Integrating Chapel programs and MPI-Based Libraries for High-performance Graph Analysis

Trevor McCrary*
tmccrary9@gatech.edu
Georgia Institute for Technology
Atlanta, GA, USA

Karen Devine*
devine.hpc@gmail.com
Sandia National Laboratories, ret.
Albuquerque, NM, USA

Andrew Younge
ajyoung@sandia.gov
Sandia National Laboratories
Albuquerque, NM, USA

ABSTRACT

We identify techniques to interface Chapel programs with parallel, distributed, MPI-based libraries written in C++ *without storing multiple copies of shared data*. This integration enables Chapel users to take advantage of the vast array of capabilities developed in parallel numerical libraries without the memory cost of duplicated data. We demonstrate two approaches to interface Chapel code with the MPI-based graph and numerical solver libraries Grafiki and Trilinos. The first uses a single Chapel executable to call a C function that interacts with the MPI libraries; it requires Chapel users to understand the Grafiki library interfaces and link their codes with the MPI-based libraries. The second uses the Unix `mmap` function to allow separate Chapel and Grafiki executables to read and write to the same block of memory on a node; it maintains greater independence between the Chapel and MPI-based codes, simplifying the Chapel user's experience. We also encapsulated the second approach in Docker/Singularity containers to maximize ease of use by Chapel users. Comparisons of the two approaches using shared and distributed memory installations of Chapel show that both approaches are feasible for sharing data between Chapel and MPI-based libraries, yielding similar scalability and performance with no penalty from using `mmap`.

KEYWORDS

Chapel, MPI, interoperability, mmap, container

ACM Reference Format:
Trevor McCrary, Karen Devine, and Andrew Younge. 2022. Integrating Chapel programs and MPI-Based Libraries for High-performance Graph Analysis. In *Proceedings of The 2022 Chapel Implementers and Users Workshop (CHIUW22).* ACM, New York, NY, USA, 10 pages.

1 INTRODUCTION

We present two methods that allow data to be *shared, rather than copied,* between partitioned global address space programs in Chapel and distributed memory, parallel algorithms that are written in C++ and use the Message Passing Interface (MPI) [12] library to exchange data. We demonstrate our methods using applications in graph analysis: a simple graph connected-component algorithm in Chapel and a graph hitting-times algorithm in the graph toolkit Grafiki [15]. However, the general capability to integrate Chapel and MPI-based libraries is valuable in many applications, as it combines the simplicity of Chapel programming with the speed and efficiency of existing high-performance MPI-based algorithms.

Chapel [3, 7] is a partitioned global address space (PGAS) language that is built for productive parallel programming at scale. Chapel simplifies parallel programming by providing thread-based parallel loops and managing the layout of arrays within a parallel computer's memory. Although Chapel arrays can be distributed across processors, Chapel users can access any array entry on any processor, without knowing on which processor the data is stored. Chapel manages the data movement or communication required to retrieve array entries. Thus, Chapel provides a very easy-to-use programming environment for parallel algorithm development; algorithms such as parallel graph connected-component labeling can be implemented in just a few lines of code.

In MPI-based libraries, data is also distributed across the memory spaces of parallel processors. The distribution, however, is determined by the programmer. Moreover, each processor can access only the data in its memory. Off-processor data must be communicated explicitly via message passing. This explicit control of data distribution and movement allows highly efficient parallel execution, but requires much more effort from the programmer. For example, Grafiki (the successor of TriData) [15] is an MPI-based library of high-performance parallel graph analysis algorithms with linear solvers from the Trilinos [6, 14] toolkit. Grafiki's graph computations are done via Trilinos' matrix-vector operations, with explicit inter-processor communication via MPI send/receive operations.

Integrating Chapel and Grafiki allows graph analysts to easily filter and manipulate graph data via Chapel, and then apply Grafiki's high-performance parallel algorithms to analyze the resulting data. The main challenges in the integration are avoiding duplication of data between Chapel and Grafiki, and insulating graph analysts from the complications of using external MPI-based libraries.

Sharing, rather than copying, data between Chapel and MPI-based libraries is crucial to success of the integration. Doubling the amount of memory required to couple two algorithms is often infeasible. In our demonstration, for example, Chapel users wish to analyze the largest connected component of graphs that fill much of their computer's memory; the available memory is insufficient to copy this component in memory for analysis in Grafiki. Our approaches rely on library interfaces with sufficient data abstraction to accommodate Chapel data structures (e.g., edge lists in Chapel arrays) without requiring data to be copied and/or reorganized for Grafiki (e.g., into compressed-sparse row graphs).

Chapel users should also be insulated as much as possible from the complexities of building external MPI-based libraries. One of our approaches addresses this issue by demonstrating how Chapel and Grafiki can share data through memory-mapped regions. Grafiki analysis can then be run as a separate executable or within a Docker container provided to the Chapel users.

*Work completed at Sandia National Laboratories, Summer 2021. [10]

The contributions of this work include two approaches for integrating Chapel programs with MPI-based C++ libraries. The first approach uses Chapel's interface to C-language functions to share data between Chapel and the MPI-based libraries. For the second approach, we developed a new Chapel data distribution Domain that uses memory-mapped regions to share data with external processes; to our knowledge, this use of memory-mapped regions in Chapel Domains has not been done before. We describe how to extend the second approach for use in Docker containers, for the greatest insulation of Chapel users from the details of the MPI-based libraries. We demonstrate our approaches using a simple graph connected-components algorithm in Chapel and the high-performance graph hitting-times algorithm in Grafiki. We run on two different Chapel environments: a single-node shared memory environment and a multi-node distributed memory environment. Our performance experiments demonstrate scalability of our methods, showing that our methods can successfully integrate Chapel and high-performance MPI-based parallel algorithms.

