that constitute the sparse matrix. This approach simplifies the implementation by directly accessing the matrix's internal structure.

The flush function implementation is straightforward and consists of two main phases. First, an index buffer is created using Chapel's internal data structure for sparse domains, which enables buffering of index values before domain insertion. This approach allows aggregate addition of values to sparse domains without requiring resizing checks for each individual insertion. This represents an additional buffering layer beyond the remote communication buffering, necessary because the buffer contains tuples of three elements that cannot be directly written to the sparse domain. Once all indices have been added to the sparse domain, the corresponding values can be inserted into the sparse array. Since the example uses compressed sparse row (CSR) format, the correct position in the underlying one-dimensional array must be determined. The implementation utilizes Chapel's `find` method from the `Search` module, which performs a binary search for the column index within the appropriate row range to locate the one-dimensional index for storing the tuple's value.

6 Experimentation

Experiments were conducted on two distinct systems with varying interconnects between compute nodes, middleware layers, and processor counts.

The first system was a Hewlett Packard Enterprise (HPE) Cray Supercomputing EX equipped with a Slingshot-11 interconnect and communication managed through libfabric. Each compute node possessed 2 AMD EPYC 7763 processors with 256 cores total and 512GB memory.

The second system was a high-performance cluster equipped with an Infiniband HDR 100GB interconnect and communication managed through GASNet. Each compute node possessed 2 AMD EPYC 7753 processors with 128 cores total and 512GB memory.

6.1 Generated Data

The Recursive Matrix (RMAT) model provides a scalable approach for generating synthetic graphs that exhibit properties commonly found in real-world networks [3]. RMAT graphs are constructed through a recursive quadrant-based partitioning process that models the way connections form in many natural and social networks. The model uses four probability parameters `a`, `b`, `c`, and `d` that control the likelihood of edges appearing in each quadrant of a recursively subdivided adjacency matrix. These parameters allow RMAT to generate graphs with realistic characteristics such as power-law degree distributions and small-world properties.

The RMAT generation process begins with an `n × n` adjacency matrix and recursively divides it into four quadrants for each bit position in the vertex indices. At each recursive level, the algorithm probabilistically assigns edges to quadrants based on the four parameters: `a` represents the probability of placement in the upper-left

```
class DestinationHandler {
    var domVal;
    var arrVal;

    proc init(domVal, arrVal) {
        this.domVal = domVal;
        this.arrVal = arrVal;
    }

    inline proc flush(ref rBuffer, const ref remBufferPtr, const ref myBufferIdx) {
        const (_, locid) = this.domVal.dist.chpl__locToLocIdx(here);
        var locIdxBuf = this.domVal.locDoms[locid]!.mySparseBlock._value.createIndexBuffer(bufSize,true,true);
        for (dstAddr, srcVal) in rBuffer.localIter(remBufferPtr, myBufferIdx) {
            assert(dstAddr == nil);
            var (i,j,_) = srcVal;
            locIdxBuf.add((i, j));
        }
        locIdxBuf.commit();
        for (dstAddr, srcVal) in rBuffer.localIter(remBufferPtr, myBufferIdx) {
            assert(dstAddr == nil);
            var (i,j,v) = srcVal;
            var (_,loc) = this.domVal.locDoms[locid]!.mySparseBlock._value.find((i,j));
            this.arrVal.locArr[locid]!.myElems._value.data[loc] = v;
        }
    }
}
```

Listing 4: User-Defined Destination Handler

```
forall (i,j) in zip(rows, cols) with (var idxBuf = SparseDom.createIndexBuffer(bufSize)) do
    idxBuf.add((i,j));
forall (i,j,v) in zip(rows, cols, vals) with (ref SparseArr) do
    SparseArr[i,j] = v;
```

Listing 5: Non-Aggregated Sparse Matrix Creation

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

Listing 6: Aggregated Sparse Matrix Creation

quadrant, b for upper-right, c for lower-left, and d for lower-right, where `a + b + c + d = 1`. For each edge, the algorithm generates random numbers to determine which quadrant the edge belongs to at each level of recursion. The final edge coordinates are computed by accumulating the quadrant selections across all SCALE levels, where `SCALE = log_2(n)`.