2 BACKGROUND

We describe the MPI, Chapel, Grafiki, Trilinos and Unix features that are important to our integration approaches.

MPI: The Message Passing Interface (MPI) enables sharing of data between distributed memory spaces. Each processor is assigned a unique "rank" in $[0, NumberOfProcessors)$. Processors can access only their local memory; they cannot access memory associated with other processors. Instead, shared data must be communicated across the interconnect network of the cluster or supercomputer. The MPI library sends buffers of data across network links from the processors owning the data to the processors needing it.

Chapel: Chapel uses the concept of *locales* to associate data with individual processors for performance and scalability [7]. Chapel's locales are analogous to MPI ranks. Chapel's global address space allows locales to access and manipulate data that are stored on other locales without explicit communication by the Chapel user; needed communication is performed by the Chapel runtime environment. Thus, underlying communication causes data access on a different locale to be more expensive than data access on the data's own locale. While Chapel can be built with an MPI back-end to do its communication, an MPI back-end in Chapel is not necessary for our proposed approach. In our work, we rely only on a one-to-one mapping between processors' Chapel locales and MPI ranks.

Chapel distributed arrays are the main data structure used in this work. We store lists of graph edges and vertices in Chapel distributed arrays. A distributed array is implemented as a collection of arrays, with one local array stored on each locale. The Chapel distributed array manages the global address space indexing that allows access of any array entry from any locale. Chapel's Domain maps describe and manage the distribution of arrays to processes. The `Block` Domain provides a commonly used distribution; in it, the indices are partitioned evenly across the locales so that the first locale has the first contiguous chunk of indices, the second locale has the next chunk, and so on. An example `Block` array with 64 elements distributed across four locales in depicted in Figure 1.

Chapel users can create their own distributions to fit their requirements. To create a custom distribution, users create custom Domain map classes that implement the Domain map Standard Interface (DSI) [1, 2]. For our second approach below, we create a custom Domain map by modifying Chapel's `Block` distribution to use memory that can be shared among processes.

Chapel supports interoperability with C code. Users can access C libraries, variables, functions, structures, and constants using the `extern` keyword. Chapel programs can call C functions, and addresses of Chapel arrays can be passed to C functions. We use this capability in our first approach below. Chapel also has a library of definitions for C datatypes (e.g., `long long int`). We use these datatypes for consistency between our Chapel and C code.

Grafiki and Trilinos: Grafiki (formerly called TriData) [15] is a library of high-performance graph and hypergraph analysis algorithms written in C++. It contains algorithms for computing hitting times, spectral clustering, and eigenvector centrality. For our demonstration, we use Grafiki's hitting time algorithm, which operates on a square, symmetric matrix that may be distributed across processors. The symmetric matrix represents the adjacency matrix of a connected graph (i.e., the graph has a single connected component). Each edge (i, j) in the Chapel data corresponds to a nonzero a_{ij} in the adjacency matrix A; each vertex corresponds to a row and column of the matrix.

Grafiki's algorithms rely on linear and eigen-solvers; for these solvers as well as for abstractions of matrix and vector operations, Grafiki uses the open-source Trilinos [6, 14] framework. Trilinos has been developed and optimized for distributed memory, shared memory, and GPU parallel performance, especially in the realm of physics-based scientific simulations. Trilinos can operate with arbitrary data distributions, including two-dimensional matrix distributions favored for reducing communication in graph analysis applications. In this work, we pass the edge and vertex lists to Grafiki with the same distribution as specified by the user in Chapel; thus, the distribution of nonzeros to processors is arbitrary and matches that of the Chapel edge list distribution.

Trilinos provides an efficient compressed sparse row matrix (`CrsMatrix`) data structure, but creation of a `CrsMatrix` from Chapel data would require creating a copy of the data in CRS format – an unacceptable requirement for this project. However, Trilinos also provides a `RowMatrix` abstraction that allows users to implement matrix operations on their data using their own underlying data structures. The `RowMatrix` abstraction supports any distribution of sparse matrix entries across processors. It does not require that matrix entries be sorted in any particular manner. Performance of a user's `RowMatrix` strongly depends on the user's implementation and data distribution. In our work, we sacrifice some computational performance by allowing our `RowMatrix` to use the Chapel edge lists directly, without any reordering or reorganization, thus avoiding an additional copy of Chapel users' data.

Unix: The Unix `mmap` function can be used to create memory-mapped regions that can be shared by independent Unix processes. With this function, two separate processes can read and write to the same block of memory by using the Unix `MAP_SHARED` flag. The `mmap` function works with the function `shm_open`, which takes

Figure 1: Example of a `Block` distribution across four locales of a Chapel distributed array with 64 elements; elements are distributed evenly across locales, with a contiguous chunk of indices assigned to each locale.

a backing file name (C string) for `mmap`. This backing file name connects `mmap` calls on separate processes to the same block of memory. Chapel has its own `mmap` function, `sys_mmap`, which takes the same arguments as the Unix function, invokes the system `mmap` function and returns an error code. Because we used the Unix `mmap` function in `sys/mman.h` in our early explorations, we continued using it with Chapel, specifying the `extern` keyword to refer to the Unix `mmap` function. However, we expect Chapel's `sys_mmap` would work identically. In our second approach below, we use `mmap` capability to share edge and vertex lists in mapped memory between separate Chapel and C++ processes or containers.