The RMAT implementation applies a final permutation step that randomly reorders the vertex indices. This permutation prevents systematic bias in vertex numbering and ensures that the generated graph properties are not artificially influenced by the recursive construction order. The permutation is applied to both source and destination vertices of each edge, effectively shuffling the final adjacency matrix while preserving the underlying structural properties determined by the RMAT parameters. The Chapel code utilized for RMAT graph generation for this work can be found in Listing 7.

6.2 Main Experimentation Kernels

Two methods for creating a sparse matrix from three arrays were implemented: a non-aggregated approach using only index buffering seen in Listing 5 and an aggregated approach combining both aggregation and index buffering seen in Listing 6. The aggregated implementation instantiates a custom destination aggregation object with the user-defined source handler. As discussed in Subsection 5.1, the source handler manages the transmission of destination handler copies to remote locales during the aggregation process. The main focus of the following study is to compare the speedup of the non-aggregated kernel versus the aggregated kernel which is a ratio calculated by $\frac{T_{NoAggregation}}{T_{Aggregation}}$.

6.3 Discussion

Figure 1 shows the speedup results on the HPE Cray Supercomputing EX system. As shown in the figure, speedup improves with both the number of locales and the size of the data. While consistent scalability might be expected for such a study, several factors influence the comparison between non-aggregation and aggregation kernels, including the specific system architecture and the nature of the matrices selected for experimentation.

First, we discuss the effects of using sparse matrices generated by the RMAT model with the default values given in Subsection 6.1. These default values produce power-law distributed matrices, where certain rows are significantly more populated than others, commonly used for scalability benchmarks in graph algorithms [13]. This random power-law distribution, combined with a sorting preprocessing step that exploits the `SparseIndexBuffer` flags for `dataSorted` and `isUnique` as shown in Listing 4, leads to irregular speedup results. As shown in Figure 1, the speedup factor on 64 and 128 compute nodes is less significant for Scale 16 graphs but improves as graph size increases. This occurs due to the ratio of parallel cores to data size when abundant hardware is available, possibly combined with the sorted power-law indices concentrating rows with more values entirely on one locale or only a small subset of locales. This data locality enables more local work, where indices can be added directly to the local subdomain without transmission to remote locales. As matrix size increases, the ratio of parallel resources to workload decreases, causing increased latency and communication costs as locales must wait for more buffers to be