3 METHODOLOGY

In the general use case that we wish to support, graph analysts load graph data into Chapel edge lists. They then filter the graph data in some way to identify vertices and edges of interest. The resulting subgraph is shared with an MPI-based library for further analysis, and results are shared back to Chapel. Our goal is to accomplish this workflow without copying or reformatting data that is to be shared between Chapel and the MPI-based library.

Our demonstration follows the pattern of this general use case. We load a graph from files that contain coordinate pairs $(source_k, target_k)$ and, optionally, a weight w_k for all edges k in the graph (i.e., nonzeros in an adjacency matrix). The edges are stored in Chapel arrays *source* and *target* distributed across locales with Chapel's `Block` distribution. We filter the graph by identifying its largest connected component via a parallel label-propagation algorithm written in a few lines of Chapel code. We then share the edge lists, along with an array identifying vertices in the largest connected component, with the MPI-based Grafiki library. Grafiki computes vertex hitting times on the largest connected component.

Both of our integration approaches create a Trilinos `RowMatrix` (in C++) to describe the matrix that is passed to Grafiki. The `RowMatrix` class directly uses our shared, distributed edge lists to answer queries about the matrix and perform matrix operations. The most important method of `RowMatrix` is sparse-matrix vector multiplication (SpMV): $y = \alpha Ax + \beta y$ for matrix A, vectors x and y, and constants α and β. In parallel with distributed data, SpMV requires communication of off-processor vector entries x_j to be used in multiplication with local matrix entries a_{ij}, and of subproducts $a_{ij}x_j$ to be accumulated into vector entries y_i. This communication is done via MPI using the communication operations from Trilinos' `CrsMatrix` implementation. However, the `localApply` — the on-processor SpMV operation — differs from Trilinos' `CrsMatrix` since our `RowMatrix` uses coordinate-formatted edge lists from Chapel rather than Trilinos' compressed-sparse row format. Our

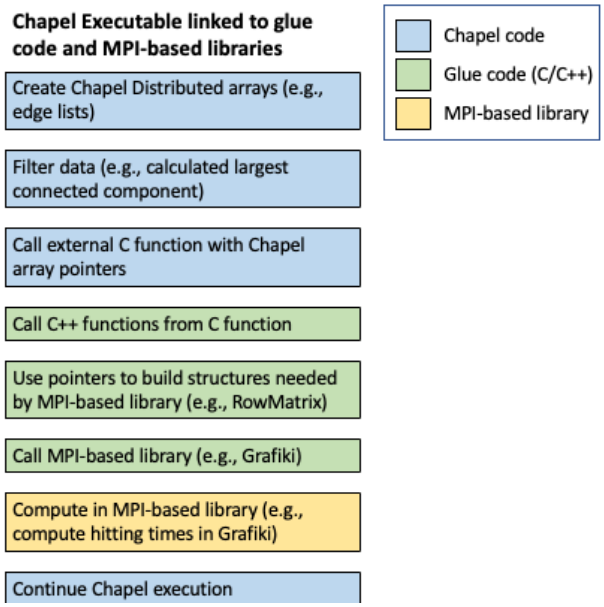

Figure 2: Method One: A single Chapel executable calls the MPI-based library; the library and "glue" code to convert from Chapel to C++ are linked into the Chapel executable.

`localApply` uses a straightforward loop over the edge lists, multiplying each edge value for edge $(source_k, target_k)$ with the appropriate, possibly communicated, x vector entry x_{target_k}. For our demonstration, we did not attempt to optimize this operation, but several optimizations (e.g., threading, GPU parallelism) are possible.

3.1 Method One: Direct Chapel-to-C calls

For our first method, we use a straightforward approach in which a single Chapel executable reads the graph data file, performs the connected-component label propagation, and then calls directly to a C function that then calls a C++ function that calls Grafiki. A high-level flow chart of its execution is in Figure 2.

This method takes advantage of Chapel's C interoperability to directly call C code. An overview of the code is shown in Figure 3. We start by having Chapel call a short C function `glueC` wrapped in `extern "C"` controls to allow it to be compiled by a C++ compiler. The C function then calls a C++ function `glueChapelGrafiki` with the same arguments. Function `glueChapelGrafiki` instantiates a `RowMatrix` object using the Chapel edge lists, and calls Grafiki. The C and C++ code are in a file separate from the Chapel code. The file is compiled by a C++ compiler and its object file is linked with the Chapel object file at build time.

```
// In Chapel source code
require "glue.h";
extern proc glueC(nEdges, *src, *tgt, ...);

coforall loc in Locales {
  on loc {
    var mySrc = source.localSubdomain();
    var myTgt = target.localSubdomain();
    glueC(mySrc.size:size_t,
          c_ptrTo(source[mySrc.low]),
          c_ptrTo(target[myTgt.low]), ...);
  }
}

// In glue.h file
extern "C" {
  int glueC(nEdge, *src, *tgt, ...);
}

// In glue.cpp C++ file
int glueChapelGrafiki(nEdge, *src, *tgt, ...) {
  Build RowMatrix(nEdge, src, tgt, ...);
  return Grafiki_hittingTimesDriver(RowMatrix, ...);
}

extern "C" {
  int glueC(nEdge, *src, *tgt, ...) {
    return glueChapelGrafiki(nEdge, src, tgt, ...);
  }
}
```

Figure 3: Method One: Chapel's path to call C++ code directly.

Each Chapel locale must call the C function in parallel, using Chapel's `coforall` parallel-for loop and its built-in *Locales* array as in Figure 3. The `coforall` loop creates a parallel task *loc* for each locale, and the subsequent `on` *loc* directive ensures that each locale calls `glueC` independently. This parallelism is needed to prevent Grafiki from hanging in collective MPI communications; all locales (i.e., all ranks) must participate in the call to `glueC`.

The short C function passes pointers from Chapel to the MPI-based library. Each locale can provide, for example, the number of edges and vertices in the locale, and pointers to its local *source* and *target* arrays. The appropriate way to get the pointer to the local arrays is shown in Figure 3: Chapel's `c_ptrTo` function obtains the C pointer to the lowest-indexed value of the array in the locale.

This approach is a simple path for integrating MPI-based C++ libraries with Chapel applications, but it has several drawbacks. It is intrusive to Chapel algorithm development. In order for Chapel to directly call to Grafiki, Chapel users must understand how to call C functions from Chapel. They must ensure that their Chapel installation, Grafiki library, and Trilinos library were built all with compatible compilers and MPI libraries. They also need to determine how to link the Grafiki and Trilinos libraries with their Chapel executable. Our next approach reduces this burden for Chapel users.

3.2 Method Two: Separate Chapel and C++ Processes

In our second method, a Chapel process interacts with an independent, concurrently running, C++ process, as outlined in Figure 4. The C++ process waits for a semaphore to be posted by Chapel before starting computation. The Chapel program reads edges into arrays that use a new `shareBlock` Domain that allows them to be shared with the C++ program through a `mmap` memory region. It filters the data (in our case, labeling connected components). It then writes metadata about the edge lists to a file and sets the semaphore indicating the C++ program can begin. The C++ program reads the metadata file to get information on accessing the shared data. It then creates a `RowMatrix` with the shared data, and calls Grafiki as in our first method. Upon completion, it resets the semaphore, indicating to the Chapel program that the Grafiki result is ready and Chapel can proceed. Details of each step follow.

To enable the Chapel data to be shared with the C++ program, we created a modified version of Chapel's `Block` Domain (from Chapel v1.23) in which the local arrays are built using `mmap` memory. We refer to this modified Domain as a `shareBlock` Domain.

Chapel code for our new `shareBlock` Domain is shown in Figure 5. The Chapel `Block` Domain stores the local arrays in a variable named `myElems`; this variable is assigned with a call to Chapel's `domain.buildArray()`. In our `shareBlock` Domain, we replace `domain.buildArray()` with a function `createMmap()` (shown in red) that calls the Unix `mmap` function to allocate memory of the requested size (the element's data type size times the domain size).

Each locale's `mmap` local array uses a separate backing file. The backing file name is generated by a new function `getBackingFile()`; it is a string consisting of the locale's ID and a counter to indicate which `shareBlock` is stored there. For example, the first `shareBlock` array allocated across four locales uses backing files `/share.bak0-1`, `/share.bak1-1`, `/share.bak2-1`, and `/share.bak3-1`. After all of the locales' local arrays are allocated with `mmap`, we increment a counter *Status* that is used to differentiate between separate `shareBlock` arrays' backing files.

To use the `mmap` region for the local array, we pass its pointer to modified versions of Chapel's `makeArrayFromPtr` and `makeArrayFromExternArray` (see Figure 6). Function `makeArrayFromPtr` is a short function that calls `makeArrayFromExternArray`. Function `makeArrayFromExternArray` creates a Domain for the new array and returns a call to Chapel's `_newArray()`, returning a new Chapel local array. We modified `makeArrayFromPtr` and `makeArrayFromExternArray` to accept a Domain as an argument. The Domain was needed because the original functions created a new Domain from 0 to the size of the array minus one, while the `Block` Domain expected indices matching the global indices of the array. Without the Domain argument, Chapel would assign a copy of the `mmap` region to the `myElems` array, rather than use the `mmap` region directly. Our modified `shareMakeArrayFromExternArray` uses the same Domain as `myElems`, so we avoid making a copy, and `myElems` refers directly to the `mmap` memory.

For the C++ program to access the `mmap` regions, it needs to know some metadata about the arrays using `shareBlock` Domains. Specifically, it needs the names of the backing files associated with