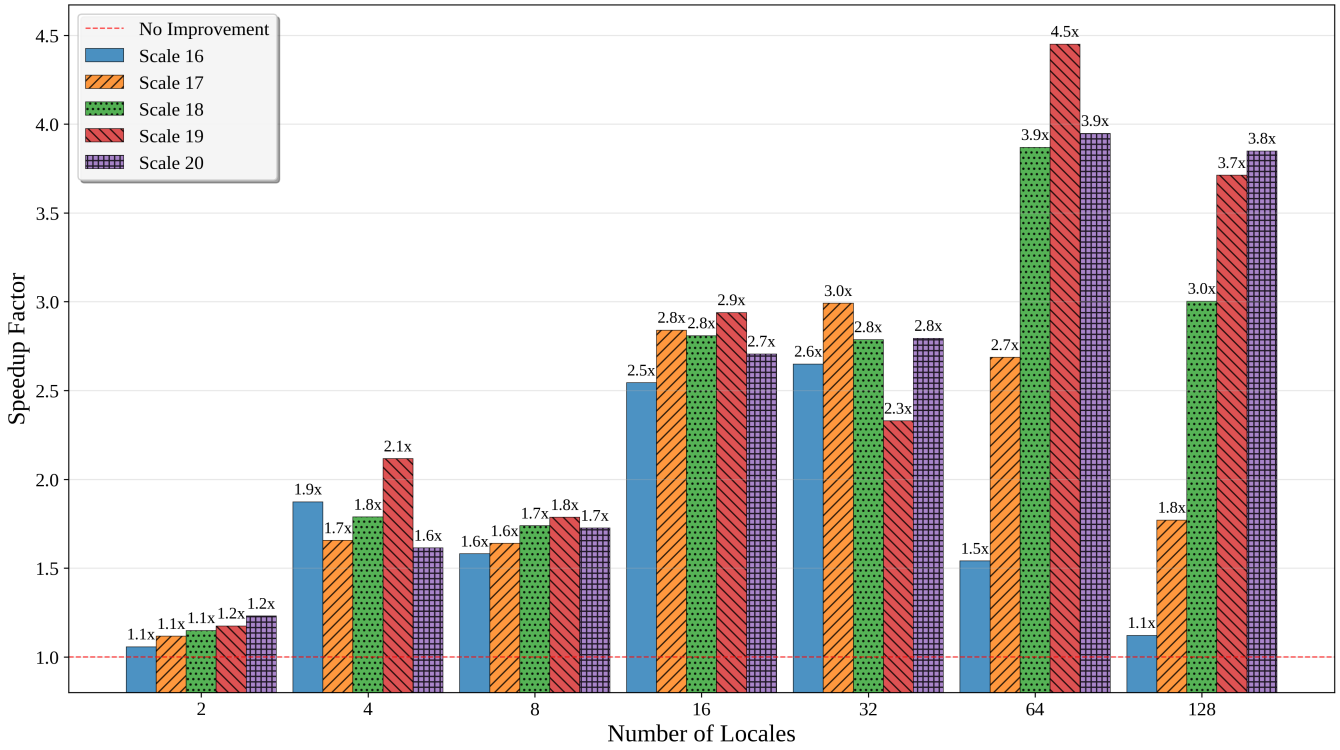

Figure 1: Speedup on the HPE Cray Supercomputing EX system.

flushed. Figure 2 shows a similar trend for the Infiniband HDR-100 system as the HPE Cray Supercomputing EX system.

7 Future Work

7.1 Design Challenges

This work currently focuses exclusively on custom destination aggregation for sparse matrices. Future work will address both technical and design challenges associated with more complex types of aggregation, such as source aggregation.

7.1.1 Source Aggregation Memory Troubles. Source aggregation presents unique challenges not encountered in destination aggregation, particularly the risk of memory exhaustion due to unbounded buffer growth. In source aggregation scenarios, remote buffers destined for local processing can only be validated during the actual write operation, creating a temporal gap between buffer allocation and capacity verification. This delay potentially leads to out-of-memory errors when buffers become oversized before flush tasks are initiated, as the system cannot predict incoming data volumes from distributed sources. This issue is less prevalent in destination buffering because buffers are flushed immediately upon reaching predetermined capacity thresholds, following a "set-and-forget" approach that maintains bounded memory usage.

7.1.2 Interface Generalization and Extensibility. The current custom aggregation interface requires further refinement and broader applicability testing beyond its current sparse matrix implementation. The prototype has been validated exclusively with sparse matrices using pre-distributed Chapel domains and arrays, raising critical questions about its extensibility to other distributed data structures. Specifically, the interface must address how users can aggregate data to remote Chapel lists, dynamic data structures, or other non-array-based collections that lack the structured indexing properties of distributed domains.

Currently, users seeking fine-grained control over remote variable initialization face significant implementation complexity. They must either utilize the replicated variable module across all locales or create block-distributed arrays sized according to the system's locale count, with each array element initialized to contain its own subsidiary data structure. This approach requires users to perform explicit indexing operations into block-distributed arrays to extract their target data structures before performing append or modification operations. Additional design questions emerge regarding whether the interface can support passing dynamic data structures such as lists as direct references to both source and destination handlers, or whether the current paradigm of passing underlying records from block-distributed arrays represents the optimal abstraction level.