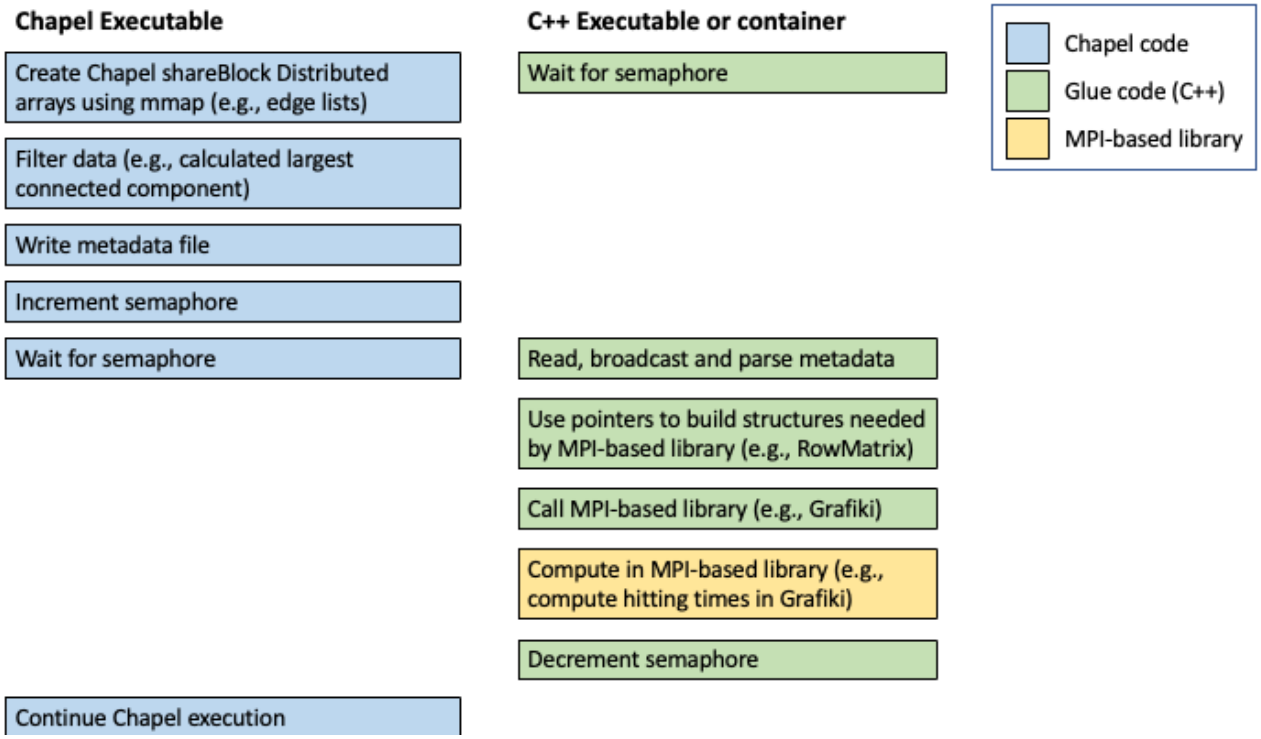

Figure 4: Method Two: Separate, concurrently running Chapel and C++ executables share graph data in `mmap` memory; execution is coordinated by a semaphore and a small metadata file.

each locale's local arrays and the size of those arrays. The Chapel program writes these fields to a metadata file, as in Figure 7. (While we used a regular file for the metadata, one could easily use a `mmap` memory space to share the metadata instead. But because the file is very small, using a regular file is feasible and straightforward.) This example file shows a matrix with 25 vertices and 105 edges distributed across four locales, with backing arrays for the *source* and *target* edge lists and the *component* labels. Locale 0 has 27 edges and 7 vertices, while other locales each have 26 edges and 6 vertices.

While the Chapel program runs, the C++ executable waits for a semaphore to indicate that the data is ready for use. After the Chapel program posts the semaphore to flag the C++ program to begin, the processor with rank 0 in the C++ program reads the entire metadata file into a buffer and broadcasts it to the other processors. Each processor then parses the buffer to extract the information (backing file names and data sizes) associated with its rank. The processors then set up pointers to their `mmap` data and call the MPI-based libraries with the data. In our demonstration, the processors creates the *source*, *target*, and *component* arrays using the backing files, collectively create a distributed `RowMatrix` wrapping the data, and call Grafiki's hitting times algorithm.

The key advantage of this method is that it is less intrusive to Chapel developers. Developers can declare distributed arrays using `shareBlock`, and the `shareBlock` class handles creating the `mmap` arrays with the required size. In addition, this method allows a C++ process to be built and run separately from the Chapel program, eliminating the need for Chapel programmers to link their code with MPI-based libraries. The disadvantage, for now, is that this method is restricted to the block distribution; extensions to other Chapel distributions are feasible but not yet done. Also, currently, we cannot reshape a `shareBlock` Domain after it is initialized.

3.3 Containerization of the C++ Library

To make Grafiki even easier to use by Chapel programmers, we have containerized our C++ glue program with Grafiki, and shown that a Chapel program and the Grafiki container can share memory through the same `mmap` array mechanism. This containerization allows Grafiki developers to provide tools that are fully encapsulated, sparing Chapel users from have to compile the Grafiki executable. Our Docker container encapsulates our C++ glue program, Grafiki and all of the libraries on which Grafiki depends (Trilinos, Kokkos, MPI, BLAS). We use Singularity [9], a container system designed for high-performance parallel computing, to instantiate our container with MPI parallelism. Our workflow then proceeds as in Method Two (Figure 4), with the user's Chapel program and and the Grafiki C++ container accessing the same `mmap` regions.

One challenge in integrating Chapel with containers was our use of semaphores. While `mmap` memory works across containers, POSIX semaphores do not. Semaphores are created on the parent process' stack. They are unusable in the container after namespace creation, which copies the semaphore rather than addressing it. Thus, the semaphore never unlocks in the C++ program. Our solution is to implement a "pseudo-semaphore" signaling mechanism. Since `mmap` works across containers, we use a single integer in `mmap`

```
class LocBlockArr {
proc init() {
    ...
    // Chapel's Block domain allocated local array as follows
    // this.myElems = this.locDom.myBlock.buildArray(...);
    // New shareBlock domain allocates local array with mmap memory
    if(this.locDom.myBlock.size) {

      var myPtr = createMmap(eltType, this.locDom.myBlock.size:uint);
      this.myElems = shareMakeArrayFromPtr(myPtr, this.locDom.myBlock.size:uint, this.locDom.myBlock);
      allLocalesBarrier.barrier();
      if(here.id == 0) then
        Status += 1;

    } else {
      this.myElems = this.locDom.myBlock.buildArray(...);
    }
}

//Chapel functions to create mmap array
proc createMmap(type eltType, size:uint):c_ptr {
    var myBytes = (size * c_sizeof(eltType)):uint;
    var fd = shm_open(getBackingFile(), O_RDWR | O_CREAT | O_EXCL, accessPerms):c_int;
    ftruncate(fd, myBytes:off_t);
    var region;
    region = mmap(nil, myBytes:size_t, PROT_READ | PROT_WRITE, MAP_SHARED, fd:fd_t, 0:off_t);
    return region:c_ptr(eltType);
}

proc getBackingFile():c_string {
    return ("/share.bak" + here.id:string + "\-" + Status:string).c_str();
}
```

Figure 5: Method Two: The new `shareBlock` Domain allocates `mmap` memory (shown in red) in place of the memory allocated in Chapel's `Block` Domain (Chapel v1.23). If the size of the domain is nonzero, an `mmap` array is created to fit the size requirements of the Domain and datatype. Backing files are tracked by incrementing *Status* after all local arrays have been allocated.

memory to mimic semaphore functionality. The value of this flag indicates which process should be working and which should be idling. Currently, we use a spin lock for this functionality.