7.2 Planned Experimental Studies

7.2.1 Community Engagement and Design Proposal. We are planning to develop a comprehensive design proposal for discussion within the Chapel community to enhance aggregation productivity for users requiring fine-grained control over distributed aggregation operations. Such control mechanisms are particularly critical

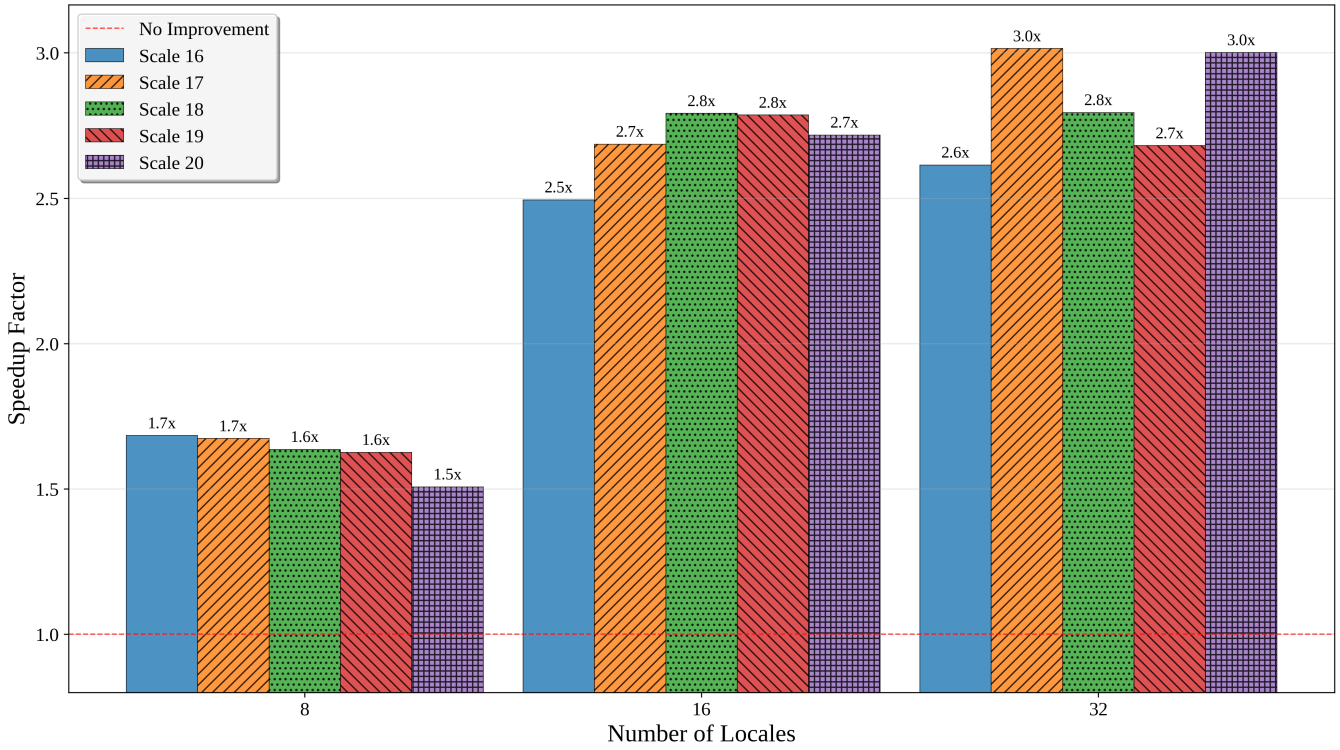

Figure 2: Speedup on the Infiniband HDR-100 system

```
proc recursiveMatrixGenerator(a, b, c, d, SCALE, n, nnz) {
    const nRange = 0..<n;
    const nnzRange = 0..<nnz;

    var randGen = new randomStream(real);
    var unifRandom = blockDist.createArray({nnzRange}, real);
    var idx = blockDist.createArray({nnzRange}, (int,int));
    idx = (1,1);

    // Calculate constants for bit-setting operations
    const cNorm = c / (1 - (a + b));
    const aNorm = a / (a + b);
    const ab = a + b;

    // Generate the indices of each quadrant for the idx
    for s in 1..SCALE {
        randGen.fill(unifRandom);
        var iiBit = unifRandom > ab;

        randGen.fill(unifRandom);
        var jjBit = unifRandom >
                    (cNorm * iiBit + aNorm * !iiBit);

        idx += assignQuadrant(iiBit, jjBit, 2**(s-1));
    }

    // Apply random permutation to vertex indices
    var permutation = permute(nRangeIndices);
    forall i in idx.domain with (var agg = new SrcAggregator(int)) {
        agg.copy(row[i], permutation[row[i]]);
        agg.copy(col[i], permutation[col[i]]);
    }
    return (row, col);
}
```