With containerization, the Chapel user is spared the chore of building Grafiki and everything on which it depends. This approach delivers the highest ease of use to graph analysts.

4 EXPERIMENTS AND RESULTS

We demonstrate our approaches with a simple Chapel graph analysis application that shares the graph with the C++ library Grafiki. Specifically, our Chapel program reads a matrix from a Matrix Market file, symmetrizing the matrix if necessary, and constructs lists of edges corresponding to the matrix nonzeros. It then identifies the largest connected component of the graph using a simple, commonly used, label propagation algorithm. For label propagation, each vertex's label is initialized to its vertex number. Then the propagation algorithm loops over edges, giving each vertex of an edge the lower-valued label of the edge's two vertices. Iteration over edges is done in parallel with respect to locales and continues until no vertex labels change. The Chapel code then counts the number of vertices with each label and identifies the label with the most vertices. The label of the largest component is passed, along with the edge lists and vertex component labels, to Grafiki, which computes hitting times within the largest component. While vertices that are not in the largest component remain in the edge lists passed to Grafiki, they are ignored in the `RowMatrix` sparse matrix-vector multiplication operation.

We tested our approaches on Sandia's Kahuna high-performance data analytics cluster. Kahuna has Dual Socket Intel E5-2683v3 2.00GHz CPUs with 28 cores and 256 GB of memory. Kahuna uses a shared-memory build of Chapel 1.23; that is, Kahuna's Chapel did not enable an MPI backend. Thus, all experiments on Kahuna could use only a single multi-core node. For Method One, in which Chapel calls Grafiki directly, only one MPI rank (and, thus, one locale) can be used. Method Two, in which separate processes are used for Grafiki and Chapel, allows more MPI parallelism, as the MPI parallelism is not tied to Chapel's shared-memory implementation; we can run Grafiki and Chapel with up to 28 ranks (locales).

```
//Chapel's makeArrayFromPtr
proc makeArrayFromPtr(
     value: c_ptr, num_elts: uint) {
    var data = chpl_make_external_array_ptr(
                    value, num_elts);
    return makeArrayFromExternArray(
                    data, value.eltType);
}
//Chapel's makeArrayFromExternArray
proc makeArrayFromExternArray(
     value: chpl_external_array, type eltType) {
    var dom = 0..number_elements\-1;
    var arr = new unmanaged
              DefaultRectangularArr(dom=dom, ...);
    dom.add_arr(arr, locking = false);
    return _newArray(arr);
}
```

```
//New shareMakeArrayFromPtr
proc shareMakeArrayFromPtr(
     value:c_ptr, num_elts:uint, dom:domain) {
    var data = chpl_make_external_array_ptr(
                    value, num_elts);
    return shareMakeArrayFromExternArray(
                    data, value.eltType, dom);
}
//New shareMakeArrayFromExternArray
proc shareMakeArrayFromExternArray(
     value:chpl_external_array, type eltType,
     dom:domain) {
    var arr = new unmanaged
                DefaultRectangularArr(dom=dom._value, ...);
    dom.add_arr(arr, locking=false);
    return _newArray(arr);
}
```

Figure 6: Method Two: Chapel's original (left) vs. our new (right) shared `makeArrayFromPtr`: in the shared version, a `shareBlock Domain` is passed as an argument to `shareMakeArrayFromExternArray` and `DefaultRectangularArr`.

```
source /share.bak0-1 /share.bak1-1 /share.bak2-1 /share.bak3-1
target /share.bak0-2 /share.bak1-2 /share.bak2-2 /share.bak3-2
component /share.bak0-3 /share.bak1-3 /share.bak2-3 /share.bak3-3
nEdge 27 26 26 26
nVtx 7 6 6 6
```

Figure 7: Method Two: Chapel shares matrix data with the separate C++ executable that calls Grafiki via a small metadata file containing per-locale backing file names and data sizes.

We also tested on Sandia's Mutrino Cray computing system with 100 Intel Haswell nodes with 96 GB of memory per node. Mutrino has a distributed memory build of Chapel 1.24 with an MPI backend. Thus, we could run both approaches with multiple ranks and locales. To evaluate the effect of inter-node communication and provide the greatest contrast to the shared-memory Kahuna experiments, we chose to assign one rank (locale) per node on Mutrino.

Method One uses only one executable; Chapel and Grafiki execute on the same cores of allocated nodes. Method Two launches two executables that run concurrently on the nodes (using SLURM's "`srun -overlap`" for each executable); cores may or may not be oversubscribed, depending on the number of ranks per node. For Method Two, we show results with a C++ executable for Grafiki; past experiences saw no performance degradation using containers.

We tested the performance of our methods with small and large graphs from the SuiteSparse matrix collection [4]. The small graph, *bcsstk29.mtx*, has 28 strongly connected components and a largest component with 13.8K vertices and 620K edges. The large graph, *GAP-kron.mtx*, has 78M strongly connected components (many of them singleton vertices), and a largest component with 63M vertices and 4.2B edges. The large test case uses over half of a Mutrino node's memory for the source and target arrays alone (> 67 GB), making the graph too large to be copied; on a single node, data sharing between Chapel and Grafiki is the only way to solve the problem.

In Table 1, we show the execution times for label propagation in our connected-component algorithm written in Chapel, and the linear solve time in Grafiki's hitting time algorithm using both integration methods and the small graph *bcsstk29.mtx*. Results are shown for both the Kahuna and Mutrino systems. Execution times for Methods One and Two are comparable, with no significant loss of performance caused by using mapped memory in Method Two.

On both platforms, we see that adding MPI ranks accelerates the linear solve in Grafiki, with reasonable scaling even for this small graph. Adding locales also accelerates each iteration of label propagation in the Chapel-based connected component algorithm. Adding locales can increase the number of iterations required for connected component labeling, as each locale operates on a subset of the edges, slowing propagation across the full graph. This effect was seen in this small graph, in which the ordering (sorted by target vertex) of the input graph was optimal for label propagation with one locale; increasing the number of locales from one to 16 increased the number of propagation iterations from two to nine.

We ran the same experiments with the large *GAP-Kron.mtx* graph; execution times are in Table 2. Again, we see that Grafiki's linear solve time scales reasonably well with the number of MPI ranks, as does Chapel's label propagation time with the number of locales. On Mutrino, we see a difference in the label propagation time for our two approaches, with per-iteration time significantly longer using Method Two. This time difference is not seen for the

Platform	Number of Locales	Chapel LabelPropagation Time per iteration (seconds) One	Two	Grafiki Hitting Times Time per iteration (seconds) One	Two
Kahuna	1	1.24	1.22	0.0285	0.0287
	2	NA	1.91	NA	0.0162
	4	NA	1.56	NA	0.0087
	8	NA	1.00	NA	0.0045
	16	NA	0.65	NA	0.0051
Mutrino	1	0.82	0.89	0.0215	0.0230
	2	0.65	0.77	0.0113	0.0120
	4	0.44	0.47	0.0058	0.0062
	8	0.22	0.29	0.0031	0.0033
	16	0.11	0.15	0.0018	0.0018

Table 1: Runtime of Chapel connected-component label propagation and Grafiki hitting times on Kahuna and Mutrino using Method One (Direct calls from Chapel to Grafiki, Section 3.1) and Method Two (Separate Chapel and Grafiki executables, Section 3.2) with small graph *bcsstk29.mtx*

Platform	Number of Locales	Chapel LabelPropagation Time per iteration (seconds) One	Two	Grafiki Hitting Times Time per iteration (seconds) One	Two
Kahuna	1	8626	7476	1187	1235
	4	NA	9103	NA	364
	16	NA	3749	NA	188
Mutrino	4	OOM	10333	OOM	371
	16	1274	3079	103	105
	64	320	824	30	26

Table 2: Runtime of Chapel connected-component label propagation and Grafiki hitting times on Kahuna and Mutrino using Method One (Direct calls from Chapel to Grafiki, Section 3.1) and Method Two (Separate Chapel and Grafiki executables, Section 3.2) with large graph *GAP-Kron.mtx*

Grafiki linear solve; Grafiki linear solve times for both approaches are nearly identical. Thus, the degredation in Chapel's label propagation must be due to our implementation of `shareBlock` rather than some inherent mapped-memory access issue on Mutrino. We suspect it arises due to differences between Chapel versions 1.23 and 1.24. Our `shareBlock` Domain was built from the `Block` Domain in Chapel 1.23; it may not exploit performance enhancements in Chapel version 1.24. More investigation and a transfer of our memory-mapped approach to newer versions of Chapel are needed.

5 CONCLUSIONS AND FUTURE WORK

We have described new methods for integrating Chapel algorithms with MPI-based numerical libraries, enabling Chapel users to take advantage of existing, optimized parallel algorithms while maintaining the simplicity of Chapel programming. Our approaches enabled integration of Chapel algorithms with MPI-based libraries *without requiring additional copies of the user data*, allowing Chapel users to solve problems that fill the computer's memory. We demonstrated our approaches with shared-memory and distributed-memory versions of Chapel, as well as with Docker/Singularity containers.

While our demonstration used a simple Chapel program for finding connected components of a graph and the MPI-based graph libraries Grafiki and Trilinos for computing hitting times within the largest component, our methodology extends beyond this single demonstration. A matrix abstraction in Trilinos (`RowMatrix`) allowed Trilinos to perform matrix operations using the Chapel-formatted edge lists directly (i.e., without copying and/or reformatting the Chapel data). With our methods, any MPI-based library could be used, provided it either has native data structures identical to Chapel's or has abstraction capabilities similar to Trilinos'. To share data from any Chapel program with MPI-based libraries, Chapel users simply need to substitute `mmap`-enabled data structures (e.g., `shareBlock`) for standard Chapel arrays in their programs.

This project raises many opportunities for future work.

Arkouda [11] is often used for large-scale graph analysis because of its elegant NumPy-like interface. Arkouda is built on Chapel, so using our methods to integrate Arkouda and MPI-based libraries would be a natural extension of this work.

An option in Chapel to use a user-defined allocator when creating arrays would simplify extension of this work to other Chapel data objects and versions. To enable Chapel to use `mmap` memory, we refactored several Chapel functions to use the `mmap` memory (see Figures 5 and 6). These functions were based on the `Block` Domain in a specific version of Chapel (version 1.23), and would need to be updated for new versions of Chapel and other Domain types. An option for a user-defined allocator would allow us to provide an allocator based on `mmap` that could be used in many Chapel objects.