Listing 7: Sparse Matrix Index Generation using the Recursive Matrix Model (RMAT) for Graph Mining

for irregular computational patterns, including sparse matrix computations, graph algorithms, and other applications characterized by unpredictable data access patterns and non-uniform workload distributions.

7.2.2 Extended Sparse Matrix Evaluation. An extended empirical study examining diverse sparse matrix types is planned for the near future. The current work focused primarily on RMAT-generated graphs due to their power-law distributions, which align with the requirements of users currently seeking efficient sparse matrix multiplication solutions in Arkouda. Future studies will encompass matrices derived from real-world applications, including scientific simulations, social network analysis, and machine learning workloads, to evaluate the generalizability of the proposed aggregation techniques across different sparsity patterns and structural characteristics. Further, many parameters exist in the prototype implementation that can have significant effects on performance such as the size of the 'SparseIndexBuffer' set to 128 elements for this work to the actual size of the buffers utilized during communication, set by default to 1,024 elements for HPE Cray Supepercomputing EX systems and 8,192 elements for Infiniband systems.

7.2.3 Large-Scale System Performance Analysis. A comprehensive system-level performance study is planned to benchmark not only the sparse matrix creation kernels presented in this paper but also the complete sparse matrix multiplication pipeline available in Arkouda. This investigation will extend beyond the current Scale 20 evaluation, which generates sparse matrices containing approximately fifty million non-zero values, to examine performance characteristics at Scale 22 and higher, corresponding to matrices with hundreds of millions to billions of non-zero elements. The study will evaluate scalability limits, memory utilization patterns, and

communication overhead scaling across varying system configurations and problem sizes, providing insights into the practical deployment boundaries of the proposed aggregation framework.

8 Conclusion

This work presents the initial steps towards a general aggregation framework in Chapel that extends beyond the `CopyAggregation` module to support arbitrary remote operations. Our prototype demonstrates how developers can achieve more granular control over distributed communication patterns while maintaining the productivity benefits of Chapel's global namespace abstraction.

The sparse matrix creation use case serves as a compelling demonstration of the framework's potential impact on irregular applications. By aggregating fine-grained remote operations that traditionally generate excessive communication overhead, our approach achieves performance improvements over a high productive fine-grained version. Our performance evaluation reveals that aggregation can effectively mitigate communication bottlenecks inherent in sparse workloads, especially those exhibiting power-law distributions characteristic of real-world graph and matrix data. These results validate the hypothesis that explicit aggregation control can provide more predictable performance benefits than relying exclusively on compiler optimizations, while still preserving the high-level programming model that makes Chapel attractive.

The broader implications of this work extend beyond sparse matrix operations to any Chapel application characterized by irregular data access patterns. Graph algorithms, sparse data structure manipulations, and other computations that generate unpredictable communication patterns stand to benefit from the customizable aggregation mechanisms we have developed. By providing users with explicit control over aggregation behavior this prototype represents a step toward resolving the fundamental tension between programming convenience and performance predictability in distributed, communication-intensive Chapel programs.

Moving forward, the roadmap toward a general aggregation framework for Chapel requires continued collaboration with the Chapel community to refine the interface design and expand its applicability. The lessons learned from this sparse matrix implementation provide valuable insights for designing aggregation abstractions that can accommodate diverse application requirements while maintaining the simplicity and expressiveness that define Chapel's programming model.

Acknowledgment

Thanks to the NSF for funding this research in part by grants CCF-2109988, OAC-2402560, and CCF-2453324.

References

[1] Oliver Alvarado Rodriguez, Zhihui Du, Joseph Patchett, Fuhuan Li, and David A. Bader. 2022. Arachne: An Arkouda Package for Large-Scale Graph Analytics. In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7. doi:10.1109/HPEC55821.2022.9991947

[2] Bears-R-Us. 2024. Arkouda Release Notes v2024.10.02. https://github.com/Bears-R-Us/arkouda/releases/tag/v2024.10.02 Interactive Data Analytics at Supercomputing Scale.

[3] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A Recursive Model for Graph Mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining (SDM)*. 442–446. arXiv:https://epubs.siam.org/doi/pdf/10.1137/1.9781611972740.43 doi:10.1137/1.9781611972740.43

[4] Bradford L Chamberlain, David Callahan, and Hans P Zima. 2007. Parallel Programmability and the Chapel Language. *The International Journal of High Performance Computing Applications* 21, 3 (2007), 291–312.

[5] Michael Ferguson. 2018. Common patterns that require aggregation. GitHub Issue. https://github.com/chapel-lang/chapel/issues/9848 Chapel Language GitHub Repository.

[6] Michael Ferguson. 2019. Language support for critical sections and aggregation. GitHub Issue. https://github.com/chapel-lang/chapel/issues/12306 Chapel Language GitHub Repository.

[7] Souvadra Hati, Akihiro Hayashi, and Richard W. Vuduc. 2025. An Asynchronous Distributed-Memory Parallel Algorithm for k-Mer Counting. In *IEEE International Parallel and Distributed Processing Symposium, IPDPS 2025, Milano, Italy, June 3-7, 2025*. IEEE, 472–483. doi:10.1109/IPDPS64566.2025.00049

[8] Louis Jenkins. 2018. Aggregation and buffering of asynchronous remote tasks via `begin on`. GitHub Issue. https://github.com/chapel-lang/chapel/issues/9727 Chapel Language GitHub Repository.

[9] Louis Jenkins. 2018. Chapel Aggregation Library. GitHub Issue. https://github.com/chapel-lang/chapel/issues/10386 Chapel Language GitHub Repository.

[10] Engin Kayraklioglu, Elliot Ronaghan, Michael P. Ferguson, and Bradford L. Chamberlain. 2022. Locality-Based Optimizations in the Chapel Compiler. In *Languages and Compilers for Parallel Computing*, Xiaoming Li and Sunita Chandrasekaran (Eds.). Springer International Publishing, Cham, 3–17.

[11] F. Miller Maley and Jason G. DeVinney. 2019. Conveyors for Streaming Many-To-Many Communication. In *2019 IEEE/ACM 9th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. 1–8. doi:10.1109/IA349570.2019.00007

[12] Michael Merrill, William Reus, and Timothy Neumann. 2019. Arkouda: Interactive Data Exploration Backed By Chapel. In *Proceedings of the ACM SIGPLAN 6th Chapel Implementers and Users Workshop*. 28–28.

[13] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. 2010. Introducing the graph 500. *Cray Users Group (CUG)* 19, 45-74 (2010), 22.

[14] Thomas B. Rolinger, Joseph Craft, Christopher D. Krieger, and Alan Sussman. 2021. Towards High Productivity and Performance for Irregular Applications in Chapel. In *2021 SC Workshops Supplementary Proceedings (SCWS)*. 1–11. doi:10.1109/SCWS55283.2021.00012

[15] Thomas B. Rolinger and Alan Sussman. 2024. Adaptive Prefetching for Fine-grain Communication in PGAS Programs. In *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 740–751. doi:10.1109/IPDPS57955.2024.00071

[16] Elliot Ronaghan. 2020. Proposal for user-facing aggregators. GitHub Issue. https://github.com/chapel-lang/chapel/issues/16963 Chapel Language GitHub Repository.

[17] Shubhendra Pal Singhal, Akihiro Hayashi, and Vivek Sarkar. 2024. Bottleneck Scenarios in Use of the Conveyors Message Aggregation Library. In *2024 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 322–324. doi:10.1109/ISPASS61541.2024.00045