Redistributing graphs' edge lists to increase locality and reduce off-processor data accesses would speed matrix-vector multiplication and other operations in both Chapel and Grafiki. Load balancing tools such as Zoltan [5] or ParMETIS [8] balance computational work while reducing off-processor data dependencies. Even simple sorting of edge lists has been shown to reduce cache misses and speed execution [13]. Our current work relies on the users' data layout; users would need to pre-process the data to increase locality.

This work allows Chapel users to leverage years of effort in MPI-based parallel computing for physics-based and graph-based applications by using libraries like Trilinos and Grafiki. Comparisons between Chapel and MPI in terms of performance and ease-of-use are beyond the scope of this work, but are important for evaluating the productivity versus performance trade-off in using Chapel. Also important is the sharing of hardware resources between Chapel and MPI processes in modern computer architectures. Our experiments showed that running separate processes (Method Two) on the same cores of a single node did not slow performance relative to running a single process (Method One). Further experiments using Trilinos' multithreaded implementation would help in understanding interaction between Chapel and multithreaded libraries. Moreover, given the extensive recent efforts to adapt MPI-based libraries to accelerator architectures, experiments using Chapel+MPI+GPU would be of great interest and potential benefit to the graph analysis community.

6 ACKNOWLEDGEMENTS

We thank several people for enabling this work. Jon Berry provided motivation and use cases for this project. Samuel Knight, Omar Aaziz and Connor Brown provided system support on Sandia's computers. Chapel developers Elliot Ronaghan, Brad Chamberlain, Michael Ferguson, and Lydia Duncan in the Chapel Discourse Group provided technical advice and got us over several hurdles. Danny Dunlavy and the conference reviewers provided valuable feedback on this paper.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

REFERENCES

[1] Bradford Chamberlain, Sung-Eun Choi, Steven Deitz, David Iten, and Vassily Litvinov. 2010. User-Defined Distributions and Layouts in Chapel: Philosophy and Framework. In *2nd USENIX Workshop on Hot Topics in Parallelism*. Cray Inc, 1–12. https://chapel-lang.org/publications/hotpar10-final.pdf

[2] Bradford Chamberlain, Steven Deitz, David Iten, and Sung-Eun Choi. 2011. Authoring User-Defined Domain Maps in Chapel. In *Chapel User's Group 2011*. Cray Inc, 1–6. https://chapel-lang.org/publications/cug11-final.pdf

[3] B. L. Chamberlain, E. Ronaghan, B. Albrecht, L. Duncan, M. Ferguson, B. Harshbarger, D. Iten, D. Keaton, V. Litvinov, and P. Sahabu. 2018. Chapel comes of age: Making scalable programming productive. In *Cray User Group 2018*. https://cug.org/proceedings/cug2018_proceedings/includes/files/pap130s2-file1.pdf

[4] Timothy A. Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Trans. Math. Software* 38, 1 (2011), 1:1 – 1:25. https://sparse.tamu.edu/

[5] K.D. Devine, E.G. Boman, R.T. Heaphy, R.H. Bisseling, and U.V. Catalyurek. 2006. Parallel Hypergraph Partitioning for Scientific Computing. In *Proc. of 20th International Parallel and Distributed Processing Symposium (IPDPS'06)*. IEEE.

[6] Michael A Heroux, Roscoe A Bartlett, Vicki E Howle, Robert J Hoekstra, Jonathan J Hu, Tamara G Kolda, Richard B Lehoucq, Kevin R Long, Roger P Pawlowski, Eric T Phipps, et al. 2005. An overview of the Trilinos project. *ACM Trans. Math. Software* 31, 3 (2005), 397–423. http://dx.doi.org/10.1145/1089014.1089021

[7] Hewlett Packard Enterprise Development LP. 2021. Chapel Documentation. https://chapel-lang.org/docs/

[8] G. Karypis and V. Kumar. 1997. *ParMETIS: Parallel Graph Partitioning and Sparse Matrix Ordering Library*. Technical Report 97-060. Dept. Computer Science, University of Minnesota. http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview

[9] Gregory M. Kurtzer, Vanessa Sochat, and Michael W. Bauer. 2017. Singularity: Scientific containers for mobility of compute. *PLOS ONE* 12, 5 (05 2017), 1–20.

[10] Trevor M. McCrary, Karen D. Devine, and Andrew J. Younge. 2022. *Integrating PGAS and MPI-Based Graph Analysis*. Technical Report SAND2022-0653R. Sandia National Laboratories. Computer Science Research Institute Summer Proceedings 2021, J.D. Smith and E. Galvan, eds., 235–249.

[11] 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. https://doi.org/10.1145/3329722.3330148

[12] Message Passing Interface Forum. 1994. *MPI: A Message-Passing Interface Standard*. Technical Report. University of Tennessee, USA.

[13] Eric T. Phipps and Tamara G. Kolda. 2019. Software for Sparse Tensor Decomposition on Emerging Computing Architectures. *SIAM Journal on Scientific Computing* 41, 3 (2019), C269–C290.

[14] The Trilinos Project Team. 2021. The Trilinos Project Website. https://trilinos.github.io/

[15] Michael Wolf, Daniel Dunlavy, Richard B Lehoucq, Jonathan W Berry, and Daniel Bourgeois. 2018. *TriData: High Performance Linear Algebra-Based Data Analytics*. Technical Report. Sandia National Laboratories.